LA MAITRISE DE LA HP-48

TOME 1 : PROGRAMMATION ET EXERCICES

Prof. J.-M. FERRARD

D3I DIFFUSION

LA MAITRISE DE LA HP-48

TOME 1 : PROGRAMMATION ET EXERCICES

2e EDITION

Jean-Michel FERRARD

Agrégé de Mathématiques
Professeur de Mathématiques Supérieures
au lycée Déodat de Séverac, Toulouse

D3I
BP 49 - Tél. 62 24 66 12
31528 RAMONVILLE ST-AGNE CEDEX

NOTICE

Les informations contenues dans cet ouvrage peuvent être changées sans préavis.

D3I et HEWLETT-PACKARD ne donnent aucune garantie en ce qui concerne l'usage qui pourrait être fait des formules et exemples qui sont décrits dans le présent livret.

Le livret et toutes les informations qu'il contient ne peuvent être copiés en tout ou partie sans notre consentement.

HEWLETT-PACKARD est une marque déposée
de HEWLETT-PACKARD CORPORATION

© D3I - 1990

TABLE DES MATIERES

INTRODUCTION..... 7

PRINCIPES DE BASE..... 11

ELEMENTS DE PROGRAMMATION..... 27

LA PILE OPERATIONNELLE..... 67
Exercices: 90

NOMBRES REELS OU COMPLEXES..... 95
Exercices: 121

LES LISTES..... 127
Exercices: 136

LES CHAINES DE CARACTERES..... 143
Exercices: 151

LES TABLEAUX..... 157
Exercices: 200

VARIABLES GLOBALES ET REPERTOIRES..... 207
Exercices: 222

EXPRESSIONS, CALCUL DIFFERENTIEL ET INTEGRAL..... 225
Exercices: 254

GRAPHISME..... 259
Exercices: 291

LA PILE OPERATIONNELLE, Corrigés..... 295

NOMBRES REELS OU COMPLEXES, Corrigés..... 329

LES LISTES, Corrigés..... 351

LES CHAINES DE CARACTERES, Corrigés..... 369

LES TABLEAUX, Corrigés..... 387

VARIABLES GLOBALES ET REPERTOIRES, Corrigés..... 423

EXPRESSIONS, CALCUL DIFFERENTIEL ET INTEGRAL, Corrigés..... 431

GRAPHISME, Corrigés..... 449

INDEX..... 463

TABLE DES MATIERES DU TOME 2

PREAMBULE..... 7

INTRODUCTION..... 9

ARITHMETIQUE..... 15

NOMBRES REELS ET COMPLEXES..... 27

POLYNOMES..... 41

FRACTIONS RATIONNELLES..... 69

CALCUL MATRICIEL..... 79

ANALYSE..... 117

DEVELOPPEMENTS LIMITES..... 135

GEOMETRIE..... 165

GEOMETRIE DIFFERENTIELLE..... 173

GRAPHISME..... 195

ENTIERS LONGS..... 205

PROBABILITES..... 215

STATISTIQUES SIMPLES..... 235

STATISTIQUES DOUBLES..... 259

BASES DE DONNEES..... 271

INDEX..... 279

INTRODUCTION

La collection "**La maîtrise de la HP48SX**", comporte deux tomes:
HP48 : Programmation et Exercices.
HP48 : Programmation et Applications.

Les rôles sont ainsi partagés:

* Le tome 1 vous permet de dominer la programmation de la HP48 :
Pour celà il est organisé en chapitres traitant chacun d'un problème particulier.
Chaque chapitre est suivi d'exercices, qui sont corrigés et commentés dans la deuxième moitié du livre.

* Le tome 2 est une collection de programmes couvrant les besoins (mathématiques en général) des utilisateurs de la HP48 .

Les deux tomes sont donc complémentaires, et vous permettent de tirer le meilleur parti de votre calculatrice HP.

Dans ce tome 1, j'ai voulu adopter une démarche progressive et pédagogique.

- Dans un premier chapitre, "**PRINCIPES DE BASE**", les notions fondamentales qui permettent de comprendre la "philosophie" de la HP48 sont présentées.
- Le chapitre suivant, "**ELEMENTS DE PROGRAMMATION**", regroupe ce qu'il est indispensable de bien connaître avant de vouloir programmer sérieusement la HP48 .

Les "travaux pratiques" viennent ensuite.
Chacun des chapitres suivants est consacré à l'étude d'un aspect particulier de la programmation de la HP48 .

* LA PILE OPERATIONNELLE.
* NOMBRES REELS OU COMPLEXES.
* LES LISTES.
* LES CHAINES DE CARACTERES.
* LES TABLEAUX.
* VARIABLES GLOBALES ET REPERTOIRES.
* EXPRESSIONS, CALCUL DIFFERENTIEL ET INTEGRAL.
* GRAPHISME.

Ces différents chapitres sont suivis par une liste d'exercices, de difficulté très variable, du plus facile au plus "coriace".

Ces exercices sont corrigés et abondamment commentés dans la 2-ème partie du livre.
De nombreux exercices reçoivent ainsi plusieurs solutions.
Chaque solution est présentée sous forme d'un listing commenté et les idées qui prévalent à cette solution sont clairement exposées.
Souvent des exemples accompagnent ces solutions, notamment sous la forme de diagrammes présentant l'état de la pile opérationnelle aux principales étapes du fonctionnement du programme.

Dans la première partie du livre, on aborde donc l'utilisation et la programmation de la HP48 à travers les principaux types d'objets qu'elle sait manipuler (Tableaux, Listes, etc...).
Là encore, j'ai particulièrement insisté sur les exemples.
La plupart des instructions à connaître sont ainsi clairement illustrées, ce qui permet un apprentissage plus rapide et plus efficace.

J'ai, parce que le sujet me semblait intéressant, ou que le manuel d'utilisation de la HP48 me semblait incomplet, développé certains points d'une façon particulièrement approfondie.
A d'autres occasions, vous découvrirez des astuces ou des points méconnus dans l'utilisation de telle ou telle instruction.

Néanmoins, il ne m'a pas semblé souhaitable, ni même possible (ne serait-ce que pour conserver à cet ouvrage des dimensions raisonnables), de traiter absolument tous les sujets.
J'ai dû faire un choix, dicté par des considérations parfois subjectives, et plus d'une fois le lecteur se trouve ramené à la consultation du manuel d'utilisation.

Mon but n'est pas ici de remplacer le manuel d'utilisation de la HP48 , mais de vous donner les moyens, en un seul volume, de bien prendre en main votre calculatrice, et en particulier de la programmer.
Une lecture croisée de ce livre et du manuel d'utilisation vous sera certainement très profitable, tant les points de vue adoptés sont complémentaires.

Vous trouverez dans ces pages, j'en suis persuadé, un volume d'information propre à assouvir votre passion pour la HP48 , et à vous donner envie d'aller encore plus loin.

Pour en terminer avec cette présentation, disons que ce livre contient 150 exercices, tous corrigés une ou plusieurs fois, le nombre de programmes commentés dépassant la barre des 200.

Ce livre n'est pas fait pour être lu "rapidement".
La quantité d'informations qu'il contient en fait un outil de base pour qui veut progresser rapidement dans l'art difficile de la programmation de la HP48 , et il vous faudra sans doute le consulter souvent.

Même les plus chevronnés devraient éprouver quelques difficultés à résoudre certains exercices !

Pour tirer le meilleur parti de cet ouvrage, je vous conseille vivement d'assimiler parfaitement le contenu des différents chapitres , avant de vous lancer dans la résolution des exercices.

Ne passez pas à la solution d'un exercice sans avoir suffisamment cherché, et puis trouvé ou "séché".

J'ai apporté le plus grand soin à la rédaction des solutions et des commentaires qui les accompagnent. Néanmoins, vous avez tout intérêt à éxécuter (en pas à pas) les programmes proposés, pour vous pénétrer de leur fonctionnement.

Ainsi vous ferez des progrès rapides, et vous serez à même de réaliser les applications les plus ambitieuses.

La perfection n'existant pas ici, je souhaite vivement que, sur un exercice donné, vous éprouviez un jour le plaisir de trouver une solution supérieure à la mienne. L'utilité que je vois à cet ouvrage trouverait ainsi sa meilleure illustration.

Toulouse, Septembre 1990.

Jean-Michel FERRARD.

PRINCIPES DE BASE

Ce chapitre rappelle les principes généraux de fonctionnement et d'utilisation de la HP48SX. Il ne s'agit que d'un tour d'horizon rapide des points les plus importants.

Certaines des notions brièvement abordées ici font l'objet de développements plus systématiques dans les chapitres suivants de ce livre. C'est le cas notamment en ce qui concerne les variables et les répertoires, ou pour ce qui est de la pile opérationnelle.

En tout état de cause, C'est dans le manuel d'utilisation de la HP48SX qu'on pourra trouver les détails qui sont ici volontairement omis.

La "Philosophie" de la HP48SX tient en quelques mots simples, et la maîtrise de cette calculatrice passe par une bonne compréhension des notions qu'ils expriment:

* Les objets.
* La pile.
* La notation polonaise inverse.
* La ligne de commande.
* L'évaluation (des objets).
* Les variables.
* Les procédures (programmes et expressions algébriques).
* Les répertoires.
* Les menus (personnels ou intégrés à la machine).

LES OBJETS:

Le terme est volontairement vague. Il désigne tout ce que la HP48SX peut traiter (nombres réels ou complexes, tableaux, listes, programmes, etc...).

Cette imprécision voulue est justifiée par le fait qu'à tous ces "objets" s'appliquent un grand nombre d'instructions communes, ce qui diminue d'autant le nombre de règles à connaître, et ce qui autorise une grande liberté d'action.

Les différents objets que la HP48SX peut traiter sont:

* LES NOMBRES REELS (Ex: **19842.245781**, ou **-7.5532E15**).

* LES NOMBRES COMPLEXES, qui apparaissent comme des couples de nombres réels (Ex: **(3,4)**, ou **(-1.65,3.33E-7)**).

* LES ENTIERS BINAIRES, qui sont des entiers positifs exprimés en binaire, en octal, en décimal, ou en héxadécimal.
  Ex: **#10010b** en base 2, **#36057o** en base 8, **#795d** en base 10, et **#A810Fh** en base 16.

* LES CHAINES DE CARACTERES (Ex: **"Hewlett-Packard"**).

PRINCIPES DE BASE.

* LES TABLEAUX, qui sont des groupements ordonnés en une ou deux dimensions (vecteurs/matrices) de nombres réels ou complexes.

  LES VECTEURS REELS (Ex: **[18 5.23 701]**).

  LES MATRICES REELLES (Ex: **[[3 8 4]**
  [0 5 7]]).

  LES VECTEURS COMPLEXES (Ex: **[(5,6) (-2,3) (0,1)]**).

  LES MATRICES COMPLEXES (Ex: **[[(-1,-2) (5,0)]**
  [(34,47) (-1,3)]
  [(121,0) (4,11)]]).

* LES LISTES, groupements ordonnés d'objets quelconques.
  Ex: **{ (5,1) 2.56 [[1 2][3 5]] "Bonjour" }**
  { { 1 2 } { } } (2 deux sous-listes, dont une vide).

* LES NOMS (Ex: **'X'**, **'Y'**, **'PGCD'**, **'PROG'**).
  Le plus souvent ils sont utilisés comme nom de variables (ils désignent un objet: le contenu de cette variable).
  Ils peuvent très bien ne référencer aucun objet (on parle alors de "variable formelle" : ceci est très utile pour résoudre des problèmes d'une manière symbolique).

* LES EXPRESSIONS ALGEBRIQUES:

  Par exemple **'SIN(X+Y*LN(Z))'** ou **'EXP(X)=2-X'** (Cette dernière étant appelée équation car elle contient l'opérateur =).
  Elles obéissent à la logique et à la tradition mathématique.

  Elles sont censées produire un résultat au niveau 1 de la pile, en n'utilisant que les arguments qu'elles contiennent (nombres, noms) et donc sans prendre d'objet sur la pile.
  C'est pourquoi beaucoup d'instructions de la HP48SX ne sont pas autorisées dans les expressions algébriques.

  On appelle fonctions les intructions autorisées (le chapitre consacré aux expressions algébriques en contient la liste).
  Parmi les fonctions, on trouve les opérateurs arithmétiques (**+**, **-**, *****, **/**, **^**) et les fonctions mathématiques usuelles (**SIN**, **EXP**, **LN**, **SQ**, **√**,etc...), mais il y en a d'autres...

  Certaines touches de la HP48SX correspondent à des "fonctions": C'est le cas par exemple de la touche +, ou de la touche gauche de la ligne de menu quand le menu **PRG HYP** est affiché (touche qui correspond alors à la fonction **SINH**).
  Dans la suite, ces touches sont appelées touches de fonction.

* LES PROGRAMMES:

  Par exemple: « **DUP2 + SQ / ^** » .
  Ils conduisent souvent à des constructions moins "lisibles" que les expressions algébriques, mais leur rôle est bien plus large.
  Les programmes (leur éxécution) ne sont pas uniquement destinés à produire un résultat au niveau 1 de la pile, mais ils peuvent y prendre un nombre quelconque de données et la modifier complètement. Ils peuvent également modifier le contenu de la mémoire et le fonctionnement même de la calculatrice (en agissant sur certains indicateurs).

* LES OBJETS GRAPHIQUES, qui représentent des images affichables à l'écran de la HP48SX.

* LES OBJETS PRÉFIXÉS (ou "marqués", ou "étiquetés"): On désigne ainsi tout objet précédé d'une "étiquette" facilitant son interprétation.
  Ex: **Somme: 2418.47** (Le nombre réel 2418.47 est préfixé par l'étiquette "Somme" de manière à ce que sur la pile, il soit interprété sans risque de confusion avec un autre nombre).

* LES OBJETS-UNITÉS : Ils désignent des grandeurs physiques et sont formés d'une partie scalaire et d'une spécification d'unité .
  Ex: **9.81_m/s^2** (la partie scalaire est 9.81, et cette grandeur est exprimée en mètres par secondes-au-carré).

* LES REPERTOIRES: Ils résultent de la division, par l'utilisateur, de l'espace mémoire disponible en "compartiments" distincts.

* LES OBJETS-SAUVEGARDES: On les utilise pour mettre "à l'abri" un objet quelconque, ou pour le dupliquer si on veut modifier l'original, de manière à conserver la version initiale de cet objet (la sauvegarde s'effectuant dans une partie de la mémoire utilisateur (**PORT0**), ou sur une carte d'extension mémoire (**PORT1** ou **PORT2**)).

* LES FONCTIONS ET COMMANDES INTEGRÉES à la HP48SX, qui sont en quelque sorte les "atomes" du langage de programmation.
  Ex: **SIN**, **+** ,... sont des fonctions intégrées car on peut les utiliser dans des expressions ou des programmes.
  DUP, **RCL**, ... sont des commandes intégrées car on peut les utiliser dans des programmes, mais pas dans des expressions algébriques.

* LES OBJETS-BIBLIOTHEQUES et LES NOMS XLIB: Ils sont les objets utilisés dans les cartes d'extension spécifiques qui sont développées par le constructeur.
  Ces objets ne peuvent pas être créés sur une HP48SX.

LA PILE OPÉRATIONNELLE:

Le concept de "pile" occupe une position centrale pour l'utilisateur de la HP48SX. Cette pile est une zone de la mémoire où sont déposés les objets en cours d'utilisation ou qui attendent leur tour.

Il s'agit bien d'une pile dans la mesure où les objets sont disposés "les uns sous les autres" suivant l'ordre dans lequel ils ont été introduits.

Un nouvel objet est ainsi placé au niveau 1 de la pile, les autres objets remontant alors tous d'un niveau.

De même l'objet situé au niveau 1 peut être retiré de la pile, les autres objets descendant tous d'un niveau.

La HP48SX permet d'effectuer toutes sortes de manipulations sur le contenu de la pile. On peut dupliquer, supprimer, ou y déplacer des objets. Le mode "Pile Interactive" facilite beaucoup ces opérations.

L'écran de la HP48SX fait apparaître, en général, les 4 premiers niveaux de la pile opérationnelle.
En activant le mode "Pile Interactive", l'utilisateur peut circuler dans la pile, visualiser et/ou manipuler les objets qu'il y a déposés.

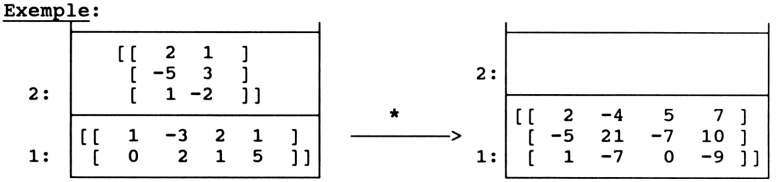
On peut pratiquement tout mettre sur la pile: des variables, des programmes, des nombres, des tableaux, des listes, etc....
Le principe est que tout est possible tant que l'utilisateur n'essaie pas de composer ensemble des objets qui ne peuvent l'être.
Mais le nombre d'opérations que l'on peut réaliser est étonnant.

L'exemple du calcul matriciel suffit à prouver les possibilités qu'offre ce concept de pile:

Considérons le problème suivant:

On veut effectuer le produit de deux matrices A et B.
Sur la plupart des calculatrices, il faut implanter soi-même un programme permettant d'effectuer ces produits.
Sur certaines machines plus récentes, ce programme est déjà implanté, mais son utilisation est longue et fastidieuse.
Sur la HP48SX il suffit d'entrer la première matrice sur la pile (la méthode est très simple), puis la seconde, avant d'appuyer sur la touche *.

Exemple:



On procède de même pour calculer l'inverse ou le déterminant d'une matrice carrée, ou pour résoudre un systême d'équations linéaires. Le gain de temps est extraordinaire.

Un chapitre entier étant consacré dans ce livre à la pile opérationnelle, il est sans doute inutile d'insister.

LA NOTATION POLONAISE INVERSE:

La notation "Polonaise Inverse", dite encore logique **"RPN"** diffère par exemple de la notation algébrique traditionnelle (la plus courante) par l'ordre dans lequel sont entrées les données lors d'un calcul numérique.

Pour calculer par exemple **'7*(5+3/11)'** il faut :
En notation algébrique traditionnelle: entrer l'expression telle qu'elle apparaît sur le papier, et valider (ENTREE, EXE,...).
En notation polonaise inverse: on entre les opérandes avant les opérateurs , ce qui peut donner ici:

3 11 / 5 + 7 *,
ou **5 3 11 / + 7 *,**
ou **7 5 3 11 / + *.**
ou **7 3 11 / 5 + *.**

On reproche à la logique **"RPN"** d'être éloignée de notre façon d'écrire les expressions algébriques. Elle a tout de même plusieurs avantages:

* Elle permet d'organiser à sa convenance les différentes étapes du calcul, et de visualiser les résultats intermédiaires.

  Par exemple, en reprenant l'exemple ci-dessus, la séquence: **3 11 / 5 + 7 *** (au clavier) permet de voir succes-sivement les valeurs de: **3/11, 5+3/11, 7*(5+3/11)**.

* Elle permet de se passer du module d'évaluation que nécessite l'utilisation de la notation algébrique (d'où des calculs souvent plus rapides).

La HP48SX vous permet d'utiliser, au choix et à tout moment, l'une ou l'autre de ces deux logiques.

Pour utiliser la notation algébrique, on crée des expressions algébriques, qui sont des objets délimités par 2 caractères ', et on les évalue, au moyen des instructions **EVAL** ou **→NUM**).

La création d'expressions algébriques est également facilitée par l'emploi de l'environnement **EQUATION-WRITER** (qui est à lui seul un véritable petit traitement de textes scientifiques) et qui permet d'entrer une expression mathématique telle qu'elle apparait sur le papier.

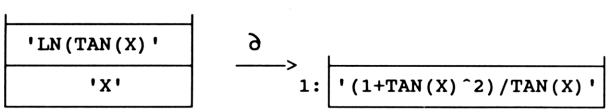
On ne perdra pas de vue cependant que la notation polonaise inverse est le langage naturel de la HP48SX, et qu'elle est optimisée pour celà.

Dans certains cas, cependant, la HP48SX s'avère un outil puissant de traitement des expressions algébriques, dans le domaine de l'algèbre symbolique (manipulation d'expressions contenant des variables formelles).

Citons entre autres la possibilité:

* De dériver une expression algébrique par rapport à une variable quelconque.

  Exemple:



* De résoudre une équation par rapport à une variable quelconque, à partir d'une estimation donnée.

* D'intégrer une expression sur un intervalle donné, par rapport à une variable donnée (intégration symbolique si l'expression initiale le permet), et avec une précision donnée (intégration numérique).

* De développer une expression, ou au contraire de la réduire par des factorisations.

etc...

▌ LA LIGNE DE COMMANDE:

On examine ici l'entrée d'informations au clavier, en ligne de commande.

L'application **EQUATION-WRITER** (entrer une expression algébrique ou un objet-unité tels qu'ils apparaissent sur le papier) n'est donc pas concernée ici (sauf par la possibilité qu'elle offre d'éditer en ligne de commande une partie ou la totalité de l'expression en cours de saisie).

L'application **MATRIX-WRITER** (entrer un tableau de nombres réels ou complexes) est concernée ici par le fait que l'entrée de chacun des coefficients du tableau (ou son édition pour modification) se fait par l'intermédiaire de la ligne de commande.

Quand l'utilisateur entre des données ou des instructions au clavier, le texte entré apparaît en ligne de commande.

Le contenu de la ligne de commande peut être annulé en cours de saisie, par un appui sur **ON** (le contenu de cette ligne de commande est alors perdu).

Le contenu de la ligne de commande est validé par la touche **ENTER** ou toute autre touche qui éxécute **ENTER** (leur nombre dépend du mode de saisie).

LES DIFFERENTS MODES DE SAISIE:

On peut utiliser plusieurs modes de saisie sur la HP48SX.

Dans chacun de ces modes, la saisie peut se faire en mode "Insertion" (chaque nouveau caractère vient s'insérer dans le texte déjà entré, en déplaçant vers la droite les caractères éventuellement situés à droite du curseur) ou en mode "Recouvrement" (chaque nouveau caractère recouvre celui qui se trouve éventuellement à la position désignée par le curseur).

Par défaut, néanmoins, on est en mode insertion.

On passe du mode insertion au mode recouvrement (et vice-versa) par un appui sur la touche correspondant au label **INS** dans le menu **EDIT** (pour celà, il faut avoir entré au moins un caractère en ligne de commande).

Un curseur clignotant indique la position où va être placé le prochain caractère entré.
Sa forme est ◀ si on est en mode insertion et ▌ si on est en mode recouvrement.

* LE MODE "SAISIE IMMEDIATE":

C'est le mode par défaut. Le fait d'appuyer sur une touche de menu (menu **VAR**, menus personnalisés ou menus intégrés à la HP48SX) éxécute l'action associée à cette touche de menu.
De même sont éxécutées les actions associées aux touches de commandes ou de fonction. Ces touches éxécutent donc elles-mêmes la commande **ENTER**.

* LE MODE "ALGEBRIQUE":

C'est le mode utilisé pour entrer des expressions algébriques.
On entre dans ce mode en tapant le caractère ', qui débute toute expression algébrique.
Le mode de saisie algébrique est reconnaissable à l'affichage de l'indicateur **ALG** en haut de l'écran.
Les touches de commandes sont éxécutées (**RCL**, **STO**, **EVAL**, **→NUM**, **CLEAR**, **SWAP**, etc...) et provoquent un retour au mode de saisie immédiate.
Les touches de fonctions ajoutent le nom correspondant en ligne de commande (avec au besoin une double parenthèse).

* LE MODE "PROGRAMME":

Dans ce mode, l'appui sur une touche quelconque écrit le mot associé en ligne de commande. Seules les commandes non programmables sont éxécutées (**ON** annule la saisie, **ENTER** la valide, on peut afficher n'importe quel menu ...).
Le mode de saisie "programme" est reconnaissable à l'affichage de l'indicateur **PRG** en haut de l'écran.
On passe automatiquement en mode "programme" quand on entre le caractère **{ }** (qui débute une liste) ou le caractère « » (qui débute un programme).

* LE MODE "PROGRAMME / ALGEBRIQUE":

Ce mode de saisie est automatiquement activé quand, en mode "programme", c'est-à-dire pendant la saisie d'une liste ou d'un programme, on appuie sur la touche ' (pour débuter la saisie d'un nom ou d'une expression).
Le mode de saisie "programme / algébrique" est reconnaissable à l'affichage des indicateurs **ALG PRG** en haut de l'écran.
Comme dans le mode "algébrique", les touches de fonctions ajoutent le nom correspondant en ligne de commande (avec au besoin une double parenthèse).
L'appui sur une touche de commande programmable (ex **SWAP**, **STO** ou **DUP**) écrit le nom de cette commande (comme dans le mode "programme") et provoque un retour au mode programme.

Remarque:
On peut "circuler" entre ces modes de saisie par des appuis successifs sur la touche **ENTRY**.
ENTRY passe du mode "saisie immédiate" au mode "programme" et alterne les modes "programme" et "programme / algébrique".

* LE MODE "ALPHA":

Il est activé, pour une touche, par un appui sur la touche α.
Il est verrouillé par deux appuis successifs sur α.
En mode "alpha", l'appui sur une touche produit l'affichage en ligne de commande du caractère alphanumérique correspondant à cette touche (caractères alphabétiques, chiffres, carac-tères correspondant aux opérations arithmétiques ...)
Si le mode "alpha" a été verrouillé par un double appui sur α, un nouvel appui sur α ramène à l'utilisation normale du clavier.
Rappelons enfin que la séquence de touches α <=┐ passe, pour une touche, du mode "majuscules" au mode "minuscules" (ou l'inverse), et que la séquence α <=┐ α définit le mode "minuscules" comme étant le mode "alpha" prioritaire (si c'était auparavant le mode "majuscules") ou vice-versa.

SAISIR PLUSIEURS OBJETS EN LIGNE DE COMMANDE:

Il est possible de saisir plusieurs objets sur la même ligne de commande. Ceux-ci doivent être isolés les uns des autres par des "séparateurs".

Les séparateurs sont:
* Les délimiteurs d'objets **( ) # " [ ] { } ' : « »** qui marquent le début ou la fin d'un objet.
* La virgule ou le point (suivant le choix fait dans le menu **MODE** (Entrée **FM,**). Par défaut, il s'agit de la virgule.
* L'espace, ou le retour à la ligne par <┘.

Certains objets sont dits structurés car ils sont composés d'un assemblage d'objets élémentaires.
Il en est ainsi des complexes, des tableaux, et des listes.
Entre les différents "sous-objets" composant cet objet structuré, on peut insérer des espaces, des virgules (si la virgule est le séparateur attitré), des retours à la ligne.
Celà permet (en séparant clairement, à la saisie, l'objet structuré entre ses différentes composantes) d'entrer des objets volumineux avec un maximum de confort.

L'EVALUATION DE LA LIGNE DE COMMANDE:

Le contenu de la ligne de commande est validé par un appui sur **ENTER** (quel que soit le mode de saisie), ou par l'appui sur une touche éxécutant **ENTER** (celà dépend du mode de saisie).

Le contenu de la ligne de commande est alors analysé. Si une erreur de syntaxe est trouvée dans un des objets qui forment cette ligne de commande, le message correspondant est affiché et on se retrouve avec la ligne de commande fautive,le curseur placé là où l'erreur a été constatée.

Si l'analyse de la ligne de commande n'a révélé aucune erreur, chacun des objets qui la constitue est, à son tour, placé au niveau 1 de la pile et évalué:

* Si cet objet est un nom sans cotes, la HP48 examine s'il s'agit du nom d'une instruction intégrée (et cette instruction est éxécutée), ou d'une variable de l'utilisateur (dans ce cas le contenu de la variable est placé sur la pile: s'il s'agit d'un programme ou d'un nom, il est évalué).
  Si ce nom sans cotes est inconnu de la HP48SX, il est placé entre deux cotes ' au niveau 1 de la pile.
* Si cet objet désigne une instruction arithmétique de la HP48SX (+,-,*,/,^,etc...) cette intruction est éxécutée.
* Dans les autres cas l'objet est placé au niveau 1 de la pile (le contenu des programmes ou des listes n'est pas évalué).

Bien sûr l'évaluation de la ligne de commande peut conduire à une erreur d'éxécution (bien qu'aucune erreur de syntaxe n'ait été détectée dans chacun des objets qui la constituent).
Dans un tel cas, et si on veut revenir à la situation précédant cette erreur, il faut:
Exécuter la commande **LAST STACK** pour récupérer l'état de la pile avant la validation de cette ligne de commande.
Exécuter la commande **LAST CMD** pour revenir à la ligne de commande qui était erronée.

Remarque:
Si on a validé une ligne de commande contenant plusieurs objets successifs, l'un d'entre eux peut correspondre à une instruction, qui est alors éxécutée.
Si cette instruction agit sur la pile, elle agit donc sur les objets qui précédaient dans la ligne de commande en cours d'évaluation.
Par exemple si on place **4 7 MAX** en ligne de commande et si on valide par **ENTER**:
4 est placé sur la pile, au niveau 1.
7 est placé au niveau 1 (4 remonte au niveau 2).
L'instruction **MAX** est éxécutée et donne 7.
Le résultat, en fin de compte, est donc: 7 au niveau 1.

L'EDITION EN LIGNE DE COMMANDE:

Remarque:
Dans l'environnement **EQUATION-WRITER**, la modification d'une partie ou de la totalité de l'expression traitée se fait en ligne de commande, (en utilisant l'entrée **EDIT** dans cet environnement), mais on peut également modifier l'expression (par exemple lui ajouter des termes) en restant à l'intérieur de l'application **EQUATION-WRITER**.

Quand on saisit une ligne de commande, quand on cherche à modifier un objet sur la pile (par **EDIT** s'il s'agit de l'objet placé au niveau 1, et par **n VISIT** si c'est l'objet situé au niveau n), ou enfin quand on cherche à modifier le contenu d'une variable (par **VISIT**, le nom de cette variable étant au niveau 1), on se trouve dans le même mini traitement de texte.

On peut également passer en édition de l'objet placé au niveau 1 en actionnant la touche ∇ (si cet objet est une matrice, on passe dans l'environnement **MATRIX-WRITER**;si c'est une expression ou un objet unité on passe dans l'environnement **EQUATION-WRITER**).

Comme on vient de le voir, l'appui sur une touche peut dépendre du mode de saisie.
Néanmoins, et dans tous les cas:

ENTER valide le texte contenu en ligne de commande.

ON annule le texte édité.

⇦ (Backspace) détruit le caractère à gauche du curseur et "tire" la suite du texte d'un caractère vers la gauche.

DEL détruit le terme situé à la position du curseur (le reste du texte est "tiré" d'un caractère vers la gauche).

◁ déplace le curseur d'un caractère vers la gauche (et remonte éventuellement à la ligne précédente).

▷ déplace le curseur d'un caractère vers la droite (et descend éventuellement à la ligne suivante).

∇ déplace le curseur sur la ligne suivante.

Δ déplace le curseur sur la ligne précédente (si l'objet édité ne comporte qu'une ligne, on active la "pile interactive").

PRINCIPES DE BASE.

La touche <⏎ crée un "retour-chariot" (passage à la ligne inséré à la position du curseur, qu'on soit en mode "insertion" ou en mode "recouvrement").

Pour détruire un "retour-chariot", deux possibilités:
- Passez au début de la ligne suivant ce retour-chariot et actionnez la touche ⌫ (BackSpace).
- Passez à la fin de la ligne précédant ce retour-chariot et actionner la touche **DEL**.

Les touches de déplacement, précédées par un appui sur ╔=>, prennent un sens un peu plus fort:

╔=> ◁ déplace le curseur au début de la ligne actuelle.
╔=> ▷ déplace le curseur à la fin de la ligne actuelle.
╔=> △ déplace le curseur sur la première ligne du texte.
╔=> ▽ déplace le curseur sur la dernnière ligne du texte.

Quand on édite un objet en ligne de commande, et si par exemple on veut modifier un objet déjà "ancien", on peut rendre actif tel ou tel menu (ce qui donne alors un sens particulier aux six touches supérieures du clavier).

En particulier, il est intéressant de se placer dans le menu **EDIT**, qui contient un certain nombre de facilités d'édition (D'ailleurs le menu **EDIT** est automatiquement sélectionné les commandes **EDIT** ou **VISIT**).

Le menu **EDIT** contient les entrées suivantes:

←SKIP	SKIP→	←DEL	DEL→	INS▪	↑STK

←SKIP: passage au début du mot précédent (éventuellement sur la ligne précédente).

SKIP→: passage au début du mot suivant (éventuellement sur la ligne suivante).

←DEL: effacement du mot situé à gauche de la position du curseur.

DEL→: effacement du mot situé à droite de la position du curseur.

INS : passe du mode "Insertion" au mode "Recouvrement" (le petit carré ▪ suivant le label **INS** montre que le mode "Insertion" est actif, ce qui se voit aussi à la forme du curseur).

↑STK: place à la position du curseur l'objet situé au niveau 1 de la pile (et qui du même coup est retiré de la pile).

Là encore, le fait d'utiliser la touche ╔=> augmente la portée des commandes **DEL→** et **←DEL**.

╔=> **←DEL**: détruit tous les caractères situés sur la même ligne que le curseur, et à gauche de celui-ci. Le reste de la ligne est "tiré" vers le bord gauche de l'écran.

╔=> **DEL→**: détruit tous les caractères situés sur la même ligne que le curseur, à partir de la position de celui-ci, et juqu'à la fin de cette ligne.

Un exemple instructif:

L'édition d'un objet en ligne de commande est un outil puissant.
Supposons par exemple que nous ayons entré les trois vecteurs ci-contre aux niveaux 3, 2, 1 et que nous voulions les "superposer" pour en faire une matrice à trois lignes.

```
3:  [ 3 7 8 4 ]
2:  [ 2 1 9 5 ]
1:  [ 7 0 8 3 ]
```

On commence par éxécuter **3 →LIST** (l'instruction **→LIST** se trouve dans le menu **PRG OBJ**).
Les 3 vecteurs sont alors les éléments de la liste, au niveau 1:
{ [3 7 8 4] [2 1 9 5] [7 0 8 3] }.

On fait alors **EDIT**, pour passer en édition de cet objet:

```
{
[ 3 7 8 4 ]
[ 2 1 9 5 ]
[ 7 0 8 3 ]
```

Le curseur est au départ sur le caractère { de début de liste. Il suffit de le remplacer par [et de remplacer le dernier } par un] (pour aller à la fin utiliser les touches de déplacements).

Le fait d'appuyer sur la touche **[]** provoque l'affichage des deux caractères [et]. A chaque fois, on élimine celui des deux qui est indésirable, avec les touches **DEL** ou ⌫ (BackSpace) ou les entrées **DEL→** ou **←DEL** du menu **EDIT**).

Il suffit alors de valider par ENTER, pour obtenir la matrice voulue au niveau 1.

```
1:  [[ 3 7 8 4 ]
     [ 2 1 9 5 ]
     [ 7 0 8 3 ]]
```

L'EVALUATION DES OBJETS:

Evaluer un objet, c'est éxécuter la ou les actions éventuellement associées à cet objet.
Il y a plusieurs façons d'évaluer un objet. L'objet évalué est celui qui se trouve au niveau 1 de la pile. Cette évaluation peut avoir des conséquences sur la pile (notamment si l'objet évalué est un programme...)
Un moyen (parmi d'autres) d'évaluer un objet est de le placer au niveau 1 de la pile, et d'appuyer sur la touche **EVAL**.

Parmi les objets que connaît la HP48SX, plusieurs cas doivent être envisagés, selon les réactions à l'instruction EVAL.

* LES DONNEES:

Ce sont des objets "inertes", qui ne sont associés à aucune action particulière. Les évaluer n'a donc ici aucun effet.

Les données sont:
* Les nombres réels ou complexes.
* Les tableaux (vecteurs ou matrices, réels ou complexes).
* Les chaînes de caractères.
* Les entiers binaires.
* Les objets graphiques.
* Les objets-unité.

PRINCIPES DE BASE.

* LES NOMS:

Un nom est un objet délimité par 2 caractères ', qui commence par une lettre, et qui ne contient pas l'un des caractères suivants: #,[,],",',{,},(,),«,»,: et _
+,-,*,/,^,√,=,<,>,≤,≥,≠,∫ ,∂ , et !
le séparateur en cours (le point ou la virgule).

Un nom peut être associé à un objet: le couple (nom, objet) est appelé une variable: Le nom est le nom de la variable, l'objet est le contenu de la variable.

Quand on évalue un tel nom, le contenu de la variable est placé sur la pile, et évalué s'il s'agit d'un nom ou d'un programme (le programme est éxécuté).

Un nom peut n'être associé à aucun objet (on parle dans ce cas de "variable formelle"). Ce nom est alors considéré comme une donnée, et l'évaluer n'a aucun effet.

* LES NOMS SANS COTES:

Un nom peut apparaître sans ses cotes ' dans les cas suivants:
* Sur la ligne de commande.
* Dans un programme ou une expression.

Ce nom peut être celui d'une variable.
Dans ce cas, évaluer le nom siginifie placer le contenu de ce nom sur la pile, et l'évaluer si ce contenu est un nom ou un programme (le programme est éxécuté)

Si ce nom n'est pas le nom d'une variable, son évaluation dans un programme ou lors de la validation de la ligne de commande place le même nom, mais avec les cotes, au niveau 1 de la pile (dans une expression, ce nom reste inchangé).

* LES PROGRAMMES:

Evaluer un programme, c'est le "lancer", l'"éxécuter".
Pour simplifier , on peut dire qu'un programme est une succession d'objets.
Evaluer ce programme, c'est provoquer l'évaluation de ces différents objets les uns après les autres.

* LES EXPRESSIONS ALGEBRIQUES:

Evaluer une expression algébrique c'est "calculer sa valeur".
Une expression peut contenir des noms sans cotes.
L'évaluation de l'expression provoque alors l'évaluation de ces noms.

* LES LISTES:

La commande **EVAL**, appliquée à une liste, a pour effet de placer sur la pile les différents objets de la pile (le premier objet de la liste est entré le premier sur la pile, et se retrouve donc au plus haut quand tous ont été entrés).
Parmi les objets de cette liste, les programmes et les noms sont évalués.

LES VARIABLES:

Une variable est la combinaison d'un nom (le nom de la variable) et d'un objet (le contenu de la variable).

Le nom de la variable est alors utilisé pour désigner (référencer) l'objet.

La HP48SX accepte deux types de variables:
Les variables globales et les variables locales. Ces dernières sont décrites dans le chapitre "Eléments de programmation".

Nous ne considérerons donc ici que des variables globales.

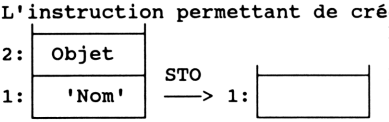
Les noms de variables sont soumis à quelques règles syntaxiques:
* Débuter par une lettre (lettres latines, grecques, et certains symboles comme ← → ↑ ↓ °)
* Contenir au plus 127 caractères (lettres ou chiffres).

Les noms de variables sont destinés à apparaître dans un menu, qui ne permet de visualiser que les 4 ou 5 premières lettres du nom. On évitera donc les noms inutilement longs.

La HP48SX fait la distinction entre majuscules et minuscules dans les noms de variable, mais les lettres sont converties en majuscules dans l'affichage des menus.

CREATION D'UNE VARIABLE:

L'instruction permettant de créer une variable est **STO**.



Le schéma ci-contre représente la création d'une variable. Le nom de cette variable est placé au niveau 1. Le contenu de la variable est placé au niveau 2.

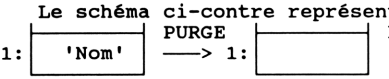
Le nom de la variable ainsi créée apparaît alors comme la première entrée du menu **VAR** correspondant au répertoire où on se trouve.

Cas particulier:
Il est fort possible que le nom placé au niveau 1 soit déjà le nom d'une variable appartenant au répertoire courant.
Le contenu précédent de cette variable est alors remplacé au profit de l'objet placé au niveau 2 (et le nom de cette variable garde la même position dans le menu **VAR**).

SUPPRESSION D'UNE VARIABLE:

L'instruction permettant de supprimer une variable est **PURGE**.

Le schéma ci-contre représente la suppression d'une variable.



Le nom de la variable disparaît du menu **VAR**.

On peut rattraper une utilisation involontaire de **PURGE** en éxécutant immédiatement la commande **LAST ARG** (on trouve alors le nom de la variable au niveau 1, et son contenu au niveau 2. Il reste à faire **STO** pour revenir à la situation de départ).

MODIFICATION DU CONTENU D'UNE VARIABLE:

Pour modifier le contenu d'une variable (hormis la modification obtenue par **STO**) on dispose de plusieurs solutions:

* On éxécute **VISIT**, après avoir placé le nom de cette variable au niveau 1.
* On actionne la séquence ╔═> ∇ , après avoir placé le nom de la variable au niveau 1. La différence avec **VISIT** est que le contenu de la variable apparaît:
  -Dans l'environnement **MATRIX-WRITER** si c'est un tableau.
  -Dans l'environnement **EQUATION-WRITER** s'il s'agit d'une expression ou un objet-unité.
* En se plaçant dans la "Pile Interactive" (presser la touche △) en sélectionnant le niveau où est le nom de la variable, et en éxécutant la séquence ╔═> **VIEW** (**VIEW** est une des entrées du menu spécifique à la pile interactive).

Le contenu de cette variable apparaît alors dans l'éditeur de la HP48SX (en ligne de commande, ou dans un des 2 environnements cités plus haut).

On effectue les modifications souhaitées et on valide par **ENTER** (ou on annule par **ON**).
En cas de validation par **ENTER**, le contenu modifié devient le nouveau contenu de la variable.

LES REPERTOIRES ET LES MENUS:

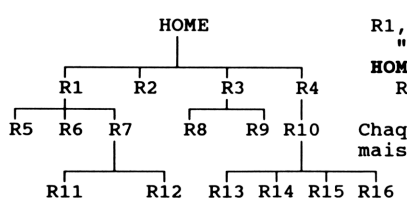
RÉPERTOIRES:

L'importante mémoire disponible sur la HP48SX permet de stocker un grand nombre de variables contenant des programmes, des données..

Il faut pouvoir accéder rapidement à chacune de ces variables.
Pour celà l'utilisateur peut compartimenter la mémoire de sa HP48SX en un ensemble de répertoires.
Ces répertoires sont reliés entre eux d'une façon hiérarchisée.
Cette structure est souvent comparée à un arbre généalogique.

A l'origine (lors de sa première utilisation, ou après toute ré-initialisation), la HP48SX ne reconnaît qu'un seul répertoire, dont le nom est **HOME**.
Au fur et à mesure de ses besoins, l'utilisateur peut créer des sous-répertoires du répertoire **HOME**, puis des sous-répertoires des répertoires ainsi créés, etc...

Le résultat est une arborescence de répertoires telle que:



R1, R2, R3, R4 sont les répertoires "fils" du répertoire **HOME**.
HOME est le répertoire "père" de R1, R2, R3, R4.

Chaque répertoire a un "père" unique mais peut avoir un nombre quelconque de répertoires "fils".

L'intérêt de l'utilisateur est évidemment de grouper dans un même répertoire des programmes ou des données ayant entre eux des liens étroits.

On peut circuler de façon simple dans cette arborescence de répertoires: Le nom de chaque répertoire "fils" apparaît dans le menu **VAR** du répertoire "père".
L'appui sur la touche de menu correspondante provoque alors le passage dans le répertoire "fils" choisi.

Inversement, l'instruction **HOME** (faire **HOME ENTER** ou actionner la touche correspondante au clavier) provoque le passage d'un répertoire quelconque au répertoire **HOME**.
Enfin, la touche **UP** du clavier passe d'un "répertoire-fils" à son son "répertoire-père".

MENUS:

Quand on est dans un répertoire particulier, la touche **VAR** fait apparaître le menu des variables de ce répertoire (désigné par "**menu VAR**": il y a autant de "menus **VAR**" que de répertoires).

De même, où qu'on soit dans l'arborescence des répertoires, il est possible d'accéder à l'un quelconque des répertoires intégrés à la HP48SX (**PRINT, I/O, MODES, MTH, PRG**, etc...)

Dans chaque répertoire, il est possible de définir un ou plusieurs menus personnalisés, par utilisation de la variable **CST**.
L'appui sur la touche **CST** provoque alors l'affichage du menu personnalisé placé alors dans la variable **CST** (Custom).
(On se reportera, dans ce livre, au chapitre consacré aux variables globales et aux répertoires).

Le menu actuellement sélectionné (**VAR, CST,** ou menu intégré) apparaît sur la ligne inférieure de l'affichage.

Il est fort possible qu'un menu comporte plus de six entrées. On accède aux entrées suivantes au moyen des touches **NEXT** et **PREV**.

Quand un menu est affiché, les six touches supérieures du clavier (dites alors touches de menu) prennent une signification liée à l'entrée qu'elles désignent dans ce menu, et qui dépend du mode de saisie.

NB: Les menus personnalisés affichés en utilisant la touche **CST** se comportent comme les menus **VAR**.

En mode "Programme": (Indicateur **PRG** affiché)
Le nom (sans cotes) de la variable ou de l'instruction est ajouté tel quel en ligne de commande.

En mode "Algébrique": (Indicateur **ALG** affiché)
S'il s'agit d'un menu **VAR**, le nom de la variable est ajouté, sans cotes, en ligne de commande.
S'il s'agit d'un menu intégré, le nom est ajouté en ligne de commande (avec éventuellement une double parenthèse), à condition que l'instruction soit une fonction, c'est-à-dire soit autorisée dans les expressions algébriques, sans quoi se produit une erreur de syntaxe).

En mode "Programme / Algébrique" : (Indicateurs **ALG PRG**)
S'il s'agit d'un menu **VAR**, le nom de la variable est ajouté, sans cotes, en ligne de commande.
S'il s'agit d'un menu intégré, le nom est ajouté en ligne de commande (avec éventuellement une parenthèse ouvrante). Si ce nom ne correspond pas à une fonction, c'est-à-dire est une commande non autorisée dans les expressions algébriques on revient au mode "Programme".

En mode "Exécution":
S'il s'agit d'un menu **VAR**:
Le contenu de la variable est placé sur la pile:
Si ce contenu est un nom il est évalué.
Si ce contenu est un programme, il est éxécuté.
S'il s'agit d'un menu intégré, l'instruction correspondante est éxécutée.

En mode "Alpha": (Indicateur α)
L'appui sur une touche de menu affiche un des caractères:
A,B,C,D,E,F (si on est en clavier "majuscules").
a,b,c,d,e,f (si on est en clavier "minuscules").
α,ß,∆,δ,ε,θ (clavier "alpha-shifté-droit").

Exceptions:

Si, dans le menu actuellement affiché, le nom est celui d'un sous-menu d'un menu intégré à la HP48SX (une telle entrée est reconnaissable à sa barre horizontale), l'appui sur la touche correspondante a pour effet d'afficher ce sous-menu (et ceci quelque soit le mode en cours, sauf si bien sûr le clavier "alpha" est activé).

Il arrive que certaines entrées de menus intégrés à la HP48SX correspondent à des opérations non programmables.
Exemple: Dans le menu **MODE**, un appui sur la touche qui correspond au label **FM**, détermine le séparateur décimal et alterne entre le point et la virgule (et ceci dans tous les modes, sauf mode "alpha").

Il y a d'autres exceptions, qu'il serait un peu long et inutile passer en revue.
Citons seulement l'exemple de **BEEP**:
* dans le menu **MODE**, un appui sur la touche correspondante active ou désactive le buzzer (sans rien afficher en ligne de commande).
* Dans le menu **CTRL**, un appui sur la touche correspondante affiche le mot **BEEP** en ligne de commande (modes **PRG**, **ALG**, ou **PRG ALG**), ou éxécute l'instruction **BEEP** (mode "Exécution").

ELEMENTS DE PROGRAMMATION

■ QU'EST CE QU'UN PROGRAMME ?:

Un programme est un objet commençant par le caractère « et se terminant par le caractère ».
On commence par entrer un programme en ligne de commande en actionnant la touche «», ce qui a pour effet de placer les caractères « et » en ligne de commande (l'un en dessous de l'autre, le curseur étant placé après le caractère « initial) et de placer le calculateur en mode "programme" (indicateur **PRG** allumé)
NB: A la fin de ce chapitre, un exemple détaillé décrit l'entrée d'un programme en ligne de commande.

Comme tous les autres objets, les programmes peuvent donc être stockés dans des variables, évalués, placés sur la pile, etc....

Un programme est associé à une série d'actions, actions qui sont réalisées quand le programme est éxécuté (appelé, évalué....). Ces actions sont potentiellement inscrites dans le contenu du programme.

Dans la pratique, toutes les actions réalisées par un programme pourraient l'être par une seule ligne de commande.
De ce point de vue un programme est donc une ligne de commande dont les effets sont stockés dans un objet, et cet objet dans une variable le plus souvent.
Cette façon de faire permet entre autres de corriger plus facilement des erreurs inscrites dans cette ligne de commande, et de répéter un grand nombre de fois les actions qui lui correspondent.

■ "EVALUER" UN PROGRAMME:

L'évaluation d'un programme (ou encore son éxécution) peut être obtenue par l'une des actions suivantes:

* Placer le programme (c'est-à-dire l'objet entre « et ») au niveau 1 de la pile, et éxécuter la commande **EVAL** (ou **→NUM**).
* Si ce programme a été placé dans une variable globale:
  - Presser la touche de menu correspondant dans le menu **VAR**.
  - Placer le nom (entre cotes ') de la variable sur la pile et éxécuter la commande **EVAL**.
  - Placer le nom (sans cotes) de la variable sur la pile (le plus souvent, le nom est entré en ligne de commande, et suivi de **ENTER**).

La plupart du temps, un programme agit sur le contenu de la pile, et en particulier son fonctionnement requiert la présence sur celle-ci d'un certains nombre d'objets. On veillera donc toujours à vérifier la conformité de l'état de la pile avant le lancement d'un programme.

■ CE QUE PEUT CONTENIR UN PROGRAMME:

Un programme peut contenir:

* Des objets quelconques:
  Nombres réels ou complexes, entiers binaires.
  Vecteurs, matrices, listes, chaînes de caractères.
  Objets graphiques, Objets-unités, Objets préfixés....
  Des noms (sans cotes ou avec cotes).
  Des expressions algébriques.
  Des programmes (objets entre « et »).
  Des instructions programmables de la HP48SX:
    Des fonctions: *****, **+**, **-**, **/**, **^**, **SIN**, **AND**, **EXP**, etc...
    Des commandes: **AXES**, **BEEP**, **CLEAR**, **DROP**, etc...

* Des structures de contrôle:
  Boucles: **... START ... NEXT (ou STEP)**
  ... FOR ... NEXT (ou STEP)
  DO ... UNTIL ... END
  WHILE .... REPEAT ... END
  Branchements conditionnels:
  IF ... THEN ... END (ou ... IFT)
  IF ... THEN ... ELSE ... END (ou ... IFTE)
  CASE ... END
  Des traitements d'erreurs:
  IFERR ... THEN ... END
  IFERR ... THEN ... ELSE ... END
  Des structures locales:
  - ... «...»
  - ... '...'

Certaines commandes de la HP48SX ne peuvent apparaître dans un programme, par exemple:
LAST CMD et **LAST STACK** (mais **LASTARG** est autorisé).
EDIT et **VISIT**.

■ L'EXECUTION SEQUENTIELLE D'UN PROGRAMME "SIMPLE":

Si on considère un programme ne contenant que des objets "naturels", c'est-à-dire pas de structure conditionnelle, de boucles, etc..., on peut se le représenter par:

« **Objet1 Objet2 Objet3 ..... ObjetN** »

(L'analogie entre ce programme et l'usage fait habituellement de la ligne de commande est alors totale).

L'évaluation d'un tel programme provoque alors, en séquence, l'évaluation de tous les objets dont il est constitué:

* Les données (nombres réels ou complexes, entiers binaires, tableaux, chaînes de caractères, objets graphiques, etc....) les noms entre cotes, les programmes (objets entre « et ») , les expressions algébriques, les listes, sont tout simplement posés au niveau 1 de la pile.
  L'action associée à ces noms entre cotes, ces programmes, ces expressions,ces listes, peut être assurée par l'instruction **EVAL** (ou **→NUM** si le résultat attendu est numérique).

* Les noms <u>sans cotes</u> et les fonctions de la HP48SX sont évalués.
Pour un nom sans cote inconnu de la HP48SX (du moins dans le répertoire où on se trouve à ce moment là), le nom est placé entre cotes sur la pile.
Si ce nom sans cote est le nom d'une variable visible dans le répertoire où on se trouve (pour les problèmes de "visibilité" des noms de variables, voir le chapitre "Variables et Répertoires"), le contenu de cette variable est évalué.
En particulier si ce nom de variable est celui d'un programme celui-ci est éxécuté. Si ce nom de variable est celui d'un sous-répertoire accessible, celui-ci devient le nouveau répertoire courant.
Une exception doit être faite pour les variables locales (nous y reviendrons plus loin), dont le contenu est posé sur la pile sans être évalué (pour provoquer cette évaluation, on fait suivre ce nom du mot **EVAL**, ou **→NUM** si le résultat attendu est numérique).

<u>Exemples:</u>

a) Considérons le programme:

```
« 1 (2,3) {4 5 6} [7 8 9]
  [[ 10 11] [ 12 13]] #14h
  "Hello" 'Name' « DROP »
  'SIN(15)' »
```

Son évaluation va créer la pile ci-contre:

```
10:  1
 9:  (2,3)
 8:  {4 5 6}
 7:  [7 8 9]
 6:  [[10 11]
     [12 13]]
 5:  #14h
 4:  "Hello"
 3:  'Name'
 2:  « DROP »
 1:  'SIN(15)'
```

(Tous les objets figurant dans ce programme sont des objets "inertes". Leur évaluation se réduit à les placer tels quels sur la pile).
(seul l'entier binaire peut être affiché différemment, si le mode en cours n'est pas le mode hexadécimal)

b) Considérons le programme: « « « « « **1** » » » » ».
Si on le place au niveau 1 de la pile:
Un appui sur EVAL donne « « « « **1** » » » ».
Un nouvel appui sur EVAL donne : « « « **1** » » ».
Un nouvel appui sur EVAL donne : « « **1** » ».
Un nouvel appui sur EVAL donne : « **1** ».
Un nouvel appui sur EVAL place **1** sur la pile.

c) Considérons le programme: « **DUP2 + 3 ROLLD *** »

```
2:  x        2:  x+y
1:  y   ---> 1:  x*y
```

L'effet de ce programme sur la pile est représenté par le diagramme ci-contre:
La pile doit contenir deux nombres x et y aux niveaux 2 et 1, et le programme les remplace par leur somme au niveau 2 et leur produit au niveau 1.

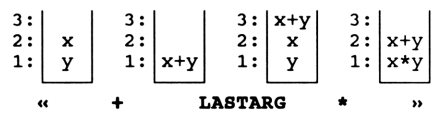
Le fonctionnement de ce programme est le suivant:

```
4:   4:  x   4:   4:  x   4:   4:
3:   3:  y   3:  x   3:  y   3:  x+y   3:
2:  x   2:  x   2:  y   2:  x+y   2:  x   2:  x+y
1:  y   1:  y   1:  x+y   1:  3   1:  y   1:  x*y
```

« **DUP2** **+** **3** **ROLLD** ***** »

Remarque: Si la commande **LASTARG** (récupération des arguments de la dernière instruction éxécutée) est active (Voir dans le menu **MODE**), alors le programme précédent s'écrit plus simplement: « **+ LASTARG *** »

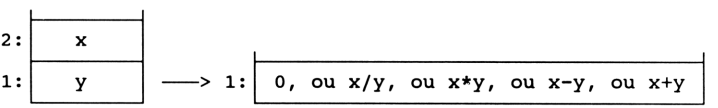
Son fonctionnement est représenté ci-contre:



Le programme précédent, quoique très simple, peut encore être écrit de plusieurs manières différentes (notamment en utilisant des variables locales).

Vous conviendrez, s'il en était besoin, qu'il y a rarement un meilleur programme possible: On s'aperçoit souvent que la recherche d'une meilleure efficacité de l'algorithme se fait au détriment de la lisibilité du programme.
Il est difficile d'optimiser simultanément celle-ci avec une plus grande rapidité et un encombrement moindre en mémoire.

d) Imaginons un programme agissant de la manière suivante:



x,y sont ici deux nombres placés respectivement au niveau 2 et au niveau 1 de la pile.
Le programme attend qu'une des touches ÷ , x , - , ou +, soit actionnée par l'utilisateur.
L'opération correspondante est alors effectuée.
Si l'utilisateur appuie sur une touche autre que ces 4 touches d'opérations, x et y sont remplacés par 0 au niveau 1.

La séquence attendant qu'une touche soit actionnée est:

DO UNTIL KEY END

A la sortie de cette structure répétitive, on trouve au niveau 1 de la pile un entier représentant la position "géographique" de la touche sur le clavier.
Dans cet entier: le chiffre des dizaines est le N° de ligne de la touche sur le clavier (de 1 à 9, de haut en bas), et le chiffre des unités est le N° de colonne (de 1 à 6, de gauche à droite).

Aux touches: ÷, x, -, + correspondent donc les entiers 65, 75, 85, 95.

On peut imaginer la solution suivante (qui évite des tests multiples)

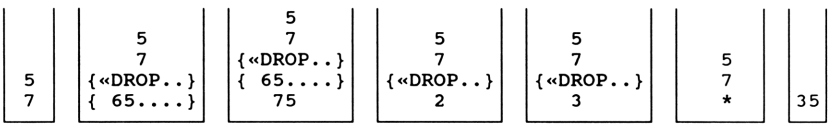
```
« { « DROP2 0 » / * - + }
  { 65 75 85 95 }
  DO UNTIL KEY END
  POS 1 + GET EVAL
»
```

Comment fonctionne ce programme?

Les deux premières instructions placent respectivement:
La liste **L1** = **{ « DROP2 0 » / * - + }** au niveau 2,
La liste **L2** = **{ 65 75 85 95 }** au niveau 1.
On a alors x, y, L1, L2 aux niveaux 4,3,2,1 de la pile.
DO UNTIL KEY END place au niveau 1 le N° c de la touche.

L'instruction **POS** calcule la position p de c dans L2.
(Donc p=0 si c n'apparaît pas dans L2, p=1 si c=65 etc..)
On ajoute ensuite 1 à cette position.
On extrait (**GET**), le programme ou l'instruction qui correspond dans L1.
On évalue ce programme ou cette instruction.

Voilà les états successifs de la pile: sur cet exemple, x=5,y=7, et on appuie sur la touche x (produit).



L'exemple précédent montre qu'il est possible d'écrire des algorithmes évolués en un minimum de temps.
Il montre notamment que l'instruction **POS** permet de gérer des choix multiples d'une manière très naturelle (on peut également utiliser ici une instruction **CASE....END**, voir plus loin).

Néanmoins la HP48SX possède toutes les instructions nécessaires permettant de prendre des décisions, de créer des boucles, de créer des structures locales, etc....

LE VRAI ET LE FAUX SUR LA PILE:

Le fonctionnement des instructions conditionnelles de la HP48SX utilise un test effectué sur le contenu de la pile.
Au moment où le test est effectué, le niveau 1 de la pile doit contenir un nombre réel, ou une expression (que le calculateur évalue lui-même) dont la valeur est un nombre réel.

* Si ce nombre réel est non nul, on considère que le test renvoie la valeur VRAI.
* Si ce nombre réel est nul, on considère que le test renvoie la valeur FAUX.

Il est bon de remarquer que l'opérateur **NOT** transforme le réel 0 en le réel 1, et tout réel non nul en le réel 0 (autrement dit il permet d'inverser le sens logique, VRAI ou FAUX, associé à un nombre réel donné).

Important: le test effectué sur le niveau 1 de la pile "consomme" le réel qui s'y trouve. Si le nombre réel servant à ce test doit être utilisé ultérieurement, il devra donc être sauvegardé (en général simplement dupliqué au niveau 2, pour qu'il reste au niveau 1 après le test).

Opérateurs logiques:

Les opérateurs logiques **NOT**, **AND**, **OR**, et **XOR** permettent de définir des tests élaborés.
Ils s'appliquent à des nombres réels (ou à des expressions à valeurs réelles), et renvoient un réel égal à 1 (valeur logique VRAI) ou à 0 (valeur logique FAUX).

NOT est un opérateur unaire (un seul argument):
NOT inverse la valeur logique du test auquel il s'applique.
Si 'Test' est un réel ou une expression dont la valeur est réelle, alors:
'Test' NOT et **'NOT Test'**
sont équivalents.
'NOT Test' renvoie la valeur VRAI si et seulement si 'Test' renvoie la valeur FAUX.

	F	V
NOT	V	F

AND, **OR** et **XOR** sont des opérateurs binaires (deux arguments):

Voici leurs "tables de vérité":

AND	F	V
F	F	F
V	F	V

OR	F	V
F	F	V
V	V	V

XOR	F	V
F	F	V
V	V	F

Supposons que 'Test1' et 'Test2' soient des réels ou des expressions à valeur réelle.
Pour simplifier, disons que 'Test*' est vraie (resp. fausse) si la valeur logique correspondant à l'évaluation de 'Test*' est le VRAI (resp. le FAUX), c-à-d si la valeur numérique de 'Test1' est un réel non nul (resp. nul).

Avec ces notations:

'Test1 AND Test2' (équivalent en RPN à: **'Test1' 'Test2' AND**)
est vraie si et seulement si 'Test1' et 'Test2' sont vraies toutes les deux.

'Test1 OR Test2' (équivalent en RPN à: **'Test1' 'Test2' OR**)
est vraie si et seulement si l'une au moins de 'Test1' ou de 'Test2' est vraie.

'Test1 XOR Test2' (équivalent en RPN à: **'Test1' 'Test2' XOR**)
est vraie si et seulement si une et une seule de 'Test1' ou de 'Test2' est vraie (c'est-à-dire si 'Test1' et 'Test2' ont des valeurs logiques contraires)

Exemple:
En supposant que X,Y,Z soient des variables dont la valeur est réelle,
L'expression 'X AND (Y XOR Z)' renvoie la valeur vraie si et seulement si:
X est 'vraie' (son contenu est un réel non nul)
et : L'un et l'un seulement des deux réels contenus dans Y et Z est non nul.

...IF...THEN...(ELSE)...END (Menu **PRG BRCH**)

Les instructions **IF...THEN...END** ou **IF...THEN...ELSE...END** sont un moyen très naturel d'éxécuter telles ou telles instructions en fonction d'un résultat calculé.

Leur fonctionnement logique peut être représenté par:
...**IF**...(Test) **THEN** (Instructions-1) **END**
et
...**IF**...(Test) **THEN** (Instructions-1) **ELSE** (Instructions-0) **END**

Le choix est effectué en fonction de l'état de la pile au moment où le mot THEN apparaît dans le programme:
À ce moment là le niveau 1 de la pile doit contenir un réel ou une expression dont la valeur est un nombre réel.

Si ce réel est non nul (valeur logique: **VRAI**), alors la séquence (Instructions-1) est éxécutée, et le programme se poursuit après le mot **END**.

Si ce réel est nul (valeur logique: **FAUX**):
Dans le cas **IF..THEN..END**, le programme se poursuit après **END**.
Dans le cas **IF..THEN..ELSE..END**, la séquence (Instructions-0) est éxécutée et le programme se poursuit après le mot END.

Exemple:
IF SAME THEN "Egaux" ELSE "Différents" END
Cette séquence place le mot "Egaux" ou le mot "Différents" au niveau 1 de la pile suivant que les objets situés aux niveaux 1 et 2 de celle-ci sont identiques ou distincts.

Ce qui précède montre que la position du mot **IF** est peu importante par rapport aux calculs qui conduisent à placer, au niveau 1 de la pile, le nombre réel devant servir au test (c'est la position du mot **THEN** qui compte).

Si on note (Instructions-T) la séquence qui conduit à produire ce réel (ou une expression à valeur réelle) au niveau 1 de la pile, les versions ci-dessous sont donc équivalentes:
* **IF** (Instructions-T) **THEN** ...
* (Instructions-T) **IF THEN** ...
* (Instructions-T1) **IF** (Instructions-T2) **THEN**...
En notant (Instructions-T1) et (Instructions-T2) un partage quelconque de la séquence (Instructions-T) en 2 sous-séquences successives.

Entrée de IF ... THEN ... (ELSE) ... END au clavier:

Il est évidemment possible de taper les mots **IF**, **THEN**, **ELSE**, ou **END**, en mode alpha, mais çà n'est pas très malin.

Le mieux est de sélectionner le menu **PRG BRCH**.

Dans le diagramme ci-dessous, on a représenté la séquence de touches nécessaire pour obtenir tel ou tel écho en ligne de commande.

La position du curseur après chaque entrée est figurée par <▪ .

Séquence de touches	Echo en ligne de commande
PRG BRCH IF	IF <▪
PRG BRCH <=┐ IF	IF <▪ THEN END
PRG BRCH ╒=> IF	IF <▪ THEN ELSE END
PRG BRCH NXT THEN	THEN <▪
PRG BRCH NXT END	END <▪
PRG BRCH PREV ELSE	ELSE <▪

On remarque tout l'intérêt des séquences "<=┐ IF" et "╒=> IF" qui placent respectivement "IF THEN END" et "IF THEN ELSE END" en ligne de commande (chaque mot sur une ligne différente), le curseur se plaçant après le IF initial.

Instructions IF ... THEN ... (ELSE) ... END imbriquées:

Il est possible d'imbriquer plusieurs instructions **IF..THEN..**:

Dans le schéma

(1) **IF** (2) **THEN** (3) **ELSE** (4) **END** (5),

où (1),(2),(3),(4),(5) désignent des séquences d'instructions, il est possible d'imbriquer une instruction **IF..THEN..(ELSE)..END** à l'intérieur d'un des blocs (2),(3),(4),(5), mais pas "à cheval" sur deux blocs consécutifs.

Les facilités qu'apportent les entrées du menu **PRG BRCH** permettent d'ailleurs de créer sans risque de chevauchement de telles structures imbriquées.

Le nombre d'instructions **IF..THEN..** imbriquées n'est pas limité.

Dans de tels cas d'imbrications, il est important de comprendre à quel **THEN** correspond un **ELSE** donné.

La règle est qu'un **ELSE** correspond au dernier **THEN** non encore associé avec un **ELSE**.

Le mieux, pour éviter les confusions, est d'utiliser (sur le papier) une écriture indentée (décalages vers la droite correspondant à des niveaux plus profonds d'imbrications).

Dans l'écriture du programme en ligne de commande, on utilisera systématiquement les séquences "<=┐ IF" et "╒=> IF" du menu **PRG BRCH**.

Exemple de structures IF .. THEN .. (ELSE) .. END imbriquées:

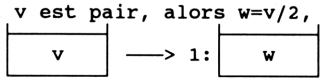
```
IF (Test1)
   THEN
      ------Bloc-1,-Exécuté-si-Test1-est-Vrai------
      ...
      IF (Test2)
      THEN
         +-------------------------------------+
         |  Bloc2,  Exécuté si Test 2 est Vrai.|
         +-------------------------------------+
      ELSE
         ----Bloc-3,-Exécuté-si-Test2-est-Faux----
         ...
         IF (Test3)
         THEN
            +-----------------------------------+
            | Bloc4, Exécuté si Test3 est Vrai. |
            +-----------------------------------+
         END                      Fin du IF (Test3)
      END                         Fin du IF (Test2)
   END                          Fin du IF (Test1).
```

La logique du test multiple apparaît mieux sur ce diagramme, où on note par exemple que le Bloc4 est éxécuté si Test1 est Vrai puis si Test2 est Faux, puis si Test3 est Vrai.

Exemples:

a) La "suite de Syracuse" est définie par la donnée d'un terme initial u (entier positif) et par la relation suivante, qui permet de passer d'un terme v au terme suivant w:

Si v est pair, alors w=v/2, sinon w=3*v+1.



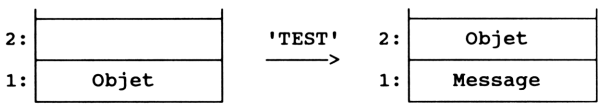
Le programme **'SYR1'** calcule le terme w succédant au terme v. Des appels répétés à 'SYR1' donnent donc tous les termes à partir du terme initial.

Texte du programme 'SYR1'	Commentaires
« DUP IF 2 MOD THEN 3 * 1 + ELSE 2 / END »	Duplique v pour le protéger. Si v est impair: Alors calculer 3*v+1. Sinon calculer v/2. Terminé.

L'instruction **2 MOD** renvoie le reste, dans la division par 2, du nombre situé au niveau 1 (ici l'entier v).Le résultat est donc 1 si v est impair, et 0 si v est pair.
On comprend donc que le sens de **IF 2 MOD THEN** soit "si le nombre au niveau 1 est un entier impair, alors..."

Le programme ci-dessus peut être écrit de manière légèrement différente.
Rappelons déjà que la séquence **DUP IF 2 MOD THEN** équivaut à:
IF DUP 2 MOD THEN , ou encore: **DUP 2 IF MOD THEN** ,
ou enfin: **DUP 2 MOD IF THEN.**
On aurait pu également inverser le test initial en écrivant:
« DUP IF 2 MOD NOT THEN 2 / ELSE 3 * 1 + END »
(mais celà fait une instruction supplémentaire).

b) Le programme **'TEST'** teste un objet au niveau 1, et renvoie l'un des commentaires suivants sur la pile:
"Pas un tableau" si cet objet n'en est effectivement pas un.
"R-vecteur", "C-vecteur", "R-matrice", "C-matrice" selon que cet objet est un vecteur ou une matrice, de nombres réels ou de nombres complexes.



Rappelons que l'instruction **TYPE** donne le type d'un objet quelconque: Le type d'un tableau réel (complexe) est 3 (4).
L'instruction **SIZE** donne la taille d'un vecteur (liste d'un ou de 2 entiers selon que c'est un vecteur ou une matrice).

Texte du programme 'TEST'	Commentaires
« DUP TYPE	Sauve l'objet, prend le type.
DUP 3 ≠	1 si type différent de 3.
OVER 4 ≠	1 si type différent de 4.
IF AND	Si type différent de 3 et 4:
THEN	Alors
DROP	Efface le type.
"Pas un tableau"	Message ad-hoc.
ELSE	Sinon (c'est un tableau):
IF 3 ==	Si tableau réel (type=3)
THEN "R."	Placer "R." au niveau 1,
ELSE "C."	Sinon placer "C.".
END	
OVER SIZE	Taille du tableau (liste).
IF SIZE 1 ==	Si liste d'un seul entier,
THEN "Vecteur"	C'est un vecteur,
ELSE "Matrice"	Sinon c'est une matrice.
END	
+	Ajoute les deux chaînes.
END	
»	

On remarquera que le nombre qui sert au premier test est obtenu par toute une série d'instructions:
DUP TYPE DUP 3 ≠ OVER 4 ≠ AND.
Ce n'est pas une situation exceptionnelle. On peut imaginer des suites d'instructions beaucoup plus importantes encore, ayant pour seul but de déposer un nombre (nul ou non nul) au niveau 1 de la pile, ce nombre étant alors utilisé dans un un test.

N.B: Le programme ci-dessus est utilisé pour illustrer l'entrée d'un programme en ligne de commande, à la fin de ce chapitre.

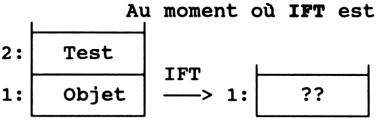
La version ci-dessus du programme **'TEST'** n'est pas la meilleure:
Elle n'est là que comme illustration de structures **IF..THEN..** imbriquées.
On pourrait par exemple utiliser encore des listes et l'instruction **POS** de recherche dans une liste.

IFT et IFTE (Menu **PRG BRCH**)

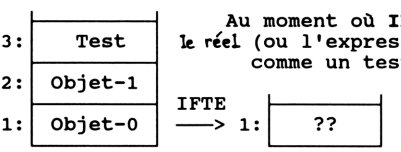
Les instructions **IFT** et **IFTE** sont équivalentes respectivement aux structures **IF..THEN..END** et **IF..THEN..ELSE..END.**

Elles sont en fait la traduction ultime de ces structures dans la logique polonaise inverse (Les structures **IF..THEN..END** n'étant implémentées que parce qu'elles sont bien assimilées du public et qu'elles conduisent à des algorithmes plus "compréhensibles").

Les schémas fonctionnels de **IFT** et **IFTE** sont les suivants:



Au moment où **IFT** est éxécutée, elle considère le réel (ou l'expression à valeur réelle) au niveau 2 comme un test:
* réel nul => FAUX
* réel non nul => VRAI
Si le test renvoie VRAI, alors l'objet situé au niveau 1 (et qui lui est de type quelconque), est évalué.
Si le test renvoie la valeur FAUX, "Objet" est supprimé.



Au moment où **IFTE** est éxécutée, elle considère le réel (ou l'expression à valeur réelle) au niveau 3 comme un test:
* réel nul => FAUX
* réel non nul => VRAI
Si le test renvoie la valeur VRAI alors l'objet "Objet-1", situé au niveau 2, est évalué. Sinon c'est l'objet "Objet-0", situé au niveau 1, qui est évalué.
"Objet-1" et "Objet-0" sont de types quelconques (en particulier ils peuvent être des programmes). Quand l'un d'eux est évalué par **IFTE**, il faut comprendre que l'autre a été éliminé, de même que le réel ayant servi au test.

Exemples:

Le programme suivant teste si un objet est un nombre complexe.
Si c'est le cas, on le remplace par sa partie imaginaire, sinon l'objet est inchangé.

Texte du programme	Commentaires
« DUP TYPE 1 == « IM » IFT »	Protège l'objet. Vrai si complexe. A faire si "vrai" Teste et agit.

Ici, **IM** doit être mis entre « et », sinon l'instruction **IM** est évaluée avant que **IFT** ne soit examinée.

Le programme **'SYR1'** vu plus haut (passage au terme suivant dans la suite de syracuse) peut s'écrire, en utilisant **IFTE:**

Texte du programme 'SYR1'	Commentaires
« DUP 2 MOD « 3 * 1 + » « 2 / » IFTE »	Sauve v et v impair? A faire si v impair. A faire si v pair. Teste parité et agit.

L'inconvénient des instructions **IFT** et **IFTE** est de produire des textes plus difficiles à lire. Mais elles peuvent minimiser la place occupée en mémoire par le programme.

Par exemple: **« "non nul" "nul" IFTE »** occupe 32.5 octets
et **« IF THEN "non nul" else "nul" END »** occupe 40 octets.

Utilisation de IFTE dans une expression algébrique:

IFTE peut être utilisée dans une expression algébrique, avec la syntaxe **'IFTE(Test, Valeur1, Valeur0)'.**
Cette fonction renvoie 'Valeur1' si "Test" est VRAI (non nul) et 'Valeur0' s'il est FAUX (nul).

Exemple: (encore la suite de Syracuse...)
Supposons que la variable N contienne un nombre entier:
L'expression **'IFTE(N MOD 2, 3*N+1, N/2)'**, quand elle est évaluée, teste si N est impair:
Si la réponse est oui (**'N MOD 2'** a renvoyé le résultat 1), alors cette fonction renvoie la valeur correspondant à l'expression **'3*N+1'**.
Si la réponse est non (n pair), elle renvoie la valeur qui correspond à l'expression **'N/2'**.

Exemple:
La fonction **'IFTE(X>1, X, IFTE(X>0, 3-2*X, X+3))'**
renvoie la valeur: de X si X>1,
de 3-2*X si X≤1 et X>0,
de X+3 si X≤1 et X≤0.

CASE...END (Menu **PRG BRCH**)

La structure **CASE..END** permet d'effectuer des choix multiples sans être obligé de créer plusieurs strutures **IF..THEN** imbriquées.

Symboliquement, on peut écrire:

```
CASE
   (Eventualité-1)  THEN  (Réponse-1)  END
   (Eventualité-2)  THEN  (Réponse-2)  END
   ..................................
   (Eventualité-n)  THEN  (Réponse-n)  END
   (Recours)
END
```

(Eventualité-1),..,(Eventualité-n) sont des séquences produisant une valeur logique vraie ou fausse (c'est-à-dire un réel - ou une expression dont la valeur est un réel - non nul, ou 0).
(Réponse-1), .. (Réponse-n) sont des séquences d'instructions.

(Recours) est une séquence facultative.

Le calculateur teste successivement la valeur logique obtenue à l'issue de, dans cet ordre: (Eventualité-1),..,(Eventualité-n), jusqu'à obtenir un premier résultat VRAI.
La séquence d'instructions (Réponse-k) correspondant à ce test positif est alors éxécutée, puis le programme continue après le END final.
Si tous ces tests renvoient la valeur logique FAUX, alors la séquence (Recours), si elle est présente, est éxécutée.
Le programme se poursuit après le END final.

Le nombre de cas différents présents dans cette structure **CASE** est quelconque, et en principe au moins supérieur ou égal à 2.

La structure:
```
CASE
   (Eventualité-1) THEN (Réponse-1) END
END
```
équivaut à:
```
IF (Eventualité-1) THEN (Réponse-1) END
```

De même:
```
CASE
   (Eventualité-1) THEN (Réponse-1) END
   (Recours)
END
```
équivaut à
```
IF (Eventualité-1) THEN (Réponse-1)
                   ELSE (Recours)  END
```

Dans les deux cas ci-dessus, on ne voit donc guère l'intérêt d'utiliser l'instruction **CASE**.

Par contre, la structure:
```
CASE
   (Eventualité-1)  THEN  (Réponse-1)  END
   (Eventualité-2)  THEN  (Réponse-2)  END
   (Eventualité-3)  THEN  (Réponse-3)  END
   (Recours)
END
```

devrait s'écrire:
```
IF  (Eventualité-1)
   THEN  (Réponse-1)
   ELSE  IF  (Eventualité-2)
             THEN  (Réponse-2)
             ELSE  IF (Eventualité-3)
                      THEN (Réponse-3)
                      ELSE (Recours)
                   END
         END
END
```

Dans ce cas, l'utilisation de **CASE..END** apporte une plus grande lisibilité.

Entrée d'une structure **CASE..END** au clavier:

Là encore, il est vivement recommandé d'utiliser les entrées du menu **PRG BRCH**.
Dans le diagramme ci-dessous, on a représenté la séquence de touches nécessaires pour obtenir tel ou tel écho en ligne de commande.

La position du curseur après chaque entrée est figurée par <▪ .

Séquence de touches	Echo en ligne de commande
PRG BRCH CASE	CASE <▪
PRG BRCH <╕ CASE	CASE <▪ THEN END END
PRG BRCH ╒> CASE	THEN <▪ END
PRG BRCH NXT THEN	THEN <▪
PRG BRCH NXT END	END <▪

Les séquences "<╕ CASE" et "╒> CASE" sont très utiles et il est logique de les utiliser systématiquement:
<╕ **CASE** : Ecrit l'ossature de la structure **CASE..END**, en plaçant déjà le **THEN..END** du premier cas.
╒> **CASE** : Permet d'ajouter de nouveaux cas.

Exemple:

Reprenons le programme qui nous a servi d'exemple, pages 30 et 31, et qui attend qu'une touche soit appuyée.
Si cette touche est ÷, x, -, ou +, alors le programme applique l'opération correspondante aux nombres x et y qui sont placés respectivement aux niveaux 2 et 1 de la pile.
Sinon, x et y sont remplacés par le résultat 0 au niveau 1.

Texte du programme	Commentaires
« DO UNTIL KEY END	Attend qu'une touche soit actionnée.
CASE	
DUP 65 ==	Si c'est la touche ÷,
THEN DROP / END	la détruit et effectue x/y
DUP 75 ==	Si c'est la touche x,
THEN DROP * END	la détruit et effectue x*y
DUP 85 ==	Si c'est la touche -,
THEN DROP - END	la détruit et effectue x-y
95 ==	Si c'est la touche +,
THEN + END	effectue x+y
DROP2 0	Sinon, détruit x,y et place 0
END	sur la pile.
»	

On se souvient que chaque Test détruit le réel qu'il prend sur la pile. Ici chaque **THEN** "consomme" l'entier représentant le N° de la touche actionnée.
C'est pourquoi on doit dupliquer cet entier avant de le tester, quitte ensuite à le détruire si un test s'est avéré positif.
Le dernier test (**95 == THEN**) ne duplique pas le N° de touche car ce N° ne sera pas réutilisé par la suite.
L'utilisation d'une variable locale contenant le N° de la touche actionnée résoudrait ce problème de duplication/destruction.

...START...NEXT et ...START...STEP (Menu **PRG BRCH**)

...FOR...NEXT et ...FOR...STEP (Menu **PRG BRCH**)

Les boucles **...START...NEXT** et **...START...STEP** permettent de répéter une séquence d'instructions un nombre déterminé de fois.
Il en est de même des boucles **...FOR...NEXT** et **...FOR...STEP**, à ceci près que ces deux dernières utilisent un compteur de boucle explicite, défini comme une variable locale.

Ces différents types de boucles peuvent s'écrire symboliquement:
(début) (fin) **START** (Instructions) **NEXT**
(début) (fin) **START** (Instructions) (pas) **STEP**
(début) (fin) **FOR** (Compteur) (Instructions) **NEXT**
(début) (fin) **FOR** (Compteur) (Instructions) (pas) **STEP**

(début) et (fin) sont 2 réels, (éventuellement des expressions à valeur réelle) qui déterminent les valeurs initiale et finale du compteur de boucle.
Si (début) est bien la valeur initiale, il est plus juste de dire que (fin) délimite avec (début) l'intervalle des valeurs "utiles" du compteur de boucle.

(pas) est un réel, ou une expression dont la valeur est réelle, qui mesure le "pas" dont il faut incrémenter le compteur quand la séquence (Instructions) a été éxécutée.

(Compteur) est un nom qui sert de compteur de boucle. Ce nom est traité comme une variable locale, c'est-à-dire une variable dont la durée de vie est limitée à l'éxécution de cette boucle.
Mis à part cette limitation, la variable servant de compteur peut être utilisée à l'intérieur de la boucle (on peut même la modifier, ce qui permet, par exemple, de sortir prématurément de la boucle, ou au contraire d'y rester plus longtemps que ce qui était prévu au départ).

Dans le cas **"START"**, le compteur est implicite (c'est-à-dire géré par la HP48SX, mais innaccessible à l'utilisateur).
On utilisera les boucles **"START"** quand le compteur de boucle est inutile dans la séquence (Instructions).

Dans les cas **"NEXT"**, l'incrément est implicite et vaut 1.
On utilisera donc ces structures dans le cas de boucles où l'incrément vaut effectivement 1 (ce qui est souvent le cas).
Les boucles **"STEP"** seront utilisées dans le cas contraire, notamment si le "pas" de la boucle est négatif ou encore s'il est utile de le modifier en cours de route.

Comment fonctionnent ces boucles?

a) Quand les objets **FOR** ou **START** sont rencontrés dans l'éxécution du programme, les valeurs (Début) et (Fin) sont retirées de pile, et la valeur (Début) est assignée au compteur.
Dans le cas d'une boucle **FOR**, cette valeur est placée dans la variable locale, créée pour l'occasion, dont le nom suit immédiatement le mot FOR (Simultanément le nom de la variable disparait de la pile).

b) La séquence (Instructions) est éxécutée.
Dans le cas d'une boucle **FOR**, rappelons qu'il est possible de modifier la valeur du compteur dans (Instructions).
Enfin on peut considérer (dans une boucle **STEP**), que le fait de placer "pas" au niveau 1 de la pile (juste avant **STEP**) est la dernière opération réalisée dans (Instructions).
Rien n'interdit donc de modifier ce "pas".
A l'issue de cette éxécution, le calculateur ajoute au compteur la valeur du "pas" de la boucle (cette valeur disparaissant alors de la pile).
Dans une boucle **START**, cette actualisation du compteur de boucle est implicite. Dans une boucle **FOR**, la nouvelle valeur est placée dans la variable utilisée à cet effet.

c) Le calculateur teste ensuite si la nouvelle valeur du compteur est comprise entre (Début) et (Fin) (quelle que soit par ailleurs la position respective de (Début) et de (Fin)).
Si la réponse est oui, on repart en b).
Si la réponse est non, le programme continue en séquence après le mot **STEP** (ou **NEXT**). Dans le cas d'une boucle **FOR**, la variable locale ayant servi de compteur est supprimée.

De ce qui précède, on tire une remarque essentielle:

la séquence (Instructions), qui constitue le corps de la boucle, est toujours éxécutée au moins une fois (tout simplement parce que le test de continuation est effectué à l'issue de l'éxécution de la séquence (Instructions)).

Exemples de boucles "START".	Nombre d'éxécutions.
« 1 10 START ... NEXT »	Exécutée 10 fois.
« 10 1 START ... NEXT »	Exécutée 1 fois.
« 10 1 START ... -1 STEP »	Exécutée 10 fois.
« 0 8 START ... 2 STEP »	Exécutée 5 fois. (0,2,4,6,8)
« 0 9 START ... 2 STEP »	Exécutée 5 fois. (0,2,4,6,8)

Entrée des boucles START ou FOR au clavier:

Il est évidemment possible de taper les mots **FOR**, **START**, **NEXT**, **STEP** en mode alpha, mais çà n'est pas très malin.

Le mieux est donc de sélectionner le menu **PRG BRCH**.

Dans le diagramme ci-après, on a représenté la séquence de touches nécessaires pour obtenir tel ou tel écho en ligne de commande.

La position du curseur après chaque entrée est figurée par <▪ .

Séquence de touches	Echo en ligne de commande
PRG BRCH FOR	FOR <▪
PRG BRCH <╗ FOR	FOR <▪ NEXT
PRG BRCH ╔> FOR	FOR <▪ STEP
PRG BRCH START	START <▪
PRG BRCH <╗ START	START <▪ NEXT
PRG BRCH ╔> START	START <▪ STEP
PRG BRCH NXT NEXT	NEXT <▪
PRG BRCH NXT STEP	STEP <▪

On remarque tout l'intérêt des séquences: "<╗ FOR" , "╔> FOR" "<╗ START" , "╔> START", et il est recommandé de les utiliser systématiquement.

Les entrées "FOR" et "START" (non préfixées), "STEP" et "NEXT", dans le menu **PRG BRCH** ne devraient être utilisées qu'en cas d'erreur (on a éxécuté <╗ **FOR**, alors qu'il fallait éxécuter ╔> **FOR**, par exemple), ou pour modifier un programme déjà ancien (transformer une boucle **FOR** en boucle **START** par ex.)

N.B: L'utilisation d'une boucle **"STEP"** peut réserver des surprises:

Considérons par exemple:
« **0 1 FOR i i 6 INV STEP** » (le "pas" est 1/6)

On pourrait s'attendre à ce que ce programme place sur la pile les 7 valeurs 0 , 1/6, 2/6, ..., 5/6, 6/6=1 (aux erreurs d'arrondi près)

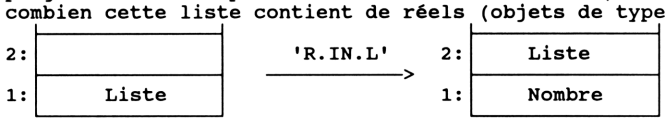
En fait le résultat sur la pile est:
La dernière valeur n'a donc pas été obtenue.
La raison est que le compteur a pris la valeur .833333333335+.166666666667, très légèrement supérieure à 1, après l'affichage de .833333333335, ce qui s'est traduit par une sortie de la boucle.

6:	0
5:	.166666666667
4:	.333333333334
3:	.500000000001
2:	.666666666668
1:	.833333333335

Pour éviter de tels problèmes, il pourra être utile de modifier très légèrement la valeur finale du compteur de boucle, ou de se ramener à un compteur prenant des valeurs entières (quitte à modifier le corps de la boucle):
Ici par exemple, on aurait pu écrire:
« **0 6 FOR i i 6 / NEXT** »

Exemple 1:

Le programme suivant prend une liste au niveau 1, et calcule combien cette liste contient de réels (objets de type 0).



Texte de 'R.IN.L'	Commentaires
« DUP SIZE IF DUP THEN 0 1 ROT FOR i OVER i GET TYPE NOT + NEXT END »	Duplique Liste L. Taille n de L. Si n > 0: Initialise total t. De i=1 à i=n: i-ème objet de L. 1 si réel, 0 sinon. Ajoute au total t. Indice i suivant...

Le test sur la taille de la liste est utile pour éviter une erreur due à l'instruction **"1 GET"** s'appliquant à une liste vide (et cherchant à en extraire le premier élément).
En effet si la liste est vide (n=0), et en l'absence de ce test, la boucle (de i=1 à i=n) est éxécutée une fois, avec i=1...
Dans le texte ci-dessus, la taille n de la boucle est dupliquée (avant d'être testée: **IF DUP THEN**), ce qui permet de la réutiliser: * Comme valeur finale du compteur si n>0.
* Comme indiquant l'absence de réel si n=0.

Exemple 2:

Le programme suivant calcule le plus petit multiple de π qui soit à moins de 1E-4 d'un entier:
Il ne prend rien sur la pile et y laisse le plus petit multiple de π qui soit solution.

Texte du programme 'Nπ'	Commentaires
« π →NUM 1E499 FOR x x DUP .5 + FLOOR - ABS IF .0001 < THEN x 2E499 ELSE π END STEP »	De x=π à x=1E499 (pour simuler une boucle infinie) Place x sur la pile. L'arrondit à l'entier le plus proche k. Distance entre x et k. Si distance inférieure à 1E-4 Alors: Sortir x au niveau 2 et un "pas" de 2E499 au niveau 1. Sinon: un pas de π. Valeur suivante de x...

Le programme s'interromp au bout de 5 secondes, avec le résultat: 354.99969889 (effectivement $113*\pi \approx 354.99969856$, l'erreur d'arrondi est due au fait que le "pas" ne vaut pas exactement π mais est est égal à sa valeur approchée en mémoire).

Le programme précédent montre qu'on peut fort bien utiliser la structure ..**FOR**...**STEP** pour répéter une séquence d'instructions un nombre de fois non connu à l'avance.

Ici on utilise une valeur de début égale à π (et un pas de π pour que les valeurs successives ainsi obtenues soient bien des multiples entiers de π), et une valeur "finale" égale à 1E499, ce qui permet de simuler l'entrée dans une boucle infinie.

De cette boucle infinie, il faut savoir sortir le moment venu, ce qui est réalisé en plaçant un "pas" de 2E499 sur la pile après avoir obtenu la solution.

Le même programme pourrait être écrit en utilisant une structure **WHILE**...**REPEAT**...**END** ou **DO**...**UNTIL**...**END** (voir plus loin).

L' intérêt est ici que la variable locale x est créée par l'instruction **FOR** elle-même, ce qui dispense l'utilisateur d'avoir à le faire.

Boucles FOR (ou START) imbriquées:

On peut bien sûr imbriquer une boucle **FOR** (ou **START**) dans la partie (Instructions) d'une autre boucle.
Le nombre de boucles imbriquées n'est pas limité. La seule condition est de ne pas écrire une boucle "à cheval" sur l'intérieur et l'extérieur d'une autre boucle. Les facilités offertes par le menu **PRG BRCH** évitent de telles erreurs.

La règle est que l'instruction **NEXT** (ou **STEP**) est relative à la dernière boucle ouverte.

Exemple:
Le programme suivant crée un tableau de 10 lignes sur 5 colonnes, le terme situé à la i-ème ligne et la j-ème colonne du tableau valant i^j (tableau créé en deux secondes).

Texte du programme	Commentaires
« 1 10 FOR i 1 5 FOR j i j ^ NEXT NEXT { 10 5 } →ARRY »	De i=1 à i=10: De j=1 à j=5: (i^j) sur la pile. Colonne j suivante... Ligne i suivante... On crée le tableau à 10 lignes et 5 colonnes.

A l'intérieur d'une boucle **FOR** , le compteur de boucle est une variable locale dont la valeur peut être utilisée. On peut en particulier l'utiliser pour définir l'intervalle des valeurs du compteur correspondant à une boucle imbriquée.

Exemple:

Le programme suivant calcule le nombre de triplets d'entiers positifs ou nuls (x,y,z) représentant des points intérieurs à la sphère de centre (0,0,0) et rayon 1000.
Il faut donc résoudre $x^2+y^2+z^2 \leq 1000$ en nombre entiers ≥ 0.
Pour celà on fait varier:
x de 0 à √1000 (de 1 en 1),
à x fixé: y de 0 à √(1000-x²) (de 1 en 1).
à x et y fixé, on calcule FLOOR(√(1000-x²-y²))+1 pour avoir le nombre d'entiers z tels que $x^2+y^2+z^2 \leq 1000$ ("+1" car il faut compter la valeur z=0).

Texte du programme	Commentaires
« 0 0 1000 √ FOR x 0 1000 x SQ - √ FOR y 1000 x SQ - y SQ - √ FLOOR 1 + + NEXT NEXT »	Initialise nombre solutions. De x=0 à x=√(1000): De y=0 à y=√(1000-x²): Calcule √(1000-x²-y²), Prend la partie entière et ajoute 1. Ajoute ce nombre à celui des solutions obtenues. Passer de y à y+1. Passer de x à x+1.

Le résultat est obtenu au niveau 1 de la pile, en 35 secondes, et vaut 17761.

Dans le même ordre d'idées, on verra l'exercice 19 du chapitre consacré aux nombres réels ou complexes, où le problème posé consiste à trouver les solutions entières positives ou nulles de l'équation x+3y+5z=n.

DO ... UNTIL ... END (Menu **PRG BRCH**)

... WHILE ... UNTIL ... END (Menu **PRG BRCH**)

Ces 2 structures permettent de répéter une séquence d'instructions un nombre de fois qui dépend de l'évaluation d'un test logique.
Elles peuvent s'écrire symboliquement:

*** DO (Instructions) UNTIL (Test) END**

La séquence (Instructions) est éxécutée de manière répétitive jusqu'à ce que le test renvoie la valeur logique VRAI (c-à-d fournisse une valeur réelle non nulle).
On peut donc considérer ce test comme un test de "sortie".

Le test est effectué sur la valeur réelle obtenue au niveau 1 de la la pile au moment où le mot END est rencontré.
(le test élimine d'ailleurs ce réel de la pile).

En particulier la position du mot **UNTIL** entre **DO** et **END** est sans importance. Ce n'est que par commodité que l'on sépare d'ordinaire la partie (Instructions) de la partie consistant à produire sur la pile le nombre réel devant être testé.
La partie (Instructions) agit en principe sur le nombre servant au test logique (sans quoi on risque, mais c'est peut-être voulu, d'entrer dans une boucle sans fin).
On note qu'(Instructions) est éxécutée au moins une fois, car le test n'est effectué qu'à l'issue de cette éxécution.

*** WHILE (Test) REPEAT (Instructions) END**

La séquence (Instructions) est éxécutée tant que le test renvoie la valeur logique VRAI (c-à-d soit un réel non nul). Si ce test renvoie la valeur FAUX (réel nul), le programme se poursuit en séquence après le mot **END.**
On peut donc considérer ce test comme un test de "non-sortie".
Le test est effectué sur la valeur réelle obtenue au niveau 1 de la la pile au moment où le mot REPEAT est rencontré. (le test élimine d'ailleurs ce réel de la pile).
La position du mot **REPEAT** entre **WHILE** et **END** est donc là très importante. La partie comprise entre **WHILE** et **REPEAT** peut par contre contenir toute une séquence d'instructions, pouvant n'avoir pour certaines d'entres elles aucun rapport avec la production du nombre réel devant être testé par le mot **REPEAT.**
Les conditions de production du nombre testé par **REPEAT** sont le plus souvent influencées par la séquence (Instructions), sans quoi on peut s'orienter vers une boucle infinie.
On note que la partie (Instructions) n'est pas éxécutée du tout si, au premier "passage", le test renvoie la valeur logique FAUX.

La principale différence logique entre ces deux structures tient au fait que la partie (Instructions):
* Est éxécutée au moins une fois avec DO...UNTIL...END
* Peut ne pas être éxécutée du tout avec WHILE...REPEAT..END

Les séquences **WHILE 1 REPEAT (Instructions) END**
ou **REPEAT (Instructions) UNTIL 0 END**
produisent des boucles sans fin.

Entrée de **DO..UNTIL..END** et **WHILE..REPEAT..END** au clavier:

Le menu **PRG BRCH** fournit un moyen simple d'entrer ces structures en ligne de commande.
La position du curseur est ici représentée par <▪.

Séquence de touches	Echo en ligne de commande
PRG BRCH <┐ DO	DO <▪ UNTIL END
PRG BRCH <┐ WHILE	WHILE <▪ REPEAT END

Exemples:

* Recherche du PGCD par l'algorithme d'Euclide:

Soit deux nombres positifs ou nuls a et b. Leur PGCD peut être obtenu de la manière suivante:
Si b est nul, alors Pgcd(a,b)=a.
Si b n'est pas nul, on calcule le reste r dans la division de a par b, et il se trouve que a et b ont même PGCD que b et r.

Le passage de (a,b) à (b,r) s'accompagne d'une diminution stricte du deuxième élément (b remplacé par r).

Il est logique qu'une application répétée de ce processus conduise, au bout d'un nombre fini d'étapes, à la recherche du Pgcd de deux entiers (s,t), avec t=0, auquel cas:
Pgcd(a,b) = Pgcd(b,r) = ... = Pgcd(s,t) = s.

On en déduit le programme suivant, qui prend 2 entiers a et b aux niveaux 2 et 1, et qui les remplace par leur Pgcd au niveau 1.

Texte de 'PGCD'	Commentaires
« WHILE DUP REPEAT SWAP OVER MOD END DROP »	Tant que b est non nul Répéter: Placer b au dessus de a. b,a,b aux niveaux 3,2,1. b au dessus de r=(a mod b). Séquence terminée. Eliminer b=0 (subsiste a).

Dans cet exemple, l'utilisation de **WHILE..** s'impose par la possibilité que, dès le départ, l'entier placé au niveau 1 soit nul (auquel cas, il ne faut pas utiliser l'instruction **MOD** qui conduirait à une erreur d'éxécution).

* Toujours la suite de Syracuse:

La suite de Syracuse consiste à partir d'un entier positif u, et à construire une suite récurrente de la manière suivante:
Si v est un terme de la suite, le suivant w est égal à v/2 si v est pair et à 3*v+1 si v est impair.
Le programme **'SYR1'** (cf plus haut) assure le passage de v à w.

On observe (mais on ne sait pas le démontrer) que, quelle que soit la valeur initiale u, la suite finit toujours par boucler entre les entiers 4, 2, et 1.

Le programme **'SYR2'** ci-dessous part de l'entier initial u et appelle le programme **'SYR1'**, jusqu'à ce que l'entier obtenu soit égal à 1.
Il place dans sur la pile tous les entiers obtenus, depuis u jusqu'à 1 (Si u=1 on obtient les entiers 1,4,2,1 aux niveaux respectifs 4,3,2,1).

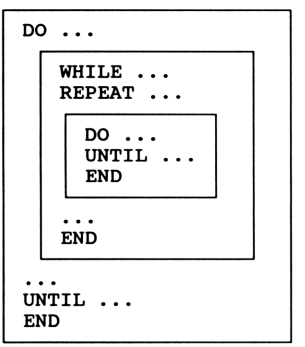
Texte de 'SYR2'	Commentaires
« DO DUP SYR1 UNTIL DUP 1 == END »	Faire: Dupliquer l'entier v. Passer de v à w. Jusqu'à ce que La copie dupliquée de w soit égale à 1. (Effectue le test).

Pour appeler 'SYR2' il faut avoir mis le programme SYR1 dans une variable (nommé 'SYR1'...) et se trouver dans le répertoire où cette opération a été effectuée.

On se rend compte rapidement que certaines valeurs initiales conduisent à des suites très courtes, d'autres au contraire produisent de suites longues et passant par des "sommets" avant de retomber sur 1.
(Essayez par exemple avec u=27).

Dans le texte précédent, la position de **UNTIL** entre **DO** et **END** n'a aucune importance:
On pourrait par exemple écrire:
« DO UNTIL DUP SYR1 DUP 1 == END »
ou « DO DUP SYR1 DUP 1 == UNTIL END »
(On choisit selon son goût...).

On peut enfin imaginer des structures
DO...UNTIL...END ou
WHILE..REPEAT..END
imbriquées, comme le suggère le schéma ci-contre.
Dans une telle situation, il est très recommandé, lors de la conception du texte, d'utiliser des indentations pour indiquer le niveau d'imbrication.



IFERR ... THEN ... (ELSE) ... END (Menu **PROG BRCH**)

Cette structure permet d'"affronter" des situations où une erreur risque de se produire, sans que le programme où elle survient ne s'interrompe. Au contraire une séquence d'instructions est prévue pour le cas où l'erreur se produirait.
Symboliquement, on peut écrire:
IFERR (Risque) THEN (Traitement) END
ou
IFERR (Risque) THEN (Traitement) ELSE (Normal) END

Avec ces notations:
(Risque) est une séquence d'instructions où une erreur peut se produire.
(Traitement) est la séquence d'instructions à éxécuter si cette erreur survient.
(Normal) est une séquence d'instructions à éxécuter uniquement si l'erreur ne s'est pas produite.
La partie **ELSE (Normal)** est facultative.

La séquence **(Risque)** peut très bien se dérouler sans la moindre erreur. Le programme continue alors:
* Après **END** si la clause **ELSE (Normal)** est absente.
* Après **ELSE** si cette clause est présente.

Si une première erreur se produit dans la séquence **(Risque)**:
* Les instructions précédant cette erreur ont été éxécutées normalement.
* Les arguments éventuels de l'instruction ayant provoqué l'erreur sont:
  - replacés sur la pile si la commande **LASTARG** est active.
  - perdus si la commande **LASTARG** est inactive.

  (La commande **LASTARG** permet de récupérer sur la pile le ou les arguments de la dernière instruction éxécutée. On peut l'activer ou la désactiver dans le menu **MODE**.Pour plus de détails, voir le chapitre "La pile opérationnelle".)
* La partie restant à traiter dans la séquence **(Risque)** est abandonnée.
* La séquence **(Traitement)** est éxécutée et le programme continue après **END**.

Instructions **ERRN** et **ERRM** (dans le menu **PRG CTRL**):

* L'instruction **ERRN** renvoie un entier binaire égal au numéro de la dernière erreur survenue.
* L'instruction **ERRM** renvoie une chaîne de caractères contenant le message relatif à la dernière erreur survenue.

(On pourra consulter le tableau des numéros et messages d'erreur dans le manuel de référence de la HP48SX).

Bien que la structure **IFERR..THEN** permette de ne pas interrompre le déroulement du programme si une erreur survient dans la séquence **(Risque)**, cette erreur est tout de même mémorisée et les instructions **ERRN** et **ERRM** renvoient le résultat qui lui correspond.

Il se peut que la séquence **(Risque)** comporte plusieurs risques d'erreur. Les instructions **ERRN** et **ERRM** (appelées par exemple dans la séquence **(Traitement)**) permettent de savoir laquelle des erreurs potentielles s'est réellement produite.

Exemples:

Le programme suivant essaie d'inverser l'objet placé au niveau 1 (nombre réel ou complexe, matrice carrée, ...) et renvoie le le message "inversible" ou "non inversible", suivant le cas.

Texte du programme	Commentaires
« DUP IFERR INV THEN DROP "non " ELSE DROP "" END "inversible" + »	Duplique l'objet. Si on ne peut l'inverser, Eliminer copie, écrire "non " Sinon éliminer inverse, et "". Compléter le message.

Le programme suivant tente de placer les 10 objets situés aux niveaux 1,2,...,10 dans un vecteur. En cas d'impossibilité (c'est-à-dire si ces 10 objets ne sont pas tous des nombres réels ou complexes) , il les place dans une liste. Si c'est encore impossible (car il n'y a pas 10 objets sur la pile), il renvoie le message "Too Few Arguments" au niveau 1.

Texte du programme	Commentaires
« IFERR 10 →ARRY THEN IFERR →LIST THEN DROP ERRM END END »	S'il y a erreur Dans l'instruction 10 →ARRY Alors Si de plus il y a erreur Dans l'instruction 10 →LIST Alors Elimine l'entier 10. Affiche "Too Few Arguments" Terminé. Terminé.

Dans le texte ci-dessus, on a supposé que la commande **LASTARG** est active. C'est pourquoi l'erreur éventuelle dans **10 →ARRY** laisse l'entier 10 sur la pile, et qu'il est nécessaire de l'éliminer par **DROP** si la deuxième erreur (dans **10 →LIST**) se produit.

Entrée d'une structure **IFERR..(ELSE)..END** au clavier:

Le menu **PRG BRCH** permet d'entrer une telle structure de façon très rapide.

La position du curseur est ici représentée par <▪.
(PRG BRCH NXT NXT peut être remplacé par PRG BRCH PREV)

Séquence de touches	Echo en ligne de commande
PRG BRCH NXT NXT <╕ IFERR	IFERR <▪ THEN END
PRG BRCH NXT NXT ╒> IFERR	IFERR <▪ THEN ELSE END
PRG BRCH NXT NXT IFERR	IFERR <▪
PRG BRCH NXT THEN	THEN <▪
PRG BRCH NXT END	END <▪
PRG BRCH NXT NXT ELSE	ELSE <▪

On utilisera donc en priorité les séquences:
<╕ IFERR , qui affiche **IFERR THEN END**
ou ╒> IFERR , qui affiche **IFERR THEN ELSE END**

■ STRUCTURES ET VARIABLES LOCALES:

Dans un programme un tant soit peu évolué, on traite des variables et leur contenu.
Ces variables peuvent être des variables globales, c'est-à-dire apparaissant dans le menu **VAR**. C'est le cas le plus souvent pour les variables dont le contenu est un programme (celui-ci apparaissant donc comme un sous-programme du programme en cours).

Une variable globale existe donc avant que ne débute le programme qui va l'utiliser et après qu'il se soit terminé.
Il est bien sûr possible de créer une variable globale à l'intérieur d'un programme (par l'instruction **STO**), et de l'éliminer avant la fin de celui-ci (par l'instruction **PURGE**).
En tout cas, on doit éviter de conserver dans un répertoire donné des variables qui ne sont plus utiles.

Pour pallier à cette difficulté, la HP48SX permet de créer une catégorie particulière de variables: les variables locales.
La création d'une ou de plusieurs variables locales va de pair avec la création d'une structure locale, qui représente le domaine d'existence et d'accessibilité de cette ou ces variables locales.

Symboliquement, les choses peuvent être représentées ainsi:

On a encadré la structure locale, et à l'intérieur de celle-ci on a séparé la partie "déclarations des variables locales" de la partie "éxécution de la structure locale", cette dernière pouvant prendre la forme d'une expression algébrique, ou d'un programme.

Premier cas: « (Avant) | → (Noms) | 'Expression' | (Après) »

Deuxième cas: « (Avant) | → (Noms) | « Instructions » | (Après) »

On note ici (Avant) et (Après) les séquences d'instructions qui précèdent ou suivent la structure locale.

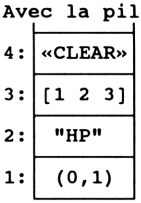
Création des variables locales:

La séquence: → **(Noms)** doit être considérée comme une seule instruction de la HP48SX. C'est elle qui crée les variables locales qui seront utilisables dans la structure locale.
D'une façon plus précise, la syntaxe est:
→ **Nom-1 Nom-2 Nom-3 (...) Nom-n**, où Nom-1, ... Nom-n sont des noms (sans cotes) soumis à la syntaxe habituelle des noms de variables (sauf qu'il est possible d'utiliser des noms apparemment réservés par la HP48SX: on peut par exemple créer une variable SIN mais la fonction Sinus devient innaccessible pendant la durée de la structure locale).

Cette instruction crée non seulement des variables portant ces noms mais elle place dans ces variables un contenu pris sur la pile, en plaçant (pour reprendre les notations ci-dessus):
* Dans Nom-1 (le premier de la liste) le contenu du niveau n.
* Dans Nom-2 le contenu du niveau n-1
...
* Dans Nom-n le contenu du niveau 1.

Exemple:

Avec la pile ci-dessous:




```
La séquence:  → a      place (0,1) dans la variable
                          locale 'a'.
La séquence:  → a b    place "HP" dans 'a' et (0,1)
                          dans 'b'.
La séquence:  → a b c d
   1) crée la variable locale 'a' et y place «CLEAR»,
   2) crée la variable locale 'b' et y place [1 2 3],
   3) crée la variable locale 'c' et y place "HP",
   4) crée la variable locale 'd' et y place (0,1).
```

Utilisation des variables locales:

La partie "Exécution" de la structure locale peut prendre la forme d'une expression algébrique (contenant en principe des références aux variables locales nouvellement créées), ou d'un programme (pouvant les évoquer lui aussi).

Dans le cas d'une expression algébrique, celle-ci est évaluée (en particulier pour ce qui concerne les références aux variables locales) et le résultat est placé au niveau 1 de la pile.

Exemple:

« → a b c '(a+b*c+a/c)/(a*b+b*c+c*a)' »

Ce programme prend a,b,c aux niveaux respectifs 3,2,1, et les remplace par la valeur de **'(a+b*c+a/c)/(a*b+b*c+c*a)'.**

Dans le cas d'un programme, les instructions le constituant sont éxécutées, mais une différence importante avec la notion de variable globale doit être notée:

Si une instruction du programme local consiste à placer, sur la pile, le nom sans cotes d'une variable locale, celà ne suffit pas à évaluer cette variable. Seul le contenu de cette variable est placé sur la pile (ce qui est important si cette variable locale contient un programme par exemple, mais pas si elle contient un objet "inerte" comme un nombre, un tableau...)

Exemples:

```
« .... (1,2)              Ici le complexe (1,2) est placé dans la
  → a                     variable locale 'a', et l'occurence du
  « .... a .... »         nom a (sans cotes) dans la structure
  ...                     locale place bien (1,2) sur la pile.
»

« .... « CLEAR »          Ici le programme « CLEAR » est placé
  → a                     dans la variable locale 'a', mais l'ins-
  « .... a .... »         -truction correspondant à la présence du
  ...                     nom a (sans cotes) dans la structure lo-
»                         -cale place le contenu « CLEAR » sur la
            pile, sans l'évaluer (il suffit alors de faire EVAL)
```

Remarque:

Les variables locales sont soumises à certaines restrictions:

* on ne peut les utiliser dans l'instruction **TAYLR.**
* on ne peut pas les détruire par **PURGE.**

Fonctions-Utilisateur:

Un programme se réduisant à: « → **(Noms) 'Expression'** » se contente de prendre un certain nombre d'objets sur la pile, qu'il place dans des variables locales, pour ensuite évaluer une expression algébrique contenant très certainement des références à ces variables locales.

Ce programme peut être placé dans une variable globale. Celle-ci est alors appelée une fonction utilisateur, en ce sens qu'elle peut être aussi bien utilisée en notations **RPN** que dans une expression algébrique.

Par exemple:

La séquence: « → **x '1/TAN(x)'** » **'COTAN' STO**
crée une fonction-utilisateur du nom de **'COTAN'**.
On pourra ainsi calculer la cotangente du nombre 1991, par exemple, en éxécutant **1911 COTAN**
ou en évaluant **'COTAN(1911)'**.

De même,

« → **a b c 'ABS(a-b)+ABS(a-c)+ABS(b-c)'** » **'PERI' STO**
crée la fonction-utilisateur **'PERI'** qui calcule le périmètre du triangle MNP, où M,N,P sont donnés par leurs coordonnées sous forme de trois nombres complexes m,n,p.
Il suffira d'éxécuter: **m n p PERI**
ou d'évaluer: **'PERI(m,n,p)'**.

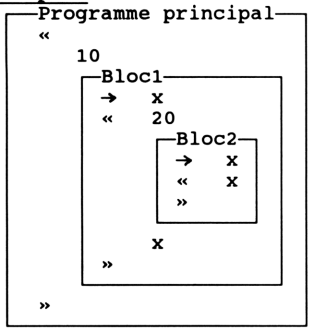
Problèmes de visibilité des variables locales:

On a dit qu'une variable locale était utilisable à l'intérieur de la partie "Exécution" de la structue locale dans laquelle elle a été déclarée.
Cette partie "Exécution" peut elle-même contenir des structures locales. Les variables locales déclarées dans une structure locale "contenant" sont encore visibles à l'intérieur de la struture locale "contenue dans".

Il peut y avoir des problèmes d'homonymie:

* Si on crée une variable locale s'appelant i par exemple, elle occulte (durant sa durée de vie) le nombre complexe i=(0,1).
* Si à l'intérieur d'une structure locale où on a déjà créé une variable locale x, on insère une structure locale où on crée à nouveau une variable locale x (ce qui n'est pas vraiment recommandé pour des questions de lisibilité):
  La variable 'x' la plus interne occulte provisoirement la plus ancienne, et la protège par la même occasion. Quand la structure locale "intérieure" est terminée, on se retrouve dans la structure "contenant", la variable x récupérant la valeur qu'elle avait au moment où a débuté la structure locale "intérieure".
* Rappelons que la HP48SX fait la différence, entre les noms de variables, entre les majuscules et les minuscules. Il n'y donc pas de conflit entre les variables 'x' et 'X'. Généralement, on convient de noter les noms de variables locales en minuscules, mais ce n'est pas une obligation.

Exemple:



L'éxécution de ce programme place 20 au niveau 2, et 10 au niveau 1.

L'explication est la suivante:
L'entrée dans 'Bloc1' se fait en créant la variable locale 'x' et en lui donnant la valeur 10.
A l'intérieur de 'Bloc1', l'entrée dans 'Bloc2' se fait en créant la deuxième variable locale 'x', qui occulte la première, et à qui on donne la valeur 20.
Dans 'Bloc2', on affiche le contenu de 'x' c'est-à-dire 20.
On sort alors de 'Bloc2' pour se retrouver dans 'Bloc1', où on affiche le contenu de 'x', c'est-à-dire 10. On voit bien que le passage dans 'Bloc2' n'a fait qu'occulter, tout en la protégeant, la première définition de la variable 'x'.

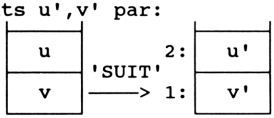
POUR OU CONTRE LES STRUCTURES LOCALES ?:

L'utilisation de structures locales a l'avantage de permettre de créer des variables dont la durée de vie est limitée à l'éxécution d'une séquence d'instructions déterminée, à l'issue de laquelle cette variable n'est plus utile.

On gagne donc le fait de ne pas avoir ni à créer cette variable par **STO**, ni à la supprimer par **PURGE** (sans quoi elle continuerait à encombrer ad aeternam le répertoire **VAR**).
En même temps on garde le confort de pouvoir désigner par un nom (le nom de la variable) un objet (le contenu de la variable) peut-être "encombrant".
Enfin l'utilisation de structures locales va dans le sens d'une plus grande modularité (et d'une meilleure lisibilité) du programme.

Néanmoins, et l'expérience de l'auteur l'illustre, une plus grande aisance dans la programmation de la HP48SX conduit souvent à limiter l'utilisation de structures locales, au profit d'une écriture davantage orientée vers la manipulation d'objets sur la pile (On ne doit pas oublier que c'est le langage naturel de la HP48SX).
Il s'ensuit des programmes plus difficiles à lire (et donc à interpréter par celui qui ne l'a pas écrit, et à corriger par celui qui l'a écrit), mais en général occupant moins de place en mémoire, et d'éxécution plus rapide....
On va illustrer ces défauts et ces avantages par deux exemples.

Exemple1:
On considère deux suites (u) et (v) dont les termes initiaux sont respectivement a et b (deux réels positifs), et qui sont en outre définies par:
On passe des termes u,v aux termes suivants u',v' par:
u' = √(u*v) v' = (u+v)/2.
Le programme 'SUIT' permet de passer des termes u,v aux termes suivants u' et v'.



Avec une structure locale, on peut l'écrire:

```
'SUIT1': « → u v « '√(u*v)' →NUM '(u+v)/2' →NUM » »
```

ou encore "plus près de la pile":

```
'SUIT2': « → u v « u v * √ u v + 2 / » »
```

Sans structure locale, on peut l'écrire:

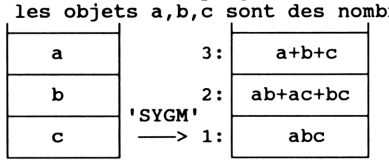
```
'SUIT3': « DUP2 * √ 3 ROLLD + 2 / »
```

La version 'SUIT1' est évidemment la plus lisible, mais la taille en mémoire est de: 72 octets pour 'SUIT1' ,
57 octets pour 'SUIT2' ,
et seulement 30 octets pour 'SUIT3' .

D'autre part les temps de calcul sont les suivants:
1000 appels de 'SUIT1': 1 mn.
1000 appels de 'SUIT2': 41 s.
1000 appels de 'SUIT3': 31 s.

Sur cet exemple, le fait de "rester sur la pile", même si celà demande un peu plus de réflexion, fait gagner du temps et de la place en mémoire.

Exemple2:
On veut écrire un programme réalisant l'opération suivante, où les objets a,b,c sont des nombres réels ou complexes.



On obtient ce qu'on appelle les fonctions symétriques élémentaires $\sigma 1$, $\sigma 2$, $\sigma 3$ de a,b,c.

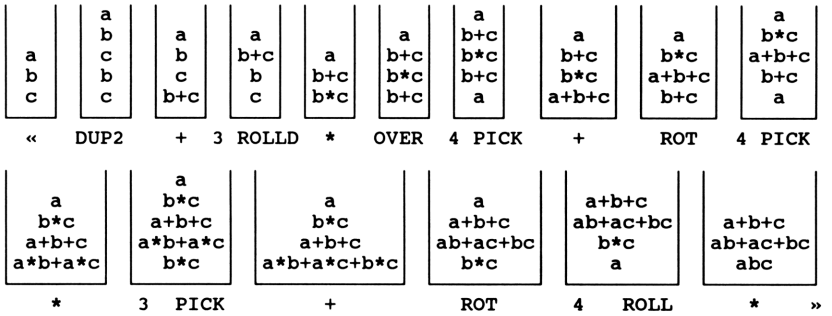
Avec une structure locale, c'est très simple:

```
'SYGM':
« → a b c
   « 'a+b+c' →NUM 'a*b+a*c+b*c' →NUM 'a*b*c' →NUM
   »
»
```

Sans structure locale, il faut réfléchir un peu, ne serait-ce que pour trouver la méthode minimisant le nombre d'opérations.
Voici une solution qui n'est peut-être pas la meilleure:
(essayez de trouver mieux...)

Texte du programme 'SYGM'	Commentaires
« DUP2 + 3 ROLLD *	a,b+c,bc aux niveaux 3,2,1.
OVER 4 PICK +	a,b+c,bc,a+b+c aux niv. 4,3,2,1.
ROT 4 PICK *	a,bc,a+b+c,a*(b+c).
3 PICK +	a,bc,a+b+c,ab+ac+bc.
ROT 4 ROLL *	a+b+c,ab+ac+bc,abc.
»	

Avouons qu'au premier coup d'oeil, il est impossible de savoir ce que fait ce programme.
On comprend son fonctionnement avec les commentaires, ou mieux en suivant les schémas ci-dessous.



Entre ces deux versions de 'SYGM' laquelle choisir? Est-ce la peine de se creuser la tête sur la 2ième version pour gagner quelques 1/100èmes de secondes d'éxécution et quelques octets en mémoire?

MISE AU POINT D'UN PROGRAMME:

La phase de mise au point d'un programme est primordiale. Menée intelligemment, elle vous fait gagner un temps précieux, et elle vous permet de programmer avec plus d'assurance.

Avant d'écrire un programme, il faut savoir ce que vous lui demandez exactement de faire:

* Quels objets va-t-il prendre sur la pile, et quelle est la meilleure façon d'organiser ces objets (faut-il les placer dans une liste, un tableau, ou ces objets occuperont-ils chacun un niveau de la pile ?).

* Quels résultats ce programme doit-il rendre et sous quelle forme?

Une fois ces questions bien posées, le programme à écrire apparaît comme une "boîte noire" permettant d'aboutir à un résultat donné avec un matériel donné, mais le contenu de cette boite noire est encore inconnu.

Tout dépend bien sûr de la complexité de l'algorithme à réaliser, mais l'expérience de l'auteur lui permet peut-être de vous donner quelques conseils:

* Tel Descartes, partagez la difficulté en modules plus simples, et essayez d'adopter une approche "du général au particulier".

* De ce point de vue, et si vous débutez, n'hésitez pas à utiliser des structures locales plutôt que de tenter de tout traiter au "ras de la pile".

* Un crayon à la main, représentez l'état de la pile opérationnelle aux moments importants et essayez de comprendre comment son contenu évolue.

ELEMENTS DE PROGRAMMATION.

D'une manière générale:

* Si vous voulez progresser rapidement, prenez le temps de bien connaître votre machine et toutes ses instructions (on en apprend tous les jours...).

* On progresse autant sinon plus en essayant d'améliorer un programme qui marche qu'en en écrivant la première version.

* Faites les exercices de ce livre, et ne vous reportez aux solutions qu'après avoir ou trouvé, ou séché.

DEBUG, SST, SST↓, NEXT, HALT , CONT, KILL (Menu **PRG CTRL**)

Suspension d'un programme:

La première page du menu **PRG CTRL** contient des commandes qui sont une aide considérable à l'écriture des programmes sur HP48SX.

Quand **HALT** est rencontrée dans un programme, l'éxécution de celui-ci est suspendue.
Un indicateur **HALT** s'allume d'ailleurs en haut de l'écran pour signaler qu'on est dans l'"environnement" provoqué par cette suspension.

L'utilisateur a alors accès à la ligne de commande et à la pile, où il peut faire toutes les opérations habituelles:

* Accéder à n'importe quel menu.
* Faire des calculs sur la pile.
* Appeler d'autres programmes.
* Tracer des graphiques, résoudre des équations.
* Accéder au contenu des variables locales visibles au moment où **HALT** a été rencontré.
* Modifier le contenu de n'importe quelle variable et pourquoi pas le contenu du programme suspendu.
* Effectuer des opérations d'entrée-sortie, par la voie série ou la voie infra-rouge.

Pendant toutes ces opérations, le programme suspendu attend au même point que l'utilisateur veuille bien en reprendre l'éxécution.

Poursuivre l'éxécution par CONT, ou abandonner par KILL:

Pour reprendre l'éxécution du programme là où il avait été suspendu, il suffit de faire **CONT** au clavier (avec la touche spécifique), ou de taper **CONT** , puis **ENTER** (**CONT** est donc une instruction programmable de la HP48SX)

Si vous avez modifié le contenu de la pile (ou de variables) pendant la période de suspension du programme, il faut l'avoir fait en connaissance de cause, car le programme repart en utilisant la pile modifiée et les contenus actuels de toutes les variables.

Pour abandonner l'éxécution du programme en cours,faites **KILL** (dans le menu **PRG CTRL**, ou au clavier en tapant **KILL ENTER**).
On revient alors à l'utilisation normale du calculateur.

Exécuter un programme "pas à pas" par SST ou SST↓:

Quand le programme est suspendu, passez dans le menu **PRG CTRL:**
Ce n'est que dans ce menu que **SST** et **STT↓** peuvent être utilisées, car elles ne sont pas des instructions programmables de la HP48.

Un appui sur la touche de menu **SST** ou **SST↓** provoque:
* Un affichage, en ligne d'état, de l'objet sur le point d'être évalué.
* L'évaluation de cet objet (qui est le plus souvent une instruction du langage de programmation)

D'autres appuis sur **SST** ou sur **SST↓** permettent donc de progresser "pas à pas" dans le déroulement du programme, ce qui est le meilleur moyen de voir les erreurs qui peuvent s'y glisser.

D'ailleurs, si l'instruction évaluée par **SST** ou **SST↓** provoque une erreur, le message correspondant est affiché, et vous pouvez:

* ou bien abandonner par **KILL** pour modifier le programme si vous avez compris comment corriger cette erreur.
* soit corriger la pile de manière à ce que cette instruction qui a provoqué l'erreur (et qui reste en attente) puisse être éxécutée par un nouvel appui sur **SST** ou **SST↓**.

Si vous pensez que le programme est au point vous pouvez le relancer d'un coup jusqu'à sa fin normale, par **CONT.**

Quelques remarques s'imposent pour nuancer l'impression qu'une seule instruction **HALT**, suivi d'appuis sur **SST** ou **SST↓**, permettent d'isoler toutes les étapes de l'éxécution d'un programme.

* Une création de plusieurs variables locales est effectuée en un seul pas. Si la partie "Exécution" de cette structure locale est une expression, elle est évaluée dans la foulée.

* Si l'instruction pointée par **SST** ou **SST↓** est **IFERR:**
IFERR..THEN..(ELSE)..END est éxécutée d'un trait.

* Si l'instruction pointée par **SST** ou **SST↓** est **EVAL** (ou **→NUM**), l'évaluation de l'objet qui en résulte est effectuée d'un trait (notamment si cet objet est un programme posé sur la pile, une expression, un nom de variable entre apostrophes).

* Si l'objet pointé par **SST** ou **SST↓** est un nom (sans apostrophes) désignant une variable globale, et si le contenu de cette variable est un programme:
Un appui sur **SST** éxécute d'un trait ce programme inclus.
Un appui sur **SST↓** "rentre" dans ce programme inclus et en éxécute la première instruction. D'autres appuis sur **SST** ou **SST↓** permettent alors d'éxécuter "pas à pas" le programme inclus.
Quand celui-ci est terminé, on revient au programme principal, qu'on peut à nouveau éxécuter "pas à pas".

La différence entre **SST** et **SST↓** tient donc au fait que **SST↓** "descend" dans les sous-programmes qui apparaissent, dans le programme principal, sous forme d'un nom global sans apostrophes.
On peut ainsi repérer plus facilement l'origine d'une erreur qui surviendrait dans l'éxécution d'un programme contenant lui-même des appels à d'autres programmes.

Exemple:

```
« HALT
  ....
  « *... »
  « *... »
  → prog1   prog2
  « ....
    prog1   EVAL
    ....
    P
    ....
    → (Noms)  'Expression'
    prog2   EVAL
    ...
    IFERR   ?...
    THEN    *...
    ELSE    *...
    END
    ....
» »
```

Sur l'exemple ci-contre, on a représenté un programme dont la première instruction est un **HALT**.

Néanmoins ce **HALT** ne permettra pas,après des appuis sur **SST** ou **SST↓**, "d'entrer" vraiment dans les objets qui sont ici encadrés.

(P : Nom d'une variable globale contenant un programme)
Si on veut pouvoir suivre "pas à pas" le déroulement des sous-programmes **prog1**, **prog2** et de la clause **IFERR...**, il faut placer des **HALT** là où se trouvent les *.

Si vous placez **HALT** à la place de ?, vous pourrez suivre le déroulement entre **IFERR** et **THEN**, mais si une erreur y survient,le programme reste suspendu sur cette erreur.

Enfin, si **P** est le nom (global) d'un programme, et si **P** est l'instruction sur le point d'être éxécutée en "pas à pas", un appui sur **SST** éxécutera le programme **P** d'un trait, alors qu'un appui sur **SST↓** permettra d'y "entrer" pour l'éxécuter lui aussi en "pas à pas".

Visualiser l'instruction suivante, sans l'éxécuter:

Il est parfois utile de savoir quelle est la prochaine instruction à éxécuter en mode "pas à pas", sans pour autant l'éxécuter.
C'est le cas par exemple si cette instruction est le nom global d'un programme, auquel cas il convient de choisir entre **SST** et **SST↓**.

Pour celà, il suffit d'actionner la touche de menu correspondant à l'entrée **NEXT** dans le menu **PRG CTRL**: l'instruction à venir est affichée (au moins en partie), mais n'est pas éxécutée.

"Débuguer" un programme avec des HALT ou avec **DBUG**:

Supposons qu'un programme ne fonctionne pas comme il le faudrait.
Pour identifier ses dysfonctionnements, on dispose de deux méthodes.

* On peut choisir de placer un ou plusieurs **HALT** dans le corps de ce programme.
  Celà présente l'avantage de provoquer la suspension de celui-ci, en un ou plusieurs points bien précis, entre lesquels il n'y a pas de problèmes.
  L'instruction **CONT** permet alors de passer d'un point d'arrêt à l'autre.
  L'inconvénient de cette méthode est qu'on doit éditer le programme pour y placer un ou plusieurs **HALT**, puis l'éditer à nouveau pour les enlever quand le programme est au point.

* Une autre méthode consiste à utiliser la commande **DBUG** du menu **PRG CTRL:**

  On place le programme au niveau 1 (soit l'objet entre « et », soit le nom du programme entre apostrophes). Les arguments éventuellement nécessaires à l'éxécution du programme doivent être placés sur la pile, au-dessus de celui-ci.

  On actionne la touche correspondant à **DBUG.**

  Le programme est alors suspendu sur sa première instruction (Tout se passe donc comme si la 1ère instruction était un **HALT**).

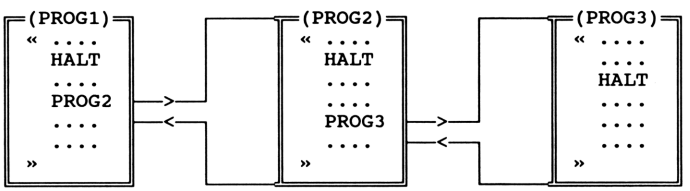
  L'avantage est que le contenu du programme n'a pas à être modifié et l'inconvénient est que l'on doit éxécuter le programme en "pas à pas", depuis le début, même si le point critique se situe assez loin de la première instruction.

Cas où plusieurs programmes sont simultanément suspendus:

Quand un programme est suspendu par **HALT**, on peut appeler d'autres programmes qui peuvent eux aussi être suspendus par un **HALT.**

Dans ce cas, l'instruction **KILL** permet d'abandonner l'éxécution de tous les programmes suspendus. Un appui sur **CONT** permet de terminer l'éxécution du dernier programme, pour retrouver le programme précédemment suspendu, au point où il en était resté.

Exemple: considérons le schéma ci-dessous:



Après le **HALT** de **'PROG1'** des appuis sur **SST** amènent à l'appel de **'PROG2'**, dont l'éxécution est suspendu sur un **HALT.**
A nouveau plusieurs **SST** amènent l'appel de **'PROG3'** suspendu lui aussi. Plusieurs **SST** (ou un **CONT**) terminent **'PROG3'** et nous ramènent dans **'PROG2'** (après l'appel de **'PROG3'**).
De même, la fin de **'PROG2'** nous ramène à l'instruction qui dans **'PROG1'** suit l'appel de **'PROG2'.**

Placer un point d'arrêt avec un message:

L'instruction **PROMPT** équivaut à l'instruction **HALT**, à ceci près qu'elle s'accompagne de l'affichage d'une chaîne de caractères en ligne d'état (c-à-d sur la ligne supérieure de l'affichage).
La syntaxe est: **Chaîne PROMPT.**

La chaîne reste affichée jusqu'à ce que l'utilisateur appuie sur **ENTER**, ou **ON**, ou **CONT** ...
Ce message peut par exemple inviter l'utilisateur à entrer telle ou telle donnée avant de continuer par **CONT.**

■ NOTES SUR LA RECURSIVITE:

On ne peut terminer ce chapitre consacré à une présentation de la programmation sur HP48SX sans parler de la possibilité d'utiliser des programmes récursifs.

Un objet est dit récursif, si sa définition contient une référence, explicite ou implicite, à l'objet lui-même.
Un programme récursif est donc un programme dont l'éxécution comporte un appel au programme lui-même.

On parlera de récursivité simple si un programme **'PROG'** contient un (ou plusieurs appels) directs à **'PROG'**, et de récursivité croisée si, par exemple un programme **'PROG1'** appelle un autre programme **'PROG2'** et si réciproquement **'PROG2'** appelle **'PROG1'**.

Un certain nombre de solutions à des exercices de ce livre utilisent des techniques récursives. Il n'est donc pas utile d'insister lourdement dans ce chapitre sur les avantages et les inconvénients qui en découlent, et qui sont développés dans les commentaires accompagnant ces solutions.

Néanmoins, quelques remarques peuvent être formulées:

* La récursivité réside en général dans la logique du problème à résoudre. Elle ne doit pas être imposée artificiellement si un algorithme itératif convient de façon naturelle.
* Dans l'écriture d'un algorithme récursif, il y a toujours une clause de sortie (on touche le fond du "puits" récursif). Il faut y réfléchir avec soin.
* D'autre part lors d'un appel récursif, veillez à l'aspect de la pile. Celle-ci doit correspondre à une logique constante pour tout nouvel appel du programme.
* En général, un programme récursif peut être dé-récursivé et donner lieu à un programme itératif (l'éxécution d'un programme récursif par la machine n'est pas autre chose).
* L'utilisation de la récursivité conduit souvant à des solutions très élégantes. Néanmoins elle peut se traduire par une rapidité moindre, ou par un encombrement plus important sur la pile (du aux appels répétés, et à l'accumulation de données en cours de traitement).
* Un programme peut par exemple contenir deux appels récursifs: veillez à ce que chacun d'eux traite la moitié du problème. Sinon il se pourrait bien que des calculs identiques soient inutilement effectués en double par ces deux appels.

Un exemple simple:

On a vu plus haut un programme calculant le PGCD de 2 entiers ≥ 0.
Le calcul du PGCD se fait par l'algorithme d'Euclide qui consiste à remarquer que:
* Le PGCD de a et b est égal à a si b est nul.
* Si b est non nul, le PGCD de a et b est égal à celui de b et r, où r est le reste dans la division de a par b.

La récursivité est donc présente dans la logique même de cet algorithme.
La clause de sortie apparaît bien (si b est nul...) et on se rend compte que cet algorithme conduit au bout d'un nombre fini d'étapes à cette clause de sortie (car la suite des entiers traités est strictement décroissante).

Version récursive de 'PGCD'	Commentaires
« IF DUP THEN SWAP OVER MOD PGCD ELSE DROP END »	Si b est non nul, Alors b au dessus de r=(a mod b). Calculer Pgcd(b,r). Sinon éliminer b=0.

On peut même raccourcir la version précédente en utilisant la commande **LASTARG** récupération du ou des derniers arguments utilisés par une instruction).

On obtient:
« IF THEN LASTARG SWAP OVER MOD PGCD END »

Récursivité et sous-programmes:

Imaginons le programme suivant:

```
« ...
  « ...
    sp  EVAL
    ...
  »
  → sp
  « ...
    sp  EVAL
    ...
  »
»
```

On constate qu'il contient une struture locale qui est l'occasion de placer dans une variable locale 'sp' un sous-programme.
Ce sous-programme est appelé depuis le programme principal par l'instruction sp EVAL.
Le contenu du sous-programme contient un appel à lui même, par: sp EVAL.
On pourrait penser que ce sous-programme est une illustration valable de la récursivité.Il n'en est rien!
En effet: dans le contenu, du sous-programme, le nom 'sp' n'est pas connu (il ne l'est que dans la struture locale suivant " → sp « ").
L'instruction "sp EVAL" dans le sous-programme va donc se contenter de placer le nom 'sp' sur la pile...

Il y a une solution qui consiste à élargir le domaine de visibilité du nom local 'sp' dans le programme principal, de manière à ce que ce domaine contienne la définition du sous-programme.
Dans la solution ci-contre la variable locale 'sp' est créée dès le départ et on lui donne le contenu fictif 0.
Le contenu du sous-programme est alors écrit à l'intérieur d'une structure locale où le nom 'sp' est visible.
On place alors ce programme dans 'sp' (qui prend alors son contenu prévu) par: 'sp' STO.

```
« 0 → sp
  « ...
    « ...
      sp  EVAL
      ...
    »
    'sp'  STO
    ...
    sp  EVAL
    ...
  »
»
```

Dans le programme principal, l'instruction sp EVAL va alors appeler le sous-programme, qui pourra s'appeler lui même par le biais de son instruction "sp EVAL".

ENTREE D'UN PROGRAMME EN LIGNE DE COMMANDE:

Pour terminer ce chapitre, voici un exemple détaillé d'entrée d'un programme en ligne de commande.
On reprend le programme 'Test' qui a servi d'exemple page 35.

On part d'une situation où il n'y a pas de ligne de commande en cours, et la position du curseur est désignée par <▪.

Séquences de touches au clavier	Ce que l'on voit du programme à l'écran
«»	« <▪ »
<=┐ ENTER	« DUP <▪ »
PRG OBJ NEXT TYPE	« DUP TYPE <▪ »
3 α ┌═> 1 (ou 3 PRG TEST NEXT =)	« DUP TYPE 3≠<▪ »
PRG STK OVER	« DUP TYPE 3≠ OVER <▪ »
4 α ┌═> 1 (ou 4 PRG TEST NEXT =)	« DUP TYPE 3≠ OVER 4≠<▪ »
PRG BRCH ┌═> IF	IF <▪ THEN ELSE END
PRG TEST AND	IF AND <▪ THEN ELSE END
∇	IF AND THEN <▪ ELSE END
DROP "" α α P <=┐ α (le double α verrouille le clavier en mode alpha) (<=┐ α passe en minuscules)	IF AND THEN DROP "P<▪ ELSE END
A S SPC U N SPC T A B L E A U	... ...ROP "Pas un tableau<▪ ... ...

α ∇ (Le α désactive le mode alpha)	IF AND THEN DROP "Pas un tab .. ELSE <▪ END
PRG BRCH ╔=> IF (ou MENU pour revenir au menu PRG BRCH qui était le menu précédent)	IF <▪ THEN ELSE END
3 α <=┐ 1 (ou 3 PRG TEST NEXT ==)	IF 3==<▪ THEN ELSE END
∇ "" α <=┐ R . (le <=┐ car le mode minuscules est actif depuis "tableau")	IF 3== THEN "R.<▪ ELSE END
∇ "" α <=┐ C . ∇	IF 3== THEN "R." ELSE "C." END <▪
PRG STK OVER PRG OBJ NEXT NEXT SIZE	THEN "R." ELSE "C." END OVER SIZE <▪ END
PRG BRCH ╔=> IF	IF <▪ THEN ELSE END
MENU SIZE 1 α <=┐ 1 (MENU pour revenir au menu précédent qui était le menu PRG OBJ NEXT NEXT)	IF SIZE 1==<▪ THEN ELSE END
∇ "" α α <=┐ V (Le double α verrouille le mode alpha, et le <=┐ pour taper une majuscule)	IF SIZE 1== THEN "V<▪ ELSE END
E C T E U R α ∇ "" α α <=┐ M A T R I C E	IF SIZE 1== THEN "Vecteur" ELSE "Matrice<▪ END
α ∇ +	IF SIZE 1== THEN "Vecteur" ELSE "Matrice" END + <▪
ENTER	« DUP TYPE 3 ≠ OVER 4 ≠ IF AND THEN DROP "Pas un tableau"

LA PILE OPERATIONNELLE

Dans le chapitre "Principes de base", nous avons abordé la notion de pile sur HP48SX pour souligner toute son importance dans le fonctionnement du calculateur.

Quand l'utilisateur valide le contenu de la ligne de commande, cette action a en général pour effet de modifier le contenu de la pile:

* Soit parce que cette ligne de commande contient un ou plusieurs objets 'inertes' qui ont été déposés sur la pile (les objets qui s'y trouvaient avant remontant alors du nombre de niveaux nécessaire pour faire de la place aux nouveaux arrivants).

* Soit parce que la ligne de commande contient des instructions dont l'éxécution influe sur la pile.

De même, la pile est constamment modifiée par l'éxécution en séquence d'un programme créé et lancé par l'utilisateur.

Ces considérations montrent l'importance fondamentale qu'occupe la notion de pile dans l'utilisation de la HP48SX, et le fait qu'il est difficile de déterminer le contenu d'un chapitre consacré à cette notion, tant elle est présente à toutes les pages de ce livre.

Dans ce chapitre, nous nous limiterons aux points suivants:

* L'affichage de la pile à l'écran.

* Touches du clavier représentant de simples opérations sur la pile (**DROP**, **SWAP**, et **CLR**).

* Manipulations d'objets sur la pile, telles que permises par les commandes du menu **PRG STK**.

* La "pile interactive".

* Utilisation de **LAST STACK** (sauvegarde de la pile).

* Utilisation de **LASTARG** (rappel sur la pile des derniers arguments utilisés).

* Influence du contenu de la pile sur la mémoire disponible et sur la rapidité des calculs.

Enfin, ce chapitre se termine par une série d'exercices dont le but est entre autres de vous rendre expert(e) dans les déplacements d'objets sur la pile opérationnelle.

L'AFFICHAGE DE LA PILE A L'ECRAN:

L'écran de la HP48SX peut, à un moment donné de l'utilisation du calculateur, montrer le contenu de 1, 2, 3, ou 4 niveaux successifs de la pile, ou bien masquer complètement le contenu de celle-ci (ce qui ne l'empêche évidemment pas d'exister en arrière-plan).

Il y a de nombreuses raisons possibles qui peuvent limiter la visibilité du contenu de la pile. Passons les en revue.

*** La ligne de commande:**

Le fait d'effectuer une saisie au clavier crée une ligne de commande où s'inscrit le texte tapé.
Dans tous les cas, cette ligne de commande élimine un niveau de visibilité sur la pile.

D'autre part, il est possible d'entrer des données sur plusieurs lignes successives, dans cette ligne de commande, et ce en utilisant la touche <┘.
La création de la ligne de commande peut donc se traduire par une occultation plus ou moins grande de la pile, et aller jusqu'à une eclipse totale de celle-ci.

On quitte la période 'entrée de la ligne de commande' par un appui sur **ENTER** (pour valider), ou par **ON** (pour annuler).
L'appui sur **ON** permet de retrouver l'affichage précédent de la pile.
L'appui sur **ENTER** le permet aussi,à ceci près que l'évaluation de la ligne de commande peut se traduire par une modification du contenu de la pile, ou par un changement dans les conditions de son affichage.

*** Les environnements spécifiques:**

La HP48SX contient un certain nombre d'applications qui se traduisent par un affichage recouvrant la pile (néanmoins celle-ci est toujours présente en fond).

Citons par exemple:
L'environnement graphique **GRAPH**.
L'environnement **EQUATION-WRITER** (pour saisir des expressions ou des objets-unités).
L'environnement **MATRIX-WRITER** (pour saisir des tableaux).
Les possibilités d'afficher temporairement un catalogue ou une configuration particulière:
Touche **REVIEW** (catalogue des entrées du menu actuel).
Catalogue des objets algébriques (╒=> **ALGEBRA**).
Configuration des entrées-sorties (**I/O SETUP**).

En général, un ou plusieurs appuis sur **ON**, par exemple, permettent de revenir à l'état d'affichage de la pile.

De même certaines instructions graphiques (**DISP**, →**LCD**, etc...) peuvent modifier temporairement l'affichage, qui ne montre alors plus l'aspect véritable de la pile.
Normalement la fin du programme en cours (ou sa suspension par **HALT**) suffit à provoquer le retour à l'affichage de la pile.
L'instruction **FREEZE** permet de "geler" un affichage graphique en attendant qu'une touche soit actionnée.

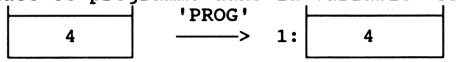
* L'aspect de la pile pendant l'éxécution d'un programme:

Supposons qu'un programme soit lancé par l'utilisateur, et qu'il ne contienne pas d'instruction provoquant des affichages graphiques à l'écran (→**LCD**, **DISP**, **PVIEW**, etc...)

L'aspect qu'avait la pile au moment où le programme est lancé ne change pas pendant l'éxécution de celui-ci (alors que le plus souvent le contenu de cette pile ne cesse de se modifier, et alors que les instructions graphiques de ce programme peuvent, elles, modifier l'aspect de l'écran).
On retrouve l'aspect normal de la pile dans les cas suivants:
* Une erreur d'éxécution survient à l'intérieur du programme.
* Une instruction **HALT** a été rencontrée dans le programme.
  (l'éxécution de celui-ci est donc suspendue)
* L'éxécution du programme est terminée.

On retiendra donc que la pile existe en permanence à l'intérieur de la HP48SX mais qu'elle n'est pas forcément visible à l'écran (au moins en partie), et que l'aspect que peut revêtir l'écran après une instruction graphique peut être trompeur.

Exemple:
Le programme suivant élève au carré le nombre placé au niveau 1 « **2 FREEZE SQ** », mais l'instruction **"2 FREEZE"** gèle l'affichage de la pile au moment où le programme est lancé.
Si on place ce programme dans la variable 'PROG':



L'affichage obtenu est trompeur! Il suffit d'appuyer sur la touche **ON** pour "dégeler" l'affichage, et voir apparaître le résultat correct au niveau 1, c'est-à-dire ...16

* Circuler dans une pile plutôt encombrée:

On sait que tout au plus quatre niveaux de la pile sont visibles à l'écran. La hauteur de la pile n'est pour autant pas aussi limitée, et on peut y mettre des centaines d'objets.
Pour "circuler" dans une pile contenant plus de 4 objets, plusieurs solutions sont envisageables, mais la plus simple est d'activer la "Pile Interactive" .
Pour celà, il suffit:
- D'appuyer sur la touche Δ (sauf si une ligne de commande, formée d'au moins deux lignes, est en cours de saisie).
- De sélectionner la commande ↑**STK** dans le menu **EDIT**.

Le mode "Pile interactive" s'accompagne de l'affichage d'un menu spécifique, que nous étudierons plus loin, et en détail, dans ce chapitre.
Un pointeur indique à l'écran un niveau particulier de cette pile (au début le niveau 1).

Les touches Δ et ∇ permettent de monter et descendre dans la pile. (l'écran se comportant comme une fenêtre sur cette pile), les déplacements pouvant être accélérés par les séquences:
<=╗ Δ et <=╗ ∇ (monter 4 à 4, ou descendre 4 à 4).
╔=> Δ et ╔=> ∇ (aller tout en haut, ou tout en bas).

On sort de la "Pile interactive" par un appui sur **ON** ou **ENTER**.

LA PILE OPERATIONNELLE.

*** Comment les objets sont affichés sur la pile:**

Chacun des niveaux occupés de la pile l'est par un objet et un seul. S'il y a n objets sur la pile, les niveaux occupés sont donc les niveaux numérotés de 1 (niveau le plus bas) à n (niveau occupé le plus haut).

Il y a deux sortes d'affichages possibles d'un objet d'un niveau visible de la pile:

* Affichage en mode 'Compact', ou 'Mono-ligne':
L'objet est affiché sur une ligne, quelque que soit son type.
Si la longueur de cet objet est supérieure ou égale à 19 caractères, seuls les 18 premiers sont réellement affichés, le caractère ... terminant la ligne pour signaler le dépassement.
Les objets situés aux niveaux 2 et au-dessus sont toujours affichés en mode 'Compact'.

* Affichage en mode 'Multi-Lignes':
D'après ce qui précède, ce mode d'affichage ne peut concerner que l'objet situé au niveau 1.
D'autre part, le mode 'Multi-Lignes' ne peut s'appliquer qu'à un objet d'un des types suivants:
- Matrices:
  Les différentes lignes d'une matrice sont affichées sur des lignes d'écran distinctes.
  Si une ligne est trop longue pour être entièrement visible elle est tronquée à ses 18 premiers caractères.
  (chaque ligne est donc affichée en mode 'Compact')
- Listes, Programmes, et Expressions algébriques:
  Ces objets sont affichés sur plusieurs lignes pour être visibles intégralement.
  Dans une liste, chaque ligne d'une matrice peut néanmoins être trop longue pour être entièrement visible.
- Nombres complexes.
- Vecteurs réels à 2 ou 3 coefficients.

On passe du mode 'Compact' au mode 'Multi-Lignes' et vice-versa, en validant l'entrée **ML** du menu **MODE** (page 2 de ce menu):
Si l'intitulé **ML** est suivi d'un petit carré ▪, alors le mode 'Multi-Lignes' est actif.
On peut aussi éxécuter **52 +/- SF** (armer l'indicateur N° -52) pour passer au mode 'Compact', et **52 +/- CF** (désarmer cet indicateur) pour passer en mode 'Multi-Lignes'.

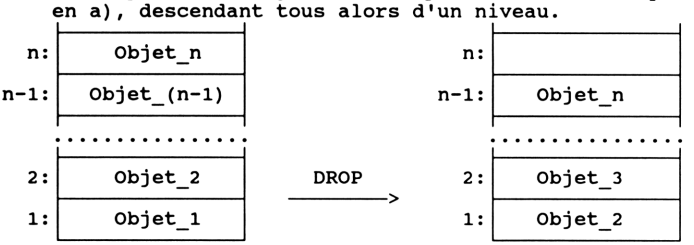
Remarque:
En mode 'Multi-Lignes', l'objet situé au niveau 1 peut réquérir, pour être totalement affiché, un nombre de lignes supérieur au nombre de lignes disponibles.
Seules certaines lignes sont affichées, et il faut passer cet objet en ligne de commande pour pouvoir y "circuler".
Pour celà, on peut:
- Actionner la touche (si aucune ligne de commande n'est présente, ce qui a l'avantage de placer les tableaux dans l'environnement **MATRIX-WRITER**, et les expressions et objets-unités dans l'environnement **EQUATION-WRITER**.
- Actionner la touche **EDIT**.
- Passer dans le mode "Pile Interactive" et choisir l'option **VIEW**.

LES TOUCHES DROP, SWAP, CLR:

Certaines touches du clavier sont dédiées à des opérations simples, et fréquentes, sur la pile. Leur action est doublée par une instruction du langage de la HP48SX

*** La touche et l'instruction DROP:**

Son action est de détruire l'objet qui se trouve au niveau 1 de la pile, les objets situés plus haut sur la pile (s'il y en a), descendant tous alors d'un niveau.

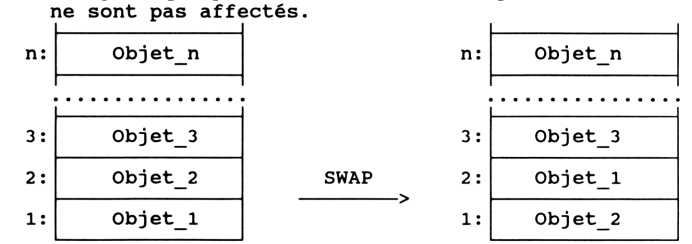


Pour éxécuter **DROP** au clavier, il faut à priori effectuer la séquence <⤶ ⇦ .
Néanmoins, et si aucune ligne de commande n'est en cours, un simple appui sur la touche ⇦ provoque l'éxécution de **DROP**.

La commande **LAST-STACK**, si elle n'est pas désactivée, permet d'annuler l'effet de **DROP** (Idem avec **LASTARG**).
Evidemment, une erreur a lieu si on actionne la touche **DROP** alors qu'aucun objet n'est placé sur la pile.

*** La touche et l'instruction SWAP:**

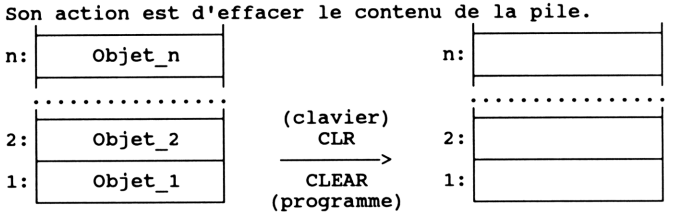
Son action est d'échanger les objets situés aux niveaux 1 et 2 de la pile.
Les objets qui pourraient être situés plus haut sur la pile ne sont pas affectés.



Pour éxécuter **SWAP** au clavier, il faut à priori effectuer la séquence <⤶ ▷ .
Néanmoins, et si aucune ligne de commande n'est en cours, un simple appui sur la touche ▷ provoque l'éxécution de **SWAP**.
La commande **LAST-STACK**, si elle n'est pas désactivée, permet d'annuler l'effet de **SWAP**.
De même, un deuxième appel de **SWAP** annule l'effet du premier.
Evidemment, une erreur a lieu si on actionne la touche **SWAP** et si la pile ne contient pas au moins 2 objets.

*** La touche CLR et l'instruction CLEAR:**

Son action est d'effacer le contenu de la pile.



La commande **LAST-STACK**, si elle n'est pas désactivée, permet d'annuler l'effet de **CLEAR**.
Aucune erreur n'a lieu si on éxécute **CLEAR** et si la pile est déjà vide.

*** La touche ENTER** (hors présence d'une ligne de commande):

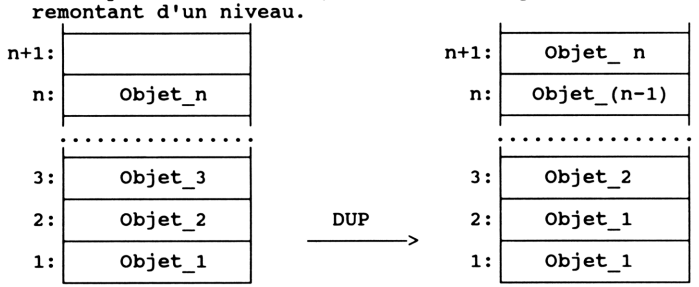
Si aucune ligne de commande n'est en cours de saisie, et si la pile n'est pas vide, un appui sur la touche **ENTER** équivaut à éxécuter l'instruction **DUP** (Voir ci-après).

LE MENU PRG STK:

Le menu **PRG STK** contient 11 commandes en rapport direct avec la pile opérationnelle.

DUP **Duplication de l'objet situé au niveau 1.**

L'instruction **DUP** permet de dupliquer l'objet qui est situé au niveau 1 de la pile. Un nouvel exemplaire de cet objet est ainsi placé au niveau 1, le contenu précédent de la pile remontant d'un niveau.

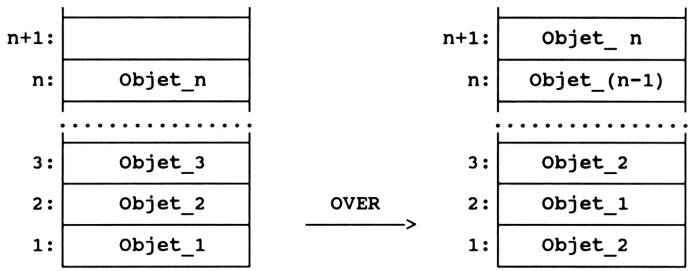


Si aucune ligne de commande n'est en cours de saisie, un appui sur la touche **ENTER** éxécute l'instruction **DUP**.
En mode de saisie "programme", les séquences <=┐ **ENTER**, et ╒=> **ENTER**, affichent **DUP** en ligne de commande.

Il faut bien sûr que la pile soit non vide au départ, sous peine d'une erreur **"DUP Error: Too Few Arguments"**.
L'instruction **DROP** annule l'effet de **DUP**. (on peut bien sûr utiliser **LAST-STACK** si cette commande est active).

OVER **Dupliquer au niveau 1 l'objet situé au niveau 2.**

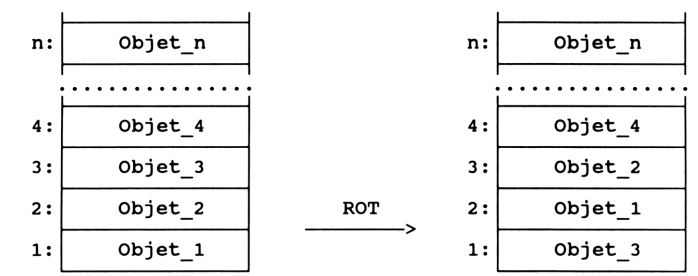
L'instruction **OVER** recopie au niveau 1 de la pile l'objet qui se trouve au niveau 2. Un nouvel exemplaire de cet objet est ainsi placé au niveau 1, le contenu précédent de la pile remontant d'un niveau.



La pile doit évidemment contenir au moins deux objets.
L'instruction **DROP** annule l'effet de **OVER**. (on peut bien sûr utiliser **LAST-STACK** si cette commande est active).
La séquence **OVER OVER** est équivalente à **DUP2**.

ROT **Déplacer vers le niveau 1 l'objet du niveau 3.**

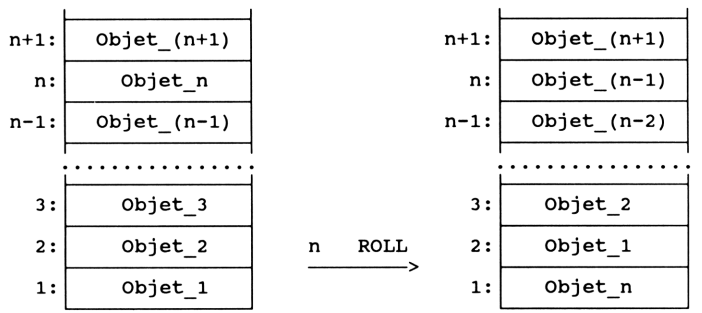
L'instruction **ROT** prend l'objet situé au niveau 3,et le place au niveau 1. Les objets situés initialement aux niveaux 1 et 2 remontent d'un niveau. Les objets qui se situent au dessus du niveau 3 ne sont pas concernés par **ROT**.



La pile doit évidemment contenir au moins trois objets.
L'instruction **ROT** est une forme plus commode de **3 ROLL**.
La commande **LAST-STACK**, si elle est active, annule l'effet de **ROT** (on peut aussi utiliser **3 ROLLD**).
La séquence **ROT ROT** est équivalente à **3 ROLLD**.
La séquence **ROT ROT ROT** laisse la pile en l'état.
La séquence **ROT SWAP** échange les objets des niveaux 2 et 3.
La séquence **SWAP ROT** échange les objets des niveaux 1 et 3.

ROLL — Descendre au niveau 1 un objet de la pile.

Son action est de descendre au niveau 1 un objet situé à un niveau quelconque de la pile.
La syntaxe est **n ROLL** (n entier) pour descendre au niveau 1 l'objet qui se trouve au niveau n.
La place laissée libre au niveau n est comblée par la remontée d'un niveau des objets qui se trouvaient préalablement aux niveaux 1,2,...,(n-1).
Les objets qui pourraient être situés au-dessus du niveau n ne sont pas concernés par la séquence **n ROLL**.



La séquence **1 ROLL** est sans effet.
La séquence **2 ROLL** est synonyme de **SWAP**.
La séquence **3 ROLL** est synonyme de **ROT**.
La séquence **DEPTH ROLL** place au niveau 1 l'objet qui se situait tout en haut de la pile.
La commande **LAST-STACK**, si elle est active, permet d'annuler l'effet de **n ROLL**.
De même, la séquence **n ROLLD** annule **n ROLL**.

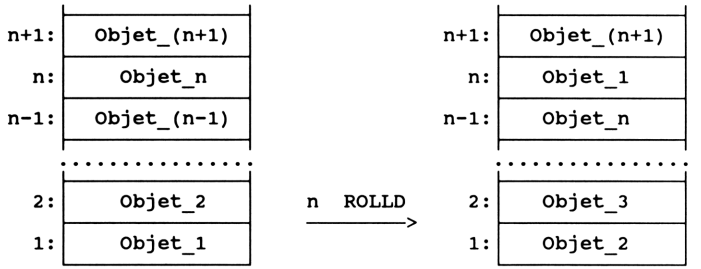
Remarque importante:

Dans ce schéma fonctionnel, ainsi que dans quelques autres qui suivent (instructions **ROLLD**, **PICK**, **DUPN**, **DROPN**), il faut considérer que l'entier n vient se placer sur la pile avant que ne soit éxécutée l'instruction, ce qui déplace d'un étage vers le haut les objets se trouvant sur la pile.
Néanmoins, quand on parle de l'objet situé au niveau n (dans l'instruction **n ROLL**) il faut comprendre le niveau n avant que l'entier n (de la séquence **n ROLL**) ne soit sur la pile.
C'est pour lever ce genre d'ambiguité qu'il semble préférable de ne pas faire figurer l'entier n sur la pile pour décrire les séquences: **n ROLL**, **n ROLLD**, **n PICK**, **n DUPN** et: **n DROPN**.

Dans ces instructions, il n'est pas vraiment nécessaire que n soit un entier. S'il est réel non entier, alors il est arrondi à l'entier ≥ 0 le plus proche.
Dans ces instructions, il y a une erreur si l'entier n est supérieur au nombre d'objets présents sur la pile, et il ne se passe rien si n est nul .

ROLLD — Remonter un objet du niveau 1 au niveau n.

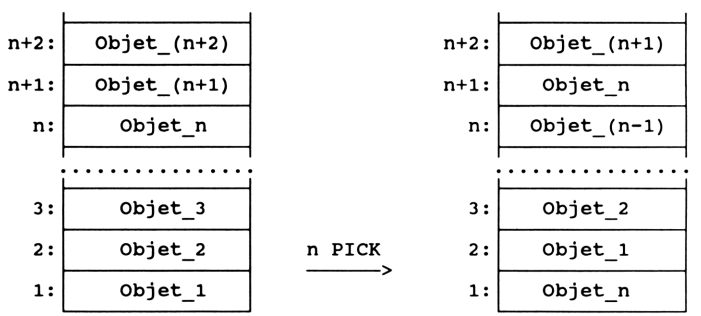
L'instruction **n ROLLD** (où n est entier ≥ 1) déplace l'objet situé au niveau 1 pour le déposer au niveau n de la pile.
Les objets initialement placés aux niveaux 2 à n descendent tous d'un étage, et ceux qui étaient placés au-dessus du niveau n ne sont pas concernés.



n ROLL et **n ROLLD** sont inverses l'une de l'autre.
L'instruction **1 ROLLD** est sans effet.
L'instruction **2 ROLLD** équivaut à **SWAP**.
L'instruction **DEPTH ROLLD** place tout en haut de la pile l'objet qui se trouvait au niveau 1.
La commande **LAST-STACK**, si elle est active, annule l'effet de l'instruction **n ROLLD**.

PICK — Copier au niveau 1 l'objet se trouvant au niveau n.

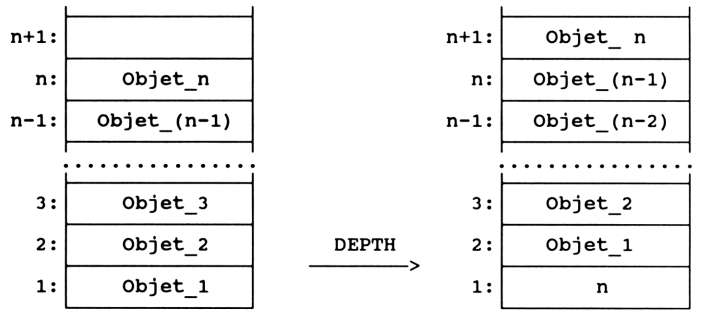
La séquence **n PICK** (où n est un entier ≥ 1), recopie au niveau 1 de la pile l'objet situé au niveau n (sans pour autant comme comme dans **ROLL** le retirer de cette position initiale).
Une copie de cet objet entre donc au niveau 1, le contenu initial remontant d'un niveau.



L'effet de **n PICK** est annulé par **DROP** (ou par **LAST-STACK** si cette commande est active).
La séquence **1 PICK** équivaut à **DUP**.
La séquence **2 PICK** équivaut à **OVER**.

DEPTH **Nombre d'objets présents sur la pile.**

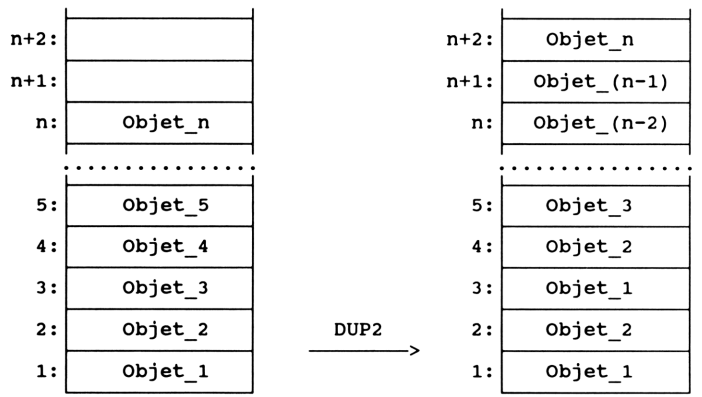
La commande **DEPTH** calcule le nombre d'objets présents sur la pile, c'est-à-dire le nombre de niveaux occupés sur celle-ci. Le nombre entier ainsi obtenu vient se placer au niveau 1, le contenu initial de la pile montant alors d'un niveau.



Si la pile est vide, **DEPTH** retourne la valeur 0.

DUP2 **Duplication des objets situés aux niveaux 1 et 2.**

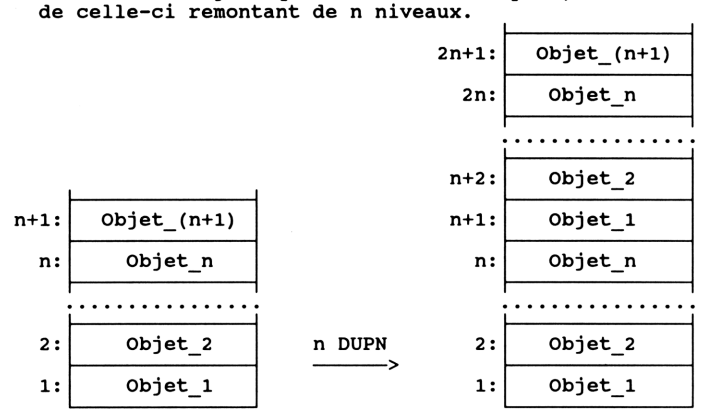
L'instruction **DUP2** duplique les deux objets situés initialement aux niveaux 1 et 2. Ces deux copies font donc leur entrée sur la pile, remontant la pile initiale de deux niveaux.



Au départ, la pile doit bien sûr contenir les deux objets à dupliquer.
DROP2, **2 DROPN** ,ou **DROP DROP**, permettent d'annuler l'effet l'effet de **DUP2**. (on peut bien sûr utiliser **LAST-STACK** si cette commande est active).

DUPN **Dupliquer les objets situés aux niveaux 1 à n.**

La séquence **n DUPN** (où n est un entier ≥ 1), duplique la partie de la pile constituée des niveaux 1 à n (compris).
Ce sont donc n objets qui entrent sur la pile, le contenu initial de celle-ci remontant de n niveaux.

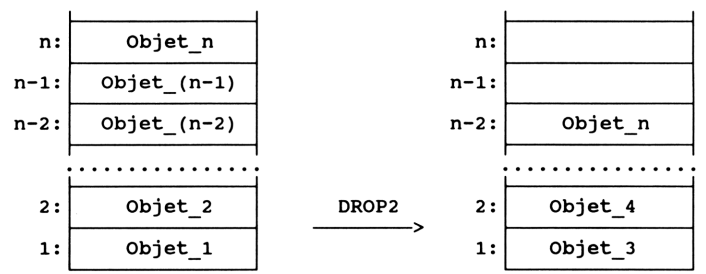


L'instruction **n DROPN** annule l'effet de **n DUPN** (si elle est active, la commande **LAST-STACK** fait de même).

1 DUPN équivaut à **DUP** , et **2 DUPN** équivaut à **DUP2**.
La séquence **DEPTH DUPN** duplique la pile complète.

DROP2 **Suppression des objets situés aux niveaux 1 et 2.**

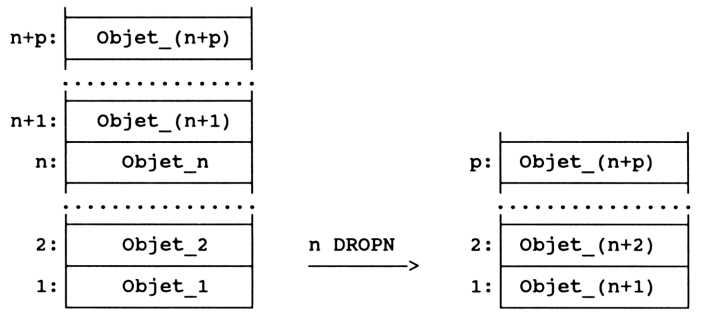
L'instruction **DROP2** supprime les 2 objets situés initialement aux niveaux 1 et 2. Ces deux objets disparaissent de la pile, ce qui en reste descendant de deux niveaux.



Bien sûr, la pile doit initialement contenir au moins 2 objets.
La commande **LAST-STACK**, si elle est active, annule l'effet de **DROP2**.
La commande **LASTARG**, si elle est active, a le même rôle de rattrapage.

DROPN **Détruire les objets situés aux niveaux 1 à n.**

La séquence **n DROPN** (où n est un entier ≥ 1), détruit la partie de la pile constituée des niveaux 1 à n (compris).
Ce sont donc n objets qui disparaissent de la pile. Les objets se situant initialement aux niveaux (n+1) et plus descendent tous de n étages.



Contrairement à **DROP** et **DROP2**,on ne peut ici revenir en arrière avec **LASTARG** (car dans la séquence **n DROPN**, l'argument sauvé est l'entier n).
Seule la commande **LAST-STACK** (si elle est active) permet d'annuler l'effet de **n DROPN**.

1 DROPN équivaut à **DROP** , et **2 DROPN** équivaut à **DROP2**.
La séquence **DEPTH DROPN** équivaut à **CLEAR.**

LA PILE INTERACTIVE:

La "Pile Interactive" est un environnement particulier de la HP48SX, qui permet de parcourir la pile, et d'effectuer aisément les opérations qui viennent d'être décrites (et quelques autres).

Puisqu'il ne s'agit pas de programmation, on ne s'attardera pas.

Il y deux façons de sélectionner le mode "Pile Interactive":

* Actionner la touche Δ (sauf si une ligne de commande formée d'au moins 2 lignes distinctes est en cours de saisie).

* Sélectionner l'entrée ↑**STK** dans le menu **EDIT.**

Un pointeur désigne alors le niveau 1 de la pile, et on peut le déplacer, pour accéder à tel ou tel niveau:

Δ ou ∇ pour monter ou descendre d'un niveau.

<=┐ Δ ou <=┐ ∇ pour monter ou descendre de 4 niveaux.

┌=> Δ ou ┌=> ∇ pour monter au niveau le plus haut, ou passer directement au niveau 1.

Le passage dans le mode "Pile Interactive" provoque l'affichage d'un menu spécifique.

* Si une ligne de commande est en cours de traitement au moment où on sélectionne la "pile interactive", alors ce menu ne contient que l'entrée **ECHO**.
* Sinon le menu contient les entrées:
  ECHO, **VIEW**, **PICK**, **ROLL**, **ROLLD**, **→LIST** sur la page 1,
  et **DUPN**, **DRPN**, **KEEP**, **LEVEL** sur la page 2.

ECHO	Place en ligne de commande l'objet pointé. Si aucune ligne de commande n'était en cours de traitement, elle est ainsi créée, et seule l'option **ECHO** reste présente dans le menu.
VIEW	Edite l'objet pointé, dans un environnement adapté: -**MATRIX-WRITER** pour les tableaux, -**EQUATION-WRITER** pour les expressions et objets-unités. -Ligne de commande pour les autres objets. la touche **EDIT** du clavier donne le même résultat, mais l'édition se fait systématiquement en ligne de commande. Si l'objet pointé est le nom d'une variable, ou un entier n, la séquence ╔=> **VIEW** permet d'éditer,comme précédemment, le contenu de cette variable, ou le contenu du niveau n. La touche **VISIT** du clavier donne le même résultat, mais l'édition se fait systématiquement en ligne de commande.
PICK	Duplique l'objet pointé en plaçant sa copie au niveau 1. (équivaut à **n PICK**, si n est le niveau pointé).
ROLL	Déplace l'objet pointé, en le mettant au niveau 1. (équivaut à **n ROLL**, si n est le niveau pointé).
ROLLD	Place au niveau pointé l'objet qui est au niveau 1. (équivaut à **n ROLLD**, si n est le niveau pointé).
→LIST	Place dans une liste les objets situés du niveau 1 au niveau pointé n (équivaut à **n →LIST**)
DUPN	Duplique les objets situés du niveau 1 au niveau pointé n. (équivaut à **n DUPN**).
DRPN	Détruit les objets situés du niveau 1 au niveau pointé n. (équivaut à **n DROPN**).
KEEP	Détruit tous les objets au-dessus du niveau pointé.
LEVEL	Place au niveau 1 l'entier n égal au niveau pointé.

Enfin:

* La touche ⇦ détruit l'objet pointé.
* **ENTER** et **ON** provoquent la sortie du mode "Pile interactive".
* Après ce retour à l'environnement normal, l'opération **LAST-STACK** (si elle est active) permet de revenir à l'état de la pile avant les modifications subies dans le mode "Pile interactive".

LA SAUVEGARDE DE LA PILE PAR LAST-STACK:

* Activer et désactiver LAST-STACK:

La commande **LAST-STACK** (accessible par une touche du clavier) permet de sauvegarder le contenu de la pile avant une éxécution de **ENTER** (Rappelons qu'il ne s'agit pas uniquement là de l'appui sur la touche **ENTER**, mais de toute commande en éxécution directe, comme l'évaluation d'un objet sur la ligne de menu, ou l'appui sur des touches du clavier provoquant elles-mêmes l'éxécution de **ENTER**).

La commande **LAST-STACK** peut être active ou inactive.
Pour modifier cet aspect du fonctionnement de la HP48SX, il faut passer dans le menu **MODES** et faire **NXT** pour voir apparaître le label **STK**.
Un appui sur la touche correspondante active ou désactive la commande **LAST-STACK**.
On reconnaît qu'elle est active à la présence du petit carré ▪ après l'intitulé **STK**.
Contrairement à la commande **LASTARG** il n'existe pas d'indicateur accessible à l'utilisateur et qui puisse activer ou désactiver **LAST-STACK**.

Evidemment, si **LAST-STACK** est désactivée, le fait d'appuyer sur la touche **LAST-STACK** provoque le message d'erreur:
LAST STACK Disabled.
Le fait que **LAST-STACK** soit active "consomme" une très faible quantité de mémoire (voir plus loin), et n'a pas d'influence décelable sur la rapidité de l'éxécution de **ENTER**..

En règle générale, il est donc recommandé de laisser **LAST-STACK** active (c'est le mode par défaut).

* LAST-STACK et l'éxécution des programmes:

Au moment où on lance l'éxécution d'un programme, de l'une des manières possibles, à savoir en général:
* ou en appuyant sur la touche de menu correspondante à sa position dans la barre de menu,
* ou en tapant son nom (sans cotes '') puis **ENTER**,
* ou en tapant son nom (avec cotes '') puis **EVAL**,

et si **LAST-STACK** est active, alors une copie de la pile est sauvegardée.

L'éxécution du programme a lieu et peut s'interrompre:

1) ou par un appui sur **ON**,

2) ou parce que le programme est réellement terminé,

3) ou par suite d'une erreur à l'éxécution,

4) ou parce que l'instruction **HALT** a été rencontrée (le programme est alors suspendu provisoirement).

Dans les trois premiers cas (arrêt définitif du programme), un appui immédiat sur **LAST-STACK** rétablit le contenu de la pile telle qu'elle était avant que ne débute le programme.

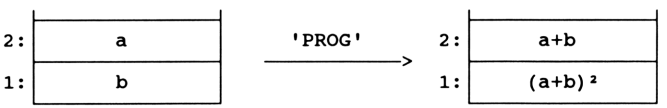
* LAST-STACK et les programmes suspendus:

Dans le cas où l'instruction **HALT** a été rencontrée, le programme se trouve suspendu (témoin **HALT** allumé).
La commande **LAST-STACK** se charge alors de sauvegarder la pile pour chaque appel de **ENTER** dans cet environnement suspendu.
La pile initiale n'est pas pour autant perdue, et elle sera récupérable par **LAST-STACK** quand l'éxécution du programme sera définitivement terminée.

On doit donc considérer que chaque instruction **HALT** nous place dans un environnement particulier, l'environnement initial restant sauvegardé en arrière-plan. Chaque instruction **HALT**, et on sait qu'elles peuvent être imbriquées, nous permet de travailler sur une pile protégée par l'utilisation de la commande **LAST-STACK**.Il y donc autant de protections possibles par **LAST-STACK** qu'il y a de programmes suspendus imbriqués.

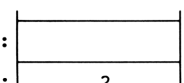
Ces explications peuvent paraître fumeuses. Essayons d'y voir plus clair avec un exemple (le lecteur inquiet peut se rassurer en pensant que la situation décrite ci-dessous a une valeur théorique certaine, mais qu'elle a bien peu de chances d'être rencontrée et utilisée dans la pratique).

Considérons le programme: 'PROG': « **+ HALT DUP SQ** »
'PROG' prend les deux nombres placés aux niveaux 1 et 2, les ajoute, est suspendu par **HALT**, puis duplique le résultat avant de l'élever au carré.
Si on tient pas compte de **HALT**, le schéma fonctionnel de PROG est donc:

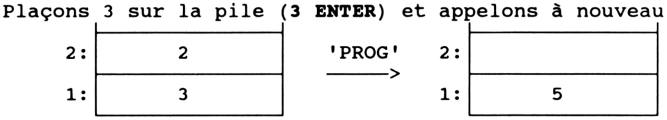


On fait **1 1 ENTER** pour placer 1 et 1 sur la pile et on appelle **'PROG'**.
Le programme est immédiatement suspendu, le nombre 2 apparaissant au niveau 1 (comme somme des deux arguments initiaux).

Faites DROP puis LAST-STACK: vous pouvez voir que le nombre 2 est détruit par DROP puis réapparaît par LAST-STACK.
La commande LAST-STACK est alors active pour sauvegarder l'état de la pile dans l'environnement créé par la suspension par HALT du premier appel de 'PROG'.

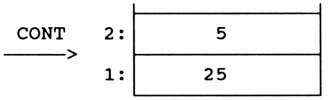


Plaçons 3 sur la pile (**3 ENTER**) et appelons à nouveau **'PROG'**



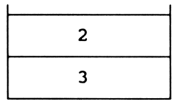
Il y a une nouvelle suspension, et là encore (faites **DROP** et **LAST-STACK** pour le vérifier) la commande **LAST-STACK** se charge de sauvegarder la pile dans l'environnement créé par la suspension par **HALT** du deuxième appel de **'PROG'**.

Continuons l'éxécution du deuxième appel de **'PROG'** par **CONT**.

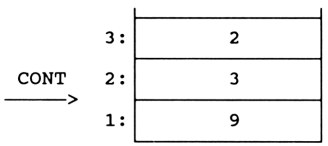


L'état de la pile résulte des instructions **DUP SQ** de **'PROG'**.
Le témoin **HALT** reste allumé pour indiquer que le premier appel de **PROG** est toujours suspendu.

Exécutons LAST-STACK. On obtient la pile:
On retrouve l'état de la pile tel que lors du deuxième appel de 'PROG'.
Lors de ce deuxième appel la pile avait donc été sauvegardée.

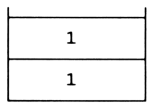


Continuons l'éxécution du premier appel de 'PROG':



Le témoin **HALT** est éteint car il n'y a plus de programme suspendu.
L'état de la pile résulte des deux instructions **DUP** et **SQ** de **'PROG'**.

Si on éxécute LAST-STACK, on retrouve la pile:
C'est l'état de la pile au moment où 'PROG' a été appelé pour la 1ère fois.



On voit sur cet exemple comment **LAST-STACK** sauvegarde la pile lors de chaque éxécution de **ENTER** et notamment lorsque l'on appelle des programmes suspendus par **HALT** à un moment de leur éxécution.
On a vu ci-dessus que **LAST-STACK** a sauvegardé simultanément l'état de trois piles:

* La pile au moment du premier appel de **'PROG'**.
* La pile au moment du deuxième appel de **'PROG'**.
* La pile lors de chaque éxécution de **ENTER** dans l'envi-ronnement créé par la suspension du deuxième appel de **'PROG'**.

Remarque importante:

Pour reprendre l'exemple ci-dessus, notez bien que si par exemple le premier appel de 'PROG' se fait par:
1 1 PROG ENTER, et non par
1 1 ENTER PROG ENTER,
alors les arguments initiaux (1 et 1), n'étant pas sur la pile au moment de l'appel de **'PROG'** (car toujours sur la ligne de commande), ne sont pas récupérables par **LAST-STACK**.

Conclusion:

L'utilisation de **LAST-STACK** est très utile (notamment pour les distraits) car elle permet de retrouver l'état de la pile avant l'éxécution d'une commande. En particulier, on peut ainsi revenir sur une erreur d'éxécution.
Néanmoins, gardez présent à l'esprit que **LAST-STACK** se contente de sauvegarder l'état de la pile, et rien d'autre.
En particulier, si la commande éxécutée modifie le contenu d'une variable (exemple: **'A' 1 PUT**), un appel de **LAST-STACK** ne permet pas de revenir sur cette modification.

LA RECUPERATION DES ARGUMENTS PAR LASTARG:

*** Activer et désactiver LASTARG:**

L'instruction **LASTARG** (accessible par une touche du clavier) permet de récupérer, sur la pile, le ou les arguments de la dernière instruction éxécutée (et ayant nécessité au moins un argument).

La commande **LASTARG** peut être active ou inactive. Pour modifier cet aspect du fonctionnement de la HP48SX, il faut passer dans le menu **MODES** et faire **NXT** pour voir apparaître le label **ARG.**

Un appui sur la touche correspondante active ou désactive la commande **LASTARG.** On reconnaît qu'elle est active au petit carré ▪ après l'intitulé **ARG.**

L'indicateur binaire -55 peut également être utilisé pour activer **LASTARG** ou le désactiver (**55 +/- CF** pour activer , **55 +/- SF** pour désactiver).

Evidemment, si **LASTARG** est désactivée, le fait d'appuyer sur la touche **LASTARG** provoque le message d'erreur:
LASTARG Disabled.

*** Le fonctionnement de LASTARG:**

Les arguments d'une instruction sont les objets pris "en entrée" sur la pile pour en définir l'éxécution.
Par exemple:

- L'instruction **DEPTH** (nombre d'objets présents sur la pile) fonctionne sans argument.
- L'instruction **SIN** (sinus d'un nombre) nécessite un argument (ce nombre).
- L'instruction **+** (addition) nécessite 2 arguments (Les 2 objets à ajouter).
- L'instruction **ROT** du menu **STACK** nécessite 3 arguments (les trois objets des niveaux 1,2,3 de la pile).

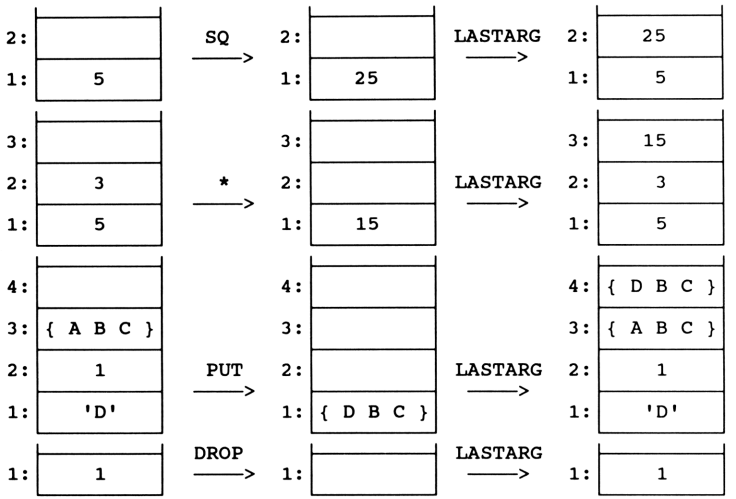
Si **LASTARG** est active, le fait de l'appeler dépose sur la pile, (et dans l'ordre initial) le ou les arguments de la dernière instruction éxécutée, sans pour autant retirer de la pile le résultat de cette instruction.

Si cette instruction ne nécessite pas d'arguments, son éxécution n'influe donc pas sur le ou les objets sauvés par **LASTARG.**

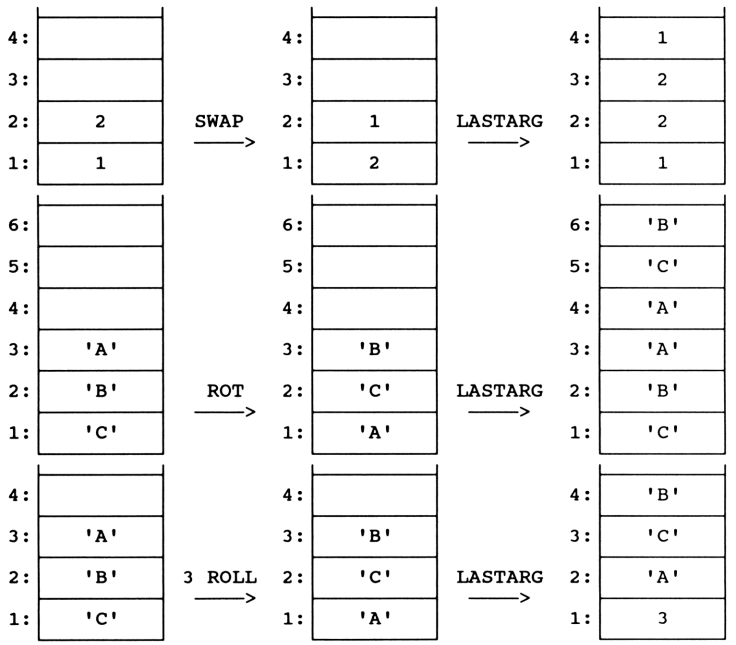
Puisque ce chapitre est consacré à la pile opérationnelle, notons que les instructions:
DROP, SWAP, DUP, OVER, DUP2, DROP2, ROT
ont pour argument(s) le ou les objets de la pile auxquels elles s'appliquent.
Les instructions:**ROLL, ROLLD, PICK, DROPN, DUPN** ont un seul argument à savoir l'entier **n** utile à leur fonctionnement.
Les instructions **CLEAR** et **DEPTH** n'ont pas d'argument.

LA PILE OPERATIONNELLE.

Quelques exemples permettent d'y voir plus clair:



(Pour **DROP** et **DROP2**, **LAST-STACK** et **LASTARG** sont équivalents)



Sur ces deux derniers exemples, on voit que l'instruction **ROT** équivaut, du point de vue fonctionnel, à **3 ROLL**, mais que ces deux instructions sont vues différemment par la commande **LASTARG:**

ROT prend 3 arguments (les trois objets aux niveaux 1,2,3 , tandis que l'instruction **3 ROLL** ne comporte qu'un argument, égal à 3.

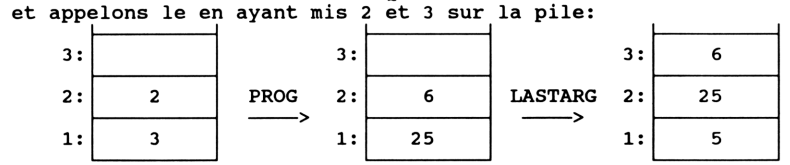
Remarques:

* Les arguments sauvés par **LASTARG** ne doivent pas obligatoirement être sur la pile au moment où l'instruction qui les utilise est appelée: autrement dit ils peuvent encore être sur la ligne de commande (ce qui est une différence avec **LASTSTACK**).
  Par exemple: Dans les séquences **3 SQ ENTER**
  et **3 ENTER SQ ENTER**,
  la commande **LASTARG** sauvegarde l'argument **3** de **SQ**.

* Le fait d'éxécuter **LASTARG** (et donc de placer sur la pile le ou les arguments que cette commande avait sauvegardés) n'élimine pas ce ou ces arguments de la partie de la mémoire où **LASTARG** les conserve.
  Il est donc tout à fait possible d'utiliser **LASTARG** plusieurs fois de suite, les arguments sauvés ne changeant pas.

* Quand vous appelez un de vos programmes, **LASTARG** mémorise le ou les arguments de la dernière instruction éxécutée dans ce programme (qui ne sont pas forcément les objets présents sur la pile au moment où le programme a été lancé).

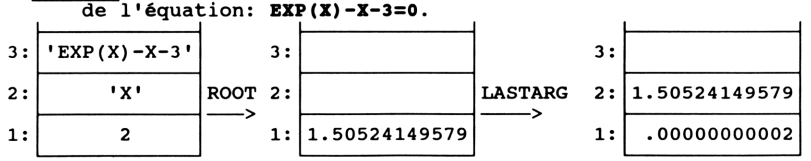
  Considérons le programme suivant:
  'PROG': « * DUP 1 - SQ »
  et appelons le en ayant mis 2 et 3 sur la pile:



  On constate bien que **LASTARG** a mémorisé l'argument (**5**), de la dernière instruction éxécutée (**SQ**) dans le programme.

* Certaines instructions de la HP48SX évaluent un programme ou une expression algébrique. C'est le cas, par exemple, de:
  COLCT, TAYLR, ISOL, QUAD, ∫, ∂, EVAL, →NUM, ROOT, etc..
  L'éxécution d'une telle instruction placera dans **LASTARG** le ou les arguments utilisés dans la dernière instruction éxécutée au sein de ce programme ou de cette expression algébrique.

  Exemple: supposons que l'on veuille trouver la racine positive de l'équation: **EXP(X)-X-3=0.**



Sur cet exemple, on a donné 2 comme point de départ.
La solution trouvée est 1.50524149579.
L'argument conservé par **LASTARG** est .00000000002.
Il représente la dernière valeur calculée de 'EXP(X)-X-3'.
Si on calcule la valeur de cette expression pour 1.50524149579 (qui est le zéro obtenu par **ROOT**),on obtient -.00000000001.
On se rend compte également que .00000000002 (l'argument sauvé par **LASTARG**) est la valeur de l'expression en 1.5052414958.
On voit sur cet exemple que l'argument conservé par **LASTARG** est la dernière valeur de l'expression examinée par le calculateur avant qu'il n'admette avoir trouvé la solution.

* Il y a encore une très grande différence entre **LAST-STACK** et **LASTARG**: cette dernière est une commande qui peut être placée dans un programme, comme n'importe quelle autre instruction.

Le programme suivant, par exemple, prend un nombre k sur la pile la pile et calcule tous les multiples n*k avec 1≤n≤10.

```
'PROG': «  1
           1  10  START
              *   LASTARG  1  +
           NEXT   DROP2
        »
```

Toujours pour l'utilisation de **LASTARG** dans un programme il est intéressant de remarquer que:

- La séquence **IF THEN LASTARG** place au niveau 1 le nombre (non nul) qui vient d'être évalué par **IF**.
- Il en est de même avec: **WHILE REPEAT LASTARG.**
  Le programme « **WHILE REPEAT LASTARG 1 - END** »
  prend un nombre au niveau 1, et lui retire 1
  jusqu'à ce que le résultat soit nul.
- De même les instructions **FOR** ou **START** de début de boucle prennent comme arguments les 2 valeurs extrêmes de la boucle.
  Par exemple: la séquence **1 10 FOR i LASTARG** produit, lors de la première entrée dans la boucle, le placement de 1 au niveau 2 et de 10 au niveau 1.

*** La commande LASTARG et les erreurs d'éxécution:**

Au cours de l'éxécution d'un programme,certaines erreurs peuvent se produire. Le programme est alors interrompu avec affichage d'un message d'erreur de la HP48SX.
Si une telle erreur survient, et si **LASTARG** est active,alors le ou les arguments de l'instruction qui ont conduit à l'erreur sont remis sur la pile (ou en tout cas ils n'en disparaissent pas). Sinon ce ou ces arguments sont perdus.

Exemple:
Avec **LASTARG** désactivée: **0 INV** produit une erreur, mais le réel 0 n'est plus sur la pile.
Avec **LASTARG** activée, la même séquence conduit à la même erreur mais 0 est encore au niveau 1 de la pile.

On sait que l'on peut inclure des contrôles d'erreur dans les programmes de la HP48SX, grâce à l'instruction:
IFERR ... THEN ... (ELSE) ... END.
L'écriture d'un programme utilisant cette instruction doit tenir compte du caractère activé ou non de **LASTARG.**

Exemple:

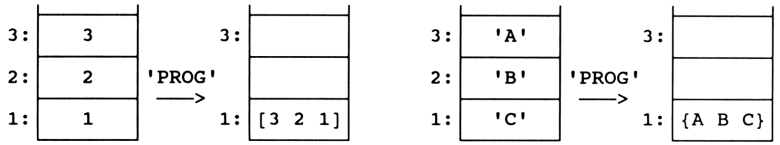
Supposons que nous voulions placer les objets des niveaux 3, 2, et 1 dans un vecteur (si c'est possible, c'est-à-dire si ces objets sont des nombres réels ou complexes).

En cas d'impossibilité de former ce vecteur, nous décidons de placer ces trois objets dans une liste (et là c'est toujours possible, à condition bien sûr qu'il y ait au moins trois objets sur la pile)

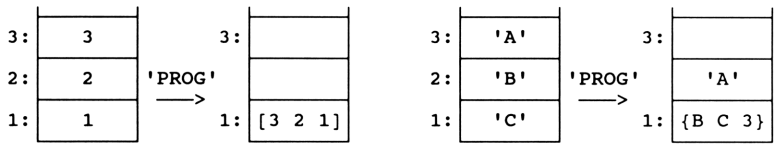
Le programme suivant semble convenir:

'PROG': « IFERR 3 →ARRY THEN 3 →LIST END »

Effectivement, si **LASTARG** est désactivée:



Par contre, si **LASTARG** est activée:



Dans le deuxième cas ci-dessus, l'instruction **3 →ARRY** a bien provoqué une erreur (sans interruption du programme grâce à l'instruction **IFERR**..) et l'argument **3** est resté sur la pile (car **LASTARG** était active).

La séquence **3 →LIST** s'est alors appliquée à une pile contenant **'A'**, **'B'**, **'C'** et **3** respectivement aux niveaux 4,3,2, et 1.

Si on sait que **LASTARG** est active, le programme ci-dessus doit donc être réécrit:

'PROG': « 3 IFERR →ARRY THEN →LIST END »

Sous cette forme il cesse de fonctionner dans le cas où **LASTARG** est désactivée (car le 3 de l'instruction **3 →ARRY** disparaît si cette instruction provoque une erreur).

Comment faire pour écrire une version de ce programme qui marche dans tous les cas,quelque soient les objets placés aux niveaux 1,2,3, et quelque soit l'état (actif ou inactif) de **LASTARG** ?

Cette question est l'objet d'un des exercices proposés à la fin de ce chapitre...

*** LASTARG pour récupérer un PURGE involontaire.**

Supposons que vous ayez éxécuté l'instruction PURGE pour détruire une variable.

Si vous le regrettez, faites **LASTARG** sans attendre, et vous obtiendrez le contenu de la variable au niveau 2, et le nom au niveau 1 (faites **STO** pour revenir au point de départ).

Attention: celà ne marche pas si vous avez détruit une liste de variables, ou le contenu d'un répertoire par **PGDIR**.

*** L'influence de LASTARG sur l'éxécution des programmes:**

Le fait que **LASTARG** soit active consomme une très faible quantité de mémoire: le ou les objets conservés sont en fait mémorisés par leur adresse en mémoire,et non par leur contenu réel.

Par contre on peut se demander si **LASTARG** ne ralentit pas l'éxécution des programmes (contrairement à **LAST-STACK** qui sauve la pile au moment où le programme est appelé, **LASTARG** mémorise successivement les arguments utilisés par chacune des instructions éxécutées dans le programme).

Pour en avoir le coeur net, rien ne vaut une petite expérience.
Considérons donc le programme suivant:

'PROG': « 1 9999 START ROT NEXT »

qui applique 9999 fois l'instruction **ROT** aux trois premiers niveaux de la pile (ce qui remet d'ailleurs la pile dans son état initial).
Plaçons par exemple **'A'**, **'B'**, **'C'** aux niveaux 1, 2, 3, avant d'appeler **'PROG'**.
Si **LASTARG** est désactivée, **'PROG'** dure 49 secondes.
Si **LASTARG** est active, la durée est 50 secondes (la différence est donc infime).

*** Conclusion:**

LASTARG ressemble à **LAST-STACK** en ceci qu'elle permet de récupérer des objets utilisés par une instruction précédente mais elle en diffère par de nombreux aspects:

* Les arguments sont sauvegardés, qu'ils aient été pris sur la la pile ou sur la ligne de commande.
* La sauvegarde des arguments ne se fait pas au détriment des résultats de l'instruction qui les utilise.
* **LASTARG** est appelée tout au long de l'éxécution du programme.
* **LASTARG** est une commande à part entière et elle peut être utilisée dans le texte d'un programme.

Enfin, même si **LASTARG** s'avère dans la pratique moins utile que **LAST-STACK** , son influence insignifiante sur les temps d'éxécution fait qu'il vaut mieux ne pas s'en priver.

■ PILE ET MEMOIRE, PILE ET RAPIDITE DES CALCULS:

Dans ce paragraphe, nous allons voir comment l'encombrement de la pile influe sur la quantité de mémoire disponible et sur la rapidité des calculs.

*** Pile et mémoire:**

Quand on crée des objets, qu'on les place sur la pile et qu'on les y manipule, il est logique de s'intéresser à la taille qu'ont ces objets (ou leur représentation) sur la pile.

Les objets que l'on croit être sur la pile (et dont on voit pourtant tout ou partie du contenu à l'écran) n'y sont pas réellement physiquement présents.
Ils y sont en fait représentés par l'adresse où on les trouve réellement en mémoire, et cette adresse n'occupe que 2.5 octets!

En particulier, tous les objets (du nombre entier à la matrice de dimensions importantes) ont le même 'poids' sur la pile.
N'allez donc pas imaginer que l'instruction **DUP**, par exemple, appliquée à une matrice, va recopier un ou un les différents éléments: elle se contente de recopier les 2.5 octets de l'adresse de cette matrice.
Enfin, on comprend que l'utilisation de la commande **LAST-STACK** n'ait pas d'influence significative sur les performances du calculateur: là encore la pile sauvegardée par **LAST-STACK** ne contient que des adresses.

*** Pile et rapidité des calculs:**

Examinons maintenant dans quelle mesure l'encombrement de la pile peut influer sur la rapidité des calculs. Maintenant que nous savons que les différents niveaux de la pile ne contiennent que des adresses, il ne nous reste que le nombre de niveaux occupés sur cette pile pour juger de son 'encombrement'.

Pour examiner si un encombrement important de la pile nuit à la rapidité des calculs, nous allons considérer les programmes suivants:

'BIGST' : « 1 FOR i i -1 STEP »
et: **'RL500' : « 1 500 START DEPTH ROLL NEXT »**

'BIGST' prend un entier **n** au niveau 1, et place l'entier n au niveau n, l'entier n-1 au niveau n-1, etc..., l'entier 1 au niveau 1.

'RL500' effectue 500 fois de suite l'opération consistant à descendre au niveau 1 l'objet situé tout en haut de la pile, et à remonter tous les autres d'un niveau.

Les temps de calcul suivants sont évidemment approximatifs et indiquent le temps que prend l'éxécution de **'RL500'** après avoir effectué la séquence: **n BIGST** ,où n est un entier prenant les valeurs 10,100,500,1000.

* Après **10 BIGST**: 7 secondes.

* Après **100 BIGST**: 10 secondes.

* Après **500 BIGST**: 17 secondes.

* Après **1000 BIGST**: 27 secondes.

(les temps de calcul ne changent pas suivant l'état, actif ou inactif, des commandes **LAST-STACK** et **LASTARG**).

On constate effectivement une baisse des performances de **'RL500'** sans que celà devienne dramatique.

Conclusion:
Un encombrement excessif de la pile ralentit le fonctionnement du calculateur.
Néanmoins ce ralentissement n'est significatif que pour des piles grossièrement surchargées, situation qui ne se produit pas dans une utilisation 'normale' de la HP48SX.

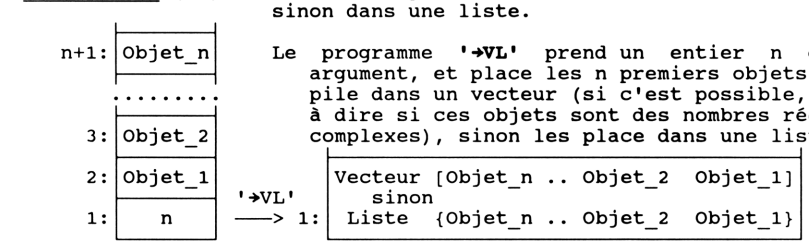
EXERCICES SUR LA PILE OPERATIONNELLE:

La difficulté des exercices est mesurée par des symboles * ou # :

- `*` Très facile.
- `**` Assez simple.
- `***` Déjà un peu plus délicat.
- `****` Plutôt difficile.

Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes précédant dans la liste des exercices.

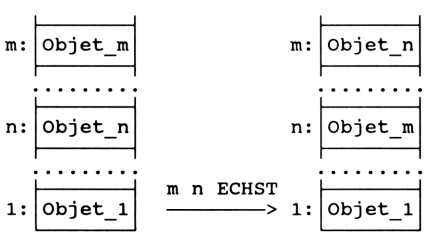
Exercice 1: (**) Mettre, les objets des niveaux 1 à n dans un vecteur sinon dans une liste.



Le programme **'→VL'** prend un entier n comme argument, et place les n premiers objets de la pile dans un vecteur (si c'est possible, c'est à dire si ces objets sont des nombres réels ou complexes), sinon les place dans une liste.

Le programme **'→VL'** doit fonctionner quelque soit l'état (actif ou inactif, de la commande **LASTARG**.
(Voir l'exemple développé dans ce chapitre)

Exercice 2: (**) Echanger deux objets sur la pile.

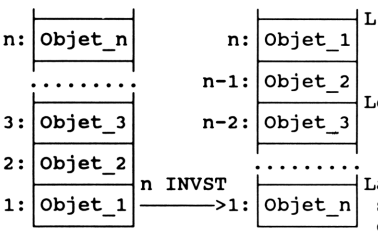


La séquence **m n ECHST** doit échanger les objets situés aux niveaux m et n de la pile.
L'ordre dans lequel on donne m et n est quelconque.
Si m=n, il ne doit rien se passer.
Les positions m et n sont relatives à la pile quand les paramètres nécessaires au fonctionnement de **ECHST** ont été lus.

Exercice 3: (*) Permuter aléatoirement les objets sur la pile.

Il s'agit ici de modifier, d'une façon aléatoire, les positions respectives des différents objets de la pile, et ce jusqu'à ce qu'une touche quelconque soit appuyée.
Dans la solution, le nom du programme est **'RNDST'**.

Exercice 4: (**) Inverser l'ordre des n premiers éléments de la pile.



L'instruction **n INVST** doit inverser les positions respectives des n premiers éléments de la pile.

Le programme s'appelle **INVST**, et il prend un entier n comme unique argument.

La présence de n n'est pas indiquée sur ce schéma fonctionnel mais elle est effective au moment de l'appel de **INVST**.

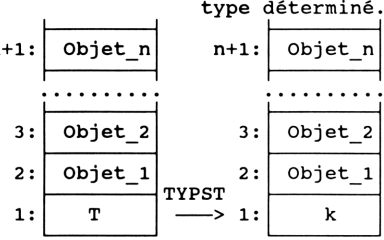
Exercice 5: (**) Eliminer les objets de la pile à partir du niveau n.

Il s'agit ici de détruire les objets de la pile, depuis celui qui est situé au niveau n, jusqu'à celui situé au sommet de cette pile.
Si n est supérieur au nombre d'objets présents sur la pile, celle-ci doit rester inchangée.
Dans la solution, le nom du programme est **'DELN'**.

Exercice 6: (***) Descendre les objets situés aux niveaux pairs, dans la limite des n premiers objets.

Il s'agit ici de considérer les n premiers éléments de la pile, et parmi eux de descendre les objets situés (dans l'état initial de la pile) aux niveaux pairs.
Si on note (P) et (I) les ensembles formés par les objets situés initialement aux niveaux pairs (resp. impairs), dans la limite des n premiers niveaux, les objets de (P) doivent à la fin se trouver tous en dessous de tous les objets de (I).
Cependant pour les objets de (P), la hiérarchie des niveaux doit être conservée (idem pour (I)).
Dans la solution, le nom du programme est **'PBIH'**.

Exercice 7: (**) Compter combien il y a, sur la pile, d'objets d'un type déterminé.



Le programme **'TYPST'** détermine le nombre d'objets qui sont de type T sur la pile.
T est un entier compris entre 0 et 9 conformément à la description des types sur la HP48SX.
Le résultat est un entier k.
Le contenu de la pile (sauf le T initial) reste inchangé.

LA PILE OPERATIONNELLE: Exercices.

Exercices 8,9,10: Manipulations de blocs sur la pile.

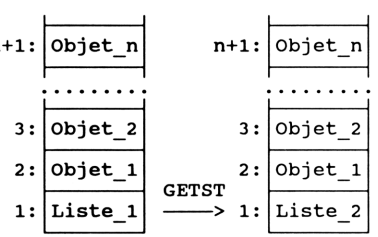
Notations pour les exercices 8,9,10:
On appellera "bloc" un groupe d'objets successifs sur la pile.
Un bloc est caractérisé par un niveau minimum 'min' et par un niveau maximum 'max'.
Si max<min, on considère que le bloc est vide.
Sa hauteur est max-min+1 (c'est le nombre d'objets dont il est constitué).
Si min < 1, on considère que min = 1.
Si max > DEPTH (hauteur de la pile), on prend max = DEPTH.
Enfin, si max ou min ne sont pas entiers, on les arrondit à l'entier le plus proche.

Exercice 8: (***) Descendre un bloc au niveau 1 de la pile.
Le programme **'ROLLB'** (**ROLL** sur un bloc) doit ramener au niveau 1 le bloc délimité par le niveau minimum 'min' et le niveau maximum 'max'.
La syntaxe doit être: **min max ROLLB** (d'abord le niveau minimum, puis le niveau maximum).

Exercice 9: (***) Remonter un bloc dans la pile.
Le programme **'ROLDB'** (**ROLLD** sur un bloc) doit remonter au niveau n de la pile le bloc allant du niveau 1 au niveau h (h=hauteur du bloc).
La syntaxe doit être: **h n ROLDB** (d'abord la hauteur du bloc, puis le niveau où le monter dans la pile).

Exercice 10: (**) Copier un bloc au niveau 1 de la pile.
Le programme **'PICKB'** (**PICK** sur un bloc) doit recopier au niveau 1 le bloc délimité par le niveau minimum 'min' et le niveau maximum 'max'.
La syntaxe doit être: **min max PICKB** (d'abord le niveau minimum, puis le niveau maximum).
Le bloc initial doit rester sur la pile.

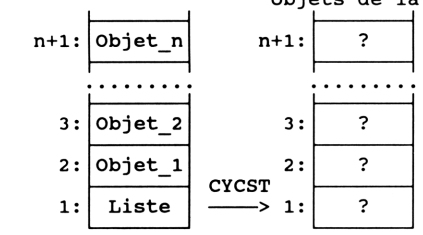
Exercice 11: (**) Extraire une liste d'objets de la pile.



Le programme **GETST** prend une liste d'entiers Liste_1 = {a b ... z}.
Il crée alors la Liste:
Liste_2 = {Oa, Ob, ... Oz}
où Oa, Ob, ... Oz, désignent les objets qui au départ se trouvent aux niveaux a,b,...,z.
Les niveaux a,b,...,z peuvent être donnés dans un ordre quelconque.
Un même niveau peut apparaître plusieurs fois.

Remarque: Dans le calcul du niveau sur la pile, la présence de Liste_1 dans le diagramme ci-dessus ne doit pas être prise en considération (Objet_1, par exemple, est considéré comme au niveau 1 de la pile).
Comme l'indique le diagramme ci-dessus, les objets restent sur la pile.

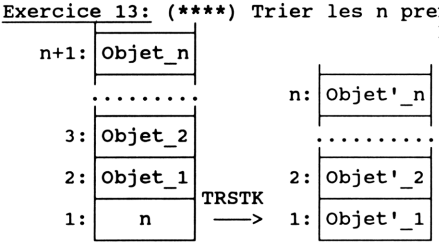
Exercice 12: (###) Permutation circulaire quelconque sur certains objets de la pile.



Le programme **CYCST** prend une liste d'entiers "Liste" = {a b ... z}. Il effectue alors une permutation circulaire sur les objets:
Oa, Ob, ... Oz,
qui se trouvent au départ aux niveaux a,b,...,z.
Ainsi Oa remplace Ob, qui est lui même envoyé là où se trouvait Oc etc..,et pour finir Oz est placé là où se trouvait Oa.

N.B: a,b,..z doivent être distincts 2 à 2, et correspondre à des niveaux occupés sur la pile.

Exercice 13: (****) Trier les n premiers objets de la pile.



Le programme **TRSTK** prend un entier n comme unique argument.
Le rôle de **TRSTK** est de trier les n premiers objets de la pile.
A la sortie, on doit avoir:
Objet'_n ≤ Objet'_(n-1)
≤ ...
≤ Objet'_2
≤ Objet'_1.
Il faut bien sûr que les différents objets considérés dans la pile initiale soient comparables au moyen de l'opérateur < (Ces objets doivent donc tous être des nombres réels, ou tous des entiers binaires, ou tous des chaînes de caractères).

Exercice 14: (****) Trier n objets suivant un critère quelconque.

Il s'agit ici, dans la limite des n premiers niveaux, de trier les objets de la pile, suivant les valeurs d'un critère de comparaison (ce qui généralise considérablement **'TRSTK'**).

Appellons O_k l'élément situé au départ au niveau k de la pile, et F(O_k) la valeur, pour cet objet, du critère utilisé.

Les objets doivent être triés suivant les valeurs décroissantes de ce critère.

Ainsi si O'_k est l'élément se retrouvant au niveau k de la pile après la procédure de tri, on doit avoir:

F(O'_n) ≤ F(O'_(n-1)) ≤ ... ≤ F(O'_2) ≤ F(O'_1).

Les valeurs du critère F doivent bien sûr pouvoir être comparées entre elles par <. Ces valeurs seront donc toutes des réels, ou toutes des chaînes, ou toutes des entiers binaires.

En général, le critère F sera à valeurs réelles.

On pourra ainsi comparer des chaînes par leur longeur,
des vecteurs par leur norme,
des réels par la valeur de leur sinus,
etc.....

Si on appelle **'TRCRT'** le programme, la syntaxe sera:

n «...» TRCRT

où n désigne jusqu'à quel niveau de la pile il faut trier, et où «...» désigne un programme, permettant de calculer la valeur du critère de comparaison pour l'objet situé au niveau 1.

NOMBRES REELS OU COMPLEXES

Dans ce chapitre, nous allons nous intéresser successivement:

* A la façon dont les nombres peuvent être entrés au clavier , et de quelle manière ils sont ou peuvent être affichés.
* Aux instructions arithmétiques et aux fonctions mathématiques qui agissent sur les nombres réels ou complexes.
* Aux prolongements complexes que permet la HP48SX pour certaines fonctions considérées habituellement comme des fonctions réelles.

▌ SAISIE ET AFFICHAGE DES NOMBRES REELS:

Il y a trois façons de considérer un nombre réel (et par conséquent un nombre complexe).
* Tel qu'il est entré au clavier par l'utilisateur.
* Tel qu'il est représenté en mémoire.
* Tel qu'il est affiché.

On verra que ces différentes manières de considérer un même nombre réel ne doivent pas être confondues.

* ENTRER UN NOMBRE REEL AU CLAVIER:

On doit distinguer deux cas selon que l'on veut utiliser ou non une notation "scientifique" (c.a.d. avec utilisation d'une puissance de 10) ou non.

Sans utilisation d'exposant:

On utilise (ou on peut utiliser) les touches suivantes:

* Chiffres de 0 à 9. On peut en entrer un nombre quelconque mais seuls les 12 premiers chiffres significatifs sont retenus (avec arrondi éventuel au douzième chiffre significatif) .
  Les zéros de tête ne sont pas significatifs.

* Séparateur décimal (le point . ou la virgule ,): le choix du séparateur décimal s'effectue dans le menu **MODES**. Selon que l'intitulé **FM,** est suivi ou non d'un petit carré ▪, le séparateur décimal est la virgule ou le point. L'autre de ces deux caractères est alors utilisé pour séparer des objets différents sur la ligne de commande. Le séparateur décimal peut être placé à un endroit quelconque dans la suite de chiffres, y compris en tête.
  Dans toute la suite , on utilisera le point comme séparateur décimal.

* Touche **+/-** de changement de signe: l'appui sur cette touche provoque un changement de signe. Il faut avoir entré au moins un chiffre pour pouvoir l'utiliser.

NOMBRES REELS OU COMPLEXES.

Avec utilisation d'exposant:

On souhaite donc entrer un nombre sous la forme **m*10^n**, où **m** est un réel, et **n** un entier (de signe quelconque).
On entre d'abord le réel **m** (ce qui précède s'applique).
On presse ensuite la touche **EEX**, ce qui a pour effet d'ajouter la lettre **E** à la fin de la représentation actuelle du nombre en ligne de commande.
On entre ensuite l'exposant, qui doit être un entier compris entre **-499** et **499** (inclus). Le signe de cet exposant (obligatoire seulement si cet exposant est négatif), peut être créé ou modifié au moyen de la touche **+/-** (et ce dès lors que l'on a appuyé sur la touche **EEX**).

Exemple: voici plusieurs manières d'entrer le même nombre au clavier, à savoir l'entier 123456789.
123456789 **ENTER** (ou 000000123456789 **ENTER**)
.123456789E9 **ENTER** (ou 0.123456789E9 **ENTER**).
1234.56789E5 **ENTER**
123456789000000E-6 **ENTER**
123456789.0001 **ENTER** (le 1 final n'est pas pris en compte car il est le 13ème chiffre significatif).
Par contre 123456789.0005 serait arrondi et mémorisé sous la forme 123456789.001

Remarque : On peut entrer une puissance de 10 en tapant directement la touche **EEX**. Celà a pour effet d'écrire **1E** en ligne de commande.
Faites par exemple **EEX +/- 8 ENTER** pour entrer 1E-8= .00000001

* **REPRESENTATION DES NOMBRES REELS EN MEMOIRE:**

Dans la mémoire de la HP48SX, un nombre réel x est toujours codé par la donnée d'une mantisse m comprise entre 1 et 10 ($1 \leq m < 10$) (12 chiffres significatifs) et d'un exposant n (compris entre -499 et 499).
On a ainsi **x=m*10^n**.
La mantisse m et l'exposant n sont représentés en "binaire codé décimal":
On sait qu'un quartet (4 bits) peut représenter les entiers de 1 à 15. Il peut donc représenter les entiers de 0 à 9.

b3	b2	b1	b0

Dans la représentation ci-contre, b0,b1,b2,b3 représentent les 4 bits d'un quartet. Leur valeur est 0 ou 1, et le "poids" du bit "bk" est 2^k.

Par exemple, le quartet

1	0	0	1

représente le chiffre:
1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 = 9.

La mantisse m est représentée sur 13 quartets:

qS	q0	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11

Les quartets q0 à q11 contiennent les chiffres significatifs, et le quartet qS contient le signe du nombre (qS=0 si le nombre est positif, qS=15=1111b si le nombre est négatif).
Cette mantisse est la représentation du réel:
m=± (q11).(q10)(q9)(q8)(q7)(q6)(q5)(q4)(q3)(q2)(q1)(q0).

L'exposant n est représenté sur 3 quartets:

qS	q1	q0

Le bit de poids fort du quartet qS représente le signe de l'exposant (bit à 0 si exposant positif et à 1 si exposant négatif). qS peut coder des entiers de -4 à 4, et q1, q0 des entiers de 0 à 9.
On comprend dans ces conditions que les exposants doivent être compris entre -499 et 499.

Les limites de la représentation en mémoire:

Ce qui précède montre que la représentation d'un nombre réel en mémoire occupe 16 quartets, soit 8 octets, et que les nombres extrêmes représentables sont (en valeur absolue):
9.99999999999E499 et 1.00000000000E-499.

Tout calcul où l'un au moins des nombres en présence n'est pas, en valeur absolue, compris entre ces deux extrêmes peut conduire à une erreur (**Overflow** ou **Underflow:** pour plus de détails voir plus loin dans ce chapitre).

* AFFICHAGE DES NOMBRES REELS A L'ECRAN:

La HP48SX possède plusieurs modes d'affichage des nombres réels.

Ces modes d'affichage s'appliquent à tout nombre réel visible à l'écran (évidemment), qu'un tel nombre occupe à lui seul un niveau de la pile ou qu'il apparaisse à l'intérieur d'un objet quelconque (sauf chaînes de caractères, et dans certains cas les expressions algébriques).

Le changement de mode d'affichage, même s'il a pour conséquence de ne montrer qu'une partie des chiffres significatifs d'un nombre réel **x**, ne modifie pas la précision avec lequel ce nombre est conservé en mémoire.

La meilleure preuve en est que lorsque l'on veut éditer (modifier) par **EDIT** ou **VISIT** un objet contenant un nombre réel, celui-ci est pendant le temps de l'édition en ligne de commande, représenté en mode "standard", c'est-à-dire avec tous ses chiffres significatifs.

Cependant: l'instruction **→STR** (transformation d'un objet en chaîne de caractères) prend en compte l'aspect "affiché" d'un nombre réel. L'instruction inverse **STR→** (évaluation d'une chaîne de caractères) ne renvoie alors que le réel qui correspond exactement à cette image; de cette double instruction il peut donc résulter une perte de chiffres significatifs pour le réel considéré.

Les différents modes d'affichage: (**STD, FIX, SCI, ENG**)

Ils sont accessibles dans le menu **MODES**, mais constituent des instructions utilisables dans un programme ou en ligne de commande.
L'examen du menu **MODES** permet de savoir quel est le mode actif (c'est celui dont l'intitulé est suivi d'un petit carré ▪).
Une pression sur une des quatre touches de menu (après avoir entré l'argument nécessaire pour les modes **FIX, SCI, ENG**) suffit à modifier le mode d'affichage.

STD **Mode d'affichage standard.**
C'est le mode par défaut (après une réinitialisation).
Les réels sont affichés avec toute leur précision:

Si le réel **x** est compris (en valeur absolue) entre .000000000001 et 999999999999, il est affiché (précédé du signe - s'il est négatif) avec tous ses chiffres significatifs (12 au plus, les zéros précédant le premier chiffre non nul ou suivant le dernier chiffre non nul ne sont pas significatifs et pas affichés), avec un séparateur décimal s'il n'est pas entier.

Dans le cas contraire, **x** est affiché en format scientifique:
x = ±(m)E(n), où:
(m) est la mantisse (1≤m<10): m est représentée sur au plus 12 chiffres, le premier étant compris entre 1 et 9 et précédant immédiatement le séparateur décimal (présent même si la mantisse est entière).
(n) est un exposant entier (-499≤ n ≤-12) ou (12≤ n ≤499).

Exemple: Voyons comment sont affichés, en mode standard différents réels **x** (Colonne de gauche = forme entrée en ligne de commande; Colonne de droite = forme affichée).

Entré	Affiché	Entré	Affiché
999999999999	999999999999	1.E11	100000000000
1000000000000	1.E12	1.E12	1.E12
.000000000001	.000000000001	12.34E10	123400000000
.0000000000001	1.E-13	12.34E11	1.234E12

FIX **Nombre fixe de décimales.**
La syntaxe est **n FIX**, où n est un entier entre 0 et 11 (après arrondi éventuel à l'entier le plus proche dans [0,11]).
Ce format permet d'afficher les réels avec une précision de n chiffres après la virgule (le n-ème étant calculé par arrondi, en fonction de la valeur conservée en mémoire).

Soit **x** un réel. On suppose que l'on est en format: **n FIX**.
Soit **x'** la valeur arrondie de **x**, à 10^(-n) près.

Si **x'** est nul alors **x** est affiché en notation scientifique **x = ±(m)E(k)**, où:
* la mantisse **m** vérifie 0 < **m** < 1 .Le séparateur décimal est précédé du chiffre 0,et suivi d'une partie décimale de n chiffres (y compris les 0 terminaux s'il y en a). Les chiffres de la partie entière de la mantisse sont séparés par tranches de trois chiffres (par le séparateur "virgule" si le "point" est le séparateur décimal, et vice-versa).
* l'exposant **k** est négatif (et k<-n).

Si **x'** n'est pas nul, il est affiché à la place de **x**:
Si possible **x'** est affiché sans exposant (en n'affichant pas les 0 terminant la partie décimale si nécessaire pour rester dans la limite de 12 chiffres affichables).
Si la notation scientifique est nécessaire,la mantisse est arrondie au n-ème chiffre après la virgule, des 0 étant éventuellement ajoutés en fin de partie décimale).

Exemples:

* Considérons le nombre **x**=.000456789012345
En mode standard, il est affiché 4.56789012345E-4
Voici comment il est affiché en mode fixe:

mode	affichage	mode	affichage
0 FIX	5.E-4	4 FIX	0.0005
1 FIX	4.6E-4	5 FIX	0.00046
3 FIX	4.568E-4	11 FIX	0.00045678901

* 123456789012 (affiché tel quel en mode standard) est affiché 123,456,789,012. (noter le séparateur décimal à la fin, et la virgule séparant les tranches de trois chiffres dans la partie entière) dans tous les modes fixes.

* 123.456 (affiché tel quel en mode standard) est affiché:
123. en mode 0 FIX, 123.46 en mode 2 FIX,
123.456 en mode 3 FIX, 123.4560 en mode 4 FIX,
123.456000000 dans les modes 9 FIX, 10 FIX, 11 FIX.

SCI **Mode scientifique.**

La syntaxe est **n SCI**, où **n** est un entier compris entre 0 et 11 (après arrondi éventuel à l'entier le plus proche dans [0,11]).

Les réels sont affichés en notation scientifique **x = ±(m)E(k)**
(**(m)**=mantisse et **(k)**=exposant) avec:

* 1 ≤ m < 10 (le chiffre précédant le séparateur décimal est donc un entier allant de 1 à 9).
Exception: pour le réel nul, m=0.

* m est composée de n+1 chiffres (donc 1 chiffre avant le séparateur décimal et n après), obtenus au besoin en complétant par des zéros terminaux.
le (n+1)ème chiffre est obtenu par arrondi.

* l'exposant k est un entier compris entre -499 et 499.

Exemple:
Considérons le nombre **x**=.000456789012345
En mode standard, il est affiché 4.56789012345E-4
Voici comment il est affiché en mode **SCI**:

mode	affichage	mode	affichage
0 SCI	5.E-4	4 SCI	4.5679E-4
1 SCI	4.6E-4	5 SCI	4.56789E-4
3 SCI	4.568E-4	11 SCI	4.56789012345E-4

Ce réel x a déjà été utilisé pour illustrer le mode **FIX**.
On constate que le mode **SCI** et le mode **FIX** peuvent parfois donner le même affichage, mais que la différence essentielle vient du fait que le mode **SCI** force toujours l'affichage en notation scientifique (mantisse et exposant).

ENG **Mode ingénieur.**

La syntaxe est **n ENG** où **n** est un entier compris entre 0 et 11 (après arrondi éventuel à l'entier le plus proche dans [0,11]).

Les réels sont affichés en notation scientifique **x = ±(m)E(k)**, (**(m)**=mantisse et **(k)**=exposant) avec:

* $1 \leq m < 1000$ (le séparateur décimal est donc précédé d'un entier allant de 1 à 999).
  Exception: pour le réel nul, m=0.

* m est composée de n+1 chiffres, obtenus au besoin en complétant par des zéros terminaux.
  Le (n+1)ème chiffre est obtenu par arrondi.

* l'exposant k est un multiple de 3:
  Si k' est l'exposant qui apparaîtrait dans l'écriture du réel en mode **SCI**, on peut dire qu'ici l'exposant k est le plus grand multiple de 3 inférieur ou égal à k'.
  Attention: si k'=-499, ce calcul conduirait à k=-501.
  Le format **ENG** est alors remplacé par le format **SCI**.

Exemples:

Considérons le nombre **x**=.000456789012345
En mode standard, il est affiché 4.56789012345E-4
Voici comment il est affiché en mode **ENG**:

mode	affichage	mode	affichage
0 ENG	500.E-6	4 ENG	456.79E-6
1 ENG	460.E-6	5 ENG	456.789E-6
3 ENG	456.8E-6	11 ENG	456.789012345E-6

En mode **4 ENG**:
* 1 est affiché 1.0000E0 ; 10 est affiché 10.000E0
* 100 est affiché 100.00E0 ; 1000 est affiché 1.0000E3
* 1E-498 est affiché 1.0000E-498 (498 est multiple de 3)
* 1E-499 est affiché 1.0000E-499 (notation SCI !)
* 1E498 est affiché 1.0000E498
* 1E499 est affiché 10.000E498

etc...

Attention:

Le mode **STD** ne nécessite pas de paramètre.
Par contre les modes **FIX**, **SCI**, **ENG** nécessitent un paramètre.
On prendra donc garde à ne pas appeler l'un de ces 3 modes sans avoir précisé ce paramètre (sans quoi l'objet qui est au niveau 1, si c'est un réel, sera considéré comme le paramètre par la HP48SX: vous risquez donc de perdre le réel situé au niveau 1 -encore qu'il soit récupérable par **LASTSTACK** ou **LASTARG** - mais en plus vous devez corriger l'effet de cette instruction ratée.

SAISIE ET AFFICHAGE DES NOMBRES COMPLEXES:

D'un point de vue mathématique, un nombre complexe peut être écrit:

En notation rectangulaire (ou algébrique, ou cartésienne):
Z = x + i*y, où x et y sont deux nombres réels:
x est la partie réelle de Z.
y est la partie imaginaire de Z.

En notation polaire (ou trigonométrique):
Z = ⟦ R, θ ⟧, où R et θ sont deux nombres réels, R≥0.
R est le module de Z.
θ est un argument de Z.

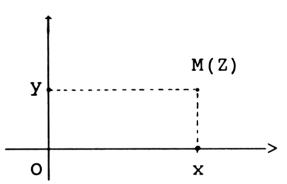
Le rapport entre les deux notations est donné par l'égalité:
⟦ R, θ ⟧ = R*(cos(θ) + i*sin(θ)), c'est-à-dire:

x = R*cos(θ)	ou encore
y = R*sin(θ)	

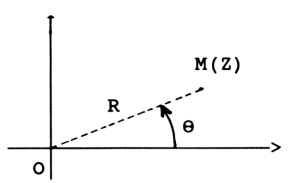
R = √(x²+y²)
cos(θ)=x/R, sin(θ)=y/R.

On remarque que les parties réelles x et y sont déterminées de façon unique par la donnée du module R et d'un argument θ.
De même le module de Z est déterminé de façon unique par la donnée de x et y, alors que θ est défini à un multiple de 2π près (si les angles sont exprimés en radians).

Le nombre complexe Z correspond de façon unique à un point M du plan.
La partie réelle et la partie imaginaire de Z sont alors l'abscisse et l'ordonnée de M.
Le module R de Z est la distance de M à l'origine.
Un argument θ de Z est un angle polaire de M.



Notation rectangulaire



Notation polaire

Pour la HP48SX, les complexes Z peuvent être entrés au clavier et/ou affichés en mode rectangulaire, ou en mode polaire.

* Saisie d'un nombre complexe sous forme rectangulaire:

Sous forme rectangulaire, le nombre complexe Z = x+i*y est représenté (x, y).
On entre donc successivement:
* le caractère ().
* la partie réelle x.
* un séparateur (la virgule en général, ou un espace, ou un saut de ligne)
* le réel y.

On peut aussi utiliser la constante i (voir plus loin).

NOMBRES REELS OU COMPLEXES.

*** Saisie d'un nombre complexe sous forme polaire:**

Sous forme polaire, un nombre complexe est représenté (R, ∠ θ).
On entre donc successivement:
* le caractère ().
* le module R.
* la virgule (touche <=┐•), facultative sauf dans les expressions algébriques.
* le séparateur ∠.
* l'argument θ.

*** Choix du mode d'affichage:**

Que l'on ait entré un nombre complexe sous forme rectangulaire ou sous forme polaire, il sera affiché en fonction du mode en cours sur la HP48SX.

- La touche **POLAR** permet d'alterner le mode rectangulaire (pas d'indicateur affiché), et le mode polaire (indicateur **R∠Z** affiché). Il se peut que la touche **POLAR** alterne entre le mode rectangulaire, et le mode sphérique (indicateur **R∠∠** affiché). Pour les nombres complexes, mode polaire et mode sphérique sont équivalents.
- Une autre méthode pour définir le mode d'affichage des nombres complexes, est d'utiliser les entrées suivantes du menu **MODES**, ou du menu **MTH VECTR**.
  XYZ: sélectionne le mode rectangulaire.
  R∠Z: sélectionne le mode polaire.
  R∠∠: sélectionne le mode sphérique.
  Le petit carré ▪ après l'un des trois labels désigne le mode en cours.
- Enfin, et c'est la seule façon de le faire de façon programmable, on peut sélectionner le mode d'affichage des nombres complexes de la manière suivante:
  16 +/- CF (désarmer indicateur -16): mode rectangulaire.
  16 +/- SF (armer l'indicateur -16):
  mode polaire si l'indicateur -15 est désarmé, et mode sphérique sinon (pour les complexes, c'est pareil).

*** Choix du mode de mesure des angles:**

Dès lors qu'on peut entrer, ou afficher, des nombres complexes sous forme polaire, le mode de mesure des angles est important.

On a le choix entre les modes: radians, degrés, grades.
Le choix se fait dans le menu **MODES** (page 3):
DEG : sélectionne le mode degrés.
RAD : sélectionne le mode radians.
GRAD: sélectionne le mode grades.
DEG, RAD, GRAD sont des instructions de la HP48SX et peuvent être utilisées dans un programme et en ligne de commande.

La touche **RAD** permet également d'alterner le mode radians avec le mode degrés ou le mode grades (suivant lequel de ces deux modes a été le dernier sélectionné par l'un des moyens précédents).

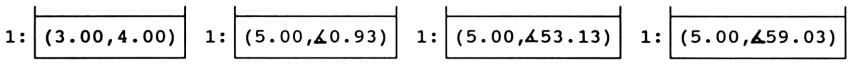
Le mode actif est rappelé par un témoin en ligne d'état:
Témoin **RAD** pour le mode radians.
Témoin **GRAD** pour le mode grades.
Pas de témoin affiché en mode degrés.

* Mode d'affichage des composantes d'un nombre complexe:

Ce qui a été dit sur les modes d'affichage des nombres réels s'applique ici aux deux composantes de chaque nombre complexe. Enfin, un nombre complexe occupe 16 octets en mémoire, et chacune des composantes d'un nombre complexe doit être, en valeur absolue, comprise entre 1.00000000000E-499 et 99999999999E499.

* Exemples:

▪On entre au clavier le nombre complexe 3+4i.
La séquence de touches est: **() 3 SPC 4 ENTER.**
On obtient les posssibles affichages suivants, en supposant que le mode d'affichage des réels est **2 FIX.**



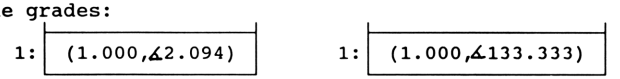
Mode rectangulaire | Mode polaire et radians | Mode polaire et degrés | Mode polaire et grades

En effet le module de Z=3+4i est |Z|=5 et un argument de Z est θ ≈ 0.93 radians ≈ 53.13 degrés ≈ 59.03 grades.

▪On entre au clavier le nombre complexe de module égal à 1 et d'argument égal à 120 degrés (c'est le nombre j).
On passe en mode degrés:
On entre au clavier: **() 1 ∠ 120 ENTER**
On obtient l'affichage (en supposant que le mode d'affichage des réels est le mode **3 FIX**)

Mode polaire (toujours en degrés) | Mode rectangulaire

Toujours avec le même nombre complexe au niveau 1, et en restant en mode polaire, on peut sélectionner le mode radians ou le mode grades:



Mode polaire, en radians | Mode polaire, en grades

* Remarques:

- Le choix du mode "degrés, radians ou grades" peut se faire en positionnant les indicateurs -17 et -18 (en tout cas le changement de mode par l'une des méthodes vues plus haut modifie ces indicateurs):
  17 +/- SF (indicateur -17 armé): mode radians.
  17 +/- CF (indicateur -17 désarmé), deux cas:
  18 +/- SF (indicateur -18 armé) : mode grades.
  18 +/- CF (indicateur -18 désarmé): mode degrés.

- Si l'affichage d'un nombre complexe peut varier suivant le mode en cours, en revanche il est toujours stocké en mémoire sous forme rectangulaire.

- Le changement de mode rectangulaire/polaire se traduit par une modification en conséquence des nombres complexes affichés, qu'ils soient isolés, ou apparaissent dans un programme, une expression, une liste, ou un tableau.
  Il en est de même, en mode polaire, si l'on change de mode radians/degrés/grades.

- Lors de la saisie d'un nombre complexe dans une expression algébrique, on doit séparer les deux composantes par le séparateur en cours, avec la touche <=┐• (la virgule).
  Si le séparateur décimal est le point: une virgule est affichée.
  Si le séparateur décimal est la virgule: un point-virgule est affiché.

- Tout nombre complexe non nul a une infinité de représentations possibles de la forme z = (R,∠θ).

  Néanmoins il n'en possède qu'une où:
  R > 0 (R = Module de Z = $\sqrt{(a^2+b^2)}$ si Z=a+ib).
  $-\pi < \theta \leq \pi$ (si on est en mode radians).
  -180 < θ ≤ 180 (si on est en mode degrés).
  -200 < θ ≤ 200 (si on est en mode grades).
  Les nombres complexes affichés en mode polaire sont toujours affichés en utilisant cette détermination de R et θ.

  Si on rentre au clavier un nombre complexe Z , sous forme polaire, en ne respectant pas ces conditions, le nombre Z est ramené à cette forme principale avant affichage.
  Si on rentre R=0 (dans l'entrée (R, θ), alors θ est automati-quement converti à zéro.

Les constantes MAXR,MINR,e,i,et :

* Les constantes MAXR et MINR:

MAXR et **MINR** désignent respectivement le plus grand et le plus petit nombre positif utilisables par la calculatrice.

Leurs valeurs sont donc:
MINR = 1.00000000000E-499 et **MAXR** = 99999999999E499.

L'apparition, au cours du calcul, d'un réel x dont la valeur absolue n'est pas comprise entre **MINR** et **MAXR** conduit à une situation d'exception désignée par un des noms suivants:
* **Negative Underflow** si -MINR < x < 0.
* **Positive Underflow** si 0 < x < MINR.
* **Negative Overflow** si x < -MAXR.
* **Positive Overflow** si MAXR < x.

La façon dont le calculateur gère ces situations d'exception dépend de l'état de certains indicateurs binaires:

Exceptions "Underflow":

Si l'indicateur -20 est désarmé (nul, ce qui est sa valeur par défaut), le résultat est considéré comme égal à 0.
Si l'indicateur -20 est armé (par **20 +/- SF**): il y a un message d'erreur et le programme en cours est interrompu.

Exceptions "Overflow":

Si l'indicateur -21 est désarmé (nul, ce qui est sa valeur par défaut), le résultat est considéré comme égal à **MAXR** ou à **-MAXR** (selon son signe).
Si l'indicateur -21 est armé (par **21 +/- SF**): il y a un message d'erreur et le programme en cours est interrompu.

Quelque soit l'état des indicateurs -20 et -21:
L'indicateur -23 est armé en cas de Negative Underflow.
L'indicateur -24 est armé en cas de Positive Underflow.
L'indicateur -25 est armé en cas d'Overflow.

Résultats infinis:
Les situations où le résultat est infini
(ex: **LN(0)**, **TAN(π/2)**, **x/0**, **(-1)!**, **ATANH(1)**):
- Donnent l'erreur **"Infinite Result"** si l'indicateur -22 est désarmé (nul, ce qui est sa valeur par défaut).
- Donnent le résultat **MAXR** ou **-MAXR** (suivant le signe) si l'indicateur -22 est armé (par **22 +/- SF**), et arment l'indicateur -26.

*** La constante e: e = 2.71828182846**

C'est la valeur de **EXP(1)** avec 12 chiffres significatifs.
Les expressions **'e^X'** et **'EXP(X)'** sont équivalentes: la seconde est cependant la plus précise.
Pour l'obtenir, on entrera la lettre **"e"** en minuscule.

*** La constante π: π = 3.14159265359**

C'est la valeur du nombre **π** avec 12 chiffres significatifs.
(Voir plus loin le mode d'évaluation symbolique de π).

*** La constante i: i = (0,1).**

Elle peut servir à entrer les nombres complexes: le nombre complexe **(x,y)** peut ainsi être entré sous la forme **'x+i*y'**.
Pour l'entrer au clavier, il faut utiliser la minuscule **"i"**.

*** Le mode d'évaluation symbolique:**

Deux indicateurs binaires interviennent dans la façon dont sont évaluées les constantes symboliques.
L'état de l'indicateur -3 peut être facilement défini par l'entrée **SYM** du menu **MODES**:
Une pression sur la touche correspondant à cette entrée inverse l'état de cet indicateur.
Si **SYM** est suivi d'un petit carré, l'indicateur -3 est à zéro (désarmé), et on est en mode "symbolique".
Sinon (pas de petit carré), l'indicateur -3 est à 1, (armé) et on est en mode numérique.
N.B: **SYM** n'est pas une instruction du langage de la HP48SX

Si l'indicateur -3 est armé (faire **3 +/- SF**):
On est en mode d'évaluation numérique des fonctions.
Les arguments symboliques d'une fonction sont remplacés par leur valeur.
Si par exemple le contenu de **'X'** est 1, l'éxécution de **EXP** avec l'argument **'X'** donnera le résultat 2.71828182846.
En particulier les 5 constantes symboliques sont remplacées par leur valeur.

Si l'indicateur -3 est désarmé (valeur par défaut: **3 +/- CF**):

On est en mode d'évaluation symbolique des fonctions.
Si par exemple le contenu de **'X'** est 1, l'éxécution de **EXP** avec l'argument **'X'** donnera l'expression **'EXP(X)'** (dont la valeur pourra être calculée par **EVAL** ou **→NUM**).
Pour ce qui est des cinq constantes symboliques précitées, tout dépend alors de l'état de l'indicateur -2.

S'il est désarmé (valeur par défaut, **2 +/- CF**),ils gardent leur forme symbolique si l'expression qui les contient est évaluée par **EVAL** (par contre **→NUM** force leur évaluation numérique).

S'il est armé (**2 +/- SF**) ils sont remplacés par leur valeur numérique.

Remarque:
Dans le cas où la constante π garde sa forme symbolique (indicateurs -2 et -3 désarmés) elle est reconnue (ainsi que $\pi/2$) par les fonctions **SIN COS** et **TAN** (à condition qu'on soit en mode radians).
Par exemple **'SIN(π)' EVAL** donne la valeur 0.

Exemples:

Le tableau suivant décrit quelques situations simples autour du nombre π. (-2) et (-3) désignent les contenus des indicateurs binaires -2 et -3.
On suppose que l'on est en mode radians.
Rappelons que: **x SF ENTER** met 1 dans l'indicateur N°x
x CF ENTER met 0 dans l'indicateur N°x

Action	Contexte	Résultat
π ENTER ou 'π' EVAL	(-3)=1 (-3)=0 et (-2)=1 (-3)=0 et (-2)=0	3.14159265359 3.14159265359 'π'
'π' →NUM	quelconque	3.14159265359
'π' SIN	(-3)=1 (-3)=0	-2.06761537357E-13 0
'SIN(π)' EVAL	(-3)=1 (-3)=0 et (-2)=1 (-3)=0 et (-2)=0	-2.06761537357E-13 -2.06761537357E-13 0
'SIN(π)' →NUM	quelconque	-2.06761537357E-13
'$\pi/3$' COS	(-3)=1 (-3)=0	.499999999997 'COS($\pi/3$)'
'COS($\pi/3$)' EVAL	(-3)=1 (-3)=0 et (-2)=1 (-3)=0 et (-2)=0	.499999999997 .499999999997 'COS($\pi/3$)'
'COS($\pi/3$)' →NUM	quelconque	.499999999997

INSTRUCTIONS SUR LES REELS ET COMPLEXES:

Dans la suite, les notations suivantes seront utilisées:

* **x, y** désignent des réels.

* **z, z'** désignent des nombres complexes: on pourra noter
  z = (x,y), ou **z = x+iy**,
  ou **z = (r,∠θ)** , le couple r, θ vérifiant:
  r > 0
  $-\pi < \theta \leq \pi$ (en mode radians) | Détermination
  -180 < θ ≤ 180 (en mode degrés) | principale de
  -200 < θ ≤ 200 (en mode grades) | l'argument de z.

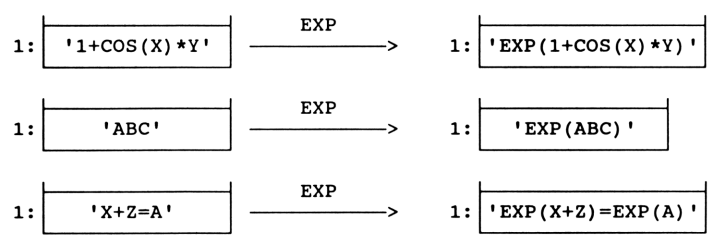
* **w, w'** désignent des nombres réels ou complexes.

Il faut savoir qu'en mémoire un nombre complexe est toujours stocké sous forme rectangulaire (x,y), même si à l'affichage le mode polaire est sélectionné.
Pour des calculs sur des nombres complexes affichés sous forme polaire, il peut s'ensuivre des erreurs d'arrondis:
Par exemple, en mode degrés:
le carré de (3,∠20) est, théoriquement, (9, 40), mais on obtient en mode standard: (9.00000000002, 40.0000000001).

D'un point de vue mathématique, un nombre tel que (x,0) est identifié au réel x (la partie imaginaire est nulle). Il en va tout autrement pour la HP48SX, qui considère les objets x et (x,0) comme différents.

N.B: La plupart des instructions qui sont décrites ci-après sont des fonctions, c'est-à-dire qu'elles peuvent s'appliquer à des des noms, des expressions algébriques, des équations.

Exemple: (pour la forme du résultat, on suppose que l'on est en mode d'évaluation symbolique des fonctions, c'est-à-dire que l'indicateur -3 est désarmé (valeur par défaut).



Dans les descriptions de fonctions ci-après, on n'a pas repris cet aspect du fonctionnemnt de la HP48SX, mais vous devez le garder présent à l'esprit.

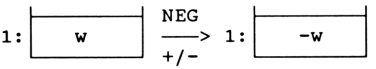
On s'est contenté de décrire comment agissent les fonctions sur des objets qui sont déjà des nombres réels ou complexes (ce qui finit tout de même par se produire quand on évalue des expressions algébriques).

NOMBRES REELS OU COMPLEXES.

*** OPERATIONS 'ARITHMETIQUES':**

Elles peuvent toutes être mises en oeuvre à partir d'une touche spécifique du clavier.

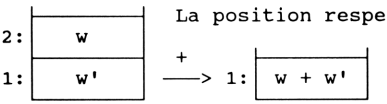
[+/-] **Changement de signe (ou symétrie par rapport à O).**
(touche **+/-** ou instruction **NEG**)



Si w=(x,y), -w=(-x,-y).

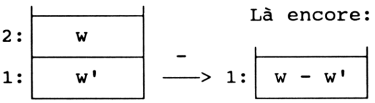
Si w=(r,∠θ), alors -w = (r,∠θ+π) (si on est en mode radians, sinon remplacer π par 180° ou par 200 gr), l'angle θ+π étant éventuellement ramené à sa valeur principale.

[+] **Addition:**



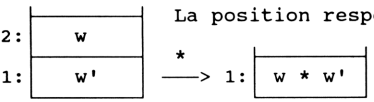
La position respective de w et w' est quelconque.
Rappelons que:
x + (x',y') = (x+x',y')
et:
(x,y) + (x',y') = (x+x',y+y').

[-] **Soustraction:**



Là encore: (x,y) - x' = (x-x',y)
x - (x',y') = (x-x',-y')
et:
(x,y) - (x',y') = (x-x',y-y').

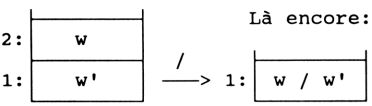
[*] **Produit:**



La position respective de w et w' est quelconque.
Rappelons que:
x * (x',y') = (xx',xy')
et que:
(x,y)*(x',y') = (xx'-yy',xy'+yx').

Sous forme polaire: (r,∠θ)*(r',∠θ') = (r+r',∠θ+θ')
(θ+θ' ramené éventuellement à sa détermination principale).

[/] **Quotient:**



Là encore: (x,y)/x' = (x/x',y/x')

$$x'/(x,y) = \left(\frac{x'x}{x^2+y^2}, \frac{-x'y}{x^2+y^2}\right)$$

$$(x',y')/(x,y) = \left(\frac{x'x+y'y}{x^2+y^2}, \frac{-x'y+y'x}{x^2+y^2}\right)$$

Sous forme polaire: (r,∠θ)/(r',∠θ') = (r/r',∠θ-θ')
(θ-θ' ramené éventuellement à sa détermination principale).

INV **Inverse:** (touche **1/x** ou instruction **INV**)

1: $\boxed{w}$ $\xrightarrow{\text{INV}}$ 1: $\boxed{1/w}$

Rappelons que si w=(x,y), alors:

$$\frac{1}{w} = \left(\frac{x}{x^2+y^2} , \frac{-y}{x^2+y^2} \right)$$

Sous forme polaire: L'inverse de $(r,\measuredangle\theta)$ est $(1/r,\measuredangle-\theta)$.

SQ **Carré:** (touche **x²**, ou instruction **SQ**)

1: $\boxed{w}$ $\xrightarrow{\text{SQ}}$ 1: $\boxed{w^2}$ Si $w = (x,y)$, $w^2 = (x^2-y^2,2xy)$.

Sous forme polaire: Le carré de $(r,\measuredangle\theta)$ est $(r^2,\measuredangle 2\theta)$.
(2θ ramené éventuellement à sa forme principale).

√ **Racine carrée:** (touche **√x**, ou instruction **√**)

1: $\boxed{w}$ $\xrightarrow{\sqrt{}}$ 1: $\boxed{\sqrt{w}}$

Si w est un réel positif ou nul , $\sqrt{w}$ est l'unique racine carrée positive ou nulle de w. Sinon, le résultat est complexe.

Tout nombre complexe non nul z possède exactement deux racines carrées complexes, qui sont opposées.

Si $z = (r,\measuredangle\theta)$, avec $r > 0$ et $-\pi < \theta \leq \pi$, alors la fonction √ de la HP48SX, appliquée à z, donne le nombre complexe:
$\sqrt{z} = (\sqrt{r},\measuredangle\theta/2) = \sqrt{r}\ (\cos(\theta/2), \sin(\theta/2))$.
L'angle polaire α de $\sqrt{z}$ vérifie donc : $-\pi/2 < \alpha \leq \pi/2$.

<u>Par exemple:</u>

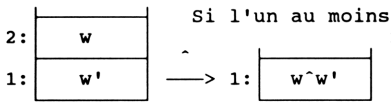
- $\sqrt{(-1)}$ est ici égal à (0,1): en effet on considère que -1 est le nombre complexe de module r=1, et d'argument $\theta=\pi$.
  Il est donc normal que $\sqrt{(-1)}=(\cos(\pi/2),\sin(\pi/2))=(0,1)$.
  Plus généralement, si x est un réel négatif, $\sqrt{(x)}=(0,\sqrt{|x|})$.
- $\sqrt{(-3,4)}$ est ici égal à (1,2): en effet les 2 racines carrées complexes de (-3,4) sont (1,2) et (-1,-2).On conserve ici celle dont l'argument est compris entre $-\pi/2$ et $\pi/2$, c'est à dire celle dont la partie réelle est positive.
- $\sqrt{0}$ donne 0, alors que $\sqrt{(0,0)}$ donne (0,0).

XROOT **Racine x-ème (x réel) d'un réel y.**
(touche $\sqrt[x]{y}$, ou instruction **XROOT**)

2: $\boxed{y}$
1: $\boxed{x}$ $\xrightarrow{\text{XROOT}}$ 1: $\boxed{\sqrt[x]{y}}$

Ici, x et y sont nécessairement des réels.
On résoud donc ici l'équation:
a^x = y,
où <u>a</u> est un réel inconnu.
Dans le cas où il y a deux réels opposés solutions, on obtient la solution ≥ 0.
<u>Par exemple:</u> La racine cubique de -8 est -2 (ici y=-8, x=3)
Il se peut qu'il n'y ait pas de solution (ex: y=-1, x=2), auquel cas on se heurte à une erreur "Undefined Result".

[^] **Exponentiation** (touche y^x ou instruction ^):



Si l'un au moins des deux arguments w, w' est un nombre complexe, alors le résultat est obtenu sous forme complexe.

Les cas usuels, pour lesquels le résultat est facilement inter-prétable, sont les suivants:

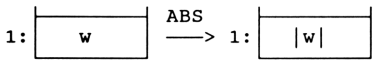
* w est un réel ou complexe quelconque et w' est un entier n strictement positif. Alors w^n est le produit de n exemplaires de w.
  Par ex: 7^5 = 7*7*7*7*7 = 16807.
  (2,1)^4 = (2,1)*(2,1)*(2,1)*(2,1) = (-7,24).
* w est un réel ou un complexe non nul et w' est un entier n strictement négatif. Alors w^n est égal à (1/w)^(-n), ce qui nous ramène au cas précédent.
  Par ex: 8^(-3) = (1/8)^3 = (.125)^3 = .001953125
  (2,1)^(-5) = (.4,-.2)^5 = (-.01216,-.01312)
* w' est nul: Alors x^0 = 1 pour tout réel x,
  z^0 = (1,0) pour tout complexe z.
* w est un réel x strictement positif, et w' est un réel x' quelconque. Alors w^w' = x^x' = EXP(x'*LN(x)).
* Dans tous les cas ci-dessus (l'exposant w' est un réel x') et si on écrit (r,∠θ) la forme polaire de w, alors:
  w^x' = (r,∠θ)^x' = (r^x',∠ x'θ).
  (L'angle x'θ ramené à sa valeur principale).

Dans les autres cas, le résultat est un nombre complexe, calculé par la formule w^w' = EXP(w'*LN(w)). Reportez vous à la description de **EXP** et **LN** dans le plan complexe.

* INSTRUCTIONS COMMUNES AUX REELS ET COMPLEXES:

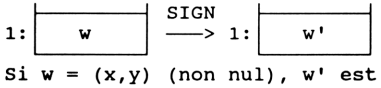
On les trouve dans la première page du menu **MTH PARTS**.

[ABS] **Module (valeur absolue) (ou distance à l'origine).**



Si x est réel |x|=MAX(x,-x).
Enfin, |(x,y)| = √(x²+y²).
et |(r,∠θ)| = r.

[SIGN] **Signe ou complexe unitaire radial:**



Si w est réel, w' représente le signe de w (-1 si w<0, 0 si w=0 et +1 si w>0).

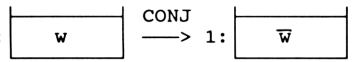
Si w = (x,y) (non nul), w' est le complexe de module 1 : w/|w|.

Ainsi $w' = \left(\frac{x}{x^2+y^2}, \frac{y}{x^2+y^2}\right)$. Si w=(0,0), alors w'=(0,0).

La formule w'=w/|w| est donc vraie pour tout w non nul.

Si w = (r,∠θ) alors w' = (1,∠θ).

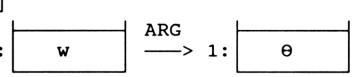
CONJ **Conjugué (ou symétrique par rapport à l'axe x'Ox).**



Si w est réel, $\overline{w} = w$.
Par contre si w=(x,y), $\overline{w}$=(x,-y)

Si w = (r,∠θ) alors $\overline{w}$ = (r,∠-θ).

ARG **Argument (ou angle polaire).**



L'instruction **ARG** donne l'angle polaire θ du nombre w.
La valeur de θ est exprimée en degrés, en grades ou en radians, selon le mode en vigueur à ce moment.

Si w=(r,∠θ), on obtient donc la deuxième composante θ.
En mode degrés: $-180 < \theta \leq 180$. En mode grades: $-200 < \theta \leq 200$
Si on est en mode radians: $-\pi < \theta \leq \pi$.
Si w est un réel x, il est considéré comme représentant (x,0).
En mode radians par exemple: Si w = 0, ou (0,0) alors θ=0.
Si w est un réel ≥ 0, alors θ = 0.
Si w est un réel < 0, alors θ = -π.

RE et **IM** **Partie réelle (ou abscisse) et partie imaginaire (ou ordonnée).**

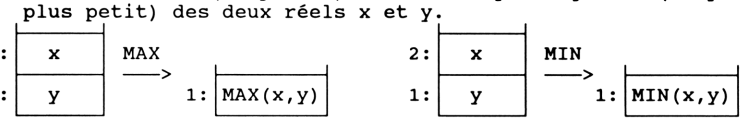
(Même si w est affiché sous forme polaire).

*** INSTRUCTIONS SPECIFIQUES AUX REELS:**

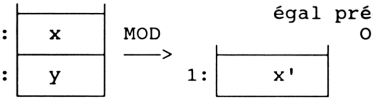
On les trouve dans les pages 2 et 3 du menu **MTH PARTS**.

MAX et **MIN** **Maximum ou minimum de deux réels.**

L'instruction **MAX** (resp. **MIN**) renvoie le plus grand (resp. le plus petit) des deux réels x et y.



MOD **Modulo (reste dans la division).**



L'instruction **MOD** calcule le réel x' = "x modulo y", égal précisément à x' = x - y*FLOOR(x/y).
On l'utilise fréquemment avec des arguments x et y entiers.
x' est alors égal au reste dans la division de x par y.

Exemples:

Supposons que n et k sont des variables contenant des entiers.
IF n 2 MOD THEN... signifie: "si n est impair, alors.."
IF n k MOD NOT THEN... signifie: "Si k divise n, alors.."
Dans une expression algébrique, la syntaxe est **'x MOD y'**.

NOMBRES REELS OU COMPLEXES.

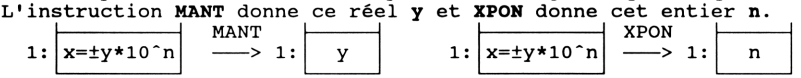
% , **%CH** , **%T** **Calculs de pourcentages.**

Niv.2	Niv.1	Instr.	Résultat	Signification
x	y	%	xy/100	x% de y (ou y% de x).
x	y	%CH	100(y-x)/x	% de x, pour passer de x à y
x	y	%T	100y/x	% de x représenté par y.

MANT et **XPON** **Mantisse ou exposant d'un nombre réel.**

Tout nombre réel x s'écrit de façon unique : x = ± y * 10^n, où
* la mantisse y est un réel (0 ≤ y < 10),
* l'exposant n est un entier positif ou négatif quelconque.

L'instruction **MANT** donne ce réel **y** et **XPON** donne cet entier **n**.



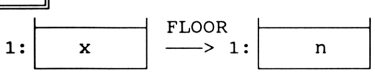
IP et **FP** **Partie entière ou partie fractionnaire.**

La représentation décimale d'un réel x s'écrit: x= ϵ n.d, où:
* ϵ vaut +1 ou -1: c'est le signe du réel x.
* n est un entier positif ou nul.
* "." est la séparateur décimal.
* La partie décimale d est un réel (0 ≤ d < 1).

Appliquées au réel x, **IP** donne l'entier **ϵn** (d'une façon imagée, **IP** donne "ce qu'il y a avant la virgule") et **FP** le réel **ϵd**. **FP** donne donc "ce qu'il y a après la virgule", avec le signe.

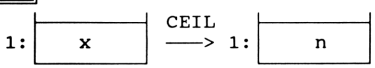
Attention:**IP** ne donne pas la partie entière au sens mathématique habituel du terme. C'est l'instruction **FLOOR** qui s'en charge.

FLOOR **Partie entière (plus grand entier inférieur ou égal).**



L'instruction **FLOOR**, appliquée au réel x, donne l'unique entier n tel que: n ≤ x < n+1.
n est donc le plus grand entier inférieur ou égal à x (c'est cet entier qu'on appelle habituellement la partie entière de x).
si x est entier, n et x sont égaux.
« **.5 + FLOOR** » arrondit à l'entier le plus proche.

CEIL **Plus petit entier supérieur ou égal.**



L'instruction CEIL , appliquée au réel x, donne l'unique entier n tel que: n-1 < x ≤ n.
n est donc le plus petit entier supérieur ou égal à x.
si x est entier, n et x sont égaux.

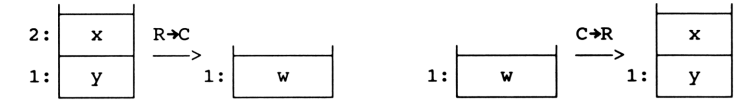
Tableau récapitulatif pour MANT, XPON, IP, FP, FLOOR, CEIL:

x	MANT	XPON	IP	FP	FLOOR	CEIL
-15.789	1.5789	1	-15	-.789	-16	-15
1578.9	1.5789	3	1578	.9	1578	1579
1.5789E15	1.5789	15	1.5789E15	0	1.5789E15	
1.5789E-15	1.5789	-15	0	1.5789E-15	0	1

* INSTRUCTIONS SPECIFIQUES AUX COMPLEXES:

R→C et **C→R** **Regroupement ou séparation de la partie réelle (ou abscisse) et de la partie imaginaire (ou ordonnée).**

(Dans le menu **PRG OBJ**, page 2)
On considère le nombre complexe w, de forme rectangulaire (x,y).
L'instruction **C→R** sépare w en x (au niveau 2) et y (au niveau 1)
L'instruction **R→C** regroupe le réel x (au niveau 2) et le réel y (au niveau 1) pour former le nombre complexe w.
Les schémas fonctionnels sont donc:

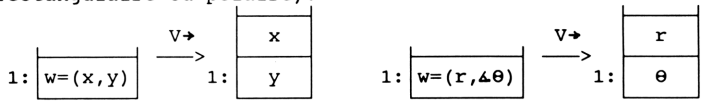


Attention : le format d'affichage des complexes (polaire ou rectangulaire) est quelconque. Que **w** soit affiché en mode rectangulaire ou polaire,**x** et **y** sont ici les deux composantes rectangulaires de **w** (c'est sous cette forme que **w** est stocké en mémoire).

N.B: l'instruction **OBJ→** (menu **PRG OBJ**, page 1) a le même effet que l'intruction **C→R**.

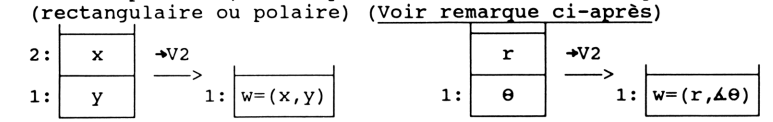
V→ et **→V2** **Séparation d'un nombre complexe en ses deux composantes, et vice-versa, en respectant le mode d'affichage en cours.**

L'instruction **V→** décompose un nombre complexe **w** en ses deux composantes, en respectant le mode d'affichage en cours (rectangulaire ou polaire).



En mode rectangulaire En mode polaire

L'instruction **→V2** recompose le nombre complexe **w** à partir de ses deux composantes, en respectant le mode d'affichage en cours (rectangulaire ou polaire) (Voir remarque ci-après)



En mode rectangulaire En mode polaire

En mode polaire, les composantes (radiale et polaire) sont ramenées, le cas échéant, à leur détermiantion principale.

Remarques:

Pour que **→V2** ait le comportement décrit ici, il faut que l'indicateur -19 soit armé (**19 +/- SF**) sinon l'objet formé ici est un vecteur ([x, y], ou [r∡θ])

La touche **2D** du clavier a le rôle de **V→** et de **→V2**.

En mode rectangulaire, les instructions **OBJ→**, **C→R** et **V→**, appliquées à un nombre complexe, sont équivalentes.

* INSTRUCTIONS DIVERSES:

→Q — Approximations rationnelles de nombres réels.

(cette instruction est directement accessible au clavier)

L'instruction **→Q** convertit un nombre réel x en son approximation fractionnaire 'a/b', où a et b sont entiers, la précision de cette approximation étant définie par le mode **FIX** en cours:

En mode **n FIX** (n décimales affichées) l'approximation s'effectue avec une précision relative de 10^(-n).

En mode **STD**, la précision relative est 10^(-11).

Exemple: 12754/38219 ≈ .333708364949

En mode **5 FIX**:

```
1: [ 0.33371 ]   ---->  1: [ '296/887' ]
```

Alors que: 296/887 ≈ 0.333709131905

En mode **STD**:

```
1: [ 0.333708364949 ]   ---->  1: [ '12754/38219' ]
```

Application aux complexes et aux expressions:

L'instruction **→Q** s'applique aux nombres complexes et aux expressions algébriques.

Un nombre complexe w=(x,y) (même s'il est affiché en polaire) sera converti en une expression algébrique 'a/b+c/d*i', où a,b,c,d sont entiers (a/b et c/d étant des approximations rationnelles de x et de y).

→Qπ — Approximations de la forme aπ/b (a,b entiers).

(Dans le menu **ALGEBRA**, page 2)

Cette approximation effectue, avec une précision relative qui dépend du nombre de décimales affichées (voir ci-dessus), et à partir d'un réel x:

* L'approximation c/d de x (c,d entiers).
* L'approximation a/b de x/π (a,b entiers).

Le résultat est l'expression:

'c/d' si d<b, ou 'a/b*π' si b<d.

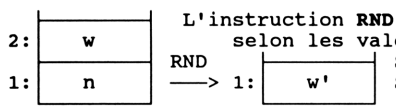
Le choix 'a/b*π' est donc effectué s'il est "plausible" que le nombre réel testé est le produit d'un réel par un rationnel.

Exemple:

```
1: [ 1.36941218 ]   ---->  1: [ '17/39*π' ]
```

RND **Arrondir un nombre réel ou complexe, ou les éléments d'un tableau.**

(dans le menu **MTH PARTS**, 4ème et dernière page)

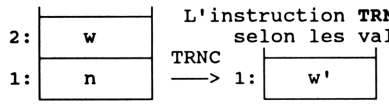


L'instruction **RND** arrondit le réel ou complexe w, selon les valeurs du paramètre (entier) n.
Si 0≤n≤11: arrondit à n décimales.
Si -11≤n≤-1: arrondit à n chiffres significatifs.
Si n=12: arrondit au nombre de décimales du mode FIX en cours.
RND peut s'appliquer à un tableau (au niveau 2): il s'applique alors à tous les éléments (réels ou complexes) de celui-ci.

Nombre w	Après 6 RND	Après 3 RND	Après -6 RND	Après -3 RND
1234.56789999	1234.5679	1234.568	1234.57	1230
1.357888-11	0	0	1.35789E-11	1.36E-11

TRNC **Tronquer un nombre réel ou complexe, ou les éléments d'un tableau.**

(dans le menu **MTH PARTS**, 4ème et dernière page)



L'instruction **TRNC** tronque le réel ou complexe w, selon les valeurs du paramètre (entier) n.
Si 0≤n≤11: tronque à n décimales.
Si -11≤n≤-1: tronque à n chiffres significatifs.
Si n=12: tronque au nombre de décimales du mode FIX en cours.
TRNC peut s'appliquer à un tableau (au niveau 2): il s'applique alors à tous les éléments (réels ou complexes) de celui-ci.

Nombre w	Après 6 TRNC	Après 3 TRNC	Après -6 TRNC	Après -3 TRNC
1234.56789999	1234.567899	1234.567	1234.56	1230
1.357888E-11	0	0	1.35788E-11	1.35E-11

MAXR et MINR **Réel positif maximum et réel positif minimum représentables en machine.**

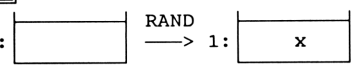
(dans le menu **MTH PARTS**, 4ème et dernière page)

La signification de **MAXR** et de **MINR** a été développée dans un paragraphe précédent.
Rappelons cependant que le contenu de **MAXR** est 9.99999999999E499 que celui de **MINR** est 1.E-499
et que l'appui sur la touche **MINR** (ou **MAXR**) amène au niveau 1:

* la valeur de **MINR** (ou de **MAXR**) si l'indicateur -3 est à 1 ou s'il est à 0 et qu'alors l'indicateur -2 est à 1.
* le nom **'MINR'** (ou **'MAXR'**) si les deux indicateurs -3 et -2 sont à 0 (désarmés).Un appui sur **→NUM** donne alors la valeur numérique correspondante.

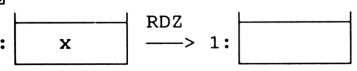
NOMBRES REELS OU COMPLEXES.

RAND **Réel aléatoire:** (Menu **MTH PROB**)



L'instruction **RAND** donne un réel (pseudo)aléatoire x ($0 \leq x < 1$). Pour obtenir un entier aléatoire compris entre 100 et 999 (inclus), on éxécutera la séquence:
« **RAND 900 * FLOOR 100 +** »

RDZ **Initialisation de la séquence de nombres aléatoires:** (Menu **MTH PROB**)

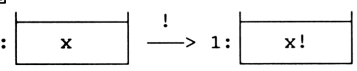


Le réel x au niveau 1 sert à générer les nombres aléatoires ultérieurs.

L'instruction **RDZ** est utile si on veut disposer plusieurs fois de suite de la même liste de nombres aléatoires (pour tester plusieurs procédures de tri sur une même liste d'entiers, par exemple).

Par exemple, après **1 RDZ**, les trois appels suivants de **RAND** donnent: .731362440213, .77207218067, et .989727311859

! **Fonction factorielle:** (Menu **MTH PROB**)



x! est la factorielle du réel x. 0! =1 et si x est un entier > 0, x! est le produit des entiers de 1 à x.

Par exemple: 10! = 10*9*8*7*6*5*4*3*2*1 = 3628800.
Le calcul de x! pour x≥254 conduit à un "Overflow".

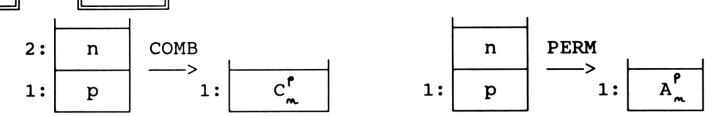
En fait x! = Γ(x+1), où Γ est l'application Gamma d'Euler.
Γ est définie sur le plan complexe privé des points (n,0) où n est un entier < 0.

Par exemple, pour tout réel x > -1: $x! = \int_0^{+\infty} t^x \, EXP(-t) \, dt.$

On peut donc calculer x! pour tout réel (x non entier négatif et x compris entre ≈-254 et ≈254).

On retrouve ainsi les résultats classiques:
(-.5)! = Γ(.5) = 1.77245385091 = $\sqrt{\pi}$
(.5)! = Γ(1.5)= .886226925453 = $(\sqrt{\pi})/2$ (à 2.10^(-12) près)

COMB et PERM **Combinaisons et arrangements.**



COMB et **PERM** donnent respectivement le nombre de combinaisons et le nombre d'arrangements de p éléments parmi n.

La syntaxe algébrique est **'COMB(n,p)'** et **'PERM(n,p)'**.
n et p doivent être entiers, avec $0 \leq p \leq n$.

'COMB(n,p)' est le nombre de parties à p éléments distincts dans un ensemble de n éléments (l'ordre dans lequel sont choisis ces éléments n'intervenant pas): COMB(n,p) = n!/(p!(n-p)!).

'PERM(n,p)' représente le nombre de choix ordonnés de p éléments distincts dans un ensemble à n éléments: PERM(n,p)= n!/(n-p)!.

*** APPLICATIONS LOGARITHMES ET EXPONENTIELLES:**

Ces fonctions s'appliquent à un argument réel ou complexe w, et renvoient un résultat réel ou complexe.
Elles sont toutes acccessibles au clavier.

Exception: les fonctions **LNP1** et **EXPM** prennent un argument réel et renvoient un résultat réel, et on les trouve en page 2 du menu **MTH HYP**.

Fonction	Commentaires
LOG	Logarithme décimal de w. LOG(w) = LN(w)/LN(10), avec LN(10) ≈ 2.30258509299
ALOG (10^x)	ALOG(w) = 10^w (fonction réciproque de w -> LOG(w)).
LN	Logarithme népérien de w. Si w est un nombre complexe et si <r,θ> est sa forme polaire, avec $-\pi < \theta \leq \pi$, alors: LN(w) = LN(<r,θ>) = LN(r) + iθ = (LN(r),θ). (La valeur de θ est obtenue en radians) Par exemple LN(i) = L((0,1)) = (LN(1),π/2) = (0,1.57079632679). De même LN(-2) = LN((2,π)) = (LN(2),π) = (.69314718056,3.14159265359).
EXP (e^x)	Fonction réciproque de LN: EXP(w) = e^w Si w est un nombre complexe et si (x,y) est sa forme rectangulaire, alors: EXP(w) = EXP((x,y)) = EXP(x)*(COS(y)+iSIN(y)) = (EXP(x)*COS(y) , EXP(x)*SIN(y)). (COS et SIN calculés en mode radians) La forme polaire de EXP(w) est donc <EXP(x),y>. En particulier, EXP((0,y))=EXP(i*y)=(COS(y),SIN(y)).
LNP1	LNP1(x) = LN(1+x). Ici x doit être un réel > -1. LNP1 est plus précise que LN pour calculer le loga-rithme népérien d'un réel très proche de 1. Supposons par exemple que x = 1E-n (n entier). Ln(1+x) et LNP1(x) donnent le même résultat si n≤11. Par contre, si n≥12, alors LN(1+x) donne 0 (car 1+x est représenté par 1 en mémoire) alors que LNP1(x) donne un résultat correct (LNP1(x)≈x).
EXPM	EXPM(x) = EXP(x)-1. Ici x doit être un réel. EXPM est plus précise que EXP pour calculer l'expo-nentielle de réels très proches de 0. Par exemple: EXP(3E-7) donne 1.0000003, et EXPM(3E-7) donne 3.00000045E-7 Donc: EXP(3E-7) ≈ 1.000000300000045 Bien sûr, si on ajoute 1 au résultat de EXPM, on retrouve exactement celui de EXP (car un réel ne peut être mémorisé qu'avec au plus 12 chiffres significatifs).

Exponentiation complexe:
La valeur de w^w', dans les cas non usuels, est calculée avec la formule w^w' = EXP(w' * LN(w)), où **LN** et **EXP** agissent comme indiqué ci-dessus si leurs arguments sont complexes.
Ceci vaut en particulier pour ALOG(w)= 10^w= EXP(w*LN(10)).

Exemples:

(-1)^(1/3) = EXP(1/3*LN(-1)) = EXP(1/3 * (0,π))
= EXP(i*π/3) = (COS(π/3) , SIN(π/3))
= (.500000000001 , .866025403784)

(1,1)^(2,-3) = EXP((2,-3) * LN((1,1))
= EXP((2,-3) * LN(<√2,π/4>)
= EXP((2,-3) * (LN(√2) , π/4)
= EXP((2,-3) * (0.3465... , 0.7853...))
= EXP((3.049... , 0.531...))
= EXP(3.049...)*(COS(0.531...),SIN(0.531...))
= (21.1014...)*(0.86226...,0.50646...))
= (18.1949946543 , 10.6870614993).

*** APPLICATIONS HYPERBOLIQUES:**

Elles sont regroupées dans le menu **MTH HYP** (page 1).
Elles s'appliquent à un argument réel ou complexe et renvoient un résultat réel ou complexe.

Fonction	Commentaires
SINH	Sinus hyperbolique: SINH(w)=(EXP(w)-EXP(-w))/2.
ASINH	Argument sinus hyperbolique (fonct. inverse de SINH) ASINH(w) = LN(w+√(w²+1)).
COSH	Cosinus hyperbolique: SINH(w)=(EXP(w)+EXP(-w))/2.
ACOSH	Argument cosinus hyperbol. (fonct. inverse de COSH) ACOSH(w) = LN(w+√(w²-1)).
TANH	Tangente hyperbolique: TANH(w) = SINH(w)/COSH(w). TANH(w) = (EXP(2*w)-1)/(EXP(2*w)+1) = (1-EXP(-2*w))/(1+EXP(-2*w)).
ATANH	Argument tangente hyperb. (fonct. inverse de TANH) ATANH(w) = (1/2)*LN((1+w)/(1-w)).

Profitons-en pour rappeler quelques formules de la trigonométrie hyperbolique: Pour tous w et w' (réels ou complexes),

COSH(w)+SINH(w) = EXP(w), COSH(w)-SINH(w) = EXP(-w)
COSH(w)²-SINH(w)²= 1, SINH(-w)=-SINH(w), COSH(-w)=COSH(w).
COSH(w+w') = COSH(w)COSH(w') + SINH(w)SINH(w').
SINH(w+w') = SINH(w)COSH(w') + COSH(w)SINH(w').
COSH(2*w) = 2*COSH(w)²-1 = 2*SINH(w)²+1.
SINH(2*w) = 2*SINH(w)*COSH(w)
TANH(w+w') = (TANH(w)+TAN(w'))/(1 + TANH(w)TANH(w')).
Dérivées: COSH'=SINH, SINH'=COSH, TANH'=1-TANH²=1/COSH²
Périodicité: COSH(w+2ikπ)=COSH(w), SINH(w+2ikπ)=SINH(w)
TANH(w+ikπ)=TANH(w).

* APPLICATIONS TRIGONOMETRIQUES CIRCULAIRES:

Elles s'appliquent à un argument réel ou complexe et renvoient un résultat réel ou complexe.
Elles sont accessibles par une touche du clavier.

Fonction	Commentaires
SIN	Sinus. Si w=(x,y): SIN(w) = (EXP(iw)-EXP(-iw))/(2i) = SIN(x)COSH(y) + iCOS(x)SINH(y)
ASIN	Arc sinus (fonction inverse de SIN) Si x est réel et appartient à [-1,1], ASIN(x) est l'angle θ tel que SIN(θ)=x, avec: ▪ $-\pi/2 \leq \theta \leq \pi/2$ si on est en mode radians. ▪ $-90 \leq \theta \leq 90$ si on est en mode degrés. ▪ $-100 \leq \theta \leq 100$ si on est en mode grades. Si w n'appartient pas à [-1,1], ASIN(w) = -i*LN(i*w+√(1-w²)).
COS	Cosinus. Si w=(x,y): COS(w) = (EXP(iw)+EXP(-iw))/2 = COS(x)COSH(y) - iSIN(x)SINH(y)
ACOS	Arc cosinus (fonct. inverse de COS) Si x est réel et appartient à [-1,1], ACOS(x) est l'angle θ tel que COS(θ)=x, avec: ▪ $0 \leq \theta \leq \pi$ si on est en mode radians. ▪ $0 \leq \theta \leq 180$ si on est en mode degrés. ▪ $0 \leq \theta \leq 200$ si on est en mode grades. Si w n'appartient pas à [-1,1], ACOS(w) = -i*LN(w+√(w²-1)).
TAN	Tangente. TAN (w) = SIN (w)/COS (w).
ATAN	Arc tangente (fonction inverse de TAN) Si x est réel ATAN(x) est l'angle θ tel que TAN(θ)=x avec: ▪ $-\pi/2 < \theta < \pi/2$ si on est en mode radians. ▪ $-90 < \theta < 90$ si on est en mode degrés. ▪ $-100 \leq \theta \leq 100$ si on est en mode grades. Si w n'appartient pas à [-1,1], ATAN(w) = (i/2)*LN((i+w)/(i-w)).

* Rappelons que le choix du mode actif peut se faire :
  - dans le menu **MODES** (le petit carré ▪ après l'un des trois intitulés **RAD** , **DEG**, **GRAD** désigne le mode choisi).
  - par éxécution des instructions **RAD**, **DEG**, ou **GRAD** (en ligne de commande ou dans un programme).
  - par modification des flags **-17** et **-18**.
  - par utilisation de la touche **RAD**.

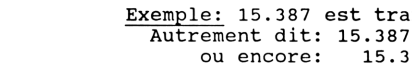
* Le mode actif intervient dans le calcul des dérivées.
  Si par exemple **'X'** est une variable formelle,
  - en mode degrés: **'SIN(X)' 'X' ∂** donne **'COS(X)*(π/180)'**.
  - en mode radians: **'SIN(X)' 'X' ∂** donne **'COS(X)'**.
  - en mode grades: **'SIN(X)' 'X' ∂** donne **'COS(X)*(π/200)'**.

* CALCULS EN MODE HMS (dans le menu TIME, page 3):

Le mode **HMS** est une façon d'interpréter l'écriture décimale d'un réel x désignant une mesure d'angle ou une mesure de temps.
Cette écriture doit être interprétée de la manière suivante:
x = H.MMSSdcm... où:
H = nombre d'unités (d'heures, de degrés).
MM = nombre de minutes (de temps ou d'angle).
SS = nombre de secondes (de temps ou d'angle).
dcm... = dixièmes, centièmes, millièmes... de secondes.
Le signe de x est quelconque.

En principe MM et SS sont compris entre 0 et 59 (du moins les instructions **→HMS**, **HMS+**, **HMS-** renvoient un tel résultat).
Par contre les arguments de **HMS→**, **HMS+** et **HMS-** peuvent très bien, et sans risque d'erreur, ne pas avoir cette propriété.
Par exemple: **1.98 3.67 HMS+** donne 6.45, ce qui signifie:
1°98' + 3.67' = 6°45'.

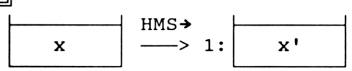
→HMS **Format décimal vers format HMS.**



→HMS convertit un réel x écrit sous forme décimale habituelle, en son écriture au format **HMS**.

Exemple: 15.387 est transformé en 15.23132
Autrement dit: 15.387° = 15° 23' 13.2"
ou encore: 15.387h = 15h 23mn 13.2 s

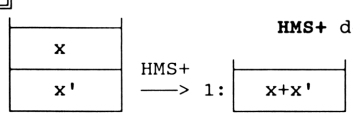
HMS→ **Format HMS vers format décimal.**



HMS→ convertit un réel x écrit au format **HMS** en son équivalent au format habituel.

Exemple: .4518 est transformé en .755
Autrement dit: 45' 18" = 0.755 °
ou encore: 45mn 18" = 0.755 h

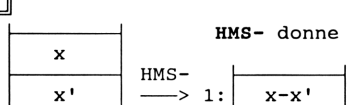
HMS+ **Addition en format HMS.**



HMS+ donne la somme x+x', sous le format **HMS**, des deux réels x et x', eux mêmes écrits au fromat **HMS**.

Exemple: 3.4825 1.3551 HMS+ donne 5.2416
Autrement dit: 3° 48' 25" + 1° 35' 51" = 5° 24' 16 "
ou encore: 3h 48mn 25s + 1h 35mn 51s = 5h 24mn 16s.

HMS- **Soustraction au format HMS.**



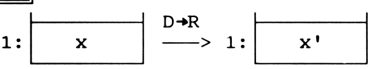
HMS- donne la différence x-x', sous le format **HMS**, des deux réels x et x', eux mêmes écrits au fromat **HMS**.

Exemple: 3.4825 1.3551 HMS- donne 2.1234
Autrement dit: 3° 48' 25" - 1° 35' 51" = 2° 12' 34 "
ou encore: 3h 48mn 25s - 1h 35mn 51s = 2h 12mn 34s.

▪ **CONVERSIONS DEGRES=>RADIANS OU RADIANS=>DEGRE:**

Les instructions **D→R** et **R→D** sont dans le menu **MTH VCTR** (page 2).

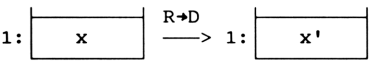
D→R **Conversion degrés=>radians.**



Autrement dit: x' = x*180/π.

L'instruction **D→R** transforme le réel x (supposé représenter un angle exprimé en degrés) en son équivalent en radians.

R→D **Conversion radians=>degrés.**



Autrement dit: x' = x*π/180.

L'instruction **R→D** transforme le réel x (supposé représenter un angle exprimé en radians) en son équivalent en degrés.

EXERCICES SUR LES NOMBRES REELS OU COMPLEXES:

La difficulté des exercices est mesurée par des symboles * ou # :

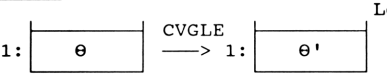
- * Très facile.
- ** Assez simple.
- *** Déjà un peu plus délicat.
- **** Plutôt difficile.
- ***** Difficile.

Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes qui précèdent dans cette liste d'exercices.

Exercice 1: (**) Doubler les instructions →HMS, HMS→, HMS+, HMS-.

Bien que celà ne revête qu'un aspect purement pédagogique, il vous est demandé d'écrire des programmes réalisant le même travail que chacune des instructions **→HMS**, **HMS→**, **HMS+**, **HMS-**.

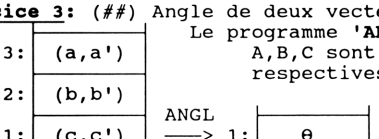
Exercice 2: (*) Convertir un angle en sa mesure principale.



Le programme '**CVGLE**' considère que le réel θ est la mesure d'un angle, et renvoie la mesure principale θ' de cet angle.

θ est interprété suivant le mode en cours (degrés,radians,grades) et: en mode degrés, -180 < θ' ≤ 180.
en mode radians, -π < θ' ≤ π.
en mode grades, -200 < θ' ≤ 200.

Exercice 3: (##) Angle de deux vecteurs.



Le programme '**ANGL**' calcule l'angle $(\vec{AB},\vec{AC})$, où A,B,C sont les points du plan de coordonnées respectives (a,a'), (b,b'), (c,c').

La valeur θ obtenue est la mesure principale de cet angle:
En mode **degrés**: -180 < θ ≤ 180
En mode radians: -π < θ ≤ π
En mode grades: -200 < θ ≤ 200

NOMBRES REELS OU COMPLEXES: Exercices.

Exercice 4: (*) Périmètre d'un triangle.

Le programme **'PERI'** calcule le périmètre du triangle ABC.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le périmètre au niveau 1.

Exercice 5:(**) Aire d'un triangle.

Le programme **'AIRE'** calcule l'aire intérieure au triangle ABC.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: l'aire au niveau 1.

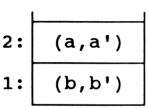
Exercice 5: (***) Projection d'un point sur une droite.

Le programme **'PROJ'** détermine la projection orthogonale H(h,h') du point A sur la droite portant les points B et C.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le nombre complexe (h,h') au niveau 1.

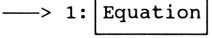
Exercice 6: (#) Symétrique d'un point par rapport à une droite.

Le programme **'SYM'** détermine le symétrique orthogonal D(d,d') du point A par rapport à la droite portant les points B et C.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le nombre complexe (d,d') au niveau 1.

Exercice 7: (***) Equation de la droite passant par 2 points donnés.



Le programme **'DRTE'** calcule l'équation de la droite qui passe par les points A(a,a') et B(b,b').



Le résultat est une expression du type 'a*X+b*Y=c', où a,b,c sont des réels (ce sont les coefficients de l'équation).

Exercice 8: (##) Perpendiculaire à une droite, passant par un point.

Le programme **'PERP'** donne l'équation de la perpendiculaire à la droite BC, et qui passe par A.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: Une équation du type 'a*X+b*Y=c'.

Exercice 9: (#) Médiatrice d'un segment.

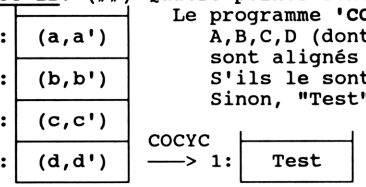
'MDTR' donne l'équation de la médiatrice du segment [A,B].
Le schéma fonctionnel est celui de l'exercice 7.

Exercice 10: (##) Bissectrice de deux demi-droites.

Le programme **'BISS'** donne l'équation de la bissectrice des deux demi-droites [A,B[et [A,C[.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: Une équation du type 'a*X+b*Y=c'.

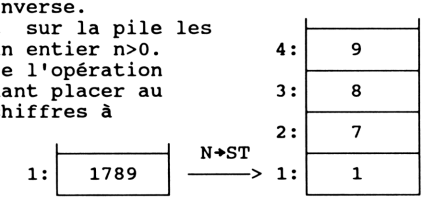
Exercice 11: (##) Trois points sont-ils alignés?
Le programme **'ALIG'** teste si les 3 points A,B,C sont alignés.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: L'entier 1 si A,B,C sont alignés,
et 0 s'ils ne le sont pas.
N.B: Il est demandé trois solutions distinctes.

Exercice 12: (##) Quatre points sont-ils alignés ou cocycliques?

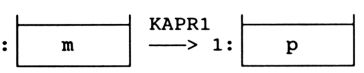


Le programme **'COCYC'** teste si les quatre points A,B,C,D (dont les coordonnées sont sur la pile) sont alignés ou cocycliques.
S'ils le sont, "Test" vaut 1.
Sinon, "Test" vaut 0.

Exercice 13: (*) Libérer les chiffres d'un entier sur la pile et opération inverse.
Le programme 'N→ST' place sur la pile les différents chiffres d'un entier n>0.
Le programme 'ST→N' réalise l'opération inverse. Il faut cependant placer au niveau 1 le nombre de chiffres à considérer.
Les programmes 'N→ST' et 'ST→N' seront utilisés dans l'exercice 14.

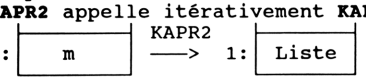


Exercice 14: (###) Algorithme de Kaprekar (première version).



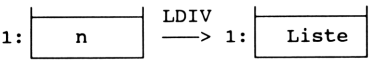
L'algorithme de Kaprekar part d'un entier m,calcule l'entier m' obtenu en écrivant les chiffres de m dans l'ordre décroissant (ex: 8917 => 9871),l'entier m" obtenu en écrivant ces chiffres dans l'ordre croissant, puis calcule l'entier p égal à m'-m".
Par exemple: si m = 8917, alors:
m' = 9871, m" = 1789, et p = 9871-1789 = 8082.
N.B: les chiffres de m ne doivent pas être tous égaux.
On peut vérifier qu'une application répétée de **KAPR1** en partant d'un entier à quatre chiffres finit par donner le résultat stationnaire 6174.
Par exemple: 1789 => 8082 => 8532 => 6174 => 6174 etc...

Exercice 15: (##) Algorithme de Kaprekar (deuxième version).
On veut généraliser l'observation faite pour les appels répétés de **KAPR1** (à partir d'un entier de 4 chiffres).
Il est assez clair que de tels appels répétés finissent toujours par nous faire obtenir un entier qui est déjà apparu dans les résultats précédents (d'où une périodicité).
L'objectif de **KAPR2** est d'étudier de telles périodicités:
On part d'un entier m.
KAPR2 appelle itérativement **KAPR1** et s'arrête quand 2 résultats identiques **'n'** ont été trouvés.
Il donne alors la liste de tous les résultats obtenus depuis m, jusqu'à la 2ème occurence de n.



NOMBRES REELS OU COMPLEXES: Exercices.

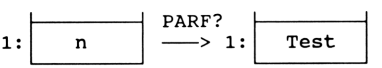
Exercice 16: (***) Liste de tous les diviseurs d'un entier positif.



Le programme **'LDIV'** donne la liste de tous les diviseurs de l'entier positif n.

Par exemple, si n=100 la liste est { 1 2 4 5 10 20 25 50 100 }.

Exercice 17: (#) Un entier positif est-il parfait?

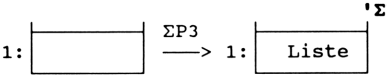


On dit qu'un entier positif n est parfait si n est égal à la somme de ses diviseurs d, 1≤d<n.

Par exemple, 28 est parfait car: 28 = 1 + 2 + 4 + 7 + 14.

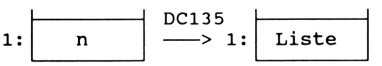
Le programme **'PARF?'** teste si l'entier n est parfait: S'il l'est "Test" vaut 1, sinon, "Test" vaut 0.

Exercice 18: (**) Nombres égaux à la somme des cubes de leurs trois chiffres.



'ΣP3' place sur la pile les entiers compris entre 100 et 999 et qui sont égaux à la somme des cubes de leurs trois chiffres.

Exercice 19: (**) Trouver toutes les solutions entières positives ou nulles de l'équation x+3y+5z=n.



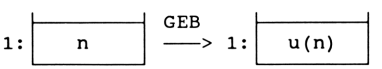
'DC135' donne la liste des décompositions possibles de l'entier n sous la forme:

n=x+3*y+5*z, où x,y,z sont des entiers positifs ou nuls.

A partir de 10 on doit ainsi trouver:

{ {0 0 2} {1 3 0} {2 1 1} {4 2 0} {5 0 1} {7 1 0} {10 0 0} }

Exercice 20: (***) n-ème terme d'une suite récurrente.

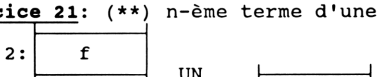


On considère la suite récurrente définie par u(1)=u(2)=1, et pour n≥3, par la relation:

u(n) = u(n-u(n-1)) + u(n-u(n-2)).

Le programme 'GEB' (du nom "Goëdel-Escher-Bach" de l'excellent ouvrage où cette suite est étudiée) doit calculer la valeur de u(n), où n est un entier.

Exercice 21: (**) n-ème terme d'une suite.

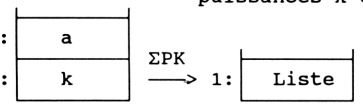


Le problème est ici de calculer le n-ème terme d'une suite u, défini par:

$$u(n) = \sqrt{f(1) + \sqrt{f(2) + \sqrt{\ldots\ldots + \sqrt{f(n-1) + \sqrt{f(n)}}}}}.$$

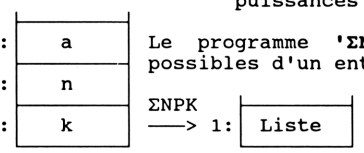
où "f" est un programme censé s'appliquer à un entier situé au niveau 1, et renvoyant un résultat numérique.

Exercice 22: (****) Décompositions d'un nombre en somme de deux puissances k-èmes.



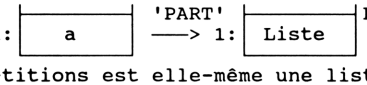
Le programme **'ΣPK'** cherche les décompositions possibles d'un entier a en somme de deux puissances k-èmes.
Les différentes solutions sont placées dans une liste au niveau 1 (cette liste est vide s'il n'y a aucune solution).
Exemple: avec a=87539319 et k=3, on doit trouver la liste
{ {436 167} {423 228} {414 255} }.
En effet: 87539319 = 436^3+167^3 = 423^3+228^3 = 414^3+255^3.

Exercice 23: (*****) Décompositions d'un entier en une somme de n puissances k-èmes.



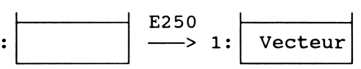
Le programme **'ΣNPK'** cherche les décompositions possibles d'un entier a comme somme de n puissances k-èmes.
Les différentes solutions sont placées dans une liste au niveau 1 de la pile.
(Cette liste est vide s'il n'y a pas de solution).
Exemple: avec a=100, n=4, k=2, on obtient la liste
{ {1 1 7 7} {1 3 3 9} {1 5 5 7} {2 4 4 8} {5 5 5 5} }
En effet, par exemple, 100 = $2^2+4^2+4^2+8^2$.

Exercice 24: (*****) Partitions d'un nombre entier.
On appelle partition d'un nombre entier a>0 toute écriture de a comme somme d'entiers > 0.



Le programme **'PART'** donne la liste de toutes les partitions possibles de l'entier n. Chacune de ces par-titions est elle-même une liste d'entiers croissants.
Exemple: avec n=5, on obtient la liste
{ {1 1 1 1 1} {1 1 1 2} {1 1 3} {1 2 2} {1 4} {2 3} {5} }

Exercice 25: (*****) Les 250 premières décimales de e.



Le programme **'E250'** place dans un vecteur les 250 premières décimales du nombre e=exp(1).
Le vecteur est formé de 28 termes qui contiennent chacun une tranche de 9 décimales (dans chaque tranche, le ou les zéros éventuels au début ne sont pas affichés mais ils existent...).
Le résultat s'écrit:
[718281828 459045235 360287471 ... 460668081]
Autrement dit: e = 2. 718 281 828 459 045 .... 460 668 0..
(les 2 derniers chiffres, 8 et 1, qui correspondent aux décimales N° 251 et 252, ne sont pas à prendre en considération).

LES LISTES

LA NOTION DE LISTE:

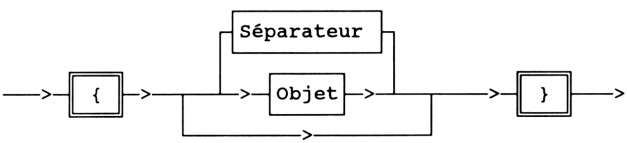
Une liste, dans le langage de la HP48SX, est, comme son nom l'indique, un objet constitué d'une succession, dans un ordre déterminé, d'objets de types quelconques.

Les listes peuvent contenir des objets hétéroclites. C'est d'ailleurs là que réside leur intérêt: on est ainsi en mesure de regrouper, dans une même entité, des objets réputés inconciliables.

Si un tableau est un groupement ordonné (en une ou deux dimensions) de nombres réels, ou bien de nombres complexes, une liste peut contenir simultanément des réels, des complexes, des tableaux, des expressions, des noms, etc...,et même des listes (en celà la notion de liste est récursive).

La syntaxe d'une liste est représentée par le diagramme:

LISTE:



Une liste débute donc par le caractère { et se termine par le caractère }.

La saisie débute par un appui sur la touche {}, qui affiche les deux caractères { et } (le curseur se trouvant alors en superposition avec le caractère } terminal).

On remarque que le mode de saisie automatiquement sélectionné est le mode "Programme" (indicateur **PRG** allumé).

Du point de vue de l'écho, en ligne de commande, des touches actionnées, tout se passe donc comme si on entrait un programme (d'ailleurs tout programme entre « et » peut être transformé en une liste, et réciproquement).

Les objets contenus dans une liste doivent être isolés les uns des autres par un séparateur (espace, virgule/point, ou retour à la ligne par <—┘).

A l'affichage, les différents objets sont séparés par un espace.

On appelle taille d'une liste le nombre d'objets qu'elle contient. Cette taille n'est limitée que par la quantité de mémoire disponible.

Exemples:

{ } est la liste vide (taille = 0)

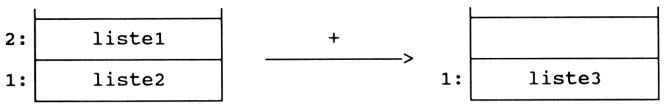
{ { } { } { } { } { } } est une liste de cinq listes vides.

{ 'A' { 1 23 } "Hello" } est une liste de 3 objets, qui sont le nom 'A', la liste { 1 23 } et la chaîne "Hello".

OPÉRATIONS SUR LES LISTES:

* LA CONCATÉNATION:

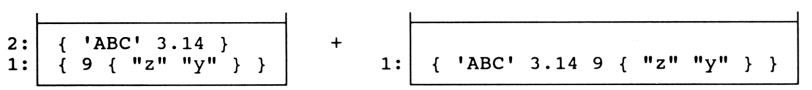
La commande " + " s'applique aux listes, de la manière suivante:

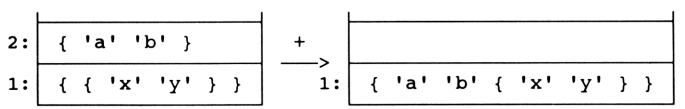


"liste3" est la liste résultant de la concaténation de "liste1" et de "liste2": les éléments de "liste3" sont ceux de "liste1" suivis de ceux de "liste2".

Dans cette addition, la liste vide { } est élément neutre.

Exemples:

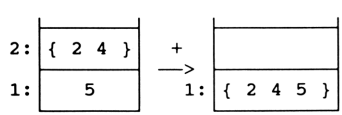


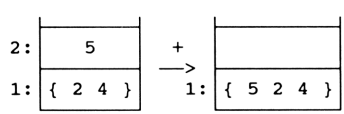


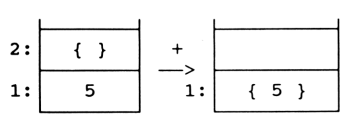
L'exemple ci-dessus montre que pour inclure une liste (ici la liste { 'x' 'y' }) dans une autre liste (ici { 'a' 'b' }) , il a été nécessaire de créer la liste dont l'unique objet est { 'x' 'y' } et concaténer les deux listes { 'a' 'b' } et { { 'x' 'y' } }.
La concaténation de { 'a' 'b' } et { 'x' 'y' } aurait donné la liste { 'a' 'b' 'x' 'y' }, ce qui n'était pas le résultat souhaité.

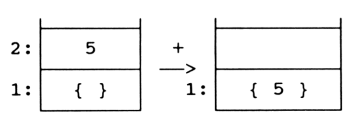
Remarque: Il est également possible d'inclure un objet (qui n'est pas une liste) dans une liste, en utilisant la touche " + ". L'objet est placé en tête ou en queue de la liste, suivant sa position, dans la pile, par rapport à cette liste:

Exemples:









* EVALUATION D'UNE LISTE:

Une liste peut être considérée comme un programme (la seule différence tenant aux délimiteurs: { et } plutôt que « et »). Non seulement une liste peut être saisie exactement comme le serait un programme (parce qu'en ligne de commande, l'appui sur la touche {} ou sur la touche «» provoque le passage en mode "Programme"), mais de plus une liste peut être évaluée (par l'instruction **EVAL**) tout comme le serait un programme.

Il y a tout de même quelques différences:

Quand on évalue une liste (touche ou instruction **EVAL**):

* Les noms sont évalués (alors que dans un programme, seuls les noms sans apostrophes sont évalués).
  De toutes façons, à l'affichage, les noms qui figurent comme objets d'une liste sont toujours montrés sans apostrophes.
  Exemple: si X est une variable globale contenant la valeur 10,
  L'évaluation de « X 'X' » place 10 au niveau 2
  et 'X' au niveau 1.
  Alors que l'évaluation de { X } place 10 au niveau 1.

  Les noms locaux sont évalués (alors que dans un programme, un nom local sans apostrophes amène le contenu de cette variable locale sur la pile, contenu qu'il faut évaluer par **EVAL** s'il s'agit par exemple d'un programme).
  Exemple:
  « « 3 2 * » → X « X » » EVAL
  se traduit par: **« 3 2 * »** au niveau 1.
  { « 3 2 * » → X « X » } EVAL
  se traduit par: **6** au niveau 1.

* Les programmes sont évalués, alors que les programmes à l'intérieur d'un autre programme sont simplement posés sur la pile quand celui-ci est évalué.
  Exemple:
  « « 3 2 * » « 7 13 + » » EVAL
  se traduit par: **« 3 2 * »** au niveau 2
  et: **« 7 13 + »** au niveau 1.
  { « 3 2 * » « 7 13 + » } EVAL
  se traduit par: **6** au niveau 2 et **20** au niveau 1.

* Tout comme dans les programmes, les commandes qui figurent comme objets d'une liste sont éxécutées quand la liste est évaluée.
  Exemple:
  « 7 → x « 1 9 FOR i x i * x + NEXT » »
  et
  { 7 → x « 1 9 FOR i x i * x + NEXT » }
  donnent le même résultat après **EVAL**.

* Les objets qui figurent dans une liste et qui ne sont pas d'un des types vus ci-dessus sont simplement placés sur la pile quand la liste est évaluée (tout comme ils le seraient dans un programme). Il en est ainsi des listes incluses dans la liste évaluée.

* Une dernière remarque: **→NUM** est apparemment sans effet sur une liste (alors que **→NUM** permet d'évaluer un programme).

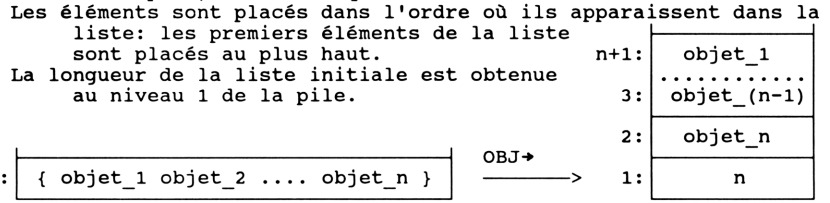
* INSTRUCTIONS DANS LE MENU PRG OBJ:

Le menu **PRG OBJ** contient certaines commandes s'appliquant, entre autres, aux listes. Toutes sauf **→LIST** s'appliquent à un ou plusieurs autres types d'objets (tableaux, chaînes).

OBJ→ **OBJ→** "libère" les éléments d'une liste en les plaçant sur la pile, comme indiqué ci-dessous.

Les éléments sont placés dans l'ordre où ils apparaissent dans la liste: les premiers éléments de la liste sont placés au plus haut.

La longueur de la liste initiale est obtenue au niveau 1 de la pile.



Dans le cas d'une liste vide, **OBJ→** renvoie uniquement l'entier 0 au niveau 1 de la pile.

Les deux commandes **OBJ→** (appliquée à une liste) et **→LIST** sont réciproques l'une de l'autre (utile pour rattraper une utilisation involontaire d'une de ces deux commandes).

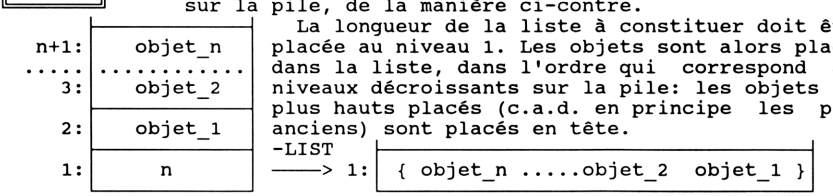
La séquence: **OBJ→** **→ARRY** transforme une liste de nombres réels ou complexes en le vecteur correspondant. Inversement, la séquence: **OBJ→** **1** **GET** **→LIST** transforme un vecteur de réels ou de complexes en la liste correspondante.

Remarque: l'instruction **LIST→** peut être utilisée (elle a le même effet que **OBJ→** sur une liste, mais contrairement à **OBJ→**, elle ne s'applique qu'aux listes. **LIST→** doit être tapé au clavier en toutes lettres.

N.B: L'instruction **OBJ→** ne provoque pas l'évaluation des objets placés dans la liste (contrairement à **EVAL**). **OBJ→** se contente de les placer sur la pile.

→LIST **→LIST** permet de constituer une liste formée d'objets pris sur la pile, de la manière ci-contre.

La longueur de la liste à constituer doit être placée au niveau 1. Les objets sont alors placés dans la liste, dans l'ordre qui correspond aux niveaux décroissants sur la pile: les objets les plus hauts placés (c.a.d. en principe les plus anciens) sont placés en tête.



La séquence: **0** **→LIST** produit une liste vide.

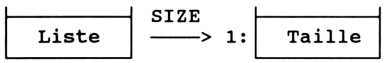
La séquence: **DEPTH** **→LIST** place tous les objets de la pile dans une liste unique.

Si **x** est un réel (non entier), la séquence: **x** **→LIST** ne génère pas d'erreur. Le réel **x** est en fait arrondi à l'entier le plus proche (si $x > 0$) et est considéré comme nul si $x \leq 0$.

Une utilisation simple de **→LIST** est permise dans l'environnement "Pile Interactive" (voir le chapitre "la pile opérationnelle").

SIZE

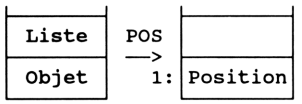
SIZE permet de calculer la taille d'une liste c'est-à-dire le nombre d'objets dont elle est formée.
La liste doit figurer elle-même sur la pile, et ne peut donc être désignée par un nom de variable: si on place un nom au niveau 1 (ex: 'ABC'), **SIZE** considère ce nom comme une expression algébrique et donne le résultat 1.



Le résultat est un entier. Cet entier est nul si la liste est vide.
Si 'Liste' contient n fois le même objet, celui-ci est compté n fois dans le calcul de la taille de la liste.

POS

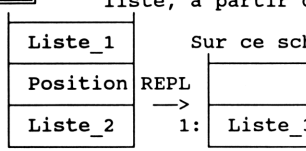
POS permet de savoir si un objet donné appartient à une liste, et si oui quelle est sa position dans la liste.



Comme le montre le schéma fonctionnel, la liste doit figurer elle-même sur la pile (Impossible de la désigner par un nom de variable).
Si 'Objet' ne figure pas dans 'Liste', 'Position' vaut 0.
Il se peut qu'un même objet figure plusieurs fois dans une liste. Si on demande la position de cet objet dans cette liste, le résultat obtenu correspond en fait à la première occurence de cet objet dans cette liste.

REPL

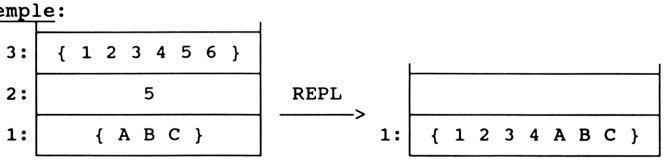
REPL permet de placer le contenu d'une liste dans une autre liste, à partir d'une position spécifiée de celle-ci.



Sur ce schéma fonctionnel, Liste_1 et Liste_2 sont deux listes.
'Position' désigne la position dans Liste_1 à partir de laquelle on va placer le contenu de Liste_2.
Liste_3 est la liste obtenue par cette "écriture" dans Liste_1 du contenu de Liste_2.

Ce placement de Liste_2 dans Liste_1 s'effectue en "écrasant" les termes qui occupaient les positions correspondantes dans Liste_1.
"Position" doit être strictement positif (s'il n'est pas entier, il est arrondi à l'entier le plus proche).
Le placement de Liste_2 dans Liste_1 peut éventuellement se traduire par un allongement de Liste_1 (par ajout des termes excédentaires de Liste_2).
Si "Position" est supérieur à la taille de Liste_1, alors Liste_2 est concaténée à Liste_1.

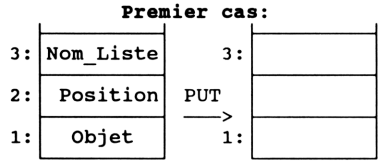
Exemple:



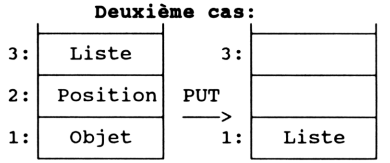
PUT

L'instruction **PUT** permet de placer un objet dans une liste. Il y a deux schémas fonctionnels possibles. 'Objet' est l'objet à placer, 'Position' est l'endroit où on veut le placer.
Dans le premier cas, on indique le nom de la variable contenant la liste ('Nom_Liste' ci dessous). Le placement s'effectue en mémoire (et les 3 arguments disparaissent de la pile).
Dans le deuxième cas, on place au niveau 3 la liste à modifier. La liste modifiée est alors obtenue au niveau 1.

Premier cas:



Deuxième cas:

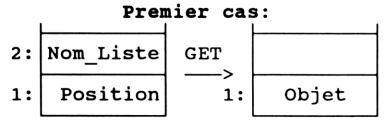


'Position' peut très bien ne pas être entier: il est alors arrondi à l'entier le plus proche. Il doit cependant être compris entre 1 et la taille de la liste d'accueil. On ne peut par exemple pas placer un élément dans la liste vide au moyen de l'instruction **PUT**. En fait, **PUT** ne permet pas d'ajouter d'élément à une liste.
Evidemment l'objet placé dans la liste prend la place de celui qui se trouvait à cet endroit.
L'utilisation de **UNDO** ne permet pas d'annuler l'effet de **PUT** dans le premier des deux cas ci-dessus.

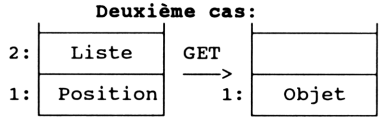
GET

L'instruction **GET** permet de lire un élément dans une liste. Il y a deux schémas fonctionnels possibles. 'Position' désigne où se trouve dans la liste l'élément visé ('Objet').
Dans le premier cas, on indique le nom de la variable contenant la liste ('Nom_Liste' ci dessous).
Dans le deuxième cas, on place au niveau 2 la liste où se trouve l'objet à isoler.
Dans les deux cas l'objet obtenu est placé au niveau 1.

Premier cas:



Deuxième cas:



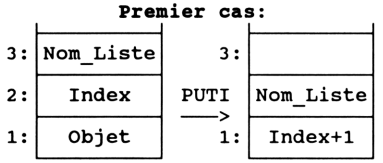
'Position' peut très bien ne pas être entier: il est alors arrondi à l'entier le plus proche. Il doit cependant être compris entre 1 et la taille de la liste lue. On ne peut par exemple pas utiliser l'instruction **GET** avec la liste vide.
Dans le premier cas ci-dessus, l'instruction **GET** ne modifie pas le contenu de la variable 'Nom_Liste' (l'objet lu n'est pas retiré de la liste).

Remarque: Dans **GET**, **PUT** et (ci après) **GETI**, **PUTI**, 'Nom_Liste' peut peut aussi bien être un nom local qu'un nom global.

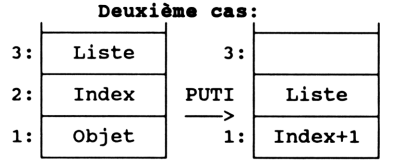
PUTI L'instruction **PUTI** permet de placer plusieurs objets à des positions successives dans une liste. Chaque appel de **PUTI** place un de ces objets. L'intérêt de **PUTI** est qu'elle in--crémente elle même la position où placer l'objet suivant.

Il y a deux schémas fonctionnels possibles. 'Objet' est l'objet à placer, 'Index' est l'endroit où on veut le placer.
Dans le premier cas, on indique le nom de la variable contenant la liste ('Nom_Liste' ci dessous). Le placement s'effectue en mémoire.
Dans le deuxième cas, on place au niveau 3 la liste à modifier. La liste modifiée est alors obtenue au niveau 2.

Premier cas:



Deuxième cas:

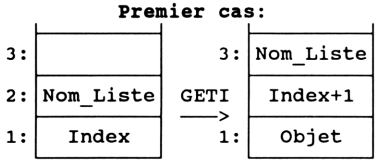


Ci-dessus, 'Index+1' désigne la position qui suit 'Index' dans la liste. Si 'Index' représente la dernière position possible dans la liste, alors 'Index+1' est égal à 1 (ce qui permet de reprendre le placement d'objets au début de la liste)
On voit que des appels successifs de **PUTI** permettent de placer des éléments à des positions successives dans une liste.

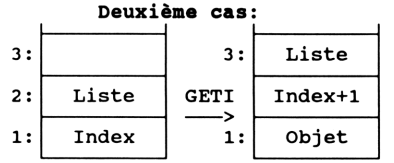
GETI **GETI** permet de lire plusieurs objets successifs dans une liste. Chaque appel de **GETI** lit un de ces objets.
L'intérêt de cette instruction est qu'elle incrémente elle même la position où lire l'élément suivant.

Il y a deux schémas fonctionnels possibles ('Index' représente la position où se trouve l'élément à lire dans la liste, et 'Objet' désigne l'élément ainsi lu).

Premier cas:



Deuxième cas:

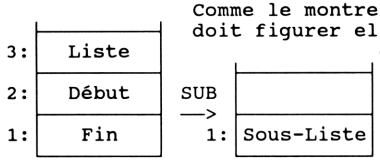


Ci-dessus, 'Index' désigne la position qui suit 'Index' dans la liste. Si 'Index' représente la dernière position possible dans la liste, alors 'Index+1' est égal à 1 (ce qui permet de reprendre la lecture d'objets au début de la liste)
On voit que des appels successifs de **GETI** permettent de lire des éléments successifs dans une liste.

Remarque: Il est important de savoir quand une succession d'appels de **GETI** ou de **PUTI** ont permis de parcourir une liste jusqu'à la fin (et nous ont donc ramené au début de cette liste).
Pour celà on peut tester l'état de l'indicateur N°**-64**.
S'il est armé (valeur = 1), **PUTI** ou **GETI** viennent à l'instant de nous ramener en début de liste. Sinon il est désarmé (à zéro).

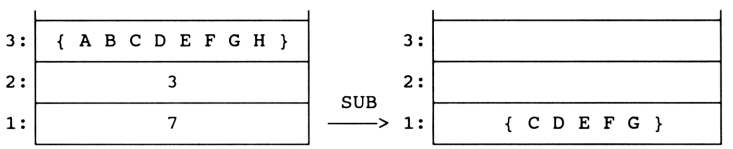
SUB

SUB permet d'extraire une sous-liste d'une liste donnée.



Comme le montre le schéma fonctionnel, la liste doit figurer elle-même sur la pile ,et ne peut donc être désignée par un nom de variable. 'Sous-Liste' désigne ici la liste des éléments de 'Liste' de la position 'Début' à la position 'Fin' (incluses).
Ces éléments apparaissent dans la sous-liste dans l'ordre où ils apparaissaient dans 'Liste'.

'Début' et 'Fin' peuvent ne pas être entiers. Ils sont alors arrondis à l'entier le plus proche.
Si 'Début' ≤ 0, l'effet est le même que si 'Début' = 1.
Si 'Fin' ≥ n (où n est le nombre d'objets de 'Liste'), l'effet est le même que si 'Fin' = n.
Toujours en appelant n le nombre d'éléments de 'Liste', La sous-liste obtenue est vide dans les cas suivants: 'Début'>n, ou 'Fin'<1, ou 'Début'>'Fin'.
La séquence: **LISTE k k SUB** permet de créer la liste réduite au k-ème objet de 'LISTE' (alors que **LISTE k GET** isole totalement cet objet).



LECTURE ET ECRITURE DANS DES LISTES:

Comme nous venons de le voir, il y a deux façons d'écrire un objet dans une liste: au moyen de l'instruction **PUT** et au moyen de l'instruction **PUTI** (La liste pouvant être "physiquement" présente sur la pile ou bien représentée par un nom, auquel cas l'écriture se fait dans le contenu de la variable ainsi référencée).

Il y a par contre 3 moyens de lire un objet dans une liste:

On peut utiliser les instructions **GET** ou **GETI** (La liste étant présente sur la pile, ou bien représentée par un nom).

On peut également utiliser l'instruction **EVAL** (ou **→NUM** qui force l'évaluation numérique des constantes symboliques et des noms).

Ainsi si A est le nom (local ou global) d'une variable dont le contenu est une liste: la séquence **'A(k)' EVAL** place le k-ème élément de la liste sur la pile (k étant un entier ou un nom désignant un entier).

Plus généralement, on peut ainsi référencer tel ou tel élément d'une liste, à l'intérieur d'une expression algébrique.
Par ex: **'A(k)*k+A(A(k))' EVAL** est syntaxiquement correct.

Il est intéressant de comparer les possibilités énumérées ci-dessus, pour tester au moins leur rapidité.
Comme nous allons le voir le résultat est instructif.

Exemple 1: On suppose l'existence d'une variable nommée A (nom global) dont le contenu est une liste de 10 éléments (J'ai utilisé la liste { 1 2 3 ... 10 }.
On va écrire un programme effectuant 50 lectures séquentielles de cette liste (du début à la fin).

```
'PROG1':  «  'A'  {1}
             1   50   START
                1   10   START   GETI   DROP   NEXT
             NEXT
             DROP2
          »
```

'PROG2': identique à PROG1, en remplaçant **'A'** par **A.** On lit donc les éléments directement dans la liste sur la pile, sans référencer cette liste par le nom de la variable.

```
'PROG3':  «  1   50   START
                1   10   FOR   k   'A'   k   GET   DROP   NEXT
             NEXT
          »
```

'PROG4': identique à PROG3, en remplaçant **'A'** par **A.**

```
'PROG5':  «  A   1   50   START
                1   10   FOR   k   DUP   k   GET   DROP   NEXT
             NEXT   DROP
          »
```

```
'PROG6':  «  1   50   START
                1   10   FOR   k   'A(k)'   EVAL   DROP   NEXT
             NEXT
          »
```

J'ai mesuré les temps de calculs nécessaires à l'éxécution de ces programmes (qui font tous la même chose).

'PROG1': 30 s, 'PROG2': 23s, 'PROG3': 17s, 'PROG4': 13s
'PROG5': 12 s, 'PROG6': 32s.

Ces résultats sont très significatifs:

L'instruction **GETI** est lente par rapport à son équivalent **GET.**

De même, l'utilisation de **EVAL**,pour lire un élément donné d'une liste, ralentit beaucoup l'éxécution.

Toujours pour une rapidité maximum, il vaut mieux placer la liste sur la pile, plutôt que de la référencer par le nom de la variable.

Enfin, on gagne encore un peu de temps (comme le montre 'PROG5' par rapport à 'PROG4') en évitant de rappeler systématiquement la liste sur la pile (on l'appelle une première fois puis on utilise **DUP** pour la conserver).

Exemple 2:

Les mêmes remarques s'appliquent dans les procédures d'écriture dans une liste: On suppose l'existence d'une variable globale nommée 'A' dont le contenu est une liste de 10 objets. On écrit un programme qui, 50 fois de suite, place 1 à chaque position de la liste.

```
'PROG1':  «   A   1   50   START
              1   10   FOR   k   k   1   PUT   NEXT
          NEXT   DROP
      »

'PROG2':  «   A   { 1 }   1   50   START
              1   10   START   1   PUTI   NEXT
          NEXT   DROP2
      »
```

Les temps de calculs sont les suivants:

'PROG1': 16 secondes. 'PROG2': 30 secondes.

Là encore, on remarque la relative lenteur de **PUTI** par rapport à l'instruction **PUT**.

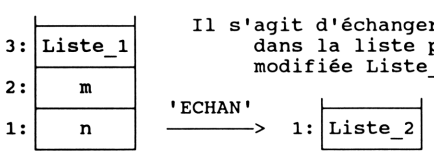
EXERCICES SUR LES LISTES:

La difficulté des exercices est mesurée par des symboles * ou # :

* Très facile.
** Assez simple.
*** Déjà un peu plus délicat.
**** Plutôt difficile.

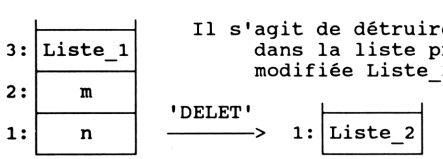
Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs programmes précédant dans cette liste d'exercices.

Exercice 1 (**) : Echanger deux éléments dans une liste.



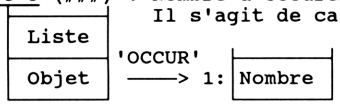
Il s'agit d'échanger les éléments d'indice m et n dans la liste présente au niveau 3. La liste modifiée Liste_2 est obtenue au niveau 1.

Exercice 2 (**) : Détruire une partie d'une liste.



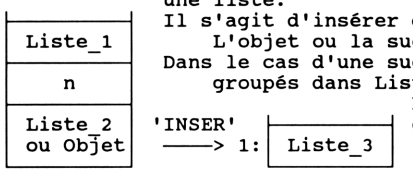
Il s'agit de détruire les éléments d'indice m à n dans la liste présente au niveau 3. La liste modifiée Liste_2 est obtenue au niveau 1.

Exercice 3 (***###***) : Nombre d'occurences d'un objet dans une Liste.



Il s'agit de calculer le nombre de fois qu'un objet apparaît dans une liste. Le résultat, ici appelé 'Nombre', vaut 0 si l'objet n'appartient pas à la liste.

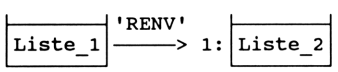
Exercice 4 (**) : Insérer un objet ou une succession d'objets dans une liste.



Il s'agit d'insérer dans Liste_1, à la position n, L'objet ou la succession d'objets du niveau 1. Dans le cas d'une succession d'objets ceux-ci sont groupés dans Liste_2. La liste modifiée Liste_3 est obtenue au niveau 1.

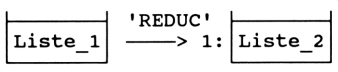
Attention: pour insérer par exemple l'objet { 4 5 } dans Liste_1, il faut placer { { 4 5 } } au niveau 1. Si on place { 4 5 } au niveau 1, alors les deux éléments 4 et 5 sont insérés successivement dans Liste_1, à partir de la position n.

Exercice 5 (*) : Inverser l'ordre des éléments dans une liste.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des mêmes éléments, mais dans l'ordre contraire.

Exercice 6 (***###***) : Eliminer les occurences multiples dans une liste.

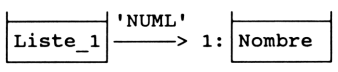


On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant qu'un exemplaire des différents éléments de Liste_1.

Ce programme élimine donc de Liste_1 tous les objets qui sont les répliques d'éléments déjà lus dans Liste_1.

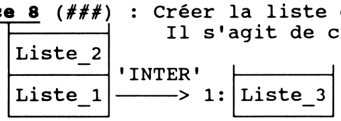
Par exemple, { 1 3 1 2 1 3 5 1 4 1 } devient { 1 3 2 5 4 }

Exercice 7 (***#***) : Nombre d'objets différents dans une liste.



Il s'agit ici de calculer le nombre d' objets réellement différents dans Liste_1. Si un objet apparait plusieurs fois dans Liste_1, il n' est donc compté qu'une seule fois.

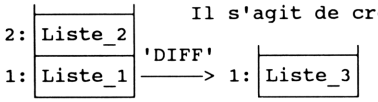
Exercice 8 (***###***) : Créer la liste des éléments communs à deux listes.



Il s'agit de créer la liste Liste_3 formée des objets qui apparaissent à la fois dans Liste_1 et dans Liste_2. Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

Par ex: les deux listes {2 1 3 4 5 1 2 1} et {1 3 1 2 4 2 2 3 2} produisent la liste { 2 3 4 1 2 1 }.

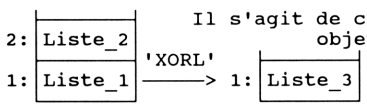
Exercice 9 (###) : Produire la soustraction de deux listes.



Il s'agit de créer la liste Liste_3 formée des objets qui appartiennent à Liste_2 mais pas à Liste_1.
Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

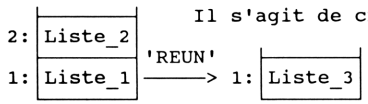
Par exemple, si Liste_1 est égale à { 2 1 3 4 5 1 2 1 } et si Liste_2 est égale { 1 3 1 2 4 2 2 3 2 } alors le résultat Liste_3 vaut { 1 5 }.
Si on inverse le rôle ci-dessus de Liste_1 et Liste_2, alors Liste_3 vaut { 3 2 2 }.

Exercice 10 (#) : Créer la liste des éléments appartenant à une seule de deux listes.



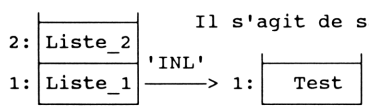
Il s'agit de créer la liste Liste_3 formée des objets qui apparaissent dans une seule des 2 Listes Liste_1 et Liste_2.
Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.
Par ex: les deux listes {2 1 3 4 5 1 2 1} et {1 3 1 2 4 7 7 3 2} produisent la liste {1 5 3 7 7} (dans votre solution l'ordre des éléments peut être différent).

Exercice 11 (#) : Créer la réunion ensembliste de deux listes.



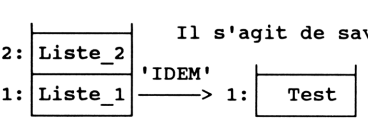
Il s'agit de créer la liste Liste_3 formée des objets qui appartiennent au moins à une des deux listes Liste_1 et Liste_2. Si un même objet apparait m fois dans Liste_1 et n fois dans Liste_2, alors il doit apparaître max(m,n) fois dans Liste_3.
Par ex: les deux listes {2 1 3 4 5 1 2 1} et {1 3 1 2 4 2 2 3 2} produisent la liste { 2 1 3 4 5 1 2 1 3 2 2 }.
(l'ordre des éléments peut être différent).

Exercice 12 (#) : Une liste est-elle incluse dans une autre liste?.



Il s'agit de savoir si Liste_1 est incluse dans Liste_2.
La réponse, nommée ici 'Test' vaut 1 si la réponse est oui, 0 sinon.
Si un même objet apparaît plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.
Par exemple { 1 2 3 1 } est incluse dans { 3 1 5 1 2 } mais pas dans { 3 1 5 2 }.

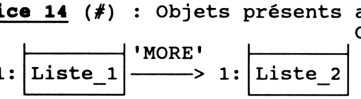
Exercice 13 (#) : Deux listes sont-elles égales, d'un point de vue ensembliste?



Il s'agit de savoir si Liste_1 est formée des mêmes objets que Liste_2.
La réponse, nommée ici 'Test' vaut 1 si la réponse est oui, 0 sinon.
Si un même objet apparaît plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

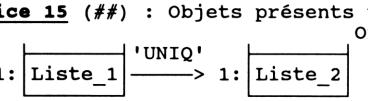
Attention:
L'égalité 'ensembliste' de 2 listes est réalisée si elles sont constituées exactement des mêmes éléments (l'ordre étant sans importance).
Par contre l'égalité de 2 listes avec l'instruction **SAME** est réalisée si les 2 listes sont formées des mêmes éléments, dans le même ordre.
Ici, pour prendre un exemple, { 2 5 3 1 2 } et { 5 2 2 1 3 } doivent être considérées comme égales.

Exercice 14 (#) : Objets présents au moins deux fois dans une liste.



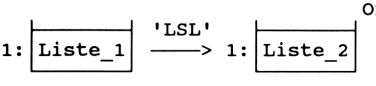
On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant que les objets qui apparaissent au moins 2 fois dans Liste_1 (chacun d'eux n'étant reproduit qu'une fois).
Par exemple, { 1 3 1 2 1 3 5 1 4 1 } devient { 1 3 }

Exercice 15 (##) : Objets présents une seule fois dans une liste.



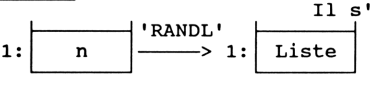
On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant que les objets qui apparaissent une seule fois dans Liste_1.
Par exemple, { 1 3 1 2 1 3 5 1 4 1 } devient { 2 5 4 }.

Exercice 16 (****) : Liste des sous-listes d'une liste donnée.



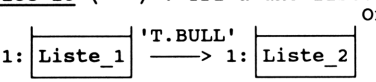
On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée de toutes les sous-listes de Liste_1 (y compris la liste vide { }).
Si la taille de Liste_1 est n, celle de Liste_2 est 2^n.

Exercice 17 (*) : Création d'une liste pseudo-aléatoire d'entiers.



Il s'agit de créer une liste aléatoire, de longueur n, formée d'entiers compris entre 0 et 999.
Une telle liste sera utile pour tester les procédures de tri des exercices 18, 19, 20.

Exercice 18 (***) : Tri d'une liste par la méthode de la bulle.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode du "tri-bulle".

La méthode de la bulle est basée sur le principe suivant:

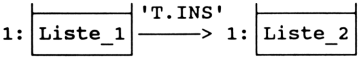
* On parcourt la liste du 1er au n-ième et dernier élément. A chaque étape de ce parcours, on compare deux élements consécutifs. On les échange s'ils ne sont pas dans l'ordre croissant.
  A la fin de ce parcours, le plus grand élément, tel une bulle, est "remonté" en dernière position, et se trouve donc bien placé.

* On parcourt alors la sous-liste des n-1 premiers éléments, en remontant ainsi le plus grand des éléments lus en avant-dernière position.

* On continue ainsi, chaque nouveau parcours plaçant un nouvel élement, jusqu'à ce qu'il ne reste plus qu' à parcourir la sous-liste des deux premiers éléments. Après un dernier échange éventuel, la liste se trouve entièrement triée.

Remarque: la liste doit être constituée d'objets comparables au moyen de l'opérateur '<' (par exemple des réels, ou des chaines de caractères, ou des entiers binaires).

Exercice 19 (****) : Tri d'une liste par la méthode d'insertion.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode d'insertion.

La méthode d'insertion est basée sur le principe suivant:

* On commence par trier les deux premiers éléments (en les échangeant s'ils ne sont pas dans le bon ordre).

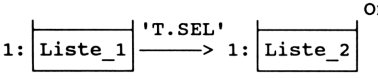
* On place ensuite le troisième élément dans cette première sous-liste de deux éléments. La sous-liste des trois premiers éléments est alors bien triée.

* On insère ensuite le 4ème élément dans cette sous-liste de trois éléments, etc...

* On continue ainsi jusqu'à n' avoir plus qu'à déterminer la position du n-ème et dernier élément dans la sous-liste triée formée des n-1 premiers éléments. A ce moment, la liste est correctement triée.

Remarque: la liste doit être constituée d'objets comparables au moyen de l'opérateur '<' (par exemple des réels, ou des chaines de caractères, ou des entiers binaires).

Exercice 20 (****) : Tri d'une liste par la méthode de sélection.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode de sélection.

La méthode de sélection est basée sur le principe suivant:

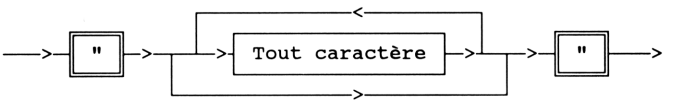
* On parcourt toute la liste (n élements), en mémorisant la position de l'élément maximum. On échange alors cet élément avec le dernier élément de la liste.
* On parcourt ensuite la sous-liste des n - 1 premiers éléments, en mémorisant, puis en plaçant en n-1ème position, le plus grand élément rencontré.
* On continue ainsi, chaque nouvelle lecture permettant de placer correctement un élément. Le dernier parcours s'effectue sur la liste des deux premiers éléments. La liste complète est alors triée dans l'ordre croissant.

Remarque: la liste doit être constituée d'objets comparables au moyen de l'opérateur '<' (par exemple des réels, ou des chaines de caractères, ou des entiers binaires).

LES CHAINES DE CARACTERES

CHAINES DE CARACTERES:

Une chaîne de caractères est une succession quelconque de caractères quelconques, encadrée par deux caractères ".
Cette notion peut être représentée par le diagramme suivant:



Les deux caractères " qui encadrent la chaîne ne doivent pas être considérés comme faisant partie de celle-ci (notamment dans le calcul de la longueur de la chaîne).

Le rôle joué par le caractère ", ne permet à priori pas de créer des chaînes contenant ce caractère. Il y a pourtant une possibilité en utilisant l'instruction **CHR**, ou les chaînes comptées (cf. plus loin).

Exemples:
"" est la chaîne vide (ne contient donc aucun caractère).
"Hewlett-Packard" et "HEWLETT-PACKARD" sont deux chaînes différentes.

* SAISIE D'UNE CHAINE:

La saisie d'une chaîne commence par la frappe du caractère "", le curseur venant alors se placer en superposition avec le 2-ème ".
Pour taper un caractère "alphabétique", on le préfixe par un appui simple sur la touche α (témoin α allumé, c'est-à-dire mode "alpha" sélectionné pour une touche).
Un double appui sur α verrouille le clavier "alpha". Un nouvel appui sur α ramène au clavier standard.

Si le clavier "alpha" est verrouillé:
La séquence <=┐ α alterne entre le clavier "alpha" et le clavier "alpha-shifté-gauche" (en gros le clavier "alpha minuscules").

En mode "alpha", la séquence ┌=> α permet d'accéder, pour une touche, au clavier "alpha-shifté-droit" (ex: caractères grecs).

Un appui sur **ENTER** (qui valide la saisie), ou **ON** (qui l'annule), ramène à l'utilisation habituelle du clavier .

CHAINES DE CARACTERES.

*** AFFICHAGE D'UNE CHAINE:**

Une chaîne peut être visible sur la pile. Elle apparaît alors avec ses délimiteurs ". Elle peut être affichée à l'écran au moyen de l'instruction **DISP** (elle apparaît alors sans délimiteurs).

Sur une ligne donnée, une chaîne n'est totalement visible que si sa longueur n'excède pas 17 caractères (si affichage sur la pile) ou 22 caractères (si affichage obtenu par **DISP**).

Tous les caractères ont un code compris entre 0 et 255. Les codes compris entre 32 et 128 sont les codes du standard ASCII.
(Un tableau des caractères et de leurs codes se trouve aux pages 722 et 723 du volume II du manuel d'utilisation de la HP48SX).

Tous les caractères de code ≥ 32 sont affichables à l'écran, les autres (caractères de code compris entre 0 et 31) sont affichés sous forme d'un petit carré ▪.

Cas particuliers:
Le caractère de code 10 représente le saut de ligne. S'il est présent dans une chaîne affichée par **DISP**, il a effectivement son rôle de saut de ligne.
Le caractère de code 0 est utilisé par la HP48SX, de façon interne. On ne peut pas éditer, en ligne de commande, une chaîne contenant le caractère 0.

*** CHAINES COMPTÉES:**

Une chaîne de caractères peut avoir une longueur quelconque.
Il est néanmoins possible de spécifier en ligne de commande la longueur de la chaîne en cours de saisie, en utilisant la notion de chaîne comptée.

Une chaîne comptée commence par la séquence **C$ n**, où n est un entier (ou réel mais il est alors arrondi à l'entier le plus proche) et se continue sur la ligne de commande:
* Les n caractères qui suivent **C$ n** sont groupés pour former une chaîne de caractères, le reste de la ligne de commande étant traité comme d'habitude. Le premier caractère pris en compte est le premier caractère suivant C$ et qui n'est pas un espace.
* Il est inutile d'entrer les caractères " pour constituer cette chaîne (si vous le faites, ils feront intégrés à la chaîne constituée).
* Si la séquence **C$ n** est suivie de moins de n caractères, la chaîne obtenue se contente de regrouper les caractères suivant **C$ n**, et sa longueur est donc inférieure à n.
* Cas particulier: La séquence **C$ $** forme une chaîne avec tous les caractères qui la suivent en ligne de commande.

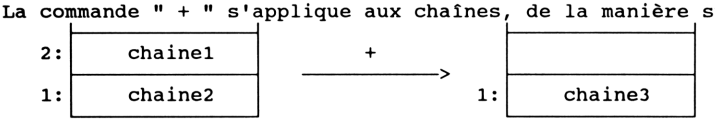
Remarque: le caractère $ est obtenu par la séquence: α <=┐ TIME
(penser à "Time is money"...)

Exemples:
Valider la ligne de commande: **C$ 6 ABCDEF 1789** place la chaîne **"ABCDEF"** au niveau 2 et l'entier **1789** au niveau 1.
Valider: **C$ $ 1789 [1 2 3 4]**
place la chaîne **"1789 [1 2 3 4]"** au niveau 1.
Valider: **C$ 10 "ABC"** place la chaîne **""ABC""** au niveau 1.

OPÉRATIONS SUR LES CHAINES DE CARACTERES:

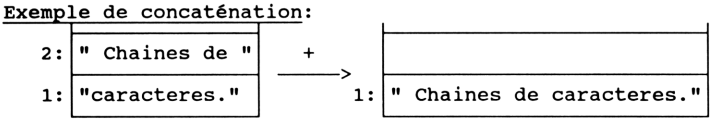
* LA CONCATÉNATION:

La commande " + " s'applique aux chaînes, de la manière suivante:



"chaine3" est la chaine résultant de la concaténation de "chaine1" et de "chaine2": les éléments de "chaine3" sont ceux de "chaine1" suivis de ceux de "chaine2".
Dans cette addition, la chaine vide "" est élément neutre.

Exemple de concaténation:



* OPÉRATIONS LOGIQUES SUR LES CHAINES:

Il est possible d'appliquer les opérateurs logiques **NOT**, **AND**, **OR**, et **XOR** aux chaines de caractères.
En fait ces opérateurs logiques doivent être considérés comme s'appliquant aux codes des caractères de la chaîne (dans le cas de **NOT**) ou des deux chaînes (cas de **AND**, **OR**, **XOR**) concernées.
NOT est un opérateur unaire;
AND, **OR** et **XOR** sont des opérateurs binaires commutatifs.

Ces codes sont des entiers compris entre 0 et 255 (donc représentés par 8 bits en numérotation binaire).

Les opérations **NOT**, **AND**, **OR** et **XOR** agissent bit par bit, de la manière suivante:

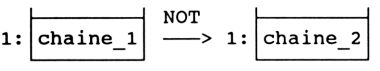
	0	1
NOT	1	0

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

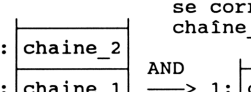
NOT effectue un 'non' logique sur les codes des caractères de la chaîne chaine_1 située au niveau 1.
(Chaque bit d'un tel code est inversé).



La chaîne ainsi modifiée est obtenue au niveau 1.
Si par exemple chaîne_1 contient le caractère "s" dont le code est 115 = 01110011b, alors ce caractère devient "α" dont le code est 140 = 10001100b.
La chaîne obtenue a évidemment même longueur que la chaîne initiale. Il est même possible d'appliquer **NOT** à la chaîne vide, qui reste inchangée.
Une telle opération est involutive (appliquée deux fois de suite, elle redonne la chaîne initiale).

AND

AND effectue un 'et' logique entre les codes des caractères se correspondant respectivement dans les chaînes chaîne_1 et chaîne_2 présentes aux niveaux 1 et 2. La chaîne ainsi modifiée est obtenue au niveau 1.

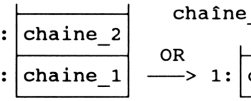


Si par exemple, un caractère ":" de chaîne_1 correspond à un caractère "k" de chaîne_2, alors, le caractère obtenu à cette position dans la chaîne chaîne_3 est "*".
En effet le code de ":" est 58 = 00111010b
et celui de "k" est 107 = 01101011b.
Puisque (00111010b **AND** 01101011b) vaut 00101010b = 42, on obtient bien le caractère "*", de code 42.

Remarques:
Il faut,sous peine d'erreur à l'éxécution, que chaîne_1 et chaîne_2 aient même longueur.
Si chaîne_1 = chaîne_2 , alors chaîne_3 = chaîne_1.

OR

OR effectue un 'ou' logique entre les codes des caractères se correspondant respectivement dans les chaînes chaîne_1 et chaîne_2 présentes aux niveaux 1 et 2. La chaîne ainsi modifiée est obtenue au niveau 1.

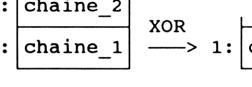


Si par exemple, un caractère "9" de chaîne_1 correspond à un caractère "C" de chaîne_2, alors, le caractère obtenu à cette position dans la chaîne chaîne_3 est "{".
En effet le code de "9" est 00111001b
et celui de "C" est 01000011b.
Puisque (00111001b **OR** 01000011b) vaut 01111011b = 123, on obtient bien le caractère "{", de code 123.

Remarques:
Il faut,sous peine d'erreur à l'éxécution, que chaîne_1 et chaîne_2 aient même longueur.
Si chaîne_1 = chaîne_2 , alors chaîne_3 = chaîne_1.

XOR

XOR effectue un 'ou exclusif' entre les codes des caractères se correspondant respectivement dans les chaînes chaîne_1 et chaîne_2 présentes aux niveaux 1 et 2. La chaîne ainsi modifiée est obtenue au niveau 1.



Si par exemple, un caractère "A" de chaîne_1 correspond à un caractère "w" de chaîne_2, alors, le caractère obtenu à cette position dans la chaîne chaîne_3 est "6".
En effet le code de "A" est 65 = 01000001b
et celui de "w" est 119 = 01110111b
Puisque (01000001b **XOR** 01110111b) vaut 00110110b = 54, on obtient bien le caractère "6", de code 54.

Remarques sur l'opérateur XOR :

Il faut, sous peine d'erreur à l'éxécution, que chaîne_1 et chaîne_2 aient même longueur.
Si chaîne_1 = chaîne_2 , alors chaîne_3 n'est constituée que de caractères de code 0.
Soient 'Code' et 'Ch1' deux chaînes de caractères et 'Ch2' le résultat de **XOR** entre 'Code' et 'Ch1'.
Alors le résultat de **XOR** entre 'Code' et 'Ch2' est 'Ch1'.
Autrement dit, deux **XOR** successifs avec la même chaîne de caractères sont sans effet sur la chaîne initiale.
Celà présente deux avantages:
* un **XOR** involontaire peut être corrigé facilement.
* l'opérateur **XOR** est utile pour les opérations de codage et de décodage de chaînes de caractères.

* COMPARAISON DES CHAINES DE CARACTERES:

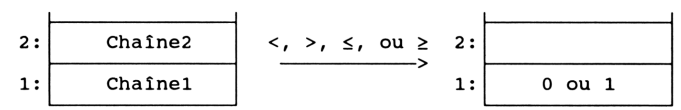
Deux chaînes de caractères peuvent être comparées au moyen des opérateurs suivants:

Test d'égalité: **SAME** ou == (renvoie 1 si les deux chaînes sont identiques et 0 sinon).

Test de différence: ≠ (renvoie 1 si les deux chaînes sont différentes, 0 si elles sont identiques).

Comparaison alphabétique: <, >, ≤, ou ≥.

Notons Chaîne1 et Chaîne2 les deux chaînes comparées ici:



Le résultat est 1 si la réponse au test de comparaison est vraie, et 0 sinon. Dans le cas de l'opérateur < , par exemple , la réponse est 1 si effectivement Chaîne2 est considérée comme inférieure à Chaîne1.

Les chaînes Chaîne1 et Chaîne2 sont comparées caractère par caractère, à partir du premier, et jusqu'à ce qu'un caractère permette de les différencier . Les deux chaînes sont alors ordonnées conformément à l'ordre de ces deux caractères.

Si aucun caractère ne permettant de différencier Chaîne1 et Chaîne2 n'est détecté avant la fin d'une de ces deux chaînes, la chaîne la plus courte est considérée comme inférieure à l'autre.
En particulier, la chaîne vide est considérée comme inférieure à toute chaîne non vide.

La comparaison s'effectue sur le code des caractères:
Deux caractères sont dans l'ordre de leurs codes.

Par exemple: " " < "!" < "0" < "1" < "A" < "a" < "«".

CHAINES DE CARACTERES.

On remarque que les caractères "0",.."9" sont ordonnés dans l'ordre naturel.
Il en est de même pour les caractères alphabétiques majuscules entre eux, et pour les caractères minuscules entre eux.
Néanmoins, un caractère alphabétique majuscule est toujours considéré comme inférieur à tout caractère minuscule.

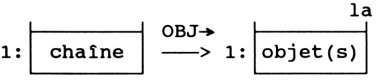
Exemples:

"Bonjour" < "bonjour" (un caractère majuscule est inférieur au caractère minuscule correspondant)
"BONJOUR" < "BONJOUR " (un espace de plus à la fin de la 2-ème chaîne)
" BONJOUR" < "BONJOUR" (l'espace qui commence la première des 2 chaînes est considéré comme inférieur à tout caractère alphabétique)

*** OPÉRATIONS DU MENU PRG OBJ:**

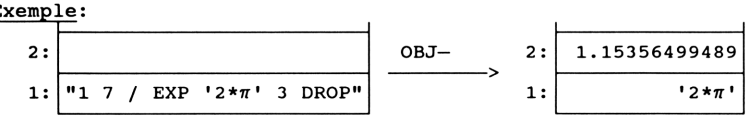
Le menu **PRG OBJ** contient des commandes concernant les chaînes de caractères (certaines de ces commandes, comme **POS**, **SUB**, **SIZE**, s'appliquent à d'autres catégories d'objets (tableaux, listes)).

OBJ→ **OBJ→** transforme une chaîne de caractères en un ou plusieurs objets. La chaîne est interprétée de la même manière que la ligne de commande lors de l'appui sur la touche **ENTER**.



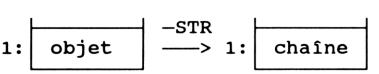
C'est pourquoi la commande **OBJ→** peut aussi bien se traduire par le placement d'un ou plusieurs objets sur la pile et/ou par l'éxécution d'une ou plusieurs instructions.

Exemple:



Sur cet exemple (assez peu probable dans la pratique tout de même), on voit que la chaîne de caractères a été évaluée comme l'aurait été la ligne de commande:
1 a été divisé par 7, on a pris l'exponentielle de ce nombre, on a ensuite placé l'expression '2*π' sur la pile, puis on y a placé l'entier 3 que l'on a immédiatement détruit par **DROP**.

→STR **→STR** transforme un objet quelconque en une chaîne de caractères. Si c'est déjà une chaîne, elle est inchangée. L'intérêt de cette commande est qu'elle permet d'intervenir sur chaque caractère apparaissant dans l'affichage d'un objet.



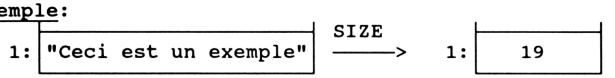
La chaîne obtenue représente l'objet tout entier (même s'il est trop grand pour être entièrement affiché) et des caractères de passage à la ligne sont insérés par endroits ce qui correspond à une représentation en 'multi-lignes' de l'objet initial (ceci vaut pour les matrices ,les listes ,les programmes).

Les commandes **→STR** et **OBJ→** (appliquée à une chaîne) sont à peu près réciproques l'une de l'autre:
Un objet transformé en une chaîne par **→STR** est ainsi 'récupéré' par **OBJ→** .
Il peut tout de même y avoir une différence, comme le montre l'exemple suivant:
Le réel '5/3' s'écrit 1.66666666667 en mode standard.
En mode 4 décimales (**4 FIX**), il s'écrit 1.6667 , mais il est encore connu en mémoire avec toute sa précision.
L'instruction **→STR** le transforme en "1.6667" , et **OBJ→** transforme cette chaîne en le réel 1.6667 (connu comme tel en mémoire, comme on s'en rend compte en revenant au mode standard d'affichage des réels).

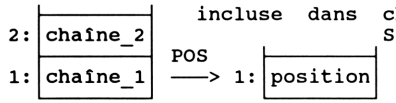
SIZE **SIZE** calcule la taille (le nombre de caractères) d'une chaîne. Le résultat est un entier positif ou nul.

(taille =0 si la chaîne est vide)

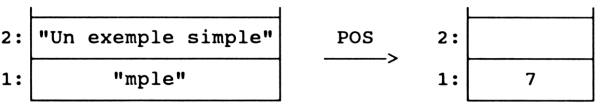
Exemple:



POS Comme l'indique le schéma fonctionnel ci-dessous, **POS** donne la position où se trouve la première occurence de chaîne_1 à l'intérieur de chaîne_2. Si chaîne_1 n'est pas incluse dans chaîne_2, le résultat est 0. Sinon l'entier obtenu représente la position du premier caractère de la 1-ière occurence de chaîne_1 dans chaîne_2.

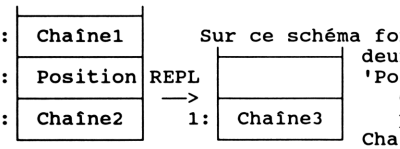


Exemple:



En effet la chaîne "mple" est ici présente deux fois dans la chaîne "Un exemple simple" . Néanmoins, la première occurence est détectée au 7-ème caractère.

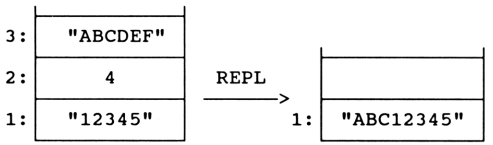
REPL **REPL** permet de placer le contenu d'une chaîne dans une autre chaîne, à partir d'une position spécifiée de celle-ci.



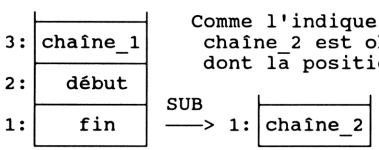
Sur ce schéma fonctionnel, Chaîne1 et Chaîne2 sont deux chaînes.
'Position' désigne la position dans Chaîne1 à partir de laquelle on va placer le contenu de Chaîne2.
Chaîne3 est la chaîne obtenue par cette "écriture" dans Chaîne1 du contenu de Chaîne2.

Ce placement de Chaîne2 dans Chaîne1 s'effectue en "écrasant" les caractères qui occupaient les positions correspondantes dans Chaîne1.
"Position" doit être strictement positif (s'il n'est pas entier, il est arrondi à l'entier le plus proche).
Le placement de Chaîne2 dans Chaîne1 peut éventuellement se traduire par un allongement de Chaîne1 (par ajout des termes excédentaires de Chaîne2).
Si "Position" est supérieur à la taille de Chaîne1, alors Chaîne2 est concaténée à Chaîne1.

Exemple:



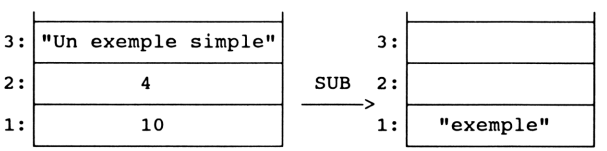
SUB L'instruction **SUB** permet d'extraire une sous-chaîne d'une chaîne de caractères donnée.



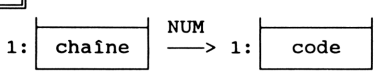
Comme l'indique le schéma fonctionnel ci-contre, chaîne_2 est obtenue en conservant les caractères dont la position dans chaîne_1 est comprise entre 'début' et 'fin' (compris).
Si 'début' ou 'fin' ne sont pas entiers, ils sont arrondis à l'entier le plus proche.

Si 'début' < 1, l'effet est la même que si 'début'=1.
Si 'fin' > n (en appelant n la taille de chaîne_1) l'effet est le même que si 'fin' = n.
Si 'début' > 'fin', la chaîne obtenue est vide.
La séquence: **i i SUB** permet d'extraire le i-ème caractère d'une chaîne.

Exemple:



NUM L'instruction **NUM** calcule le code du caractère débutant la chaîne placée au niveau 1.
Le résultat est donc un entier compris entre 0 et 255.

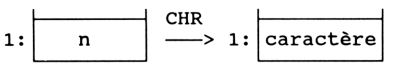


Dans le cas d'une chaîne formée d'un seul caractère, **NUM** et **CHR** sont deux instructions réciproques (vopir **CHR** ci-après).

Remarque: on ne confondra pas les 2 instructions **NUM** et **→NUM**.

CHR

CHR transforme un entier n en le caractère de code n (tout caractère possède un code compris entre 0 et 255). Le résultat est une chaîne de longueur 1.



Le caractère est affiché sous forme d'un carré ▪ si son code n'est pas ≥ 32.
(voir le tableau de correspondance, pages 722 723 du manuel d'utilisation de la HP48SX).

Remarques:
Il n'y a pas d'erreur si n est un réel non entier ou non compris entre 0 et 255:
* Si n < 0, n est considéré comme nul.
* Si n est positif, il est arrondi à l'entier le plus proche et remplacé par le reste dans sa division par 256 (pour obtenir ainsi un code compris entre 0 et 255).

La séquence **10 CHR +** permet d'ajouter un 'retour-chariot' à la fin d'une chaîne.
La séquence **34 CHR +** permet d'ajouter le caractère ".

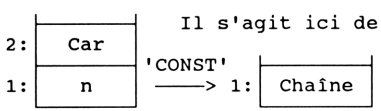
EXERCICES SUR LES CHAINES DE CARACTERES:

La difficulté des exercices est mesurée par des symboles * ou # :

* Très facile.
** Assez simple.
*** Déjà un peu plus délicat.
**** Plutôt difficile.

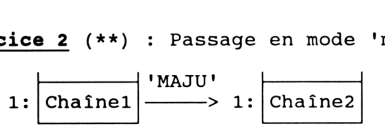
Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes qui précèdent dans cette liste d'exercices.

Exercice 1 (*) : Création d'une chaîne répétitive.



Il s'agit ici de créer une chaîne constituée de la répétition de n exemplaires de 'Car' qui peut être un caractère ou une chaîne de caractères.

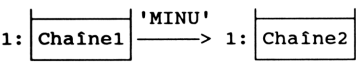
Exercice 2 (**) : Passage en mode 'majuscules'.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par passage des caractères en majuscules.

Le passage en majuscules n'a de sens que pour les caractères alphabétiques.Les caractères non alphabétiques pourront être affectés d'une manière non évidente par **'MAJU'**.
En fait, le passage d'une lettre 'minuscule' à la majuscule qui lui correspond s'effectue en retranchant 32 au code du caractère.
Il est donc logique que **'MAJU'** transforme par exemple { dont le code est 123, en [, dont le code est 123 - 32 = 91.

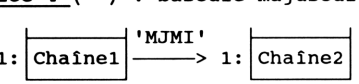
Exercice 3 (**) : Passage en mode 'minuscules'.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par passage des caractères en minuscules.

Le passage en minuscules n'a de sens que pour les caractères alphabétiques.

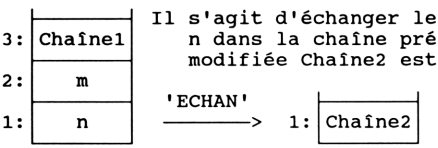
Exercice 4 (**) : bascule majuscules/minuscules.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par transformation des majuscules en minuscules et des minuscules en majuscules.

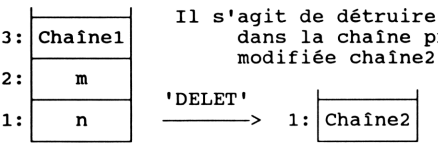
La notion de majuscules ou de minuscules n'a de sens que pour les caractères alphabétiques.

Exercice 5 (**) : Echanger deux caractères dans une chaîne.



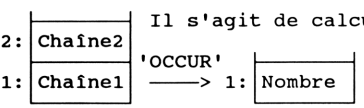
Il s'agit d'échanger les caractères de position m et n dans la chaîne présente au niveau 3. La chaîne modifiée Chaîne2 est obtenue au niveau 1.

Exercice 6 (**) : Détruire une partie d'une chaîne de caractères.



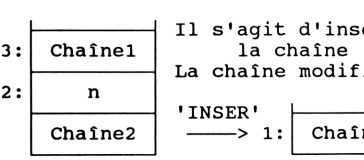
Il s'agit de détruire les caractères d'indice m à n dans la chaîne présente au niveau 3. La chaîne modifiée chaîne2 est obtenue au niveau 1.

Exercice 7 (###) : Nombre d'occurences d'une chaîne dans une autre.



Il s'agit de calculer le nombre de fois que 'Chaîne1' apparaît dans 'Chaîne2'. Le résultat, ici appelé 'Nombre', vaut 0 si 'Chaîne1' n'apparait pas dans 'Chaîne2'.

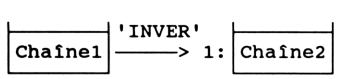
Exercice 8 (**) : Insérer une chaîne dans une autre.



Il s'agit d'insérer dans Chaîne1, à la position n, la chaîne 'Chaîne2'.
La chaîne modifiée 'Chaîne_3' est obtenue au niveau 1 de la pile.
Si n est supérieur à la taille de 'Chaîne1', l'insertion est en fait une concaténation de 'Chaîne2' à 'Chaîne1'.

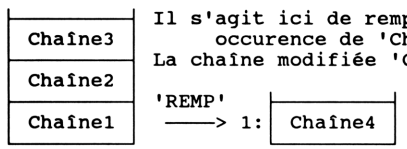
Si n est négatif ou nul, l'effet doit être le même que si n=1.

Exercice 9 (*) : Inverser l'ordre des caractères d'une chaîne.



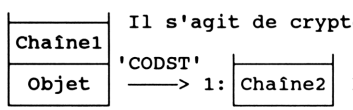
On prend la chaîne 'Chaîne1', et on la remplace par la chaîne 'Chaîne2' formée des mêmes caractères, mais dans l'ordre inverse.

Exercice 10 (***) : Recherche et remplacement dans une chaîne.



Il s'agit ici de remplacer, dans 'Chaîne3', chaque occurence de 'Chaîne2' par 'Chaîne1'. La chaîne modifiée 'Chaîne4' est obtenue au niveau 1 de la pile. Evidemment, si 'Chaîne2' n'est pas détectée dans 'Chaîne3' alors 'Chaîne4'='Chaîne3'.

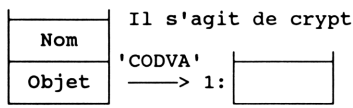
Exercice 11 (##) : Cryptage d'une chaîne de caractères.



Il s'agit de crypter (coder) une chaîne de caractères 'Chaîne1' en utilisant un code qui peut être un objet quelconque. La chaîne ainsi cryptée est obtenue au niveau 1 de la pile.

Ce codage doit être réversible:si on applique la même opération à 'Chaîne2', et avec le même code,on doit obtenir de nouveau la chaîne initiale 'Chaîne1'.

Exercice 12 (##) : Cryptage du contenu d'une variable.

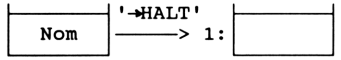


Il s'agit de crypter le contenu d'une variable dont le nom apparaît au niveau 2 , en utilisant un objet quelconque comme code.

Ce codage doit être réversible:si on applique la même opération à la même variable, et avec le même code, on doit retrouver le contenu initial de cette variable.

N.B: Dans ma solution, le contenu de la variable est transformé en chaîne de caractères codée. Pour assurer la réversibilité de cette opération, le contenu réel de la variable ne doit pas être une chaîne de caractères.

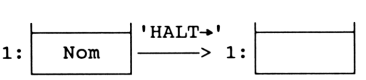
Exercice 13 (##) : Placer des **HALT** dans un programme.



On considère un programme stocké dans une variable dont le nom est placé au niveau 1. Il s'agit de modifier ce programme en plaçant un **HALT** après chaque caractère « (donc au minimum au début du programme, et plus généralement au début de chaque structure locale).

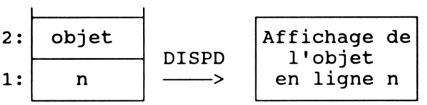
CHAINES DE CARACTERES: Exercices.

Exercice 14 (##) : Enlever les **HALT** d'un programme.



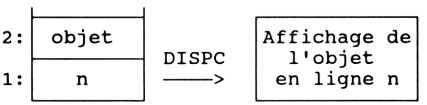
On considère un programme stocké dans une variable dont le nom est placé au niveau 1.
Il s'agit de modifier ce programme en retirant les instructions **HALT** qui pourraient s'y trouver (le programme **HALT→** défait donc ce qu'a fait le programme **→HALT**).

Exercice 15 (***) : Affichage cadré à droite d'un objet quelconque.



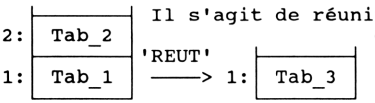
Il s'agit ici d'afficher un objet quelconque à partir de la ligne n, avec le schéma fonctionnel de l'instruction **DISP**, à ceci près que sur chaque ligne concernée, l'affichage doit être cadré à droite (alors que **DISP** cadre à gauche).

Exercice 16 (***) : Affichage centré d'un objet quelconque.



Il s'agit ici d'afficher un objet quelconque à partir de la ligne n, avec le schéma fonctionnel de l'instruction **DISP**, à ceci près que sur chaque ligne concernée, l'affichage doit être centré.

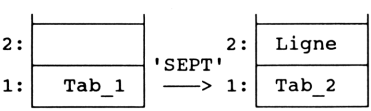
Exercice 17 (***) : réunion de deux tableaux.



Il s'agit de réunir deux tableaux 'Tab_1' et 'Tab_2', de manière à n'en plus faire qu'un, nommé ici 'Tab_3' et obtenu au niveau 1 de la pile.

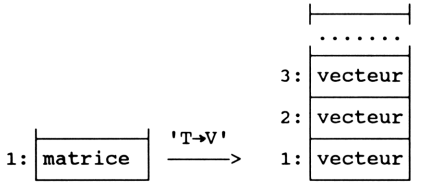
Le tableau 'Tab_3' doit être la superposition de 'Tab_2' et de 'Tab_1', comme si on avait posé 'Tab_2' sur 'Tab_1'.
'Tab_1' et 'Tab_2' peuvent être des vecteurs ou des matrices, de nombres réels ou complexes. Il est indispensable que ces deux tableaux aient le même nombre de colonnes.

Exercice 18 (***) : séparation d'un tableau en sa ligne supérieure et le reste du tableau.



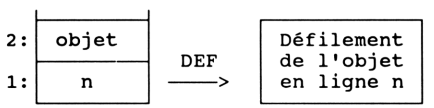
On part d'un matrice d'au moins 2 lignes, placée au niveau 1 . On en extrait la ligne supérieure, que l'on place au niveau 2.
Le reste du tableau est obtenu au niveau 1.

Exercice 19 (**) : répartir les lignes d'une matrice sur la pile.



Il s'agit ici de séparer la matrice située au niveau 1 en ses différentes lignes.
Les vecteurs obtenus sur la pile sont placés en conformité avec la position qui était la leur dans la matrice.
Au niveau 1, par exemple, on trouve la ligne inférieure.

Exercice 20 (**) : défilement d'un message à l'écran.



Il s'agit ici de faire défiler un objet quelconque sur la ligne n.
Le défilement est périodique, d'un caractère à la fois de la droite vers la gauche.

N.B: La longueur de l'objet peut excéder 22 caractères, mais cet objet ne doit pas être représenté sur plusieurs lignes.

LES TABLEAUX

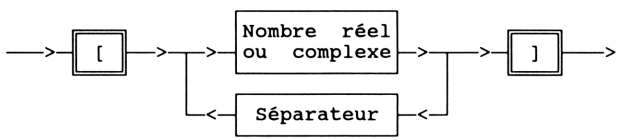
LA NOTION DE TABLEAU:

Un tableau est un groupement ordonné (en une ou deux dimensions) de nombres réels ou complexes.
On distingue deux sortes de tableaux: les vecteurs et les matrices.

* LES VECTEURS:

Un vecteur est un tableau à une seule ligne dont la syntaxe peut être représentée par le diagramme suivant:

VECTEUR:



Un vecteur est donc une succession ordonnée de nombres réels ou complexes, délimitée par les caractères [et].

Contrairement aux listes (qui ont quelques points communs avec les vecteurs), un vecteur doit contenir au moins un nombre, et les coefficients qui forment ce vecteur ne peuvent être des expressions, même si ces expressions renvoient de façon évidente à un résultat numérique.

Il est par exemple impossible de créer le vecteur: ['2^3' 'π'].

La taille d'un vecteur se mesure au nombre de coefficients, réels ou complexes, dont il est formé.
Il suffit que l'un de ces coefficients soit complexe pour que le vecteur tout entier soit considéré comme tableau complexe (même si ce nombre complexe n'a de complexe que l'écriture, comme par exemple (1,0) qui en fait représente le nombre réel 1).

A l'affichage, les différents coefficients sont séparés par un espace (sauf cas particulier lié à la notation polaire, comme on le verra plus loin).
Si le vecteur est complexe, tous les coefficients sont affichés sous forme complexe, même ceux éventuellement dont la partie imaginaire est nulle (c'est-à-dire qui sont réels).

Si le vecteur est trop long pour être totalement affichable sur une ligne, seule la partie affichable apparait.
Le mode multilignes ne s'applique donc pas aux vecteurs (il y a une exception: les vecteurs réels à 2 ou 3 coefficients).
On verra ci-après qu'il y a plusieurs façons de "visiter" un vecteur un peu long.

Exemples:

[1.23 -8.45 0 1991] est un vecteur réel à 4 coefficients.

[(2,-3) (1,1) (-4,5)] est un vecteur complexe.

[1] et [(1,0)] sont deux vecteurs distincts.
Le premier est réel, le second complexe (bien que d'un point de vue mathématique 1 et (1,0) représentent le même nombre).

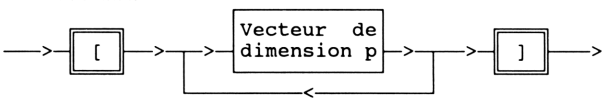
[1 (2,3) 0 -7] est un tableau complexe, qui pourra être entré sous cette forme, mais qui sera affiché sous la forme:
[(1,0) (2,3) (0,0) (-7,0)].

* LES MATRICES:

Une matrice à n lignes et p colonnes est une succession ordonnée de n vecteurs de même dimension p (n≥1, p≥1).

On peut la représenter par le schéma suivant:

MATRICE (p COLONNES):



La matrice est considérée comme complexe si l'un au moins des vecteurs est complexe , c'est-à-dire si l'un au moins des coefficients est complexe.

Il y a plusieurs façons de saisir une matrice, comme nous allons le voir plus loin.

A l'affichage, si le mode multiligne est actif et si la matrice est placée au niveau 1 de la pile, elle est affichée à raison d'une ligne d'écran par ligne de matrice (dans les limites de l'écran bien sûr, notamment pour ce qui concerne la longueur de chacun des vecteurs qui constituent la matrice).

Rappelons que le mode 'Multi-lignes' peut être activé ou désactivé en actionnant la touche correspondant au label **ML** dans le menu **MODES** (page 2).

Exemples:

[[1 55 13 -8]] est une matrice réelle (1 ligne et 4 colonnes).
On ne la confondra pas avec le vecteur [1 55 13 -8].

[[1][55][13][-8]] est une matrice réelle à quatre lignes et une colonne.

[[(1,2) (3,-4) (1,0)] est une matrice complexe à 3 lignes et 3
[(0,0) (0,1) (3,7)] colonnes (on dit une matrice carrée
[(4,0) (-2,0) (0,0)]] d'ordre 3).
A la saisie de cette matrice, les coefficients qui sont réels peuvent être saisis comme tels.

Remarque:
La distinction entre matrices et vecteurs est ici conforme aux conventions habituelles.
Il faut cependant noter que la commande **TYPE** de la HP48SX (dont le rôle est d'identifier les différentes catégories d'objets) ne différencie que les tableaux réels d'une part (type 3) et les tableaux complexes d'autre part (type 4), et ne permet pas de savoir si on a affaire à un vecteur ou à une matrice.
Pour le savoir, il suffit d'utiliser l'instruction **SIZE** (décrite plus loin).

N.B:
Dans toute la suite, pour plus de commodité, on appelle matrice de format {n,p} toute matrice à n lignes et p colonnes, et vecteur de format {n} tout vecteur ayant n coefficients.

SAISIR UN TABLEAU:

* EN LIGNE DE COMMANDE:

Pour saisir un vecteur en ligne de commande, on actionne la touche []. Le curseur vient se placer en superposition du caractère] terminal (en principe on est en mode "insertion").
On entre alors chacun des coefficients, que l'on sépare avec, indifféremment:
* un espace ou une virgule.
* un saut de ligne (touche <—┘).
* le caractère ∡ dans certains cas particuliers (en tout cas seulement pour les vecteurs réels à 2 ou 3 coefficients).

La saisie se termine par **ENTER** (ou par un appui sur la touche ▷ si on a d'autres objets à saisir après ce vecteur).

Pour saisir une matrice en ligne de commande, on actionne deux fois la touche [] (de manière à obtenir [[]] en ligne de commande, le curseur étant placé à hauteur du premier caractère] (et en principe en mode insertion).

On entre alors les coefficients de la première ligne, ce qui détermine la longueur de chacune de lignes.
On passe en suite par dessus le caractère] de fin de première ligne, (le curseur est donc placé sur le dernier]), et on entre les lignes suivantes.
Chacune des lignes suivantes peut être entrée entre [et], mais on peut se contenter d'entrer les termes des dernières lignes à la suite les uns des autres, sans les séparer par des [et].

Exemple: on veut entrer la matrice
```
[[ 3  -5 ]
 [ 6   0 ]
 [ 8   1 ]]
```

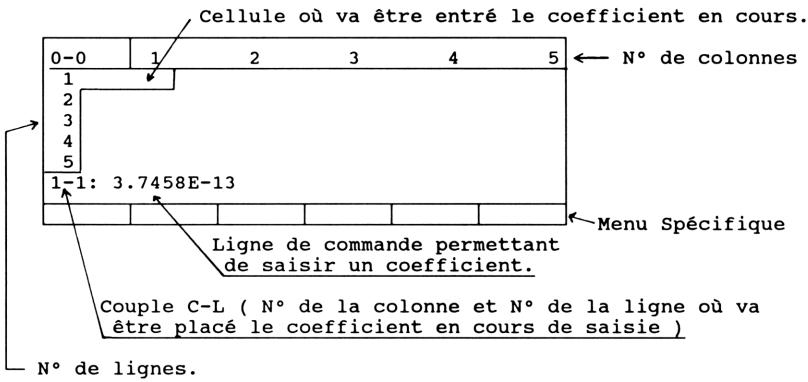
On actionne deux fois la touche [] On entre: 3 SPC 5 +/- On se déplace d'une position par ▷ et on entre 6 SPC 0 SPC 8 SPC 1 On fait ENTER et on obtient, au niveau 1 de la pile:	[[]] [[3 -5]] [[3 -5]6 0 8 1] [[3 -5] [6 0] [8 1]]

* L'ENVIRONNEMENT MATRIX-WRITER:

L'environnement **MATRIX-WRITER** est une application intégrée à la HP48SX et qui permet, entre autres, d'entrer des tableaux d'une manière très "conviviale".

On entre dans **MATRIX-WRITER** par un appui sur la touche **MATRIX.**

L'affichage de la pile est alors momentanément recouvert (la pile est évidemment toujours présente en arrière-plan), l'écran prenant l'aspect suivant:



On entre chaque coefficient dans la ligne de commande spécifique à cette application.

On valide par **ENTER** cette entrée d'un coefficient, dont la valeur est placée dans la "cellule" active.

Le "curseur de cellule" se déplace alors sur la case voisine (en principe la case située immédiatement à droite), la HP48SX attendant que vous entriez un nouveau coefficient en ligne de commande.

On entre ainsi les coefficients de la première ligne.

Quand celle-ci est terminée, un appui sur la touche ∇ permet de passer à la ligne suivante:
La HP48SX sait alors quel est le nombre de colonnes de votre matrice.
Quand vous validez par **ENTER** le dernier coefficient d'une ligne, le passage à la ligne suivante ~~suivante~~ est alors automatique.

On valide la matrice ainsi formée par un appui sur **ENTER** à un moment où la ligne de commande est vide (si une ligne n'est pas complète à ce moment là, elle est complétée par des 0).

A tout moment, on peut annuler la saisie en cours par un appui sur la touche **ON.**

On peut se déplacer dans la matrice en cours de saisie.
◁ ▷ △ ▽ déplacent le curseur de cellule d'une position dans la direction indiquée

Ce parcours peut être accéléré au moyen de la touche ╔═>.
╔═> ▷ : passe à l'extrémité droite de la ligne en cours.
╔═> ◁ : passe à l'extrémité gauche de la ligne en cours.
╔═> △ : passe à la première ligne dans la colonne en cours.
╔═> ▽ : passe à la dernière ligne dans la colonne en cours.

Si la matrice est de dimensions trop larges pour être totalement visible, l'écran est considéré comme une fenêtre permettant, comme indiqué ci-dessus, de "circuler" dans la matrice.

L'entrée dans MATRIX-WRITER s'accompagne de l'affichage d'un menu spécifique, de deux pages:

EDIT	VEC	←WID	WID→	GO→	GO↓

+ROW	-ROW	+COL	-COL	→STK	↑STK

EDIT: Passe en ligne de commande le contenu de la cellule pointée, pour édition de son contenu (utilisé pour modifier un terme déjà entré).

VEC : Si "Vec" est suivi d'un petit carré ▪,les tableaux à 1 ligne sont considérés comme des vecteurs, sinon ils sont considérés comme des matrices à une ligne.

←WID: Diminue la largeur des cellules, pour qu'une colonne de plus soit visible (on peut ainsi choisir d'en voir 1,2,3,4 ou 5). Si la taille d'un coefficient est trop grande pour qu'il soit totalement visible dans sa colonne, seul le début est affiché, le caractère ... signalant le dépassement.

WID→: Augmente la largeur des cellules, (une colonne de moins est visible)

GO→ et GO↓:
Un petit carré ▪ indique si une de ces deux commandes est active, et indique dans quel ordre la matrice est remplie (quand on saisit plusieurs termes à la suite par **ENTER**).
Si **GO→** est active, le sens de parcours dans la matrice est ligne par ligne (et sur une ligne de gauche à droite).
Si **GO↓** est active, le sens de parcours naturel est colonne par colonne (et dans une colonne de haut en bas).
On peut désactiver ces deux commandes. Dans ce cas, le curseur ne se déplace pas automatiquement après une saisie (il faut alors entrer les coefficients un par un et forcer le déplacement de la cellule avec les touches ◁ ▷ △ ▽).

+ROW: pour insérer une ligne de 0, à la position du curseur.

-ROW: pour détruire la ligne où se trouve le curseur.

+COL: pour insérer une colonne de 0, à la position du curseur.

-COL: pour détruire la colonne où se trouve le curseur.

→STK: copie le contenu de la cellule au niveau 1 de la pile.

↑STK: passe dans la "Pile Interactive" (on retrouve la matrice en cours de saisie après être sorti de ce mode par **ON** ou **ENTER**)

* REMARQUES SUR L'ENVIRONNEMENT MATRIX-WRITER:

- La saisie d'un terme en ligne de commande s'effectue en mode "programme".

  Il est donc possible d'entrer des commandes, des fonctions, bref d'entrer un "programme" permettant de calculer la valeur du terme en cours, cette valeur étant calculée (et placée dans la cellule pointée par le curseur de cellule), au moment où vous appuyez sur **ENTER**.

  Exemple: vous voulez entrer un coefficient égal à LN(1+√3/2).

  Vous pouvez entrer:
  3 √ 2 / 1 + LN puis **ENTER**
  ou encore:
  'LN(1+√3/2)' EVAL (ou **→NUM**) puis **ENTER**
  par contre **'LN(1+√3/2)' ENTER** conduirait à une erreur.
  Imaginons que la valeur LN(1+√3/2) intervienne souvent (isolée ou dans une expression) dans la valeur des différents termes de la matrice.
  On peut effectuer l'instruction **'LN(1+√3/2)' EVAL 'B' STO** (en ligne de commande avant d'entrer dans **MATRIX-WRITER**, ou bien même dans la ligne de commande de **MATRIX-WRITER** à condition de valider **'LN(1+√3/2)' EVAL 'B' STO** par un appui sur **ENTER**).
  Il suffit alors, dans la ligne commande de **MATRIX-WRITER**, d'effectuer:
  La séquence **B ENTER** pour placer la valeur LN(1+√3/2) dans la cellule en cours.
  La séquence **'B+SQ(B)' EVAL ENTER** pour entrer par ex. la quantité égale à $B+B^2$.

  Une autre possibilité, si des valeurs doivent être utilisées souvent, est de les placer sur la pile, avant d'entrer dans **MATRIX-WRITER**.
  Une fois dans **MATRIX-WRITER**, on sélectionne, quand c'est nécessaire, la "pile interactive" (par le label **↑STK**).
  Dans ce mode "pile interactive", il suffit alors (au moyen de la touche **ECHO**) de placer la ou les valeurs utiles en ligne de commande, avant de revenir à **MATRIX-WRITER**.

- Les coefficients du tableau sont affichés dans le mode en cours (notamment si c'est un mode **n FIX**).
  Par contre, si vous éditez l'un des coefficients en ligne de commande, il y est affiché en mode standard.

- Supposons que le tableau que vous allez saisir dans MATRIX-WRITER doive contenir au moins un coefficient complexe.
  Le premier coefficient à entrer doit être entré sous forme complexe, même s'il est réel (sinon il y aura une erreur quand vous tenterez ensuite d'entrer un coefficient complexe).

  Autrement dit, la façon dont on entre le premier coefficient (réel ou complexe) détermine si le tableau est un tableau réel ou un tableau complexe.
  Si le tableau est défini comme tableau complexe, vous pouvez alors entrer sous forme réelle les coefficients réels (ils seront automatiquement convertis à la forme complexe).

- Votre ligne de commande, dans **MATRIX-WRITER**, peut être saisie sur plusieurs lignes (l'affichage du tableau est momentanément supprimé).
  Tant que votre ligne de commande tient sur une seule ligne, vous activez la pile interactive par un appui sur la touche , et vous activez le menu **EDIT** par un appui sur .
  Plus généralement, vous pouvez sélectionner tel ou tel menu, en restant dans l'environnement **MATRIX-WRITER**.

- L'environnement **MATRIX-WRITER** est utile pour modifier (ou lire) un tableau existant. Voici comment on peut activer ce mode:

  Placer le tableau au niveau 1 , et actionner la touche ∇ (sauf si une ligne de commande est en cours de saisie) .
  (Un appui sur la touche **EDIT** placerait le tableau entier en ligne de commande).

  On obtient le même résultat avec ⌐=> ∇ si le tableau est représenté par son nom au niveau 1.
  (Un appui sur la touche **VISIT** placerait le tableau entier en ligne de commande).

  Le tableau étant sur la pile, passer dans l'environnement "Pile interactive", pointez le niveau où se trouve la matrice, et actionnez la touche de menu **VIEW**.
  (ou ⌐=> **VIEW** si le tableau est représenté par son nom).

LA NOTATION POLAIRE:

Ce paragraphe ne concerne que les vecteurs réels ayant 2 ou 3 coefficients.

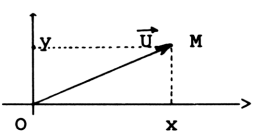
Pour simplifier nous les appelerons respectivement les 2D-vecteurs et les 3D-vecteurs.

*** Interprétation géométrique d'un 2D-vecteur:**

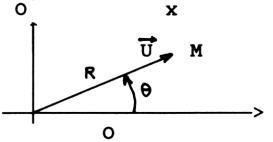
Un 2D-vecteur représente de façon unique un vecteur U ou un point M du plan (avec sa structure euclidienne orientée, et rapporté à un repère orthonormé direct)

Le vecteur U ou le point M sont déterminés de manière unique:

- Par le couple x, y de leurs coordonnées cartésiennes (ou rectangulaires):
  x = abscisse de M, de U.
  y = ordonnée de M, de U.



- Ou par un couple R, θ de coordonnées polaires:
  R = ± distance de M à l'origine
  = ± longueur du vecteur U.
  θ = angle polaire de M, de U.



Les relations entre ces deux systèmes de coordonnées sont:
x = R*cos(θ) et y = R*sin(θ).

* Déterminations principales des coordonnées (2D-vecteurs):

Il y a une infinité de couples R,θ représentant un même point:

- Si R est nul, θ est quelconque (on est à l'origine).
- Si un couple R,θ convient (avec R non nul) alors les autres couples qui conviennent sont:
  R , θ+2kπ
  -R, θ+π+2*k*π (k entier positif ou négatif)

Ici π est une mesure de l'angle plat, en radians: il faut le remplacer par 180 si on est en mode degrés.
et par 200 si on est en mode grades.

Néanmoins, pour tout point (vecteur) du plan, il existe un couple unique R,θ de coordonnées polaires le représentant , à condition d'imposer:
Si R=0, alors θ=0
Sinon R>0 et : $-\pi < \theta \leq \pi$ si on est en mode radians.
$-180 < \theta \leq 180$ si on est en mode degrés.
$-200 < \theta \leq 200$ si on est en mode grades.

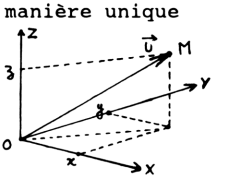
Ces valeurs de R et θ sont appelées déterminations principales des coordonnées polaires du point M (du vecteur U).

*** Interprétation géométrique d'un 3D-vecteur:**

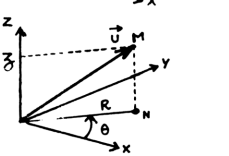
Un 3D-vecteur représente de façon unique un vecteur U ou un point M de l'espace à 3 dimensions (muni de sa structure euclidienne orientée, et rapporté à un repère orthonormé direct).

Le vecteur U ou le point M sont déterminés de manière unique

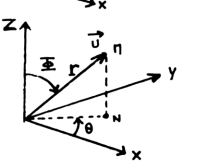
Par le triplet x,y,z de leurs coordonnées cartésiennes (ou rectangulaires):
x = abscisse de M, de U.
y = ordonnée de M, de U.
z = cote de M, de U



Par un triplet R,θ,z de coordonnées polaires (ou cylindriques):
R,θ = coordonnées polaires de la projection N de M sur le plan Oxy.
z = cote de M, de U.



Par un triplet r,θ,Φ de coordonnées sphériques:
r = distance à l'origine.
θ = angle polaire/Ox de la projection N de M sur le plan Oxy.
Φ = mesure de l'angle (Oz,OM)
On peut dire que θ est la longitude de M et que Φ est la colatitude de M.



Les relations existant entre ces systèmes de coordonnées sont:

x = R*cos(θ) y = R*sin(θ)	et	x = r*sin(Φ)*cos(θ) y = r*sin(Φ)*sin(θ) z = r*cos(Φ)

*** Déterminations principales des coordonnées (3D-vecteurs):**

S'il y a un seul triplet de coordonnées cartésiennes repérant un point ou un vecteur donné, il y a en fait une infinité de triplets de coordonnées polaires ou sphériques possibles.

Néanmoins il y a unicité du triplet (R,θ,z) de coordonnées polaires si on suppose:
R=θ=0 (on est sur l'axe Oz)
ou R>0 et $-\pi < \theta \leq \pi$ si on est en mode radians.
$-180 < \theta \leq 180$ si on est en mode degrés.
$-200 < \theta \leq 200$ si on est en mode grades.
On dit que ces R,θ,z sont les déterminations principales des coordonnées polaires du point M (du vecteur U).

De même il y a unicité du triplet (r,θ,Φ) de coordonnées sphériques, avec les conditions suivantes (on suppose ici que les angles sont exprimés en radians):
Si on est à l'origine: R=θ=Φ=0
Si on est sur l'axe Oz, avec z>0 : R>0, θ=0, Φ=0.
Si on est sur l'axe Oz, avec z<0 : R>0, θ=0, $\Phi=\pi$.
Si on n'est pas sur l'axe Oz : R>0, $-\pi<\theta\leq\pi$, $0<\Phi<\pi$.
(Si on est en mode degrés, remplacer π par 180, et si on est en mode grades, le remplacer par 200).
On dit que ces r,θ,Φ sont les déterminations principales des coordonnées sphériques du point M (du vecteur U).

*** Choix du mode degrés/radians/grades:**

Dès lors qu'on peut entrer, ou afficher, des 2D-vecteurs ou des 3D-vecteurs sous forme polaire ou sphérique, le mode de mesure des angles est important.

On a le choix entre les modes: radians, degrés, grades.
Le choix se fait dans le menu **MODES** (page 3):
DEG : sélectionne le mode degrés.
RAD : sélectionne le mode radians.
GRAD: sélectionne le mode grades.
DEG, **RAD**, **GRAD** sont des instructions de la HP48SX et peuvent être utilisées dans un programme et en ligne de commande.

La touche **RAD** permet également d'alterner le mode radians avec le mode degrés ou le mode grades (suivant lequel de ces deux modes a été le dernier sélectionné par l'un des moyens précédents).

Le mode actif est rappelé par un témoin en ligne d'état:
Témoin **RAD** pour le mode radians.
Témoin **GRAD** pour le mode grades.
Pas de témoin affiché en mode degrés.

Le choix du mode "degrés, radians ou grades" peut se faire en positionnant les indicateurs -17 et -18 (en tout cas le changement de mode par l'une des méthodes vues plus haut modifie ces indicateurs):
17 +/- SF (indicateur -17 armé): mode radians.
17 +/- CF (indicateur -17 désarmé), deux cas:
18 +/- SF (indicateur -18 armé) : mode grades.
18 +/- CF (indicateur -18 désarmé): mode degrés.

*** Choix du mode d'affichage:**

Que l'on ait entré un 2D-vecteur ou un 3D-vecteur sous forme rectangulaire polaire, ou sphérique, il est affiché selon le mode en cours sur la HP48SX.

- Pour définir le mode d'affichage des 2D ou 3D-vecteurs, on peut utiliser les entrées suivantes du menu **MODES**, ou du menu **MTH VECTR**.
  XYZ: mode rectangulaire (pas d'indicateur affiché).
  R∠Z: mode polaire (indicateur **R∠Z** affiché).
  R∠∠: mode sphérique (indicateur **R∠∠** affiché).
  Le petit carré ▪ après l'un des trois labels désigne le mode en cours.

- On peut sélectionner le mode d'affichage des 2D ou 3D-vecteurs de la manière suivante:
  16 +/- CF (désarmer le flag -16): mode rectangulaire.
  16 +/- SF (armer le flag -16):
  - mode polaire si le flag -15 est désarmé (**15 +/- CF**)
  - mode sphérique s'il est armé (par **15 +/- SF**)

  Cette méthode est la seule qui soit programmable.

- La touche **POLAR** permet d'alterner le mode rectangulaire (pas d'indicateur affiché), et:
  - Soit le mode polaire (indicateur **R∠Z** affiché).
  - Soit le mode sphérique (indicateur **R∠∠** affiché).

  Tout dépend, entre ces deux derniers modes, quel a été le dernier sélectionné par l'une des méthodes précédentes.

- Pour un 2D-vecteur, il n'y a aucune différence entre le mode polaire et le mode sphérique.

*** Affichage d'un 2D ou d'un 3D-vecteur:**

Quelque que soit le mode en cours, les 2D-vecteurs et les 3D-vecteurs sont toujours stockés, en mémoire, sous forme rectangulaire.
Les 2D-vecteurs et 3D-vecteurs sont affichés en 'Multi-lignes' (si ce mode est actif: voir dans le menu **MODES**, page 2, le label **ML**).
Un tel vecteur occupe donc 1,2 ou 3 lignes à l'écran, de manière à être totalement visible.

Ce qui a été dit sur l'affichage des réels continue à s'appliquer aux composantes d'un 2D ou 3D-vecteur (que l'on soit en mode rectangulaire, polaire ou sphérique), notamment pour ce qui concerne les modes **STD**, **FIX**, **SCI** et **ENG**.

A l'affichage, les composantes d'un 2D-vecteur ou d'un 3D-vecteur sont séparées par un espace (éventuellement un saut de ligne).
De plus:
En mode polaire: la deuxième composante (angle polaire ou longitude) est précédé du caractère ∠ .
En mode sphérique: la deuxième et la troisième composante sont chacune précédées du caractère ∠ .

Le changement de mode rectangulaire/polaire/sphérique affecte tous les 2D ou 3D-vecteurs affichés, qu'ils soient isolés, ou apparaissent dans un programme ou une liste.

Exemple:
Considérons le 3D-vecteur [2 2 1].
Il peut être affiché de l'une des manières suivantes:
(on suppose que le mode d'affichage des réels est **2 FIX**)

Mode rectangulaire: [2.00 2.00 1.00]

Mode polaire/degrés: [2.83 ∠45.00 1.00]
Mode polaire/radians: [2.83 ∠0.79 1.00]
Mode polaire/grades: [2.83 ∠50.00 1.00]

Mode sphérique/degrés: [3.00 ∠45.00 ∠70.53]
Mode sphérique/radians: [3.00 ∠0.79 ∠1.23]
Mode sphérique/grades: [3.00 ∠50.00 ∠78.37]

*** Saisie d'un 2D-vecteur:**

Un 2D-vecteur peut être saisi en mode rectangulaire, c-à-d sous la forme [x y].

Il peut être également saisi sous forme polaire, c-à-d en entrant un couple R, θ de coordonnées polaires.
Pour celà on procède comme précédemment, à ceci près qu'on fait précéder l'angle θ du caractère ∠ .

Le vecteur est alors affiché suivant le mode en cours, et en utilisant les déterminations principales des coordonnées.
En mémoire, il est stocké sous forme rectangulaire.

*** Saisie d'un 3D-vecteur:**

Un 3D-vecteur peut être saisi en mode rectangulaire, c-à-d sous la forme [x y z].

Il peut être également saisi sous forme polaire, c-à-d en entrant un triplet R, θ, z de coordonnées polaires.
Pour celà on procède comme précédemment, à ceci près qu'on fait précéder l'angle θ du caractère ∠ .

Il peut être également saisi sous forme sphérique, c-à-d en entrant un triplet r, θ, Φ de coordonnées sphériques.
Pour celà on procède comme précédemment, à ceci près qu'on fait précéder l'angle θ , et l'angle Φ, du caractère ∠ .

Le vecteur est alors affiché suivant le mode en cours, et en utilisant les déterminations principales des coordonnées.
En mémoire, il est stocké sous forme rectangulaire.

*** MATRIX-WRITER et les modes polaire ou sphérique:**

Quelque soit le mode en cours, la saisie d'un 2D-vecteur ou d'un 3D-vecteur,dans l'environnement **MATRIX-WRITER**, ne peut se faire que sous forme rectangulaire (idem pour la lecture et/ou la modification d'un vecteur existant).
On utilisera donc de préférence, si le mode polaire ou sphérique est actif, la saisie ou l'édition en ligne de commande.

OPERATIONS ARITHMETIQUES SUR LES TABLEAUX:

Un certain nombre d'opérations arithmétiques sont accessibles directement au clavier: leur emploi est conditionné par le respect de certaines règles mathématiques (en ce qui concerne le format des tableaux sur lesquelles on opère).

+ Addition de deux tableaux:

```
2: [ A ]
                 +
1: [ B ]  ---->  1: [ A + B ]
```

La commande + permet d'ajouter terme à terme deux tableaux ayant exactement le même format.Si l'un d'eux est un tableau complexe, alors leur somme est un tableau complexe.

Exemples:

```
2: [[ 0  1  8  3  1 ]
     [ 4  2  9  4  2 ]]
1: [[ 4  6  0  1  1 ]
     [ 8  7  7  1  0 ]]
                 ---->
2:
1: [[ 4  7  8  4  2 ]
     [12  9 16  5  2 ]]

2: [ (4,3) (5,0) (6,7) ]
1: [ 2  1  5 ]
                 +
                 ---->
2:
1: [ (6,3) (6,0) (11,7) ]

2: [ 1 ∡60 3 ]
1: [ 2 ∡30 1 ]
                 +
                 ---->
1: [ 2.91 ∡39.90 4.00 ]
```

On est ici en mode polaire/degrés et en mode 2 FIX.

- Soustraction de deux tableaux:

```
2: [ A ]
                 -
1: [ B ]  ---->  1: [ A - B ]
```

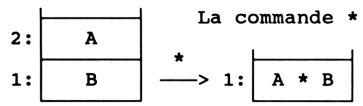
La commande - permet de retrancher terme à terme 2 tableaux ayant exactement le même format.Si l'un d'eux est un tableau complexe, alors leur différence est un tableau complexe.

Exemples:

```
2: [[ 1  5  2  3  4 ]
     [ 0  1  8  3  1 ]
     [ 4  2  9  4  2 ]]
1: [[ 0  1  9  5  2 ]
     [ 4  6  0  1  1 ]
     [ 8  7  7  1  0 ]]
                 -
                 ---->
2:
1: [[ 1  4 -7 -2  2 ]
     [-4 -5  8  2  0 ]
     [-4 -7  2  3  2 ]]

2: [ (4,3) (5,0) (6,7) ]
1: [ 2  1  5 ]
                 -
                 ---->
2:
1: [ (2,3) (4,0) (1,7) ]
```

Produit d'une matrice par un tableau:



La commande * permet de multiplier une matrice A par un tableau B.
Le produit de deux tableaux est effectué selon les règles mathé-matiques habituelles (voir plus loin).

Si l'un au moins des deux tableaux A ou B est un tableau complexe, alors le produit A*B est un tableau complexe.

Il y a deux cas suivant que B est une matrice ou un vecteur:
Supposons que A soit une matrice de format {n,p}.

* Si B est une matrice, elle doit être de format {p,q} et le résultat est une matrice de format {n,q}.

* Si B est un vecteur, il doit être de format {p} et le résultat est un vecteur de format {n}.

Rappels:

Dans le cas où B est une matrice:

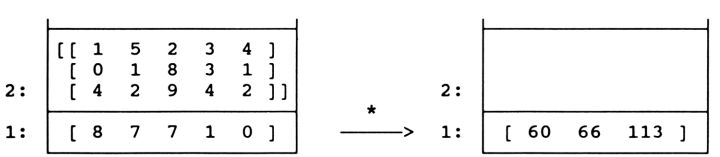
Le coefficient situé à l'intersection de la ligne i et de la colonne j du produit A*B est obtenu en ajoutant les produits terme à terme des coefficients de la i-ème ligne de A par ceux de la j-ème colonne de B.

Dans le cas où B est un vecteur:

Le i-ème coefficient du vecteur A*B est obtenu en ajoutant les produits terme à terme des coefficients de la i-ème ligne de A par ceux du vecteur B (Si B est un 2D ou un 3D-vecteur, il s'agit des coefficients rectangulaires).

Rappelons enfin que dans le cas de deux matrices carrées A et B de même format, les deux produits AB et BA sont possibles mais donnent en général des résultats différents.

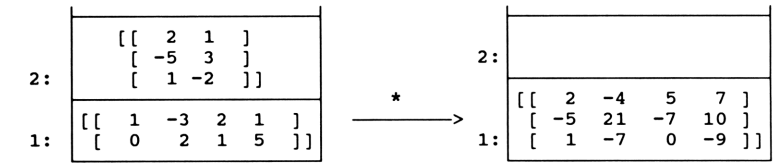
Exemple:



Sur cet exemple, on a effectué le produit d'une matrice A de format {3 5} par un vecteur V de format {5}.

On a ainsi obtenu un vecteur W=A*V de format {3}, dont le 3-ème coefficient, par exemple, est obtenu par le calcul:
113 = 4*8 + 2*7 + 9*7 + 4*1 + 2*0

Exemple:

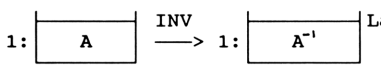


Sur cet exemple, on a effectué le produit d'une matrice de format {3 2} par une matrice de format {2 4}. On a ainsi obtenu une matrice de format {3 4}.

Pour prendre un exemple, le coefficient 10 du produit (situé en ligne 2 et en colonne 4) a été obtenu par le calcul: 10 = (-5)*1 + 3*5 (somme des produits terme à terme des coefficients -5,3 de la 2ème ligne de A, par les coef--ficients 1,5 de la 4-ème colonne de B).

INV

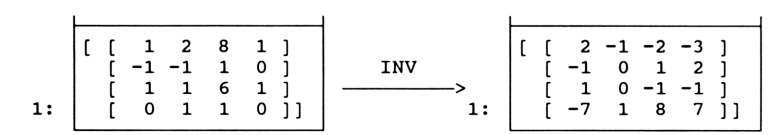
Inverse d'une matrice carrée:



La commande **INV** (directement acces--sible au clavier avec la touche 1/x) permet de calculer l'inverse d'une matrice carrée A.

Bien entendu, et sous peine de provoquer une erreur ("Infinite Result"), la matrice carrée A doit être inversible, c'est à dire de déterminant non nul.Le résultat est alors la matrice carrée de même format que A, et dont le produit commutatif avec A est égal à la matrice identité d'ordre n.

Exemples:



On vérifie que les produits AB et BA des deux matrices ci-dessus sont égaux à la matrice identité d'ordre 4.

Remarque:

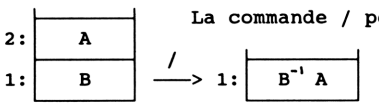
L'exemple ci-dessus est assez particulier. En général, l'inversion d'une matrice carrée A s'accompagne d'erreurs d'arrondi et d'une division par le déterminant de A (qui lui aussi est sujet à de telles erreurs).

Sur l'exemple ci-dessus, le déterminant de la matrice initiale est égal à 1, et la calculatrice l'obtient sans erreur aucune. C'est pourquoi les coefficients de l'inverse sont encore entiers.

Voir en exercice une utilisation de l'instruction **RSD** pour obtenir des calculs plus précis.

/ Division d'un tableau par une matrice carrée: (÷ au clavier)

```
2: | A |
1: | B |  ---/---> 1: | B⁻¹ A |
```



La commande / permet de calculer le produit de l'inverse B de la matrice carrée B par le tableau A.
Evidemment, la matrice carrée B doit être inversible.

Si l'un au moins des deux tableaux A ou B est un tableau complexe, alors le résultat est un tableau complexe.

Supposons que B soit une matrice carrée de format {n,n}.
* Si A est une matrice, elle doit être de type {n,p} et alors le résultat est une matrice de format {n,p}.
* Si A est un vecteur, il doit être de format {n} et alors le résultat est un vecteur de format {n}.

Dans tous les cas, le résultat (s'il est calculable) a le format du tableau A.

Exemples:

```
2: | [ -6  5  0 -1 ] |
   | [[  1  2  8  1 ] |
   |  [ -1 -1  1  0 ] |
   |  [  1  1  6  1 ] |
1: |  [  0  1  1  0 ]] |  ---/---> 1: | [ -14  4 -5  40 ] |
```

```
   | [ [ -6  3  2 ] |
   |   [  5  1 -5 ] |
   |   [  0  4  6 ] |
2: |   [ -1  7  0 ] ] |
   | [[  1  2  8  1 ] |
   |  [ -1 -1  1  0 ] |
   |  [  1  1  6  1 ] |
1: |  [  0  1  1  0 ]] |  ---/---> 1: | [ [ -14 -24 -3 ] |
                                      |   [   4  15  4 ] |
                                      |   [  -5  -8 -4 ] |
                                      |   [  40  61 29 ] ] |
```

Le premier exemple ci-dessus peut être interprété comme la résolution du système linéaire:

$$\begin{cases} x + 2y + 8z + t = -6 \\ -x - y + z = 5 \\ x + y + 6z + t = 0 \\ y + z = -1 \end{cases}$$

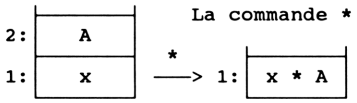
et ce système admet pour solution unique le quadruplet: (x,y,z,t) = (-14,4,-5,40).

Le 2ème exemple effectue la résolution simultanée des 3 systèmes linéaires (ayant la même partie "premiers membres"):

$$(I)\begin{cases} x + 2y + 8z + t = -6 \\ -x - y + z = 5 \\ x + y + 6z + t = 0 \\ y + z = -1 \end{cases} \quad (II)\begin{cases} \ldots = 3 \\ \ldots = 1 \\ \ldots = 4 \\ \ldots = 7 \end{cases} \quad (III)\begin{cases} \ldots = 3 \\ \ldots = 1 \\ \ldots = 4 \\ \ldots = 7 \end{cases}$$

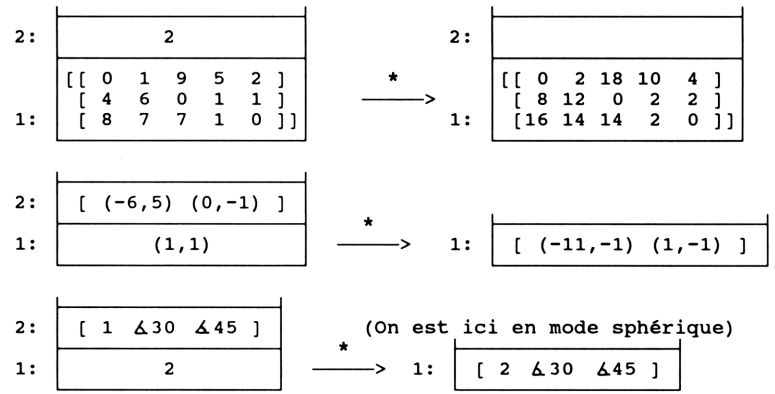
dont les solutions sont:
(I): (-14,4,-5,40) (II): (-24,15,-8,61) (III): (-3,4,-4,29)

* Produit d'un tableau par un scalaire:

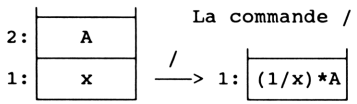


La commande * permet de calculer le produit d'un tableau A et d'un scalaire x.
Le résultat est un tableau de même format que A.
Si A est un tableau complexe, ou si x est complexe, alors x*A est un tableau complexe. De plus la position respective de x et de A sur la pile est sans importance.

Exemples:

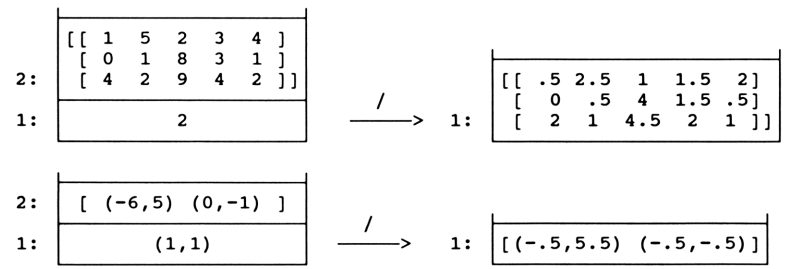


/ Division d'un tableau par un scalaire: (÷ au clavier)



La commande / permet de calculer le quotient d'un tableau A par un scalaire x.
Le résultat est un tableau de même format que A.
Si A est un tableau complexe, ou si x est complexe, le résultat est un tableau complexe.

Exemples:



NEG Opposé d'un tableau:

```
1: [ A ]  NEG ---> 1: [ - A ]
```

La commande **NEG** (accessible au clavier au moyen de la touche **+/-**) transforme le tableau A en son opposé. Le résultat -A est un tableau de même format que A, dont les termes sont les opposés respectifs de ceux de A.

Exemples:

```
1: [[ 0 -1  9 -5 -2 ]
    [ 4  6  0 -1  1 ]
    [ 8 -7 -7  1  0 ]]  NEG ---> 1: [[ 0  1 -9  5  2 ]
                                     [-4 -6  0  1 -1 ]
                                     [-8  7  7 -1  0 ]]

1: [ (1,-3) (-5,7) ]  NEG ---> 1: [ (-1,3) (5,-7) ]

1: [ 2 ∡75 ∡30 ]  NEG ---> 1: [ 2 ∡-125 ∡170 ]
```

Sur l'exemple ci-dessus, on est en mode sphérique/grades.

```
1: [ 1 ∡45  10 ]  NEG ---> 1: [ 1 ∡-135  -10 ]
```

Sur l'exemple ci-dessus, on est en mode polaire/degrés.

SQ Carré d'une matrice carrée:

```
1: [ A ]  SQ ---> 1: [ A² ]
```

La commande **SQ** (accessible au clavier au moyen de la touche x^2) , calcule le carré A*A de la matrice carrée A. Le résultat est une matrice carrée de même format que A, et de même type (réel ou complexe).

Exemples:

```
1: [ [  1  2  8  1 ]
     [ -1 -1  1  0 ]
     [  1  1  6  1 ]
     [  0  1  1  0 ]]  SQ ---> 1: [ [  7  9 59  9 ]
                                   [  1  0 -3  0 ]
                                   [  6  8 48  7 ]
                                   [  0  0  7  1 ]]

1: [[ (1,1) (0,1) (1,0) ]
    (1,2) (2,1) (1,3) ]
    (0,3) (2,2) (3,1) ]]  SQ ---> 1: [[ (-2,6)   (0,5)   (1,3)  ]
                                      [ (-10,11) (-3,13) (0,19) ]
                                      [ (-8,18)  (3,14)  (4,17 ) ]]
```

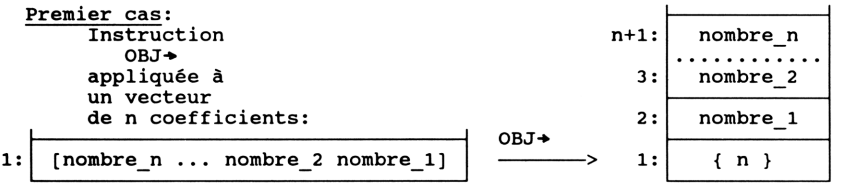
LE MENU "PRG OBJ" ET LES TABLEAUX:

Le menu **PRG OBJ** contient quelques instructions applicables aux tableaux: **OBJ→ ,→ARRY, R→C, C→R, SIZE, PUT, GET, PUTI, GETI.** Toutes sauf **→ARRY** s'appliquent à d'autres catégories d'objets.

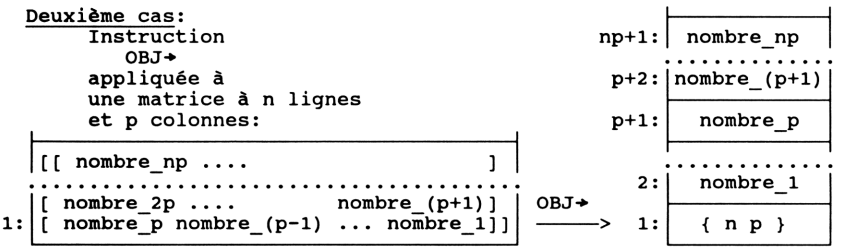
OBJ→ **OBJ→** libère les éléments d'un tableau en les plaçant sur la pile, comme indiqué ci dessous.

Si dans le tableau, un élément a précède immédiatement un élément b (en adoptant l'ordre de gauche à droite sur une ligne, et d'une ligne à la ligne inférieure), alors a sera placé sur la pile immédiatement au-dessus de b.
Le format du tableau ainsi 'dispersé' est obtenu au niveau 1 de la pile (on obtient une liste d'un entier pour un vecteur, de deux entiers pour une matrice).

Premier cas:
Instruction OBJ→ appliquée à un vecteur de n coefficients:



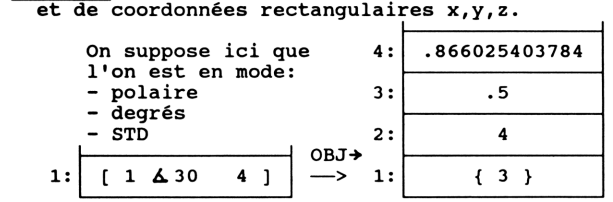
Deuxième cas:
Instruction OBJ→ appliquée à une matrice à n lignes et p colonnes:



Remarques:
OBJ→, appliquée à un 2D-vecteur ou à un 3D-vecteur, donne les composantes rectangulaires de ce vecteur, indépendamment du mode d'affichage en cours.
L'instruction **ARRY→** a le même effet que **OBJ→** sur un tableau. Néanmoins, **ARRY→** doit être tapé en toutes lettres et ne s'applique qu'à des tableaux.

Exemple: Soit le vecteur de coordonnées polaires 1,30°,4 et de coordonnées rectangulaires x,y,z.

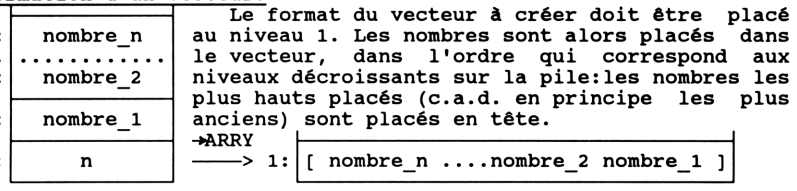
On suppose ici que l'on est en mode:
- polaire
- degrés
- STD



→ARRY

→ARRY permet de constituer un tableau formé de nombres pris sur la pile, de la manière suivante:

Formation d'un vecteur:



Le format du vecteur à créer doit être placé au niveau 1. Les nombres sont alors placés dans le vecteur, dans l'ordre qui correspond aux niveaux décroissants sur la pile:les nombres les plus hauts placés (c.a.d. en principe les plus anciens) sont placés en tête.

Remarques:

Le format peut aussi bien être donné sous la forme de l'entier n que sous la forme de la liste { n } réduite à cet entier (le résultat est le même).

n doit être au moins égal à 1.

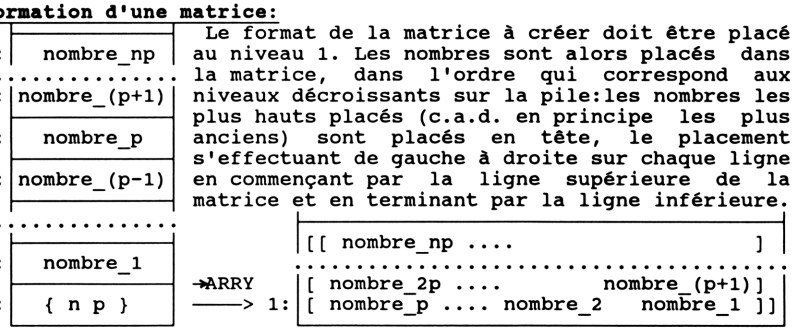
Si tous les objets présents sur la pile sont des nombres réels ou complexes, la séquence **DEPTH →ARRY** place tous ces nombres dans un seul vecteur.

Si l'un au moins des nombres pris sur la pile est un nombre complexe, le vecteur ainsi formé est de type complexe.

n n'est pas nécessairement un entier: dans le cas où n est réel, il est arrondi à l'entier le plus proche.

Si n=2 ou 3, le nD_vecteur obtenu est affiché en fonction du mode en cours, mais les nombres pris sur la pile sont ses composantes rectangulaires.

Formation d'une matrice:



Le format de la matrice à créer doit être placé au niveau 1. Les nombres sont alors placés dans la matrice, dans l'ordre qui correspond aux niveaux décroissants sur la pile:les nombres les plus hauts placés (c.a.d. en principe les plus anciens) sont placés en tête, le placement s'effectuant de gauche à droite sur chaque ligne en commençant par la ligne supérieure de la matrice et en terminant par la ligne inférieure.

Sur cet exemple, on crée une matrice à n lignes et à p colonnes.

Remarques:

Le format de la matrice est obligatoirement une liste { n p } de deux éléments n et p (n = lignes, p = colonnes).

n et p doivent être au moins égaux à 1.

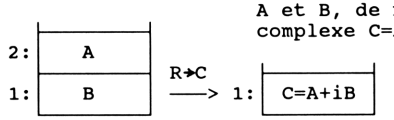
Si l'un au moins des nombres pris sur la pile est un nombre complexe, la matrice ainsi formée est de type complexe.

n et p ne sont pas nécessairement des entiers: dans le cas où n ou p est réel, il est arrondi à l'entier le plus proche.

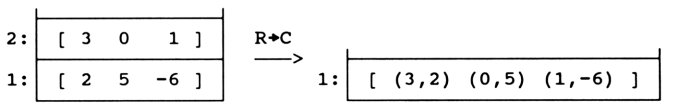
OBJ→ et **→ARRY** sont réciproques l'une de l'autre (utile pour rattraper une utilisation involontaire).

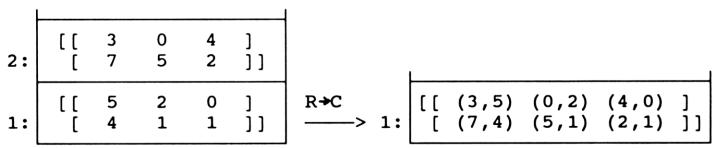
R→C

L'instruction **R→C** permet de regrouper deux tableaux réels A et B, de même format, en un seul tableau complexe C=A+iB, de même format que A et B. Si α et ß sont deux réels situés respectivement dans A et B, en une même position donnée , le coefficient de C correspondant à cette position est le nombre complexe (α,ß).



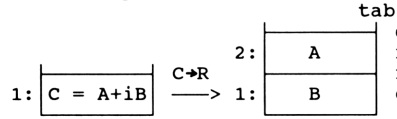
Exemples:



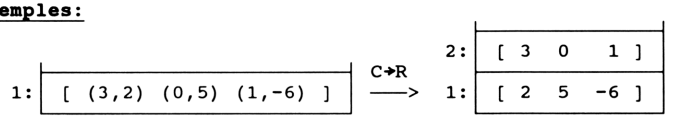


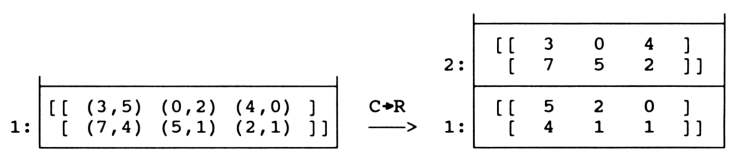
C→R

C→R permet de décomposer un tableau complexe C en deux tableaux réels A et B ,de même format que C, dont les coefficients sont respectivement égaux aux parties réelles et aux parties imaginaires des coefficients de C.



Exemples:



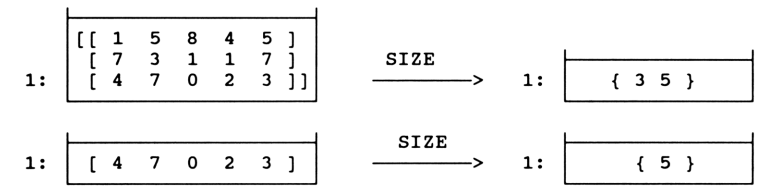


Remarque:
Il est clair que les deux instructions R→C et C→R sont réciproques l'une de l'autre.

SIZE **SIZE** permet de connaître la taille d'un tableau.

La taille d'une matrice est une liste { lig col }, où lig est le nombre de lignes et col le nombre de colonnes.
La taille d'un vecteur est une liste { coeff }, où coeff est le nombre de coefficients du vecteur.
Aucune différence n'est faite entre les tableaux réels et les tableaux complexes.

Exemples:



Attention:
L'instruction **SIZE** fonctionne quand on place une expression niveau 1 de la pile, et en particulier si on y place un nom de variable (le résultat donné par **SIZE** dans ce dernier cas est 1).
Pour connaître la taille d'un tableau par **SIZE**, et si ce tableau est stocké dans une variable, on veillera donc à placer le tableau sur la pile, et pas le nom de cette variable.

PUT L'instruction **PUT** place un nombre dans un tableau. Il y a deux schémas fonctionnels possibles.
'Nombre' est le nombre à placer, 'Position' est l'endroit où on veut le placer.

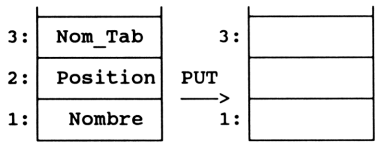
Dans le premier cas:
On indique le nom de la variable contenant le tableau ('Nom_Tab' ci dessous).
Le placement s'effectue en mémoire (et les trois arguments disparaissent de la pile).

Dans le deuxième cas:
On place au niveau 3 le tableau à modifier.
Le tableau modifié est alors obtenu au niveau 1.

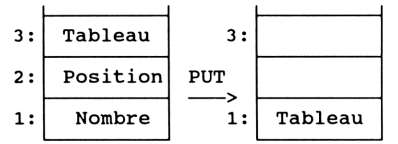
Important:
L'instruction **PUT** ne permet pas de placer un nombre complexe dans un tableau réel (même si ce nombre a une partie imaginaire nulle).
On peut par contre placer un réel x dans un tableau complexe (ce réel étant transformé en son équivalent complexe (x,0)).
D'autre part si on place un réel dans un 2D-vecteur ou un 3D-vecteur à l'aide de **PUT** (ou **PUTI**, voir plus loin), ce réel est considéré comme une composante rectangulaire, quelque soit le mode d'affichage (polaire ou sphérique).

LES TABLEAUX.

Premier cas:



Deuxième cas:



'Position' détermine l'endroit où placer 'Nombre' dans le tableau:

1) Si le tableau est un vecteur de format { n }:
'Position' est un entier k (1≤k≤n) ou est une liste {k} réduite à un tel entier k.

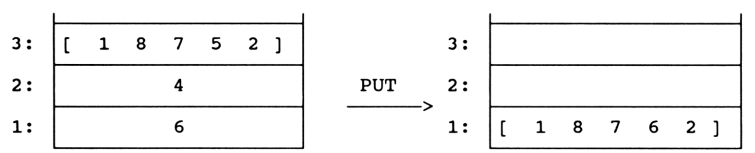
2) Si le tableau est une matrice de format { n p }:
'Position' peut être:

* Une liste { lig col } de 2 entiers 'lig' et 'col', ce qui signifie que l'on veut placer le nombre à l'intersection de la ligne 'lig' (1≤lig≤n) et de la colonne 'col' (1≤col≤p).

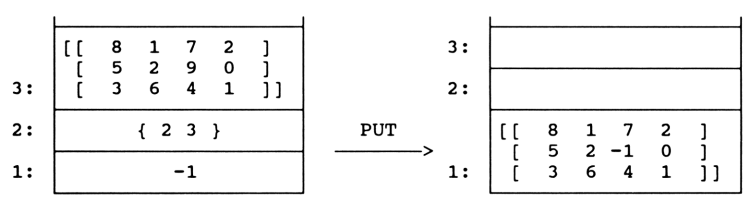
* Un entier k désignant une position dans la matrice:
L'ordre des positions dans une matrice est défini par un parcours de gauche à droite sur chaque ligne, et d'une ligne à la ligne inférieure.

Ainsi 1≤ k ≤np, et avec les notations précédentes, pour les "matheux":
k = (lig-1)*p + col.
lig = floor((k-1)/p) + 1,
col = ((k-1) modulo p) + 1.

Exemples:(en se limitant au cas où le tableau est physiquement présent sur la pile):



Sur cet exemple, le nombre 6 a été placé en 4-ème position dans le vecteur. On aurait pu également placer {4} au niveau 2 de la pile, pour indiquer la position.



Sur cet exemple, le nombre -1 a été placé à l'intersection de la ligne 2 et de la colonne 3 . Pour définir cette position, on aurait tout aussi bien placer l'entier 7 au niveau 2 de la pile.

GET

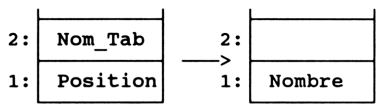
L'instruction **GET** permet de lire un élément d'un tableau. Il y a deux schémas fonctionnels possibles. 'Position' désigne où se trouve dans le tableau le nombre visé ('Nombre').

Dans le premier cas:
On indique le nom de la variable contenant le tableau ('Nom_Tab' ci dessous).

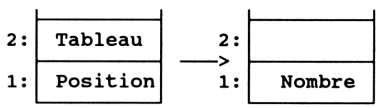
Dans le deuxième cas:
On place au niveau 2 le tableau où se trouve le nombre à isoler.

Dans les deux cas le nombre obtenu est placé au niveau 1.

Premier cas:

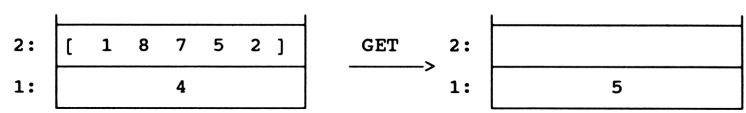


Deuxième cas:

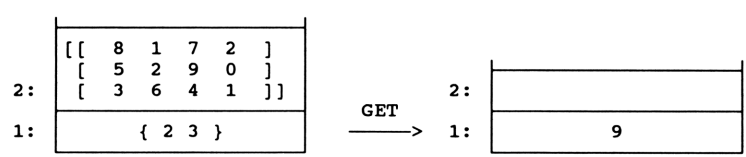


'Position' précise l'endroit où trouver 'Nombre' dans le tableau. (Voir à ce sujet la description de l'instruction **'PUT'**)

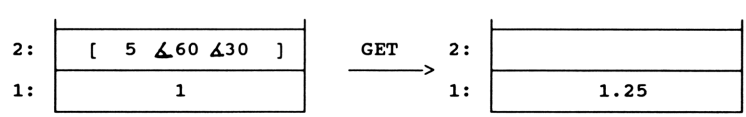
Exemples:(en se limitant au cas où le tableau est physiquement présent sur la pile):



Sur cet exemple, le nombre 5 a été trouvé en 4ème position dans le vecteur. On aurait pu également placer {4} au niveau 1 de la pile, pour indiquer la position.



Sur cet exemple, le nombre 9 a été trouvé à l'intersection de la ligne 2 et de la colonne 3 . Pour définir cette position, on aurait tout aussi bien placer l'entier 7 au niveau 1 de la pile.



Ici, on est en mode sphérique/degrés.
Le réel obtenu correspond à la 1ère composante rectangulaire du vecteur, à savoir: 5*sin(30°)*cos(60°) = 5/4 = 1.25

PUTI

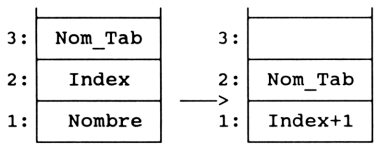
L'instruction **PUTI** permet de placer plusieurs nombres à des positions successives dans un tableau.
Chaque appel de **PUTI** place un de ces nombres.
L'intérêt de cette instruction est qu'elle incrémente elle même la position où placer le nombre suivant.
Il y a deux schémas fonctionnels possibles. 'Nombre' est le nombre à placer, 'Index' est l'endroit où on veut le placer.

Dans le premier cas, on indique le nom de la variable contenant le tableau ('Nom_Tab' ci dessous). Le placement s'effectue en mémoire.

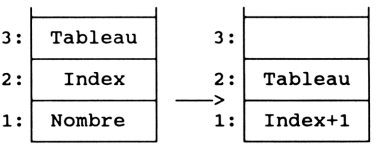
Dans le deuxième cas, on place au niveau 3 le tableau à modifier. Le tableau modifié est alors obtenu au niveau 2.

Important: L'instruction **PUTI** ne permet pas de placer un nombre complexe dans un tableau réel (même si ce nombre a une partie imaginaire nulle).
On peut par contre placer un réel x dans un tableau complexe (ce réel étant transformé en son équivalent complexe (x,0)).

Premier cas:



Deuxième cas:



Ci-dessus, 'Index+1' désigne la position qui suit 'Index' dans le tableau.

Si 'Index' représente la dernière position possible dans le tableau, alors 'Index+1' nous ramène à la première position dans le tableau (ce qui permet de reprendre le placement de nombres en début de la liste).

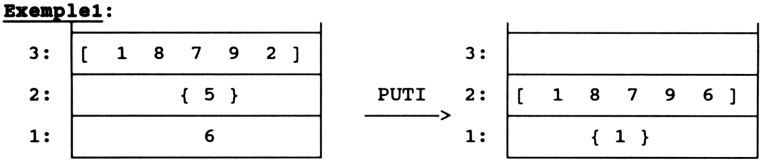
On voit que des appels successifs de **PUTI** permettent de placer des éléments à des positions successives dans un tableau.

Là encore, la position dans le tableau peut être indiquée de deux manières différentes (c'est vous qui choisissez):
* Par un entier k ou par une liste { k } réduite à cet entier, dans le cas d'un vecteur.
* Par un entier k, ou par une liste { lig col } de deux entiers ('lig' pour le numéro de ligne, 'col' pour le numéro de colonne), dans le cas d'une matrice.

Il faut remarquer que l'instruction **PUTI** renvoie la position suivante en utilisant le format que vous aviez choisi.

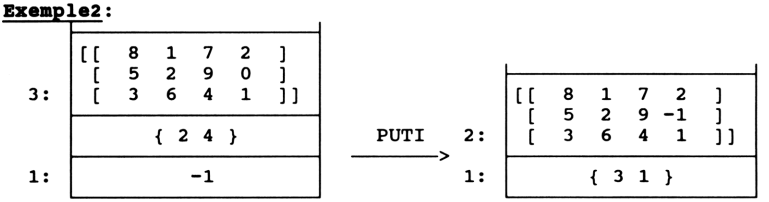
Remarque:
Si l'indicateur N°**-64** est armé (valeur = 1), **PUTI** vient à l'instant de nous ramener en début de tableau.
Sinon il est désarmé (à zéro).

Exemple1:



Sur cet exemple, le nombre 6 a été placé en 5ème et dernière position dans le vecteur.
Le fait qu'on soit revenu en début de tableau a armé l'indicateur N°-64
La position suivante donnée par **PUTI** est donc la première.
On aurait pu également placer 5 au niveau 2 de la pile, pour indiquer la position. La nouvelle position obtenue au niveau 1 aurait alors été l'entier 1 seul.

Exemple2:



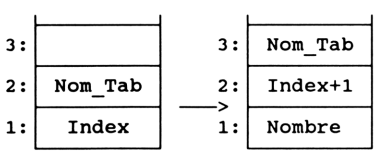
Ici le nombre -1 a été placé à l'intersection de la ligne 2 et de la colonne 4.
La position suivante indiquée par **PUTI** est définie par (ligne 3,colonne 1).
On aurait pu de même indiquer la position de placement par l'entier 8 au niveau 2. **PUTI** aurait alors donné l'entier 9 comme position suivante.

GETI

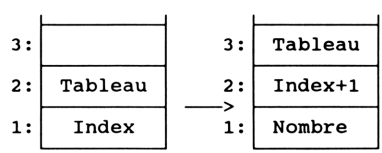
GETI permet de lire des nombres successifs d'un tableau. Chaque appel de **GETI** lit un de ces nombres. L'intérêt de cette instruction est qu'elle incrémente elle même la position où lire le nombre suivant.

Il y a deux schémas fonctionnels possibles ('Index' représente la position où se trouve le nombre à lire dans le tableau, et 'Nombre' désigne le nombre ainsi lu).

Premier cas:

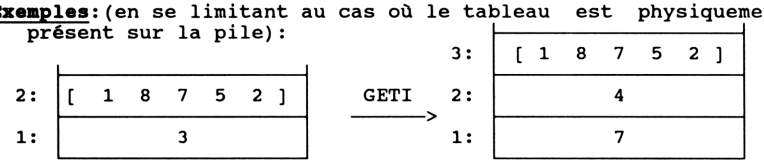


Deuxième cas:

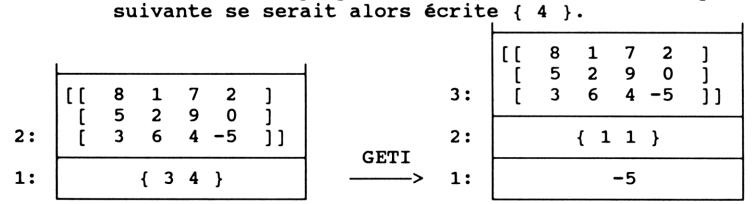


Voir la description de l'instruction **'PUTI'** pour ce qui est de la signification de 'Index' et de 'Index+1'.

Exemples:(en se limitant au cas où le tableau est physiquement présent sur la pile):



Ici le nombre 7 a été lu en 3-ème position, et la position suivante donnée par **GETI** est la 4-ème position.
On aurait très bien pu placer { 3 } au niveau 1. La position suivante se serait alors écrite { 4 }.



Ici le nombre -5 a été lu en position (ligne 3, colonne 4), c'est-à-dire en dernière position dans la matrice.
La position suivante que donne **GETI** est (ligne 1,colonne 1) c'est-à-dire la première position dans la matrice.
On aurait très bien pu placer 12 au niveau 1 . La nouvelle position aurait alors été l'entier 1 au niveau 2.

Remarque:
Si l'indicateur N°**-64** est armé (valeur = 1), **GETI** vient à l'instant de nous ramener en début de tableau.
Sinon il est désarmé (à zéro).

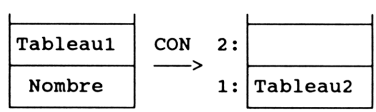
LE MENU "MTH MATR":

Le menu **MTH MATR** contient 9 commandes s'appliquant aux tableaux.

CON

L'instruction **CON** permet de créer des tableaux constants. Il y a plusieurs schémas fonctionnels possibles, dans lesquels 'Nombre' désigne le coefficient constant dont on veut remplir un tableau ('Nombre' est toujours placé au niveau 1):

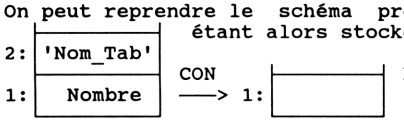
1) On peut tout d'abord indiquer les dimensions du tableau à créer en donnant un tableau 'receveur' (Ici 'Tableau1'). Il y a 2 cas suivant la nature du tableau receveur.



Premier cas: 'Tableau1' est un tableau réel. 'Nombre' doit être un réel. 'Tableau2' est alors un tableau réel de même format que 'Tableau1', mais dont tous les coefficients sont égaux à 'Nombre'.

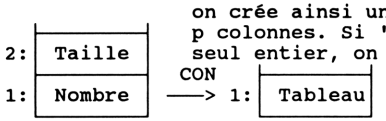
Deuxième cas: 'Tableau1' est un tableau complexe. 'Nombre' peut être un nombre réel ou complexe. 'Tableau2' est alors un tableau complexe (même si 'Nombre' est réel), de même format que 'Tableau1', de coefficients tous égaux à 'Nombre'.

2) On peut reprendre le schéma précédent, le tableau 'receveur' étant alors stocké dans une variable dont le nom 'Nom_Tab' est placé au niveau 2. Le tableau constant ainsi créé est placé directement dans la variable de nom 'Nom_Tab'.



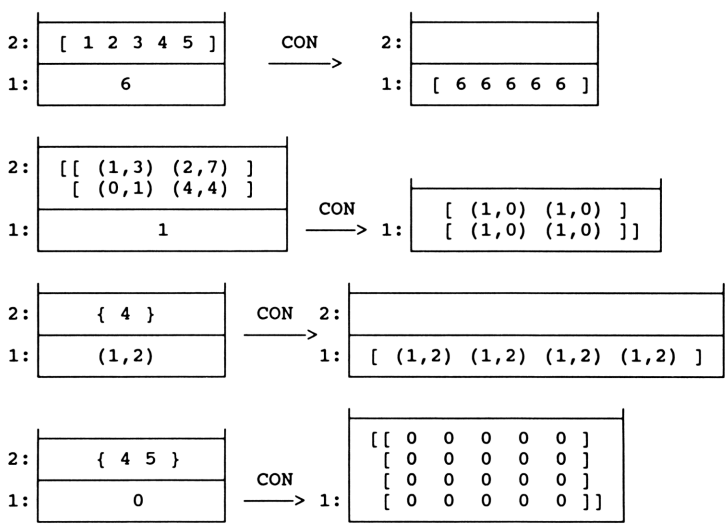
3) On peut encore indiquer les dimensions du tableau à créer au moyen d'une liste placée au niveau 2.

Si 'Taille' est une liste { n p } de deux entiers, on crée ainsi une matrice constante de n lignes et p colonnes. Si 'Taille' est une liste { n } d'un seul entier, on crée ainsi un vecteur constant de n termes.



Le tableau obtenu, de coefficients tous égaux à 'Nombre', est réel ou complexe suivant que 'Nombre' est réel ou complexe.

Exemples: (En se limitant aux cas 2) et 3)).

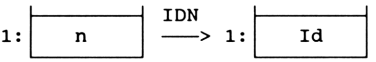


IDN

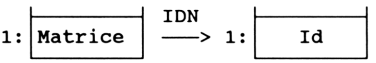
L'instruction **IDN** permet de créer des matrices 'identité'. Par définition la matrice identité d'ordre n (n étant un entier ≥ 1) est la matrice réelle ou complexe à n lignes et n colonnes dont tous les coefficients sont nuls, sauf les coefficients diagonaux qui valent 1 (les coefficients diagonaux sont ceux qui ont même indice de ligne et de colonne).

LES TABLEAUX.

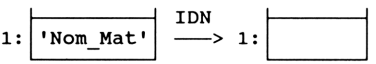
Il y a trois schémas fonctionnels possibles:



Ici n est un entier ≥1. On obtient la matrice identité d'ordre n au niveau 1 de la pile. On obtient ici une matrice réelle.

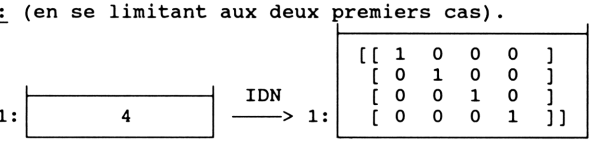


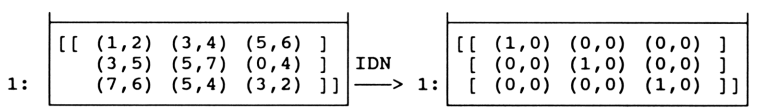
Ici 'Matrice' est une matrice carrée. On obtient la matrice identité de même format que la matrice initiale, et de même type (réel ou complexe).

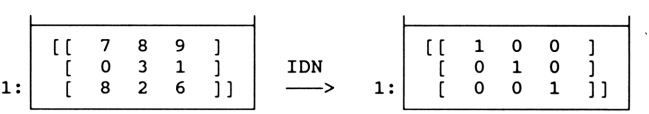


Ici 'Nom_Mat' est le nom d'une variable contenant une matrice carrée . Le contenu de cette variable est remplacé par la matrice identité de même format, et de même type (réel ou complexe).

Exemples: (en se limitant aux deux premiers cas).



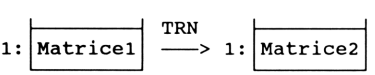




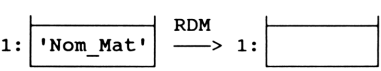
TRN

L'intruction **TRN** permet de transposer une matrice.

Premier cas: Matrice1 est ici une matrice présente physiquement sur la pile. La Matrice transposée (Matrice2) est obtenue au niveau 1.



Deuxième cas: 'Nom_Mat' est ici le nom d'une variable dont le le contenu est une matrice. La transposition est effectuée en mémoire, dans le contenu de la variable.



L'intruction **TRN** ne s'applique pas aux vecteurs.

Si la matrice initiale est de format { n p } (n lignes et p colonnes), alors la matrice obtenue est de format { p n }.

Plus précisément: pour tous indices i,j avec 1 ≤ i≤ p, 1 ≤ j≤ n, l'élément d'indice (i,j) de la matrice finale est égal:

* à l'élément d'indice (j,i) de la matrice initiale, dans le cas de matrices réelles.

* au conjugué de l'élément d'indice (j,i) de la matrice initiale, dans le cas de matrices complexes (dans ce cas, le terme mathématique adéquat est celui de matrice 'transconjuguée').

Autrement dit: Les lignes de la matrice finale sont les colonnes de la matrice initiale (après conjugaison dans le cas de matrices complexes).

Exemples:

```
1:  [[ 1  5  6  4 ]          TRN          1:  [[ 1  7  3 ]
     [ 7  2  0  6 ]      ------->              [ 5  2  1 ]
     [ 3  1  9  2 ]]                           [ 6  0  9 ]
                                               [ 4  6  2 ]]

1:  [[ (1,3) (2,4) ]          TRN          1:  [[ (1,-3) (0,-2) (7,-6) ]
     [ (0,2) (3,5) ]      ------->              [ (2,-4) (3,-5) (9,0)  ]]
     [ (7,6) (9,0) ]]
```

Remarques:

La définition mathématique habituelle de la transposition est obtenue par la séquence : **TRN CONJ** (ou **CONJ TRN**), dans laquelle tout élément de la matrice initiale voit simplement ses indices de ligne et de colonne inversés.

L'instruction **TRN** est involutive: Si on l'applique deux fois de suite à une matrice donnée, on retombe sur la matrice initiale.

RDM

L'intruction **RDM** permet de redimensionner un tableau.

Premier cas: Tableau1 est ici un tableau présent physiquement sur la pile. Le tableau redimensionné (Tableau2) est obtenu au niveau 1.

```
2: Tableau1
1: Taille      RDM ---> 1: Tableau2
```

Deuxième cas: 'Nom_Tab' est ici le nom d'une variable dont le le contenu est un tableau. Le 'redimensionnement' est effectué en mémoire, dans le contenu de la variable.

```
2: 'Nom_Tab'
1: Taille      RDM ---> 1:
```

Pour comprendre comment fonctionne l'instruction **RDM**, il faut se souvenir que dans un tableau, les coefficients sont repérés par un numéro de position:les positions croissantes (de la première à la dernière) sont obtenues en parcourant le tableau de la ligne supérieure à la ligne inférieure, et de gauche à droite sur une ligne donnée.

Dans les deux schémas fonctionnels ci-dessus, 'Taille' représente le format du tableau à obtenir:

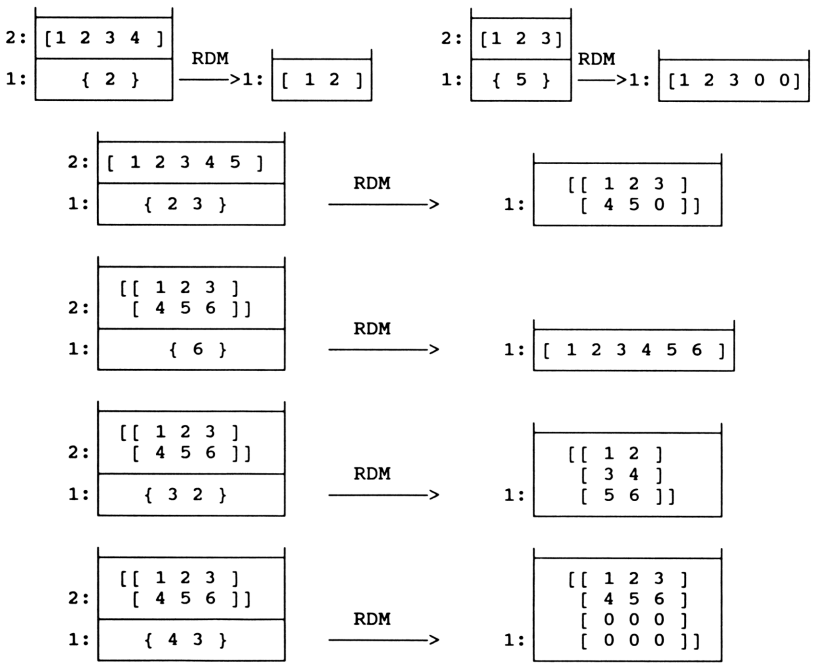
* 'Taille' est une liste { lig col } si on veut obtenir une matrice ayant 'lig' lignes et 'col' colonnes.

* 'Taille' est une liste { coeff } si on veut obtenir un vecteur ayant 'coeff' coefficients.

L'instruction **RDM** permet donc de transformer une matrice en vecteur, et vice-versa.

Si le format du tableau à obtenir comporte moins d'éléments que le format du tableau initial, les coefficients excédentaires de celui-ci sont perdus.
Dans le cas contraire, le tableau final est obtenu en complétant avec des coefficients nuls (0 dans le cas de tableaux réels, (0,0) dans le cas de tableaux complexes).

Exemples: (en se limitant à des tableaux réels)



DET

DET calcule le déterminant Det(A) d'une matrice carrée A.

```
       |---------|  DET      |---------|
  1:   |    A    |  ---> 1:  | Det(A)  |
```

Det(A) est un nombre réel ou complexe suivant que la matrice A est de de type réel ou complexe.

D'un point de vue théorique, la matrice A est inversible si et seulement si son déterminant est non nul.

Dans la pratique, le calcul de Det(A) par la HP48SX s'effectue le plus souvent avec une certaine erreur d'arrondi.

Exemples:

```
1:  [[ (1,3) (2,7) (4,1) ]
       (2,5) (0,1) (6,0) ]      DET
       (4,7) (9,3) (8,1) ]]    ---> 1:  (-6.00000000005,114)
```

En réalité, le déterminant de cette matrice est (-6,114).

```
1:  [[  5  7  3 11  ]
     [  7 11 15 19  ]
     [  3  4  5  6  ]         DET
     [  9 15 21 27  ]]       ---> 1:  -2.70540000002E-9
```

En réalité, le déterminant de cette matrice est nul (le déterminant d'une matrice à coefficients entiers est un entier).

```
1:  [[  1  1  1  1  ]
     [  1  2  4  8  ]
     [  1  3  9 27  ]         DET
     [  1  4 16 64  ]]       ---> 1:  12
```

Ici, le déterminant est obtenu sans erreur d'arrondi.

RSD

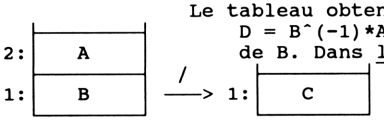
L'instruction **RSD** calcule le tableau A-BC, où B est une matrice et où A et C sont des tableaux (vecteurs ou matrices).

```
  3:  |  A  |
  2:  |  B  |
  1:  |  C  |  ---> 1:  | A-BC |
```

Les tableaux A,B,C sont soumis à des contraintes de format: D'une part le produit BC doit être possible (revoir les conditions qui permettent de multiplier une matrice par un tableau), et BC doit avoir exactement le même format que A (pour que l'opération A-BC soit possible).

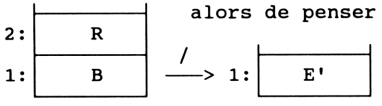
L'instruction **RSD** est utile pour accroître la précision de la commande / (division d'un tableau par une matrice carrée inversible).

Considérons en effet l'opération suivante, où A est un tableau, et où B est une matrice carrée inversible:



Le tableau obtenu est, en théorie, égal à: D = B^(-1)*A, où B^(-1) est la matrice inverse de B. Dans la pratique, on obtient un tableau C qui est une approximation de D. (Car il y a des erreurs d'arrondi dans la recherche de B^(-1)).

Notons E, le tableau E = D-C qui indique l'erreur commise.
L'égalité théorique B*D=A s'écrit B*(C+E)=A, c'est-à-dire:
B*E = A-B*C = **RSD**(A,B,C) = R (en notant R le tableau résiduel de A,B,C dans cet ordre), ou encore E=B^(-1)*R.
Il reste alors à calculer une approximation E' de E au moyen du schéma fonctionnel ci-contre. Il est permis alors de penser que la somme D'=C+E' constitue une bonne approximation de D (en tout cas une meilleure approximation que ne l'était le résulat C).



Exemple:

On veut calculer, avec une précision accentuée, la solution du système linéaire ci-contre.

```
56*X - 32*Y + 9*Z = 8
25*X - 17*Y + 5*Z = 56
31*X - 23*Y + 7*Z = 354
```

Pour celà plaçons le vecteur [8 56 354] dans la variable A
et la matrice [[56 -32 9]
[25 -17 5] dans la variable B.
[31 -23 7]]

La séquence: **A B /** place, au niveau 1 de la pile, le vecteur [97.3750000122 831.375000059 2351.00000014]. On obtient ainsi une approximation de la solution [X Y Z] du système.

Plaçons ce vecteur dans une variable nommée C, et éxécutons la séquence: **A B C RSD**.
On obtient, au niveau 1 de la pile, le vecteur (C'est le vecteur R des explications ci-dessus):
[-.0000000552 -.000000002 -.0000000012].

Avec ce vecteur au niveau 1, éxécutons la séquence: **B /** .
On obtient au niveau 1 de la pile, le vecteur
[-1.22000000007E-8 -5.90000000034E-8 -1.40000000008E-7].
(C'est le vecteur E' des explications ci-dessus).

Avec ce vecteur sur la pile, éxécutons la séquence: **C +** .
On obtient, au niveau 1 de la pile, le vecteur:
[97.375 831.375 2351]
(C'est le vecteur D' des explications ci-dessus).

Ce vecteur D' fait mieux que d'approcher fidèlement la solution du système initial: Il représente la solution exacte de ce système!, à savoir:
X = 97.375 = 779/8, Y = 831.375 = 6651/8, Z = 2351.

Remarque:La suite d'opérations décrite ci-dessus peut paraître un peu compliquée. Le mieux est d'en faire un programme. On trouvera un exercice proposant de rédiger ce programme et un autre appliquant cette technique à la recherche de l'inverse, avec précision accentuée, d'une matrice carrée.

ABS **ABS** calcule la norme 'euclidienne' d'un tableau A.

```
1: [ A ]  ABS ---> 1: [ Abs(A) ]
```

Si on note conventionnellement a, b, c,...., z les différents coefficients de la matrice ou du vecteur A, alors:

<u>Si A est un tableau réel:</u>
$ABS(A) = \sqrt{(a^2 + b^2 + c^2 + \ldots\ldots + z^2)}$

<u>Si A est un tableau complexe:</u>
$ABS(A) = \sqrt{(|a|^2 + |b|^2 + |c|^2 + \ldots\ldots + |z|^2)}$,
où $|a|$, $|b|$, ..., $|z|$ désignent les modules de a,b,c,...,z.
(Si $a = (\alpha,\beta)$, alors $|a|^2 = \alpha^2+\beta^2$)

<u>Exemples:</u>

```
1: [[ 5 1 2 6 ]
    3 4 0 2 ]
    7 3 9 1 ]]  ABS ---> 1: [ 15.3297097168 ]
```

(Le nombre obtenu est $\sqrt{(235)}$).

```
1: [ (4,1) (9,7) (8,0) ]  ABS ---> 1: [ 14.5258390463 ]
```

(Le nombre obtenu est $\sqrt{(211)}$).
En partant de ce vecteur sur la pile, la séquence:
DUP CONJ DOT donne le résultat (211,0).

<u>Remarques:</u>
L'instruction **ABS** donne le résultat 0 si et seulement si tous les coefficients du tableau initial sont nuls.

RNRM **RNRM** calcule la norme de ligne ("Row Norm") d'un tableau A.

```
1: [ A ]  RNRM ---> 1: [ Norme de ligne de A ]
```

Si A est une matrice notée par exemple:

```
      [[  a  b  ...  z  ]
       [  a' b' ...  z' ]
A =    ..................
       [  a" b"      z" ]],
```

alors la norme de ligne de A est:
$Max(|a|+|b|+..+|z|, |a'|+|b'|+..+|z'|, \ldots, |a"|+|b"|+..+|z"|)$

Si A est un vecteur noté par exemple: A = [a a' .... a"],
alors sa norme de ligne est Max ($|a|$, $|a'|$,..., $|a"|$),
ce qui revient à appliquer le schéma précédent en imaginant que le vecteur est en fait une matrice à une colonne.
Le tableau A est réel ou complexe.
Le résultat de **RNRM** est toujours un nombre réel, positif ou nul.
(nul seulement si tous les coefficients de A sont nuls).

Dans les notations ci-dessus, $|a|$ désigne:
* La valeur absolue de a si a est un réel.
* Le module $\sqrt{(\alpha^2+\beta^2)}$ de a si a est le complexe (α,β).

Exemples:

```
      [[ -5  1  2  6 ]
         3 -4  0 -2 ]
1:       7  3 -9  1 ]]      RNRM
                          ------>  1:   20
```

Le nombre obtenu l'est sur la ligne 3, où:
$|7| + |3| + |-9| + |1| = 20.$

```
1:  [ (4,1) (9,7) (8,0) ]   RNRM
                          ------>  1:  11.401754251
```

Le nombre obtenu l'est pour le 2-ième coefficient, où on a:
$|\ (9,7)\ | = \sqrt{(130)} = 11.401754251$

```
1:  [ [ 4 ] [-9 ] [-8 ] ]   RNRM
                          ------>  1:   9
```

Le nombre obtenu ici correspond à la valeur absolue de l'unique terme de la deuxième ligne de cette matrice à 3 lignes et 1 colonne.

CNRM **CNRM** calcule la norme de colonne ("Column Norm") d'un tableau A.

```
1:  A      CNRM
         ------>  1:  Norme de colonne de A
```

Si A est une matrice notée par exemple:

```
      [[  a  b  ...  z  ]
       [  a' b' ...  z' ]
A =    ..................
       [  a" b"      z" ]],
```

alors la norme de colonne de A est:
Max($|a|+|a'|+..+|a"|$, $|b|+|b'|+..+|b"|$, ..., $|z|+|z'|+..+|z"|$)

Si A est un vecteur noté par exemple: A = [a a' .... a"],
alors sa norme de colonne est $|a| + |a'| + ...+ |a"|$,
ce qui revient à appliquer le schéma précédent en imaginant que le vecteur est en fait une matrice à une colonne.

Le tableau A est réel ou complexe.
Le résultat de **CNRM** est toujours un nombre réel, positif ou nul.
(nul seulement si tous les coefficients de A sont nuls).

Dans les notations ci-dessus $|a|$ désigne:
* La valeur absolue de a si a est un réel.
* Le module $\sqrt{(\alpha^2+\beta^2)}$ de a si a est le complexe (α,β).

Exemples:

```
      [[ -5  1  2  6 ]
         3 -4  0 -2 ]
1:       7  3 -9  1 ]]      CNRM
                          ------>  1:   15
```

Le nombre obtenu l'est sur la colonne 1, où:
$|-5| + |3| + |7| = 15.$

```
1: [ (4,1) (9,7) (8,0) ]   CNRM ---->  1: 23.5248598766
```

Le nombre obtenu est |4+i| + |9+7i| + 8 = √17 + √130 + 8.

```
1: [ [ 4 ] [-9 ] [-8 ] ]   CNRM ---->  1: 21
```

Le nombre obtenu ici correspond à la somme des valeurs absolues des termes de l'unique colonne de cette matrice.

Remarque:
Pour une matrice A (pas un vecteur), la norme de colonne de A est égale à la norme de ligne de t(A) (la transposée de A) , et la norme de ligne de A est égale à la norme de colonne de t(A).

LE MENU "MTH VECTR":

Le menu **MTH VECTR** contient 6 instructions applicables aux vecteurs.

Parmi celles-ci, **ABS** a été étudiée dans le paragraphe précédent.

D'autres part, le menu **MTH VECTR** contient les entrées:
XYZ, R∠Z, R∠∠ (non programmables) qui permettent de choisir l'un des trois modes d'affichage des 2D et des 3D-vecteurs.
R→D, D→R (conversions entre degrés et radians): elles sont évoquées dans le chapitre "Nombres réels et complexes".

CROSS

CROSS permet de calculer le produit vectoriel W de deux vecteurs U et V (Notation habituelle W = U∧V).
U et V doivent être des vecteurs ayant 2 ou 3 coefficients. Si U ou/et V n'en a que deux, le 3ème coeff. est considéré égal à 0 (3ème coeff. rectangulaire)
Le résultat est un vecteur W ayant 3 coefficients . W est de type complexe si l'un au moins des 2 vecteurs U ou V est de type complexe.

```
2: U
1: V   CROSS ----> 1: W = U∧V
```

Si on note conventionnellement, en mode rectangulaire:
U = [a b c] et V = [a' b' c'],

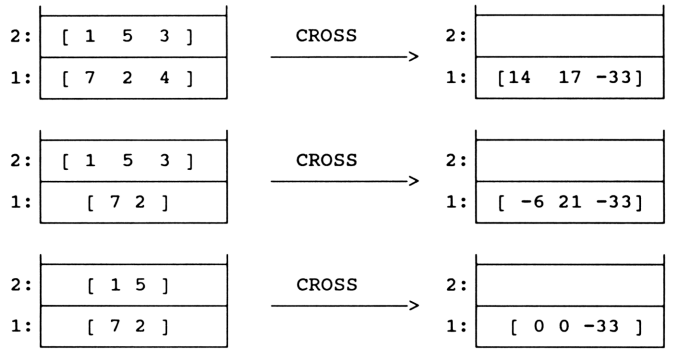
alors W = [bc'-cb' ca'-ac' ab'-ba'], ce qui correspond

à la disposition classique:

$$\begin{vmatrix} a \\ b \\ c \end{vmatrix} \wedge \begin{vmatrix} a' \\ b' \\ c' \end{vmatrix} = \begin{vmatrix} bc'-cb' \\ ca'-ac' \\ ab'-ba' \end{vmatrix}$$

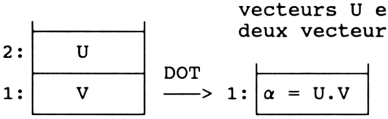
Tel est le calcul effectué en mémoire.
A l'écran, évidemment, les 2D et 3D-vecteurs en présence sont affichés suivant le mode en cours :
* rectangulaire.
* polaire.
* sphérique.

Exemples: (en se limitant à des vecteurs réels)



DOT

DOT permet de calculer le produit scalaire $\alpha = U.V$ de deux vecteurs U et V, de type réel ou complexe, les deux vecteurs U et V ayant le même nombre de coefficients.



Si on note symboliquement:

U = [a b c ..... z]

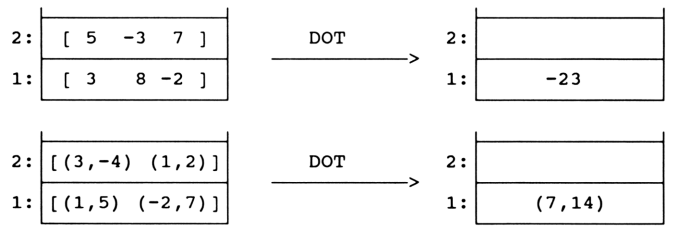
et V = [a' b' c' ..... z'],

Alors $\alpha = U.V = aa' + bb' + cc' + ... + zz'$.

Le résultat α est un nombre complexe si l'un au moins des deux vecteurs U ou V est un vecteur complexe.

Remarque: dans le cas de 2D ou 3D-vecteurs, le calcul du produit scalaire est effectué avec les coordonnées rectangulaires, et ceci quelque soit le mode d'affichage en cours (rectangulaire, polaire, ou sphérique).

Exemples:



Remarque:

La séquence **CONJ DOT** permet de calculer ce que l'on appelle souvent le produit scalaire hermitien des deux vecteurs U et V de type complexe.

Si U = [a b c ..... z] et V = [a' b' c' ..... z'],

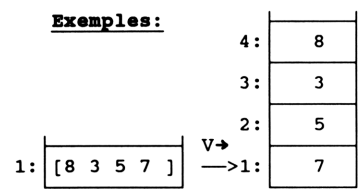
on obtient: $a\overline{a}' + b\overline{b}' + c\overline{c}' + ... + z\overline{z}'$.

V→ L'instruction **V→** sépare un vecteur réel en ses différents coefficients.
Dans le cas d'un 2D-vecteur, ou d'un 3D-vecteur, cette dissociation se fait en respectant le mode d'affichage en cours (rectangulaire, polaire, ou sphérique).

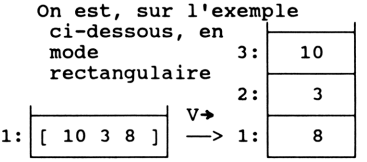
Les différences avec **OBJ→** (appliqué à un tableau), ou **ARRY→**, sont les suivantes:
OBJ→ s'applique à des vecteurs ou des matrices, complexes ou réelles, alors que **V→** ne s'applique qu'à des vecteurs réels.
Contrairement à **OBJ→**, **V→** ne place pas la taille du vecteur au niveau 1 de la pile.
OBJ→, appliquée à un 2D ou un 3D-vecteur, fournit les composantes rectangulaires de celui-ci, alors que **V→** donne ses composantes relativement au mode d'affichage en cours.

Remarque: si un vecteur réel est au niveau 1 de la pile, les touches **2D** et **3D** ont le même effet que l'instruction **V→** (quelque soit la dimension de ce vecteur).

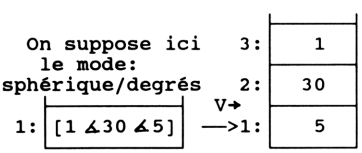
Exemples:



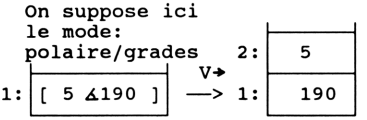
On est, sur l'exemple ci-dessous, en mode rectangulaire



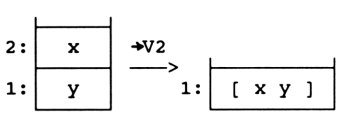
On suppose ici le mode: sphérique/degrés



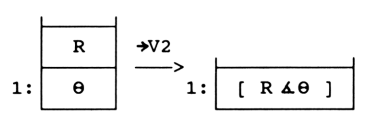
On suppose ici le mode: polaire/grades



→V2 L'instruction **→V2** recompose un 2D-vecteur à partir de ses composantes, en respectant le mode d'affichage en cours, (rectangulaire, ou polaire/sphérique)
(Voir remarque ci-après)
les niveaux 1 et 2 de la pile doivent contenir deux réels.



En mode rectangulaire

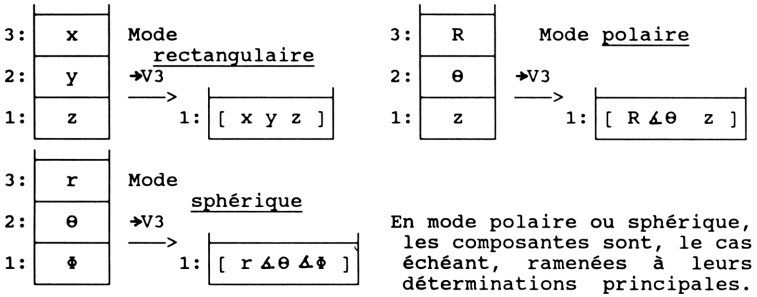


En mode polaire ou sphérique.

En mode polaire/sphérique, les composantes (radiale, polaire) sont ramenées, le cas échéant, à leur détermination principale.

Remarques:
Pour que **→V2** ait le comportement décrit ici, il faut que l'indicateur -19 soit désarmé (**19 +/- CF**) sinon l'objet formé est un nombre complexe (x, y) ou (r∠θ).
Si les niveaux 1 et 2 de la pile contiennent chacun un nombre réel, la touche **2D** à le même effet que l'instruction **→V2**.

→V3 L'instruction **→V3** recompose un 3D-vecteur à partir de ses composantes, en respectant le mode d'affichage en cours, (rectangulaire, ou polaire/sphérique)
Les niveaux 1,2 et 3 de la pile doivent évidemment contenir 3 nombres réels.



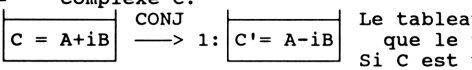
En mode polaire ou sphérique, les composantes sont, le cas échéant, ramenées à leurs déterminations principales.

Remarque: Si les niveaux 1, 2, 3 de la pile contiennent chacun un nombre réel, la touche **3D** à le même effet que l'instruction **→V3**.

LE MENU "MTH PARTS" ET LES TABLEAUX:

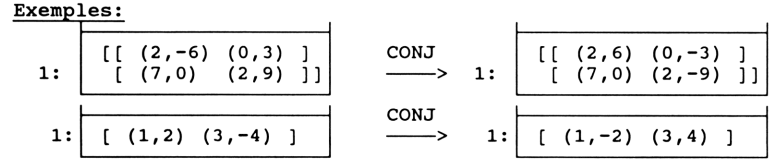
Le menu **MTH PARTS** contient quelques instructions qui peuvent s'appliquer aux tableaux: **ABS, CONJ, RE, IM, RND, TRNC.**
ABS a été étudiée dans un paragraphe précédent (on la trouve dans les menus **MTH VECTR** et **MTH MATR**).
RND (arrondir un réel) et **TRNC** (tronquer un réel) s'appliquent à tous les réels figurant dans un tableau. Ces instructions sont étudiées dans le chapitre "nombres réels et nombres complexes".

CONJ L'intruction **CONJ** permet de conjuguer le tableau réel ou complexe C.



Le tableau C' obtenu a même format que le tableau C.
Si C est un tableau réel, C'=C.

Exemples:



Remarque:
L'instruction **CONJ** est involutive: appliquée deux fois à un tableau donné, elle redonne le tableau initial.

RE

RE calcule le tableau réel A représentant la partie réelle du tableau réel ou complexe C.
Les tableaux A et C ont même format.
Si C est un tableau réel, alors A=C.

```
1: C = A+iB  ----RE---->  1: A
```

Exemples:

```
1: [ (3,2) (0,5) (1,-6) ]  ----RE---->  1: [ 3 0 1 ]

1: [[ (3,5) (0,2) (4,0) ]   ----RE---->  1: [[ 3 0 4 ]
    [ (7,4) (5,1) (2,1) ]]                    [ 7 5 2 ]]
```

IM

IM calcule le tableau réel B représentant la partie imaginaire du tableau réel ou complexe C.
Les tableaux B et C ont même format.
Si C est un tableau réel , alors B est un tableau nul.

```
1: C = A+iB  ----IM---->  1: B
```

Exemples:

```
1: [ (3,2) (0,5) (1,-6) ]  ----IM---->  1: [ 2 5 -6 ]

1: [[ (3,5) (0,2) (4,0) ]   ----IM---->  1: [[ 5 2 0 ]
    [ (7,4) (5,1) (2,1) ]]                    [ 4 1 1 ]]
```

FONCTIONS APPLICABLES AUX TABLEAUX:

* Les fonctions qui s'appliquent aux 'expressions-tableaux':

Un certain nombre d'instructions parmi celles qui viennent d'être décrites sont des fonctions, c'est-à-dire qu'elles peuvent être utilisées dans une expression algébrique.

On peut les appliquer à des sous-expressions représentant des tableaux (dénommées ici 'expressions-tableaux'), certaines de ces sous-expressions pouvant par exemple se réduire à un nom local ou global.

Les fonctions applicables aux 'expressions-tableaux' sont:

* Les fonctions arithmétiques: **+ - * / INV NEG SQ**
* Les fonctions **ABS, RE, IM, CONJ** (un argument)
* Les fonctions **RND, TRNC** (deux arguments):
Si 'A' est le nom d'une variable contenant un tableau, l'expression 'RND(A,3)' renvoie ce tableau avec des coefficients arrondis à la troisième décimale.

Si par exemple, A et B sont deux variables (locales ou globales) dont le contenu est un tableau, on peut très bien envisager de créer puis d'évaluer l'expression (dont on ne prétend pas qu'elle soit très utile):

'NEG(A/RE(B)*INV(A-B)+CONJ(IM(B)+SQ(A)))*ABS(A)'

Si par exemple A = [[(1,2) (3,4)] et B = [[(6,0) (1,1)]
[(1,0) (3,5)]] [(3,9) (2,2)]]

alors l'évaluation de l'expression ci-dessus donne la matrice:
[[(-1.07188113107,65.2051668164) (120.82583632 ,297.741640885)]
[(-108.668964592,58.3487753405) (86.7129387266,271.660836845)]]

On comprend bien pourquoi des instructions comme:
→ARRY, OBJ→, PUT, GET, PUTI, GETI, SIZE, RDM, CON, IDN, TRN, RSD, R→C, C→R, CROSS, DOT
ne puissent être intégrées à des expressions algébriques.

C'est moins évident pour: **DET**, **RNRM**, **CNRM** qui ne s'appliquent pourtant qu'à un seul argument et qui se contentent de renvoyer un résultat réel. Mais c'est ainsi.
Une expression algébrique telle que: **'DET(B)*A'** constitue donc une erreur de syntaxe.

* Les instructions qui s'appliquent aux 'noms-tableaux':

Dans les pages précédentes, on a vu que certaines instructions (qui ne sont pas des fonctions) s'appliquent néanmoins à des noms de variables.

Résumons nos observations en appelant, pour l'occasion, 'A' le nom d'une variable (globale ou locale) dont le contenu est un tableau:

Les instructions **PUT** et **PUTI** permettent de placer un élément directement dans le contenu de la variable 'A'.

Les instructions **GET** et **GETI** permettent d'extraire directement un élément du contenu de la variable 'A' un élément.

RDM permet de redimensionner le contenu de la variable 'A'

TRN permet de transposer le contenu de la variable 'A'

CON permet de mettre un tableau constant dans la variable 'A'

IDN permet de mettre la matrice identité dans la variable 'A'

■ LECTURES ET ECRITURES DANS UN TABLEAU:

*** L'ECRITURE DANS UN TABLEAU:**

Comme nous venons de le voir, il y a deux façons de placer un nombre dans un tableau:
* au moyen de l'instruction **PUT**.
* au moyen de l'instruction **PUTI**.

Le tableau peut être "physiquement" présent sur la pile, ou bien représenté par un nom, auquel cas l'écriture se fait dans le contenu de la variable ainsi référencée.

* LA LECTURE DANS UN TABLEAU:

Il y a par contre 3 moyens de lire un nombre dans un tableau:

* Avec GET et GETI:

On peut utiliser les instructions **GET** ou **GETI** (Le tableau étant présent sur la pile, ou bien représenté par un nom).

* Avec EVAL:

On peut également utiliser l'instruction **EVAL** (ou **→NUM**):

Ainsi si A est le nom (local ou global) d'une variable dont le contenu est un vecteur:
La séquence **'A(k)' EVAL** place le k-ème élément du vecteur sur la pile (k étant un entier ou un nom désignant un entier).

De même si A est le nom d'une variable dont le contenu est une matrice:
la séquence **'A(i,j)' EVAL** place au niveau 1 de la pile le nombre qui se trouve, dans la matrice 'A', à l'intersection de la ligne i et de la colonne j (i et j étant des entiers ou des noms désignant des entiers).

Plus généralement, on peut ainsi référencer tel ou tel élément d'un tableau, à l'intérieur d'une expression algébrique.

Par ex: **'A(k)*k+A(A(k))' EVAL** est syntaxiquement correct pour un vecteur.

Remarque:
Dans le cas d'un 2D ou 3D-vecteur, le k-ième coefficient est ici la k-ième composante rectangulaire (le mode d'affichage n'intervient pas).

Il est intéressant de comparer les différentes possibilités énumérées ci-dessus, pour tester au moins leur rapidité.
Comme nous allons le voir le résultat est instructif.

* TEST COMPARATIF (OPERATIONS DE LECTURE):

On suppose l'existence d'une variable nommée A (nom global) dont le contenu est un tableau réel de 20 lignes et 20 colonnes (J'ai utilisé le tableau constant de coefficients nuls).
On va écrire un programme effectuant une lecture séquentielle de ce tableau (du début à la fin).

```
'PROG1': «   A  { 1 1 }
             1   400   START   GETI   DROP   NEXT
             DROP2
         »
```

'PROG2': identique à **PROG1**, en remplaçant **A** par **'A'**. On lit donc les éléments dans le contenu de la variable **'A'**.

'PROG3': identique à **PROG1**, en remplaçant { 1 1 } par 1.
Le pointeur de position utilisé par **GETI** est donc un entier et non pas une liste de deux entiers.

'PROG4':
```
« 1   20   FOR  i
     1   20   FOR   j
        'A'   i  j   2  →LIST   GET   DROP
     NEXT
  NEXT
»
```

'PROG5': identique à **PROG4**, en remplaçant **'A'** par **A**.

'PROG6':
```
« A  1   20   FOR  i
     1   20   FOR   j
        DUP   i  j   2  →LIST   GET   DROP
     NEXT
  NEXT   DROP
»
```

'PROG7':
```
«  A
  1   400   FOR  i   DUP  i   GET   DROP   NEXT
  DROP
»
```

'PROG8':
```
« 1   20   FOR  i
     1   20   FOR   j
        'A(i,j)'  EVAL   DROP
     NEXT
  NEXT
»
```

J'ai mesuré les temps de calculs nécessaires à l'éxécution de ces programmes (qui font tous la même chose).
'PROG1': 36 s, 'PROG2': 38s, 'PROG3': 11s, 'PROG4': 33s
'PROG5': 29 s, 'PROG6': 29s, 'PROG7': 9s, 'PROG8': 36s.

Ces résultats sont très significatifs:
Les instructions de lecture dans un tableau sont beaucoup plus rapides quand on utilise un entier pour pointer les différentes positions dans le tableau (plutôt que d'utiliser une liste de deux entiers dans le cas d'une matrice).

L'instruction **GETI** est un peu plus lente que **GET**.
De même,l'utilisation de **EVAL** pour lire un élément donné du tableau ralentit beaucoup l'éxécution.

Toujours pour une rapidité maximum, il vaut mieux placer le tableau sur la pile, plutôt que de le référencer par le nom de la variable.

*** TEST COMPARATIF (OPERATIONS D'ECRITURE):**

Les mêmes remarques s'appliquent aux procédures d'écriture dans un tableau:
On suppose l'existence d'une variable globale nommée 'A' dont le contenu est un tableau réel de 10 lignes et de 10 colonnes.
On écrit un programme qui place 1 à chaque position du tableau.

'PROG1':
```
«  A   1   100   FOR   i   i   1   PUT   NEXT
»
```

'PROG2':
```
«  'A'  1  100  FOR  i  DUP  i  1  PUT  NEXT
»
```

```
'PROG3':  «   A   1
              1   100   START   1   PUTI   NEXT
              DROP
          »
```

'PROG4': Comme 'PROG3', mais en remplaçant A par 'A'.

```
'PROG5':  «   A   1  10  FOR  i
                 1  10  FOR  j
                    i   j   2   →LIST   1   PUT
                 NEXT
              NEXT
          »
```

'PROG6': Comme 'PROG5', mais en remplaçant A par 'A'.
et "i j 2 →LIST" par "DUP i j 2 →LIST"

```
'PROG7':  «   A   { 1 1 }
              1   100   START   1   PUTI   NEXT
              DROP
          »
```

'PROG8': Comme 'PROG7', mais en remplaçant A par 'A'.

Les temps de calculs sont les suivants:

'PROG1': 4.2 s, 'PROG2': 3.3 s, 'PROG3': 4.5 s, 'PROG4': 4.0 s
'PROG5': 9.2 s, 'PROG6': 8.0 s, 'PROG7': 10.1s, 'PROG8': 9.6s.

Là encore, on remarque la relative lenteur des opérations d'écriture quand on utilise une liste de deux entiers pour désigner les positions dans la matrice (l'écriture est deux fois plus rapide quand on utilise un entier).

Néanmoins, et c'est là une différence avec **GET** et **GETI**, les opérations d'écriture sont légèrement plus rapides quand on utilise le nom 'A' de la matrice (plutôt que de placer son contenu sur la pile.

L'explication est la suivante:

Quand on place la matrice sur la pile (et non 'A', son nom), la HP48SX sauvegarde, à chaque écriture d'un coefficient, la matrice comme dernier argument utilisé (c'est le rôle de **LASTARG**), ce qui l'oblige à recopier toute cette matrice quelque part (en effet, la variable 'A' reste inchangée).

Quand on utilise le nom 'A', les modifications s'effectuent dans la variable elle-même, et il n'est pas nécessaire de stocker tout le tableau dans la mémoire réservée à **LASTARG**.

EXERCICES SUR LES TABLEAUX:

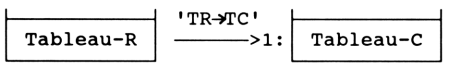
La difficulté des exercices est mesurée par des symboles * ou # :

```
*     Très facile.
**    Assez simple.
***   Déjà un peu plus délicat.
****  Plutôt difficile.
```

Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes précédant dans cette liste d'exercices.

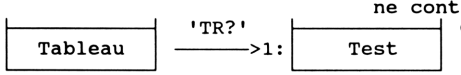
Exercice 1 (*) : Transformer un tableau réel en tableau complexe.

Le programme **'TR→TC'** transforme le tableau réel au niveau 1 en un tableau complexe de partie imaginaire nulle. Si le tableau initial est complexe, il est inchangé.



Exercice 2 (*) : Tester si un tableau est formé de nombres réels.

Le programme **'TR?'** teste si le tableau présent au niveau 1 ne contient que des nombres réels. On obtient 1 au niveau 1 si la réponse est oui, 0 sinon.

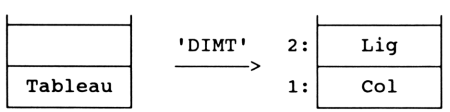


Rappelons que l'instruction **TYPE** de la HP48SX permet de savoir si un tableau est au format réel (Type=3) ou au format complexe (type=4).

La réponse au problème posé ici doit être 'oui' si le tableau est au format réel, ou s'il est au format complexe mais ne contient que des nombres dont la partie imaginaire est nulle.

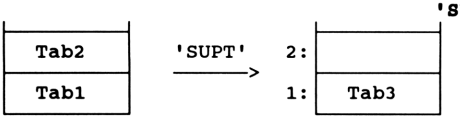
Exercice 3 (*) : Dimensions d'un tableau.

'DIMT' place au niveau 1 le nombre de colonnes et au niveau 2 le nombre de lignes du tableau placé au niveau 1.



Si 'Tableau' est un vecteur alors 'Col' est égal au nombre de ses coefficients, et 'Lig' est égal à 1.

Exercice 4 (##) : Superposition de deux tableaux.

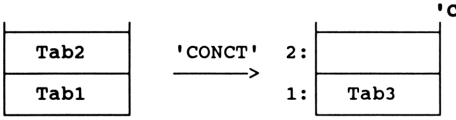


'SUPT' superpose les tableaux Tab1 et Tab2, qui doivent avoir le même nombre de colonnes, en un tableau Tab3 (où Tab2 apparaît au dessus de Tab1).

La forme du tableau 'Tab3' est donc $Tab3 = \frac{Tab2}{Tab1}$

Tab1 et/ou Tab2 peuvent être des vecteurs (le nombre de leurs coefficients est considéré comme le nombre de colonnes).
Si l'un des deux tableaux 'Tab1' ou 'Tab2' est au format complexe alors 'Tab3' est au format complexe.
Dans tous les cas 'Tab3' est une matrice ayant au moins 2 lignes.

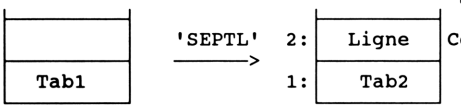
Exercice 5 (###) : Concaténation de deux tableaux.



'CONCT' "accole" les tableaux Tab1 et Tab2, qui doivent avoir le même nombre de lignes , en un tableau Tab3 (où Tab2 apparaît à gauche de Tab1).

La forme du tableau 'Tab3' est donc Tab3 = Tab2|Tab1.
Tab1 et/ou Tab2 peuvent être des vecteurs (et ils sont considérés comme tableaux à une seule ligne).
Si l'un des deux tableaux 'Tab1' ou 'Tab2' est au format complexe alors 'Tab3' est au format complexe.
Dans tous les cas 'Tab3' est une matrice ayant au moins 2 lignes.

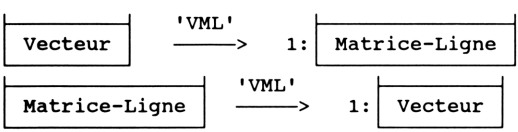
Exercice 6 (***) : Séparer un tableau de sa ligne supérieure.



'SEPTL' isole la 1-ère ligne du tableau 'Tab1'. Cette ligne est placée au niveau 2, et ce qui reste du tableau (Tab2) est placé au niveau 1.

'Ligne' est obtenue sous forme de vecteur.
Si Tab1 est une matrice à 1 ligne, elle se transforme en vecteur.
Si Tab1 est un vecteur, il reste inchangé.

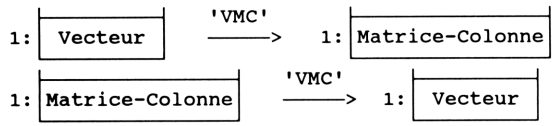
Exercice 7 (**): Transformation réversible d'un vecteur en matrice à une seule ligne.



'VML' transforme de façon réversible une matrice-ligne en le vecteur correspondant.

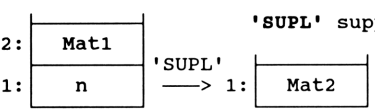
LES TABLEAUX: Exercices.

Exercice 8 (**): Transformation réversible d'un vecteur en matrice à une seule colonne.



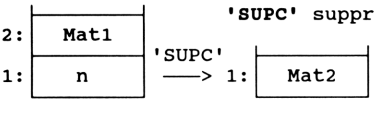
'VMC' transforme de façon réversible une matrice-colonne en le vecteur correspondant.

Exercice 9 (***): Supprimer une ligne dans une matrice.



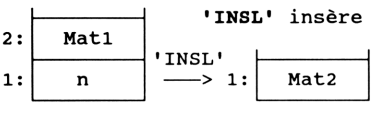
'SUPL' supprime la n-ème ligne de la matrice 'Mat1' placée au niveau 2. La matrice modifiée, 'Mat2', est obtenue au niveau 1.

Exercice 10 (##): Supprimer une colonne dans une matrice.



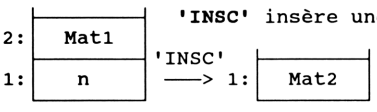
'SUPC' supprime la n-ème colonne de la matrice 'Mat1' placée au niveau 2. La matrice modifiée, 'Mat2', est obtenue au niveau 1.

Exercice 11 (***): Insérer une ligne de zéros dans une matrice.



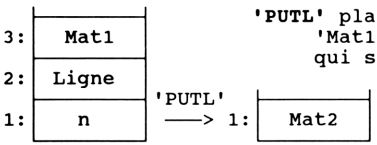
'INSL' insère une ligne de zéros dans la matrice 'Mat1' placée au niveau 2. Cette ligne de zéros devient la n-ème ligne de la matrice modifiée , 'Mat2', obtenue au niveau 1.

Exercice 12 (##): Insérer une colonne de zéros dans une matrice.



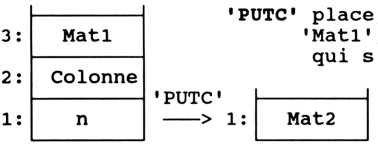
'INSC' insère une colonne de zéros dans la matrice 'Mat1' placée au niveau 2. Cette colonne de zéros devient la n-ème colonne de la matrice modifiée , 'Mat2', obtenue au niveau 1.

Exercice 13 (**): Placer une ligne donnée dans une matrice donnée.



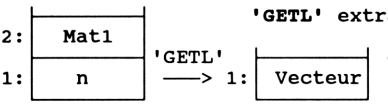
'PUTL' place, à la n-ème ligne de la matrice 'Mat1', et en remplacement de la ligne qui s'y trouvait précédemment, la ligne représentée au niveau 2. La matrice modifiée 'Mat2' est obtenue au niveau 1.
('Ligne' peut être un vecteur ou une matrice ligne).

Exercice 14 (##): Placer une colonne donnée dans une matrice donnée.



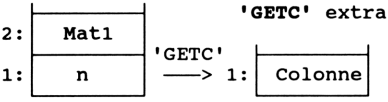
'PUTC' place, à la n-ème colonne de la matrice 'Mat1', et en remplacement de la colonne qui s'y trouvait précédemment, la colonne représentée au niveau 2 (et qui est un vecteur ou une matrice à une colonne). La matrice modifiée 'Mat2' est obtenue au niveau 1.

Exercice 15 (**): Extraire une ligne donnée d'une matrice donnée.



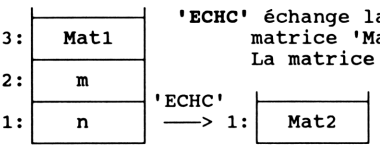
'GETL' extrait la n-ème ligne de la matrice 'Mat1'. Le résultat est un vecteur obtenu au niveau 1.

Exercice 16 (**): Extraire une colonne donnée d'une matrice donnée.



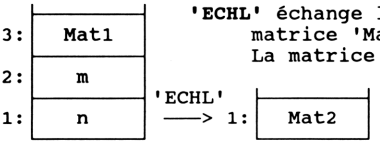
'GETC' extrait la n-ème colonne de la matrice 'Mat1'.Le résultat est une matrice uni-colonne, obtenue au niveau 1.

Exercice 17 (***): Echanger deux colonnes dans une matrice.



'ECHC' échange la n-ème et la m-ème colonnes de la matrice 'Mat1' placée au niveau 3. La matrice modifiée 'Mat2' est obtenue au niveau 1 de la pile.

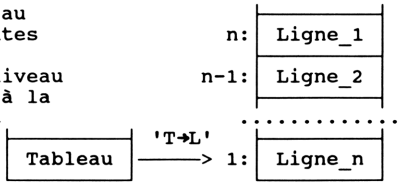
Exercice 18 (##): Echanger deux lignes dans une matrice.



'ECHL' échange la n-ème et la m-ème lignes de la matrice 'Mat1' placée au niveau 3. La matrice modifiée 'Mat2' est obtenue au niveau 1 de la pile.

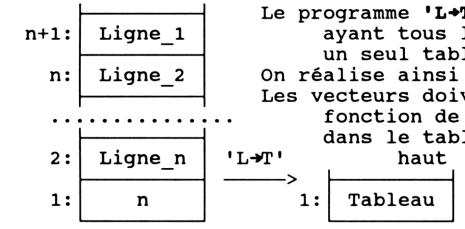
Exercice 19 (***) : Séparer un tableau en ses différentes lignes.

'T→L' sépare le tableau placé au niveau 1 en ses différentes lignes.
Chaque ligne est placée à un niveau de la pile correspondant à la position qu'elle occupait dans le tableau.
Ainsi la première ligne du tableau est placée au plus haut sur la pile.



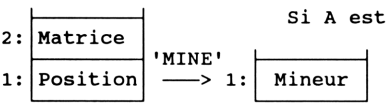
LES TABLEAUX: Exercices.

Exercice 20 (***) : Regrouper n lignes en un seul tableau.



Le programme **'L→T'** regroupe n lignes (n vecteurs ayant tous le même nombre de coefficients) en un seul tableau de n lignes.
On réalise ainsi l'opération inverse de **'T→L'**.
Les vecteurs doivent être placés sur la pile en fonction de la position qu'ils vont occuper dans le tableau (ainsi le vecteur le plus haut placé sur la pile deviendra la première ligne du tableau).
Le nombre n de lignes à regrouper doit être placé au niveau 1.
(Si n=1, on obtient le vecteur qui était au niveau 2).

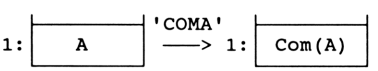
Exercice 21 (##): Mineur d'un coefficient d'une matrice carrée.



Si A est une matrice carrée et si a est un coefficient de A, le mineur de a est égal au déterminant de la matrice obtenue en supprimant de A la ligne et la colonne de a.

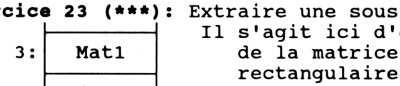
Le programme **'MINE'** doit calculer le mineur du coefficient de la matrice donnée au niveau 2, la position de ce coefficient dans cette matrice étant placée au niveau 1 (sous forme d'une liste { n p } de 2 entiers: n pour le numéro de ligne et p pour le numéro de colonne).

Exercice 22 (**): Comatrice d'une matrice carrée.



Si A est une matrice carrée, sa comatrice est la matrice notée **Com(A)**, carrée de même format que A, et dont le terme général (celui qui est placé en ligne i et en colonne j) est égal à: (-1)^(i+j)* (Dij), où "Dij" est le mineur du terme d'indices i et j de la matrice A.
Le rôle du programme **'COMA'** est donc de calculer la comatrice de la matrice carrée A placée au niveau 1.

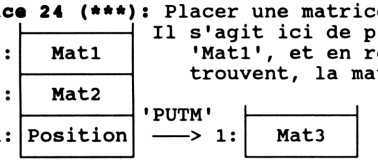
Exercice 23 (***): Extraire une sous-matrice d'une matrice.



Il s'agit ici d'extraire une sous-matrice 'Mat2' de la matrice 'Mat1'. 'Mat2' est le tableau rectangulaire délimité dans 'Mat1' par les positions 'Debut' et 'Fin'.
Ces deux positions ont la forme d'une liste {n p} de 2 entiers (n pour l'indice de colonne et p pour l'indice de ligne).

'Début' désigne le coin supérieur gauche de la matrice à extraire et 'Fin' la coin inférieur droit.
Si par exemple 'Debut' = { 2 3 } et 'Fin' = { 4 7 },celà signifie que l'on veut extraire la sous-matrice constituée des termes dont l'indice de ligne est compris entre 2 et 4, et dont l'indice de colonne est compris entre 3 et 7 (on obtient ainsi une sous-matrice de format { 3 5 }).

Exercice 24 (***): Placer une matrice à l'intérieur d'une autre.

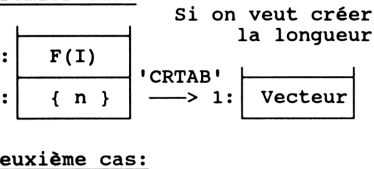


Il s'agit ici de placer, à l'intérieur de la matrice 'Mat1', et en remplacement des termes qui s'y trouvent, la matrice 'Mat2', en positionnant le coin supérieur gauche de 'Mat2' à la position de 'Mat1' donnée par une liste placée au niveau 1 (et qui a la forme { n p }, où n et p représentent respectivement un indice de ligne et un indice de colonne) .
La matrice modifiée, notée ici 'Mat3', est obtenue au niveau 1.
'Mat2' doit alors apparaître comme une sous-matrice de 'Mat3'.

Exercice 25 (***): Créer un tableau dont le terme général répond à une formule donnée.

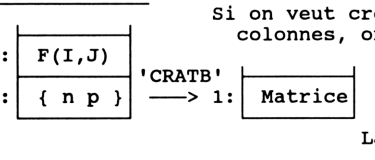
Il s'agit ici de créer un tableau dont le terme général obéit à une formule prédéfinie.
Notons I l'indice d'un coefficient d'un vecteur, et I,J les indices définissant respectivement l'indice de ligne et l'indice de colonne d'un coefficient d'une matrice.

Premier cas:



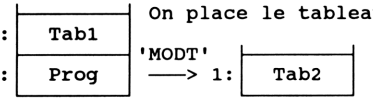
Si on veut créer un vecteur de longueur n,on place la longueur au niveau 1 (liste { n }), et la formule au niveau 2 (sous forme d'une expression algébrique par rapport à la variable I).
Le vecteur est obtenu au niveau 1

Deuxième cas:



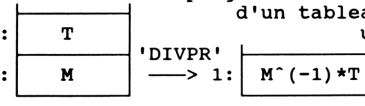
Si on veut créer une matrice à n lignes et p colonnes, on place le format au niveau 1,sous forme d'une liste { n p }, et la formule au niveau 1 (sous forme d'une expression algébrique par rapport aux variables I et J).
La matrice est obtenue au niveau 1.

Exercice 26 (***): Application d'une même procédure à tous les éléments d'un tableau.



On place le tableau à modifier ('Tab1') au niveau 2. 'Prog' est ici un programme (en notation **RPN**) qui est destiné à s'appliquer à chaque élément de 'Tab1', et à fournir ainsi les éléments du tableau modifié 'Tab2' (de même format que 'Tab1').
'Prog' doit s'appliquer au nombre placé au niveau 1 de la pile, et doit le remplacer par un nouveau (en principe) nombre.

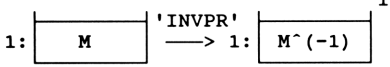
Exercice 27 (***):Division d'un tableau par une matrice en précision accentuée.



Le programme **'DIVPR'** permet d'effectuer la division d'un tableau T par une matrice carrée M, avec une précision améliorée par rapport à l'utilisation de /, et ceci grâce à l'utilisation de **RSD**.

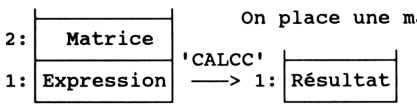
LES TABLEAUX: Exercices.

Exercice 28 (##): Inversion d'une matrice carrée, en précision accentuée.



Il s'agit ici de calculer l'inverse d'une matrice carrée M, avec une précision supérieure à celle qu'on obtient par **INV**, et ce en utilisant le programme **'DIVPR'**.

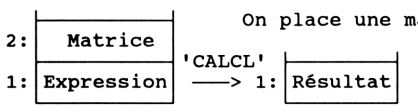
Exercice 29 (####): Calculs sur les colonnes d'une matrice.



On place une matrice A au niveau 2 de la pile. Le problème est ici d'évaluer une expression algébrique ou un programme où apparaissent éventuellement les noms C1, C2 C3, ... etc, représentant les différentes colonnes de A (numérotées de gauche à droite, et considérées comme des matrices à une colonne).
Le résultat de l'évaluation est placé au niveau 1 de la pile.

Par exemple: (pour une matrice ayant au moins 4 colonnes).
On peut chercher à évaluer: **'(C1+2*C3)/ABS(C4)'** ou ce qui revient au même, le programme « **C3 2 * C1 + C4 ABS / »**.
L'utilisation de la synatxe **RPN** permet ici d'utiliser des instructions interdites dans les expressions algébriques. (par exemple **TRN**, **RDM**, **CON**, **IDN**, etc...)

Exercice 30 (####): Calculs sur les lignes d'une matrice.



On place une matrice A au niveau 2 de la pile. Le problème est ici d'évaluer une expression algébrique ou un programme où apparaissent éventuellement les noms L1, L2 L3, L4 ... etc, représentant les différentes lignes de A (numérotées de haut en bas , et considérées comme des vecteurs).
Le résultat de l'évaluation est placé au niveau 1 de la pile.

Par exemple: (pour une matrice ayant au moins 4 lignes).
On peut chercher à évaluer: **'(L2+ABS(L1)*CONJ(L3)'**,ou ce revient au même le programme « **L1 ABS L3 CONJ * L2 + »**.
L'utilisation de la synatxe **RPN** permet ici d'utiliser des instructions interdites dans les expressions algébriques. (par exemple **CROSS**, **DOT**, etc ...)

VARIABLES GLOBALES ET REPERTOIRES

CREER, MODIFIER, SUPPRIMER UNE VARIABLE GLOBALE:

* CREATION D'UNE VARIABLE GLOBALE:

Cas général:

L'instruction qui place un objet dans une variable est: **STO**.
La syntaxe est: **Objet Nom STO**,
où 'Nom' est le nom de la variable, et où 'Objet' est l'objet qu'on veut y mettre (et qui devient le contenu de la variable).

Si cette variable n'existait pas encore, elle est créée par cette instruction. Son nom devient alors la première entrée du menu **VAR** dans le répertoire où vous vous trouvez.

Cas particuliers:

Certains noms de variables sont réservés à un usage précis par la HP48SX. Ces variables peuvent être créées comme il vient d'être dit (sauf **PICT**), mais il est recommandé de ne les utiliser que dans le sens que leur attribue la HP48SX.

Il s'agit des variables:

ΣDAT: matrice statistique.
ΣPAR: paramètres statistiques.
EQ: équation courante, utilisée dans **SOLVE** et **PLOT**.
PPAR: paramètres de tracé.
CST: description du menu-utilisateur courant.
ALRMDAT: description de l'alarme en cours de traitement.
PRTPAR: paramètres d'impression.
IOPAR: paramètres relatifs aux opérations d'entrée-sortie.
PICT: image graphique en cours.
n1, **n2**, ...: entiers symboliques utilisés par **ISOL**.
s1, **s2**, ...: signes symboliques utilisés par **ISOL** et **QUAD**.

Certaines des variables ci-dessus peuvent également être crées par des instructions particulières:

Par exemple:
ΣDAT peut être créée par **STOΣ** ou **Σ+** (dans le menu **STAT**).
PPAR est automatiquement créée avec ses valeurs par défaut (si du moins elle n'existe pas déjà dans le répertoire courant) par le seul fait de sélectionner le menu **PLOT**.
EQ peut être créée par l'instruction **STEQ** (dans les menus **PLOT** et **SOLVE**)

* MODIFICATION D'UNE VARIABLE GLOBALE:

Il y a plusieurs façons ensuite de modifier le contenu d'une variable (ces opérations nécessitent que l'on soit encore dans le répertoire où cette variable a été créée).

* En y plaçant un nouveau contenu par **STO**.

* Si le label de la variable est visible dans la ligne de menu (menu **VAR** affiché) alors il est possible de placer l'objet au niveau 1 dans la variable, en actionnant successivement:
  - La touche "Shift-gauche": <=
  - La touche de menu correspondant au nom de cette variable.

* En éditant et en modifiant ce contenu:
  - en plaçant le nom au niveau 1 ou en ligne de commande, et en actionnant la touche **VISIT** (le contenu de la variable est alors placé en ligne de commande).
  - en plaçant le nom en ligne de commande ou au niveau 1 et en actionnant => ∇ .
  - en passant dans la "pile interactive", en sélectionnant le niveau où se trouve le nom, et en éxécutant la séquence => **VIEW**.
  - Dans les deux cas ci-dessus:
    Les tableaux sont placés dans **MatrixWriter**.
    Les expressions et objets-unités sont placés dans **EquationWriter**.

  La "visite" du contenu d'une variable (à des fins de modifications) se termine par **ENTER** (pour valider les modifications) ou sur **ON** (pour les annuler).

* En utilisant les instructions du menu **STORE** (accessible par la séquence => **VAR**), mais celà n'est possible que dans certains cas (voir le paragraphe consacré à ce menu).

  Ex: si la variable **'A'** contient un nombre réel:
  1 'A' STO+
  ajoute 1 au contenu de cette variable.

* Si le contenu de la variable est un tableau ou une liste, certaines instructions de la HP48SX permettent de modifier tout ou partie de ce tableau ou de cette liste.

  Ex: si la variable **'A'** contient un tableau ou une liste:
  'A' 1 0 PUT
  place le nombre 0 en première position dans ce tableau ou dans cette liste.

Cas particuliers:

Certaines des variables utilisées par la HP48SX dans un but spécifique (et évoquées ci-dessus) peuvent être modifiées comme on vient de le voir, mais également par des moyens particuliers.

Par exemple:

* La variable **ΣDAT** peut être modifiée par **Σ+** et **Σ-** (ajout et retrait de lignes dans la matrice statistique).
* La variable **ΣPAR** (paramètres statistiques) peut également être modifiée par **XCOL**, **YCOL**, **BARPLOT**, etc...
* La variable **PPAR** (paramètres de tracé) peut également être modifiée par certaines instructions du menu **PLOTR** (**XRNG**, **YRNG**, **INDEP**, **DEPN**, **RES** etc...).

*** SUPPRESSION D'UNE VARIABLE GLOBALE:**

L'instruction **PURGE** permet d'effacer une variable du répertoire courant.
Si par exemple **'A'** est le nom de cette variable, la syntaxe est: **'A' PURGE.**
Le nom **A** disparait alors du menu **VAR** actuel.

Il est possible de détruire simultanément plusieurs variables, en indiquant leurs noms dans une liste. Pour effacer par exemple les variables **A,B** et **C**, on éxécutera la séquence:
{ A B C } PURGE

Remarques:

* La suppression d'une liste de variables est définitive. On sera donc prudent dans ce maniement de **PURGE.**
* Le fait d'éxécuter **PURGE** pour effacer une variable qui n'existe pas en réalité dans le répertoire ne conduit pas à une erreur d'éxécution.
  Ceci est utile , notamment dans un programme , pour s'assurer qu'une variable n'est pas présente dans le répertoire.
  Par exemple, si un programme doit dériver une expression par rapport à une variable **'X'**, il pourra être utile d'insérer **'X' PURGE** au début de ce programme , de manière à ce que la dérivation se fasse d'une manière symbolique (et non en un point particulier).
* Cas particulier: la matrice **ΣDAT** (matrice statistique) peut également être 'purgée' par l'instruction **CLΣ.**

Pour les distraits:

* On peut revenir par **LASTARG** sur l'une des opérations suivantes (à condition de le faire immédiatement)
  - **STO** (et son équivalent <=┐ suivi d'un appui sur la touche de menu correspondant au nom de la variable).
  - **PURGE** (si appliquée à un nom de variable).

  Le nom de la variable est alors placé au niveau 1, et son contenu précédent au niveau 2. Un appui sur **STO** permet alors de rétablir la situation antérieure.

REPERTOIRES SUR LA HP48SX:

*** CREATION D'UN REPERTOIRE:**

Comme on l'a vu dans le chapitre 'Principes de base', il est possible de compartimenter la mémoire disponible en créant une arborescence de répertoires (en anglais, répertoire se dit directory).
Cette arborescence prend sa source dans le répertoire **HOME** (qui existe d'origine) et se ramifie au gré de l'utilisateur.

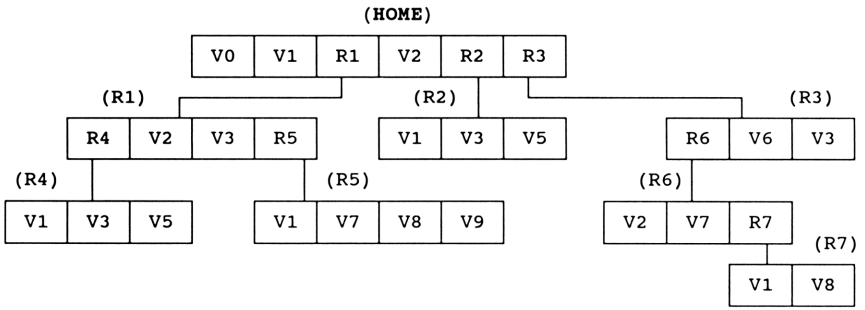
Chaque répertoire peut contenir un nombre quelconque d'entrées de sous-répertoires (que l'on appelle ses répertoires 'fils').
Inversement chaque répertoire, à l'exception de **HOME**,est le fils d'un unique répertoire (qu'on appelle évidemment son 'père').

Dans un répertoire donné, l'instruction permettant de créer une entrée de sous-répertoire est **CRDIR** (Create Directory) qui est décrite dans le paragraphe suivant.
Le nom de ce sous-répertoire apparait alors dans le menu **VAR** et on accède à ce sous-répertoire (vide pour l'instant) en évaluant ce nom (tapez le nom puis **ENTER**, ou appuyez sur la touche de menu correspondant à ce nom dans le menu **VAR**).
Dans le menu **VAR**, les noms de sous-répertoires sont surmontés d'une barre horizontale, afin qu'on les distingue des autres noms de variables.

*** UN EXEMPLE D'ARBORESCENCE:**

Pour en faciliter la lecture, on a représenté les entrées dans chaque menu **VAR**: les noms **V0**, **V1**, **V2**, etc.. désignent des variables, les noms **R1**, **R2**, etc... désignent des entrées de sous-répertoires.

Au dessus de chaque menu **VAR**, on a fait figurer entre parenthèses le nom du répertoire concerné.



*** COMPRENDRE ET UTILISER LA NOTION DE REPERTOIRE:**

Quelques éclaircissements s'imposent, et le schéma précédent permet de mieux comprendre les remarques qui suivent.
D'ailleurs , dans tout ce paragraphe, les exemples utilisés reprendront les notations de ce schéma.

* Un répertoire peut contenir un nombre quelconque de variables et d'entrées de sous-répertoires (la seule limitation est fournie par la quantité de mémoire disponible).

* Dans un répertoire, un nom ne peut être utilisé qu'une fois.

* On peut utiliser un même nom (pour contenir le même objet ou des objets différents), à condition que ce soit dans des répertoires distincts.

* Problèmes d'homonymie et de visibilité:

Supposons que nous ayons créé une variable nommée **A** dans un répertoire **R**.

On ne peut rappeler **A** (par **RCL**), la visiter (par **VISIT**) la purger (**PURGE**) que si on se trouve dans le répertoire **R**.

Néanmoins, le domaine d'où l'on peut 'voir' une variable ne se limite pas forcément au répertoire où elle a été créée.

Par 'voir' une variable, on entend ici la possibilité de l'évaluer (c'est-à-dire d'en utiliser le contenu).

Supposons que, dans un répertoire donné **R**, on veuille évaluer une variable nommée **V**. On éxécute par exemple **V ENTER**, ou **'V' EVAL**.
- La HP48SX cherche d'abord le nom **V** dans le répertoire **R**: Si elle le trouve, elle évalue ce nom.
- Si ce nom n'appartient pas au répertoire courant, alors la HP48SX remonte dans l'arborescence (de **R** vers **HOME**), jusqu'à ce qu'elle rencontre le nom **V** dans un des répertoires 'ancêtres' de **R**.
- Le nom V est alors évalué, avec la valeur qui correspond à la 1ère localisation dans cette recherche ascendante.
- Evidemment le nom **V** peut très bien n'être pas localisé dans cette recherche (auquel cas **V ENTER** dépose **'V'** au niveau 1 de la pile).
- Attention : cette recherche ne se traduit pas, pour l'utilisateur, par un changement de répertoire.

Vu 'depuis le haut' de l'arborescence, on peut donc dire qu'une variable créée dans un répertoire particulier est visible dans ce répertoire et dans toute sa 'descendance' (sauf bien sûr si ce nom est réutilisé à un endroit quelconque de cette descendance, auquel cas cette nouvelle variable occulte l'ancienne dans le reste de la descendance).

Exemples: nous utilisons l'arborescence décrite ci-dessus.

Si on est dans le répertoire **R5**:
* La variable **V1** est présente dans **R5**. Elle l'est également dans **HOME**. Néanmoins c'est sa valeur dans **R5** qui est considérée.
* La variable **V2** n'est pas présente dans **R5**.Elle l'est dans **R1** et dans **HOME**.C'est le contenu qu'elle a dans **R1** (le répertoire père de **R5**) qui est pris en considération.

Si on est dans le répertoire **R7**:
* La variable **V0** n'y est pas présente,non plus que dans **R6**, non plus que dans **R3**, mais elle l'est dans **HOME**. C'est cette présence qui est détectée si on cherche à évaluer **V0** à l'intérieur de **R7** (une telle évaluation ne nous fait pas pour autant changer de répertoire).
* La variable **V9** (qui n'est présente que dans **R5**) n'est pas visible dans **R7** (car le répertoire **R5** n'est pas rencontré sur la route ascendante qui mène du répertoire **R7** au répertoire **HOME**).

Si on est dans **HOME**, seules les variables **V0**, **V1**, **V2** (avec le sens qu'elles ont dans **HOME**) et les entrées **R1**, **R2**, **R3** de de sous-répertoires sont visibles.

La variable **V0**,qui est définie dans **HOME** et dont le nom n'est repris dans aucun sous-répertoire est visible de partout.

La variable **V2**, telle que définie dans HOME, est visible dans **HOME**, **R2**, et **R3**. Dans **R1**,**R4**,**R5**, c'est sa définition de **R1** qui est utilisée. Dans **R6** et **R7**, c'est celle de **R6**.

* DEPLACEMENTS DANS L'ARBORESCENCE:

Le passage à un répertoire s'effectue en tapant son nom puis **ENTER** (ou, bien sûr, en plaçant ce nom dans un programme, ou encore en appuyant sur la touche de menu si ce nom de sous-répertoire y apparait).

Ce changement de répertoire ne s'accompagne pas d'un affichage du menu **VAR** du nouveau répertoire, sauf si celui du répertoire initial était affiché au moment du déplacement.

Notons tout d'abord qu'on peut accéder à **HOME** depuis partout. (faites **HOME ENTER**, ou bien appuyez sur la touche **HOME**).
De même la touche **UP** du clavier (l'instruction correspondante est **UPDIR**) permet de monter d'un niveau dans l'arborescence. Elle est sans effet si on est dans **HOME**.

Il y a, comme pour l'évaluation des variables, des règles de visibilité à respecter.

Supposons que l'on soit dans un répertoire source **S** , et que l'on veuille aller dans un répertoire but **B**, en une seule étape, par **B ENTER**:

* La HP48SX examine le menu **VAR** du répertoire **S**. Si elle y trouve le nom **B** (c'est-à-dire si **B** est un répertoire fils de **S**),alors le changement de répertoire s'effectue.

* Sinon, la recherche continue de manière ascendante, en direction de **HOME**, jusqu'à ce que le nom **B** soit détecté dans le menu **VAR** d'un répertoire **P** . Le changement de répertoire est alors effectué (**P** est ici le répertoire père de **B**).

* Si le nom **B** n'est pas détecté dans cette recherche ascendante, la séquence **B ENTER** dépose **'B'** au niveau 1 de la pile.

Exemples:

En reprenant le schéma ci-dessus, voyons quels déplacements sont possibles en une seule étape (tous les déplacements sont évidemment possibles en plusieurs étapes).

Depuis **HOME**, on peut accéder à **R1**, **R2**, **R3**.

Depuis **R1**, on peut accéder à **R4** et **R5** (les fils de R1), à **HOME**, mais aussi à **R2** et à **R3** (la séquence: **R2 ENTER** provoque une recherche ascendante; le nom **R2** est trouvé dans **HOME**, et le changement de répertoire est effectué).

Depuis **R7**, on peut accéder à **R6**, **R3**, **HOME**, **R1** et **R2**.

Pour aller de **R7** à **R1**, il suffit de faire **R1 ENTER**.

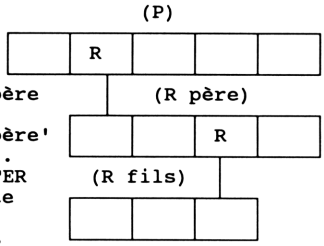
Pour aller de **R1** à **R7**, il faut faire successivement:
R3 ENTER (passage de **R1** à **R3**),
R6 ENTER (passage de **R3** à **R6**),
R7 ENTER (passage de **R6** à **R7**).
ou encore **{ R3 R6 R7 } EVAL**.

Remarque:

Il peut se produire des homonymies de répertoires (bien qu'on n'en voit pas l'intérêt pratique).

Supposons que l'on soit dans un répertoire nommé **R**, et que dans ce répertoire, on éxécute l'instruction **'R' CRDIR.**

On crée ainsi un répertoire fils de **R** (qui porte le même nom que son père...).



On a représenté la situation sur le schéma ci-contre, où on appelle P le répertoire-père du répertoire R initial.

Si on éxécute R ENTER dans 'R-père' alors on passe dans 'R-fils'.

Par contre, si on éxécute R ENTER dans 'R-fils', alors on reste dans 'R-fils'.

Pour accéder à 'R-père' depuis 'R-fils', il faut par exemple éxécuter **UPDIR**, ou bien encore la séquence: **P R ENTER**, c'est-à-dire faire un crochet par le répertoire **P**.

*** SUPPRIMER UN SOUS-REPERTOIRE:**

L'instruction permettant de supprimer un sous-répertoire est **PGDIR** (dans le menu **MEMORY**). Cette instruction n'est pas "rattrapable" par **LASTARG** (prudence!!).

On peut supprimer un sous-répertoire par **PURGE** à condition qu'il soit vide.

On peut supprimer le contenu du répertoire en cours par **CLVAR** (voir plus loin).

* LES OBJETS-REPERTOIRES:

Un répertoire est une variable en ce sens qu'il a un nom et un contenu. Celui-ci est appelé objet-répertoire et est un des types d'objets reconnus par la HP48SX (l'instruction **TYPE**, appliquée à un objet-répertoire, donne le résultat 15).

Exception: le contenu de **HOME** n'est pas un objet-répertoire.

Supposons par exemple que **'REP'** soit le nom d'un répertoire.

L'objet-répertoire contenu dans **'REP'** peut être placé au niveau 1 de la pile:

- En éxécutant **'REP' RCL.**
- En actionnant ╔═> puis la touche de menu correspondant au nom **'REP'** (à condition que le menu **VAR** soit affiché et que ce nom y soit visible).

Cet objet-répertoire peut alors être stocké dans une variable au moyen, par exemple de l'instruction **STO.**

Attention:

On ne peut pas stocker un objet-répertoire dans une variable dont le contenu est déjà un objet-répertoire (même si ce dernier est vide).

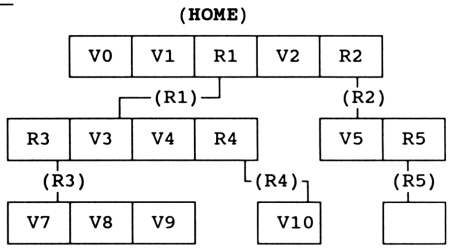
Une conséquence importante est qu'on ne peut modifier le contenu d'une variable désignant un répertoire en utilisant **VISIT** (ou toute opération de "visite" équivalente).

Une telle "visite" est autorisée, mais un appui sur **ENTER** provoquerait une erreur (on sort donc par **ON**).

Par contre, un objet-répertoire peut être modifié par **EDIT.**

* EXEMPLES D'OBJETS-REPERTOIRES:

Considérons l'arborescence ci-contre:



R1,R2,.., R5 sont des noms de répertoires.
V0,V1,..,V10 sont des noms de variables.
Pour simplifier, on suppose que le contenu de V0 est 0, que celui de V1 est 1 etc... et que celui de V10 est 10.
On remarque que le contenu de R5 est vide.
Voici l'aspect que prennent sur la pile les objets-répertoires correspondants aux noms 'R1',..,'R5', tels qu'ils peuvent être obtenus par **RCL** (ou ╔═> suivi du label ad-hoc du menu **VAR**).

'R5': (se placer dans **R2** et faire **'R5' RCL**, ou ╔═> **R5**)
L'objet-répertoire est

```
DIR
END
```

'R4': L'objet-répertoire est

```
DIR
  V10 10
END
```

'R3':

```
DIR
  V7 7
  V8 8
  V9 9
END
```

'R2':

```
DIR
  V5 5
  R5
    DIR
    END
END
```

'R1':

```
DIR
   R3
      DIR
        V7 7
        V8 8
        V9 9
      END
   V3 3
   V4 4
   R4
     DIR
       V10 10
     END
END
```

On voit sur ces exemples la façon dont est présenté un objet-répertoire:
Il commence par DIR, se termine par END.
Entre ces deux extrêmes apparaissent les noms des variables de ce répertoire, suivis de leur contenu.
En particulier, les sous-répertoires du répertoire ainsi décrit apparaissent avec leur nom suivi de l'objet-répertoire correspondant.

Remarque: Les décalages à droite qui indiquent la profondeur dans l'arborescence n'apparaissent que lorsque l'on édite l'objet répertoire en ligne de commande.

Il est possible de créer soi-même un objet-répertoire:
La séquence: **DIR V1 (1,2) REP DIR V2 "ABC" (ENTER)**

place l'objet répertoire ci-contre sur la pile.
Il suffit alors de faire
'REP1' STO
pour que cet objet-répertoire devienne le contenu d'un nouveau répertoire baptisé 'REP1'.

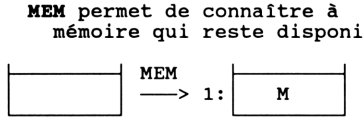
```
DIR
   V1 (1,2)
   REP2
      DIR
        V2 "ABC"
      END
END
```

DANS LE MENU MEMORY:

Le menu **MEMORY** regroupe des instructions qui ont un rapport étroit avec les notions de variable globale et de répertoire.
N.B: certaines instructions de ce menu, notamment celles qui traitent des objets-sauvegardes, ne sont pas décrites ici.

MEM

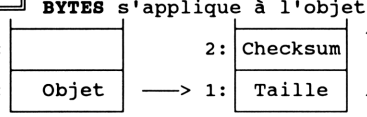
MEM permet de connaître à tout moment la quantité M de mémoire qui reste disponible pour l'utilisateur.



Cette quantité est exprimée en octets (8 bits).
La cellule de base pour la HP48SX étant le quartet (4 bits), la quantité de mémoire disponible est donnée avec une précision de un demi-octet.

BYTES

BYTES s'applique à l'objet placé au niveau 1, et renvoie:

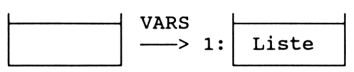


Au niveau 2:
Une somme de contrôle,sous forme d'un entier binaire.
Au niveau 1:
La taille de cet objet, exprimée en octets.

La somme de contrôle permet de détecter une éventuelle erreur dans la copie de l'objet (si l'objet a par exemple été copié dans un livre où la somme de contrôle est indiquée).
Si l'objet au niveau 1 est un nom, celui-ci doit être un nom de variable (éventuellement de répertoire) présent dans le menu **VAR** actuel. La taille et la somme de contrôle obtenues sont relatives au couple (nom de variable,contenu de variable).

VARS

L'instruction **VARS** donne la liste de tous les noms de variable et de sous-répertoire qui apparaissent dans le menu **VAR** actuel (et dans l'ordre où ces noms apparaissent).



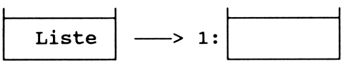
La liste obtenue est vide si le répertoire dans lequel on se trouve est vide.
L'utilisation de **VARS** (suivie par une modification de la liste obtenue par **EDIT**) précède souvent un appel de **ORDER** (voir ci-dessous).

ORDER

Redéfinir l'ordre des variables dans le menu **VAR**.

On sait que la création d'une variable globale , ou d'un sous-répertoire, se traduit par l'apparition d'un nom en tête du menu **VAR** actuel.
L'utilisateur peut vouloir modifier l'ordre dans lequel apparaissent les noms du menu **VAR**. C'est le rôle que remplit l'instruction **ORDER**.
Il est ainsi recommandé de placer à des positions adjacentes des noms ayant des liens de signification.
Les noms les plus fréquemment utilisés dans le répertoire peuvent être placés dans les 6 premières positions (de façon à être accessibles immédiatement, sans qu'il soit nécessaire d'utiliser les touches **NEXT** ou **PREV**).

Le schéma fonctionnel de **ORDER** est:



Le résultat apparaît sur la ligne de menu si le répertoire **VAR** est affiché.

'Liste' doit être une liste de noms (variables ou sous-répertoires) du menu **VAR**.

Les noms qui apparaissent dans 'Liste' formeront (dans l'ordre où ils apparaissent dans cette liste),les premières entrées du nouveau répertoire **VAR**.

Principe du fonctionnement de **ORDER**:

Supposons que Liste = { Nom_1, Nom_2, ..., Nom_(n-1), Nom_n }
ORDER s'intéresse successivement aux noms:

Nom_n, Nom_(n-1), ... , Nom_2, Nom_1

c'est à dire que 'Liste' est parcourue à l'envers.

Pour chaque Nom_k ainsi rencontré:

ORDER retire le nom 'Nom_k' de la pile, puis le place en position 1 dans le menu **VAR**.

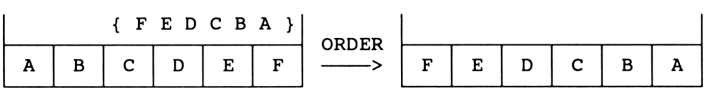
Evidemment, si ce nom n'apparaît pas dans le menu **VAR**, le fonctionnement de **ORDER** s'interrompt sur une erreur (Undefined Name). Néanmoins, au moment où cette erreur est décelée, certains déplacements peuvent avoir déjà être réalisés.

On comprend que les noms du répertoire **VAR** qui ne figurent pas dans 'Liste' soient relégués après ceux qui y figurent.

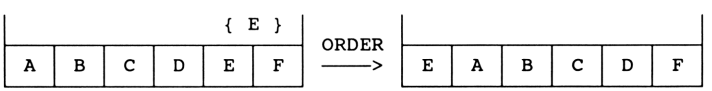
Il est notamment recommandé (ne serait ce que pour des raisons de rapidité d'éxécution) de limiter 'Liste' aux seuls noms qui doivent être déplacés les uns par rapport aux autres.

Exemples d'utilisation de **ORDER**:

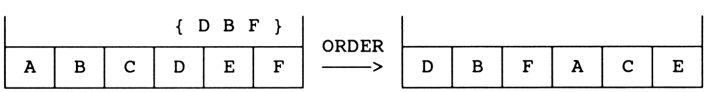
Supposons que les 6 premières entrées du menu **VAR** soient A, B, C, D, E, F (noms relatifs à des variables ou à des sous-répertoires).



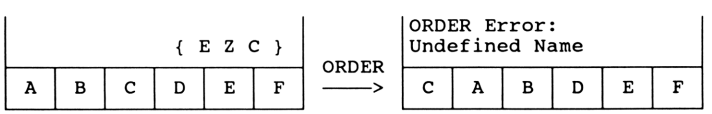
Ici, les 6 premières entrées du répertoire sont concernées.



Ici, on se contente de ramener 'E' en première position.



Ici, A,C,E sont relégués après D,B,F.



Ici, il y une erreur d'éxécution (car 'Z' n'est pas un nom du répertoire). On constate néanmoins que le nom 'C' a été placé en tête du répertoire, ce qui démontre bien qu'**ORDER** parcourt à l'envers la liste qu'on lui donne.

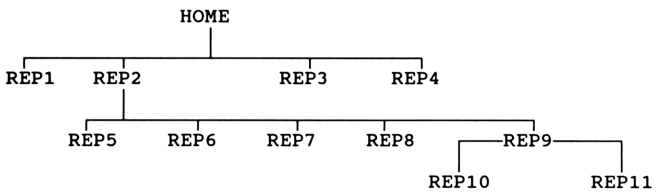
PATH

PATH donne le chemin permettant d'aller de **HOME** au répertoire **VAR** dans lequel on se trouve.

'Chemin' est ici une liste dont le premier élément est **HOME**, dont le dernier élément est le nom du répertoire où on se trouve, et dont les éléments intermédiaires sont les noms des sous-répertoires qui permettent de passer de **HOME** au répertoire **VAR** actuel.

Remarque: Dans le cas d'un affichage normal de la pile à l'écran, le chemin conduisant de **HOME** au répertoire actuel apparaît sur la ligne supérieure de l'écran (ligne d'état).

Supposons par exemple que l'arborescence de votre HP48SX soit:



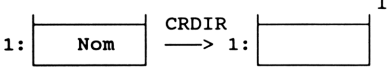
où **REP1**, **REP2**,.., **REP11** désignent les différents répertoires et sous-répertoires (Par ex: les noms **REP1**, **REP2**, **REP3**, **REP4** apparaissent dans le menu **HOME**, les noms **REP10**, **REP11** sont dans le menu **VAR** correspondant au répertoire **REP9**).

Si on est dans **HOME**, **PATH** donne: { **HOME** }
Si on est dans **REP3**, **PATH** donne: { **HOME** **REP3** }
Si on est dans **REP6**, **PATH** donne: { **HOME** **REP2** **REP6** }
Si on est dans **REP10**, **PATH** donne:{ **HOME** **REP2** **REP9** **REP10** }

Rappel: Il est possible d'évaluer une liste par **EVAL**.Dans le cas où cette liste est un chemin, cette évaluation permet de "parcourir" ce chemin, en partant du répertoire courant (il faut évidemment que ce chemin soit correct).

CRDIR

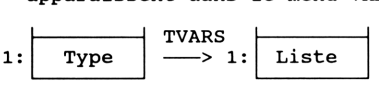
L'instruction **CRDIR** permet de créer un sous-répertoire dans le répertoire où on se trouve à ce moment là.
Le schéma fonctionnel est:



Ici, 'Nom' est le nom du répertoire à créer. Ce nom ne doit pas être déjà un nom de sous-répertoire dans le menu **VAR** actuel.

Attention: Si 'Nom' est le nom d'une variable de votre menu **VAR** alors cette variable est supprimée puis le sous-répertoire est créé.
La création du sous-répertoire se traduit par l'apparition de son nom dans le menu **VAR** actuel (le nom étant surmonté d'une petite barre, propre aux noms de sous-répertoires).
Un appel de ce nom vous fera passer dans ce sous-répertoire, vide pour l'instant.

TVARS L'instruction **TVARS** donne la liste de tous les noms de variable (du menu **VAR** actuel) dont le contenu est du type spécifié au niveau 1 (liste établie dans l'ordre où les noms apparaissent dans le menu **VAR**)



La liste obtenue est vide si le répertoire actuel ne contient pas de variable dont le conte-nu soit du type spécifié.

Par exemple: la séquence **15 TVARS ORDER** sélectionne la liste de tous les sous-répertoires du répertoire actuel, puis réorganise **VAR** pour que ces sous-répertoires apparaissent en tête.

LE MENU STORE:

Le menu **STORE** (pour reprendre le nom qui lui était donné sur la HP28S) est accessible par la séquence ╔=> **VAR**.

Le menu **STORE** contient 9 instructions qui permettent de modifier rapidement le contenu d'une variable globale. Ces instructions ne peuvent cependant s'appliquer qu'à certains types de variables.

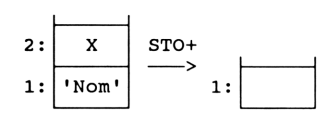
Ces instructions nécessitent un ou deux arguments, dont le nom de la variable à modifier.

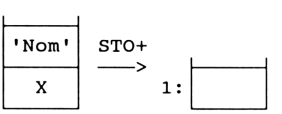
Dans les schémas fonctionnels ci-dessous, 'Nom' désigne le nom de cette variable, et X désigne le deuxième argument que nécessite éventuellement l'instruction décrite. L'opération envisagée entre X et le contenu de la variable 'Nom' doit être "légale".

Les 7 premières instructions du menu **STORE** se réfèrent à une opération arithmétique simple **(+, -, *, /, NEG, INV, CONJ)**.
Il faut ici imaginer que, sur la pile, le nom 'Nom' est remplacé par le contenu de la variable 'Nom', que l'opération arithmétique en question est effectuée, puis que le résultat est placé dans la variable 'Nom'.

N.B: La variable 'Nom' peut aussi bien être une variable locale qu'une variable globale.

STO+

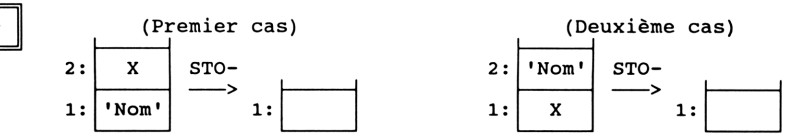




STO+ ajoute X au contenu de 'Nom'. Le résultat va dans 'Nom'.

Bien sûr, le contenu de la variable 'Nom' et l'objet X doivent pouvoir être ajoutés, mais il y a de nombreuses possibilités.
Par exemple:
X ou le contenu de 'Nom' peut être une liste, l'autre étant un objet quelconque.
X et le contenu de 'Nom' peuvent être des tableaux de même format.
X et le contenu de 'Nom' peuvent être des chaînes.
etc.....

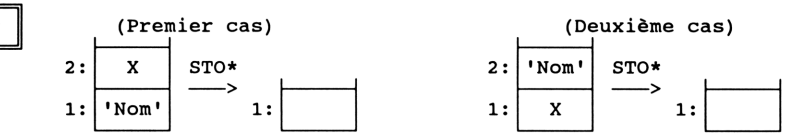
STO-



STO- place dans la variable 'Nom':
Dans le premier cas : X - (contenu de 'Nom').
Dans le deuxième cas: (contenu de 'Nom') - X.

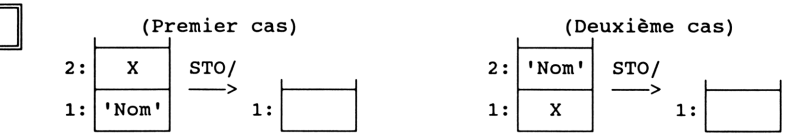
Là encore les limites de l'instruction sont fixées par le domaine de validité de l'opération "soustraction".

STO*



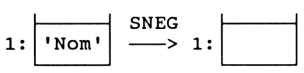
STO* place dans la variable 'Nom':
Dans le premier cas : X * (contenu de 'Nom').
Dans le deuxième cas: (contenu de 'Nom') * X.
Si le contenu de 'Nom' ou si X est un nombre, les 2 cas donnent le même résultat.
Si X et le contenu de 'Nom' sont des tableaux, l'ordre des facteurs est important.

STO/



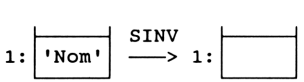
STO/ place dans la variable 'Nom':
Dans le premier cas : (inverse(contenu de 'Nom')) * X.
Dans le deuxième cas: (inverse de X) * (contenu de 'Nom').
L'objet inversé doit être un nombre non nul, ou une matrice carrée inversible.

SNEG



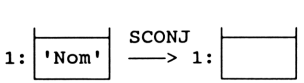
SNEG remplace le contenu de 'Nom' par son opposé.

SINV



SINV remplace le contenu de 'Nom' par son inverse.

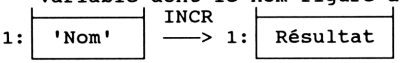
SCONJ



SCONJ remplace le contenu de 'Nom' par son conjugué.

INCR Incrémentation du contenu d'une variable.

L'instruction **INCR** incrémente d'une unité le contenu de la variable dont le nom figure au niveau 1.



Le résultat de cette incrémentation (qui devient le nouveau contenu de la variable) est de plus renvoyé au niveau 1.

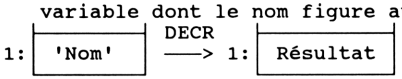
Le contenu de la variable doit ici être un réel.

Si on veut incrémenter le contenu d'une variable sans pour autant renvoyer le résultat au niveau 1, on utilisera:

'Nom' 1 STO+

DECR Décrémentation du contenu d'une variable.

L'instruction **DECR** décrémente d'une unité le contenu de la variable dont le nom figure au niveau 1.



Le résultat de cette décrémentation (qui devient le nouveau contenu de la variable) est de plus renvoyé au niveau 1.

Le contenu de la variable doit ici être un réel.

Si on veut décrémenter le contenu d'une variable sans pour autant renvoyer le résultat au niveau 1, on utilisera:

'Nom' 1 STO-

Les instructions **INCR** et **DECR** sont utiles dans les structures répétitives, s'il faut à la fois incrémenter ou décrémenter un compteur, et tester ensuite la nouvelle valeur de ce compteur.

INSTRUCTIONS DIVERSES:

CLVAR **Effacer toutes les variables d'un répertoire.**

L'instruction **CLVAR** est accessible par ╔═> **DEL** .

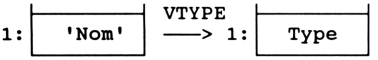
CLVAR efface toutes les variables présentes dans le menu **VAR** actuel . **CLVAR** efface aussi les sous-répertoires s'ils sont vides.

CLVAR est en fait équivalent à **VARS PURGE.**

Remarques:

* Les variables ainsi effacées ne sont pas récupérables. Il faut donc être très prudent avec cette instruction.
* Quand on appuie sur les touches ╔═> **DEL** , le nom **CLVAR** est placé en ligne de commande, et il faut faire **ENTER** pour confirmer.
  Cette exception à la règle commune dénote bien le caractère définitif et dangereux de cette instruction.
* Si le répertoire actuel contient un sous-répertoire non vide, alors **CLVAR** s'interrompt sur une erreur:
  CLVAR Error: Non-Empty Directory.
  Néanmoins, les variables dont le nom précédaient dans la liste donnée par **VARS** sont supprimées (ce qui prouve que **CLVAR** efface les noms de variables dans l'ordre où elles apparaissent dans le menu **VAR**).

VTYPE **Type de l'objet contenu dans une variable.**



L'instruction **VTYPE** donne le type de l'objet contenu dans la va-riable dont le nom a été placé au niveau 1.

Le résultat est un entier compris entre 0 et 19.

Si 'Nom' n'est pas un nom figurant dans le menu **VAR**, alors le résultat est -1.

L'instruction **VTYPE** se trouve dans le menu **PRG OBJ** (page 2), alors que l'instruction **TYPE** se trouve dans cette même page de **PRG OBJ**, ainsi que dans la page 1 de **PRG TEST**.

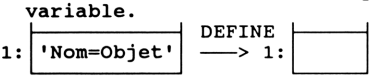
L'instruction **VTYPE** équivaut à peu près à **RCL TYPE**.

Rappelons que les types d'objets reconnus par la HP48SX sont désignés par un entier compris entre 0 et 19:

0: réel.	7: Nom local.	14: Nom XLIB.
1: complexe.	8: Programme.	15: Répertoire.
2: chaîne.	9: Expression.	16: Bibliothèque.
3: R-tableau.	10: Entier binaire.	17: Objet-sauvegarde.
4: C-tableau.	11: Objet-graphique.	18: Fonction intégrée.
5: liste.	12: Objet-marqué.	19: Commande intégrée.
6: Nom global.	13: Objet-unité.	

DEFINE

L'instruction **DEFINE** est un moyen de stocker un objet dans une variable.



On place au niveau 1 une équation dont le membre de gauche est un nom.

DEFINE a pour effet de placer dans la variable 'Nom' l'objet situé dans le membre droit de l'équation (cet objet est donc un nom, une expression ou un nombre réel ou complexe).

L'indicateur -3 a ici un rôle à jouer:

S'il est armé (mode d'évaluation numérique) l'expression du membre droit est calculée avant d'être placée dans la variable.

S'il est désarmé (mode d'évaluation symbolique), l'expression est placée telle quelle dans la variable.

Rappelons que le moyen le plus simple de positionner le flag -3 est d'actionner la touche **SYM** du menu **MODES**:

Si **SYM** est suivi d'un petit carré, alors on est en mode symbolique, sinon on est en mode numérique.

LE CATALOGUE REVIEW:

La touche **REVIEW** (accessible par <=┐ ▽) permet d'obtenir un affichage du catalogue de la page courante du menu **VAR** actuel.

Chaque entrée de cette page du menu **VAR** est ainsi affichée sur une ligne, par son nom suivi de son contenu (ou du début de celui-ci).

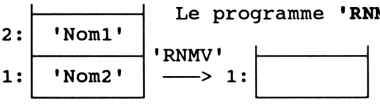
Si le menu **VAR** contient plus de six entrées (c'est-à-dire est formé de plusieurs pages), il convient de se placer d'abord sur la page à afficher, au moyen des touches **NEXT** et **PREV**.

EXERCICES (VARIABLES GLOBALES ET REPERTOIRES):

La difficulté des exercices est mesurée par des symboles *:

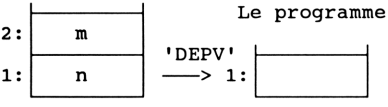
```
*     Très facile.
**    Assez simple.
***   Déjà un peu plus délicat.
****  Plutôt difficile.
***** Difficile.
```

Exercice 1 (***) : Renommer une variable.



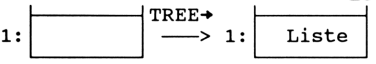
Le programme **'RNMV'** doit renommer la variable dont le nom actuel est 'NOM1', et lui donner le nom 'NOM2'. La position de la variable dans le menu **VAR** doit rester la même.
(Cette variable peut très bien désigner un répertoire).

Exercice 2 (***) : Déplacer une entrée du menu **VAR**.



Le programme **'DEPV'** doit déplacer le nom dont la position est la m-ième dans le menu **VAR**, et le placer en n-ème position dans ce menu.
Ce nom peut aussi bien désigner une variable qu'un répertoire.

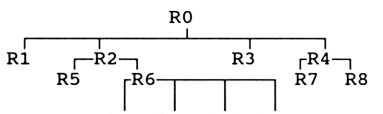
Exercice 3 (*****) : Affichage de l'arborescence.



Le programme **'TREE→'** crée une liste dont le contenu représente l'arborescence de répertoires à partir du répertoire actuel considéré comme racine de cet arbre.

Dans cette liste, chaque nom de répertoire est suivi par la liste des noms de ses sous-répertoires.

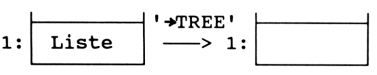
Supposons par exemple que l'on soit dans le répertoire 'R0' que l'arborescence à partir de 'R0' soit:



Alors la liste donnée par **'TREE→'** doit être:

```
{ R0 { R1 R2 { R5 R6 { R9 R10 R11 R12 } } R3 R4 { R7 R8 } } }
```

Exercice 4 (*****): Créer une arborescence.



Le programme **'→TREE'** crée une arborescence, en partant du répertoire actuel, et à l'aide d'une liste donnée au niveau 1

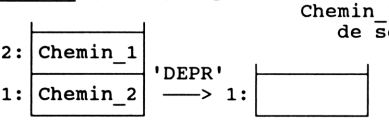
Cette liste est constituée des noms des sous-répertoires qu'il faut créer dans le répertoire de départ, chacun de ces noms 'R' étant suivi par la liste des sous-répertoires qu'il faut éventuellement créer dans le sous-répertoire 'R'.

Si par exemple, on est dans le répertoire 'R0' et que l'on éxécute **'→TREE'** avec la liste:
{ R1 R2 { R5 R6 { R9 R10 R11 R12 } } R3 R4 { R7 R8 } },
alors on crée l'arborescence représentée dans l'exercice 6.

Exercice 5 (****): Exécuter un programme dans toute une arborescence.

'ALLR' permet d'éxécuter le programme Prog (donné par un nom de variable ou par un contenu) dans le répertoire où **'ALLR'** est appelé et dans toute sa descendance de sous-répertoires.

Exercice 6: (*****) Déplacer une arborescence.



Chemin_1 et Chemin_2 sont deux chemins de sous-répertoires tels qu'on peut en obtenir par l'instruction **PATH**. Soit 'Rep1' le dernier nom de la liste Chemin_1, et 'Rep2' le dernier nom de Chemin_2:

'DEPR' transporte dans 'Rep2' le contenu complet de 'Rep1', avec toute l'arborescence de variables et de sous-répertoires qui peut s'y trouver.

Lors du parcours dans l'arborescence décrit par Chemin_2, il peut arriver qu'un nom de répertoire soit inconnu. Dans ce cas, ce répertoire est créé.

EXPRESSIONS CALCUL DIFFERENTIEL ET INTEGRAL

Dans un souci d'aller à l'essentiel, on ne trouvera pas ici la totalité de ce qui peut être dit sur les expressions algébriques.

Sont donc développés ici les points les plus importants, ainsi que les notions qui nécessitent (c'est subjectif) un point de vue différent de celui adopté par le manuel d'utilisation de la HP48SX.

Notamment, on ne revient pas ici sur l'utilisation de l'application **EquationWriter**, car celle-ci est bien décrite, et accompagnée de nombres exemples, dans le manuel d'utilisation de la HP48SX

EXPRESSIONS ALGEBRIQUES:

Si on peut se risquer à une définition, une expression algébrique est une combinaison (correcte tant du point de vue de la syntaxe de la HP48SX que du point de vue de la logique et de la tradition mathématique) d'arguments et de fonctions agissant sur ces arguments.

On parle encore d'opérandes et d'opérateurs.

Le texte qui constitue cette expression algébrique doit être placé entre deux cotes (caractère '), tout comme les noms de variables (qui en un sens sont des expressions particulières bien que la HP48SX les considère comme des objets spécifiques).

Par exemple: **'SIN(X+1)*EXP(X^2+Y^2)+5*(2-LN(3))'**.

* EQUATIONS:

Les équations sont des expressions algébriques particulières. Ce sont celles où l'opérateur "=" est présent.

On peut considérer l'opérateur "=" au même titre que les opérateurs +,-,*,/ par exemple, à ceci près que l'opérateur = ne peut être présent qu'une seule fois dans une expression algébrique.

Le mot équation n'a pas ici le sens mathématique traditionnel.

Par ex: **'Y+Z/EXP(X)=INV(TAN(W))'**, **'1=3'**, sont des équations.

Souvent la HP48SX identifiera l'équation **'LEFT=RIGHT'** et l'expression **'LEFT-RIGHT'** (c'est le cas pour l'instruction **ROOT**). De même l'expression **'EXPR'** sera identifiée avec l'équation **'EXPR=0'**.

Dans certains cas, la différence entre équation et expression algébrique (ne contenant pas le signe =) est significative. Il en est ainsi par exemple pour l'instruction **DRAW**.

* PROGRAMMES AGISSANT COMME DES EXPRESSIONS ALGEBRIQUES:

On dit qu'un programme respecte la syntaxe des expressions algébriques s'il n'utilise aucun objet initialement présent sur la pile, et s'il renvoie un résultat au niveau 1 de celle-ci.
Toute expression algébrique peut être traduite en un progamme qui lui est équivalent:

Par exemple:
L'expression **'LN(X)=2-X'** équivaut au programme « **X LN 2 X - =** »
Le programme « **1 X EXP + Y - Z /** » équivaut à **'(1+EXP(X)-Y)/Z'**.
Par contre « **DUP 1 + X *** » ne respecte pas la syntaxe des expressions algébriques car il utilise l'objet situé au départ au niveau 1 de la pile.
Il en est de même pour « **X 1 + DROP** » qui ne délivre aucun résultat sur la pile.

L'intérêt des programmes par rapport aux expressions algébriques tient à la possibilité qu'ils offrent d'utiliser un plus grand nombre d'instructions.

Par exemple:
Le programme (où A et X sont 2 variables dont les contenus sont respectivement une matrice carrée et un nombre réel):
« **A DUP IDN X * - DET** » (qui calcule DET(A-X*I), où I la matrice identité de même format que A),
respecte la syntaxe des expressions algébriques, mais il ne possède pas d'équivalent sous forme d'un telle expression.
(Car les instructions **DUP IDN** et **DET** y sont interdites).
Il en est de même pour le programme « **X Y R→C SQ C→R +** ».

* SAISIE DES EXPRESSIONS ALGEBRIQUES:

Le texte d'une expression algébrique correspond à un mode de saisie particulier sur la HP48SX (le mode **ALG**, ou le mode **PRG ALG**).
Dans ce mode de saisie, l'appui sur une touche du menu **VAR** (si un menu est présent), ajoute le nom de la variable en ligne de commande (sans cotes), et l'appui sur une touche représentant une fonction ajoute le nom de cette fonction (éventuellement avec une double parenthèse si cette fonction la nécessite).

Par contre, l'appui sur une touche représentant une commande non permise dans les expressions (**ENTER**, **STO**, **SWAP**, **DROP**, etc) provoque une sortie du mode algébrique (et une évaluation de cette commande si on est en mode **ALG**)

* L'INSTRUCTION DEFINE:

L'instruction **DEFINE** (qui est programmable, et qui est également accessible par la touche **DEF** du clavier), permet de définir des "fonctions-utilisateurs" ayant un nombre quelconque d'arguments.
Exemple:
la séquence **'F(X,Y,Z,T)=X*T+EXP(Y*Z)' DEFINE**
crée la variable **F**, considérée comme fonction-utilisateur à quatre arguments.
On pourra utiliser **F** dans les expressions algébriques ou en notation **RPN**.
Ainsi les séquences:
'F(3,2,-1,4)' EVAL et « **3 2 1 NEG 4 F** » **EVAL**
donnent le même résultat (12.1353352832)

* OPERATEURS DANS LES EXPRESSIONS ALGEBRIQUES:

On doit distinguer les types d'opérateurs suivants:
* Les opérateurs unaires, qui ne nécessitent qu'un argument:
* Les opérateurs binaires, qui nécessitent deux arguments;
* Les opérateurs nécessitant plus de deux arguments. Par exemple:
  L'opérateur **IFTE** nécessite trois arguments.
  L'opérateur **Σ** nécessite 3 arguments.
  L'opérateur ∫ nécessite 4 arguments.
* Les fonctions utilisateurs, qui peuvent nécessiter un nombre quelconque d'arguments.

(Les opérateurs soulignés sont inversibles par la HP48SX)

Opérateurs unaires.
Opérateurs arithmétiques: - (ou NEG), √, SQ, INV
Fonctions trigonométriques: ACOS, ACOSH, ASIN, ASINH, ATAN ATANH, COS, COSH, SIN, SINH, TAN, TANH.
Fonctions logarithmiques: ALOG, EXP, EXPM, LN, LNP1, LOG.
Diverses fonctions sur les nombres réels : CEIL, FLOOR, FP, IP, MANT, XPON, !.
Diverses fonctions sur les réels ou complexes: ABS, ARG, CONJ ,IM, RE, RND, SIGN.
Opérateurs de conversion: D→R, R→D.
Opérateur logique : NOT
Opérateur de dérivation: ∂

Opérateurs binaires
Le signe égalité: =.
Opérateurs arithmétiques: +,-,*,/,^ (dérivables et inversibles)
Opérateurs de comparaison: <, ≤, ==, ≠, ≥, >.
Opérateurs logiques: AND, OR, XOR.
Diverses fonctions sur les nombres réels : MAX, MIN, MOD, RND, TRNC, XROOT, COMB, PERM
Opérateurs de pourcentages: %, %CH, %T

* OPERANDES DANS LES EXPRESSIONS ALGEBRIQUES:

Les opérandes (les arguments), dans une expression algébrique, sont les objets sur lesquels agissent les opérateurs (les fonctions).

Il est évidemment nécessaire que le type de l'opérande soit conforme à la nature de l'opérateur qui l'utilise. En cas de non respect de cette condition, l'utilisateur est confronté à un message d'erreur.

EXPRESSIONS.

Le plus souvent, l'expression algébrique est illégale du point de vue syntaxique, ce que signale le message d'erreur **"Syntax Error"** lorsque vous essayez de valider une telle expression.
Ex: **'SIN([1 2 3])'** **'√{1 2}'** **'7 AND #4'** etc....

Parfois l'erreur ne se manifeste que lorsque l'expression est évaluée (on parle alors d'erreur d'éxécution).
Par exemple, on peut entrer **'CEIL((0,1))'** sur la pile, mais une tentative d'évaluation provoque le message **"Bad Argument Type"**.

Les opérandes d'une expression algébrique peuvent être:
* Des nombres réels ou complexes.
* Des noms: **'SIN(X)+Y+1'**
  Ces noms peuvent être des noms de variable existante (donc référencer un contenu particulier), ou au contraire ne correspondre à aucun contenu (on parle alors de variable formelle).
* Des expressions (sans cotes): **'SIN(X+EXP(1+3*(X+7)))+Y'**
  (Ici l'argument de **SIN** est l'expression **X+EXP(1+3*(X+7))**.

*** POSITION PHYSIQUE ET NIVEAU LOGIQUE DANS UNE EXPRESSION:**

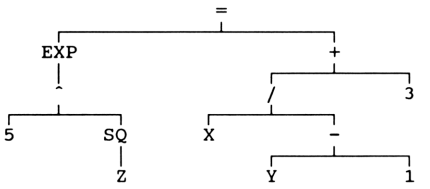
Les expressions algébriques peuvent toutes être traduites en notation polonaise inverse.
C'est sous cette forme qu'elles sont codées en mémoire (elles sont uniquement marquées comme objet algébrique, pour être affichées comme telles si l'utilisateur les rappelle sur la pile).

Cette traduction logique d'une expression algébrique peut être représentée par une structure arborescente, dans laquelle chaque opérateur est le "père" d'un ou de plusieurs objets (son ou ses arguments), suivant qu'il s'agit d'un opérateur unaire ,d'un opérateur binaire, ternaire ou plus, ou d'une "fontion-utilisateur".

Par exemple:
L'expression 'SIN(X)' peut s'écrire

'X+Y' peut s'écrire

Le schéma ci-contre représente la forme logique de l'équation: 'EXP(5^SQ(Z))=X/(Y-1)+3'
On y voit les différents "atomes" constituant cette expression.



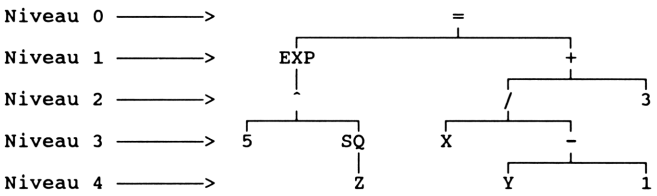
La taille d'une expression est le nombre d'objets dont elle est constituée. La taille de l'expression ci dessus est dont 13.
Dans une expression, un objet se voit attribuer une position "physique" et un niveau logique.
La position physique d'un objet dans une expression indique son rang d'apparition dans celle-ci, quand elle est lue de gauche à droite.

Dans l'équation **'EXP(5^SQ(Z))=X/(Y-1)+3'**, par exemple, les différents objets sont **EXP,5,^,SQ,Z,=,X,/,Y,-,1,+,3** dont les positions physiques sont respectivement 1,2,....,13.

Le niveau logique d'un objet mesure le degré "d'enfouissement" de cet objet dans l'arbre représentant l'expression.

Les niveaux sont comptés à partir du niveau 0 (l'objet situé à la racine de cet arbre) chaque objet "fils" ayant un niveau logique supérieur d'une unité à celui de son objet "père".

Voici quels sont les différents niveaux logiques dans l'expression **'EXP(5^SQ(Z))=X/(Y-1)+3'.**



▮ DANS LE MENU ALGEBRA:

N.B: Certaines instructions du menu **ALGEBRA** ne seront pas décrites ici. C'est le cas notamment des instructions **MATCH↑**, **MATCH↓**, **APPLY**, et **QUOTE**.

COLCT **Regroupements de termes dans une expression.**

1: Expression1 COLCT —> 1: Expression2

COLCT transforme 'Expression1' en 'Expression2' par des évaluations numériques, des regroupements, des factorisations.

Evaluations numériques:
Si une sous-expression ne contient pas de noms, mais uniquement des données numériques, elle est évaluée.
Exemple: **'3^2+7*11*X+45/4*Y'** devient **'9+77*X+11.25*Y'**.

Regroupements dans des sommes:

Dans une suite d'additions ou de soustractions, **COLCT**:
- Regroupe les termes purement numériques:
  Ex: **'3+X-5*Y+7-11*(3+2)'** devient **'-45+X-5*Y'**.
- Réorganise les sous-expressions, en regroupant les termes égaux ou ne différant que par un facteur numérique.
  Ex: **'Y-SIN(X)+3*Y+5*SIN(X)+Y'** devient **'4*SIN(X)+5*Y'**
  '2*X+3*X' devient **'5*X'** (il y a regroupement) .
  '2*X+Y*X' devient **'X*Y+2*X'** (pas de regroupement).

Regroupements dans des produits:

Dans une suite de produits ou de quotients, **COLCT**:
- Regroupe les termes purement numériques:
  Ex: **'7*(X*3)/(2*(5+11))'** devient **'.65625*X'**.
- Réorganise les sous-expressions, en regroupant les termes égaux (calcule donc les exposants qui résultent de ces regroupements):
  Ex: **'X^A*Y/(X*X)*SQ(Y)*INV(X)'** devient **'X^(-3+A)*Y^3'**

EXPAN **Développer les produits ou les puissances.**

1: Expression1 —EXPAN—> 1: Expression2

EXPAN transforme 'Expression1' en 'Expression2' en y développant (au moins partiellement) les produits (* ou / ou SQ) et les puissances (^).

Les expressions (ou sous-expressions) développables par **EXPAN** sont les suivantes (X,Y,Z désignent ici des noms ou des expressions quelconques), en espérant n'oublier personne...

Avant	Après EXPAN	
-(-X)	X	(Dans chaque ligne le signe ± signifie + ou -)
SQ(X)	X*X	(idem avec X*X)
SQ(X ± Y)	X^2 ± 2*X*Y + Y^2	
X^9	X*X^8	(idem avec X^n, où n est un entier positif)
X^(Y + Z)	X^Y * X^Z	
X^(Y - Z)	X^Y / X^Z	

Avant	Après EXPAN
X*(Y ± Z)	X*Y ± X*Z
(X ± Y)*Z	X*Z ± Y*Z
(X ± Y)/Z	X/Z ± Y/Z
X/(Y ± Z)	INV(INV(X)*Y ± INV(X)*Z)

Avant	Après
INV(X*Y)	INV(X)/Y
INV(X/Y)	INV(X)*Y
INV(INV(X))	X
INV(X^Y)	X^(-Y)
INV(X^(-Y))	X^Y

Comment fonctionne EXPAN:

EXPAN ne tente pas d'effectuer tous les développements possibles en une seule passe.

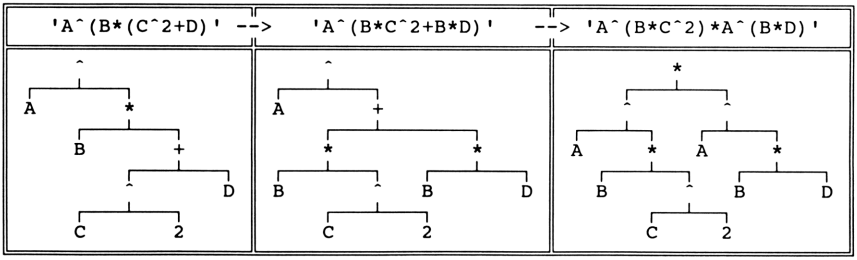
Au contraire, **EXPAN** effectue une recherche de la première sous-expression à développer, la recherche s'effectuant à partir du niveau logique 0 (la racine de l'arbre figurant cette expression), en progressant vers les "feuilles" de l'arbre (c.a.d vers les objets les plus "imbriqués").

Chaque direction de l'arbre est ainsi explorée jusqu'à ce qu'une sous-expression soit développable. Celle-ci est développée, ce qui stoppe la progression de **'EXPAN'** dans cette direction.

Le manuel d'utilisation de la HP48SX propose un bon exemple de ce fonctionnement par étapes.

L'expression **'A^(B*(C^2+D)'** devient ainsi (après chaque appel de **EXPAN**): **'A^(B*C^2+B*D)'** , puis **'A^(B*C^2)*A^(B*D)'**, puis enfin **'A^(B*(C*C))*A^(B*D)'**.

Voici représentées les deux premières étapes:



Remarques:
Contrairement à **EXPAN**, **COLCT** fait le plus gros du travail en une seule passe. Une expression modifiée par **COLCT** devient alors, souvent, invariante par cette instruction.
Pour une réduction complète d'une expression, il est utile d'utiliser conjointement **EXPAN** (développer les produits) et **COLCT** (regrouper les termes dans les sommes consécutives à ces développements).

Exemple:
Soit à développer l'expression **'(X+2*Y)*(Y-3*X)*(X+Y)'**
On applique plusieurs fois **EXPAN**, puis une fois **COLCT**.

1) **EXPAN** donne: **'(X+2*Y)*(Y-3*X)*X+(X+2*Y)*(Y-3*X)*Y'**

2) **EXPAN** donne: **'((X+2*Y)*Y-(X+2*Y)*(3*X))*X**
+((X+2*Y)*Y-(X+2*Y)*(3*X))*Y'

3) **EXPAN** donne: **'(X+2*Y)*Y*X-(X+2*Y)*(3*X)*X**
+((X+2*Y)*Y*Y-(X+2*Y)*(3*X)*Y)'

4) **EXPAN** donne: **'(X*Y+2*Y*Y)*X-(X*(3*X)+2*Y*(3*X))*X**
+((X*Y+2*Y*Y)*Y-(X*(3*X)+2*Y*(3*X))*Y)'

5) **EXPAN** donne: **'X*Y*X+2*Y*Y*X-(X*(3*X)*X+2*Y*(3*X)*X)**
+(X*Y*Y+2*Y*Y*Y-(X*(3*X)*Y+2*Y*(3*X)*Y))'

6) **COLCT** donne: **'-(3*X^3)-8*X^2*Y-3*X*Y^2+2*Y^3'.**

QUAD et ISOL — Solution générale ou valeur principale.

QUAD et **ISOL** résolvent, d'une manière symbolique, des problèmes qui peuvent admettre un certain nombre, voire une infinité, de solutions distinctes.
Ces solutions diffèrent les unes des autres par la valeur donnée à un ou plusieurs paramètres dont les noms sont fixés par la HP48SX.

On distingue:
Des paramètres de signe: **'s1'**, **'s2'**, **'s3'**, etc....
Des paramètres entiers quelconques: **'n1'**, **'n2'**, **'n3'**, ...

La HP48SX utilise en priorité les noms **'s1'** et **'n1'**. Dans la mesure où l'expression obtenue par **QUAD** ou **ISOL** nécessite plusieurs paramètres de signes ou entiers, alors elle utilise les noms suivants (dans l'ordre croissant des indices) .

Chacun de ces paramètres possède une valeur dite "principale":
La valeur principale de **'s1'**, **'s2'**, **'s3'** etc... est 1.
La valeur principale de **'n1'**, **'n2'**, **'n3'** etc... est 0.

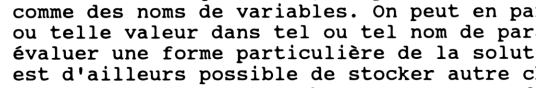
On peut vouloir obtenir l'expression générale de la solution (avec les paramètres éventuels) ou l'expression "principale" qui serait obtenue à partir de la solution générale en donnant aux éventuels paramètres leur valeur principale.

EXPRESSIONS.

Le flag -1 de la HP48SX vous permet de choisir:
S'il est armé (c'est-à-dire s'il contient la valeur 1: pour celà faites **1 +/- SF**), on est en mode "valeur principale".
S'il est désarmé (c'est-à-dire s'il contient la valeur 0: pour celà faites **1 +/- CF**), on est en mode "solution générale".

En mode "solution générale", on peut traiter les noms de paramètres comme des noms de variables. On peut en particulier stocker telle ou telle valeur dans tel ou tel nom de paramètre, de manière à évaluer une forme particulière de la solution générale.
Il est d'ailleurs possible de stocker autre chose qu'un entier dans 'n1' (etc...) et autre chose que -1 ou 1 dans 's1' (etc...) mais quel intérêt?.

ISOL **Isoler la 1-ère occurence d'un nom dans une expression.**



'ISOL' "résoud" Expression1 en isolant la 1-ère occurence explicite de la variable dont le nom est placé au niveau 1. Si cette occurence est argument d'une fonction de la HP48SX, cette fonction doit être inversible.

Si le nom qui est placé au niveau 1 apparaît plusieurs fois dans 'Expression1', seule la première occurence est isolée, et 'Expression2' contiendra les autres références à ce nom.

Remarques:
Contrairement, à **QUAD**, **ISOL** travaille de manière entièrement symbolique: l'expression obtenue n'est pas évaluée avant d'être affichée (même si elle contient des références à des variables ayant un contenu précis) (sauf si on est en mode d'évaluation numérique des fonctions c'est-à-dire si le label **SYM** n'est pas suivi d'un petit carré dans le menu **MODES**).

Dans le même ordre d'idée, **ISOL** ne cherche pas les références implicites (cachées) à la variable à isoler.

Solutions générales:
Dans certains cas, l'inversion de la fonction contenant la première occurence de la variable à isoler conduit à une indétermination: il peut ainsi y avoir, d'un point de vue mathématique, plusieurs solutions, ou même une infinité.
Par exemple:
Isoler **'X'** dans **'SQ(X)=Y'** conduit aux 2 solutions: **X = ±√Y**.
Isoler **'X'** dans **'SIN(X)=Y'** donne une infinité de solutions:
X=(-1)^n*ASIN(Y)+nπ, où n est un entier quelconque.
La forme du résultat dépend alors du mode courant:
* Mode **"Solution générale"** (flag -1 désarmé par **1 +/- CF**).
* Mode **"Valeur principale"** (flag -1 armé par **1 +/- SF**).
(Voir plus haut dans ce chapitre).

Voici, regroupées dans un tableau, les différentes possiblités d'isoler une variable 'X' dans une expression.
Dans ce tableau, Y et Z désignent des expressions quelconques et on suppose que l'on est en mode "solution générale", et en mode "radians".

Avant	Après 'X' ISOL
'COS(X)=Y'	'X=s1*ACOS(Y)+2*π*n1'
'SIN(X)=Y'	'X=ASIN(Y)*(-1)^n1+π*n1'
'COSH(X)=Y'	'X=s1*ACOSH(Y)+2*π*i*n1'
'SINH(X)=Y'	'X=ASINH(Y)*(-1)^n1+π*i*n1'
'ALOG(X)=Y'	'X=LOG(Y)+2*π*i*n1/2.30258509299'
'EXP(X)=Y'	'X=LN(Y)+2*π*i*n1'
'EXPM(X)=Y'	'X=LNP1(Y)+2*π*i*n1'
'X^Y=Z'	'X=EXP(2*π*i*n1/Y)*Z^INV(Y)'
'Y^X=Z'	'X=(LN(Z)+2*π*i*n1)/LN(Y)'

Avant	Après 'X' ISOL	Avant	Après...
'ACOS(X)=Y	'X=COS(Y)'	'ASIN(X)=Y'	'X=SIN(Y)'
'TAN(X)=Y'	'X=ATAN(Y)+π*n1'	'ATAN(X)=Y'	'X=TAN(Y)'
'ACOSH(X)=Y'	'X=COSH(Y)'	'ASINH(X)=Y'	'X=SINH(Y)'
'TANH(X)=Y'	'X=ATANH(Y)+π*i*n1'	'ATANH(X)=Y'	'X=TANH(Y)'
'LN(X)=Y'	'X=EXP(Y)'	'LNP1(X)=Y'	'X=EXPM(Y)'
'LOG(X)=Y'	'X=ALOG(Y)'	'INV(X)=Y'	'X=INV(Y)'
'SQ(X)=Y'	'X=s1*√Y'	'√X=Y'	'X=SQ(Y)'
'-X=Y'	'X=-Y'	'X+Y'	'X=-Y'
'X-Y'	'X=Y'	'X*Y=Z'	'X=Z/Y'
'X/Y=Z'	'X=Z*Y'	'Y/X=Z'	'X=Y/Z'

Remarques:

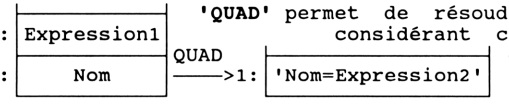
Dans ce tableau, le nombre **π** est obtenu sous cette forme si le flag -2 est désarmé (**2 +/- CF**). S'il est armé (**2 +/- SF**), π est remplacé par sa valeur numérique (3.14159265359).

Il en est de même pour le nombre complexe **i**.

Pour **SIN**,**COS**,et **TAN**, le résultat dépend du mode radians, degrés ou grades.

Par exemple **'SIN(X)=Y' 'X' ISOL** donne:
'X=ASIN(Y)*(-1)^n1+180*n1' en mode degrés.
'X=ASIN(Y)*(-1)^n1+200*n1' en mode grades.

QUAD — Résolution d'une équation du second degré.



'QUAD' permet de résoudre Expression1 en la considérant comme une équation de degré 2 par rapport à la variable dont le nom est placé au niveau 1.

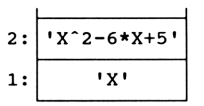
Si Expression1 est effectivement une expression (ou une équation) polynômiale de degré 2 par rapport à la variable indiquée, alors la solution obtenue dans Expression2 est exacte.

Sinon Expression1 est assimilée à son approximation polynômiale degré 2, à l'origine pour la variable spécifiée, obtenue par la formule de Taylor (voir exemple plus loin).

Si on est en mode solution générale (flag désarmé par **1 +/- CF**) l'expression obtenue contient le nom 's1' représentant un signe arbitraire.

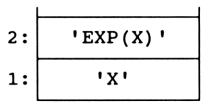
L'instruction **QUAD** évalue l'expression symbolique représentant la solution, avant affichage. On veillera donc à ce que les variables que l'on souhaite voir apparaître de façon formelle dans la solution soient préalablement purgées.

Exemples:



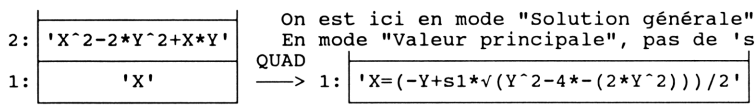
Le schéma ci-dessous est obtenu en mode "solution générale". En mode "valeur principale" on aurait obtenu:'X=5' (c'est la solution particulière obtenue en donnant à 's1' la valeur 1.

On aurait obtenu le même résultat en plaçant au niveau 2 toute expression équivalente à **'X^2-6*X+5'** comme par exemple:
'X^2=6*X-5'; **'X*(X-2)+4=4*X-1'**; etc....



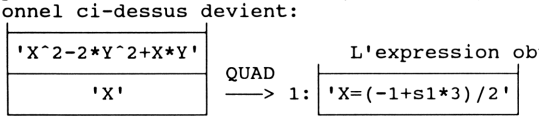
On est ici en mode "solution générale". En mode "valeur principale", on aurait obtenu: 'X=(-1,1)', c'est-à-dire: 'X=-1+s1*(0,1)' avec s1=1.

Dans cet exemple **'EXP(X)'** n'est pas une expression polynômiale par rapport à 'X'.
'QUAD' considère en fait cette expression **F(X)=EXP(X)** et l'identifie avec le polynôme obtenu par la formule de Taylor:
P(X) = F(0)+F'(0)*X+F"(0)/2*X^2 = 1 + X + (X^2)/2.
C'est l'équation **1+X+X²/2** (=0) qui est ainsi résolue (et les 2 solutions de cette équation sont les complexes : -1 ± i).



On est ici en mode "Solution générale". En mode "Valeur principale", pas de 's1*'.

Sur cet exemple,'Y' est une variable formelle (pas de contenu).
Si on effectue **COLCT** on trouve: **'X=.5*(√(9*Y^2)*s1-Y)'.**
Si on place la valeur 1 dans 'Y' (1 'Y' STO), le schéma fonc-tionnel ci-dessus devient:

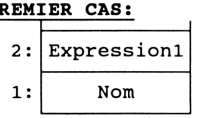


L'expression obtenue ci-dessus a été évaluée.

Remarque: compte tenu de la forme du résultat obtenu avec **QUAD**, l'instruction **DEFINE** permet de stocker, dans la variable du membre de gauche, le contenu du membre droit.

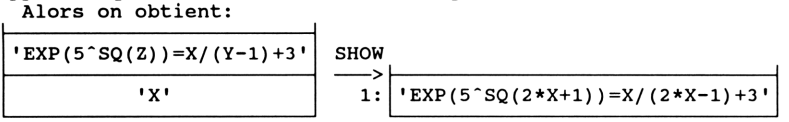
SHOW **Expliciter les références implicites à une variable.**

PREMIER CAS:



L'instruction **SHOW** examine, dans Expression1, s'il se trouve des noms de variables dont le contenu contient une référence au nom placé au niveau 1.
SHOW remplace alors un tel nom de variable par une expression équivalente explicitant cette référence.
Ces références peuvent être directes ou indirectes.

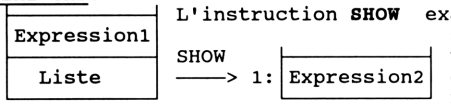
Exemple:
Supposons que 'Y' contienne '2*X' et que 'Z' contienne 'Y+1'.
Alors on obtient:



La référence (directe) de 'Y' à 'X' a donc été explicitée, de même que la référence (indirecte) de 'Z' à 'X'.

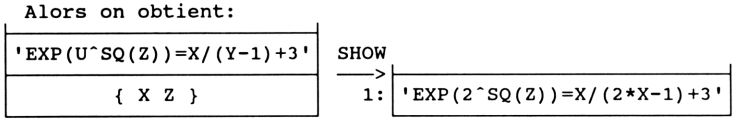
Remarque: dans ce 1-er cas, **SHOW** travaille de manière symbolique, c'est-à-dire qu'elle ne cherche pas à évaluer complètement les noms présents dans l'expression.

DEUXIEME CAS:



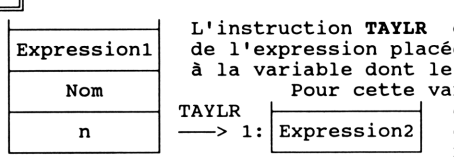
L'instruction **SHOW** examine, dans Expression1, s'il se trouve des noms de variables dont le contenu contient une référence au nom (aux noms) figurant dans la liste présente au niveau 1.
SHOW remplace alors un tel nom de variable par une expression équivalente explicitant cette référence.
Ces références peuvent être directes ou indirectes.
La différence avec le premier cas est qu'ici **SHOW** évalue les noms qui figurent dans 'Expression1' et qui n'apparaissent pas dans la liste.

Exemple:
Supposons que 'Y' contienne '2*X', que 'U' contienne 2, et que 'Z' contienne 0.
Alors on obtient:



La référence (directe) de 'Y' à 'X' a donc été explicitée, de même que U a été évalué (car 'U' ne figure pas dans la liste) alors que 'Z' n'a pas été évalué (il figure dans la liste).

TAYLR **Formule de Taylor à l'origine.**



L'instruction **TAYLR** calcule le développement de l'expression placée au niveau 3, par rapport à la variable dont le nom apparaît au niveau 2. Pour cette variable, Le développement est effectué à l'origine et est poussé à l'ordre n (où n est un entier).

Le résultat est calculé avec la formule suivante:

$$F(0) + \frac{dF(0)}{dX}*X + \frac{d^2F(0)}{dX^2}*X^2 + ... + \frac{d^nF(0)}{dX^n}*X^n \ ,$$

(en appelant X la variable et F l'expression).

Le résultat obtenu par **TAYLR** est une "bonne approximation" de l'expression initiale, pour les valeurs proches de zéros de la variable (la qualité de cette approximation dépendant de beaucoup de choses: le comportement de la fonction, l'ordre n auquel on pousse le développement, la distance de l'origine où on veut utiliser cette approximation ...).

Comment TAYLR travaille:

Supposons que nous voulions appliquer l'instruction **TAYLR** à une expression F (contenant des références aux variables X,Y,Z,...), par rapport à la variable 'X', et à l'ordre n.

TAYLR commence par évaluer l'expression F, pour ce qui est des variables Y,Z,... (normal car l'une d'elles pourrait contenir une référence à X). La variable 'X' n'est pas évaluée.

TAYLR calcule les n premières dérivées de l'expression obtenue, de manière symbolique, par rapport à la variable spécifiée.

Ces n expressions symboliques sont alors évaluées en donnant la valeur 0 à la variable X (sans "écraser" cependant le contenu que pourrait avoir cette variable dans le menu **VAR**, contenu qui va rester intact pendant toute l'opération).

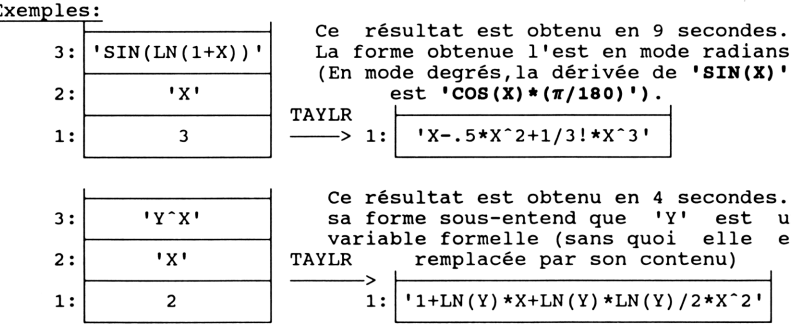
TAYLR forme alors le résultat en utilisant la formule ci-dessus.

Remarques:

Le fait que **TAYLR** calcule n dérivées de manière symbolique implique des temps de calculs parfois importants, ainsi qu'une occupation significative de la mémoire.

Si l'expression initiale est une fonction polynômiale de degré ≤n alors **TALYR** (à l'ordre n) donne la forme développée et exacte de cette fonction ordonnée suivant les puissances croissantes.

Exemples:



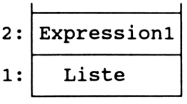
Ce résultat est obtenu en 9 secondes.
La forme obtenue l'est en mode radians.
(En mode degrés,la dérivée de **'SIN(X)'** est **'COS(X)*(π/180)')**.

Ce résultat est obtenu en 4 secondes.
sa forme sous-entend que 'Y' est une variable formelle (sans quoi elle est remplacée par son contenu)

(**WHERE**)

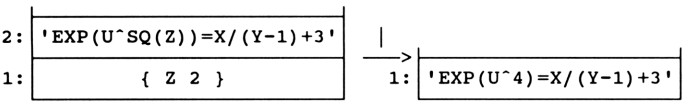
L'instruction **WHERE** permet d'évaluer, au moins partiellement, une expression , en donnant provisoirement un contenu à un ou plusieurs des noms qui y figurent (sans pour autant stocker ces contenus dans ces noms comme le ferait l'instruction **STO**).

Le schéma fonctionnel est:

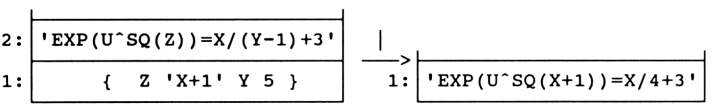


La liste a le format suivant:
{ nom1 exp1 nom2 exp2 ... } où nom1, nom2, etc... sont les noms qu'il faut évaluer, et où exp1, exp2, etc..., sont les contenus qu'il faut leur donner dans cette évaluation.

Exemples:



Ici l'expression a été évaluée par rapport au nom 'Z', auquel on a donné pour l'occasion la valeur 2.
Notez bien que cette opération:
- ne crée pas la variable 'Z' si elle n'existe pas.
- ne modifie pas sa valeur si elle existe.
- n'évalue pas les autres noms dans l'expression, même s'ils correspondent à des variables existantes.



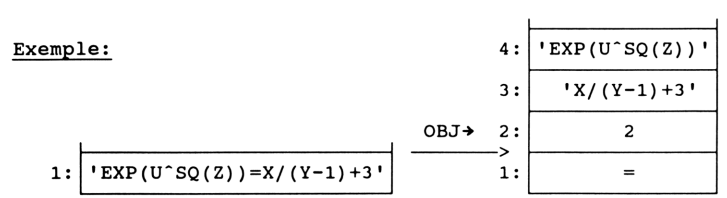
Ici l'expression a été évaluée par rapport au nom 'Z', auquel on a attribué l'expression 'X+1', et par rapport au nom Y, auquel on a donné la valeur 5.

Remarque: L'opérateur WHERE | peut être utilisé dans une expression algébrique.
La syntaxe est alors, en reprenant les notations ci-dessus:
'Expression|(nom1=exp1 , nom2=exp2 ...)'
Le caractère | doit être tapé en mode **ALG** ou en mode **ALG PRG**.
En mode **ALG** l'appui sur | s'accompagne de l'ouverture de parenthèses.
On peut ainsi créer l'expression:
'EXP(U^SQ(Z))|(Z=X+1,U=2)'
dont l'évaluation donne: 'EXP(2^SQ(X+1))'.
Attention: dans cette méthode d'utilisation de |, l'évaluation par **EVAL** est complète (dans l'exemple ci-dessus, X est une variable formelle)

INSTRUCTIONS DIVERSES:

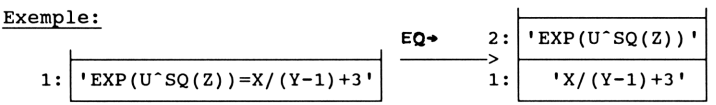
OBJ→ L'instruction **OBJ→** (menu **PRG OBJ**, page 1), appliquée à une expression, la sépare en :
- son opérateur racine d'une part (au niveau 1).
- le nombre d'arguments de cet opérateur (au niveau 2).
- chacun de ces arguments (non évalués), aux niveaux supérieurs (un argument par niveau).

Exemple:



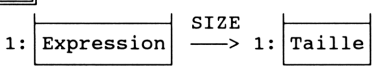
EQ→ L'instruction **EQ→** (menu **PRG OBJ**, page 1), appliquée à une équation, la sépare en :
- son membre de gauche, non évalué, au niveau 2.
- son membre de droite, non évalué, au niveau 1.

Exemple:



Remarque:
Appliquée à une expression qui n'est pas une équation, **EQ→** renvoie cette expression au niveau 2, et le nombre 0 au niveau 1.

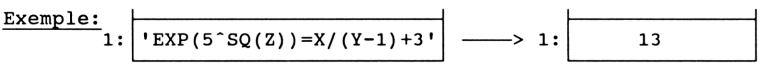
SIZE **Taille d'une expression.**



L' instruction **SIZE** renvoie la taille de l'expression située au niveau 1, c'est-à-dire le nombre d'objets élémentaires dont elle est constituée.

On peut trouver l'instruction **SIZE** dans les menus **PRG OBJ** (p. 3) et **PRG DSPL** (page 2).

Exemple:

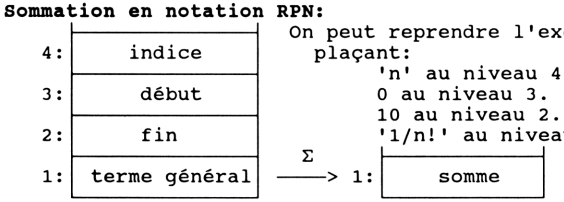


Σ **Sommations.**

L'opérateur Σ permet d'effectuer des sommations, à condition de préciser le terme général de cette sommation, et les valeurs initiale et finale de l'indice de sommation.
On peut utiliser une notation algébrique, ou la notation **RPN**.

Sommation en notation algébrique:
La syntaxe est **'Σ(indice=début,fin,terme général)'**.
Par exemple: évaluer l'expression 'Σ(n=0,10,1/n!)' revient à calculer la somme du terme 1/n!, pour toutes les valeurs entières de l'indice n, de n=0 à n=10 (valeurs comprises) (le résultat est 2.71828180115)

Sommation en notation RPN:



On peut reprendre l'exemple ci-dessus en plaçant:
'n' au niveau 4.
0 au niveau 3.
10 au niveau 2.
'1/n!' au niveau 1.

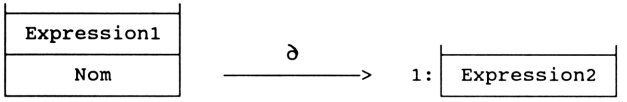
Remarque: dans les deux cas ci-dessus, la sommation s'effectue pour toutes les valeurs entières de l'indice qui vérifient:
début ≤ indice ≤ fin.
En particulier, si la valeur 'début' est supérieure à la valeur 'fin', le résultat de la sommation est 0.

DERIVATION:

La HP48SX vous donne la possibilité de dériver une expression algébrique par rapport à une variable spécifiée.
Il faut distinguer plusieurs aspects de la dérivation:
* La dérivation "pas-à-pas", qui permet comme son nom l'indique de dériver une expression algébrique par étapes.
* La dérivation des fonctions définies par l'utilisateur.
* La définition de nouvelles dérivées par l'utilisateur.
* La dérivation complète (en une seule passe) d'une expression algébrique, au moyen de l'opérateur ∂ .

Pour se limiter à l'essentiel (et au plus utile dans la pratique) nous nous limiterons au quatrième de ces points (de loin le plus fréquemment utilisé). Pour le reste, on se reportera au manuel d'utilisation de la HP48SX.

Le schéma fonctionnel est le suivant:



'Expression2' est le résultat de la dérivation de **'Expression1'** par rapport à la variable dont le nom est placé au niveau 1.

L'opérateur ∂ applique les règles usuelles de dérivation:
Dérivation d'une somme, d'une différence.
Dérivation d'un produit, d'un quotient, d'une puissance.
Dérivation d'une composition de fonctions.
∂ utilise aussi sa bibliothèque de dérivées usuelles.
Voici d'ailleurs les fonctions dérivables par la HP48SX.

Avant ...	et après 'X' ∂	Avant ...	et après 'X' ∂
'COS(X)'	'-SIN(X)'	'ACOS(X)'	'-INV(√(1-X^2))'
'SIN(X)'	'COS(X)'	'ASIN(X)'	'INV(√(1-X^2))'
'TAN(X)'	'1+TAN(X)^2'	'ATAN(X)'	'INV(1+X^2)'
'COSH(X)'	'SINH(X)'	'ACOSH(X)'	'1/(√(X-1)*√(X+1))'
'SINH(X)'	'COSH(X)'	'ASINH(X)'	'INV(√(1+X^2))'
'TANH(X)'	'1/COSH(X)^2'	'ATANH(X)'	'INV(1-X^2)'
'LN(X)'	'1/X'	'EXP(X)'	'EXP(X)'
'LNP1(X)'	'1/(X+1)'	'EXPM(X)'	'EXP(X)'
'SQ(X)'	'2*X'	'√X'	'1/(2*√X)'
'ABS(X)'	'SIGN(X)'	'INV(X)'	'-1/X^2'
'CONJ(X)'	1		
'LOG(X)'	'1(/X*2.30258509299)'		
'ALOG(X)'	'2.30258509299*ALOG(X)'		
'ARG(X)'	'(1/X-CONJ(1/X))/(2*i)'		

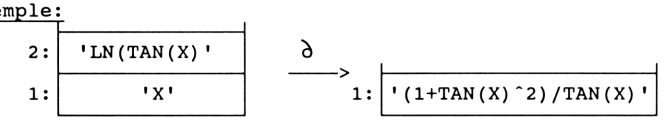
REFERENCES IMPLICITES A LA VARIABLE DE DERIVATION:

La variable de dérivation est souvent présente de façon explicite dans l'expression à dériver. Il se peut également que certains noms dans l'expression fassent référence implicitement à cette variable de dérivation.
Ces références implicites éventuelles sont toutes résolues avant dérivation .Si par exemple, **'Y'** contient **'LN(X)'**, alors la dérivation de **'X*Y'** par rapport à **'X'** donne **'LN(X)+X*(1/X)'**.

Comme le montre l'exemple précédent, l'opérateur de dérivation ne cherche pas à effectuer toutes les simplifications possibles.
Ce problème est d'autant plus épineux si on doit dériver plusieurs fois de suite une même expression. Le résultat peut vite devenir illisible.
Il se pourra donc que l'expression symbolique d'une dérivée mérite quelques 'aménagements' (par **EXPAN** et **COLCT** notamment...)

Il ne faut pas demander l'impossible à la HP48SX, et en particulier de voir toutes les astuces qui peuvent simplifier l'expression obtenue.

Exemple:



Par **EXPAN**, on obtient **'1/TAN(X)+TAN(X)^2/TAN(X)'**
Puis par **COLCT**: **'TAN(X)+INV(TAN(X)'.**
Ne comptez cependant que sur vous même pour montrer que cette dérivée s'écrit plus simplement: **'2/SIN(2*X)'**

INFLUENCE DU MODE DEGRES/RADIANS/GRADES:

L'expression des dérivées des fonctions **SIN, COS, TAN, ASIN, ACOS, ATAN, ARG** dépend du mode en cours.
Pour **'SIN(X)'**, **'COS(X)'** et **'TAN(X)'**:
les dérivées sont mulipliées par **'π/180'** en mode degrés et par **'π/200'** en mode grades.
Pour **'ASIN(X)'**, **'ACOS(X)'**, **'ATAN(X)'** et **'ARG(X)'**:
les dérivées sont mulipliées par **'180/π'** en mode degrés et par **'200/π'** en mode grades.

Encore ces expressions sont-elles obtenues avec l'indicateur -2 désarmé (par **2 +/- CF**), pour que le nombre π reste sous forme symbolique. Si vous armez cet indicateur (par **2 +/- SF**), les expressions π/180 ou 180/π, par exemple, seront remplacées par leur valeur:
π/180≈1.74532925199E-2 et 180/π≈57.2957795131

DERIVATION SYMBOLIQUE OU DERIVATION NUMERIQUE ?:

L'opérateur de dérivation évalue l'expression obtenue avant de l'afficher.
Toutes les variables que vous voulez voir apparaître de manière symbolique dans l'expression de la dérivée (la variable de dérivation mais aussi les autres noms figurant dans l'expression) doivent donc être préalablement purgées.

Si vous voulez calculer la dérivée d'une expression pour une valeur donnée de la variable, il faut:
* Ou bien calculer directement la dérivée en ce point en ayant préalablement mis la valeur voulue dans la variable de dérivation.
* Ou bien calculer la dérivée de manière symbolique, puis placer la valeur considérée dans la variable de dérivation, avant d'évaluer l'expression symbolique de la dérivée (cette méthode permet d'évaluer la dérivée en plusieurs points sans pour autant re-dériver l'expression initiale).

Le rôle de l'indicateur -3:

Ce qui précède s'applique si l'indicateur -3 est désarmé c'est-à-dire si on est en mode d'évaluation symbolique.
Si vous avez armé cet indicateur alors l'opérateur de dérivation cherchera à exprimer numériquement l'expression obtenue (et un message d'erreur apparaîtra si par exemple un nom représente une variable formelle).
Rappelons que le label **SYM**, dans le menu **MODES**, permet de choisir entre le mode d'évaluation symbolique (**SYM** suivi d'un petit carré) et le mode d'évaluation numérique.

■ INTEGRATION:

Le problème est ici de calculer l'intégrale d'une application F(X) sur un segment [a,b], c'est à dire de calculer:

$$\int_a^b F(X)dX$$

Dans tous les cas, il faudra préciser:
La fonction F à intégrer.
La variable X par rapport à laquelle on intègre.
Les extrémités a,b de l'intervalle d'intégration.

On sait que si F est une application continue sur le segment [a,b], alors il existe une infinité d'applications dérivables G(X) sur [a,b] (différant les unes des autres d'une constante) telles que: pour tout x de [a,b], G'(x) = F(x), et :

$$\int_a^b F(X)dX = G(b)-G(a).$$

Tout le problème est de savoir si seule nous intéresse la valeur G(b)-G(a) (intégration numérique) ou s'il nous faut connaître l'expression d'une primitive G de F (intégration symbolique).

La HP48SX permet d'intégrer symboliquement un grand nombre d'expressions. Néanmoins celà n'est pas toujours possible (le problème de l'intégration symbolique est beaucoup plus complexe que celui de la dérivation symbolique).

Nous verrons plus loin quels types de fonctions la HP48SX est susceptible d'intégrer symboliquement.

* INTEGRATION ET NOTATION ALGEBRIQUE:

La syntaxe est la suivante:
'∫(début,fin,fonction,variable)'

L'instruction (la touche) **EVAL** provoque alors le calcul de:
cette intégrale, et le résultat est une expression pouvant contenir des sous-expressions de la forme:
'Expr|variable=fin- Expr|variable=début'
correspondant aux termes qui ont pu être intégrés symboliquement (dans une somme, certains peuvent l'être alors que les autres ne le sont pas),
et des expressions contenant encore le signe d'intégration (et qui correspondent aux termes que la HP48SX n'a pas pu intégrer symboliquement).

Exemples:

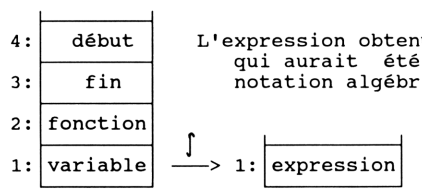
'∫(1,2,1/X,X)' **EVAL** donne:
'1*(LN(X)/∂X(X))|(X=2) - 1*(LN(X)/∂X(X))|(X=1)'.
Une nouvelle utilisation de **EVAL** donne la valeur numérique de l'intégrale, c'est-à-dire: .69314718056

'∫(1,2,EXP(X^2),X)' **EVAL** donne:
'∫(1,2,EXP(X^2),X)': l'intégration symbolique a échoué.
L'utilisation de **→NUM** donne la valeur numérique de l'intégrale (voir plus loin pour les problèmes de précision de l'intégration numérique).

'∫(1,2,X+EXP(X^2),X)' **EVAL** donne:
'X^(2)/(2)|(X=2)-(X^(2)/(2)|(X=1))+∫(1,2,EXP(X^2)),X)'
On constate qu'ici le terme X a été intégré symboliquement (primitive X^2/2) mais que EXP(X^2) n'a pu l'être.
L'utilisation de **→NUM** donne alors la valeur numérique.

* INTEGRATION SUR LA PILE:

Le schéma fonctionnel est le suivant:



L'expression obtenue ici est la même que celle qui aurait été obtenue en utilisant la notation algébrique.

Résumons quelques observations:

Dans les deux cas ci-dessus (notation algébrique puis EVAL) ou intégration sur la pile, la HP48SX fournit un résultat sous forme d'une expression.

Si cette expression ne contient pas le signe ∫, l'intégration symbolique a réussi, et un appui sur **EVAL** permet d'obtenir le résultat définitif de cette intégration:

* Celui peut ne pas être numérique car il peut subsister des références à des variables formelles.

* Si ce résultat est numérique, il n'y a pas à se poser de problèmes de précision puisque la valeur de l'intégrale résulte de l'évaluation d'une primitive.

Si cette expression contient encore le signe ∫, c'est que l'intégration symbolique a échoué (au moins en partie).
On peut alors obtenir la valeur numérique de l'intégrale en éxécutant **→NUM**. Le problème est alors de connaître la précision du résultat obtenu (voir plus loin).

Il y a deux cas où on peut forcer l'évaluation numérique d'une intégrale (sans rechercher une primitive symbolique):

* On peut utiliser la notation algébrique, et évaluer cette expression par **→NUM** plutôt que par **EVAL**.

* On peut placer le calculateur en mode d'évaluation numérique des fonctions (passer dans le menu **MODES** et actionner la touche de menu correspondant à **SYM**, pour faire disparaître le petit carré après **SYM**).

* RECHERCHE D'UNE PRIMITIVE SYMBOLIQUE:

Les extrémités de l'intervalle d'intégration ne sont pas nécessairement des nombres: elles peuvent être des noms et même des expressions.

Ainsi:
'∫(0,X,1/(1+T^2),T)' **EVAL** donne
'1*(ATAN(T)/∂T(T))|(T=X)-1*(ATAN(T)/∂T(T))|(T=0)', qui après **EVAL** donne (à condition que X soit une variable formelle): **'ATAN(X)'**
Sur cet exemple, on a donc réellement calculé une primitive symbolique de 1/(1+X^2).

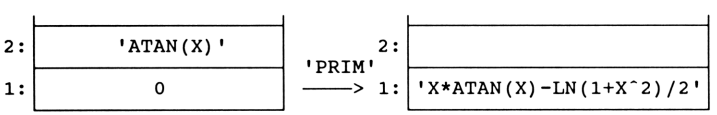
Le programme suivant (nommé ici **'PRIM'**):

« 'X' DUP PURGE ROT OVER ∫ EVAL »

part de la pile suivante:
* Une fonction **F(X)** au niveau 2.
* Un réel **a** au niveau 1.

et calcule (du moins tente de le faire), la primitive G(X) de la fonction F(X) et qui s'annule au point a.

Exemple (en mode radians): en six secondes,



* FONCTIONS PRIMITIVABLES PAR LA HP48SX:

Nous appelerons ici fonctions primitivables celles que la HP48SX est capable d'intégrer symboliquement.

Pour simplifier la description des fonctions primitivables par la HP48SX, nous les considérerons comme fonctions d'une variable symbolique X.

* Si F(X) est la dérivée d'une fonction G(X) intégrée à la HP48SX, alors F(X) est primitivable.
Il suffit donc de lire "à l'envers" le tableau des dérivées présenté plus haut pour prendre connaissance d'un certain nombre de fonctions primitivables.

Par exemple:
La HP48SX sait que:
une primitive de COS(X) est SIN(X).
une primitive de 1/√(1-X²) est ASIN(X).
une primitive de TAN(X)² est TAN(X)-X.
une primitive de 1/X est LN(X).
une primitive de X^a est X^(a+1)/(a+1), où a est un exposant différent de -1.
etc....

* Si F(X),G(X),H(X),etc... sont primitivables, alors la somme aF(X)+bG(X)+cH(X)+... est primitivable.
Par exemple:
La HP48SX sait qu'une primitive de 2*X+3*COS(X) est X^2+3*SIN(X) (en mode radians).
Les fonctions polynômes sont primitivables.

* Si F(X) est primitivable, alors F(aX+b) est primitivable.
Par exemple:
La HP48SX sait qu'une primitive de 1/(2X-3) est LN(2X-3)/2.
Si on veut primitiver 1/(X^2+2*X+2), on n'y arrive pas. Il suffit de l'écrire 1/((X+1)^2+1) pour que la HP48SX donne une primitive égale à ATAN(X+1).

* Les primitives d'un certain nombre de fonctions intégrées à la HP48SX sont primitivables (en plus de celles qui sont déjà des dérivées de fonctions intégrées).
Par exemple:
La HP48SX sait qu'une primitive :
- de LN(X) est X*LN(X)-X.
- de ATAN(X) est X*ATAN(X)-LN(1+X^2)/2.
- de ASIN(X) est X*ASIN(X)+√(1-X^2)
Par contre les primitives de ASINH(X) et de ACOSH(X) sont inconnues de la HP48SX.

* Un certain nombre de primitives autres que celles décrites précédemment sont connues de la HP48SX.
Par exemple:
La HP48SX sait qu'une primitive de '1/(SIN(X)*COS(X))' est: 'LN(TAN(X)'.

* **INTEGRATION NUMERIQUE**:

On se place ici dans le cas où on demande l'évaluation numérique d'une intégrale. La HP48SX utilise un algorithme permettant d'obtenir une valeur approchée de l'intégrale.
Il faut évidemment se poser le problème de la précision du résultat obtenu.

La règle est la précision relative du résultat est indiquée par le format d'affichage en cours:

Plus précisément:
* Si on est en mode **n FIX** (n décimales affichée), la précision relative du calcul est 10^(-n).
En mode **2 FIX**, les intégrales sont donc calculées à 1% près.
* En mode **STD**, le calcul est effectué avec une précision maximum.

Evidemment, le temps de calcul est fonction de la précision souhaitée! (en mode **STD**, ça peut être long).
Dans tous les cas, le calcul numérique d'une intégrale s'accompagne de la création d'une variable nommée **IERR**, et dont le contenu majore l'erreur absolue commise.

Exemple:
On veut calculer l'intégrale de $1/(X^2-3*X+2)$ entre 3 et 4.
On crée **'∫(3,4,1/(X^2-3*X+2),X'** puis on fait **→NUM**.
Le résultat théorique est $Ln(4/3)\approx.287682072449$.
Les résultats sont les suivants:

Mode	Résultat	Temps	IERR
2 FIX	0.29	1 s	2.87E-3
6 FIX	0.287682	6 s	2.876709E-7
STD	0.287682072452	12 s	2.87...E-11

* **INTEGRALES GENERALISEES:**

On peut vouloir calculer des intégrales généralisées avec la HP48 (quand la fonction à intégrer devient "infinie" pour une des bornes d'intégration, ou quand l'une des bornes d'intégration est elle-même à l'infini).

Exemple:
Intégrer **'LN(X)'** sur **[0,1]**.
On effectue **'∫(0,1,LN(X),X)** **→NUM**, en mode **4 FIX**
L'intégrale est généralisée en 0.
La valeur théorique de cette intégrale est -1.
Le résultat obtenu par la HP48SX, en 15 secondes, est -1.0000 et le contenu de **IERR** est 0.0001

LE MENU SOLVE (SAUF SOLVR ET ROOT):

Le menu **SOLVE**, et (surtout) son sous-menu **SOLVR** permettent d'évaluer, de manière conviviale, une ou plusieurs expressions données, et de chercher à les "résoudre".

La HP48SX utilise le nom réservé **'EQ'** pour y placer l'expression en cours, c'est-à-dire celle que vous êtes sur le point de traiter dans le menu **SOLVR**. Chaque répertoire de la HP48SX peut contenir sa propre expression en cours **'EQ'**.

Ici, une "expression" peut être aussi bien:

* Une expression algébrique ne contenant pas l'opérateur =:
  La résoudre signifiera ici résoudre l'équation **'EXPR=0'**.
* Une équation (c'est-à-dire une expression contenant le signe = et se présentant donc sous la forme **'LEFT=RIGHT'**.
  La "résoudre" signifie ici chercher pour quelle valeur d'une variable donnée l'égalité est réalisée.
* Un programme respectant la syntaxe algébrique, c'est-à-dire ne prenant rien sur la pile et y laissant un résultat au niveau 1.
* Le nom d'une variable dont le contenu est une "expression".
* Une liste formée d'objets tels que décrits dans les quatre points précédents.

EXPRESSIONS.

On accède au menu **SOLVE** en actionnant la touche correspondante du clavier (la touche de préfixe étant donc <=┐).

De deux choses l'une:

* Ou bien vous avez déjà défini (dans le répertoire actuel) une expression en cours (dans **'EQ'**) et la HP48SX vous le dans un message en haut de l'écran.

  Exemple: **Current equation:**
  EQ: 'COS(X)+2-X'

* Ou bien vous n'avez encore pas (dans ce répertoire) défini une telle "expression en cours", et la HP48SX vous le rappelle également: **No current equation.**
  Enter eqn, press NEW

  La deuxième partie de ce message est une invitation à créer, d'une certaine manière (il y en a d'autres), l'expression en cours (au moyen de la touche **NEW**).

  Rien ne vous oblige d'ailleurs à suivre ce conseil (faites **ON** pour effacer ce message).

A tout moment on peut réafficher le message indicatif ci-dessus au moyen de la touche **REVIEW** (à condition bien sûr d'être dans le menu **SOLVE**).

*** CREER OU MODIFIER L'EXPRESSION EN COURS:**

Les touches **NEW,EDEQ,STEQ,CAT** permettent toutes de choisir une nouvelle expression en cours, ou d'intervenir sur son contenu. Ces quatre touches figurent également dans le menu **PLOT** (qui lui possède le sous-menu **PLOTR** permettant d'étudier graphiquement l'expression en cours).

Il est bien sûr possible de modifier le contenu de **'EQ'** par les moyens habituels:

* Visite du contenu de **'EQ'** (après avoir placé le nom **'EQ'** au niveau 1 de la pile, actionner la touche **VISIT**, ou les touches ┌=> si vous voulez que l'édition se fasse dans **EquationWriter**).
* Création d'une "expression" au niveau 1, puis utilisation de **'EQ' STO** (ou si le nom **EQ** est visible dans le menu **VAR**, la touche <=┐ suivi de l'appui sur la touche de menu qui désigne **'EQ'**)

NEW

On actionne cette touche après avoir placé une "expression" au niveau 1 (s'il n' y a rien au niveau 1, un bref message : **Enter eqn, Press NEW** vous le signale).

La HP48SX passe alors en mode alpha, cependant qu'un message vous invite à entrer le nom de la variable devant contenir cette expression: **Name the equation,**
press ENTER.

Remarque:
Si l'"expression" que vous avez placée au niveau 1 de la pile n'est pas réellement une expression algébrique (objet entre deux '), alors la HP48SX ajoute **.EQ** en ligne de commande, ce qui sera utile (si vous conservez cette "extension", mais vous n'y êtes pas obligé) dans le catalogue des "expressions" (voir **CAT** plus loin).

Vous avez deux possibilités:

1) Taper effectivement le nom de variable puis **ENTER**:
   Supposons par exemple que vous tapiez **EXPR** (puis **ENTER**).
   Alors l'expression est placée dans la variable nommée **'EXPR'** et le nom **'EXPR'** lui-même est placé dans la variable **'EQ'**.
   L'expression qui était au niveau 1 est donc devenue l'expression en cours, tout en étant "protégée" par son placement dans la variable **'EXPR'**.

2) Taper directement **ENTER** ce qui a pour effet de placer dans **'EQ'** le contenu du niveau 1. L'expression en cours est donc créée, mais sera détruite si vous définissez un nouveau contenu pour la variable **'EQ'**.
   Dans ce cas, il est plus simple d'utiliser **STEQ**.

Dans les deux cas, la HP48SX confirme par un message le nouveau contenu de l'expression en cours.

EDEQ

Cette touche du menu **SOLVE** place le contenu de l'expression en cours en ligne de commande pour d'éventuelles modifications.
Comme d'habitude, on valide les modifications par **ENTER**, ou on les annule par **ON**.

Remarque:
Supposons que la variable **'EQ'** contienne le nom **'EXPR'**, la variable **'EXPR'** contenant elle même l'expression algébrique **'COS(X)+2-X'**.
Le fait d'activer **EDEQ** va placer **'COS(X)+2-X'** en ligne de commande (et pas le nom **'EXPR'** qui est le contenu immédiat de **'EQ'**: c'est une différence avec **'EQ' VISIT**).
Les modifications éventuelles apportées à cette expression sont donc répercutées dans le contenu de **'EXPR'**:
La variable **'EQ'** contiendra encore le nom **'EXPR'**.

STEQ

Cette touche place tous simplement le contenu du niveau 1 dans la variable **'EQ'**, créant ainsi l'expression en cours.
Elle est donc équivalente à: **'EQ' STO**.

Remarques:
* Parmi les quatre touches **NEW,EDEQ,STEQ,CAT,** seul **STEQ** est le nom d'une instruction programmable de la HP48SX.
* Si on préfixe la touche de menu **STEQ** d'un appui sur ╔=> alors on place l'"expression en cours" au niveau 1. l'instruction programmable **RCEQ** (Recall Equation) a le même effet.

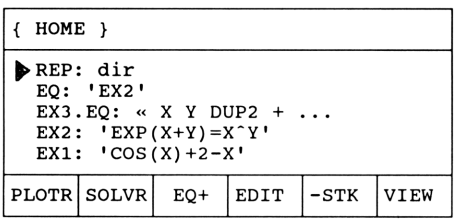
Supposons que la variable **'EQ'** contienne le nom **'EXPR'**, la variable **'EXPR'** contenant elle même l'expression algébrique **'COS(X)+2-X'**.
La séquence ╔=> **STEQ** (l'instruction **RCEQ**) place sur la pile l'expression **'COS(X)+2-X'**, et pas le nom **'EXPR'**: c'est une différence avec: **'EQ' RCL**).

CAT **CATALOGUE DES EXPRESSIONS.**

Le catalogue des expressions est accessible par une touche **CAT** dans le menu **SOLVE** et dans le menu **PLOT.**
Il est également accessible, depuis n'importe quel menu, par la séquence ╔=> **ALGEBRA.**

Le catalogue des expressions donne accès à un menu spécifique, et il met à votre disposition un affichage d'un certain nombre d'objets (un pointeur permet de circuler dans cet affichage, pour désigner tel ou tel de ces objets).

Exemple d'affichage:



Sont affichés, par leur nom suivi d'une partie au moins de leur contenu:
* Les sous-répertoires du répertoire courant.
* Les expressions algé-briques.
* Les variables dont le nom se termine par .EQ

Déplacements verticaux dans le catalogue:

Les touches Δ et ∇ permettent de monter ou de descendre d'une ligne le pointeur du catalogue.
Préfixées par <=╗, elles permettent de le déplacer d'une page vers le haut ou vers le bas.
Préfixées par ╔=>, elles permettent de le déplacer tout au début ou tout à la fin du catalogue.

Changer de répertoire:

On peut, tout en restant dans l'application "catalogue des expressions", changer de répertoire (et donc de catalogue).
La touche **HOME** passe dans le répertoire **HOME.**
La touche **UP** passe dans le répertoire père.
Si le pointeur du catalogue désigne un sous-répertoire, alors: un appui sur **ENTER** passe dans ce sous-répertoire (idem avec un appui sur l'entrée **VIEW** du menu du catalogue).

Le menu du catalogue des expressions:

PLOTR: passe dans le menu **PLOTR**, en faisant de l'expression sélectionnée l'expression en cours.
SOLVR: passe dans le menu **SOLVR**, en faisant de l'expression sélectionnée l'expression en cours.
EQ+: permet de constituer une liste avec les éléments pointés, quand on souhaite étudier ensemble plusieurs expressions.
La séquence <=╗ **EQ+** retire le dernier élément de la liste.
La sortie par **PLOTR** ou **SOLVR** fait de cette liste l'expression en cours.
EDIT: place l'expression pointée en ligne de commande, pour d'éventuelles modifications (confirmer par **ENTER**, annuler par **ON**).
Les modifications sont répercutées dans le contenu de la variable qui était pointée.

→STK: copie sur la pile l'objet pointé (sous forme d'un objet marqué par le nom de la variable).
VIEW: visualise l'objet pointé (sans possiblité de modifier). Cette visualisation cesse quand on relache la touche **VIEW**.
ORDER: fait de l'entrée sélectionnée la première entrée du menu **VAR** actuel.
PURG: détruit l'objet pointé du menu **VAR** et donc supprime son l'affichage.
FAST: cette commande fonctionne en bascule.
Si **FAST** est suivi d'un petit carré ▪, seuls les noms des variables sont affichés, ce qui accélère l'affichage.

Sortir du "catalogue des expressions":

Un appui sur **ON** permet de sortir du catalogue (et le contenu de **'EQ'** n'a alors subi aucune modification).
Un appui sur **SOLVR** ou sur **PLOTR** permet de sortir du catalogue et d'accéder au menu ainsi désigné, tout en faisant de l'entrée sélectionnée (ou de la liste constituée par **EQ+**) l'expression en cours.
On remarquera que dans le catalogue des expressions, ce n'est pas parce qu'un objet est pointé (ou une liste est formée) qu'il (ou elle) devient pour autant l'expression en cours.

LE MENU SOLVR:

Le passage au menu **SOLVR** peut se faire:

* Par l'entrée correspondante du menu **SOLVE**.

* En tapant directement ╔═> suivi de la touche **SOLVE**, et ce depuis n'importe quel menu.

* Par l'entrée correspondante du "catalogue des expressions" (l'objet pointé ou la liste formée devenant l'expression en cours).

Dans les deux premiers cas ci-dessus l'accès à **SOLVR** est refusé si l'expression en cours n'a pas été choisie (c'est-à-dire si la variable **'EQ'** est absente du répertoire).
On se heurte d'ailleurs au message: **Error: No Current Equation**

L'accès au menu **SOLVR** produit un écran particulier, où la ligne de menu est composée des intitulés:

* De chacun des noms intervenant dans l'expression.

* De l'intitulé **EXPR=**.

* De l'intitulé **NXEQ** (Next Equation) si l'expression en cours est une liste.

La ligne supérieure de l'écran indique l'expression qui est en cours de traitement (dans le cas où une liste a été constituée, il s'agit au départ de la première expression de cette liste, et des appuis sur **NEXT** permettent de passer d'une expression de la liste à une autre)

EXPRESSIONS.

*** UTILISATION DES TOUCHES DE VARIABLES:**

Les intitulés des variables qui interviennent dans l'expression en cours d'étude apparaissent en noir sur fond blanc, ce qui est une façon de rappeler que les touches correspondantes ont un fonctionnement particulier.

Plus précisément, supposons que le nom X figure dans l'expression cours:

* L'appui sur la "touche X" place l'objet qui est au niveau 1 dans la variable 'X'.
  Un message en haut de l'écran confirme cette opération.

* La séquence ╔═> "Touche X" place le contenu de la variable 'X' au niveau 1 de la pile.

* La séquence <═╗ "Touche X" résoud l'expression par rapport à la variable 'X'.
  Le message **Solving For X** est affiché pendant que dure cette résolution.

  Par "résoudre l'expression", on entend ici trouver la valeur de X qui:
  * annule l'expression algébrique en cours d'étude (s'il s'agit d'un programme obéissant à la syntaxe algébrique, on cherche X de telle manière que l'évaluation de ce programme renvoie 0 sur la pile).
  * égalise les deux membres de l'équation en cours d'étude.

  (Voir plus loin, pour plus de détails).

* On peut toujours rappeler le contenu d'une variable, visiter ce contenu,ou purger cette variable, en faisant précéder l'appui sur la touche correspondant à cette variable par la cote ', et en utilisant les touches **RCL**, **VISIT**, ou **PURGE** (c'est donc la même chose qu'avec les menus habituels).

*** UTILISATION DE LA TOUCHES EXPR:**

Le fait d'appuyer sur la touche **EXPR=** provoque l'évaluation de l'expression en cours d'étude.

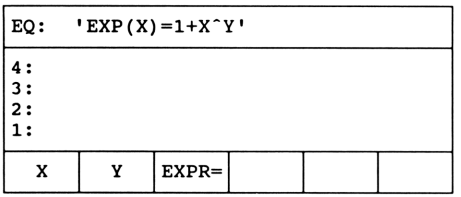
Si celle ci est une expression algébrique (ou un programme agissant de la même manière), le résultat est envoyé au niveau 1, sous forme d'un objet étiqueté (l'étiquette étant EXPR).

Si celle-ci est une équation:
La valeur du membre gauche est envoyée au niveau 2 de la pile (objet étiqueté par **LEFT**).
La valeur du membre droit est envoyée au niveau 1 de la pile (objet étiqueté par **RIGHT**).

Cette touche est utile pour vérifier si l'extracteur de solutions de la HP48SX a bien fonctionné.

Elle l'est également si on souhaite effectuer des évaluations répétées de l'expression en cours, pour différentes valeurs des variables qui y figurent.

Exemple:
Supposons que **'EQ'** contienne l'équation: **'EXP(X)=1+X^Y'**,
X et Y étant pour l'instant des variables formelles.



Le fait d'actionner la touche **EXPR=** va placer:
LEFT: 'EXP(X)' au niveau 2.
RIGHT: '1+X^Y' au niveau 1.

Le fait de taper (par ex.) 1.57 suivi de la touche X place cette valeur dans la variable 'X'.

L'appui sur **EXPR=** place alors:
LEFT: 4.80664819378 au niveau 2.
RIGHT: '1+1.57^Y' au niveau 1.

* **RESOLUTION D'EQUATIONS DANS LE MENU SOLVR:**

Le fait d'appuyer sur la touche <=┐ puis sur une touche de menu menu correspondant à une variable lance l'algorithme de résolution de l'expression par rapport à cette variable.

Soit vous avez préalablement donné à cette variable une valeur particulière. L'algorithme utilise alors cette valeur comme point de départ de son processus de recherche (s'il y a plusieurs solutions, elles pourront ainsi être obtenues successivement en donnant à la variable des valeurs de départ "proches" de ces solutions).

Soit la variable par rapport à laquelle on résoud l'expression n'a pas de valeur au départ (variable formelle). Alors la HP48SX lui donne la valeur initiale 0.

Pendant la recherche d'une solution, la HP48SX affiche le message **"Solving For"** (suivi du nom de l'inconnue) sur la ligne haute de l'écran.

Attention: Il est nécessaire que toutes les autres variables qui figurent dans l'expression soient définies, sans quoi vous vous heurtez à une erreur **"Undefined Name"**.

Pendant la recherche de la solution, vous pouvez appuyer sur une touche quelconque sauf **ON** (sinon vous interrompez les itérations) et vous voyez s'afficher l'intervalle dans lequel la HP48SX est en train de chercher une solution.
Vous pouvez ainsi visualiser la progression de l'algorithme.

La recherche d'une racine de l'expression peut être un succès ou un échec (notamment s'il n'y pas de solution!).
Cette recherche s'effectue avec une précision de 12 chiffres.

Si l'algorithme trouve une solution, il le manifeste par un des messages suivants:
"Zero": l'algorithme a trouvé une valeur de l'inconnue qui résoud exactement l'expression.
"Sign Reversal": l'algorithme vient de trouver une valeur de l'inconnue qui est la plus proche possible de la solution réelle (modifier cette valeur au niveau du 12ème chiffre significatif change le signe de l'expression).

La solution obtenue est alors placée au niveau 1, sous forme d'un objet étiqueté (l'étiquette étant le nom de la variable par rapport à laquelle on a résolu l'équation)

Reprenons l'exemple précédent (Equation **'EXP(X)=1+X^Y'**):

On place 3 dans la variable **'Y'** (on cherche donc à résoudre l'équation **'EXP(X)=1+X^3'**).

a) On place -1 dans X et on lance la recherche en 'X':
On trouve **X: -.825155469689** ("Sign Reversal", en 2 s).

b) On place 0 dans X et on lance la recherche en 'X':
On trouve immédiatement **X: 0** ("Zero") qui est solution évidente du problème.

c) On place 1 dans X et on lance la recherche en 'X':
On trouve **X: 1.54500727907** ("Sign Reversal", en 3.5 s).

d) On place 4 dans X et on lance la recherche en 'X':
On trouve **X: 4.56703683694** ("Sign Reversal", en 4s).

On a ainsi isolé 4 solutions du problème (ce sont les seules).

L'algorithme peut échouer dans sa recherche d'une solution. Il le signalera par un message du type:

"Bad Guesse(s)": l'algorithme se heurte à des erreurs lors de l'évaluation de la procédure (essayez par ex. de résoudre l'équation **'EXP(X)=1+X^Y'**, par rapport à Y, en donnant une valeur négative à X).

"Constant?": Partout où l'algorithme a évalué l'expression, il a trouvé la même valeur (il a donc abandonné...).
Essayez par exemple **'FLOOR(SIN(X)=CEIL(SIN(X)'**.

"Extremum": L'algorithme n'a pas trouvé de solution mais a trouvé un extremum de la valeur absolue de l'expression.
Si vous essayez de résoudre **'EXP(X)=2*X-1'**,(qui n'a pas de solution), vous obtiendrez le message "Extremum" avec le résultat **X: .693147174306** (ou une autre approchante).
Effectivement, l'expression **'EXP(X)-2*X+1'** est toujours >0 et son minimum est atteint pour la valeur qui annule sa dérivée **'EXP(X)-2'** c'est-à-dire **LN(2)≈.69314718056.** (on remarque qu'ici la valeur est obtenue avec une précision très moyenne).

Affoler la HP48SX ?:

Il est possible de donner des équations bizarres à résoudre:

Considérons par exemple l'équation **'FLOOR(X)=CEIL(X)'**, dont les solutions sont évidemment tous les nombres entiers.
Si on lance la recherche en ayant donné à X une valeur entière, on obtient immédiatement cette valeur comme "Zero" (c'est normal).

Mais si on donne par exemple 0.5 comme point de départ, on obtient la solution **X: 108296286805** (qui est effectivement correcte, mais pourquoi celle-là?).

Encore plus pervers: L'équation **'FLOOR(SIN(X))=CEIL(SIN(X))'** qui admet pourtant les solutions X=k*π/2 (k entier) n'est jamais résolue...(on obtient **Constant?** au bout d'un laps de temps important).

Utilisation de listes pour démarrer la recherche:

Le contenu initial donné à la variable par rapport à laquelle on résoud l'équation en cours peut être une liste de deux ou ou trois valeurs:

* Une liste { a b } de deux réels a et b permet d'indiquer à la HP48SX un segment [a,b] où se trouve la solution cherchée.
* Une liste { x a b } permet également d'orienter les recherches dans le segment [a,b], en précisant de plus une estimation x de la variable inconnue.

L'intérêt qu'il y a à donner ainsi des indications supplémentaires est double:

* Dans le cas où le problème posé admet plusieurs solutions, on est ainsi en mesure de mieux les obtenir isolément.
* Ces précisions supplémentaires ne peuvent qu'accélérer l'obtention de la solution.

On remarque d'ailleurs que lorsqu'on interromp les recherches par un appui sur **ON**, la **HP48SX** place au niveau 1 une liste {x a b}, où x désigne l'approximation trouvée jusqu'ici, et où a et b délimitent le segment dans lequel le "solveur" est en train de progressser.

Il suffit alors de replacer cette liste dans la variable inconnue et de relancer la recherche au point où elle s'était interrompue.

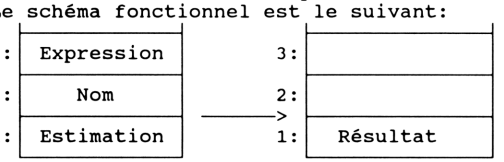
ROOT

L'instruction **ROOT** (qui figure dans le menu **SOLVE**) est celle qu' utilise la HP48SX pour résoudre une expression dans l'environnement interactif créé par le sous-menu **SOLVR**.

Tout ce qui a été dit continue donc de s'appliquer (sauf bien entendu les messages d'accompagnement).

L'intérêt de **ROOT** est de pouvoir être incluse dans un programme.

Le schéma fonctionnel est le suivant:



'Expression' est un objet algébrique (avec ou sans le signe =) ou un programme respectant la syntaxe algébrique.
'Nom' est le nom de la variable inconnue.

(Ce nom doit être un nom global: le résulat est alors obtenu tant au niveau 1 que dans cette variable globale).

'Estimation' est la valeur initiale à donner à l'inconnue comme point de départ des itérations (comme il a été dit ci-dessus, on peut donner un nombre ou une liste).

Remarque:

Si l'algorithme échoue dans sa recherche d'une solution (pour cause de **"Bad Guess(es)"** ou de **"Constant?"**, le message correspondant est affiché et le calculateur s'arrête sur une erreur.

Par contre: si **ROOT** trouve un extremum, et non une solution, aucun message ne permet de le savoir. Le résultat est obtenu au niveau 1 comme s'il s'agissait d'une solution.

A l'utilisateur d'être prudent (par exemple en calculant l'expression pour cette valeur de l'inconnue).

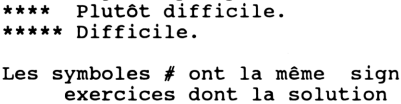
▮ EXERCICES SUR LES EXPRESSIONS:

La difficulté des exercices est mesurée par des symboles * ou # :

- `*` Très facile.
- `**` Assez simple.
- `***` Déjà un peu plus délicat.
- `****` Plutôt difficile.
- `*****` Difficile.

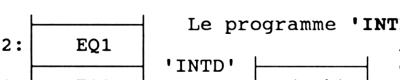
Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes précédant dans cette liste d'exercices.

Exercice 1: (***) Point d'intersection de deux droites.



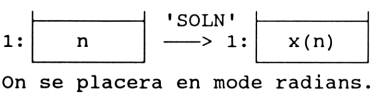
Le programme **'INTD'** donne le point d'intersection A(a,b) des deux droites D1 et D2 dont les équations **EQ1** et **EQ2** sont placées aux niveaux 1 et 2. **EQ1** et **EQ2** sont des expressions du type **'αX+ßY+σ=0'**, ou **'αX+ßY=σ'**, ou **'ßY+αX=σ'** ou .... (bref toute forme usuelle d'une équation de droite).
On se limitera au cas où Y est réellement présent dans les équations, et où les deux droites ne sont pas parallèles.

Exercice 2: (***) Dérivée n-ème de **'X^(n-1)*LN(X)'**.



'DERN' calcule la dérivée n-ème de la fonction **'X^(n-1)*LN(X)'** par rapport à la variable for--melle 'X'.Le résultat est une expression (simple) placée au niveau 1.

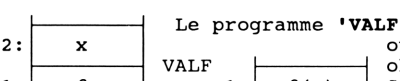
Exercice 3: (***) Calculer une solution particulière de **TAN(X)=X.**



'SOLN' calcule la solution x(n) de l'équation TAN(X)=X, telle que $n\pi-\pi/2 \leq X \leq n\pi+\pi/2$. (n entier positif ou négatif).

On se placera en mode radians.

Exercice 4: (*) Evaluer un programme ou une expression.



Le programme **'VALF'** évalue au point x l'expression ou le programme f, de manière à obtenir un résultat noté ici f(x). Si f est un programme, il est tout simplement évalué.
Si f est une expression, la valeur x est placée dans la variable 'X' puis l'expression f est évaluée (f est donc en principe une expression par rapport à la variable X)

Exemple: avec x=5 au niveau 2, le résultat doit être EXP(5) aussi bien si f est le programme « **EXP** » que l'expression **'EXP(X)'**

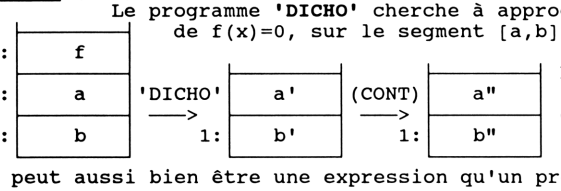
Note sur les exercices 5 à 11: La HP48SX possède ses propres algorithmes de recherche de solutions à une équation (**ROOT**) ou d'intégrales (∫).

Néanmoins, il n'est pas possible de régler la précision de **ROOT** (ce qui est tout de même un peu gênant: on peut parfois se contenter d'un précision moindre si les calculs s'en trouvent accélérés) et on ne peut pas non plus facilement connaître les résultats intermédiaires de la recherche.

D'autre part, la programmation des méthodes usuelles de résolution de f(x)=0 ou d'intégration est une démarche utile.

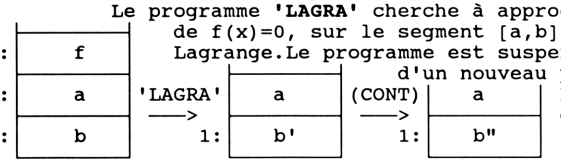
Pour ces raisons, ces méthodes font l'objet des exercices 5 à 11.

Exercice 5: (***###***) Résolution de f(X)=0 par dichotomie.



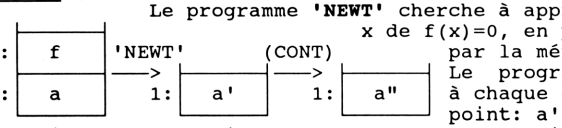
Le programme **'DICHO'** cherche à approcher la solution x de f(x)=0, sur le segment [a,b], par la méthode de la dichotomie. Le programme est suspendu à chaque calcul d'un nouveau segment [a',b'], [a",b"], etc...
f peut aussi bien être une expression qu'un programme (cf ex4).

Exercice 6: (***###***) Résolution de f(X)=0 par la méthode de Lagrange. (ou méthode des parties proportionnelles)



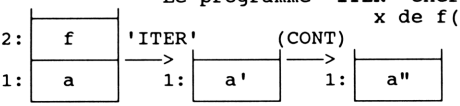
Le programme **'LAGRA'** cherche à approcher la solution x de f(x)=0, sur le segment [a,b], par la méthode de Lagrange.Le programme est suspendu à chaque calcul d'un nouveau point b',b", etc... La suite b,b',b",.. doit converger vers la solution x.
Dans ce schéma fonctionnel, a est la valeur fixe de la méthode, et b', b" etc...sont les valeurs successives calculées.
Le fait que "a" aparaisse à chaque **HALT** permet de le modifier avant de faire **CONT**.
f peut aussi bien être une expression qu'un programme (cf ex).

Exercice 7: (***) Résolution de f(X)=0 par la méthode de Newton.



Le programme **'NEWT'** cherche à approcher la solution x de f(x)=0, en partant du point a, par la méthode de Newton. Le programme est suspendu à chaque calcul d'un nouveau point: a',a".
La suite a,a',a" doit converger vers la solution. Le fait que le programme soit suspendu permet de relancer la recherche à partir d'un nouveau point a.
f doit être une expression algébrique par rapport à X (car f doit être dérivée). f ne peut donc ici pas être un programme.

Exercice 8: (##) Résolution de f(X)=X par itérations.



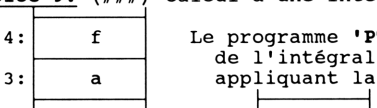
Le programme **'ITER'** cherche à approcher la solution x de f(x)=x, en partant du point a, par itérations. Le programme est suspendu à chaque calcul d'un nouveau point: a'=f(a), a"=f(a'),...

La suite a,a',a" peut converger vers la solution. Le fait que le programme soit suspendu permet de relancer la recherche à partir d'un nouveau point a.

f peut aussi bien être une expression qu'un programme (cf ex4).

La convergence est probable si a est "suffisamment" proche de la solution x et si la dérivée de f, en x, est inférieure à 1 en valeur absolue (convergence d'autant plus rapide que f'(x) est proche de zéro).

Exercice 9: (###) Calcul d'une intégrale (méthode du point-milieu)



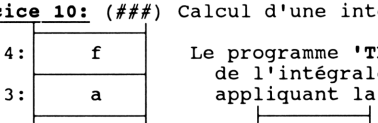
Le programme **'PTMI'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode du point milieu.

Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

$$\int_{\alpha}^{ß} f(x)dx = (ß-\alpha)f((\alpha+ß)/2).$$

f peut aussi bien être une expression qu'un programme (cf ex4).

Exercice 10: (###) Calcul d'une intégrale (méthode des trapèzes)



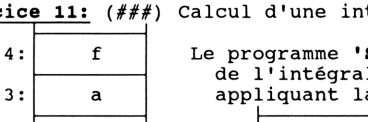
Le programme **'TRPZ'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode des trapèzes.

Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

$$\int_{\alpha}^{ß} f(x)dx = (ß-\alpha)(f(\alpha)+f(ß))/2.$$

f peut aussi bien être une expression qu'un programme (cf ex4).

Exercice 11: (###) Calcul d'une intégrale (méthode de Simpson)



Le programme **'SMPS'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode de Simpson.

Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

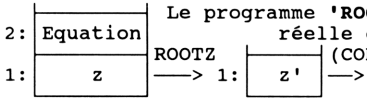
$$\int_{\alpha}^{ß} f(x)dx = (ß-\alpha)\left[\frac{f(\alpha)+4f((\alpha+ß)/2)+f(ß)}{6}\right]$$

f peut aussi bien être une expression qu'un programme (cf ex5).

On donnera deux solutions, l'une directe (comme pour **'PTMI'** et **'TRPZ'**), et l'autre utilisant **'PTMI'** et **'TRPZ'**.

Exercice 12: (*****) Chercher une solution complexe d'une équation.

L'instruction **ROOT** de la HP48SX ne permet que d'obtenir des solutions réelles.



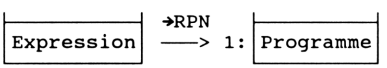
Le programme **'ROOTZ'** permet de chercher une solution réelle ou complexe d'une équation exprimée par rapport à la variable 'Z'. La recherche se fait par itérations à partir d'un point initial z placé au niveau 1.

L'objet placé au niveau 2 peut être une équation ou une expression algébrique (dans tous les cas par rapport à la variable Z).

On utilisera pour la suite z,z',z" (suite de nombres complexes censée converger vers une solution) une méthode inspirée de l'algorithme de Newton.

Pour plus de précisions voir le programme **'SXY'** dans le chapitre **"Analyse"** du volume **'Programmation et applications'**.

Exercice 13: (*****) Transformer une expression en programme **RPN**.



'→RPN' transforme l'expression algébrique (ou équation) placée au niveau 1 en un programme en notation polonaise inverse, équivalent à cette expression.

Pour prendre un exemple:

'EXP(5^SQ(Z))+X/(Y-1)^A'

devient: **« 5 Z SQ ^ EXP X Y 1 - A ^ / + »**.

GRAPHISME

Dans ce chapitre, nous allons voir les différentes instructions se rapportant à l'écran-graphique de la HP48SX.

LES ROLES DE L'ECRAN:

Un point de l'écran est habituellement appelé un **pixel**.

On dira qu'un des pixels de l'écran qu'il est allumé ou éteint.

L'écran de la HP48SX est une matrice rectangulaire haute de 64 pixels, et large de 131 pixels.

Il permet d'afficher tantôt le contenu de la pile (ou tout du moins d'une partie de celle-ci), tantôt des images graphiques.

L'écran de la HP48SX peut afficher:

1) Le contenu de la pile (ou du moins d'une partie de celle-ci).

   C'est son utilisation la plus courante.
   Il est facile de constater que dans ce cas l'affichage est tout de même graphique, en ce sens que les caractères qui y apparaissent sont affichés par pixels.

   Une première remarque (mais nous y reviendrons):

   Les caractères qui apparaissent dans la ligne d'état (notamment ceux qui indiquent le chemin en cours), et ceux qui apparaissent dans la ligne de menu, sont de même (petite) taille.

   Les caractères qui apparaissent dans l'image donnée par l'écran des niveaux 1 à 4 (au plus) de la pile, sont d'une taille supérieure.

   On verra plus loin que les caractères peuvent encore être affichés suivant une nouvelle taille (intermédiaire entre les deux précédentes): c'est la taille qu'emploie en particulier l'instruction **DISP**.

2) Le contenu d'un objet (le plus souvent une chaîne de caractères formant un message), avec l'instruction **DISP**:

   L'instruction **DISP** permet d'afficher un "message" à partir de l'extrémité gauche de l'écran, sur une ou plusieurs lignes.

Compte tenu du format de caractère utilisé par **DISP** (format intermédiaire, comme il a été dit plus haut), on peut ainsi accéder à 7 lignes distinctes (depuis la ligne 1, en haut de l'écran, jusqu'à la ligne 7 située juste au-dessus de la ligne des menus.

3) Le contenu de **PICT** (du moins en partie).

On va bientôt voir l'importance du nom PICT pour le graphisme de la HP48SX.
Disons simplement que **PICT** est un nom réservé par la HP48SX pour désigner une partie de la mémoire où l'utilisateur peut intervenir pour constituer une image graphique.
Cette image graphique peut être modifiée à l'aide d'un très grand nombre d'instructions de la HP48SX (citons en particulier l'instruction **LINE** permettant d'y tracer un segment de droite) mais ces modifications ne seront visibles à l'écran (pendant leur éxécution ou après coup) que si l'utilisateur décide d'afficher à l'écran le contenu de **PICT**.
La touche **GRAPH** permet en particulier d'accéder à un environnement très convivial, où le contenu de **PICT** est affiché, et où l'utilisateur peut facilement intervenir.
Il est également possible d'afficher le contenu de **PICT** sans passer par l'environnement **GRAPHICS** (utile notamment dans les programmes: l'instruction **PVIEW** est alors toute indiquée).

4) Le contenu d'un objet graphique quelconque:

Les objets-graphiques forment une des catégories d'objets reconnues par la HP48SX.
Le contenu de **PICT** est un objet graphique qui ne diffère des autres que par le grand nombre d'instructions qui lui sont spécifiques, et par l'environnement **GRAPHICS**, qui permet de l'observer et de le modifier à loisir.
Néanmoins tout objet-graphique peut être affiché à l'écran (instruction **→LCD**).
La HP48SX possède un arsenal d'instructions permettant de produire des objets-graphiques:
→GRO, →LCD, BLAN, GOR, GXOR, REPL, SUB, ...

Remarque importante:

L'environnemnt **GRAPHICS** ou le menu **PLOTR** (dans le menu **PLOT**) permettent d'accéder au contenu de **PICT**, et d'y demeurer jusqu'à nouvel ordre.
Néanmoins, notamment dans les programmes, les instructions d'affichage d'objets-graphiques produisent des images "volatiles", ou encore non permanentes. L'écran reprend en effet son rôle habituel d'affichage de la pile dans les cas suivants (ce qui a pour effet de faire disparaître l'image graphique):
* Le programme en cours vient de se terminer normalement.
* Le programme en cours est suspendu par un **HALT**.
* Le programme en cours s'est interrompu sur une erreur.

Dans les deux premiers cas ci-dessus, il est possible de prolonger la durée de l'affichage graphique, jusqu'à ce qu'une touche soit actionnée: c'est le rôle joué par l'instruction **FREEZE** qui permet comme son nom l'indique de "geler" tout ou partie de l'affichage.

GENERALITES SUR LES OBJETS-GRAPHIQUES:

DIMENSIONS D'UN OBJET-GRAPHIQUE:

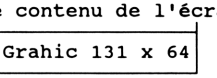
Un objet-graphique est toujours rectangulaire (il sert de cadre à une image formée de pixels allumés ou non, cette image ayant, elle, une forme quelconque).
Ses dimensions sont donc:
- sa largeur L (dimension horizontale), exprimée en pixels.
- sa hauteur H (dimension verticale), exprimée en pixels.

Les dimensions d'un objet-graphique peuvent très bien excéder les dimensions de l'écran.
Il est bon de savoir que l'objet graphique contenu dans **PICT** est toujours de dimensions au moins égales à celle de l'écran, sauf si l'on décide de "purger" **PICT** (auquel cas **PICT** reste connu de la HP48SX, mais avec des dimensions nulles). De même une réinitialisation de la HP48SX "annule" **PICT**.
Attention:
Les objets-graphiques sont gourmands en mémoire.
Un objet-graphique de taille LxH occupe environ L*H/8 octets.
Par exemple l'image graphique de l'écran, obtenue par **LCD→**, occupe 1098 octets en mémoire.

OBJETS-GRAPHIQUES SUR LA PILE:

Lorsqu'un objet-graphique est placé sur la pile (placer un objet graphique sur la pile et afficher un objet graphique à l'écran sont deux choses totalement différentes), alors il est signalé de la manière suivante: **Graphic L x H**

Exemple: L'instruction LCD→ (qui transforme le contenu de l'écran en un objet-graphique) produit,
au niveau 1 de la pile:



Une première remarque s'impose: quand un objet-graphique est sur la pile, son contenu n'est pas visible. Seules sont indiquées ses dimensions.

CONTENU D'UN OBJET-GRAPHIQUE:

Pour visualiser le contenu de cet objet-graphique (à ne pas confondre avec l'image qui est codée dans ce contenu), il faut utiliser **EDIT** (ou **VISIT**). Le contenu de l'objet-graphique est alors visible en ligne de commande.

Le contenu se présente alors sous la forme: **GROB L H .....**
Ici **GROB** signifie **"Graphic Object"**, L et H sont encore la largeur et la hauteur, et le contenu effectif de l'objet est ici représenté par .....
Le contenu est en fait une succession de caractères "hexa" (0 1 2 ... 9 A B C D E F), chaque couple de caractères représentant un entier compris entre 0 et 255.
Un tel entier permet de coder un segment de 8 pixels successifs sur une même ligne.
On comprend qu'il ne soit pas commode de modifier un objet graphique en ligne de commande ! On n'aura donc pas dans la pratique à visiter le contenu effectif d'un objet-graphique.

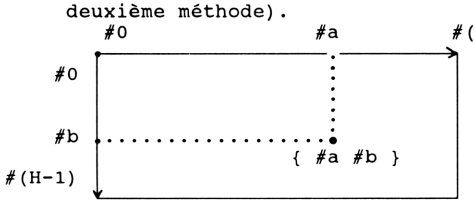
Exemple:
Placez-vous en mode "grades":
L'indicateur **GRAD** doit apparaître en ligne d'état.
Dans le menu **PRG DSPL** (page 4), actionner l'instruction **LCD→**
Vous obtenez **Graphic 131 x 64** au niveau 1.
Appuyez sur la touche ▽ , pour passer le contenu de l'objet graphique en ligne de commande.
Vous devez obtenir:
GROB 131 64 8B191000....
Ici #8h = #1000b, #Bh = #1011b.
Il faut placer #Bh avant #8h et on obtient: #10111000b.
Si on lit ces 8 bits à l'envers (!), on trouve #00011101b.
Ces 8 bits représentent l'état des 8 premiers pixels en haut à gauche de l'écran (segment horizontal)

REPERER UN POINT DANS UN OBJET GRAPHIQUE:

Il est important d'identifier la position d'un point précis dans un objet graphique.
Cette position est déterminée (normal) par deux coordonnées, mais il y a pour celà deux méthodes.
Ces deux méthodes ont un point commun important:
La coordonnée horizontale précède toujours la coordonnée verticale.

a) Par une liste de deux entiers binaires:

C'est la façon la plus naturelle, et elle a l'avantage d'être immuable (de ne pas dépendre de tels ou tels réglages de l'utilisateur, comme on va s'en rendre compte avec la deuxième méthode).



Si l'objet graphique a:
une largeur de L pixels,
une hauteur de H pixels,
L'axe horizontal est gradué de #0 à #(L-1).
L'axe vertical est gradué de #0 à #(H-1).

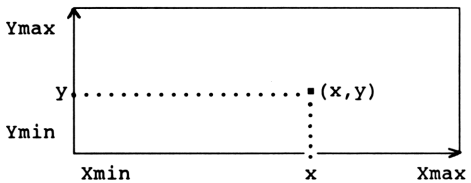
b) Par un nombre complexe:

Cette méthode suppose que l'on considère l'objet-graphique comme une fenêtre sur un plan (P) rapporté à un repère Oxy.
Un pixel de l'objet-graphique (ou plutôt le milieu de ce pixel) désigne alors un point M de ce plan.
Ce point M possède une abscisse x et une ordonnée y.
Le pixel est donc repéré par le nombre complexe (x,y).

Dans la pratique (mais çà n'est pas obligatoire), on réserve l'utilisation de cette notation à l'objet graphique contenu dans **PICT**, et ce pour plusieurs raisons.
* L'utilisation de nombres complexes comme coordonnées correspond à une application aux mathématiques des objets-graphiques, et le menu **PLOTR** (qui utilise le contenu de **PICT**) est tout indiqué pour les tracés mathématiques.
* Les dimensions de **PICT** sont utilisées par la HP48SX pour traduire des coordonnnées d'un système à l'autre (instructions **PX→C** et **C→PX**).

Les instructions **PMIN** et **PMAX** (décrites plus loin) permettent de définir d'une part:

* Les coordonnées (Xmin,Ymin) du coin inférieur gauche de l'objet-graphique contenu dans **PICT**.
* Les coordonnées (Xmax,Ymax) du coin supérieur droit de l'objet-graphique contenu dans **PICT**.



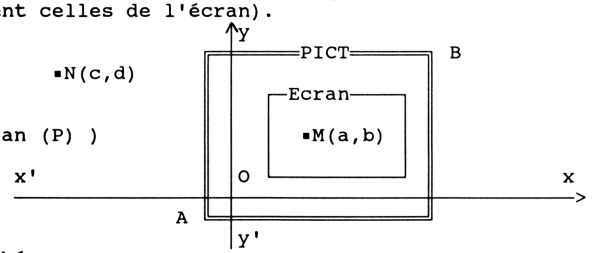
On remarque que l'axe vertical est ici dirigé vers le haut, ce qui n'est pas le cas quand les coordonnées sont exprimées en entiers binaires.

"PICT" A L'ECRAN:

Pour en terminer avec ces généralités, il est un point qu'il faut avoir bien compris.
Quand on effectue des tracés mathématiques (notamment dans le sous-menu **PLOTR** du menu **PLOT**), et que ces tracés apparaissent à l'écran, il faut distiguer trois choses distinctes:

* L'écran, qui est une fenêtre sur le contenu de **PICT**:
  On a vu que les dimensions de **PICT**,par défaut, sont celles de l'écran. Dans un tel cas, l'écran permet donc de voir l'intégralité du contenu de **PICT**.
  Dans le cas où les dimensions de **PICT** excèdent celles de l'écran, celui-ci n'est donc qu'une fenêtre permettant de voir une partie du contenu de **PICT**.Il est alors bien sûr possible de déplacer cette fenêtre (nous verrons comment) de manière à accéder à l'intégralité de **PICT**.
* Le contenu de **PICT**, qui est une fenêtre sur un plan infini rapporté à un repère Oxy.
* Le plan mathématique (P) contenant les représentations graphiques en cours d'étude.

Nous pouvons schématiquement représenter cette situation de la manière ci-dessous (où on voit que les dimensions de **PICT** excèdent celles de l'écran).



Sur se schéma:
A est le point (Xmin,Ymin). B est le point (Xmax,Ymax).
Le point M(a,b) est visible à l'écran.
Le point origine sera visible à l'écran si on déplace la fenêtre que l'écran ouvre sur le contenu de **PICT**.
Le point N(c,d) ne pourra être visualisé à l'écran, car il n'est pas dans **PICT** (solution: modifier les positions de A et/ou B, à l'aide par exemple de **PMIN** et/ou **PMAX**)

LE MENU PRG DSPL:

Le menu **PGR DSPL** (DSPL pour: Display) contient 24 instructions se rapportant directement aux objets-graphiques, et parfois exclusivement à l'objet-graphique contenu dans **PICT**.

N.B: Dans certains des diagrammes fonctionnels ci-dessous, on utilisera **GROB, GROB1, GROB2,** etc... pour désigner des objets-graphiques.

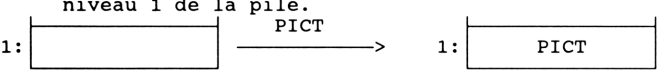
N.B: dans certains des diagrammes fonctionnels ci-dessous, le mot 'position' désigne le couple de coordonnées qui permet d'identifier un point particulier d'un objet-graphique.
'position' pourra donc être:
- Une liste { #a #b } si on utilise les coordonnées-pixels.
- Un complexe (a,b) si on utilise les coordonnées-utilisateur (ce qui en principe est plutôt réservé à l'objet graphique contenu dans **PICT**).

Si une instruction nécessite les coordonnées de 2 points, alors ces coordonnées doivent être toutes données:
- en utilisant les coordonnées-pixels.
ou bien:
- en utilisant les coordonnées-utilisateurs.

On ne peut pas, dans une même instruction, "panacher" ces deux types de coordonnées.

PICT

Cette instruction a pour effet de placer le nom **PICT** au niveau 1 de la pile.



Le contenu de **PICT** peut alors être placé sur la pile à l'aide de l'instruction **RCL**.

Certaines instructions comme **GOR,GXOR,REPL,SUB** peuvent utiliser le nom **PICT** sur la pile:

Elles utilisent alors le contenu de **PICT**, et elles envoient l'"objet-graphique résultat" directement dans **PICT**.

Remarques:

PICT PURGE élimine le contenu de **PICT** de la mémoire: l'entrée dans l'environnement **GRAPH** recrée alors un **PICT** "blanc" de dimensions égales à celles de l'écran.

PICT STO place l'objet graphique du niveau 1 pour en faire le contenu de **PICT**: si cet objet est de dimensions inférieures à celles de l'écran, le contenu de **PICT** est complété "en blanc" pour atteindre exactement ces dimensions (l'objet graphique initial étant alors placé dans le coin supérieur du contenu de **PICT**).

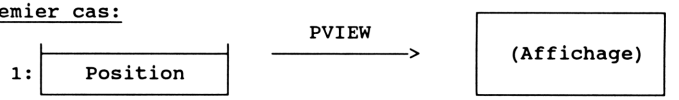
PVIEW

Cette instruction permet d'afficher le contenu de **PICT** à l'écran.

Tant qu'il y reste visible, les modifications qu'on peut apporter au contenu de **PICT** sont donc répercutées immédiatement à l'écran (encore que celà dépende des dimensions du contenu de **PICT**: si celui-ci est de dimensions supérieures à celles de l'écran, on ne peut pas tout voir à la fois).

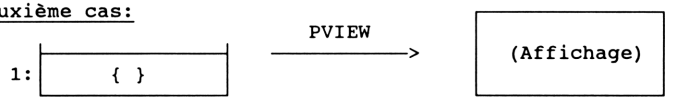
Il y a deux cas possibles:

Premier cas:



Le contenu de **PICT** est affiché en plaçant dans le coin supérieur gauche de l'écran le point désigné par 'Position'.
Si 'Position' = {#0 #0}, on voit ainsi la partie supérieure gauche du contenu de **PICT** (et donc la totalité de celui-ci si ses dimensions sont celles de l'écran).
On visualise ainsi une fenêtre (aux dimensions de l'écran) sur le contenu de **PICT**. Cette fenêtre ne doit pas "déborder" de cet objet-graphique.
L'image ainsi obtenue devra être "gelée" au moyen de l'instruction **FREEZE** si on veut qu'elle reste affichée à l'issue du programme en cours.

Deuxième cas:

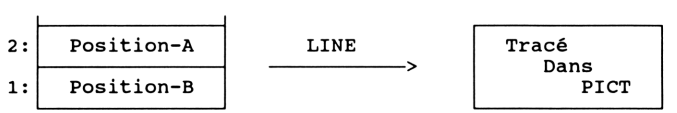


On a donc placé une liste vide au niveau 1.

Dans ce cas le contenu de **PICT** est affiché centré à l'écran et le mode "scrolling" est activé:
Les touches de direction permettent de déplacer la fenêtre que l'écran ouvre sur le contenu de **PICT**.
Préfixées par ╔=>, ces touches de directions permettent un accès immédiat aux "bords" de l'objet-graphique contenu dans **PICT**.
On sort du mode "scrolling" par un appui sur **ON** (dans un programme contenant la séquence **{ } PVIEW**, cet appui sur **ON** permet de sortir du mode "scrolling", mais n'interromp pas le programme).
Le mode "scrolling" n'est utile que si les dimensions du contenu de **PICT** sont supérieures à celles de l'écran.

LINE

L'instruction **LINE** permet de tracer, dans l'objet graphique contenu dans **PICT**, le segment de droite joignant les points A et B désignés par leurs positions 'Position-A' et 'Position-B'.



Le tracé s'effectue en mode **OR** (tous les points du segment sont donc allumés).
Le segment peut très bien "déborder" du contenu de **PICT**: seuls les points appartenant à celui-ci sont tracés (il n'y a pas d'erreur).
Le tracé n'est visible à l'écran que si le contenu de **PICT** est affiché (et encore celà dépend-il des dimensions de celui-ci).

TLINE

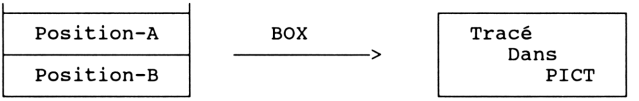
Cette instruction est analogue à **LINE**, la seule différence étant que le tracé de segment se fait en mode **XOR:**
Sur le chemin de ce segment:
Les points qui étaient "éteints" sont "allumés".
Les points qui étaient "allumés" sont "éteints".
Par exemple, **TLINE** permet d'effacer un segment de droite qui vient d'être tracé par **LINE.**

Essayez par exemple le programme suivant (où on suppose que les dimensions de **PICT** sont celles de l'écran).

```
« ERASE
  { #0d  #0d }   PVIEW
  DO
      { #0d  #0d }   { #130d  #63d }
      DUP2   LINE   TLINE
  UNTIL  0   END
»
```

BOX

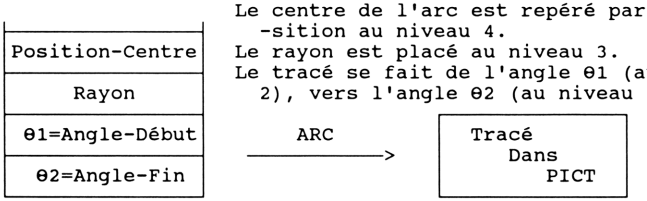
L'instruction **BOX** trace, dans l'objet graphique contenu dans **PICT**, un rectangle dont les points A et B, repérés par leurs coordonnées "Position-A" et "Position-B" sont 2 sommets opposés.



Si ce rectangle déborde du contenu de **PICT**, seule la partie qui y appartient est tracée (il n'y a pas d'erreur).
Le tracé n'est visible à l'écran que si le contenu de **PICT** est affiché (et encore celà dépend-il des dimensions de celui-ci).

ARC

L'instruction **ARC** permet de tracer un arc de cercle dans le contenu de **PICT.**
Le schéma fonctionnel est le suivant:



Le centre de l'arc est repéré par sa po-
-sition au niveau 4.
Le rayon est placé au niveau 3.
Le tracé se fait de l'angle θ1 (au niveau 2), vers l'angle θ2 (au niveau 1).

Le tracé se fait dans le sens trigonométrique (c-a-d dans le sens inverse de celui des aiguilles d'une montre).
Les angles θ1 et θ2 sont des réels exprimés dans le mode courant (degrés, radians, grades). Ce sont des angles polaires par rapport à un axe horizontal, orienté de gauche à droite.
Le tracé n'est visible à l'écran que si le contenu de **PICT** est affiché (et encore celà dépend-il des dimensions de celui-ci).

De deux choses-l'une:

Ou bien "Position-centre" est une liste de 2 entiers binaires et "Rayon" est exprimé en pixels (sous la forme également d'un entier binaire).
Ou bien "Position-centre" est un nombre complexe et "Rayon" est exprimée en "unités-utilisateur" (sous la forme d'un nombre réel positif)

PIXON

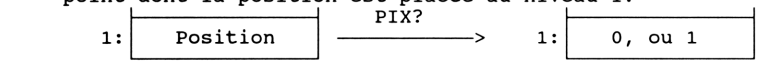
Cette instruction allume, dans le contenu de **PICT**, le point dont la position est placée au niveau 1.
Si ce point est "à l'extérieur" du contenu de **PICT**, il n'y pas d'erreur.

PIXOFF

Cette instruction éteint,dans le contenu de **PICT**, le point dont la position est placée au niveau 1.
Si ce point est "à l'extérieur" du contenu de **PICT**, il n'y pas d'erreur.

PIX?

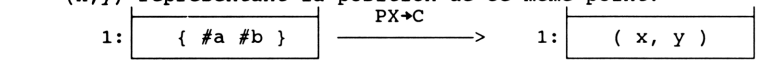
Cette instruction teste, dans le contenu de **PICT**, l'état du point dont la position est placée au niveau 1.



Le résultat est 1 si le point est allumé, 0 sinon.

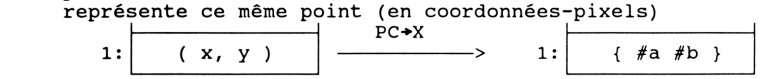
PX→C

L'instruction **PX→C** (Pixel vers Complexe) transforme la liste {#a #b} des coordonnnées-pixels d'un point en le nombre complexe (x,y) représentant la position de ce même point.



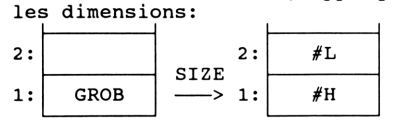
PC→X

L'instruction **PC→X** (Complexe vers Pixel) transforme le nombre complexe (x,y) représentant les coordonnées-utilisateur d'un point en la liste { #a #b } de deux entiers binaires qui représente ce même point (en coordonnées-pixels)



SIZE

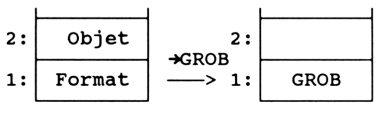
L'instruction **SIZE**, appliquée à un objet-graphique, en donne les dimensions:



Ici, #L et #H sont deux entiers binaires désignant respectivement la largeur et la hauteur (expri--mées en pixels) de l'objet gra--phique "GROB".

→GROB

L'instruction **→GROB** transforme en objet-graphique l'objet qui est placé au niveau 2 de la pile, en utilisant la spécification de format placée au niveau 1.



Les caractères qui figurent dans "Objet" sont ainsi transformés en leur image graphique, de taille:
* petite: si "format"=1
* moyenne: si "format"=2
* grande: si "format"=3

On peut également spécifier le format 0 qui équivaut au format 3, sauf pour les expressions et les objets-unités (pour lesquels le format 0 signifie: image graphique de cet objet dans l'application EquationWriter: cette image étant de taille au moins égale à celle de l'écran privé de la ligne des menus).

D'une façon précise: L'image graphique d'un caractère (lettre, chiffre, etc...) tient dans une matrice de taille :
4x6 pixels si "format"=1
6x8 pixels si "format"=2
6x10 pixels si "format"=3
(D'abord la largeur, ensuite la hauteur)

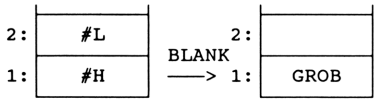
La petite taille est celle que la HP48SX utilise par exemple dans la ligne d'état pour afficher le chemin en cours, ou les indi--cateurs **RAD**, **USR**, etc...

La taille moyenne est celle qu'utilise l'instruction **DISP**.

La grande taille est celle que la HP48SX utilise pour afficher le contenu de la pile, en utilisation normale.

BLANK

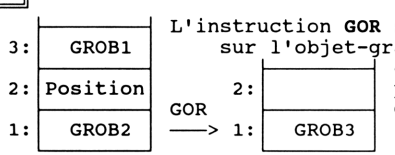
L'instruction **BLANK** crée un objet graphique "blanc" ayant les dimensions spécifiées:
Largeur = #L pixels.
Hauteur = #H pixels.



GOR

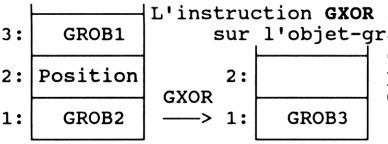
Superposition de deux objets-graphiques, en mode **OR**.

L'instruction **GOR** superpose l'objet-graphique **GROB2** sur l'objet-graphique **GROB1**, en positionnant le coin supérieur gauche de **GROB2** au point dont la position (relative à **GROB1**) est indiquée au niveau 2.



La superposition se fait en mode **OR**, ce qui signifie que sur la partie de **GROB1** où s'effectue la superposition, sont allumés les pixels qui étaient allumés dans l'un ou dans l'autre (ou dans les deux) objets graphiques **GROB1** et **GROB2**.
Voir des remarques importantes après **REPL**.

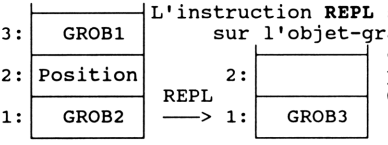
GXOR Superposition de deux objets-graphiques, en mode **XOR**.



L'instruction **GXOR** superpose l'objet-graphique **GROB2** sur l'objet-graphique **GROB1**, en positionnant le coin supérieur gauche de **GROB2** au point dont la position (relative à **GROB1**) est indiquée au niveau 2.

La superposition se fait en mode **XOR**, ce qui signifie que sur la partie de **GROB1** où s'effectue la superposition, sont allumés les pixels qui étaient allumés dans l'un et dans l'un seulement des objets graphiques **GROB1** et **GROB2**.
L'instruction GXOR est utile pour effacer un objet-graphique qui vient d'être superposé à un autre objet-graphique à l'aide de **GOR** ou de **REPL**.
Voir des remarques importantes après **REPL**.

REPL Superposition de 2 objets-graphiques, en mode **écrasement**.



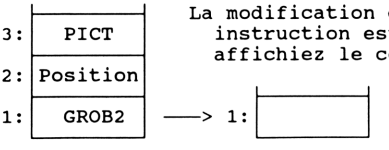
L'instruction **REPL** superpose l'objet-graphique **GROB2** sur l'objet-graphique **GROB1**, en positionnant le coin supérieur gauche de **GROB2** au point dont la position (relative à **GROB1**) est indiquée au niveau 2.

La superposition se fait en mode **écrasement**, ce qui signifie que sur la partie de **GROB1** où s'effectue la superposition, sont allumés les pixels qui étaient allumés dans **GROB2**, et uniquement ceux-là.

Remarques sur les instructions GOR, GXOR, REPL:

L'objet graphique **GROB2** peut "déborder" de l'objet graphique **GROB1** sur lequel il se superpose. celà ne produit pas d'erreur: les points de **GROB2** qui "dépassent" ne sont tout simplement pas pris en considération.
Néanmoins, la "position" placée au niveau 2 doit tomber dans les limites de l'objet-graphique **GROB1** (sans quoi une erreur "Bad Argument Value" se produit)
Sur un fonc blanc (le "fond" est ici la partie de **GROB1** sur laquelle s'effectue la superposition), les instructions **GOR**, **GXOR**, et **REPL** sont équivalentes.
Sur un fond noir, l'instruction **GXOR** provoque une écriture en vidéo inverse de l'objet-graphique **GROB2**.
L'objet graphique **GROB3** a toujours les dimensions de **GROB1**.
On ne peut pas placer le nom **PICT** à la place de **GROB2**.

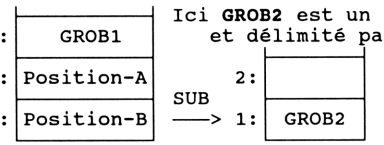
Si on remplace **GROB1** par le nom **PICT**, la superposition se fait directement dans le contenu de **PICT:** le schéma fonctionnel est alors:



La modification de **PICT** qui résulte de cette instruction est visible à condition que vous affichiez le contenu de **PICT** à l'écran.

SUB

L'instruction **SUB** permet d'extraire une fenêtre d'un objet graphique.

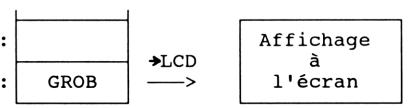


Ici **GROB2** est un objet graphique extrait de **GROB1** et délimité par la fenêtre dont les points A,B (de positions Position-A et Posi--tionB dans **GROB1**) sont 2 coins opposés.

Ici **GROB1** peut être remplacé par le nom **PICT** (**GROB2** est alors directement extrait du contenu de **PICT**).
Si la fenêtre définie sur **GROB1** par les deux points A et B "dépasse" de **GROB1**, il n'y a pas d'erreur: seuls les points appartenant réellement à **GROB1** sont pris en considération.
Par contre, il se produit une erreur si aucun des deux points A et B n'est en fait élément de **GROB1**.

→LCD

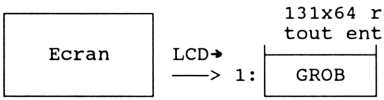
L'instruction **→LCD** affiche à l'écran l'objet graphique placé au niveau 1 de la pile.



L'objet **GROB** est cadré dans le coin supérieur gauche de l'écran.
Cet affichage recouvre l'écran tout entier, à l'exception de la ligne des menus (**GROB** étant complété "en blanc" s'il est de dimensions inférieures à cette partie de l'écran).
Si les dimensions de **GROB** sont supérieures à cette partie de l'écran, les points excédentaires ne sont pas affichés (en tout cas il n'y a pas d'erreur).
Le nom **PICT** ne peut pas servir ici à désigner le contenu de **PICT** (on peut faire **PICT RCL →LCD**).
Si on est en mode d'éxécution directe, l'affichage obtenu par **→LCD** persiste jusqu'à ce qu'une touche soit actionnée (**→LCD** "gèle" donc elle-même son affichage).
Si par contre l'instruction **→LCD** est éxécutée dans un programme, il faudra utiliser **FREEZE** pour geler l'affichage.

LCD→

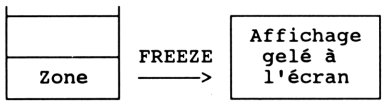
L'instruction **LCD→** crée l'objet graphique **GROB** de taille 131x64 représentant l'image de l'écran tout entier (y compris donc la ligne des menus si elle est présente).



Si l'instruction **LCD→** est éxécutée dans un programme, l'objet graphique obtenu reflète l'état de l'écran au moment où le programme a été lancé (ou au moment où le programme, suspendu par un **HALT**, a été relancé). L'image obtenue n'est donc pas nécessairement celle de la pile au moment où l'instruction **LCD→** est rencontrée.

FREEZE

L'instruction **FREEZE** permet de "geler" tout ou partie de l'affichage de l'écran.



Il faut attendre qu'une touche soit actionnée pour que l'affichage soit remis à jour.
"Zone" est ici un entier qui définit la partie de l'écran qui est "gelée".

Zone = 4*M+2*P+E, avec:
M=1 si on veut geler la ligne de menu, 0 sinon.
P=1 si on veut geler la partie correspondant à la pile, 0 sinon.
E=1 si on veut "geler" la ligne d'état, 0 sinon.
Ainsi Zone = 4+2+1 = 7 si on veut "geler" tout l'écran.

L'instruction **FREEZE** est utile pour qu'un affichage obtenu par **PVIEW** (sauf **{ } PVIEW**), ou **→LCD** persiste après l'achèvement de la séquence d'instructions en cours.

TEXT

L'instruction **TEXT** (qui ne prend pas d'argument sur la pile et n'y renvoie aucun résultat) provoque le retour à l'affichage de la pile, si précédemment le contenu de **PICT** a été affiché par **PVIEW.**

Considérons par exemple les programmes suivants:

```
'PROG1': « CLEAR  {#0 #0}  PVIEW  1  WAIT  »
'PROG2': « CLEAR  {#0 #0}  PVIEW  1  WAIT  TEXT  2  WAIT »
```

On appelle l'un quelconque de ces programmes, après avoir par exemple placé 4,3,2,1 aux niveaux 4,3,2,1.

'PROG1' affiche le contenu de **PICT** pendant une seconde, puis la fin du programme marque le retour à l'affichage de la pile (vidée par **CLEAR**).

'PROG2' affiche de même le contenu de **PICT** pendant une seconde, puis l'instruction **TEXT** ramène à l'affichage de la pile au moment où le programme a été appelé (pile "4,3,2,1"), cet affichage de pile durant 2 secondes.
'PROG2' s'arrête alors sur un retour à l'affichage de la pile en cours (donc la pile vidée par **CLEAR**)

Si on fait suivre l'instruction **{#0 #0} PVIEW** de **7 FREEZE**, alors le retour à l'affichage de la pile actuelle (dans **PROG1** comme dans **PROG2**) ne s'effectue que lorsque vous actionnez une touche.

On remarquera également que **FREEZE** gèle l'affichage obtenu par **PVIEW** ou par **TEXT** (si ces 2 instructions sont appelées dans un programme, c'est l'écran obtenu par la dernière d'entre elles qui est "gelé").

Enfin il est bon de savoir que l'instruction **TEXT** est sans effet sur un affichage graphique obtenu par **→LCD.**

LE MENU PLOT (SAUF PLOTR):

Dans le menu **PLOT**, et plus particulièrement dans son sous-menu **PLOTR**, sont regroupées les instructions qui permettent de mener à bien des tracés à vocation mathématique.

Il y a plusieurs modes de tracés:

Tracés "mathématiques":
- * mode **FUNCTION** (on trace des fonctions du type Y=F(X)).
- * mode **CONIC** (on trace des coniques du type F(X,Y)=0 où F est une fonction polynôme de degré 2 au plus en X,Y)
- * mode **POLAR** (on trace des courbes en polaires R=F(θ)).
- * mode **PARAMETRIC** (on trace des courbes X=F(T), Y=G(T)).
- * mode **TRUTH** (on trace des zones correspondant à la valeur logique VRAI d'une expression booléenne).

Tracés "statistiques":
- * mode **BAR** (diagrammes à barre des données figurant dans une colonne de la matrice statistique **ΣDAT**).
- * mode **HISTOGRAM** (histogramme des fréquences des données apparaissant dans une colonne donnée de **ΣDAT**).
- * mode **SCATTER** (nuage des points (x,y) où les (x,y) sont les couples de valeurs se correspondant dans deux colonnes spécifiées de **ΣDAT**).

Nous ne considérerons ici que les tracés "mathématiques".

L'instruction **DRAW** effectue le tracé de l'expression en cours en utilisant des paramètres de tracés.

L'expression en cours peut être une expression algébrique, un programme satisfaisant à la syntaxe algébrique, une équation, ou même une liste de tels objets (si on souhaite tracer plusieurs courbes successivement).
Elle doit être placée dans la variable réservée **'EQ'**.

On peut encore placer dans **'EQ'** le nom d'une variable dont le contenu est une expression (ce qui est un moyen de protéger celle-ci des changements que vous pourriez faire subir au contenu de **'EQ'**).

Les paramètres de tracés doivent être placés dans une variable réservée **PPAR**. Nous verrons que le sous-menu **PLOTR** permet de régler à sa guise ces différents paramètres.

Il peut exister une variable **'EQ'** et une variable **'PPAR'** par répertoire.
Si vous entrez dans **PLOT** sans avoir défini le contenu de **'EQ'**, un message vous invite à le faire (sinon un message vous donne le contenu de **'EQ'** ainsi que le type de tracé en cours).
Par exemple:

```
Plot type: FUNCTION
EQ: 'SIN(X)'
```

Si vous entrez dans **PLOT** sans avoir défini de variable **'PPAR'**, la **HP48SX** la crée automatiquement avec ses valeurs par défaut, qui sont:
'PPAR' : { (-6.5,-3.1) (6.5,3.2) X 0 (0,0) FUNCTION Y }
Le sens du contenu de **'PPAR'** est défini plus loin.

*** CRÉÉR L'EXPRESSION EN COURS:**

C'est un préalable obligatoire avant que de passer dans le sous-menu **PLOTR**.
On passe dans le menu **PLOT** en actionnant la touche correspondante au clavier.

Le menu **PLOT** contient les entrées:
NEW, EDEQ, STEQ, CAT
que l'on utlise pour créer ou modifier le contenu de l'expression en cours, d'une façon en tout point identique à celle décrite dans le chapitre précédent (pages 246 à 249).
On se reportera à ces pages où sont décrites les instructions communes aux menus **PLOT** et **SOLVE**.

*** CHOISIR LE MODE DE TRACÉ:**

Il est également recommandé de choisir le type de tracé avant d'entrer dans le sous-menu **PLOTR** (bien que ce ne soit pas obligatoire: le sous-menu **PLOTR** contient lui-même une entrée de menu **PTYPE**, permettant de le faire).

Le menu **PLOT** contient une entrée **PTYPE** de sous-menu, lui même contenant les entrées:
FUNC (pour **FUNCTION**), **CONIC**, **POLAR**, **PARA** (pour **PARAMETRIC**) et **TRUTH**.
Les autres entrées **BAR**, **HIST**, et **SCATT** concernent les tracés statistiques.

Une pression sur une touche de menu désignant un mode de tracé fait de celui-ci le mode de tracé courant, et provoque un retour au menu **PLOT** (cependant qu'un message confirme).
Le nom désignant ce mode de tracé est alors placé dans la variable **'PPAR'**, en sixième et avant-dernière position.

*** L'EXPRESSION EN COURS ET LE MODE DE TRACÉ:**

Le mode de tracé indique à la HP48SX de quelle manière elle doit interpréter le contenu de la variable **'EQ'**.

On utilise ici, par commodité, certains noms de variable comme X, Y, T, θ:
Ces noms n'ont évidemment rien d'obligatoire!

Dans le mode **FUNCTION:**
L'expression doit être du type F(X), ou F(X)=G(X).
Dans le premier cas, la HP48SX trace les points (X,F(X)).
Dans le deuxième cas, elle trace les points (X,F(X)) et les points (X,G(X)).
Le mode **FUNCTION** est le mode par défaut de la HP48SX.

Dans le mode CONIC:
L'expression doit être du type F(X,Y), ou F(X,Y)=G(X,Y), F et G étant des polynômes du second degré (au plus) par rapport aux variables X et Y.
Dans le premier cas, la HP48SX trace la conique d'équation F(X,Y)=0.
Dans le deuxième cas, elle trace les coniques d'équations F(X,Y)=0 et G(X,Y)=0.

Dans le mode POLAR:
L'expression doit être du type F(θ), ou du type F(θ)=G(θ).
Dans le premier cas, la HP48SX trace les points de coordonnées polaires (F(θ),θ) (c'est-à-dire la courbe R=F(θ)).
Dans le deuxième cas, elle trace les courbes en polaires R=F(θ) et R=G(θ).

Dans le mode PARAMETRIC:
L'expression doit être du type F(T), ou du type F(T)=G(T), où les fonctions F et G **sont à valeurs complexes**.
Dans le premier cas, la HP48SX trace la courbe paramétrée:
X = RE(F(T)) , Y = IM(F(T)).
Dans le deuxième cas, elle trace les courbes paramétrées:
d'abord: X = RE(F(T)) , Y = IM(F(T))
ensuite: X = RE(G(T)) , Y = IM(G(T)).

Dans le mode TRUTH:
L'expression doit être du type F(X,Y) où la fonction F(X,Y) est à valeurs réelles (en principe à valeurs 0 ou 1 c'est-à-dire ayant un caractère de test logique).
La HP48SX trace l'ensemble des points (X,Y) pour lesquels la valeur de F(X,Y) est non nulle (le test F(X,Y) est vrai).

*** TRACÉS DE LISTES D'EXPRESSIONS:**

Dans les modes **FUNCTION, CONIC, POLAR, PARAMETRIC,** il est possible de définir, comme expression courante, une liste d'expressions ou d'équations
La HP48SX tracera alors successivement, sur un même écran les courbes résultant de ces différentes expressions.
Bien sûr, ces différentes expressions seront interprétées selon le même mode de tracé.

*** TRACÉS D'EXPRESSIONS OU DE PROGRAMMES ?**

La HP48SX trace des expressions algébriques ou des programmes.
Un tel programme doit respecter la syntaxe algébrique (ne rien prendre sur la pile et y déposer un réel au niveau 1).

Par exemple:
'LN(EXP(X)-X-1)' équivaut à **« X EXP LAST - 1 - LN »**
(les temps de tracé sont les mêmes...)
Certaines instructions de la HP48SX sont interdites dans les expressions algébriques. Il pourra alors devenir nécessaire d'utiliser un programme pour écrire la fonction à tracer.
Exemple: Si A est une matrice carrée,le programme suivant calcule les valeurs du polynôme caractéristique de A, c'est-à-dire le polynôme P(X)=Det(A-XI):
« X A DUP IDN ROT * - DET »
et ce programme peut être placé dans **'EQ'** pour être tracé.

*** TRACÉ D'UNE EQUATION:**

Si l'expression en cours est une équation, la HP48SX trace simultanément les deux memebres de l'équation.
Il y a cependant une petite exception:
Si le membre gauche de l'équation est un nom (et même si ce nom celui d'une variable dont le contenu est une expression).
Si de plus l'indicateur -30 est désarmé (à 0, valeur par défaut) alors seul le membre droit de l'équation est tracé.
Dans toute la suite, on supposera qu'il n'en est pas ainsi.

*** TRACÉS PAR SEGMENTS OU PAR POINTS:**

Dans les modes de tracé **FUNCTION**, **CONIC**, **POLAR** ou **PARAMETRIC**, la HP48SX trace les segments reliant les différents couples successifs de points obtenus.

Ce n'est néanmoins qu'une situation "par défaut", qu'il est possible de changer:

Dans le menu **MODES** (page 2): l'entrée **CNCT** peut être activée (tracés "connectés") ou désactivée (tracés "déconnectés").
Le caractère actif de cette option est reconnaissable à la présence du petit carré ▪.
On peut également intervenir par programme en positionnant l'indicateur -31.
-31 CF (désarme le flag -31): tracés connectés.
-31 SF (arme le flag -31): tracés non connectés.

LE MENU PLOTR:

Le menu **PLOT** a permis de choisir l'expression en cours et le mode de tracé.

Le menu **PLOTR** est accessible:
- Par son entrée dans le menu **PLOT**
- Par la séquence ╔═> **PLOT**.

L'entrée dans **PLOTR** s'accompagne d'un message indiquant l'état des principaux paramètres de traçage, ainsi que le contenu de l'expression en cours.

Exemple:

```
Plot type: FUNCTION
EQ: 'SIN(X)'
Indep: 'X'
x:      -6.5      6.5
y:      -3.1      3.2
```

Indep indique le nom de la variable indépendante (celle dont les valeurs sont utilisées pour calculer les coordonnées des points à tracer).
x: et **y:** indiquent les valeurs extrêmes sur les axes de coordonnées (on parle respectivement de la gamme horizontale et de la gamme verticale).

A tout moment, dans le menu **PLOTR**, on pourra afficher de nouveau ces renseignements par un appui sur la touche **REVIEW**.

Passons en revue les différentes entrées du menu **PLOTR**. Toutes ces entrées correspondent à des instructions programmables.

Certaines d'entres elles n'ont pas le même fonctionnement selon qu'elles sont appelées par la touche de menu dans **PLOTR** ou qu'elles apparaissent dans un programme.

ERASE

Cette instruction efface le contenu de **PICT**, sans en modifier les dimensions.
ERASE est utile si on veut démarrer un tracé, pour débarasser **PICT** de l'image d'un tracé précédent.

DRAW

DRAW affiche le contenu de **PICT**, et y trace l'expression en cours (sans effacer ce qui peut déjà se trouver dans **PICT**, d'où l'utilité de **ERASE**)

Si **DRAW** est appelé par sa touche de menu dans **PLOTR**, il commence par tracer les axes de coordonnées.
Le tracé terminé, on se retrouve dans l'"environnement graphique" (voir plus loin). Ce tracé peut d'ailleurs être prématurément interrompu par un appui sur **ON**.

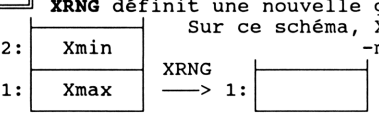
Si **DRAW** est utilisé dans un programme, il ne trace pas les axes (on utilisera alors l'instruction **DRAX**) et la fin du tracé ne signifie pas le passage dans l'environnement **GRAPHICS** (on utilisera l'instruction **GRAPH**).

AUTO

En mode **FUNCTION**, **AUTO** trace l'expression en cours en modifiant l'échelle verticale de manière à ce que le tracé occupe au mieux l'écran.
Si le mode de tracé est **PARAMETRIC** ou **POLAR**, l'instruction **AUTO** cadre au mieux les échelles horizontale et verticale, avant de tracer l'expression.
Le tracé terminé, on se retrouve dans l'"environnement graphique" (voir plus loin). Ce tracé peut d'ailleurs être prématurément interrompu par un appui sur **ON**.
Dans un programme,**AUTO** se contente d'effectuer les modifications d'échelle, sans effectuer le tracé.

XRNG

XRNG définit une nouvelle gamme verticale.



Sur ce schéma, Xmin et Xmax deviennent respectivement la valeur minimum et la valeur maximum de la gamme horizontale. Cette modification est confirmée, dans le menu **PLOTR**, par l'affichage de l'"état en cours".

YRNG analogue à **XRNG**, mais pour la gamme verticale.

Remarques sur les échelles obtenues par XRNG et YRNG:

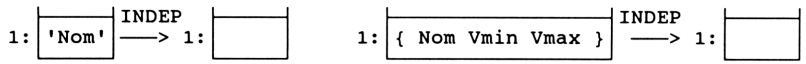
YRNG est utile avec **DRAW** mais pas avec **AUTO** (qui génère la gamme d'affichage vertical permettant le meilleur cadrage).

Il est à remarquer que **XRNG** et **YRNG** ne modifient rien au contenu de **PICT** (qui correspond peut-être pourtant à un tracé de fonction avec une autre échelle).
Il est donc prudent d'associer **XRNG** et **YRNG** à l'appel de **ERASE**.

Les deux premiers éléments de la variable **PPAR** sont:
Le complexe des coordonnées du point inférieur gauche A.
Le complexe des coordonnées du point supérieur droit B.
Par défaut: **PPAR** = { (-6.5,-3.1) (6.5,3.2) .... }

Les instructions **XRNG** et **YRNG** modifient donc les coordonnées de ces points extrêmes A = (Xmin, Ymin), et B = (Xmax, Ymax).

INDEP L'instruction **INDEP** précise le nom, et éventuellement la gamme des valeurs, de la variable indépendante.



'Nom' est ici le nom donné à la variable indépendante (Par défaut la HP48SX considère que c'est 'X').
La variable indépendante est celle dont les différentes valeurs permettent de calculer les coordonnées des points à tracer.

Dans le mode **FUNCTION** la variable indépendante est naturellement associée aux valeurs de l'abscisse.
Dans le mode **POLAR**, la variable indépendante définit l'angle polaire des points à tracer.
Dans le mode **PARAMETRIC**, la variable indépendante donne les valeurs du paramètre T définissant la position (X(T),Y(T)) du point courant (les noms X,Y,T sont ici donnés à titre indicatif).
Dans les modes **CONIC** et **TRUTH** qui nécessitent l'évaluation de 2 variables, il convient de définir une variable indépendante par **INDEP** (associée à l'axe horizontal) et une variable dépendante par **DEPND** (associée à l'axe vertical).

Dans le deuxième diagramme ci-dessus, on donne non seulement le nom de la variable indépendante, mais encore la gamme des valeurs qu'elle est autorisée à prendre (de la valeur minimum **Vmin** à la valeur maximum **Vmax**)

La gamme des valeurs que prend la variable indépendante est, par défaut, celle associée à l'axe horizontal.

Dans les modes **POLAR** et **PARAMETRIC**, la gamme des valeurs de la variable indépendante n'a en général aucun rapport avec la gamme de l'axe horizontal.
Vous pourrez ainsi tracer (en mode radians) la courbe en polaire d'équation R=1+COS(T), en définissant:
* L'expression **'1+COS(T)'** dans **'EQ'**.
* Le mode de tracé **POLAR**.
* Une gamme horizontale de -1 à 2: **1 +/- 2 XRNG**
* Une gamme verticale de -2 à 2: **2 +/- 2 YRNG**
* Une variable indépendante T variant de 0 à 6.28 (≈2π):
{ T 0 6.28 } INDEP

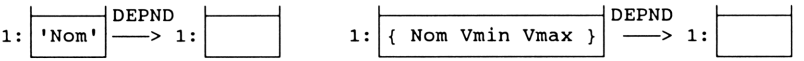
Dans les modes **FUNCTION**, **CONIC** et **TRUTH**, le fait de définir une gamme de valeurs de la variable indépendante strictement incluse dans la gamme de l'axe horizontal permet de limiter le tracé et donc de gagner un peu de temps.

Le nom de la variable indépendante, associé éventuellement à la gamme des valeurs correspondante dans une liste, est le 3-ème élément de la variable **PPAR** contenant les paramètres de tracé.

Il est tout à fait possible de modifier la gamme des valeurs de la variable indépendante, sans rappeler le nom de celle-ci:
Par exemple: **0 5 INDEP**
ou encore **{ 0 5 } INDEP**
définit la gamme [0,5] comme nouvelle gamme de valeurs pour la variable dépendante, sans changer le nom de celle-ci.

DEPN Le label **"DEPN"** correspond ici à l'instruction **DEPND.**

L'instruction **DEPND** permet de définir, dans des conditions analogues à **INDEP**, le nom (et éventuellement la gamme des valeurs [Vmin,Vmax]) de la variable dépendante.



La variable dépendante est utilisée pour les tracés en mode **CONIC** et **TRUTH.** Elle est alors associée à l'axe vertical.

Si on ne précise pas la gamme des valeurs de cette variable dépendante, la HP48SX utilise celle de l'axe y.

Supposons que la gamme de "l'axe y" soit -3.1, 3.1 (valeurs par défaut). On peut définir une gamme -1, 1 pour la variable dépendante (ce qui limitera le tracé dans la bande horizontale contenue entre l'ordonnée y=-1 et l'ordonnée y=1).

Une telle limitation peut permettre de gagner du temps dans les tracés, notamment ceux en mode **TRUTH.**

Si par exemple on veut tracer l'ensemble des points vérifiant '(X+3*(X²+Y²)*Y)² > 1', on peut procéder comme suit:

* Placer '(X+3*(X^2+Y^2*Y)^2>1' dans **'EQ'.**

* Sélectionner le mode **TRUTH.**

* Laisser les gammes horizontale et verticale à leurs valeurs par défaut (x:-6.5 6.5 et y:-3.1 3.2).

* Prendre X comme variable indépendante variant de -2 à 2:
  { X -2 2 } INDEP.

* Prendre Y comme variable dépendante variant de -2 à 2:
  { Y -2 2 } DEPND.

* Tracer par **DRAW** (après avoir fait **ERASE** le cas échéant)

Le nom de la variable dépendante, associé éventuellement à la gamme des valeurs correspondante dans une liste, est le dernier élément de la variable **PPAR** (paramètres de tracé).

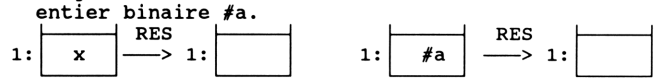
Il est tout à fait possible de modifier la gamme des valeurs de la variable dépendante, sans rappeler le nom de celle-ci:

Par exemple: **0 5 DEPND**
ou encore **{ 0 5 } DEPND**
définit la gamme [0,5] comme nouvelle gamme de valeurs pour la variable dépendante, sans changer le nom de celle-ci.

PTYPE L'entrée **PTYPE**, dans le menu **PLOTR**, permet de choisir le mode de tracé (sous-menu analogue à celui du menu **PLOT**).

RES

L'instruction **RES** permet de définir la résolution du tracé.
L'instruction **RES** prend un argument déterminant l'intervalle entre deux valeurs consécutives de la variable indépendante.
Cet argument est donné sous forme d'un nombre réel x ou d'un entier binaire #a.



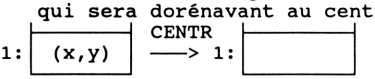
Dans le premier cas ci-dessus: la variable indépendante sera incrémentée de x unités-utilisateur à chaque calcul d'un nouveau point.
Cette méthode s'applique à tous les modes de tracé.

Dans le second cas ci-dessus, qui ne s'applique qu'aux modes de tracé **FUNCTION,CONIC,TRUTH** (c'est-à-dire ceux où la variable indépendante est directement reliée à l'axe horizontal):
La variable indépendante est incrémentée de la valeur qui correspond à un déplacement de #a pixels, à chaque calcul d'un nouveau point.

La valeur x=0, ou la valeur #a=#0 produisent un retour aux valeurs par défaut qui sont:
En mode **FUNCTION**, **CONIC**, et **TRUTH**: un pixel.
En mode **POLAR**: 2°, 2 grades, ou $\pi/90$ radians, suivant le mode en cours de calcul des angles.
En mode **PARAMETRIC**: 1/130 de l'amplitude de la gamme des valeurs de la variable indépendante.
Evidemment, une augmentation de **RES** se traduit par une moindre précision dans le tracé (mais celui-ci est plus rapide).
Le réel ou l'entier binaire définissant la résolution est le quatrième élément de la variable **PPAR** contenant les paramètres de tracé.

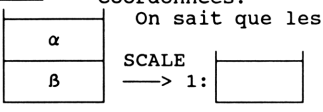
CENT Le label "**CENT**" correspond ici à l'instruction **CENTR**.

L'instruction **CENTR** permet de choisir les coordonnées du point qui sera dorénavant au centre de l'écran.



Ici le fait de définir le point de coordonnées (x,y) comme point central de l'affichage modifie en principe les gammes des axes de coordonnées.

SCALE L'instruction **SCALE** définit l'échelle sur les axes de coordonnées.



On sait que les axes sont gradués tous les 10 pixels:
La séquence: α ß **SCALE**
(où α et ß sont deux réels > 0)
signifie que: 10 pixels correspondent à: α unités sur l'axe horizontal.
ß unités sur l'axe vertical.

L'instruction **SCALE** influe en général sur les gammes des axes de coordonnées.
Elle est utile pour que les pixels de l'écran, sur lesquels on peut déplacer le réticule dans l'application **GRAPHICS**, aient une abscisse et une ordonnée simple.

RESET

Attention **RESET** n'est pas une instruction de la HP48SX: elle n'est accessible que dans le menu **PLOTR**.

RESET réinitialise à leurs valeurs par défaut:

* Les gammes de valeurs des axes de coorodonnées:
  Axe x: -6.5 à 6.5
  Axe y: -3.1 à 3.2

* Les variables dépendante et indépendante:
  Variable dépendante: 'X'.
  Variable indépendante: 'Y'.

* La résolution: (voir ci-dessus l'instruction **RES**).
  0 est placé en 4-ème position dans **'PPAR'**.

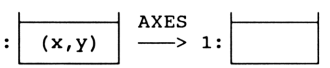
* Les coordonnées du point d'intersection des axes: en (0,0).

* les dimensions de **PICT** (et efface **PICT**):
  131 pixels de large et 64 pixels de haut, c'est-à-dire les dimensions de l'écran.

RESET ne modifie pas le mode de tracé en cours (ce qui est heureux si on vient de le définir dans le menu **PLOT** avant d'entrer dans le sous-menu **PLOTR**).

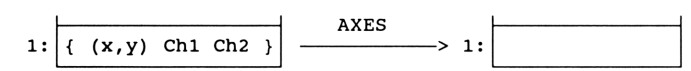
AXES

L'instruction **AXES** permet de choisir les coordonnées du point où viendront se croiser les axes horizontal et vertical.



Ici on désigne le point (x,y) comme point d'intersection des axes.
Par défaut, le point d'intersection des axes de coordonnées est le point (0,0).
Ce point est placé en 5-ème position dans la variable **'PPAR'** des paramètres de tracé.

L'instruction **AXES** admet également le schéma fonctionnel suivant



Ch1 et Ch2 sont ici des chaînes de caractères qui seront ajoutées aux extrémités des axes de coordonnées, pour les commenter, si on utilise l'instruction **LABEL** (voir plus loin).

DRAX

L'instruction **DRAX** trace dans **PICT** les axes de coordonnées.

Rappelons que **DRAW** et **AUTO**, s'ils sont appelés en utilisant les touches correspondantes dans le menu **PLOTR**, éxécutent **DRAX** avant de tracer l'expression en cours.
Par contre, éxécutées dans un programme, **DRAW** et **AUTO** ne tracent pas les axes de coordonnées.

LABEL

L'instruction **LABEL** ajoute des commentaires aux extrémités des axes de coordonnées (donc dans le contenu de **PICT**).

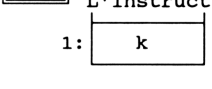
Ces "commentaires" sont:
* Les valeurs extrêmes de l'ordonnée et de l'abscisse (en respectnat le format d'affichage des réels).
* Les noms de la variable indépendante (à l'extrémité positive de l'axe x) et de la variable dépendante (à l'extrémité positive de l'axe y).

Ces noms peuvent être remplacés par des commentaires tels que définis par l'instruction **AXES** (voir ci-dessus).

Remarque:
Il n'est guère possible d'effacer de tels "commentaires" (mieux vaudrait tout retracer)

*H

L'instruction ***H** permet de "zoomer" suivant l'axe horizontal.

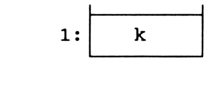


k est ici un réel positif.
Il représente le coefficient par lequel on multiplie l'échelle horizontale.

Si k>1, on effectue donc un zoom "arrière", car la nouvelle gamme horizontale est k fois plus large qu'auparavant.
Si k<1, on efectue un zoom "avant" (Si par exemple k=0.5, la nouvelle gamme horizontale est deux fois plus étroite, pour une largeur de **PICT** inchangée).

*W

L'instruction ***W** permet de "zoomer" suivant l'axe vertical, d'une manière analogue à ce que fait ***H** pour l'axe horizontal.

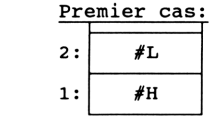


k est ici un réel positif.
Il représente le coefficient par lequel on multiplie l'échelle verticale.

PDIM

L'instruction **PDIM** permet de définir les dimensions de **PICT**.

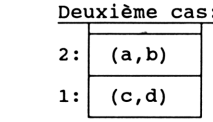
Premier cas:



Ici on définit un nouveau **PICT**:
- de largeur #L pixels.
- de hauteur #H pixels.

Les points inférieur gauche et supérieur droit gardant les mêmes coordonnées, on modifie donc l'échelle.

Deuxième cas:



Ici on définit un nouveau **PICT**:
- dont le coin inférieur gauche a pour coordonnées (a,b).
- dont le coin supérieur droit a pour coordonnées (c,d).

(et ceci sans changer les échelles horizontale et verticale, ce qui modifie bien les dimensions de **PICT**).

Dans les deux cas, le contenu précédent de **PICT** est perdu.

REMARQUES DIVERSES SUR PLOTR:

La touche ╔=>, associée aux touches du menu **PLOTR**, permet de rappeler sur la pile les principaux paramètres de tracé.

╔=> **DRAW:** éxécute **RCEQ** (rappelle l'expression en cours).
(<=╗ **DRAW** éxécute **STEQ**).

╔=> **XRNG:** renvoie la gamme horizontale, aux niveaux 2 (pour Xmin) et 1 (pour Xmax).

╔=> **YRNG:** renvoie la gamme verticale, aux niveaux 2 (pour Ymin) et 1 (pour Ymax).

╔=> **INDEP:** renvoie le nom de la variable indépendante (placée en tête d'une liste { Nom Vmin Vmax } si on a défini une gamme de valeurs pour cette variable).

╔=> **DEPN:** renvoie le nom de la variable dépendante (placée en tête d'une liste { Nom Vmin Vmax } si on a défini une gamme de valeurs pour cette variable).

╔=> **RES:** renvoie le réel ou l'entier binaire qui indique la résolution.

╔=> **CENT:** renvoie les coordonnées du point central de l'écran.

╔=> **SCALE:** renvoie l'échelle horizontale (au niveau 2) et l'échelle verticale (au niveau 1).

╔=> **AXES:** renvoie les coordonnées du point où se coupent les axes de coordonnées.

╔=> **PDIM:** renvoie les dimensions de **PDIM** (largeur au niveau 2 et hauteur au niveau 1), en entiers binaires.

Il est bien sûr possible de prendre connaissance de nombreux paramètres de tracé:

* en rappelant le contenu de **PPAR** sur la pile.

* en activant la touche **REVIEW** dans le menu **PLOTR**.

Si vous avez modifié certains paramètres de tracé, il est très recommandé de faire précéder l'appel de **DRAW** ou de **AUTO** d'un appel de **ERASE** pour nettoyer le contenu de **PICT** des tracés qui pourraient encore s'y trouver (car obtenus avec des paramètres de tracé différents).

L'ENVIRONNEMENT GRAPHICS (GÉNÉRALITÉS):

Quand on lance le tracé de l'expression en cours, depuis le menu **PLOTR**, au moyen des touches de menu **DRAW** ou **AUTO**, le tracé est effectué, puis la HP48SX vous place dans un "environnement" particulier appelé **GRAPHICS**, où il vous est possible d'effectuer une analyse approfondie du tracé précédent.
Si on appuie sur **ON** pendant le tracé, on passe immédiatement dans l'environnement **GRAPHICS**.

N.B: Dans un programme , l'instruction **DRAW** dessine la courbe à l'écran (**PICT** est donc visible momentanément) et la fin normale du programme provoque un retour à l'affichage de la pile (en attendant éventuellement que vous actionniez une touche si vous avez "gelé" l'affichage avec **FREEZE**).

On peut également accéder à **GRAPHICS** directement, par un appui sur la touche **GRAPH** du clavier (préfixée par <=╗).

L'environnement **GRAPHICS** permet de visualiser le contenu de **PICT** (notamment si ses dimensions excèdent celles de l'écran) , d'y intervenir, et d'analyser certains aspects du tracé de l'expres-sion en cours.

L'environnement **GRAPHICS** donne accès à 3 pages de menu:
La première sert à analyser l'expression en cours et son tracé. Encore faut-il que vous ayez défini une expression en cours (dans la variable **'EQ'**).
Les deux pages suivantes permettent de modifier le contenu de **PICT**, d'une manière très "conviviale".

Un réticule peut être déplacé par l'utilisateur pour accéder à tel ou tel point de l'écran. A l'entrée dans **GRAPHICS**, ce réticule est au centre de l'écran (il est souvent masqué par les axes de coordonnées après un tracé mathématique).

Il est possible de placer une "marque" à la position du réticule, une seule marque pouvant être placée un instant donné: cette marque sert à définir, avec la nouvelle position du réticule, le couple de points utile à certaines instructions.

On sort de l'environnement **GRAPHICS** en appuyant sur la touche **ON**.

Il est possible d'accéder à **GRAPHICS** dans un programme, au moyen de l'instruction **GRAPH**. Un appui sur la touche **ON** permet alors de poursuivre l'éxécution de ce programme.

* UTILISATION DES TOUCHES DE CLAVIER (SAUF MENUS):

On suppose donc que l'on vient d'entrer dans **GRAPHICS**.
Certaines touches du clavier (hormis les touches de menu) ont alors un rôle particulier:

Δ ∇ ◁ ▷ : déplacent le réticule d'un pixel dans la direction correspondante.
Préfixées par ╔=>, elles permettent d'accéder directement au bord de l'écran, puis à l'extrémité de **PICT** dans cette direction.

GRAPH: active le défilement de **PICT** en mode scrolling (utile si les dimensions de **PICT** sont bien supérieures à celles de l'écran). Dans le mode scrolling, les touches de menu sont effacées.
Un nouvel appui sur **GRAPH** ramène à **GRAPHICS**.
REVIEW: affiche l'état des paramètres de tracé (l'affichage persistant tant que la touche ∇ est pressée).
STO: place sur le contenu de **PICT** sur la pile (mais on demeure dans **GRAPHICS**).
ENTER: place sur la pile les coordonnées (x,y) du point où se trouve le réticule.

+/- : affiche le réticule en mode XOR (utile s'il se déplace sur un fond noir). Un nouvel appui sur **+/-** ramène à l'affichage normal.
DEL: Place une marque à la position du réticule si aucune marque n'est encore placée.
Si une marque est déjà placée, efface le rectangle situé entre cette marque (qui n'est pas déplacée) et la position du réticule.
CLR: Efface le contenu de **PICT** (sauf la marque s'il y en a une).
***** : Place une marque à la position du réticule s'il n'y en a pas encore dans **PICT**.
Efface la marque si le réticule est sur cette marque.
S'il y a déjà une marque dans **PICT**, et si le réticule n'est pas placé sur cette marque, la marque est déplacée à la position du réticule.
- : efface la ligne du menu.
Un nouvel appui sur **-** restaure la ligne du menu.
+ : remplace la ligne du menu par l'affichage des coordonnées du point où se trouve le réticule.
Un nouvel appui sur **+** restaure la ligne du menu.
ON: quitte l'environnement **GRAPHICS**.

ANALYSE DE FONCTIONS DANS GRAPHICS:

On considère ici les entrées de la 1ère page du menu spécifique à l'environnement **GRAPHICS**.
On suppose que l'on a défini une expression en cours, et que l'on vient de la tracer (le plus souvent, on a effectué le tracé au moyen des entrées **DRAW** ou **AUTO** du menu **PLOTR**, et la fin de ce tracé a provoqué le passage dans **GRAPHICS**).

ZOOM

ZOOM est une entrée de menu dans **GRAPHICS**, qui autorise des opérations de zoom avant ou arrière sur le tracé.
Le menu **ZOOM** contient les entrées **XAUTO**, **X**, **Y**, **XY**, et **EXIT** (cette dernière ramenant au menu principal de **GRAPHICS**).

Les quatre options **XAUTO**, **X**, **Y**, **XY** demandent un coefficient de zoom k (on entre k en ligne de commande et on valide par **ENTER**. On peut renoncer en appuyant sur **ON**.
Le coefficient k doit être positif:
S'il est supérieur à 1, il s'agit d'un zoom arrière (en effet si k=2, par exemple, l'échelle est multipliée par 2, ce qui permet de voir deux fois plus de points dans cette direction, les dimensions de **PICT** restant les mêmes).
Si k est compris entre 0 et 1, il s'agit d'un zoom avant (d'autant plus prononcé que que k est proche de 0).

XAUTO: on multiplie l'échelle horizontale par k, et la gamme verticale est automatiquement ajustée par la HP48SX pour que le nouveau tracé occupe au mieux l'écran.
X: On multiplie l'échelle horizontale par k.
Y: On multiplie l'échelle verticale par k.
XY: On multiplie les deux échelles par le coefficient k.

Les quatre options **XAUTO**, **X**, **Y**, **XY** retracent la courbe en tenant compte des nouvelles gammes horizontale et/ou verticale.
C'est une différence avec les instructions ***H** et ***W** du menu **PLOTR**, qui modifient les échelles, sans retracer la courbe.

Z-BOX

S'il n'y a pas de marque: en place une à la position du réticule.

S'il y a une marque:

Si le réticule est placé de façon à délimiter, avec la marque, une zone rectangulaire d'intérieur non vide: **Z-BOX** retrace la courbe en prenant pour gammes de coordonnées celles qui correspondent à cette zone (on "zoome" dans ce rectangle).

Si le réticule et la marque délimitent un segment horizontal: **Z-BOX** retrace la courbe en conservant la gamme verticale, mais avec la gamme horizontale définie par ce segment.

Si le réticule et la marque délimitent un segment vertical: **Z-BOX** retrace la courbe en conservant la gamme horizontale, mais avec la gamme verticale définie par ce segment.

Si on éxécute <=┐ **Z-BOX**, on obtient comme précédemment une modification de la gamme horizontale mais la gamme verticale est calculée, au mieux, par la HP48SX (autoscaling).

CENT

Cette option retrace la courbe, en faisant de la position du réticule le point central du nouveau tracé.

COORD

Remplace la ligne du menu par l'affichage des coordonnées du point où se trouve le réticule.

Un appui sur une touche de menu (ou sur les touches + ou -) restaure la ligne du menu.

Remarque: les déplacements du réticule se trouvent ralentis par le calcul des coordonnées du point qu'il désigne, quand ces coordonnées sont affichées.

LABEL

Ajoute des "commentaires " aux extrémités des axes (même fonctionnement que l'instruction **LABEL** qui est présente dans le menu **PLOTR**).

FCN

L'option **FCN** de la 1ère page du menu spécifique à **GRAPHICS** permet d'accéder à un sous-menu où on peut analyser l'expression en cours et son tracé.

Il est ici indispensable que le mode de tracé soit **FUNCTION.**
La variable **'EQ'** peut contenir une expression, une équation, ou une liste d'expressions ou d'équations.
L'expression en cours ne peut pas être un programme.

LE SOUS-MENU FCN DU MENU GRAPHICS:

Un certain nombre d'options du menu **FCN** conduisent à des résultats numériques.
Le résultat obtenu est alors affiché à la place de la ligne de menu (appuyer sur une touche de menu pour la réafficher) et est envoyé sur la pile sous forme d'un objet étiqueté, l'éti-quette (**Root**, **Area**, **Extrm**..) facilitant son identification ultérieure (quand on revient à l'affichage de la pile).

Le nom X utilisé ici pour désigner la variable indépendante est évidemment choisi pour faciliter les explications, mais il n'a rien d'obligatoire.

Il se peut que le contenu de **'EQ'** soit une liste:
{ F(X) G(X) H(X) ... }
Dans ce cas, les opérations du sous-menu **FCN** se rapportent à l'expression qui figure en tête de liste (**ISECT** constituant cependant un cas particulier).
Il est possible d'effectuer une permutation circulaire sur les éléments de cette liste,avec **NXEQ**, pour sélectionner l'entrée suivante de la liste.

Remarque: les options du menu **FCN** ne sont pas des instructions programmables (sauf l'homonymie avec **ROOT**).

ROOT

Cherche la racine de l'expression en cours, la plus proche de la position du réticule.

Par "racine de l'expression en cours", on entend ici:

* Si cette expression est effectivement du type 'F(X)' (sans le signe =): une valeur α qui annule F.

* Si cette expression est une équation F(X)=G(X): une valeur α qui annule G.

Le réticule est alors placé sur le point de coordonnées (α,0).
La valeur α est envoyée sur la pile (étiquette **Root**).

ISECT

Identique à **ROOT** dans le cas d'une seule expression F(X).

Dans le cas d'une seule équation F(X)=G(X), **ISECT** cherche une valeur α telle que F(α)=G(α).
Dans le cas où **'EQ'** contient une liste { F(X) G(X) H(X) .. }, **ISECT** cherche une valeur α telle que F(α)=G(α) (éventuellement en se limitant aux membres droits dans le cas où F et/ou G sont des équations).

La solution (α,ß) trouvée est la plus proche de la position actuelle du réticule.
Le réticule est alors placé sur le point (α,ß) qui représente la solution.
Le nombre complexe (α,ß) est envoyé sur la pile (avec l'étiquette **I-sect**) et affiché.

SLOPE — Calcule la dérivée F'(α), si F(X) est l'expression en cours, et si α est l'abscisse du réticule.

Le réticule est placé sur le point (α,F(α)).

Dans le cas où l'expression en cours est une équation G(X)=F(X), c'est le membre de droite F(X) qui est ici dérivé.
Le résultat F'(α) est affiché et envoyé sur la pile (avec l'étiquette **Slope**).

AREA — Si aucune marque n'est présente sur l'écran, **AREA** en place une à la position du réticule.
Sinon **AREA** calcule l'intégrale de l'expression F(X) en cours, de l'abscisse α de la marque à l'abscisse ß du réticule.

Dans le cas où l'expression en cours est une équation G(X)=F(X), c'est le membre de droite F(X) qui est ici intégré.

Le résultat I de l'intégration est affiché, et envoyé sur la pile (avec l'étiquette **Area**).

EXTR — **EXTR** recherche la racine α (la plus proche de l'abscisse du réticule) qui annule la dérivée de l'expression en cours.

Dans le cas d'une équation G(X)=F(X), c'est le membre de droite F qui est considéré.

Le réticule est placé au point (α,F(α)) (représentatif d'un point à tangente horizontale et donc le plus souvent d'un extremum).
Le nombre complexe (α,F(α)) est affiché et placé sur la pile avec l'étiquette **Extrm**.

EXIT — **EXIT** permet de sortir de **FCN** et de revenir à **GRAPHICS**.

F(X) — Cette option du menu **FCN** calcule précisément la valeur F(α) (de l'expression en cours F(X)) correspondant à l'abscisse α du réticule.

Dans le cas d'une équation G(X)=F(X), c'est le membre de droite F qui est considéré.

Le réticule est placé sur le point (α,F(α)).
La valeur F(α) est affichée et placée sur la pile ("tag" F(x)).

Remarque:
Il se peut que la solution trouvée par **ROOT**, **ISECT**, **EXTR** conduise à un point situé en dehors de la partie qui est visible à l'écran.
Dans ce cas, la HP48SX affiche brièvement le message **OFF SREEN** pour vous prévenir.

F' Cette option calcule la dérivée F'(X) de l'expression en cours, et la place en tête d'une liste constituée en outre du contenu précédant de **'EQ'**.

Les éléments de cette liste sont alors tracés (après effacement du contenu de **PICT**).

La dérivée F' devient donc l'expression en cours.

Si l'expression en cours au départ est une équation G(X)=F(X), c'est le membre de droite F(X) dont la dérivée est ajoutée en début de liste.

NXEQ Si le contenu de **'EQ'** est une liste { A(X) B(X) ... Z(X) }, alors **NXEQ** opère une permutation circulaire sur les objets de cette liste, le contenu de **'EQ'** devenant { B(X) ... Z(X) A(X) }.

La nouvelle expression en cours dans le sous-menu **FCN** est donc B(X), tandis que l'ancienne, A(X), a été repoussée à la fin.

Cette modification est confirmée par l'affichage de la nouvelle expression "prioritaire" dans **FCN**.

AJOUT D'ELEMENTS A PICT (MENU GRAPHICS):

Nous nous intéressons donc aux pages 2 et 3 du menu **GRAPHICS**.

DOT+ Cette fonction marche "en bascule". Elle est active si le label **DOT+** est suivi d'un petit carré ▪.
Si cette fonction est active, elle allume le pixel situé à la position du réticule. Les déplacements de celui-ci permettent donc d'effectuer des tracés.

DOT- Cette fonction marche "en bascule". Elle est active si le label **DOT-** est suivi d'un petit carré ▪.
Si cette fonction est active, elle éteint le pixel situé à la position du réticule. Les déplacements de celui-ci permettent donc d'effacer des tracés.

Remarque:

A un moment donné, une seule des deux fonctions **DOT+** et **DOT-** peut être activée.
Si vous activez **DOT+** et que **DOT-** est active, alors vous désactivez **DOT-** (et inversement).
Les options **DOT+** et **DOT-** peuvent très bien être toutes les deux inactives (les déplacements du réticule ne provoquant alors ni tracé ni effacement).

LINE

Si aucune marque n'est visible, **LINE** place une marque à la position du réticule.
Sinon **LINE** trace le segment reliant la position de la marque et celle du réticule.
Ce segment est tracé en mode "OR", c'est-à-dire qu'il recouvre ce qui se trouve sur son passage.
La marque est déplacée à la position du réticule.
Remarque: si le réticule est sur la marque, **LINE** efface la marque (mais allume le point)

TLINE

Si aucune marque n'est visible, **TLINE** place une marque à la position du réticule.
Sinon **TLINE** trace le segment reliant la position de la marque et celle du réticule.
Ce segment est tracé en mode "XOR", c'est-à-dire qu'il inverse l'état des pixels sur son passage.
TLINE et utile si on veut effacer un segment de droite tracé précédemment par **LINE**, ou pour tracer une droite sur un fond sombre.
La marque n'est pas déplacée à la position du réticule (deux appels successifs de **TLINE**, sans changer la position du réticule, s'annulent donc).

BOX

Si aucune marque n'est visible, **BOX** place une marque à la position du réticule.
Sinon **BOX** trace le rectangle dont deux coins opposés sont la position de la marque et celle du réticule.
Ce rectangle est tracé en mode "recouvrement".
La marque n'est pas déplacée par **BOX**.

CIRCL

Si aucune marque n'est visible, **CIRCL** place une marque à la position du réticule.
Sinon **CIRCL** trace le cercle dont le centre est la marque et qui passe par la position du réticule.
Ce cercle est tracé en mode "recouvrement".
La marque n'est pas déplacée par **CIRCL**.

MARK

Place la marque à la position du réticule.
Si le réticule est sur la marque, **MARK** efface cette marque.
(la touche * a le même rôle).

REPL

REPL nécessite qu'un objet-graphique soit plcaé sur la pile sinon un message d'erreur est brièvement affiché (sans pour autant nous faire sortir de l'environnement graphique).
REPL place alors l'objet graphique dans **PICT** en positionnant son coin supérieur gauche à la position du réticule (peu importe d'ailleurs si les dimensions de cet objet-graphique sont telles qu'il ne tienne pas entièrement).

SUB

Si aucune marque n'est visible, **SUB** place une marque à la position du réticule.
Sinon **SUB** place sur la pile l'objet-graphique représentant le rectangle dont 2 coins opposés sont la position de la marque et celle du réticule.
le rectangle n'est pas effacé de **PICT.**
La marque n'est pas déplacée par **BOX.**

DEL

Si aucune marque n'est visible, **DEL** place une marque à la position du réticule.
Sinon **DEL** efface le rectangle dont deux coins opposés sont la position de la marque et celle du réticule.
La marque n'est pas déplacée par **BOX.**

+/-

Cette fonction marche en bascule (elle est active si son label est suivi d'un petit carré ▪).
Si cette otpion est active, le réticule est affiché en mode **XOR:** (en noir sur fond blanc, en blanc sur fond noir).
Sinon il est affiché en mode "recouvrement".
La touche **+/-** du clavier a le même rôle.

KEYS

Cette fonction permet d'effacer la ligne de menu.
L'appui sur une touche de menu, ou sur la touche -, provoque le retour à l'affichage de cette ligne de menu.
La touche **-** du clavier a le même rôle que **KEYS.**

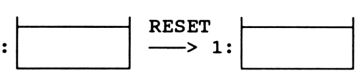
EXERCICES SUR LE GRAPHISME:

La difficulté des exercices est mesurée par des symboles * ou # :

- `*` Très facile.
- `**` Assez simple.
- `***` Déjà un peu plus délicat.
- `****` Plutôt difficile.
- `*****` Difficile.

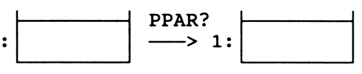
Les symboles # ont la même signification, mais désignent des exercices dont la solution utilise un ou plusieurs des programmes précédant dans cette liste d'exercices.

Exercice 1 (*): revenir au paramètres graphiques par défaut.



Le programme **'RESET'** doit placer dans **'PPAR'** la liste des para-mètres graphiques par défaut.

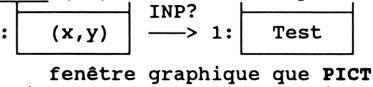
Exercice 2 (#): être certain de la présence d'un **'PPAR'** valable.



Le programme **'PPAR?'** doit tester la présence de **'PPAR'** dans le répertoire et la créer si elle ne s'y trouve pas.

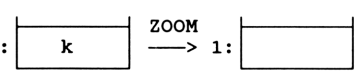
Le programme **'PPAR?'** doit également recréer **'PPAR'** si son contenu n'est pas conforme à ce qu'attend la HP48SX.

Exercice 3 (##): Tester si un point est dans la fenêtre graphique.



Il s'agit de tester si le point de coordonnées (x,y) du plan est ou n'est pas à l'intérieur de la fenêtre graphique que **PICT** ouvre sur le plan.
S'il y est, Test vaut 1, sinon Test vaut 0.

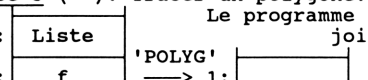
Exercice 4 (***): Zoom sur la fenêtre graphique.



Il s'agit ici d'effectuer un zoom sur la fenêtre graphique.
k (réel > 0) est le coefficient de ce zoom (zoom avant si k<1, zoom arrière si k>1).

Le centre de la fenêtre graphique doit être invariant dans cette opération (qui consiste donc à multiplier par le coefficient k l'amplitude de l'intervalle des 2 coordonnées,en conservant le centre de la fenêtre).

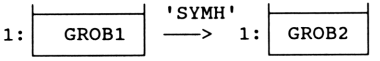
Exercice 5 (**): Tracer un polygône.



Le programme **'POLYG'** trace la ligne brisée qui joint les points (a,b),(c,d), etc.... qui figurent dans la liste placée au niveau 2.
Au niveau 1 le réel f vaut zéro si on ne veut pas que la ligne brisée se referme sur elle-même.
Si f est non nul, le dernier segment tracé est celui qui joint le dernier point (y,z) de la liste au premier point (a,b): on obtient ainsi un polygône fermé.

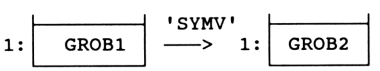
GRAPHISME: Exercices.

Exercice 6 (**): Symétrie horizontale sur un objet-graphique.



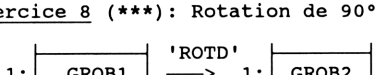
'SYMH' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une symétrie d'axe horizontal.

Exercice 7 (**): Symétrie verticale sur un objet-graphique.



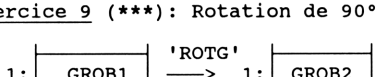
'SYMV' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une symétrie d'axe vertical.

Exercice 8 (***): Rotation de 90° à droite sur un objet-graphique.



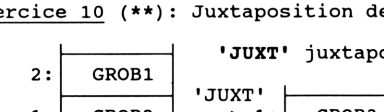
'ROTD' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une rotation de 90° dans le sens des aiguilles d'une montre (donc "vers la droite").

Exercice 9 (***): Rotation de 90° à gauche sur un objet-graphique.



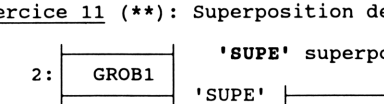
'ROTG' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une rotation de 90 degrés dans le sens inverse des aiguilles d'une montre (sens "trigonométrique").

Exercice 10 (**): Juxtaposition de deux objets-graphiques.



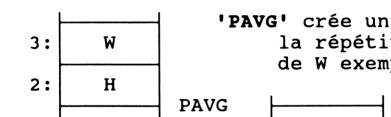
'JUXT' juxtapose deux objets-graphiques **GROB1** et **GROB2** (ayant la même hauteur), pour en faire un objet-graphique **GROB3** (**GROB1** et **GROB2** y sont donc placés côte à côte, **GROB1** à gauche de **GROB2**).

Exercice 11 (**): Superposition de deux objets-graphiques.



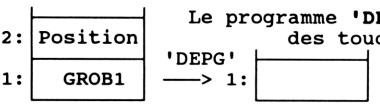
'SUPE' superpose deux objets-graphiques **GROB1** et **GROB2** (ayant la même largeur), pour en faire un objet-graphique **GROB3** (**GROB1** et **GROB2** y sont donc placés l'un au dessus de l'autre, **GROB1** au-dessus de **GROB2**).

Exercice 12 (##): Pavage obtenu à partir d'un objet-graphique.



'PAVG' crée un objet graphique **GROB2** constitué de la répétition de **GROB1** (**GROB2** est un pavage de W exemplaires de **GROB1** en largeur sur H exemplaires de **GROB1** en hauteur).

Exercice 13 (*****): Déplacements d'un objet-graphique dans **PICT**.



Le programme **'DEPG'** permet de déplacer, à l'aide des touches de curseur, l'objet-graphique GROB1 sur le contenu affiché de PICT.

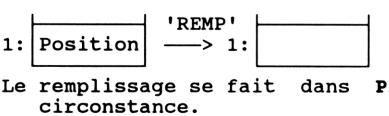
Initialement, le coin supérieur gauche de **GROB1** est placé sur le point ayant la position spécifiée au niveau 2.

Les déplacements se font avec les touches de curseur, d'un pixel à la fois dans la direction choisie, mais peuvent être accélérés en utilisant les préfixes:

- <=┐ : déplacement de 2 pixels.
- ╔=> : déplacement de 4 pixels.
- α : déplacement de 8 pixels.
- α <=┐ : déplacement de 16 pixels.
- α ╔=> : déplacement de 32 pixels.

Dans la solution, on suppose que les dimensions de **PICT** sont celles de l'écran (131x64 pixels).
On sort du programme par un appui sur une touche autre que celles mentionnées ci-dessus (mais éviter **ON**)

Exercice 14 (*****): remplissage d'un contour fermé.

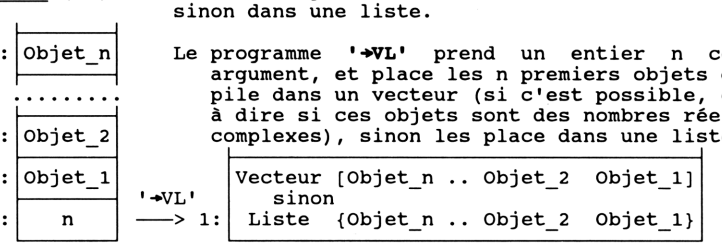


'REMP' remplit le contour fermé entourant le point A dont la position figure au niveau 1.

Le remplissage se fait dans **PICT**, qui est affiché pour la circonstance.
Le contour fermé entourant A peut être quelconque.
Si ce contour est mal fermé, il y aura "débordement".
Dans la solution, les dimensions de **PICT** sont quelconques.

LA PILE:
Corrigés des Exercices

Exercice 1 (**) : Mettre les objets des niveaux 1 à n dans un vecteur sinon dans une liste.



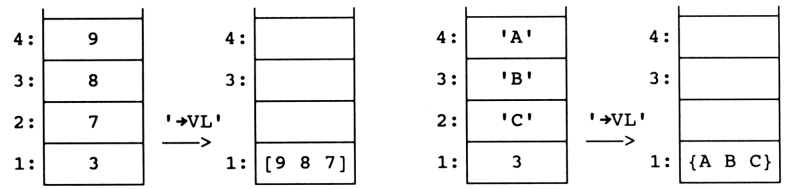
Le programme **'→VL'** prend un entier n comme argument, et place les n premiers objets de la pile dans un vecteur (si c'est possible, c'est à dire si ces objets sont des nombres réels ou complexes), sinon les place dans une liste.

On sait que l'état de la commande **LASTARG** peut être connu grâce à l'indicateur binaire N°-55.
Si cet indicateur est désarmé (si sa valeur est 0) alors **LASTARG** est active (c'est la valeur par défaut quand on réinitialise le calculateur).
Si cet indicateur est armé (c.a.d s'il vaut 1) alors **LASTARG** est inactive.

Première méthode:
Une première idée consiste à prendre l'état des indicateurs de la HP48SX (instruction **RCLF**), à désarmer l'indicateur -55, à effectuer l'opération souhaitée sur la pile, puis à replacer les indicateurs dans leur état initial (**STOF**).
(comme on l'a vu dans le chapitre consacré à la pile opérationnelle,le fait de connaître l'état de l'indicateur -55 permet d'écrire un programme résolvant le problème posé ici).

Texte du programme '→VL' (première version)	Commentaires
« RCLF → flags	'flags' représente l'état de tous les indicateurs binaires.
« -55 CF	On désarme l'indicateur -55, pour que LASTARG soit active.
IFERR →ARRY	Si on ne peut mettre les n objets dans un vecteur,
THEN →LIST END	Alors on les met dans une liste.
flags STOF	On remet les indicateurs binaires dans leur état initial.
»	
»	

Exemple:



Rappelons que dans le deuxième cas ci-dessus (celui où l'instruction **→ARRY** conduit à une erreur, le fait que **LASTARG** soit active permet de conserver l'argument 3 au niveau 1 de la pile, de manière à pouvoir effectuer **3 →LIST**.

Deuxième méthode:

Il y a une autre possibilité, un peu plus astucieuse, mais un peu moins 'lisible'.

Notons que l'instruction **-55 FC?** renvoie 1 sur la pile si l'indicateur -55 est désarmé (**LASTARG** active) et 0 sinon, et que l'instruction **n DROPN**:

* Détruit l'élément au niveau 1, si n=1.
* Laisse la pile inchangée si n=0.

Ces deux remarques montrent qu'il y a moyen de simuler le fonctionnement de **LASTARG**, même si elle est désactivée:

On met l'entier argument de **'→VL'** dans une variable locale n.

Si une erreur se produit lors de **n →ARRY**, on replace n sur la pile et on le détruit immédiatement si **LASTARG** est active (sinon on ne fait rien: c'est le sens de **-55 FC? DROPN**).

Dans tous les cas, l'instruction **→LIST** est en mesure de s'appliquer correctement.

Texte du programme '→VL' (deuxième version)	Commentaires
« → n « IFERR n →ARRY THEN n -55 FC? DROPN →LIST END » »	n : nombre d'objets à regrouper. Si on ne peut en faire un vecteur Alors: Remettre n sur la pile. L'enlever si LASTARG active. Faire une liste des n objets.

Troisième méthode:

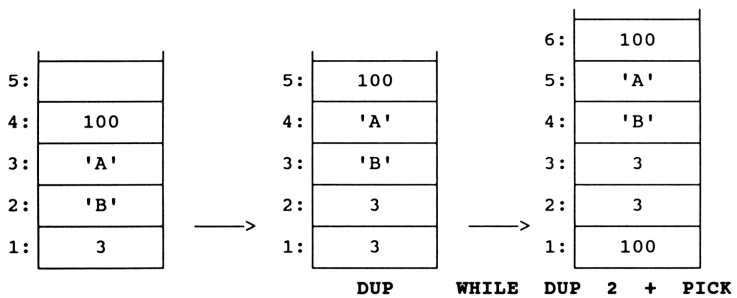
Il y a enfin une troisième méthode qui consiste à d'abord contrôler que les objets situés aux niveaux 1 à n sont des nombres réels ou complexes, (et alors on fait **n →ARRY**, sinon **n →LIST**).

Il convient d'imaginer ici un parcours des n éléments à grouper, cette lecture s'interrompant dès qu'un de ces n objets n'est pas reconnu comme nombre réel ou complexe (et alors on doit grouper les n objets dans une liste). Si cette lecture arrive à son terme, c'est qu'il est possible de mettre les n objets dans un vecteur.

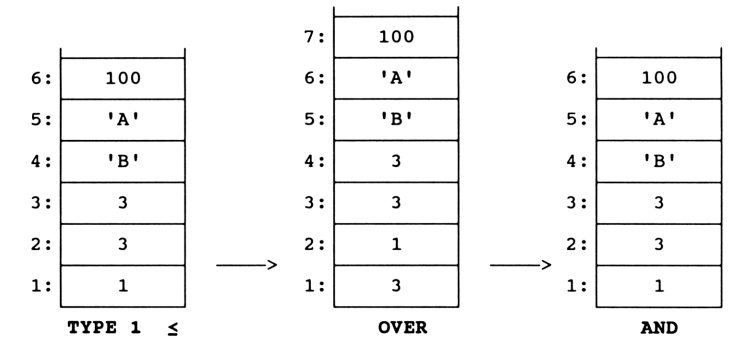
Texte du programme '→VL' (troisième version)	Commentaires
« DUP WHILE DUP 2 + PICK TYPE 1 ≤ OVER AND REPEAT 1 - END IF THEN →LIST ELSE →ARRY END »	Sauvegarder n sur la pile. Tant que L'élément lu est de type ≤ 1 (c.a.d. est réel ou cplxe) et que le compteur de lecture n'est pas descendu à zéro, répéter: on décrémente le compteur de lecture des n objets. A la sortie, si ce compteur est non nul, on fait une liste, sinon on fait un vecteur.

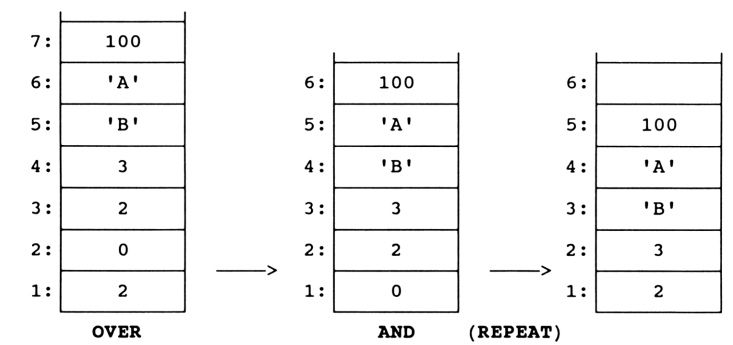
Ici, on utilise un compteur permettant de parcourir les n objets (depuis le plus haut placé, vers le plus bas: le compteur étant initialisé avec la valeur n). Ce compteur est laissé sur la pile, ce qui complique légèrement la tâche (il faut en effet tenir compte de sa présence pour calculer le niveau où se trouvent les objets à examiner).

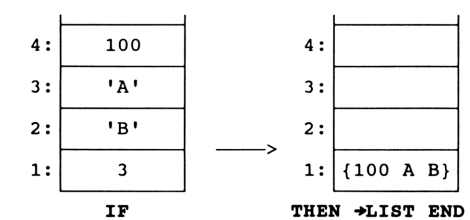
Examinons ce qui se passe sur un exemple:



LA PILE: Corrigé des Exercices.



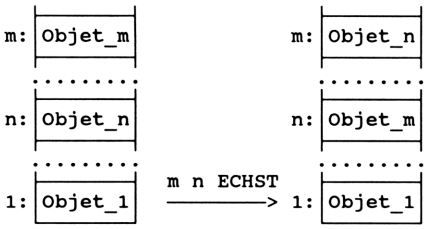




On voit sur cet exemple dans quelles conditions la boucle WHILE..REPEAT se termine:
* soit par lecture d'un objet de type ≥ 2
* soit parce que tous les objets sont réels ou complexes.

Le IF terminal teste si l'on s'est arrêté avec un compteur 0 (et alors →**ARRY**) ou avec un compteur non nul (→**LIST**).

Exercice 2: (**) Echanger deux objets sur la pile.



La séquence **m n ECHST** doit échanger les objets situés aux niveaux m et n de la pile.
L'ordre dans lequel on donne m et n est quelconque.
Si m=n, il ne doit rien se passer.
Les positions m et n sont relatives à la pile quand les paramètres nécessaires au fonctionnement de **ECHST** ont été lus.

C'est le type même de problème dont la solution connaît un grand nombre de variantes.

Pour simplifier, appelons H et B les deux objets à échanger sur la pile (H étant le plus haut placé, et B le plus bas).
En gros, il y a quatre opérations à réaliser:
* Descendre H en bas de la pile.
* Remonter H à sa nouvelle position.
* Descendre B au bas de la pile.
* Remonter B à sa nouvelle position.

Tout dépend dans quel ordre on va réaliser ces opérations.

Chacun des choix possibles (il y en a 6, car il n'est pas question de 'remonter' un objet avant de l'avoir descendu!) conduit à un programme spécifique (les variantes de l'un à l'autre sont minimes mais incontournables).

Ce n'est pas tout: Supposons que l'on commence par descendre H au niveau 1, et qu'on veuille le remonter immédiatement à la position qu'il occupera à la fin (sans s'occuper de B pour l'instant). Il y a deux positions où placer H (en pensant que B va descendre incessamment et libérer son niveau). On peut en effet remonter H:
* Juste au dessus de l'objet B (fig. 1)
* Juste en dessous de l'objet B (fig. 2)

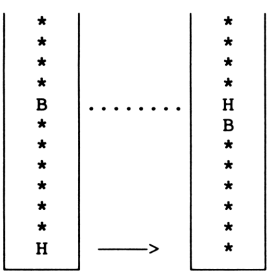


figure 1

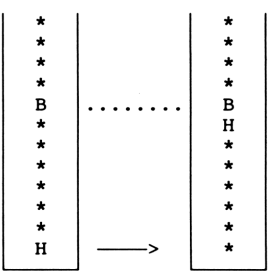


figure 2

Selon que l'on l'une ou l'autre de ces deux solutions, le niveau où on devra aller chercher B n'est pas le même.

On obtient ainsi deux variantes dans chacun des deux cas où on décide préalablement de descendre et de remonter un des deux objets avant de s'occuper de l'autre.
Au total, on peut donc imaginer 8 versions différentes de la solution.

On remarque rapidement qu'il est important, pour les deux arguments m et n représentant les niveaux des deux objets à échanger, de savoir lequel des deux est le plus petit.
En effet si on commence par faire descendre H, l'objet B monte d'un niveau (fig. 3).
Inversement, si c'est B qui descend en premier, H reste au même niveau (fig. 4).

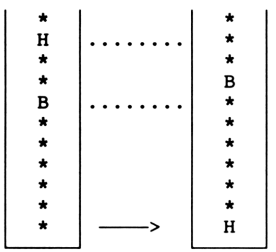


figure 3

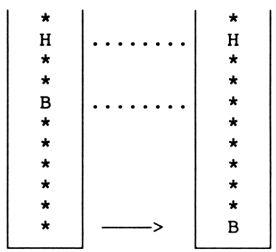


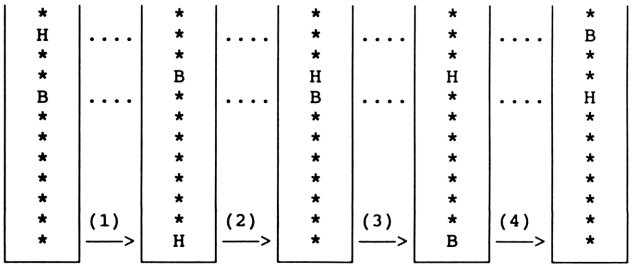
figure 4

Le programme devra donc commencer par placer les arguments dans un ordre déterminé (c'est plus logique que de demander à l'utilisateur du programme de mémoriser l'ordre dans lequel il doit donner ces deux niveaux).

Enfin on se rend compte très vite que le programme qu'on imagine facilement ne fonctionne pas si les 2 niveaux m et n donnés au départ sont égaux. Il faudra donc prévoir un test pour ne rien faire dans ce cas.

Première version:

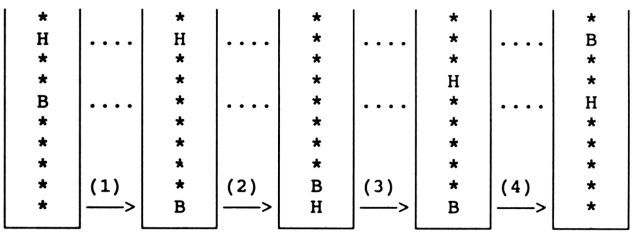
On réalise successivement les opérations suivantes:
(1): Descendre l'élément haut H, au niveau 1.
(2): Remonter H juste au dessus de B.
(3): Descendre l'élément bas B, au niveau 1.
(4): Remonter l'élément B à la place qu'occupait H au départ.



Texte du programme 'ECHST' (première version)	Commentaires
« DUP2 IF < THEN SWAP END → m n « IF m n ≠ THEN m ROLL n 1 + ROLLD n ROLL m ROLLD END » »	On duplique les deux niveaux. On 'Swappe' éventuellement , pour être sûr que le niveau minimum est en dessous de l'autre. m: niveau haut, n: niveau bas. Si ce n'est pas le même niveau: Descendre l'élément haut H, Le remonter au dessus de B, Descendre l'élément bas B, Remonter B au niveau occupé initialement par H.

Deuxième version:

On réalise successivement les opérations suivantes:
(1): Descendre l'élément bas B, au niveau 1.
(2): Descendre l'élément haut H, au niveau 1.
(3): Remonter l'élément H.
(4): Remonter B à la place qu'occupait H au départ.

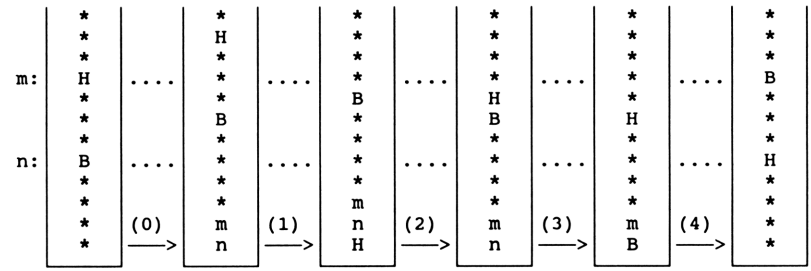


On voit que l'instruction (3) doit placer H au niveau immédiatement supérieur à celui qu'occupait B au départ, et ceci en prévision de la remontée imminente de B.

Texte du programme 'ECHST' (deuxième version)	Commentaires
« DUP2 IF < THEN SWAP END → m n « IF m n ≠ THEN n ROLL m ROLL n 1 + ROLLD m ROLLD END » »	On duplique les deux niveaux. On 'Swappe' éventuellement , pour être sûr que le niveau minimum est en dessous de l'autre. m: niveau haut, n: niveau bas. Si ce n'est pas le même niveau: Descendre l'élément bas B, Descendre l'élément haut H, Remonter H, Remonter B.

Troisième version:

Nous allons reprendre la première version du programme **ECHST** mais en nous contraignant à ne pas créer les variables locales m et n.
Celà a pour conséquence de nous obliger à conserver ces valeurs sur la pile, et leur présence modifie le niveau où se trouvent les objets à échanger!
Dans une telle situation on ne s'en sort qu'en dessinant à la main les différents états de la pile opérationnelle, pour savoir où trouver qui, et quand.



Dans le diagramme ci-dessus, on a fait figurer la présence des indices m (niveau haut) et n (niveau bas), le temps que ces indices sont utiles sur la pile.

Le passage désigné par (0) correspond à la dépose de ces deux indices sur la pile (suivie de l'instruction consistant à les 'swapper' pour être certain que m>n).

La simple consultation de ce diagramme doit vous faire comprendre facilement le programme ci-dessous (et la création d'un tel diagramme est un élément nécessaire de l'écriture rapide et sûre de ce programme).

Texte du programme 'ECHST' (troisième version)	Commentaires

« DUP2
IF < THEN SWAP END
IF DUP2 ≠ THEN
OVER 2 + ROLL
OVER 3 + ROLLD
1 + ROLL
SWAP ROLLD
ELSE
DROP2
END
»
	On est sûr que le niveau haut se trouve à l'étage 1 de la pile.
	Si les 2 niveaux sont distincts:
	On descend l'élément H.
	On le remonte au-dessus de B.
	On descend l'élément B.
	On le remonte à la place qu'occupait H au départ.
	Sinon, on élimine ces 2 niveaux égaux.

Quatrième version:

Pour en terminer avec les multiples possiblités d'écrire **'ECHST'**, nous allons voir qu'il est possible de travailler avec une liste contenant la partie utile de la pile.
L'intérêt de cette méthode est qu'elle ne nous oblige pas à réfléchir intensément sur la position qu'occupent sur la pile les éléments à échanger, à tel ou tel moment de l'éxécution du programme.

On procède de la manière suivante:
* On place dans une liste les objets de la pile, du niveau 1 au niveau le plus haut des deux éléments à échanger.
* On lit dans cette liste les deux objets à échanger.
* On écrit dans cette liste chacun de ces deux objets à la place qu'occupait l'autre.
* On libère cette liste sur la pile.

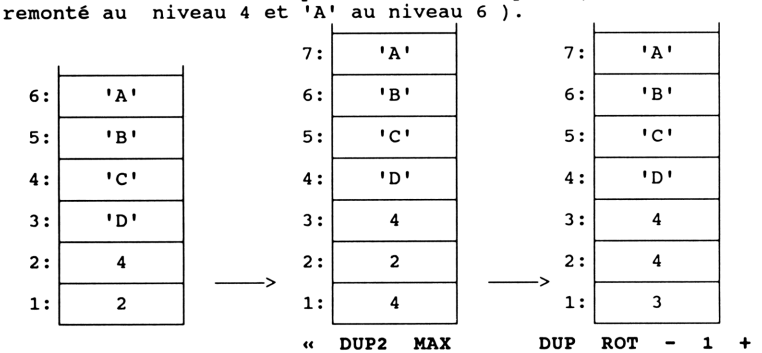
Cette méthode est plus éloignée de la gestion de la pile que les précédentes (encore que le texte contienne de nombreuses instructions **DUP2,DUP,ROT,ROLL,OVER,DROP**). Elle est un peu plus lente (le fait de former la liste, puis de la défaire).

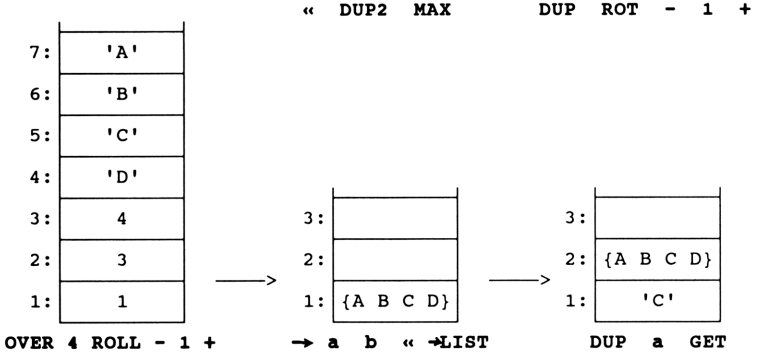
Elle a quelques avantages, notamment le fait qu'on n'a pas à se préoccuper de la comparaison préalable des deux niveaux (ils peuvent même être égaux).

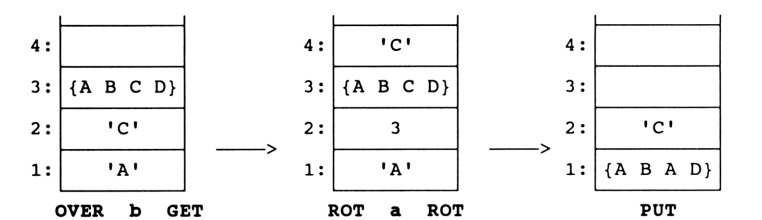
Il y a cependant une petite difficulté à régler:
Si on désigne par m et n les niveaux des objets à échanger, et si on suppose par exemple que m>n, alors on crée la liste L des m premiers éléments de la pile.
La position x d'un objet dans la liste n'est cependant pas égale à la valeur y du niveau qu'il occupait sur la pile.
Précisément, on a ici : x = m+1-y.
(par exemple l'objet qui était au niveau m devient le premier élément de la liste).
Il faut donc remplacer (pour les besoins des opérations de lecture et d'écriture dans la liste), les niveaux m et n par les positions correspondantes a et b dans la liste.

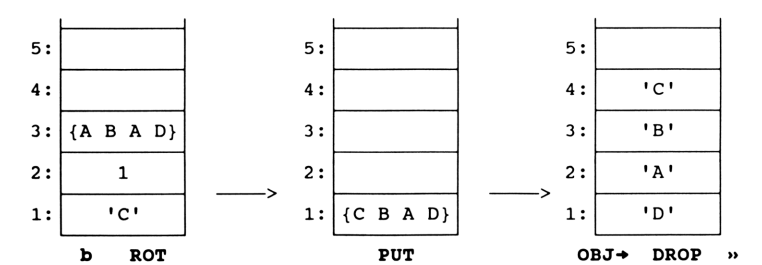
Texte du programme 'ECHST' (quatrième version)	Commentaires
`« DUP2 MAX`	Maximum des deux niveaux.
`DUP ROT - 1 +`	On calcule la position qu'auront
`OVER 4 ROLL - 1 +`	les deux objets dans la liste.
`→ a b`	On appelle a,b ces 2 positions.
`« →LIST`	Forme la liste utile.
`DUP a GET`	Lit l'objet A.
`OVER b GET`	Lit l'objet B.
`ROT a ROT PUT`	Place B à la place qu'occupait A
`b ROT PUT`	Place A à la place qu'occupait B
`OBJ→ DROP`	Libère la liste et en détruit le
`»`	nombre d'éléments.
`»`	

Voyons comment fonctionne cette dernière version de **'ECHST'**:
On veut échanger les objets 'C' et 'A' aux niveaux 2 et 4.
La 1ère représentation ci-dessous nous place au moment où les entiers 4 et 2 ont été placés sur la pile ('C' est donc remonté au niveau 4 et 'A' au niveau 6).









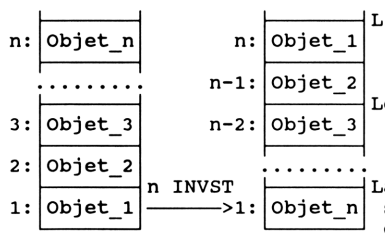
Exercice 3: (*) Permuter aléatoirement les objets sur la pile.

Il s'agit ici de modifier, d'une façon aléatoire, les positions respectives des différents objets de la pile, et ce jusqu'à ce qu'une touche quelconque soit appuyée.

La solution est très simple:
* Une instruction **DO...UNTIL KEY END** attend qu'une touche soit pressée.
* Entre **DO** et **UNTIL**, on calcule un niveau aléatoire occupé,par **DEPTH RAND * CEIL** (**CEIL** pour obtenir un niveau allant de 1 à la hauteur de la pile), et on appelle l'instruction **ROLL** pour descendre au niveau 1 l'élément du niveau calculé (ce qui a pour effet de remonter les autres d'un étage).

Texte du programme 'RNDST'	Commentaires
« DO DEPTH RAND * CEIL ROLL UNTIL KEY END DROP »	Répéter: Choisir un niveau aléatoire. Descendre l'objet du niveau. Jusqu'à ce qu'une touche soit appuyée (et on élimine l'écho de cette touche).

Exercice 4: (**) Inverser l'ordre des n premiers éléments de la pile.



L'instruction **n INVST** doit inverser les positions respectives des n premiers éléments de la pile.

Le programme s'appelle **INVST**, et il prend un entier n comme unique argument.

La présence de n n'est pas indiquée sur ce schéma fonctionnel mais elle est effective au moment de l'appel de **INVST**.

Première solution:

On fait successivement descendre au niveau 1:
* l'objet situé au niveau 2,
* l'objet situé au niveau 3,
etc...
* l'objet situé au niveau n.

Il faut tout de même prendre certaines précautions:
* Si n est ≤ 1, la pile doit rester inchangée.
* Si n est supérieur à la hauteur de la pile, il faut se limiter à cette hauteur.

Si on veut donc contrôler ces situations (afin qu'elles n'interrompent pas le programme sur une erreur),il faut en tenir compte dans la boucle qui s'occupe de descendre les éléments utiles de la pile.

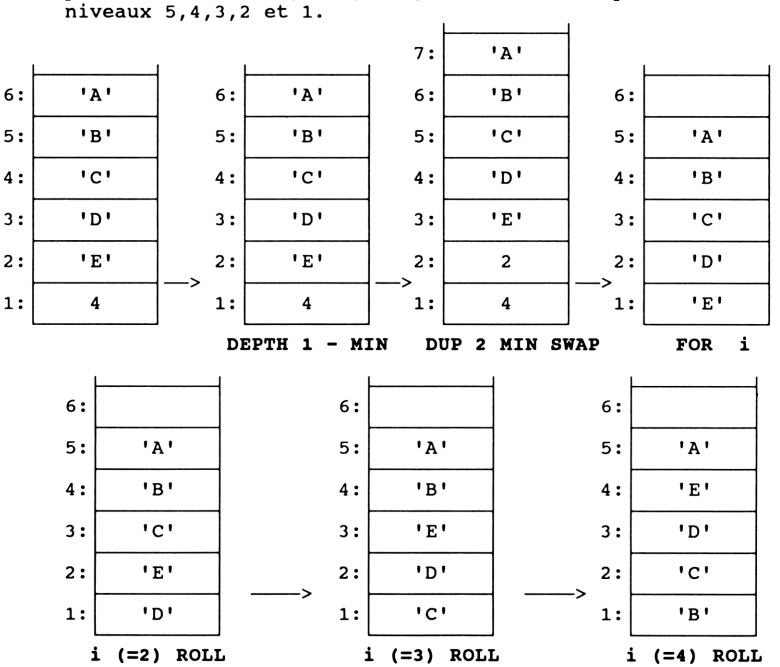
LA PILE: Corrigé des Exercices.

Texte du programme 'INVST' (première version)	Commentaires
« DEPTH 1 - MIN DUP 2 MIN SWAP FOR i i ROLL NEXT »	Indice maximum i de la boucle. Indice minimum i de la boucle. Min au dessus de max. Pour les indices i croissants: Descendre, au niveau 1, l'objet situé au niveau i.

Dans le texte ci-dessus, la séquence **DEPTH 1 -** calcule la profondeur réelle de la pile (on enlève 1 pour ne pas tenir compte de l'entier n situé au niveau 1).

De la manière dont le texte est écrit, il ne peut pas se produire d'erreur d'éxécution (notamment pour les valeurs extrêmes de n ou de la profondeur de pile).

Exemple:
On veut inverser l'ordre des quatre premiers éléments d'une pile contenant 'A', 'B', 'C', 'D' et 'E' respectivement aux niveaux 5,4,3,2 et 1.



Deuxième solution:

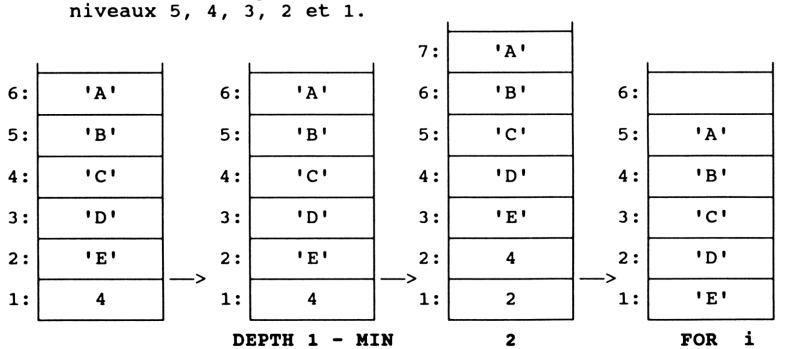
La première méthode consistait à descendre un à un les objets concernés. Ici, au contraire, on va les remonter.
On effectue successivement les opérations suivantes:
* Remonter l'objet du niveau 1, vers le niveau n.
* Remonter l'objet du niveau 1 (et qui vient de descendre du niveau 2 par l'opération ci-dessus), au niveau n-1.
* etc.. .
* Remonter l'objet du niveau 1 (et qui est, petit à petit, descendu du niveau n-1 au niveau 1), au niveau 2.
  Cette dernière opération a également pour effet de placer au niveau 1 l'objet qui au départ était au niveau n et qui petit à petit est descendu au niveau 2.

On comprend ici que l'instruction utilisée soit: **i ROLLD** , où i est un indice de boucle décroissant de n à 2.
Néanmoins, et comme pour la première méthode, il est utile de prévoir le cas **n > DEPTH**.

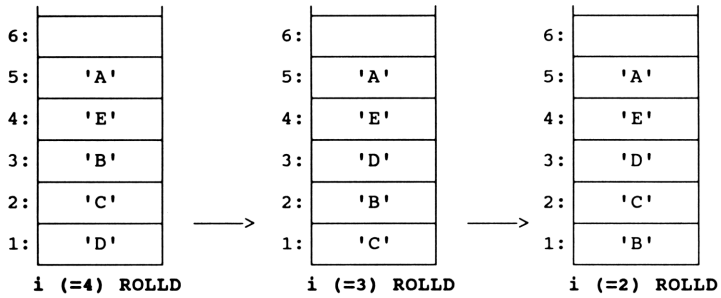
Texte du programme 'INVST' (deuxième version)	Commentaires
« DEPTH 1 - MIN 2 FOR i i ROLLD -1 STEP »	Indice maximum i de boucle. Indice miminum i de boucle. Pour les indices i décroissants: L'objet du niveau 1 remonte au niveau i.

Remarque: là encore, aucune erreur d'éxécution n'est possible, notamment si l'entier n est égal à 1, car la boucle est alors éxécutée une seule fois, avec l'indice i=1 (et l'instruction **1 ROLLD** est neutre)
De même, si la pile est vide, seule est éxécutée **0 ROLLD**, qui là aussi est une instruction inoffensive.

Exemple:
Nous reprenons l'exemple qui a servi à illustrer la première version de **'INVST'**: on veut inverser l'ordre des 4 premiers éléments d'une pile contenant 'A', 'B', 'C', 'D' et 'E' aux niveaux 5, 4, 3, 2 et 1.



LA PILE: Corrigé des Exercices.



Troisième solution:

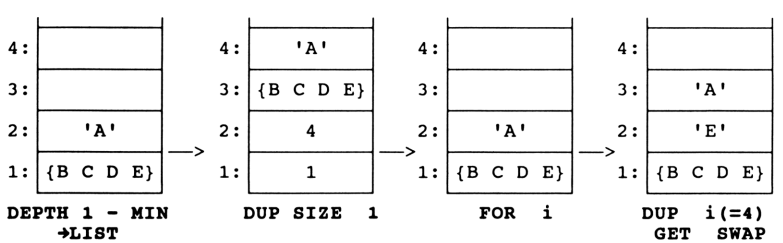
Il existe une autre possiblité, utilisant une liste.

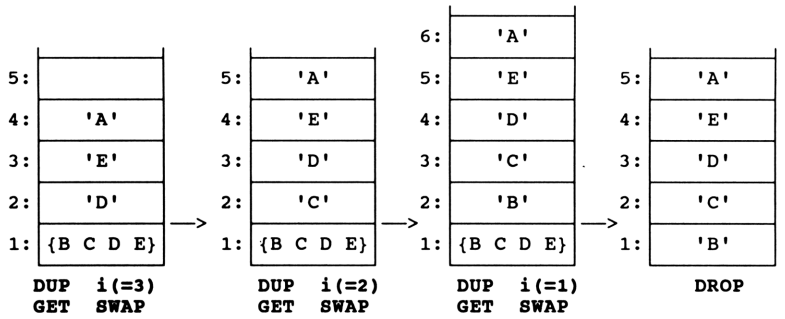
* On place les n premiers éléments de la pile dans une liste. Dans cette liste l'ordre des éléments est l'ordre inverse des niveaux de la pile: par exemple, l'objet qui était au niveau n se retrouve en première position dans la liste.
* Il suffit alors en gardant cette liste sur la pile d'en lire successivement tous les éléments, du dernier au premier.

Texte du programme 'INVST' (troisième version)	Commentaires
« DEPTH 1 - MIN →LIST DUP SIZE IF DUP THEN 1 FOR i DUP i GET SWAP -1 STEP ELSE DROP END DROP »	Nombre d'objets à considérer. On les met dans une liste dont on calcule la longueur n. Si cette longueur est > 0, alors de i=n à 1 en décroissant: on lit le i-ème élément. on remet la liste au niveau 1. Sinon on détruit la longueur = 0 Détruire la liste.

Exemple:

Nous reprenons l'exemple qui a servi à illustrer la première version de **'INVST'**: on veut inverser l'ordre des 4 premiers éléments d'une pile contenant 'A', 'B', 'C', 'D' et 'E' aux niveaux 5, 4, 3, 2 et 1.





Exercice 5: (**) Eliminer les objets de la pile à partir du niveau n.

Il s'agit ici de détruire les objets de la pile, depuis celui qui est situé au niveau n, jusqu'à celui situé au sommet de cette pile.
Si n est supérieur au nombre d'objets présents sur la pile, celle-ci doit rester inchangée.

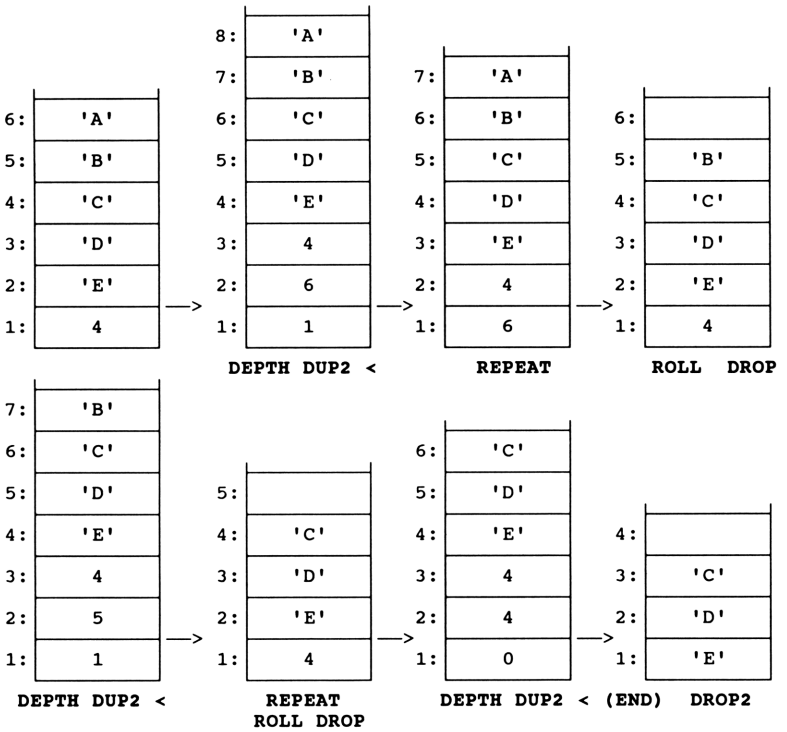
On va appeler **DELN** le programme réalisant l'élimination des objets de la pile à partir du niveau n.

Première méthode:
On fait descendre au niveau 1 l'élément le plus haut de la pile, et on le détruit, jusqu'à ce que la hauteur de celle pile soit inférieure à n.
Pour que la syntaxe **n DELN** supporte un argument réel (comme **n ROLL, n ROLLD, n PICK** etc ...), il convient d'arrondir l'argument n à l'entier le plus proche.
Il faut faire en sorte également que la pile soit entièrement effacé si n ≤ 1 (et en particulier si n<0, il ne doit pas y avoir d'erreur d'éxécution).
Pour cette raison, la solution suivante utilise une instruction **WHILE...REPEAT** (qui a l'avantage de faire un test de validité avant de détruire un élément de la pile).

Texte du programme 'DELN' (première version)	Commentaires
« .5 + FLOOR WHILE DEPTH DUP2 < REPEAT ROLL DROP END DROP2 »	On arrondit n à l'entier proche. Tant que la hauteur de pile est ≥ n, Répéter: Descendre l'élément le plus haut et le détruire. Fin. Effacer n et hauteur de pile.

LA PILE: Corrigé des Exercices.

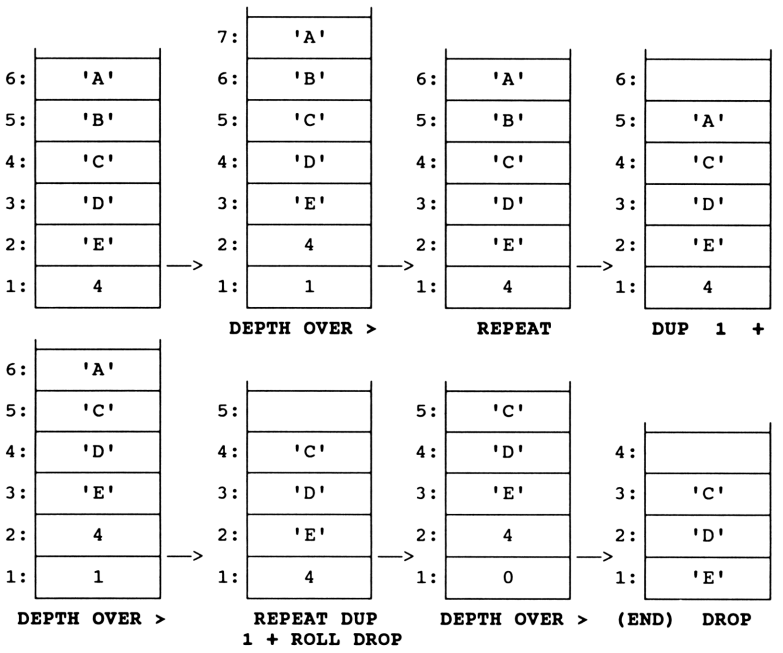
Exemple: Supposons que la pile contienne 'A', 'B', 'C', 'D', 'E' aux niveaux respectifs, 5,4,3,2,1, et qu'on veuille effacer la pile à partir du niveau 4 (donc effacer 'A',et 'B').
Nous nous plaçons après la séquence **.5 + FLOOR**.



Deuxième méthode:
Au lieu de faire descendre les éléments situés au sommet de la pile, on va faire descendre les éléments situés au niveau n (ceux situés au dessus descendent d'un étage à chaque fois).

Texte du programme 'DELN' (deuxième version)	Commentaires
« .5 + FLOOR WHILE DEPTH OVER >= REPEAT DUP 1 + ROLL DROP END DROP »	Arrondir n à l'entier proche. Tant que: la hauteur de pile est ≥ n, Répéter: Hauteur de l'objet à descendre Le descendre, le détruire. Fin. Effacer l'entier n.

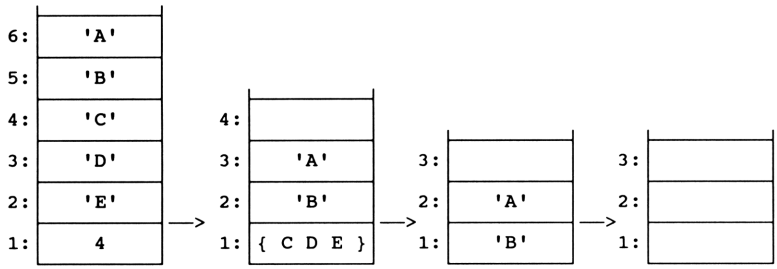
Exemple: on reprend l'exemple précédent.
La pile contient 'A','B','C','D','E' aux niveaux 5,4,3,2,1, et on veut effacer la pile à partir du niveau 4.
Nous nous plaçons après la séquence **.5 + FLOOR.**

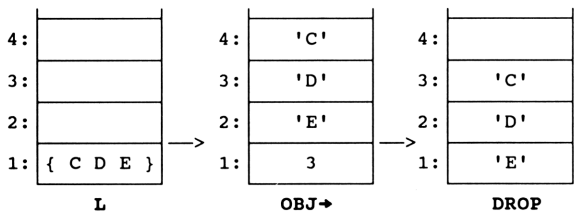


Troisième méthode:
Cette méthode, très différente des deux premières, utilise une liste:
* On place les n-1 premiers éléments de la pile (ceux qu'ils ne faut pas détruire) dans une liste dont on fait une variable locale.
* On nettoie ce qui subsiste de la pile, par **CLEAR**.
* On rappelle la liste et on la libère sur la pile.

Texte du programme 'DELN' (troisième version)	Commentaires
« DEPTH MIN 1 - →LIST → L « CLEAR L OBJ→ DROP » »	Nombre d'éléments à protéger. On en fait une liste, que l'on met dans la variable L. On nettoie ce qui reste. On libère L sur la pile, et on efface sa longueur.

Exemple: on reprend l'exemple précédent.
La pile contient 'A','B','C','D','E' aux niveaux 5,4,3,2,1, et on veut effacer la pile à partir du niveau 4.





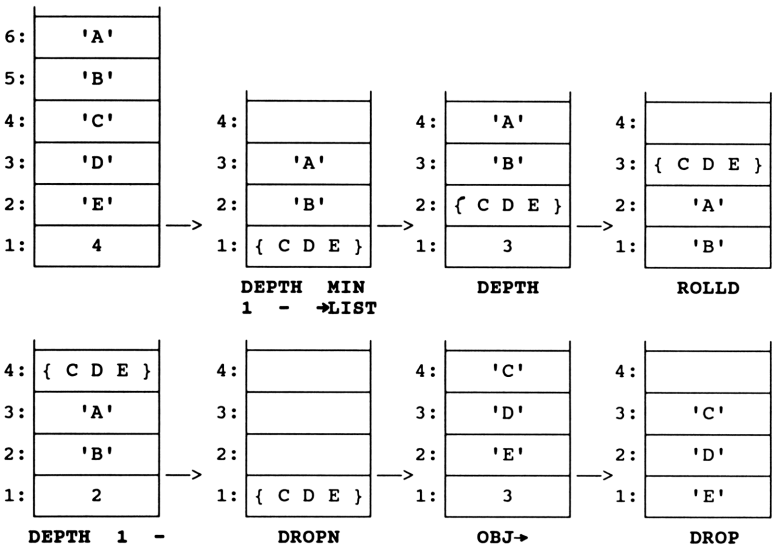
Remarque:
Dans cette troisième version, aucune erreur d'éxécution n'est possible, et il n'est pas utile d'arrondir l'argument n à l'entier le plus proche.

Quatrième méthode:
On peut modifier la troisième version de **'DELN'** si on s'impose de ne pas placer la liste des éléments à conserver dans une variable. Il faut donc conserver cette liste sur la pile, mais l'utilisation de **CLEAR** est alors impossible.

Texte du programme 'DELN' (quatrième version)	Commentaires
« DEPTH MIN 1 -	Nombre d'éléments à protéger.
→LIST	On en fait une liste, que l'on
DEPTH ROLLD	place au sommet de la pile.
DEPTH 1 - DROPN	On détruit toute la pile, sauf
	la liste au sommet!
OBJ→ DROP	On libère cette liste et on en
»	efface la longueur.

Là encore, aucune erreur n'est possible (en particulier parce que les instructions **→LIST, ROLLD, DROPN**, prennent un argument qui s'il est non entier, est arrondi à l'entier ≥0 le plus proche).

Exemple: reprenons une dernière fois l'exemple qui nous a servi à illustrer les différentes versions de **'DELN'**.
La pile contient 'A','B','C','D','E' aux niveaux 5,4,3,2,1, et on veut effacer la pile à partir du niveau 4.



Exercice 6: (**) Descendre les objets situés aux niveaux pairs, dans la limite des n premiers objets.

Il s'agit ici de considérer les n premiers éléments de la pile, et parmi eux de descendre les objets situés (dans l'état initial de la pile) aux niveaux pairs.
Si on note (P) et (I) les ensembles formés par les objets situés initialement aux niveaux pairs (resp. impairs), dans la limite des n premiers niveaux, les objets de (P) doivent à la fin se trouver tous en dessous de tous les objets de (I).
Cependant pour les objets de (P), la hiérarchie des niveaux doit être conservée (idem pour (I)).

Il s'agit ici d'un exercice sans grande application pratique, mais dont la solution demande un minimum de réflexion.
On peut hésiter entre deux alternatives:
* Remonter les objets des niveaux impairs.
* Descendre les objets des niveaux pairs.

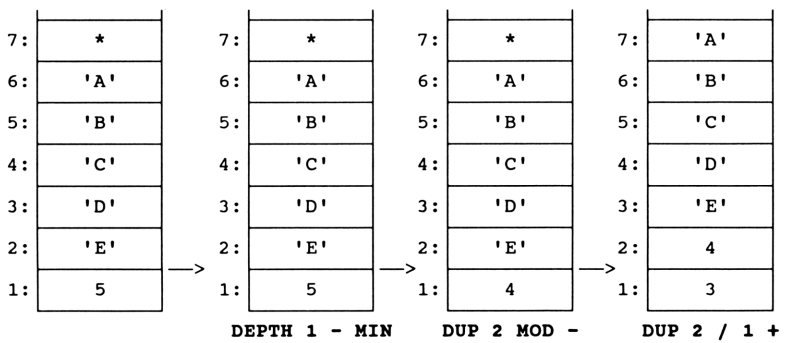
La première idée est plutôt à écarter, car pour remonter un élément dans la pile, il faut déjà le faire descendre au niveau 1. Autant donc prendre la deuxième possiblité.
Celle-ci consistera à descendre un à un les éléments situés initialement à des niveaux pairs. Deux remarques s'imposent:
* Commencer par descendre les éléments les plus hauts placés (pour conserver la hiérarchie initiale des niveaux).
* Considérer que le fait de descendre un élément au niveau 1, remonte d'un étage ceux qui auparavant étaient situés en dessous de lui.

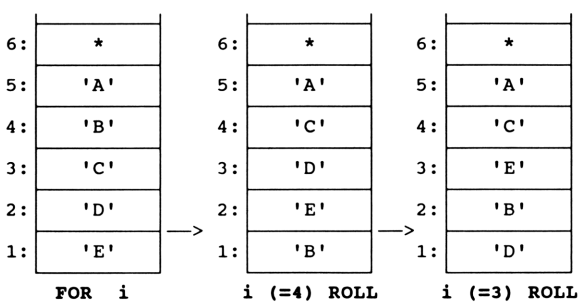
Appelons **PBIH** (les objets pairs en bas, les objets impairs en haut) le programme. Ce programme attend l'entier n au niveau 1 de la pile.
La solution suivante s'inspire des remarques ci-dessus.

Texte du programme 'PBIH'	Commentaires
« DEPTH 1 - MIN DUP 2 MOD - DUP 2 / 1 + FOR i i ROLL -1 STEP »	Si n>DEPTH, prendre n=DEPTH. Indice i maximum. Indice i minimum. Pour les indices i décroissants: Descendre l'objet du niveau i. Au suivant...

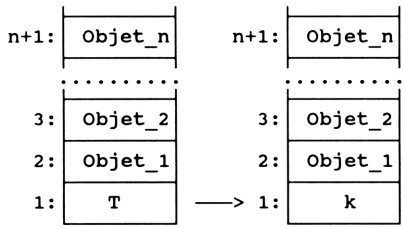
Exemple:

La pile contient 'A','B','C','D','E' aux niveaux 5, 4, 3, 2, 1 (plus d'éventuels objets au-dessus) et on veut appliquer ce qui précède aux objets de niveau ≤ 5.
Ci dessous , le premier diagramme se situe au moment où l'argument 5 a été placé sur la pile (les objets de la pile sont donc montés d'un niveau. Considérez leur position avant l'arrivée de 5).





Exercice 7: (***) Compter combien il y a, sur la pile, d'objets d'un type déterminé.

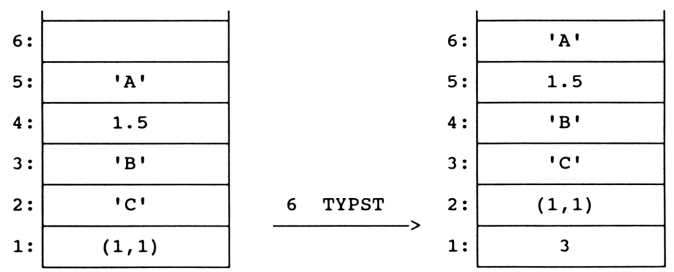


Le programme **TYPST** doit calculer le nombre d'objets qui sont de type T sur la pile.
T est un entier compris entre 0 et 9 conformément à la description des types sur la HP48SX.
Le résultat est un entier k.
Le contenu de la pile (sauf le T initial) reste inchangé.

Ici on va utiliser un compteur sur la pile (le futur résultat k), initialisé à 0, et auquel on va ajouter 1 à chaque fois qu'un objet de type T sera repéré sur la pile.
L'instruction qui permet d'étudier ces objets est évidemment **PICK** (l'objet doit rester à sa place dans la pile, seule une copie doit descendre au niveau 1).
Dans la solution suivante, on ne crée pas de variable locale: le compteur et la valeur T restent sur la pile. Il faut tenir compte de leur présence dans les niveaux à examiner.

Texte du programme 'TYPST'	Commentaires
«	
0	Compteur k initialisé à 0.
3	Niveau minimum à considérer.
DEPTH 1 -	Niveau maximum à considérer.
FOR i	Pour les indices i croissants:
i PICK	Copier l'objet du niveau i.
TYPE	Prendre son type.
3 PICK	Type T copié au niveau 1.
==	1 s'ils sont égaux, 0 sinon.
+	On ajoute ce résultat à k.
NEXT	Indice suivant...
SWAP DROP	Effacer le type T.
»	

Exemple: La pile contient 'A', 1.5, 'B', 'C', (1,1) aux niveaux 5,4,3,2,1 et on veut savoir combien elle contient de noms, c'est-à-dire d'objets de type 6.



Exercice 8: (***) Descendre un bloc au niveau 1 de la pile.

Le programme **'ROLLB'** (**ROLL** sur un bloc) doit ramener au niveau 1 le bloc délimité par le niveau minimum 'min' et le niveau maximum 'max'.
La syntaxe doit être: **min max ROLLB** (d'abord le niveau minimum, puis le niveau maximum).

On fait descendre l'objet situé au niveau max, puis l'objet situé au niveau max-1, etc... pour terminer par l'objet situé au niveau min: Ici les niveaux correspondent à la pile initiale.

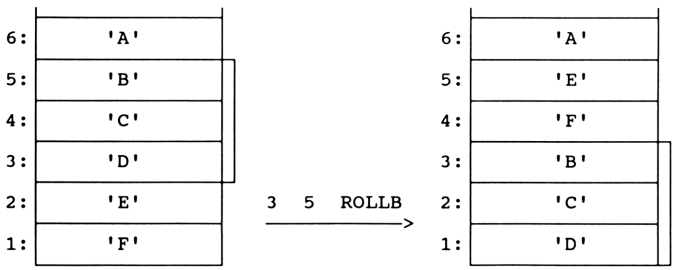
Ces descentes d'éléments modifient constamment ces niveaux initiaux et on se rend compte que le niveau où il faut aller chercher tel ou tel élément est en fait toujours le même.

Un parcours par niveaux décroissants s'impose pour conserver aux éléments du bloc leurs positions respectives d'origine.

Texte du programme 'ROLLB'	Commentaires
« DEPTH 2 - MIN	Calcul du niveau maximum 'max'.
SWAP	'max' au dessus de 'min'.
1 MAX	Calcul du niveau 'min'.
WHILE	
DUP2 ≥	Tant que 'max' est ≥ 'min',
REPEAT	Répéter:
OVER 2 + ROLL	Descendre un objet du bloc.
3 ROLLD	Le placer au-dessus de min et
1 +	max et ajouter 1 à min.
END	
DROP2	Effacer max et min.
»	

Remarque: compte tenu des précautions prises au début du programme, aucune erreur d'éxécution n'est possible.

Exemple: Attention à bien donner le niveau minimum du bloc avant le niveau maximum (sinon le bloc est considéré comme vide et il ne se passe rien).



Exercice 9: (***) Remonter un bloc dans la pile.

Le programme **'ROLDB'** (**ROLLD** sur un bloc) doit remonter au niveau n de la pile le bloc allant du niveau 1 au niveau h (h=hauteur du bloc).
La syntaxe doit être: **h n ROLDB** (d'abord la hauteur du bloc, puis le niveau où le monter dans la pile).

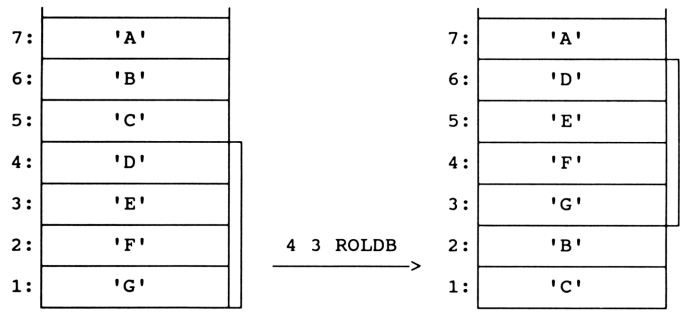
Une première idée est de monter l'objet du niveau 1 vers le niveau n. L'objet initialement au niveau 2, maintenant au niveau 1, serait alors remonté au niveau n+1, etc...

On se rend compte cependant que cette méthode ne marche que si le niveau n de destination est supérieur à la hauteur h du bloc. Dans le cas contraire, c'est-à-dire si la position actuelle du bloc et sa position finale se chevauchent, il y a des erreurs.

Le plus simple est en fait de travailler sur des listes:
* On met le bloc dans une liste que l'on monte au niveau n.
* On met les n-1 premiers éléments de la pile dans une autre liste.
* On additionne ces deux listes.
* On libère sur la pile la liste obtenue.

Texte du programme 'ROLDB'	Commentaires
« → n « →LIST n ROLLD n 1 - →LIST + OBJ→ DROP » »	n: niveau où porter le bloc. Le bloc dans une liste. Monte cette liste au niveau n. Les n-1 premiers éléments dans une liste, qu'on ajoute à la première, et qu'on libère. On efface le nombre d'éléments.

Exemple: Monter un bloc de hauteur 4, à la hauteur 3.



Remarque sur les exercices 8 et 9:
On constate facilement que monter le bloc de hauteur h vers le niveau n (par **ROLDB**) c'est exactement descendre au niveau 1 (par **ROLLB**) le bloc constitué par les éléments dont le niveau va de h+1 à h+n-1.
Autrement dit:
h n ROLDB peut être remplacé par: **(h+1) (n+h-1) ROLLB.**
De même:
m n ROLLB peut être remplacé par: **(m-1) (n-m+2) ROLDB.**

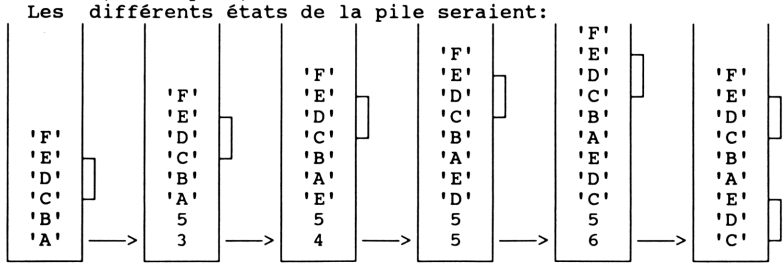
Exercice 10: (**) Copier un bloc au niveau 1 de la pile.
Le programme **PICKB** (**PICK** sur un bloc) doit recopier au niveau 1 le bloc délimité par le niveau minimum 'min' et le niveau maximum 'max'.
La syntaxe doit être: **min max PICKB** (d'abord le niveau minimum, puis le niveau maximum).
Le bloc initial doit rester sur la pile.

Nous allons voir trois solutions de cet exercice.

Première solution:
On se rend compte que le texte du programme **'ROLLB'** peut être repris tel quel, à la seule différence que l'on doit y remplacer l'instruction **ROLL** par l'instruction **PICK**.

Texte du programme 'PICKB' (première version)	Commentaires
« DEPTH 2 - MIN SWAP 1 MAX WHILE DUP2 ≥ REPEAT OVER 2 + PICK 3 ROLLD 1 + END DROP2 »	Calcul du niveau maximum 'max'. 'max' au dessus de 'min'. Calcul du niveau 'min'. Tant que 'max' est ≥ 'min', Répéter: Copier un objet du bloc. Le placer au-dessus de min et max et ajouter 1 à min. Effacer max et min.

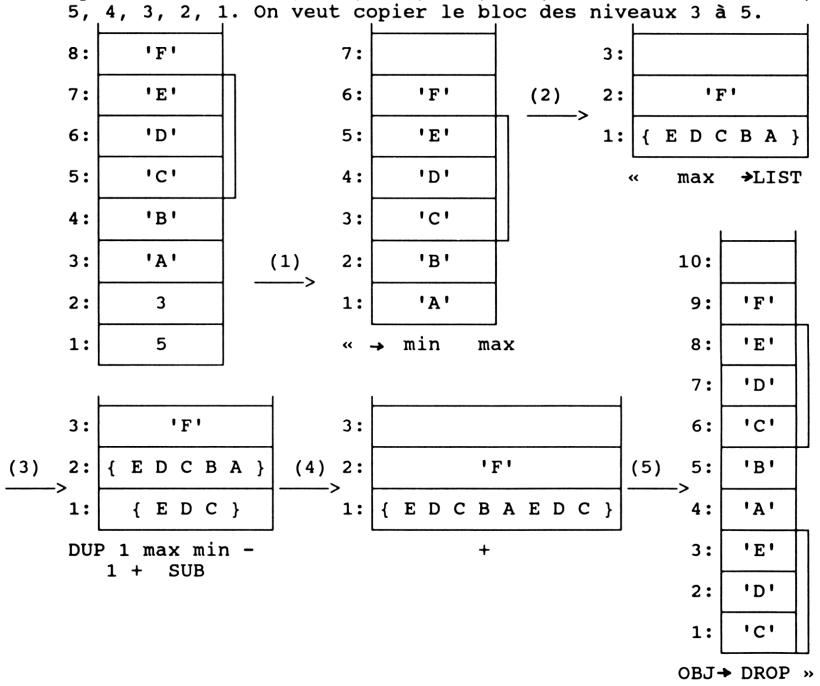
Exemple: supposons que l'on veuille copier au niveau 1 le bloc allant, au départ, du niveau 3 au niveau 5.
Les différents états de la pile seraient:



Deuxième solution:
Ici nous allons travailler sur des listes.
Pour effectuer : **min max PICKB** on fait successivement:
* On place les objets des niveaux 1 à max dans une liste L.
* Tout en conservant L sur la pile, on en extrait la liste L' des (max-min+1) premiers éléments (les éléments de L' ne sont autres que les éléments du bloc à copier).
* On ajoute L' à la suite de L
* On libère sur la pile la liste obtenue.

Texte du programme 'PICKB' (deuxième version)	Commentaires
« → min max « max →LIST DUP 1 max min - 1 + SUB + OBJ→ DROP » »	Deux variables locales min, max. Les max premiers éléments dans une liste qu'on duplique. On en extrait la liste des élts du bloc. On ajoute les 2 listes et on libère le résultat sur la pile (DROP pour effacer la lon-gueur de cette liste).

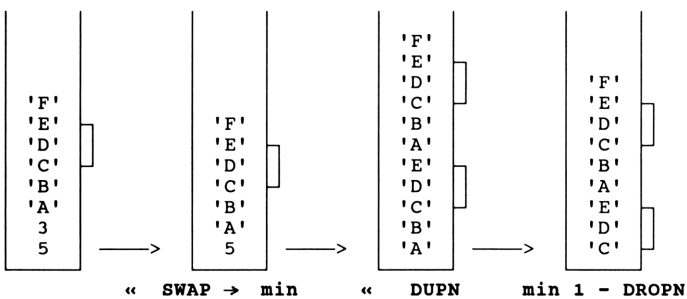
Exemple: reprenons l'exemple qui nous a servi à illustrer la première version de **'PICKB'**.
La pile contient 'F', 'E', 'D', 'C', 'B', 'A' aux niveaux 6, 5, 4, 3, 2, 1. On veut copier le bloc des niveaux 3 à 5.



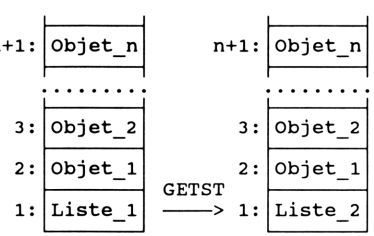
Troisième solution:
Cette dernière solution de **PICKB** sera en fait la plus simple.
On veut donc effectuer: **min max PICKB**. Pour celà:
* On duplique les max premiers éléments de la pile (**DUPN**).
* On détruit les min-1 premiers éléments de la pile (**DROPN**)

Texte du programme 'PICKB' (troisième version)	Commentaires
« SWAP → min « DUPN min 1 - DROPN » »	'min' dans une variable locale. On duplique les max premiers éléments de la pile. On détruit les (min-1) premiers (qui ont été dupliqués à tort)

Exemple: reprenons l'exemple qui nous a servi à illustrer les 2 premières versions de **'PICKB'**.
La pile contient 'F', 'E', 'D', 'C', 'B', 'A' aux niveaux 6, 5, 4, 3, 2, 1. On veut copier le bloc des niveaux 3 à 5.



Exercice 11: (**) Extraire une liste d'objets de la pile.



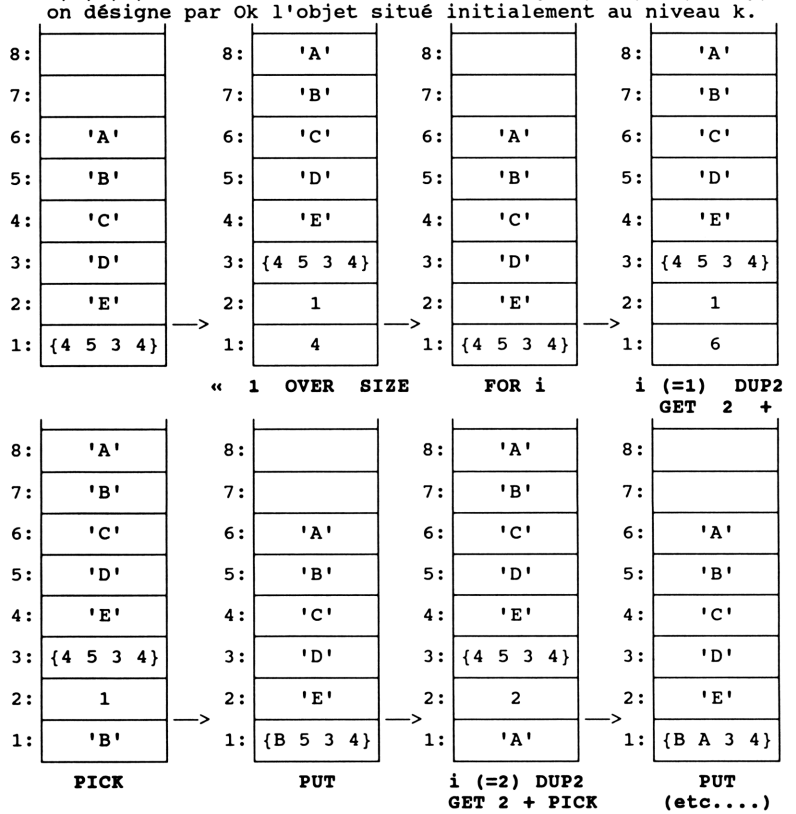
Le programme **GETST** prend une liste d'entiers Liste_1 = {a b ... z}.
Il crée alors la Liste:
Liste_2 = {Oa, Ob, ... Oz}
où Oa, Ob, ... Oz, désignent les objets qui au départ se trouvent aux niveaux a,b,...,z.
Les niveaux a,b,...,z peuvent être donnés dans un ordre quelconque.
Un même niveau peut apparaître plusieurs fois.

Remarque: Dans le calcul du niveau sur la pile, la présence de Liste_1 dans le diagramme ci-dessus ne doit pas être prise en considération.

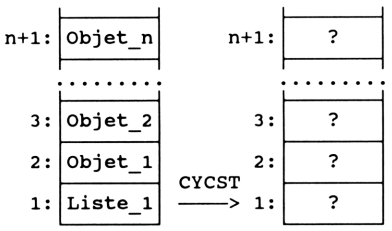
La solution est plutôt simple: on laisse Liste_1 au niveau 1, et petit à petit, on remplace chacun de ses éléments (qui désigne un niveau n de la pile), par l'objet trouvé à ce niveau).

Texte du programme 'GETST'	Commentaires
« 1 OVER SIZE FOR i i DUP2 GET 2 + PICK PUT NEXT »	De 1 à la longueur de la liste: Tout en laissant la liste et i sur la pile, on lit le i-ème élément de la liste. Niveau où trouver l'objet. On le copie au niveau 1. On le place dans la liste. Au suivant...

Exemple: La pile contient 'A', 'B', 'C', 'D', 'E' aux niveaux 5,4,3,2,1 et on veut obtenir la liste { O4, O5, O3, O4 }, où on désigne par Ok l'objet situé initialement au niveau k.



Exercice 12: (###) Permutation circulaire quelconque sur certains objets de la pile.



Le programme **CYCST** prend une liste d'entiers Liste_1 = {a b ... z}. Il effectue alors une permutation circulaire sur les objets:
Oa, Ob, ... Oz,
qui se trouvent au départ aux niveaux a,b,...,z.
Ainsi Oa remplace Ob, qui est lui même envoyé là où se trouvait Oc etc..,et pour finir Oz est placé là où se trouvait Oa.

N.B: Il faut veiller à ce que les entiers a,b,..z de Liste_1 soient distincts deux à deux, et correspondent bien à des niveaux occupés sur la pile.

Ici, on s'en sort facilement si on sait qu'une permutation circulaire peut s'écrire comme une succession d'échanges de deux éléments.

Prenons un exemple:

Soit la liste Liste_1 = { 4 1 3 2 5 }.

Ainsi on veut:
* Placer au niveau 1 l'objet initialement au niveau 4.
* Placer au niveau 3 l'objet initialement au niveau 1.
* Placer au niveau 2 l'objet initialement au niveau 3.
* Placer au niveau 5 l'objet initialement au niveau 2.
* Placer au niveau 4 l'objet initialement au niveau 5.

Cette permutation circulaire peut être obtenu en effectuant successivement, et dans cet ordre:
* L'échange de l'objet du niveau 2, avec celui du niveau 5.
* L'échange de l'objet du niveau 3, avec celui du niveau 2.
* L'échange de l'objet du niveau 1, avec celui du niveau 3.
* L'échange de l'objet du niveau 4, avec celui du niveau 1.

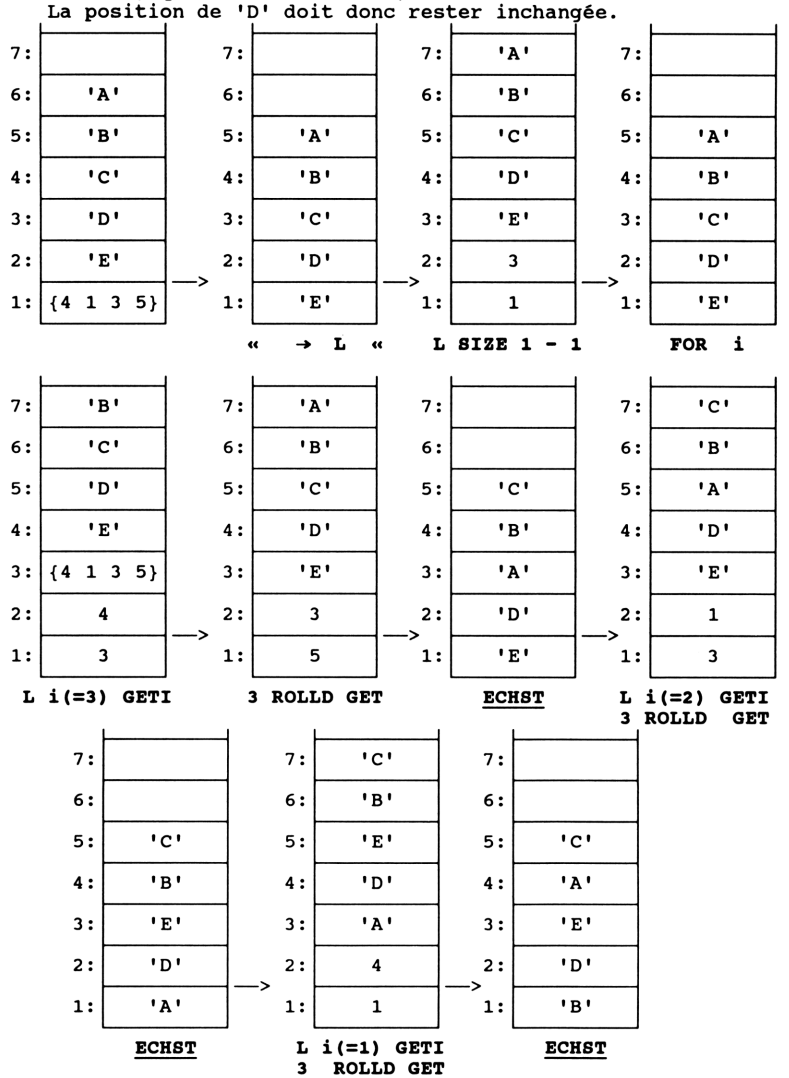
Bien sûr, le programme **'CYCST'** va appeler le programme **'ECHST'** (échange de deux éléments sur la pile: voir exercice 2).

Texte du programme 'CYCST'	Commentaires
« → L « L SIZE 1 - 1 FOR i L i GETI 3 ROLLD GET ECHST -1 STEP » »	Liste dans variable locale L. De i = taille(liste)-1 à 1, en décroissant: Lire les éléments d'indice i et (i+1) de L Echanger objets à ces niveaux. Indice suivant...

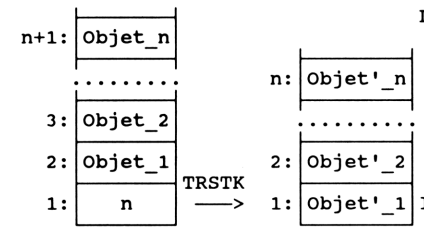
Exemple: La pile contient 'A', 'B', 'C', 'D', 'E' aux niveaux 5,4,3,2,1 et on veut la modifier en utilisant la permutation circulaire définie par la liste { 4 1 3 5 }. Autrement dit:

* 'B', qui est au niveau 4, doit aller au niveau 1.
* 'E', qui est au niveau 1, doit aller au niveau 3.
* 'C', qui est au niveau 3, doit aller au niveau 5.
* 'A', qui est au niveau 5, doit aller au niveau 4.

La position de 'D' doit donc rester inchangée.



Exercice 13: (****) Trier les n premiers objets de la pile.



Le programme **TRSTK** prend un entier n comme unique argument.
Le rôle de **TRSTK** est de trier les n premiers objets de la pile.
A la sortie, on doit avoir:
Objet'_n ≤ Objet'_(n-1)
≤ ...
≤ Objet'_2
≤ Objet'_1.
Il faut bien sûr que les différents objets considérés dans la pile initiale soient comparables au moyen de l'opérateur < (Ces objets doivent donc tous être des nombres réels, ou tous des entiers binaires, ou tous des chaînes de caractères).

La méthode de tri utilisée ici est la méthode d'insertion:
* On place correctement l'élément du niveau 2 par rapport à l'élément du niveau 1 (ce qui se traduit tout au plus par un échange de ces deux éléments s'ils ne sont pas bien placés au départ).
* On insère l'élément du niveau 3 à sa position correcte relativement aux éléments des niveaux 1 et 2 (qui eux, maintenant, sont bien placés l'un par rapport à l'autre).
* On insère l'élément du niveau 4 par rapport aux éléments des niveaux 1,2, et 3, déjà triés.
* etc...
* A la fin, on insère l'élément du niveau n parmi les éléments des niveaux 1,2,..,n-1 (qui sont déjà triés les uns par rapport aux autres).

On comprend donc qu'il faille créer une boucle dont l'indice i (i représentant le niveau de l'objet qu'on doit insérer) varie de 2 à n.

Pour un indice i donné, il faut trouver la position d'insertion de l'objet A qu'on vient de trouver au niveau i.
Cet objet est placé au niveau 1.
Sa position d'insertion est obtenue en créant un compteur k décroissant (on compare A avec l'objet B placé au niveau k, et tant que cet objet B est inférieur à A, on décrémente le compteur).
L'objet A est alors placé à la position définie par la valeur finale du compteur (cette valeur est 1 si tous les objets B comparés à A ont été jugés inférieurs à A).

Remarques:
Cette méthode est plutôt rapide (en tous cas elle a un très bon rapport rapidité/complexité).
Les temps de calculs t suivants ont été obtenus pour le tri de n entiers aléatoires.
* n = 10, t = 1s.
* n = 20, t = 3 à 4s.
* n = 50, t = 16s.
* n = 100, t = 51s.

Evidemment, les temps de calculs augmentent pratiquement proportionnellement au carré du nombre de termes à trier...
Pour une étude plus approfondie des méthodes de tri, reportez vous aux exercices 18,19,20 du chapitre consacré aux listes.

Texte du programme 'TRSTK'	Commentaires
« IF DUP 1 > THEN 2 SWAP FOR i i ROLL i 1 + WHILE DUP PICK 3 PICK < REPEAT 1 - END 1 - ROLLD NEXT ELSE DROP END »	Si le nombre n d'objets à trier est supérieur à 1, alors: De i=2 à i=n, Descend objet A du niveau i. Compteur initialisé à (i+1). Tant que L'objet lu est inférieur à l'objet A, Décrémenter le compteur. On remonte l'élément A à sa position d'insertion. Passer au niveau i suivant... Sinon effacer n.

Exemple: On considère la pile suivante:

Niveau 6	Niveau 5	Niveau 4	Niveau 3	Niveau 2	Niveau 1
413	797	572	390	898	548

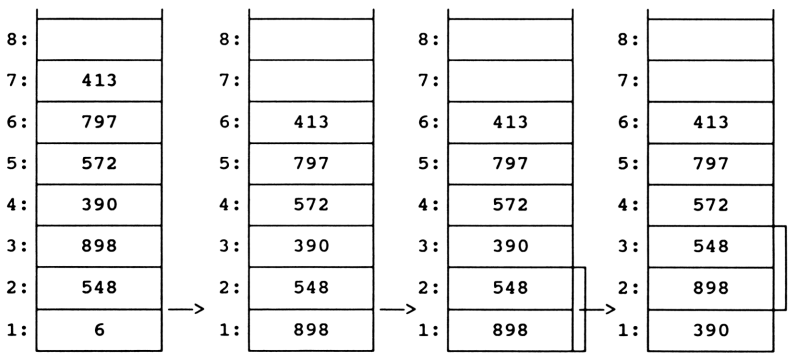
Après appel de **TRSTK**, on doit obtenir:

Niveau 6	Niveau 5	Niveau 4	Niveau 3	Niveau 2	Niveau 1
390	413	548	572	797	898

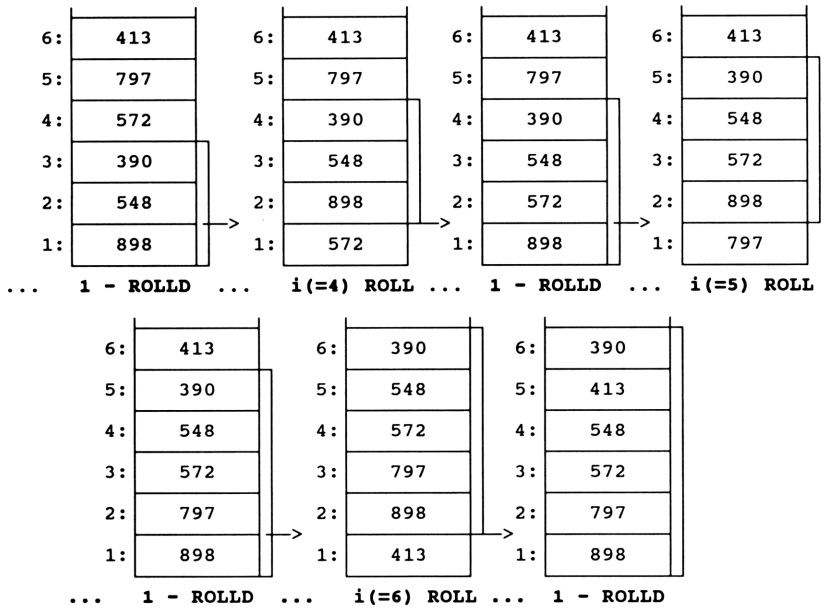
Pour obtenir les piles représentées ci-dessous, on a placé deux **HALT** dans le programme **TRSTK**:

* Après **i ROLL** (l'élément du niveau i vient d'être descendu pour qu'on en trouve la position d'insertion).
* Après **1 - ROLLD** (cet élément vient d'être remonté à sa position d'insertion).

A chaque étape, on a fait figurer sur le coté de la pile quels sont les éléments déjà triés les uns par rapport aux autres.



Situation de départ ... **i(=2) ROLL** .... **1 - ROLLD** ... **i(=3) ROLL**



Exercice 14: (****) Trier les n premiers objets suivant un critère quelconque.

Il s'agit ici, dans la limite des n premiers niveaux, de trier les objets de la pile, suivant les valeurs d'un critère de comparaison (ce qui généralise considérablement **'TRSTK'**).
Les objets doivent être triés suivant les valeurs décroissantes de ce critère.
Si on appelle **'TRCRT'** le programme, la syntaxe sera:

n «...» TRCRT

où n désigne jusqu'à quel niveau de la pile il faut trier, et où «...» désigne un programme, permettant de calculer la valeur du critère de comparaison pour l'objet situé au niveau 1.

La solution est en fait très simple quand on connaît **TRSTK**, car il suffit de modifier très légèrement ce programme.

On met la définition du critère de comparaison dans une variable locale (ci-dessous nommée p), et pour comparer deux éléments, il suffit d'évaluer le critère p pour chacun de ces deux éléments, avant de comparer les résultats obtenus.

Exemples d'utilisation:

n « » TRCRT est équivalent à: **n TRSTK** (car ici le critère de comparaison consiste à garder tel quel l'élément sur la pile).
n « TYPE » TRCRT permet de trier n objets suivant leur type.
n « →STR SIZE » TRCRT permet de trier n objets suivant la longueur de la chaîne qui permet de les représenter.

Texte du programme 'TRCRT'	Commentaires
« IF OVER 1 > THEN → p « 2 SWAP FOR i i ROLL i 1 + WHILE DUP PICK p EVAL 3 PICK p EVAL < REPEAT 1 - END 1 - ROLLD NEXT » ELSE DROP2 END »	Si le nombre n d'objets à trier est supérieur à 1, alors on met le critère dans 'p', et de i=2 à i=n, Descend objet A du niveau i. Compteur initialisé à (i+1). Tant que Le critère pour l'objet lu est < à celui de A, à l'objet A, Décrémenter le compteur. On remonte l'élément A à sa position d'insertion. Passer au niveau i suivant... Sinon effacer n et la définition du critère de comparaison.

Exemples:

7: "Bonjour"	7: 3.14159265359
6: « DROP »	6: (1.5,2.7)
5: (1.5,2.7)	5: "Bonjour"
4: # Ah	4: [[1]]
3: 3.14159265359	3: { A B C }
2: [[1]]	« DROP »
1: { A B C }	1: # Ah

7 «TYPE» TRCRT
——————>
(1 seconde)

7: "Bonjour"	7: # Ah
6: « DROP »	6: "Bonjour"
5: (1.5,2.7)	5: [[1]]
4: # Ah	4: « DROP »
3: 3.14159265359	(1.5,2.7)
2: [[1]]	{ A B C }
1: { A B C }	1: 3.14159265359

7 « →STR SIZE »
TRCRT
——————>
(3 secondes)

NOMBRES REELS OU COMPLEXES : Corrigés des Exercices

Exercice 1: (**) Doubler les instructions **→HMS, HMS→, HMS+, HMS-.**

Bien que celà ne revête qu'un aspect purement pédagogique, il vous est demandé d'écrire des programmes réalisant le même travail que chacune des instructions **→HMS**, **HMS→**, **HMS+**, **HMS-**.

Soit x un nombre réel et soit ϵ son signe.

Avec l'interprétation décimale habituelle, x s'écrit:
x = ϵA.bcdefgh.... = ϵa + ϵbc/100 + ϵde.fgh.../10000
où a est un entier positif, b,c,d,e,f,g,h... étant les chiffres décimaux succcessifs.
On peut noter x = ϵA + ϵB/100 + ϵC/10000, avec:
B entier (0 ≤ B ≤ 99), C réel (0 ≤ C < 100).
A partir de x, on trouve ϵA par l'instruction **IP**.
L'instruction **FP** transforme x en y = ϵB/100 + ϵC/10000
IP et FP transforment alors 100y en ϵB et ϵC/100.
Il suffit de multiplier ce résultat par 100 pour obtenir ϵC.
On voit ainsi comment isoler ϵA, ϵB, ϵC. Ici l'utilisation de **IP** et **FP** est indispensable pour conserver le signe.
La connaissance de ϵA, ϵB, ϵC permet de reconstruire x par les opérations successives (notation **RPN**):
« ϵC 100 / ϵB + 100 / ϵA + »

Avec l'interpréation **HMS**, le même réel x peut s'écrire:
x = ϵH.qrstuvw... = ϵH + ϵqr/60 + ϵst.uvw.../3600.
= ϵH + ϵM/60 + ϵS/3600, avec:
H entier positif, M entier (0≤M≤59), S réel (0≤S<60).
Pour isoler ϵH, ϵM, ϵS à partir de x, où pour au contraire reconstituer x, on applique la méthode décrite ci-dessus (à ceci près que le nombre 100 est remplacé par 60).

On notera **'→→HMS'** et **'HMS→→'** les programmes qui doublent les instructions **→HMS** et **HMS→**. Ils sont pratiquement identiques (on pourrait exploiter cette similitude).

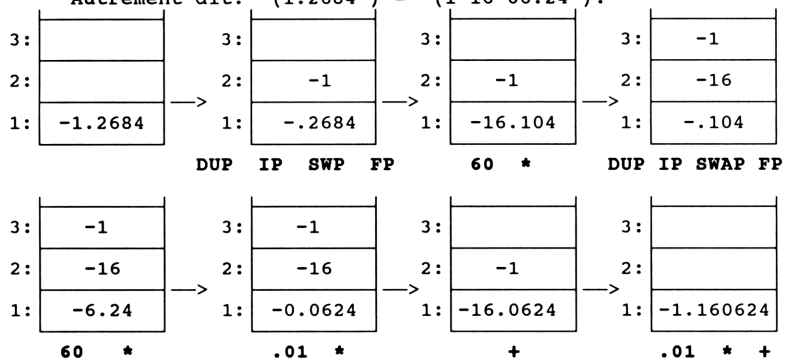
Le programme **'→→HMS'**, par exemple, et en reprenant les notations ci-dessus, à isoler ϵH, ϵM, ϵS et à reformer un nombre décimal à partir de ces données.

Les programmes doublant **HMS+** et **HMS-** sont évidents: il suffit de convertir les réels du format **HMS** vers le format habituel, de les additionner ou de les soustraire, et de remettre le résultat au format **HMS**.

REELS OU COMPLEXES: Corrigés.

Texte du programme '→→HMS'	Commentaires
« 1 2 START DUP IP SWAP FP 60 * NEXT 1 2 START .01 * + NEXT »	On isole les parties H, M, S du du réel initial. On forme l'écriture décimale au format HMS.

Exemple: -1.2684 est transformé par →→HMS en -1.160624
Autrement dit: -(1.2684°) = -(1°16'06.24").

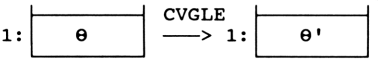


Texte du programme 'HMS→→'	Commentaires
« 1 2 START DUP IP SWAP FP 100 * NEXT 1 2 START 60 / + NEXT »	On isole les composantes décima- -les du réel initial. On reforme l'écriture décimale habituelle de ce réel.

Texte du programme 'HMSP'	Commentaires
« HMS→→ SWAP HMS→→ + →→HMS »	On met les deux réels au format décimal classique. On les ajoute. On met le résultat au format HMS

Texte du programme 'HMSM'	Commentaires
« HMS→→ SWAP HMS→→ SWAP - →→HMS »	Les 2 réels au format classique. On les met dans l'ordre initial. On les soustrait puis on met le résultat au format HMS

Exercice 2: (*) Convertir un angle en sa mesure principale.

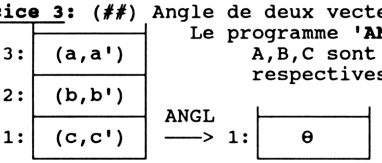


Le programme '**CVGLE**' considère que le réel θ est la mesure d'un angle, et renvoie la mesure principale θ' de cet angle.
θ est interprété suivant le mode en cours (degrés,radians,grades)

Il suffit de créér le nombre complexe de module 1 et d'argument θ, la HP48SX se chargeant elle-même de convertir θ. On décompose ensuite le complexe obtenu pour récupérer θ'.
Le flag -16 doit être armé pour pouvoir utiliser **→V2** et **V→** en mode polaire.

Texte du programme 'CVGLE'	Commentaires
« RCLF -16 SF 1 ROT →V2 V→ 3 ROLLD DROP STOF »	Sauve les flags, arme flag -16. Crée (1,_θ') ou [1,_θ']. Sort 1,θ' et remonte θ'. élimine 1 et rétablit les flags.

Exercice 3: (**##**) Angle de deux vecteurs.

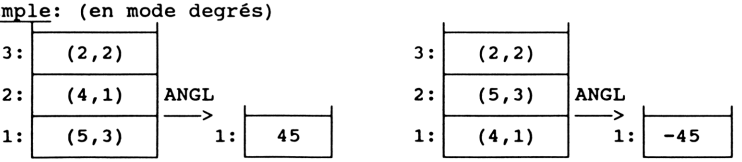


Le programme '**ANGL**' calcule l'angle $(\vec{AB},\vec{AC})$, où A,B,C sont les points du plan de coordonnées respectives (a,a'), (b,b'), (c,c').
La valeur θ obtenue est la mesure principale de cet angle:
En mode degrés: $-180 < \theta \leq 180$
En mode radians: $-\pi < \theta \leq \pi$
En mode grades: $-200 < \theta \leq 200$

Pour déterminer l'angle $\theta = (\vec{AB},\vec{AC})$ il faut calculer l'angle $\theta'=(\vec{Ox},\vec{AC})$ (angle polaire du vecteur $\vec{AC}$), puis l'angle polaire $\theta"=(\vec{Ox},\vec{AB})$ du vecteur $\vec{AB}$.
L'angle θ cherché est alors égal à θ'-θ" (après conversion éventuelle en la mesure principale).
La conversion (par utilisation de **CVGLE**) est nécessaire car à priori, le résultat de θ'-θ" est compris entre -2π et 2π (en mode radians) ou -360 et 360 (en mode degrés), ou -400 et 400 (en mode grades).

Texte du programme 'ANGL'	Commentaires
« 3 PICK - ARG SWAP ROT - ARG - CVGLE »	Vecteur $\vec{AC}$ Angle polaire du vecteur $\vec{AC}$ Vecteur $\vec{AB}$ Angle polaire du vecteur $\vec{AB}$ Angle $(\vec{AB},\vec{AC})$ puis conversion en sa mesure principale.

Exemple: (en mode degrés)



Exercice 4: (*) Périmètre d'un triangle.

Le programme **'PERI'** calcule le périmètre du triangle ABC.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le périmètre au niveau 1.

Le périmètre est bien sûr égal à ABS(AB)+ABS(BC)+ABS(CA).
La seule fantaisie qu'on peut ici se permettre consiste à calculer cette somme en un minimum d'instructions, et si possible sans créer de variable locale.

Texte du programme 'PERI'	Commentaires
« 3 DUPN 0 1 3 START 3 ROLLD - ABS + NEXT »	On duplique A,B,C. Périmètre initialisé à 0. Trois fois de suite: Monter le périmètre actuel au niveau 3. La longueur d'un coté. S'ajoute au périmètre. Passer au coté suivant...

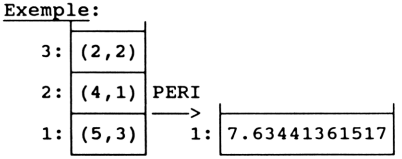
Exercice 5:(**) Aire d'un triangle.

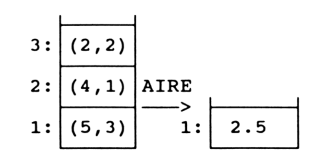
Le programme **'AIRE'** calcule l'aire intérieure au triangle ABC.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: l'aire au niveau 1.

L'aire intérieure au triangle ABC est (1/2)*ABS(AB AC).
Rappelons que le produit vectoriel de [x,y] et de [x',y'] est égal à [0, 0, xy'-yx'] dont le module est |xy'-yx'|.
A partir de 2 complexes z = (x,y) et z' = (x',y'), la quantité xy'-yx' est égale **IM(CONJ(z)*z')**. En effet:
CONJ(z)*z' =(x,-y)*(x',y') = (xx'+yy',xy'-yx').
La solution est alors très simple:

Texte du programme 'AIRE'	Commentaires
« OVER - 3 ROLLD - CONJ * IM ABS 2 / »	Le vecteur $\overrightarrow{BC}$ placé au niveau 3. Le conjugué du vecteur $\overrightarrow{BA}$. Produit des deux complexes, puis module de la partie imaginaire. Pour obtenir l'aire du triangle.

Exemple:





Exercice 5: (***) Projection d'un point sur une droite.

Le programme **'PROJ'** détermine la projection orthogonale H(h,h') du point A sur la droite portant les points B et C.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le nombre complexe (h,h') au niveau 1.

Il suffit de savoir que le point H vérifie : $\overrightarrow{BH} = (\overrightarrow{BA}.\overrightarrow{U}) \overrightarrow{U}$, où où $\overrightarrow{U}$ est le vecteur unitaire égal à $\overrightarrow{BC}$/ABS($\overrightarrow{BC}$), et où le point entre $\overrightarrow{BA}$ et $\overrightarrow{U}$ désigne le produit scalaire.
L'instruction **SIGN** calcule le vecteur $\overrightarrow{U}$ à partir de $\overrightarrow{BC}$.
Le produit scalaire de [x,y] et [x',y'] est xx'+yy'. A partir des nombres complexes z=(x,y) et z'=(x',y'), la quantité xx'+yy' est égale à **RE(z * CONJ(z'))**:
z*CONJ(z') = (x,y)*(x',-y') = (xx'+yy', -xy'+yx').

Texte du programme 'PROJ'	Commentaires
«	
OVER -	Vecteur $\overrightarrow{BC}$.
SIGN	Vecteur unitaire $\overrightarrow{U}$.
ROT 3 PICK -	Vecteur $\overrightarrow{BA}$.
OVER CONJ	Conjugué du vecteur $\overrightarrow{U}$.
* RE	Produit puis partie réelle.
*	On multiplie $\overrightarrow{BA}.\overrightarrow{U}$ par $\overrightarrow{U}$.
+	On ajoute ce vecteur au point B.
»	

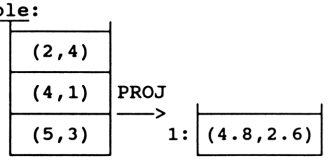
Exercice 6: (#) Symétrique d'un point par rapport à une droite.

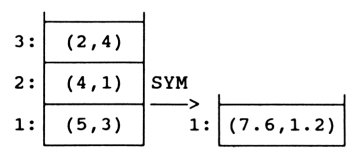
Le programme **'SYM'** détermine le symétrique orthogonal D(d,d') du point A par rapport à la droite portant les points B et C.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: le nombre complexe (d,d') au niveau 1.

Il suffit de remarquer qu'en appelant H la projection orthogonale de A sur la droite (BC), alors $\overrightarrow{AD} = 2\overrightarrow{AH}$.
En utilisant le programme **'PROJ'**, la solution est immédiate.

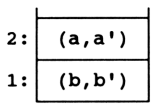
Texte du programme 'SYM'	Commentaires
«	
3 PICK 3 ROLLD	Le point A aux niveaux 3 et 4.
PROJ	Calcul de la projection H.
OVER - 2 *	2 fois le vecteur $\overrightarrow{AH}$.
+	On ajoute le résultat au point A
»	

Exemple:





Exercice 7: (***) Equation de la droite passant par 2 points donnés.



Le programme **'DRTE'** calcule l'équation de la droite qui passe par les points A(a,a') et B(b,b'). Le résultat est une expression du type 'a*X+b*Y=c', où a,b,c sont des réels (ce sont les coefficients de l'équation).

Une équation de la droite AB est 'αX+ßY=τ', avec:
α = a'-b', ß = b-a, et τ = ba'-ab'.

Texte du programme 'DRTE'	Commentaires
« DUP2 - C→R	a-b au dessus de a'-b'.
'X' *	(a'-b')*X.
SWAP NEG 'Y' *	(b-a)*Y.
+	(a'-b')*X+(b-a)*Y.
3 ROLLD	Au dessus de (a,a') et de (b,b')
CONJ	(a,a') au dessus de (b,-b').
* IM	Calcul de ba'-ab'.
=	On obtient l'équation.
»	

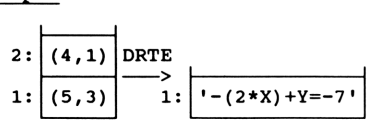
Exercice 8: (##) Perpendiculaire à une droite, passant par un point.

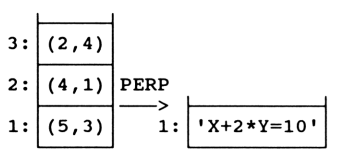
Le programme **'PERP'** donne l'équation de la perpendiculaire à la droite BC, et qui passe par A.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: Une équation du type 'a*X+b*Y=c'.

Il suffit ici de connaître un 2-ème point de la perpendiculaire.
B et C ont pour coordonnées respectives (b,b') et (c,c').
(b-c,b'-c') représente donc un vecteur directeur de la droite BC.
Si on multiplie ce nombre complexe par (0,1), on obtient un nombre complexe représentant un vecteur othogonal à celui-là.
On ajoute aux coordonnées de A les composantes du nombre complexe obtenu pour avoir un deuxième point de la droite cherchée.

Texte du programme 'PERP'	Commentaires
« -	Vecteur $\overrightarrow{CB}$.
(0,1) *	Vecteur orthogonal à $\overrightarrow{CB}$.
OVER +	On l'ajoute à A, en gardant A.
DRTE	Equation de la perpendiculaire.
»	

Exemple:





Exercice 9: (#) Médiatrice d'un segment.

'MDTR' donne l'équation de la médiatrice du segment [A,B].
Le schéma fonctionnel est celui de l'exercice 7.

La solution la plus simple consiste à observer que la médiatrice du segment AB est la perpendiculaire à ce segment en son milieu. On utilise donc le programme **'PERP'**.

Texte du programme 'MDTR'	Commentaires
« DUP2 + 2 / 3 ROLLD PERP »	Milieu du segment AB. Placé au dessus de (a,a'),(b,b') Equation de la médiatrice.

Exercice 10: (##) Bissectrice de deux demi-droites.

Le programme **'BISS'** donne l'équation de la bissectrice des deux demi-droites [A,B[et [A,C[.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: Une équation du type 'a*X+b*Y=c'.

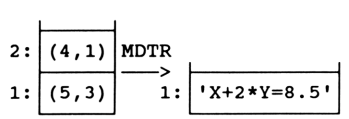
Il suffit de trouver un deuxième point de cette bissectrice, car A est déjà connu.

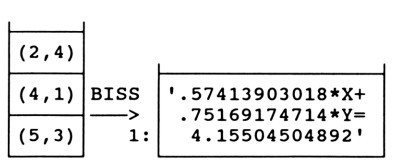
On observe que si $\vec{U}$ et $\vec{V}$ sont les vecteurs unitaires définis par $\vec{U}$ = SIGN($\vec{AB}$) et $\vec{V}$ = SIGN($\vec{AC}$), alors le point obtenu en ajoutant $\vec{U}+\vec{V}$ aux composantes de A appartient à la bissectrice.

Remarque: Si les deux demi-droites [A,B[et [A,C[sont opposées, cette méthode ne marche pas (car le vecteur U+V est nul).
Dans ce cas, la bissectrice n'est autre que la perpendiculaire à la droite BC, menée par A, ce qui nous ramène à **'PERP'**.

Texte du programme 'BISS'	Commentaires
« 3 PICK - SIGN SWAP 3 PICK - SIGN + OVER + DRTE »	Vecteur $\vec{V} = \vec{AC}/\|\vec{AC}\|$. Vecteur $\vec{U} = \vec{AB}/\|\vec{AB}\|$. Vecteur $\vec{U}+\vec{V}$. Ajouté à A en gardant A. Equation de la bissectrice.

Exemple:





Exercice 11: (##) Trois points sont-ils alignés?

Le programme **'ALIG'** teste si les 3 points A,B,C sont alignés.
Pile fonctionnelle de départ: Voir exercice 3.
Pile fonctionnelle finale: L'entier 1 si A,B,C sont alignés,
et 0 s'ils ne le sont pas.

On peut envisager trois solutions:

Première méthode:
On détermine l'équation de la droite BC (programme **'DTRE'**) et on examine si le point A vérifie ou non cette équation.
Il faut placer dans 'X' et dans 'Y' les composantes du nombre complexe (a,a') qui représente A, puis évaluer l'équation 'αX+βY=τ' au moyen de l'instruction **→NUM** (le résultat est nul si l'équation est vérifiée, non nul sinon).

Deuxième méthode:
Les points A,B,C sont alignés si et seulement si l'angle $(\overrightarrow{AB},\overrightarrow{AC})$ est égal à 0 modulo π (en radians).
L'idée est donc de calculer une mesure θ de l'angle $(\overrightarrow{AB},\overrightarrow{AC})$ (programme **'ANGL'**), et de vérifier si l'angle 2*θ est égal à zéro modulo 2π (programme **'CVGLE'**).

Troisième méthode:
Les points A(a,a'), B(b,b'), C(c,c') sont alignés si et seulement si le déterminant ci-contre est égal à 0.

$$\begin{vmatrix} a & a' & 1 \\ b & b' & 1 \\ c & c' & 1 \end{vmatrix}$$

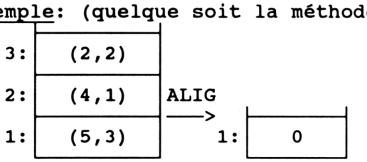
Remarque: pour chacune de ces méthodes, il est recommandé de tenir compte des erreurs d'arrondi possibles. Un réel pourra ainsi être considéré comme nul s'il est inférieur à 1E-10 en valeur absolue.

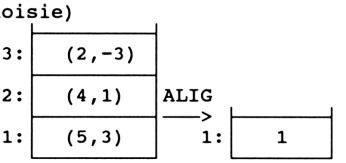
Texte du programme 'ALIG' (Première solution)	Commentaires
« DRTE SWAP C→R 'Y' STO 'X' STO →NUM 1E-10 < { X Y } PURGE »	Equation de la droite BC. Coordonnées a, a' de A. a' va dans Y, et a va dans X. Evalue la différence d des deux membres de l'équation. 1 si |d|<1E-10. Purge les variables X et Y.

Texte du programme 'ALIG' (Deuxième solution)	Commentaires
« ANGL 2 * CVGLE ABS 1E-10 < »	Calcule l'angle θ=$(\overrightarrow{AB},\overrightarrow{AC})$. 2θ. Convertit en mesure principale Φ 1 si |Φ|<1E-10.

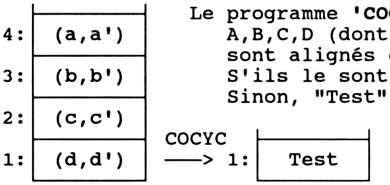
Texte du programme 'ALIG' (Troisième solution)	Commentaires
« C→R 1 4 ROLL C→R 1 7 ROLL C→R 1 { 3 3 } →ARRY DET ABS 1E-10 < »	c, c', et 1. b, b', et 1. a, a', et 1. Calcule le déterminant D. 1 si |D|<1E-10.

Exemple: (quelque soit la méthode choisie)





Exercice 12: (##) Quatre points sont-ils alignés ou cocycliques?

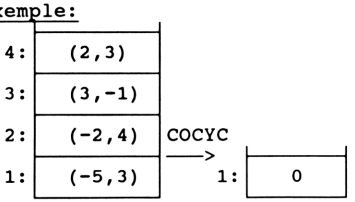


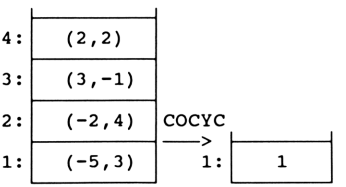
Le programme **'COCYC'** teste si les quatre points A,B,C,D (dont les coordonnées sont sur la pile) sont alignés ou cocycliques.
S'ils le sont, "Test" vaut 1.
Sinon, "Test" vaut 0.

Les quatre points A,B,C,D sont alignés ou cocycliques si les angles $(\vec{AC},\vec{AD})$ et $(\vec{BC},\vec{BD})$ sont égaux modulo π.
Il suffit donc de calculer ces deux angles θ et θ' par **'ANGL'** et de vérifier si 2*(θ-θ') est égal à 0 modulo 2π (**'CVGLE'**).

Texte du programme 'COCYC'	Commentaires
« 3 DUPN ANGL 5 ROLLD ROT DROP ANGL - 2 * CVGLE ABS 1E-10 < »	Angle θ=$(\vec{BC},\vec{BD})$. Sauvegarde θ, et élimine B. Angle θ'=$(\vec{AC},\vec{AD})$, puis 2(θ-θ'). Convertit en mesure principale Φ 1 si |Φ|<1E-10.

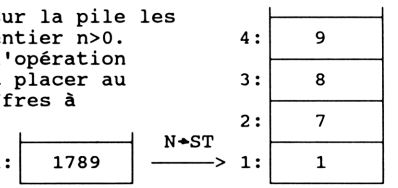
Exemple:





Exercice 13: (*) Libérer les chiffres d'un entier sur la pile et opération inverse.

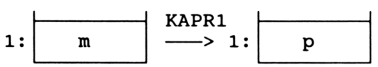
Le programme 'N→ST' place sur la pile les différents chiffres d'un entier n>0.
Le programme 'ST→N' réalise l'opération inverse. Il faut cependant placer au niveau 1 le nombre de chiffres à considérer.
Les programmes 'N→ST' et 'ST→N' seront utilisés dans l'exercice 14.



Texte du programme 'N→ST'	Commentaires
« 1 OVER XPON START DUP 10 MOD SWAP 10 / FLOOR NEXT »	De 1 jusqu'à l'exposant de n: On isole le dernier chiffre. On isole le reste du nombre.

Texte du programme 'ST→N'	Commentaires
« 2 SWAP START 10 * + NEXT »	Nombre de fois qu'il faut: multiplier l'entier du niveau 1 par 10 et y ajouter le chiffre du niveau 2.

Exercice 14: (***###***) Algorithme de Kaprekar (première version).



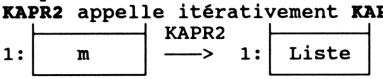
L'algorithme de Kaprekar part d'un entier m,calcule l'entier m' obtenu en écrivant les chiffres de m dans l'ordre décroissant (ex: 8917 => 9871),l'entier m" obtenu en écrivant ces chiffres dans l'ordre croissant, puis calcule l'entier p égal à m'-m".(Si m=8917: m'=9871, m"=1789, p=9871-1789=8082).
Ainsi: 1789 => 8082 => 8532 => 6174 => 6174 etc...

Il est nécessaire d'utiliser le programme **TRSTK** (tri des n premiers niveaux de la pile, voir Exercice 13 du chapitre consacré à la pile opérationnelle).

Texte du programme 'KAPR1'	Commentaires
« DUP XPON 1 + → n « N→ST n TRSTK 1 n 2 * FOR k k PICK 2 STEP n ST→N n 1 + ROLLD n ST→N SWAP - » »	n: nombre de chiffres. Place ces chiffres sur la pile. Les trie. Duplique les n premiers niveaux en inversant l'ordre. On obtient l'entier m". On le place au niveau n+1. On obtient l'entier m'. On trouve p = m'-m".

Exercice 15: (##) Algorithme de Kaprekar (deuxième version).

On veut généraliser l'observation faite pour les appels répétés de **KAPR1** (à partir d'un entier de 4 chiffres).
Il est assez clair que de tels appels répétés finissent toujours par nous faire obtenir un entier qui est déjà apparu dans les résultats précédents (d'où une périodicité).
L'objectif de **KAPR2** est d'étudier de telles périodicités:
On part d'un entier m.
KAPR2 appelle itérativement **KAPR1** et s'arrête quand 2 résultats identiques 'n' ont été trouvés. Il donne alors la liste de tous les résultats obtenus depuis m, jusqu'à la 2ème occurence de n.



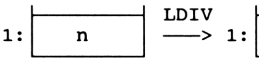
Texte du programme 'KAPR2'	Commentaires
« { } WHILE OVER DUP2 POS NOT REPEAT + SWAP KAPR1 SWAP END + SWAP DROP »	Liste vide L. Tant que l'entier au-dessus de L n'appartient pas à L: Ajouter cet entier n à L. Placer n au niveau 1. Appliquer "Kaprekar". Résultat n' au dessus de L. Ajouter l'élément prouvant la périodicité.

Voici une autre version de **'KAPR1'** utilisant un **'DO...UNTIL'**.

Texte du programme 'KAPR2'	Commentaires
« { } OVER DO + SWAP KAPR1 SWAP UNTIL OVER DUP2 POS END + SWAP DROP »	n au-dessus de L, au dessus de n Répéter: Ajouter n à L. Appliquer "Kaprekar" à n. Résultat n' au dessus de L. Jusqu'à ce que n' soit déjà dans la liste L. Ajouter l'élément prouvant la périodicité.

Exemples:
Voici quelques exemples de listes obtenues:
{ 1789 8082 8532 6174 6174 } (La suite stationne en 6174 pour tout entier initial de 4 chiffres).
{ 12345 41976 82962 75933 63954 61974 82962 } (période 4)
{ 20461 62964 71973 83952 74943 62964 } (période 4)
{ 68393 64944 51975 81972 85932 74943 62964 71973 83952 74943 } (là aussi la période est 4)
{ 918682 861732 752643 530865 829962 771723 653544 310887 873522 651744 620874 851742 750843 840852 860832 862632 642654 420876 851742 } (la période est 7).

Exercice 16: (***) Liste de tous les diviseurs d'un entier positif.



Le programme **'LDIV'** donne la liste de tous les diviseurs de l'entier positif n.

Par exemple, si n=100 la liste est { 1 2 4 5 10 20 25 50 100 }.

On remarque que lorsqu'un diviseur k de n est trouvé, alors un autre diviseur, n/k, est obtenu au même moment.

Il suffit alors d'examiner tous les diviseurs possibles de n, de 1 à √n: La remarque précédente permet alors d'obtenir les diviseurs de n qui sont supérieurs à √n.

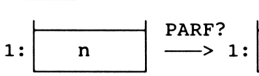
On utilise donc deux listes qui augmentent parallèlement et qui seront réunies à la fin.

Un test en fin de programme permet d'éviter (si n est un carré parfait), que le diviseur √n ne soit compté deux fois.

Texte du programme 'LDIV'	Commentaires

`« → n`
`  « { 1 }`
`  { } n +`
`  2 n √ FOR k`
`  IF n k MOD NOT`
`  THEN`
`  SWAP k +`
`  n k / ROT +`
`  END`
`  NEXT`
`  IF n √ DUP FLOOR ==`
`  THEN`
`  2 OVER SIZE SUB`
`  END`
`  +`
`  »`
`»`
	On veut la liste de ses diviseurs
	Début Liste L1 des diviseurs ≤ √n
	Fin Liste L2 des diviseurs > √n.
	De k=2 à k=√n:
	Si n est divisible par k,
	Alors:
	Ajouter k à L1.
	Ajouter n/k au début de L2.
	
	Aller à l'entier k suivant...
	Si n est un carré parfait,
	
	Enlever le 1er terme (√n) de L2
	
	Ajouter L1 et L2.

Exercice 17: (#) Un entier positif est-il parfait?



On dit qu'un entier positif n est parfait si n est égal à la somme de ses diviseurs d, 1≤d<n.

Par exemple, 28 est parfait car: 28 = 1 + 2 + 4 + 7 + 14.

Le programme **'PARF?'** teste si l'entier n est parfait: S'il l'est "Test" vaut 1, sinon, "Test" vaut 0.

C'est très facile en utilisant le programme précédent:

On crée la liste de tous les diviseurs de n, et on les retranche successivement de n (à l'exception de n lui-même).

L'entier n est parfait si le résultat obtenu est nul (il suffit alors de faire un **"NOT"** pour obtenir 1).

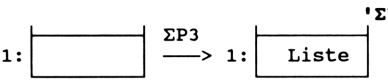
Texte du programme 'PARF?'	Commentaires
« LDIV OBJ→ 2 SWAP START SWAP - NEXT NOT »	Crée la liste des diviseurs de n (n en est le dernier élément). Libère cette liste sur la pile. Autant de fois que de diviseurs propres de n: Enlever un diviseur propre. 1 si n est parfait, 0 sinon.

Remarque: le programme

```
« { }
  DO
  IF OVER PARF? THEN OVER + END
  SWAP 1 + SWAP
  UNTIL KEY END DROP
»
```

Crée la liste des entiers parfaits ≥ n, où n est l'entier que vous placez au niveau 1. Le programme peut être interrompu par un appui sur une touche quelconque.
On se retrouve avec une liste d'entiers parfaits au niveau 1, et au niveau 2 le dernier entier examiné.
On trouve ainsi la liste { 6 28 496 ... }

Exercice 18: (**) Nombres égaux à la somme des cubes de leurs trois chiffres.



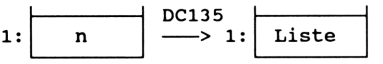
'ΣP3' place sur la pile les entiers compris entre 100 et 999 et qui sont égaux à la somme des cubes de leurs trois chiffres.

Il suffit de créer trois boucles imbriquées pour les trois chiffres c,d,u, de calculer l'entier cdu (en base 10) et l'entier c^3+d^3+u^3, puis de conserver cdu si les deux résultats précédents sont égaux.

Texte du programme 'ΣP3'	Commentaires
« 1 9 FOR c 0 9 FOR d 0 9 FOR u c 100 * d 10 * + u + DUP c 3 ^ d 3 ^ + u 3 ^ + ≠ DROPN NEXT NEXT NEXT »	Chiffre des centaines. Chiffre des dizaines. Chiffre des unités. Nombre cdu. Sauvegarde cdu. Somme Σ des trois cubes. Si Σ≠cdu, on détruit cdu. Unité suivante... Dizaine suivante... Centaine suivante...

On trouve les 4 solutions (153, 370, 371, et 407) en 1mn47s.

Exercice 19: (**) Trouver toutes les solutions entières positives ou nulles de l'équation **x+3y+5z=n**.



'DC135' donne la liste des décompositions possibles de l'entier n sous la forme:
n=x+3*y+5*z, où x,y,z sont des entiers positifs ou nuls.
A partir de 10 on doit ainsi trouver:
{ {0 0 2} {1 3 0} {2 1 1} {4 2 0} {5 0 1} {7 1 0} {10 0 0} }

Il faut créer trois boucles imbriquées pour décrire les valeurs possibles des trois variables x,y,z.

Mais plutôt que de considérer trois variables x, y, z augmentant avec un pas de 1, on va considérer trois variables x, b, c augmentant avec des pas respectifs de 1,3,5, le problème étant alors de résoudre x+b+c=n.
* x varie de 0 à n.
* Pour une valeur fixée de x, b varie de 0 à n-x (de 3 en 3).
* Pour des valeurs fixées de x et b, c varie de 0 à n-x-b, avec un incrément de 5.

On est ainsi certain d'examiner toutes les possiblités, en évitant surtout des essais inutiles.

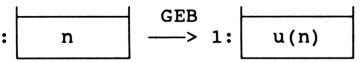
Texte du programme 'DC135'	Commentaires

`« → n`
`« { }`
`0 n FOR x`
`0 n x - FOR b`
`0 n x - b - FOR c`
`IF x b + c + n ==`
`THEN`
`x b 3 / c 5 /`
`3 →LIST 1 →LIST`
`+`
`END`
`5 STEP`
`3 STEP`
`NEXT`
`»`
`»`
	n: nombre qu'on veut décomposer.
	Liste L vide pour l'instant.
	De x=0 à x=n (de 1 en 1)
	De b=0 à b=n-x (de 3 en 3)
	De c=0 à c=n-x-b (de 5 en 5)
	Si (x+b+c)=n,
	Alors:
	On crée la liste
	{ x, y(=b/3), z(=c/5) }
	et on l'ajoute à L.
	
	Valeur suivante de c...
	Valeur suivante de b...
	Valeur suivante de x...

Exemples:

Valeur de n	Liste obtenue
0	{ {0 0 0} }
1	{ {1 0 0} }
2	{ {2 0 0} }
3	{ {0 1 0} {3 0 0} }
5	{ {0 0 1} {2 1 0} {5 0 0} }
15	{ {0 0 3} {0 5 0} {1 3 1} {2 1 2} {3 4 0} {4 2 1} {5 0 2} {6 3 0} {7 1 1} {9 2 0} {10 0 1} {12 1 0} {15 0 0} }

Exercice 20: (***) n-ème terme d'une suite récurrente.



On considère la suite récurrente définie par u(1)=u(2)=1, et pour n≥3, par la relation:
u(n) = u(n-u(n-1)) + u(n-u(n-2)).
Le programme **'GEB'** calcule la valeur de u(n) où n est un entier.

L'intérêt de cet exercice est dans la comparaison de ses deux solutions. On y voit en particulier que l'emploi de la récursivité est parfois un très mauvais choix.

Première méthode (récursive):
Elle conduit à un texte très lisible. On veut calculer u(n):
Si n=1 ou 2, le résultat est connu. Sinon il faut appeler **'GEB'** pour calculer u(n-1),u(n-u(n-1),u(n-2) et u(n-u(n-2)).

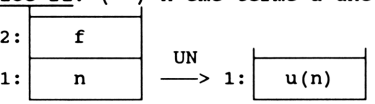
Texte du programme 'GEB' (Version récursive)	Commentaires
« → n « IF n 2 ≤ THEN 1 ELSE n n 1 - GEB - GEB n n 2 - GEB - GEB + END » »	On veut calculer u(n). u(1)=u(2)=1. On calcule n - u(n-1). On calcule u(n-u(n-1)). On calcule n - u(n-2). On calcule u(n-u(n-2)). On obtient u(n).

Deuxième méthode (itérative):
Ici, on va calculer la liste de tous les u(k) avec 1≤k≤n, ce qui paradoxalement va donner une solution plus rapide.

Texte du programme 'GEB' (Version itérative)	Commentaires
« IF DUP 2 ≤ THEN DROP 1 ELSE { 1 1 } 3 ROT FOR k DUP k OVER k 1 - GET - GET OVER k OVER k 2 - GET - GET + + NEXT DUP SIZE GET END »	u(1)=u(2)=1. Liste initiale. De k=3 à k=n: Sauve la liste, place k. C'est u(k-1). On obtient k-u(k-1). C'est u(k-u(k-1)). Sauve la liste, place k. C'est u(k-2). On obtient k-u(k-2). C'est u(k-u(k-2)). On obtient u(k). On l'ajoute à la liste. Au suivant... On extrait le dernier (u(n)). (Si on enlève "DUP SIZE GET", on obtient la liste des u(k),k≤n)

Comparaison des deux méthodes:
Le texte de la version récursive est beaucoup plus clair.
On note toutefois que chaque appel de **'GEB'** provoque 4 appels récursifs de **'GEB'**. Les 'feuilles' de l'arbre de ces appels récursifs sont atteintes quand on a besoin de u(1) et de u(2), qui sont connus, mais le prix à payer en temps de calcul est élevé et croît très rapidement avec n.
Les temps de calculs sont:
'GEB' récursif: u(5):2s u(7):4s u(10):17s u(11):29s
'GEB' itératif: u(10):1s u(20):<3s u(50):7s u(100):30s.

Exercice 21: (**) n-ème terme d'une suite.



Le problème est ici de calculer le n-ème terme d'une suite u, défini par:

$$u(n) = \sqrt{f(1) + \sqrt{f(2) + \sqrt{\ldots\ldots + \sqrt{f(n-1) + \sqrt{f(n)}}}}}.$$

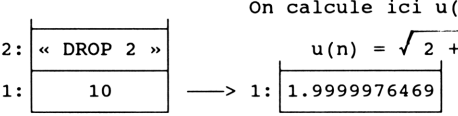
où "f" est un programme censé s'appliquer à un entier situé au niveau 1, et renvoyant un résultat numérique.

La solution est très simple: Il suffit de créer une boucle avec un indice k variant de k=n à k=1, en décroissant (de manière à effectuer les calculs en partant de la racine carrée la plus profonde).

Texte du programme 'UN'	Commentaires
« 0 SWAP 1 FOR k √ k 3 PICK EVAL + -1 STEP √ SWAP DROP »	Pour initialiser le résultat r. De k=n à k=1, en décroissant. On calcule √r. On calcule f(k). On l'ajoute à √r. Une dernière fois √ Eliminer le programme f.

Exemples:

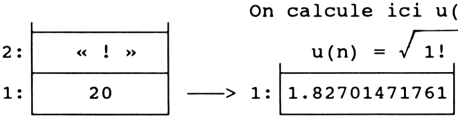
On calcule ici u(10) avec:



$$u(n) = \sqrt{2 + \sqrt{2 + \sqrt{2 \quad \ldots + \sqrt{2}}}}\ .$$

Il est facile de voir que Cette suite converge vers 2.

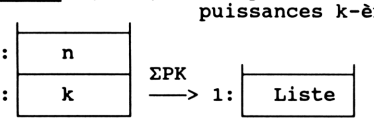
On calcule ici u(20) avec:



$$u(n) = \sqrt{1! + \sqrt{2! + \sqrt{3! \quad \ldots + \sqrt{n!}}}}\ .$$

On constate que u(n) reste constant (sur 12 chiffres) à partir de n=10.

Exercice 22: (****) Décompositions d'un nombre en somme de deux puissances k-èmes.



Le programme **'ΣPK'** cherche les décompositions possibles d'un entier n en somme de deux puissances k-èmes.

Les différentes solutions sont placées dans une liste au niveau 1 (cette liste est vide s'il n'y a aucune solution).

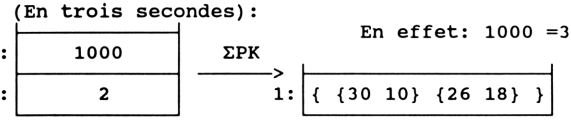
Exemple: avec n=87539319 et k=3, on doit trouver la liste { {436 167} {423 228} {414 255} }.
En effet: 87539319 = 436^3+167^3 = 423^3+228^3 = 414^3+255^3.

Si n=a^k+b^k, avec 1≤a≤b, alors 1≤a^k≤n/2. Donc 1≤a≤(n/2)^(1/k).
On peut par conséquent se limiter aux valeurs entières comprises entre 1 et (n/2)^(1/k).
Pour chacune de ces valeurs a, on calcule c=n-a^k, puis la racine k-ème b' de c (instruction **XROOT**)
On remplace b' par sa partie entière b (de manière à être sûr de ne trouver que des solutions entières), puis on teste si a^k+b^k=n.
Si cette égalité est réalisée, on ajoute la liste {a b} à la liste des solutions (liste vide au début).

Texte du programme 'ΣPK'	Commentaires
« → n k « { } 1 n 2 / k XROOT FOR a n a k ^ - DUP k XROOT FLOOR IF DUP k ^ ROT == THEN a 2 →LIST 1 →LIST + ELSE DROP END NEXT » »	On veut décomposer n en somme de deux puissances k-èmes. Initialise la liste L des solut. De a=1 à a=(n/2)^(1/k): On calcule n-a^k. On calcule b'=(n-a^k)^(1/k). b = partie entière de b'. Si b^k = n-a^k, alors: Liste { b a }. On l'ajoute à la liste L. Sinon détruire b. Valeur suivante de a.

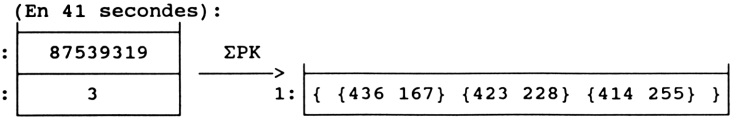
Exemples:

(En trois secondes):



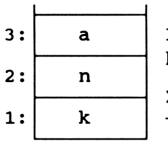
En effet: $1000 = 30^2+10^2 = 26^2+18^2$.

(En 41 secondes):



En effet: 87539319 = 436^3+167^3 = 423^3+228^3 = 414^3+255^3.

Exercice 23: (*****) Décompositions d'un entier en une somme de n puissances k-èmes.



Le programme **'ΣNPK'** cherche les décompositions possibles d'un entier a comme somme de n puissances k-èmes.
Les différentes solutions sont placées dans une liste au niveau 1 de la pile.

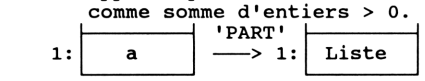
Exemple: avec a=100, n=4, k=2, on obtient la liste
{ {1 1 7 7} {1 3 3 9} {1 5 5 7} {2 4 4 8} {5 5 5 5} }
En effet, par exemple, $100 = 2^2+4^2+4^2+8^2$.

Supposons que a s'écrive comme une somme de n puissances k-èmes:
a = (i1)^k + (i2)^k + (i3)^k + ... + (in)^k, avec
1 ≤ i1 ≤ i2 ≤ i3 ≤ ... ≤ in.
On a alors: a ≥ n*[(i1)^k], c'est-à-dire i1 ≤ (a/n)^(1/k).
Il suffira donc d'étudier, dans une boucle ascendante, toutes les valeurs possibles de i1, de 1 jusqu'à (a/n)^(1/k).
Notons i, l'une de ces valeurs:
On calcule b=a-i^k. Le problème est alors d'écrire b comme une somme de n-1 puissances k-èmes, et cette idée débouche sur un algorithme récursif.
Il faut éviter qu'une même solution ne soit trouvée 2 fois. on décompose donc b=a-i^k en une somme de n-1 puissances k-èmes, du type j^k, avec j ≥ i.

Texte du programme 'ΣNPK'	Commentaires
`« 0 → sp`	Initialise variable locale 'sp'.
`« « → a n k m`	Début sous-programme de décomp. d'un entier a en somme de n puissances j^k, avec j≥m.
`« { }`	Initialise liste solutions.
`IF n 1 > THEN`	Si n est supérieur à 1:
`m`	De i=m
`a n / k XROOT`	à i=(a/n)^(1/k).
`FOR i`	
`a i k ^ -`	b=a-i^k.
`n 1 - k i`	On cherche à décomposer b en une Σ de n-1 puiss. j^k avec j≥i.
`sp EVAL`	
`IF DUP SIZE THEN`	S'il y a des solutions:
`1 OVER SIZE FOR j`	On ajoute i au début de
`DUP j GET i 1 →LIST`	chacune de ces solutions
`SWAP + j SWAP PUT`	
`NEXT`	
`+`	Ajouter à liste totale.
`ELSE DROP END`	Sinon effacer solut. vide.
`NEXT`	Vers valeur suivante de i....
`ELSE`	Sinon
`IF a DUP k XROOT`	Si a est lui-même une puissance
`FLOOR DUP k ^ ROT ==`	k-ème, l'ajouter à la liste.
`THEN + ELSE DROP END`	Sinon l'effacer.
`END`	
`» 'sp' STO`	
`1 sp EVAL`	Appeler le sous-programme en lui donnant un minimum initial de 1.
`» »`	

Exercice 24: (*****) Partitions d'un nombre entier.

On appelle partition d'un nombre entier a>0 toute écriture de a comme somme d'entiers > 0.



Le programme **PART** donne la liste de toutes les partitions possibles de l'entier n. Chacune de ces par--titions est elle-même une liste d'entiers croissants.

Exemple: avec n=5, on obtient la liste

{ {1 1 1 1 1} {1 1 1 2} {1 1 3} {1 2 2} {1 4} {2 3} {5} }

Là encore il faut créer une boucle portant sur un entier i (i allant de 1 à n/2) l'idée étant, à i fixé, d'ajouter i aux partitions de l'entier n-i (d'où un algorithme récursif).

De même que dans l'exercice précédent, il faut éviter de trouver plusieurs fois la même solution (à l'ordre près des termes). Pour celà les partitions de n-i devront être cherchées au moyen d'entiers j≥i.

Texte du programme 'PART'	Commentaires
`« 0 → sp`	Initialise variable locale 'sp'.
`« « → a i`	Début ss-pg cherchant partitions de a avec entiers j≥i.
`« { }`	Initialise liste solutions.
`IF i a 2 / ≤`	Si i≤(a/2) alors:
`THEN`	
`i a 2 / FOR j`	de j=i à j=a/2,
`a j - j`	Chercher partitions de a-j
`sp EVAL`	avec entiers ≥ j.
`1 OVER SIZE FOR k`	Ajouter j au début des
`j OVER k GET`	partitions obtenues
`+ k SWAP PUT`	pour a-j.
`NEXT`	
`+`	Ajouter à liste des parti-
	tions de a.
`NEXT`	Valeur suivante de j...
`END`	
`a 1 →LIST`	Ajouter { a } comme partition
`1 →LIST +`	particulière de a.
`»`	
`»`	
`'sp' STO`	Met le sous-programme dans la variable locale 'sp'.
`1 sp EVAL`	Demande partitions de a, avec entiers ≥ 1.
`»`	
`»`	

Remarque:

Comme dans l'exercice précédent, on utilise un sous-programme s'appelant lui-même (récursivité).

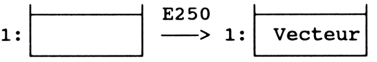
On comprend pourquoi il est nécessaire de créer une variable 'sp' ayant un contenu fictif au départ (ici 0).

En effet une structure telle que:

« « .... sp EVAL ... » → sp « ... ne peut convenir.

(simplement parce que le nom 'sp' n'est pas connu à l'intérieur du sous-programme ainsi défini).

Exercice 25: (*****) Les 250 premières décimales de e.



Le programme **'E250'** place dans un vecteur les 250 premières décimales du nombre e=exp(1). Le vecteur est formé de 28 termes qui contiennent chacun une tranche de 9 décimales (dans chaque tranche, le ou les zéros éventuels au début ne sont pas affichés mais ils existent...).

On considère la suite Un = 1 + 1/1! + 1/2! + 1/3! + ... + 1/n!
On montre que les suites de termes généraux Un et Un +1/(n*n!) sont adjacentes et convergent toutes deux vers le nombre e.

En particulier, pour tout n: | Un - e | < 1/(n*n!).

Pour n = 144, 1/(n*n!) ≈ 1.25*10^(-252). Les deux nombres U(144) et e ont donc les mêmes 251 premières décimales.
Le problème devient donc: calculer les 250 premières décimales de U(144), de façon exacte.

Pour celà, on considère la suite V(0), V(1), V(2), ..., V(144), définie par récurrence, par:
V(0) = 1 et pour tout k (0≤k≤143), V(k+1)=1+V(k)/(144-k).

Ainsi V(1) = 1+1/144, V(2) = 1 + 1/143 + 1/(143*144) , etc....
et enfin:
V(143) = 1 + 1/2! + 1/3! + ... + 1/144!.
V(144) = 1 + V(143) = U(144).

On va représenter les termes V(0), V(1), ... ,V(144) (ou du moins leur partie décimale) sur la pile par leur représentation en base 10^9 (c'est à dire en tranches de 9 décimales: la raison de ce choix apparaît plus loin).

Il faut prévoir 28 tranches de 9 décimales (28*9=252).
Le niveau 28 accueille la tranche des neuf premiers chiffres (sous forme d'un entier compris entre 0 et 999999999).
Le niveau 27 accueille la tranche des neuf chiffres suivants.
etc...
Le niveau 1 accueille la tranche des neuf derniers chiffres.

Initialement, la partie décimale de V(0) est nulle, et il faut donc placer des zéros aux 28 premiers niveaux de la pile.

Ensuite, il faut entrer dans une boucle, pour un indice k variant de 144 à 2 (avec un pas de -1), une étape de cette boucle consistant à transformer la partie décimale du réel V en celle du réel 1+V/k.
A la sortie de cette boucle, on obtient la partie décimale de V(143), qui est aussi celle de V(144), et donc celle de U(144).
Il ne reste plus alors qu'à grouper dans un vecteur les entiers placés aux niveaux 28,27,....,1 de la pile.
Dans ce vecteur où apparaissent 28 entiers compris entre 0 et 999999999, chaque entier représente une tranche de 9 décimales (attention: les zéros qui peuvent débuter une telle tranche n'apparaissent pas mais ils existent).
La dernière tranche contient les décimales N° 244 à 252. Les deux derniers chiffres ne sont donc pas à prendre en compte ici.

Texte du programme 'E250'	Commentaires
`« 29 144`	t=nombre de tranches + 1.
`→ t n`	n= indice du terme U à calculer.
`« 2 t START 0 NEXT`	Des zéros aux niveaux 1,2,..,28.
`n 2 FOR k`	Entrée dans la boucle, où on passe de V à 1+V/k.
`1`	La partie entière de V.
`2 t START`	Début division par k.
`1000000000 *`	Report dans la division.
`t ROLL`	Tranche suivante.
`+`	Ajouter au report.
`k`	On va diviser par k.
`DUP2 / IP`	Quotient entier de la tranche par k.
`3 ROLLD`	
`MOD`	Reste dans la division de la tranche par k.
`NEXT`	Passer à la tranche suivante.
`DROP`	Eliminer le dernier reste.
`-1 STEP`	Passer à l'entier k suivant.
`t 1 - →ARRY`	Vecteur de 28 tranches.
`»`	
`»`	

Remarque: On constate que la division "par tranches" comporte une multiplication du reste "r" de la division précédente par 1000000000,que ce résultat est ajouté à la tranche suivante, elle même comprise entre 0 et 999999999.
Le reste "r" est, selon le cas:
* initialement la partie entière de V, c'est-à-dire 1.
* Ensuite, un reste dans une division par un entier k compris entre 2 et 144.

dans tous les cas, "r" est inférieur à 144.
Les opérations décrites ci-dessus se font donc sans perte de précision, puisque la HP48SX travaille sur 12 termes significatifs.
On comprend ainsi pourquoi il a été possible de travailler avec des tranches de 9 décimales.

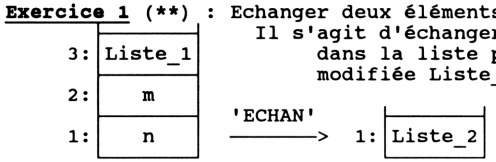
Le résultat est obtenu en 2 mn 46 s:

```
[ 718281828   459045235   360287471   352662497   757247093
  699959574   966967627   724076630   353547594   571382178
  525166427   427466391   932003059   921817413   596629043
  572900334   295260595   630738132   328627943   490763233
  829880753   195251019   011573834   187930702   154089149
  934884167   509244761   460668081   ]
```

Les deux derniers chiffres (soulignés) sont inexacts (çà n'est pas surprenant).

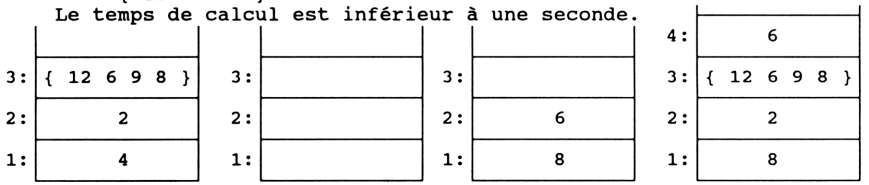
LES LISTES : Corrigés des Exercices

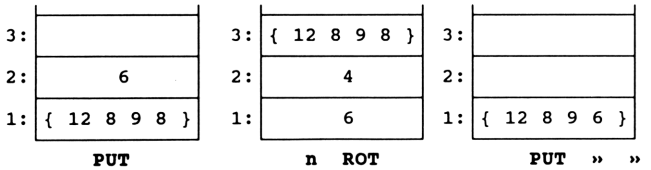
Exercice 1 (**) : Echanger deux éléments dans une liste.
Il s'agit d'échanger les éléments d'indice m et n dans la liste présente au niveau 3. La liste modifiée Liste_2 est obtenue au niveau 1.



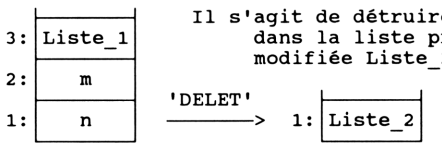
Texte du Programme 'ECHAN'	Commentaires
« → L m n « L m GET L n GET L m ROT PUT n ROT PUT » »	L : la liste m et n : positions des éléments à échanger. On place les deux éléments sur la pile. Chacun des 2 éléments est placé à la position qu'occupait l'autre.

Remarque:
Une erreur se produit si les indices m et n ne sont pas compris entre 1 et la taille de la liste à modifier.
Exemple:
On veut échanger les éléments d'indices 2 et 4 de la liste { 12 6 9 8 }
Le temps de calcul est inférieur à une seconde.





Exercice 2 (**) : Détruire une partie d'une liste.

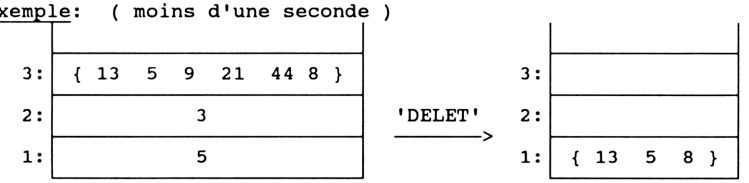


Il s'agit de détruire les éléments d'indice m à n dans la liste présente au niveau 3. La liste modifiée Liste_2 est obtenue au niveau 1.

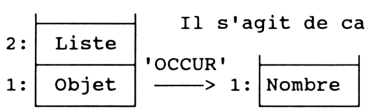
Texte du Programme 'DELET'	Commentaires
« IF DUP2 ≤ THEN → L m n « L 1 m 1 - SUB L n 1 + L SIZE SUB + » ELSE DROP2 END »	On teste si m ≤ n. Si m ≤ n, on crée les variables locales L (liste), m et n, on crée la sous-liste des m-1 premiers élements, on ajoute la sous-liste des objets postérieurs au n-ième Si m < n, on conserve la liste initiale.

Remarques:
Si m < n, le programme **'DELET'** ne modifie pas la liste.
Si m = n, seul le n-ième élément de la liste est supprimé.
Si m < 1, l'effet est le même que si m=1.
Si n > taille(Liste), l'effet est le même que si n=taille(Liste).

Exemple: (moins d'une seconde)



Exercice 3 (***###***) : Nombre d'occurences d'un objet dans une Liste.

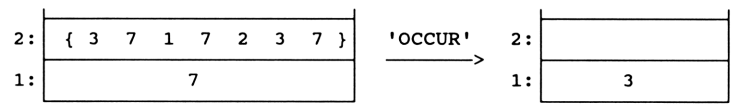


Il s'agit de calculer le nombre de fois qu'un objet apparaît dans une liste.
Le résultat, ici appelé 'Nombre', vaut 0 si l'objet n'appartient pas à la liste.

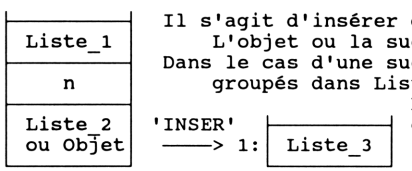
L'idée du programme **'OCCUR'** est de créer un compteur (placé au niveau 3 et initialisé à 0), auquel on ajoute un à chaque fois qu'une occurence de 'Objet' est rencontrée dans 'Liste'.
Pour éviter le problème lié au fait que l'instruction **POS**, qui détecte la présence d'un objet dans une liste, n'en détecte en fait que la première occurence, on détruit chaque occurence détectée de 'Objet' dans 'Liste', au moyen du programme **'DELET'**.

Texte du programme 'OCCUR'	Commentaires
« 0 3 ROLLD WHILE DUP2 POS DUP REPEAT ROT SWAP DUP DELET ROT 1 + 3 ROLLD SWAP END 3 DROPN »	Compteur initialisé à 0, placé au niveau 3. Tant qu''Objet' est dans 'Liste': * détruire l'occurence détectée au moyen de 'DELET'. * incrémenter le compteur. Quand 'Objet' n'est plus détecté dans 'Liste', on efface la pile à l'exception du compteur.

Exemple: (en une seconde)



Exercice 4 (**) :Insérer un objet ou une succession d'objets dans une liste.



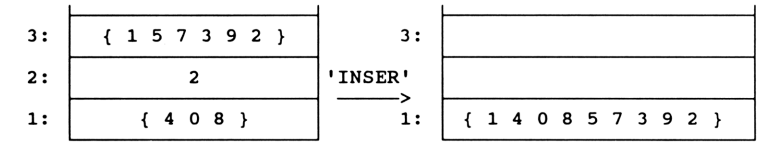
Il s'agit d'insérer dans Liste_1, à la position n, L'objet ou la succession d'objets du niveau 1. Dans le cas d'une succession d'objets ceux-ci sont groupés dans Liste_2.
La liste modifiée Liste_3 est obtenue au niveau 1.

Attention: pour insérer par exemple l'objet { 4 5 } dans Liste_1, il faut placer { { 4 5 } } au niveau 1. Si on place { 4 5 } au niveau 1, alors les deux éléments 4 et 5 sont insérés successivement dans Liste_1, à partir de la position n.

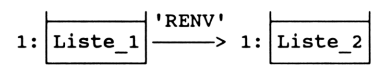
Le principe de **'INSER'** est simple:On coupe la liste Liste_1 en 2 parties, au niveau de la position d'insertion souhaitée, on ajoute l'objet à insérer à la fin de la 1ère partie, avant de compléter par la deuxième partie.

Texte du programme 'INSER'	Commentaires
« → L m a « L 1 m 1 - SUB a + L m L SIZE SUB + » »	L:liste, m:position, a:objet. On crée la sous-liste formée des m-1 premiers éléments. On lui ajoute l'objet a . On crée la sous-liste formée des éléments à partir du m-ième. On l'ajoute à la liste formée précédemment.

Exemple: (En moins d'une seconde)



Exercice 5 (*) : Inverser l'ordre des éléments dans une liste.



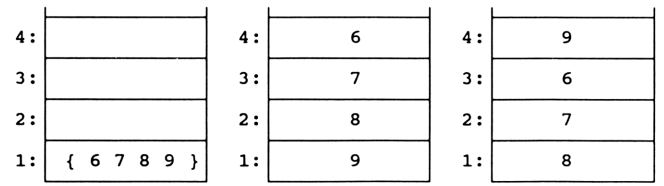
On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des mêmes éléments, mais dans l'ordre contraire.

Il suffit ici de répartir les objets de Liste_1 sur la pile, puis de les élever dans celle-ci, de la manière suivante:

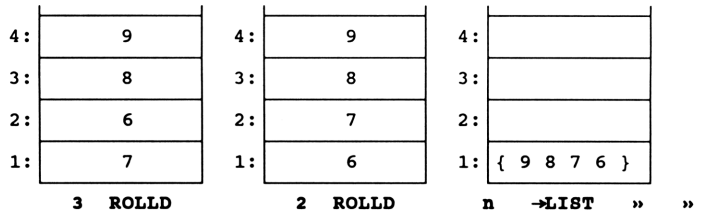
* L'objet situé au niveau 1 est envoyé au niveau n.
* L'objet situé initialement au niveau 2 (et qui vient juste de descendre au niveau 1) est envoyé au niveau n-1,
* etc...
* A la fin, l'objet situé initialement au niveau n-1, et qui petit à petit est descendu au niveau 1, est placé au niveau 2, sautant ainsi par dessus l'objet situé au départ au niveau n et qui maintenant se trouve au niveau 1.

Texte du Programme 'RENV'	Commentaires
« OBJ→ → n « n 2 FOR i i ROLLD -1 STEP n →LIST » »	On place sur la pile les objets de la liste. n est la longueur de la liste. Début de boucle. Les éléments remontent sur la pile, de façon à inverser leurs positions respectives. Fin de boucle. On reconstitue la liste.

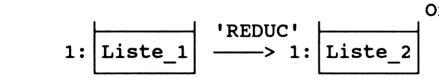
Exemple: Voyons en détail l'inversion de { 6 7 8 9 }.



« OBJ→ → n « 4 ROLLD



Exercice 6 (***###***) : Eliminer les occurences multiples dans une liste.



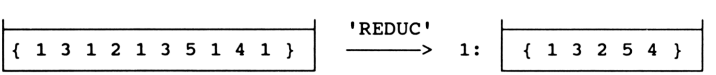
On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant qu'un exemplaire des différents éléments de Liste_1.

L'idée est ici de créer, petit à petit, Liste_2, à partir d'une liste vide: on lit chaque élément de Liste_1, et on l'ajoute à Liste_2 à condition qu'il n'y figure pas déjà.

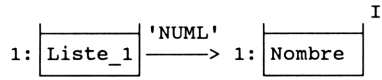
Texte du programme 'REDUC'	Commentaires
« RENV OBJ→ { } 1 ROT START SWAP DUP2 POS IF THEN DROP ELSE + END NEXT »	On inverse l'ordre des objets de la liste, et on les répartit sur la pile. Va devenir la liste Liste_2. Début de boucle (de 1 à taille de Liste_1) L'objet est-il dans Liste_2 ? Si oui on le détruit, Sinon on l'ajoute à Liste_2. Fin de boucle.

Remarque:
L'appel à **'RENV'** (inversion de l'ordre des éléments de la liste est utile pour que les éléments de Liste_2 apparaissent dans l'ordre où ils apparaissaient (pour la première fois) dans Liste_1.

Exemple: (en une seconde)



Exercice 7 (#) : Nombre d'objets différents dans une liste.

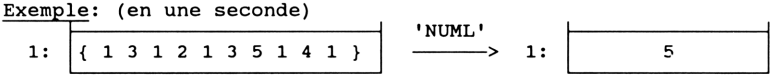


Il s'agit ici de calculer le nombre d' objets réellement différents dans Liste_1. Si un objet apparait plusieurs fois dans Liste_1, il n' est donc compté qu'une seule fois.

La solution est évidente: Il suffit d'appeler **'REDUC'**, qui ne garde qu'un exemplaire de chaque élément de Liste_1, et de calculer la taille de la liste obtenue, par **SIZE**.

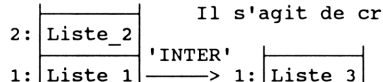
Texte du programme 'NUML'	Commentaires
« REDUC SIZE »	No comment...

Exemple: (en une seconde)



car les éléments différents de Liste_1 sont ici 1,3,2,5,4.

Exercice 8 (###) : Créer la liste des éléments communs à deux listes.



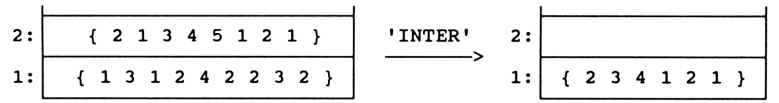
Il s'agit de créer la liste Liste_3 formée des objets qui apparaissent à la fois fois dans Liste_1 et dans Liste_2. Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

L'idée est ici de créer, petit à petit, Liste_3, à partir d'une liste vide: On lit chaque objet 'a' de Liste_2.

Si 'a' est détecté dans Liste_1, on ajoute 'a' à Liste_3 et on détruit, dans Liste_1, cette occurence de l'objet 'a' (utilisation du programme **'DELET'**). Cet effacement est nécessaire pour donner aux différentes occurences de 'a' dans Liste_1 des rôles différents.

Texte du programme 'INTER'	Commentaires
« → L1 « OBJ→ { } SWAP IF DUP THEN L1 1 ROT START ROT DUP2 POS IF DUP THEN ROT SWAP DUP DELET 3 ROLLD SWAP + SWAP ELSE DROP2 END NEXT END DROP » »	L1 = Liste_1 On place les objets de Liste_2 sur la pile. { } va devenir Liste_3. Si Liste_2 n'est pas vide: Pour chaque objet de Liste_2: Appartient-il à Liste_1 ? Si oui, on le retire de Liste_1 et on l'ajoute à Liste_3. Sinon, on clarifie la pile. Fin de la boucle. Fin de test. On efface ce qu'il reste de la liste Liste_1 (Si Liste_2 était non vide) ou 0.

Exemple: (en moins de deux secondes)



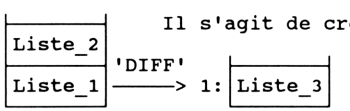
Remarque:
Dans le programme ci-dessus, Liste_2 est 'répartie' sur la pile, Liste_3 s'y constitue petit à petit. Il est important que vous éxécutiez ce programme pas à pas pour comprendre comment les différents objets à gérer évoluent sur la pile.

On pourrait écrire une version plus lisible de **'INTER'** en créant une variable locale contenant Liste_2:

```
«  →  L2  L1
   «  { }
      IF  L2  SIZE  THEN
          1  L2  SIZE  FOR  k
             L1  L2  k  GET  POS  DUP
             IF  THEN
                 L1  SWAP  DUP  DELET  'L1'  STO
                 L2  k  GET  +
             ELSE  DROP  END
          NEXT
      END
   »
»
```

Dans cette version, l'ordre des éléments obtenus dans Liste_3 est différent de celui obtenu avec la première version:
Sur l'exemple ci-dessus, on obtient { 2 1 3 4 1 2 }.

Exercice 9 (###) : Produire la soustraction de deux listes.



Il s'agit de créer la liste Liste_3 formée des objets qui appartiennent à Liste_2 mais pas à Liste_1.
Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

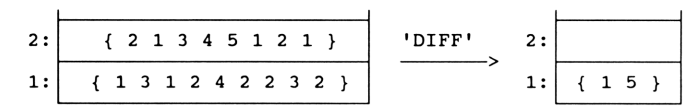
L'idée est ici de créer, petit à petit, Liste_3, à partir d'une liste vide:
On lit successivement chaque objet 'a' de Liste_2.
Si 'a' n'est pas détecté dans Liste_1, on l'ajoute à Liste_3 sinon on détruit, dans Liste_1, cette occurence de 'a'. (utilisation du programme **'DELET'**). Cet effacement est nécessaire pour donner aux différentes occurences de 'a' dans Liste_1 des rôles différents.

On constate que l'algorithme utilisé est très proche de celui du programme **'INTER'**. Dans les deux listings, les différences entre **'INTER'** et **'DIFF'** sont d'ailleurs très minimes.

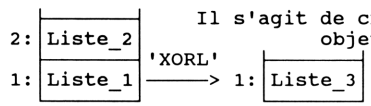
Texte du programme 'DIFF'	Commentaires

`« → L1`
`« OBJ→`
`{ }`
`SWAP`
`IF DUP THEN`
`L1 1 ROT START`
`ROT DUP2 POS`
`IF DUP THEN`
`ROT SWAP DUP`
`DELET SWAP DROP`
`ELSE`
`DROP ROT + SWAP`
`END`
`NEXT`
`END`
`DROP`
`»`
`»`
	L1 = Liste_1
	On place les objets de Liste_2 sur la pile. { } va devenir Liste_3.
	Si Liste_2 n'est pas vide:
	Pour chaque objet de Liste_2:
	Appartient-il à Liste_1 ?
	Si oui, on le retire de Liste_1 et on l'enlève de la pile.
	
	Sinon, on l'ajoute à Liste_3
	
	Fin de la boucle.
	Fin de test.
	On efface ce qu'il reste de la liste Liste_1 (Si Liste_2 était non vide) ou 0.

Exemple: (en moins de deux secondes)



Exercice 10 (**##**) :Créer la liste des éléments appartenant à une seule de deux listes.



Il s'agit de créer la liste Liste_3 formée des objets qui apparaissent dans une seule des 2 Listes Liste_1 et Liste_2. Si un même objet est présent plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

Le problème est ici assez simple, en utilisant le programme **'DIFF'** (Exercice 9):

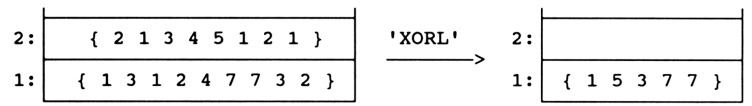
* On crée la liste Liste_4 des éléments de Liste_2 qui ne sont pas dans Liste_1.

* On crée la liste Liste_5 des éléments de Liste_1 qui ne sont pas dans Liste_2.

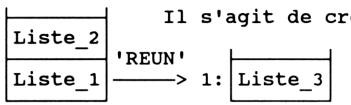
* On ajoute les deux listes obtenues, par concaténation.

Texte du programme 'XORL'	Commentaires
« DUP2 DIFF SWAP ROT DIFF + »	On duplique Liste_2 et Liste_1. On crée (Liste_2 - Liste_1). On la place au niveau 3, et Liste_1 au dessus de Liste_2. On crée (Liste_1 - Liste_2). On ajoute les deux listes ainsi obtenues.

Exemple: (en deux secondes)



Exercice 11 (#) : Créer la réunion ensembliste de deux listes.



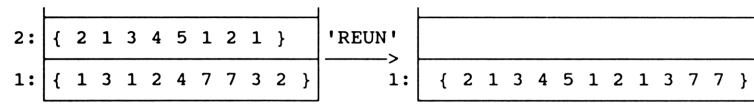
Il s'agit de créer la liste Liste_3 formée des objets qui appartiennent au moins à une des deux listes Liste_1 et Liste_2. Si un même objet apparait m fois dans Liste_1 et n fois dans Liste_2, alors il doit apparaître max(m,n) fois dans Liste_3.

Il est clair qu'une simple concaténation de Liste_2 et de Liste_1 ne répond pas à la question (sauf si Liste_1 et Liste_2 n'ont aucun élément en commun).
Là encore, la solution est très simple, en utilisant **'DIFF'**:

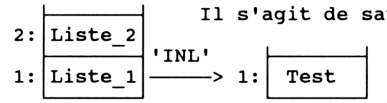
* On crée la liste Liste_4 des éléments de Liste_1 qui ne sont pas dans Liste_2.
* On ajoute (par concaténation) Liste_2 et Liste_4.

Texte du programme 'REUN'	Commentaires
« OVER DIFF + »	Liste_2 sous Liste_1. On crée (Liste_1 - Liste_2). On ajoute Liste_2 à la liste qui vient d'être créée.

Exemple: (en moins de deux secondes)



Exercice 12 (#) : Une liste est-elle incluse dans une autre liste?.

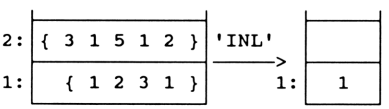


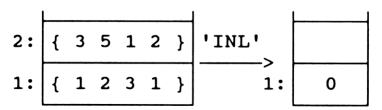
Il s'agit de savoir si Liste_1 est incluse dans Liste_2.
La réponse, nommée ici 'Test' vaut 1 si la réponse est oui, 0 sinon.
Si un même objet apparaît plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

La solution est basée sur l'observation suivante:
Dire que Liste_1 est incluse dans Liste_2, c'est dire que la liste (Liste_1 - Liste_2) obtenue par **'DIFF'** est la liste vide (autrement dit, il n'y a aucun objet qui soit dans Liste_1 et qui ne soit pas dans Liste_2).

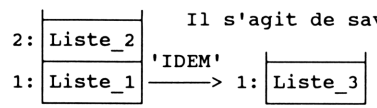
Texte du programme 'INL'	Commentaires
« SWAP DIFF SIZE NOT »	Liste_2 sous Liste_1. (Liste_1 - Liste_2) Taille de cette liste. 1 si taille nulle, 0 sinon.

Exemple: (en moins d'une seconde)





Exercice 13 (#) : Deux listes sont-elles égales, d'un point de vue ensembliste?

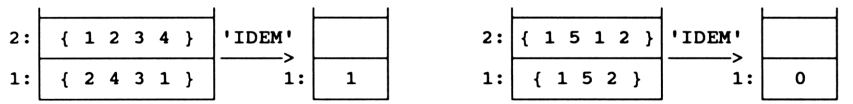


Il s'agit de savoir si Liste_1 est formée des mêmes objets que Liste_2.
La réponse, nommée ici 'Test' vaut 1 si la réponse est oui, 0 sinon.
Si un même objet apparaît plusieurs fois dans une liste, ses différentes occurences doivent être considérées comme autant d'objets différents.

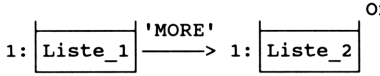
Il est clair que l'instruction **SAME** (ou **==**) ne répond pas à la question. En effet cette instruction considère par exemple que les listes { 1 2 3 4 } et { 2 4 3 1 } sont différentes alors que, du point de vue ensembliste qui nous occupe ici, elles sont égales.
La réponse est très simple en utilisant le programme **'XORL'** (exercice 10): il suffit d'appliquer **'XORL'** aux deux listes considérées et de tester si la liste obtenue est vide ou non. On obtient une liste vide si et seulement si Liste_1 et Liste_2 sont égales d'un point de vue ensembliste.

Texte du programme 'IDEM'	Commentaires
« XORL SIZE NOT »	Liste des objets qui sont dans 1 (et une seule) des 2 listes. Taille de cette liste. 1 si taille nulle, 0 sinon.

Exemple: (en 1 à 2 secondes)



Exercice 14 (#) : Objets présents au moins deux fois dans une liste.



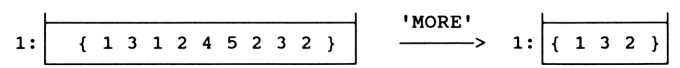
On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant que les objets qui apparaissent au moins 2 fois dans Liste_1 (chacun d'eux n'étant reproduit qu'une fois).

La solution utilise **'REDUC'** (exercice 6) et **'DIFF'** (exercice 9):

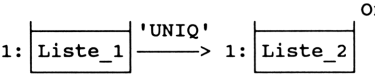
* On crée une liste en éliminant de 'Liste_1' toutes les occurences multiples, au moyen de **'REDUC'**: Chaque élément de Liste_1 est conservé une seule fois.
* On soustrait la liste obtenue de 'Liste_1', au moyen de **'DIFF'**. On obtient ainsi la liste des objets figurant au moins 2 fois dans Liste_1. Mais, dans cette liste, il peut y avoir des répétitions (de la part d'éléments qui figuraient au moins trois fois dans 'Liste_1').
* On applique **'REDUC'** à la liste obtenue, ce qui donne le résultat souhaité.

Texte du programme 'MORE'	Commentaires
« DUP REDUC DIFF REDUC »	On duplique Liste_1. On élimine les répétitions. On soustrait la liste obtenue de Liste_1. On élimine les répétitions éven-tuelles de cette dernière liste.

Exemple: (en 2 secondes)



Exercice 15 (**##**) : Objets présents une seule fois dans une liste.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 obtenue en ne conservant que les objets qui apparaissent une seule fois dans Liste_1.

Comme dans l'exercice précédent, la solution utilise **'REDUC'** et **'DIFF'**:

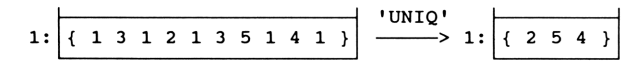
* On crée une liste Liste_3 en éliminant les répétitions dans 'Liste_1' au moyen de **'REDUC'**: chaque objet de Liste_1 est conservé une seule fois.

* On soustrait 'Liste_3' de 'Liste_1', au moyen de **'DIFF'**. On obtient la liste des objets figurant au moins 2 fois dans Liste_1 (Dans cette liste, il peut y avoir des répétitions provenant des éléments qui figuraient au moins trois fois dans 'Liste_1').

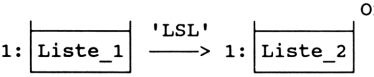
* La liste qui vient d'être obtenue est alors soustraite de Liste_3.

Texte du programme 'UNIQ'	Commentaires
« DUP	On duplique Liste_1.
REDUC	Eliminer répétitions (Liste_3).
SWAP	Liste_3 au dessus de Liste_1, au
OVER	dessus de Liste_3.
DIFF	Liste des répétitions.
DIFF	Liste des objets uniques.
»	

Exemple: (en 2 secondes)



Exercice 16 (********) : Liste des sous-listes d'une liste donnée.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée de toutes les sous-listes de Liste_1 (y compris la liste vide { }).

Si la taille de Liste_1 est n, celle de Liste_2 est 2^n.

L'idée qui conduit à la solution tient à la remarque suivante:

* Le résultat est évident si Liste_1 est vide.
* Sinon, une sous_liste quelconque de Liste_1 contient ou ne contient pas le n-ème et dernier élément de Liste_1. On est ainsi amené à considérer les sous_listes d'une liste contenant initialement n-1 éléments.

Ce raisonnement (élémentaire) débouche naturellement sur un algorithme récursif:
Soit Liste_1 une liste de n objets. On veut créer la liste Liste_2 de toutes les sous-listes de Liste_1.
* Si n = 0, alors Liste_2 est égale à { { } } (La seule sous-liste de Liste_1 est en effet la liste vide).
* Si n > 0: soit Liste'_1 la liste formée des n-1 premiers éléments de Liste_1.
  On crée la liste Liste'_2 des sous-listes de Liste'_1.
  On crée Liste"_2 en ajoutant le n-ième élément de Liste_1 à chaque élément de Liste'_2.
  On effectue la concaténation de Liste'_2 et Liste"_2.
* Dans tous les cas, on a obtenu Liste_2.

Texte du programme 'LSL' (Version récursive)	Commentaires

« IF DUP { } ==
THEN
1 →LIST
ELSE
DUP SIZE
DUP2 GET
IF DUP TYPE 5 == THEN
1 →LIST
END
→ a
« 1 - 1 SWAP SUB
LSL
 	
DUP 1 OVER SIZE FOR k
DUP k GET a +
k SWAP PUT
NEXT
+
»
END
»
	Liste_1 vide ?
	
	Si oui: placer {{}} sur la pile.
	Sinon:
	Taille de Liste_1.
	Dernier objet de Liste_1.
	Si c'est une liste, le mettre entre { et }.
	a = dernier objet de Liste_1.
	
	On a enlevé 'a' de Liste_1.
	On calcule les sous-listes de la liste des n-1 premiers objets de Liste_1.
	On duplique Liste'_2.
	On ajoute 'a' à tous les objets de Liste'_2.On obtient alors Liste"_2.
	On ajoute Liste'_2 et Liste"_2.
	
	Fin du cas "Liste_1 non vide".

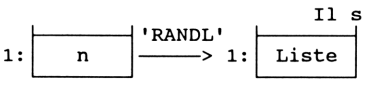
N.B: le dernier élément de Liste_1 peut être lui-même une liste. Avant de l'ajouter aux objets de la liste Liste'_2, il est nécessaire de le placer entre les caractères { et }.
C'est le sens du test " **IF DUP TYPE 5 == THEN ...**".

Version non récursive de 'LSL': Il est très facile de modifier **'LSL'** pour en donner une version itérative (non récursive). Les deux programmes sont très proches: la différence la plus notable tient dans la nécessité de créer une boucle externe contrôlant le parcours de Liste_1.
Supposons par exemple que Liste_1 soit égale à { 1 2 3 }.
* On commence par mettre { { } } sur la pile.
* On traite le premier élément de Liste_1, pour aboutir à la liste { { } { 1 } }.
* On traite le deuxième élément de Liste_1,pour aboutir à la liste { { } { 1 } { 2 } { 1 2 } }.
* On traite le troisième élément de Liste_1, pour obtenir la liste:
  { { } {1} {2} {1 2} {3} {1 3} {2 3} {1 2 3} }.

Texte du programme 'LSL' (Version itérative)	Commentaires
« DUP SIZE → L n « { { } } IF n THEN 1 n FOR j L j GET IF DUP TYPE 5 == THEN 1 →LIST END → a « DUP 1 OVER SIZE FOR k DUP k GET a + k SWAP PUT NEXT » + NEXT END » »	L=Liste_1 et n=taille(Liste_1) Si Liste_1 est non vide: Début boucle externe. j-ème élément de Liste_1. Si c'est une liste, le mettre entre { et }. a = dernier objet de Liste_1. Dupliquer liste du niveau 1. On va parcourir cette liste. On ajoute 'a' à tous les objets de cette liste. Ajouter listes niveaux 1 et 2. Fin boucle externe. Fin du cas "Liste_1 non vide".

Comparaison des deux programmes:
'LSL'-récursif occupe 169 octets en mémoire.
'LSL'-itératif en occupe 181,5.
Les deux programmes ont été testés sur la liste {1 2 ... 7}.
On obtient les temps de calcul suivants (la liste obtenue est de taille 128):
'LSL'-récursif: 31 s 'LSL'-itératif: 31 s
On ne peut donc pas départager facilement ces deux versions!

Exercice 17 (*) : Création d'une liste pseudo-aléatoire d'entiers.



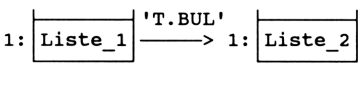
Il s'agit de créer une liste aléatoire, de longueur n, formée d'entiers compris entre 0 et 999.
Une telle liste sera utile pour tester les procédures de tri des exercices 18, 19, 20.

Texte du programme 'RANDL'	Commentaires
« → n « 1 n START RAND 1000 * FLOOR NEXT n →LIST » »	n = longueur de la liste. Début de boucle. Réel aléatoire 0<x<1. Réel aléatoire 0<x<1000. Entier aléatoire 0≤x≤999. Fin de boucle. On met ces n entiers dans une liste.

Remarques:
Une liste aléatoire de 500 entiers est ainsi créée en 9 s.

La commande **RDZ** permet de créer des listes pseudo-aléatoires identiques (ce qui sera utile pour tester la rapidité des méthodes de tri).
Si on fait, par exemple: **1 RDZ 100 RANDL** , alors on obtient toujours la même liste aléatoire de 100 entiers.

Exercice 18 (***) : Tri d'une liste par la méthode de la bulle.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode du "tri-bulle".

Tout d'abord, si la longueur de Liste_1 est inférieure à 2, il n'y a rien à modifier.

Sinon, il faut prévoir deux boucles:

* Une boucle externe indiquant la longueur de la sous-liste à parcourir (Indice i, variant en décroissant de la taille de liste_1 à 1).

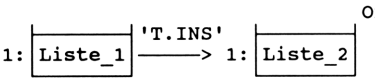
* Une boucle interne effectuant le parcours dans cette sous liste. On lit les deux éléments d'indice j et j+1 et on les échange s'ils sont mal placés (Indice j variant de 1 à i-1).

(Voir plus loin la comparaison des différentes méthodes de tri).

Texte du Programme 'T.BUL'	Commentaires

« DUP SIZE
IF DUP 1 > THEN
2 FOR i
 	
1 i 1 - FOR j
 	
j GET
OVER j 1 + GET
IF DUP2 > THEN
ROT j ROT PUT
j 1 + ROT PUT
ELSE DROP2 END
NEXT
-1 STEP
 	
ELSE DROP END
»
	On duplique 'Liste_1'.
	Si elle contient au moins 2 élts:
	Boucle externe (i=longueur de la sous-liste à parcourir).
	Boucle interne (j=position dans la sous-liste).
	Liste_1(j).
	Liste_1(j+1).
	Si mal placés, on les échange.
	
	Sinon, clarifier pile.
	On continue dans sous-liste
	Le i-ème objet est maintenant bien placé.
	Si taille(Liste_1)<2, on conserve Liste_1.

Exercice 19 (****) : Tri d'une liste par la méthode d'insertion.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode d'insertion.

(Voir plus loin la comparaison des différentes méthodes de tri).

Tout d'abord, si la longueur de Liste_1 est inférieure à 2, il n'y a rien à modifier.

Sinon, il faut prévoir une boucle externe correspondant à la recherche de la position d'insertion du i-ème élémént (i est un indice croissant de 2 à n).

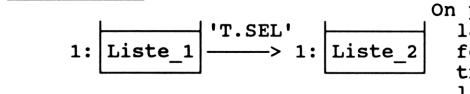
On recherche la position où placer ce i-ème élément. Pour celà on utilise un indice j, dont la valeur initiale est i-1. On décrémente j tant que l'élément d'indice j est supérieur à l'objet qu'on veut insérer (qui est Liste_1(i)).

On arrête cette décrémentation quand on est arrivé en début de liste ou quand on a trouvé un élément inférieur ou égal à Liste_1(i). Ce dernier doit alors être inséré après le j-ème élément de Liste_1.

Texte du programme 'T.INS'	Commentaires
« DUP SIZE IF DUP 1 > THEN → n « 2 n FOR i DUP i GET i 1 - → aide j « WHILE DUP j 1 MAX GET aide > j 1 ≥ AND REPEAT 'j' 1 STO- END IF j i 1 - < THEN DUP 1 j SUB aide + OVER j 1 + i 1 - SUB + OVER i 1 + n SUB + SWAP DROP END » NEXT » ELSE DROP END »	On duplique 'Liste_1'. Si taille(Liste_1) ≥ 2, alors: Boucle externe (Placer le i-ème objet). aide=Liste_1(i), et j indice décroissant pour recherche. Tant que l'indice j est ≥ 1 et que Liste_1(j) > aide, (remarquer j 1 MAX pour éviter l'erreur 0 GET) décrémenter j. Si j (Position après laquelle insérer Liste_1(i)), est < (i-1), alors il faut effectivement insérer Liste_1(i). On vient d'insérer le i-ème élément. Si taille(Liste_1)≤1, on ne fait rien.

Remarques:
* Dans le listing ci-dessus, l'instruction **"j 1 MAX GET"** (plutôt que **"j GET"**) est nécessaire pour que la recherche de la position d'insertion ne nous conduise pas à l'erreur **"0 GET"** dans le cas où l'élément Liste_1(i) doit être inséré en début de liste.
* L'insertion effective est éxécutée de la manière suivante: (il s'agit de placer le i-ème élément après le j-ème).
  On crée la liste des élts d'indice 1 à j (**"1 j SUB"**).
  On y ajoute le i-ème élément (" aide + ").
  On y ajoute la liste des élts d'indice (j+1) à (i-1) (Instruction:**" j 1 + i 1 - SUB +"**).
  On y ajoute la liste des élts d'indice (i+1) à n (où n est la taille de la liste complète): **"i 1 + n SUB +"**

Exercice 20 (****) : Tri d'une liste par la méthode de sélection.



On prend la liste Liste_1, et on la remplace par la liste Liste_2 formée des éléments de Liste_1, triés dans l'ordre croissant, par la méthode de sélection.
(Voir plus loin la comparaison des différentes méthodes de tri).

Si la longueur de Liste_1 est inférieure à 2, on ne fait rien.
Sinon, il faut prévoir une boucle externe (indice i décroissant de la taille de la liste à 2): pour i fixé, on lit la liste des i premiers éléments, en mémorisant la position du maximum rencontré dans cette lecture.
Cette lecture utilise une boucle interne (indice j croissant de 2 à i). On utilise d'autre part une variable pour contenir la position du maximum rencontré, et une autre pour la valeur de ce maximum provisoire). Au départ de la lecture, on pose que le maximum est le premier élément de la liste (c'est pourquoi l'indice j varie de 2 à i).

Texte du programme 'T.SEL'	Commentaires
`« DUP SIZE`	On duplique 'Liste_1'.
`IF DUP 1 > THEN`	Si taille(Liste_1) > 1 alors:
`  2 FOR i`	Boucle externe (on recherche le
`  DUP 1 GET 1`	maximum des i premiers élts).
`  → aide k`	aide=valeur temporaire du max.
	k=position temporaire du max.
`  « 2 i FOR j`	Boucle interne (j=position de
	de l'élément lu).
`  DUP j GET`	
`  IF DUP aide > THEN`	Si Liste_1(j) > aide, alors:
`  'aide' STO`	on met Liste_1(j) dans aide
`  j 'k' STO`	et on met j dans k.
`  ELSE DROP END`	
`  NEXT`	
`  k OVER i GET PUT`	On échange le i-ème élément
`  i aide PUT`	avec le k-ème.
`  »`	
`-1 STEP`	Fin de parcours (le i-ème élt
	est bien placé).
`ELSE DROP END`	Si taille(Liste_1) ≤ 1, on n'a
`»`	rien à faire.

Remarque:
Dans le listing ci-dessus, on constate que l'inversion du i-ème et du k-ème élément est effectuée,même lorsqu'elle est inutile (c'est à dire si le i-ème élément est bien placé au départ, ce qui correspond à i=k). Le test qui permettrait d'éviter cette inversion inutile (mais très rapide) et qui s'écrirait **"IF k i < THEN"** nuirait aux performances de la procédure de tri (Il s'agit en effet d'un cas improbable sur une liste aléatoire).

Comparaison des trois méthodes de tri:

Les trois méthodes de tri ont été testées sur des listes de 10, 20, ou 50 éléments, dans les cas suivants:

* Listes croissantes { 1 2 3 .... }.
* Listes décroissantes.
* Listes aléatoires (obtenues par le programme 'RANDL' de l'exercice 17).

	Longueur	'T.BUL'	'T.INS'	'T.SEL'
Listes croissantes:				
	10	2.3 s	0.5 s	3.1 s
	20	11.1 s	1.5 s	12.3 s
	50	1 mn 34s	5.4 s	1 mn 27s
Listes décroissantes:				
	10	5.1 s	3.6 s	2.5 s
	20	27.3 s	14.1 s	10.6 s
	50	5 mn 30s	1 mn 32s	1 mn 23s
Listes aléatoires:				
	10	4 s	2.1 s	2.5 s
	20	18.4 s	8.5 s	10.3 s
	50	3 mn 34s	1 mn 01s	1 mn 11s

Les résultats ci-dessus appellent plusieurs remarques:

* La méthode de la bulle a des performances médiocres.

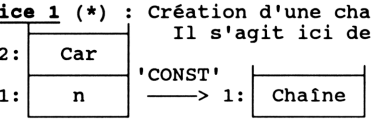
* La méthode de tri par sélection est stable relativement au degré de 'désordre' de la liste à trier.

* La méthode de tri par insertion apparaît comme la plus rapide des trois. C'est particulièrement net dans le cas des suites déjà triées dans l'ordre croissant (ce qui est logique car la position d'insertion de chaque élément est évidente).

On préférera donc (mais il existe d'autres méthodes de tri), le tri par insertion, surtout si l'on doit trier des listes déjà partiellement ordonnées (ce qui est souvent le cas).

CHAINES : Corrigés des Exercices

Exercice 1 (*) : Création d'une chaîne répétitive.

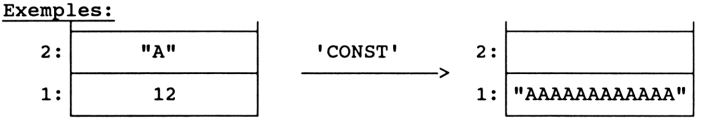


Il s'agit ici de créer une chaîne constituée de la répétition de n exemplaires de 'Car' qui peut être un caractère ou une chaîne de caractères.

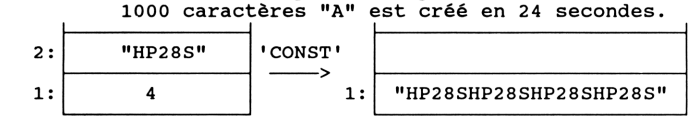
Il suffit de partir d'une chaîne vide "", et d'ajouter n fois 'Car' à cette chaîne.

Texte du programme 'CONST'	Commentaires
« "" 1 ROT START OVER + NEXT SWAP DROP »	"" deviendra la chaîne répétitive De 1 à n : On ajoute 'Car' à la chaîne en construction (sans perdre 'Car') On efface 'Car'. seule la chaîne construite reste sur la pile.

Exemples:



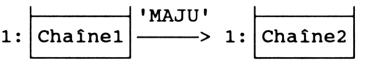
Pour donner un exemple de temps d'éxécution, une chaîne de 1000 caractères "A" est créé en 24 secondes.



Remarque: Si l'entier n placé au niveau 1 est nul ou négatif, le programme **'CONST'** donne une chaîne constituée d'un seul exemplaire de 'Chaîne1'.Si on veut que le résultat soit, dans une telle situation, la chaîne vide, il suffit de modifier le programme **'CONST'** de la manière suivante:

```
« " "   IF   OVER  1  ≥   THEN
       1   ROT   START   OVER   +   NEXT
   ELSE   SWAP   DROP   END
   SWAP   DROP
»
```

Exercice 2 (**) : Passage en mode 'majuscules'.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par passage des caractères en majuscules.

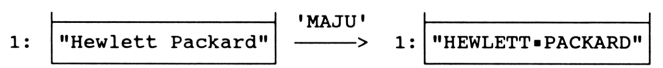
Il suffit de mettre à 0 le bit ci-contre (et qui correspond à 2^5=32 dans l'écriture binaire représentant le code d'un caractère).

*	*	0	*	*	*	*	*

Pour celà il faut faire un **AND** de chaque caractère de 'Chaîne1' avec le caractère "ß", de code 223 (11011111 en binaire).
On crée pour celà une chaîne constante avec ce caractère, et on fait un **AND** entre cette chaîne constante et 'Chaîne1'.
Le programme **'MAJU'** appelle donc le programme **'CONST'**.
Remarque: ß s'obtient par la séquence de touche α ╔═> B.

Texte du programme 'MAJU'	Commentaires
« DUP SIZE	Calcul de la longueur de Chaîne1.
"ß"	Caractère ß, de code 223.
SWAP CONST	On crée une chaîne constante , de même longueur que Chaîne1 , et formée de caractères "ß".
AND	On effectue un AND entre cette chaîne constante et Chaîne1.
»	

Exemple:



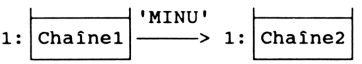
Dans cet exemple, l'espace (code 32 = 00100000b) a été transformé en le caractère de code 0 (non affichable, ni même éditable).

Si on veut que seuls les caractères minuscules de "a" à "z" soient affectés par le programme **'MAJU'**, il faut trouver une autre solution: il suffit de faire un **AND** entre le caractère de ß (de code 223) et les seuls caractères compris entre "a" et "z".

Le programme suivant répond à la question (il a néanmoins l'inconvénient d'être un peu plus lent que la première version):

```
« DUP SIZE "ß" → ch n c
  « "" 1 n FOR k
       ch k k SUB
       IF DUP "a" ≥ OVER "z" ≤ AND
          THEN c AND END
       +
    NEXT
  »
»
```

Exercice 3 (**) : Passage en mode 'minuscules'.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par passage des caractères en minuscules.

Il suffit de mettre à 1 le bit ci-contre (et qui correspond à 2^5=32 dans l'écriture binaire représentant le code d'un caractère).

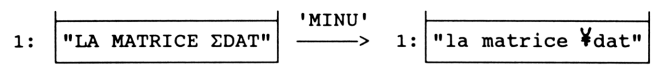
*	*	1	*	*	*	*	*

Pour celà il faut faire un **OR** de chaque caractère de 'Chaîne1' avec le caractère 'espace' dont le code est 32 (00100000 en binaire). On crée pour celà une chaîne constante avec ce caractère, et on fait un **OR** entre cette chaîne constante et 'Chaîne1'.

Le programme **'MINU'** appelle donc le programme **'CONST'**.

Texte du programme 'MINU'	Commentaires
« DUP SIZE " " SWAP CONST OR »	Longueur n de 'Chaîne1'. Caractère 'Espace' Chaîne de n espaces. On fait un OR entre cette chaîne et 'Chaîne1'.

Exemple:



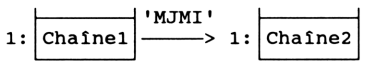
Dans cet exemple, le caractère Σ, de code 133 = 10000101b a été transformé en le caractère ¥,de code 165 = 10100101b.

Si on veut que seuls les caractères majuscules de "A" à "Z" soient affectés par le programme **'MINU'**, il faut trouver une autre solution: il suffit de faire un **OR** entre le caractère 'espace' et les seuls caractères compris entre "A" et "Z".

Le programme suivant répond à la question (il a néanmoins l'inconvénient d'être un peu plus lent que la première version):

```
« DUP SIZE → ch n
   « "" 1 n FOR k
        ch k k SUB
        IF DUP "A" ≥ OVER "Z" ≤ AND
           THEN " " OR END
        +
     NEXT
   »
»
```

Exercice 4 (**) : bascule majuscules/minuscules.



On prend la chaîne Chaîne1, et on la remplace par la chaîne Chaîne2 obtenue par transformation des majuscules en minuscules et des minuscules en majuscules.

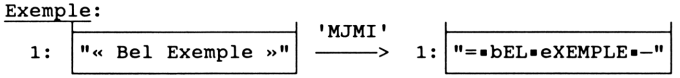
Il suffit d'inverser le bit ? ci-contre (et qui correspond à 2^5=32 dans l'écriture binaire représentant le code d'un caractère).

*	*	?	*	*	*	*	*

Pour celà il faut faire un **XOR** de chaque caractère de 'Chaîne1' avec le caractère 'espace' dont le code est 32 (00100000 en binaire). On crée pour celà une chaîne constante avec ce caractère, et on fait un **XOR** entre cette chaîne constante et 'Chaîne1'.
Le programme **'MJMI'** appelle donc le programme **'CONST'**.

Texte du programme 'MJMI'	Commentaires
« DUP SIZE	Longueur n de 'Chaîne1'.
" "	Caractère 'Espace'
SWAP CONST	Chaîne de n espaces.
XOR	On fait un XOR entre cette chaîne et 'Chaîne1'.
»	

Exemple:

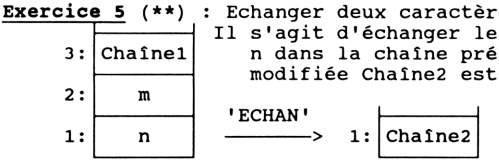


Dans cet exemple:
Les espaces (code 32 = 00100000b) sont transformés en des caractères de code 0.
Le caractère « (code 146 = 10010010b) est transformé en le caractère =, de code 178 = 10110010b.
Le caractère » (code 147 = 10010011b) est transformé en le caractère —, de code 179 = 10110011b.

Si on veut que seuls les caractères alphabétiques soient affectés par le programme **'MJMI'**, il faut trouver une autre solution:il suffit de faire **XOR** entre le caractère 'espace' et les seuls caractères alphabétiques.
(pour tester si un caractère est alphabétique on lui applique l'instruction **" " OR**, et on voit si le résultat est compris entre "a" et "z").
Le programme suivant répond à la question (il a néanmoins l'inconvénient d'être un peu plus lent que la première version):

```
« DUP SIZE → ch n
   « "" 1 n FOR k
        ch k k SUB DUP " " OR
        IF DUP "a" ≥ SWAP "z" ≤ AND
           THEN " " XOR END +
        NEXT
   »
»
```

Exercice 5 (**) : Echanger deux caractères dans une chaîne.

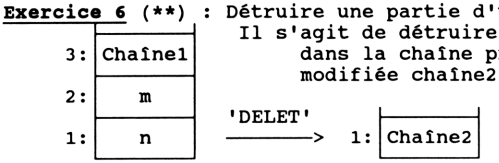


Il s'agit d'échanger les caractères de position m et n dans la chaîne présente au niveau 3. La chaîne modifiée Chaîne2 est obtenue au niveau 1.

Si l'une au moins des deux positions m et n est incorrecte (c'est à dire n'est pas comprise entre 1 et la taille de la chaîne), Chaîne1 restera inchangée. Il doit en être de même si m=n.

Texte du programme 'ECHAN'	Commentaires
« DUP2 MIN 3 ROLLD MAX IF DUP2 ≠ OVER 5 PICK SIZE ≤ AND 3 PICK 1 ≥ AND THEN → c m n « c 1 m 1 - SUB c n n SUB + c m 1 + n 1 - SUB + c m m SUB + c n 1 + c SIZE SUB + » ELSE DROP2 END »	Position minimum au niveau 2, maximum au niveau 1. Si m est différent de n, et si max(m,n)≤taille(chaîne), et si min(m,n)≥1, Alors: c=chaîne, m=position mini, n=position maxi. m-1 premiers caractères de c. on ajoute le n-ème caractère. on ajoute les caractères du (m+1)-ème au (n-1)-ème. on ajoute le m-ème caractère. on ajoute les caractères à partir du (n+1)-ème. Si données incorrectes on efface les deux indices.

Exercice 6 (**) : Détruire une partie d'une chaîne de caractères.



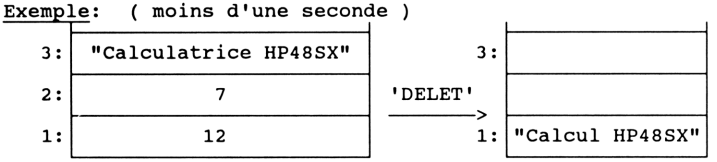
Il s'agit de détruire les caractères d'indice m à n dans la chaîne présente au niveau 3. La chaîne modifiée chaîne2 est obtenue au niveau 1.

Texte du Programme 'DELET'	Commentaires
« IF DUP2 ≤ THEN → C m n « C 1 m 1 - SUB C n 1 + C SIZE SUB + » ELSE DROP2 END »	On teste si m ≤ n. Si m ≤ n, on crée les variables locales C (chaîne), m et n, on crée la sous-chaîne des m-1 premiers caractères, on ajoute la sous-chaîne des caractères suivant le n-ième Si m < n,on conserve la chaîne initiale.

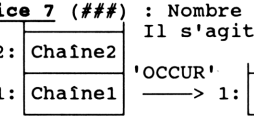
CHAINES: Corrigés des Exercices.

Remarques:
Le listing ci-dessus est essentiellement le même que celui de **'DELET'** (Exercice 2 sur les listes).
Si m < n, le programme **'DELET'** ne modifie pas la chaîne.
Si m = n, seul le n-ième élément de la chaîne est supprimé.
Si m < 1, l'effet est le même que si m=1.
Si n>taille(Chaîne),l'effet est le même que si n=taille(Chaîne).

Exemple: (moins d'une seconde)



Exercice 7 (***###***) : Nombre d'occurrences d'une chaîne dans une autre.



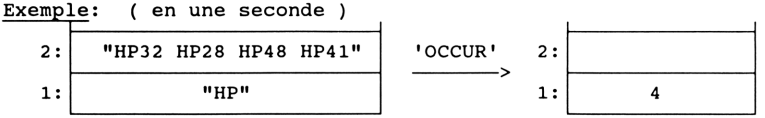
Il s'agit de calculer le nombre de fois que 'Chaîne1' apparaît dans 'Chaîne2'. Le résultat, ici appelé 'Nombre', vaut 0 si 'Chaîne1' n'apparait pas dans 'Chaîne2'.

L'idée du programme **'OCCUR'** est de créer un compteur (placé au niveau 3 et initialisé à 0), auquel on ajoute un à chaque fois qu'une occurrence de 'Chaîne1' est rencontrée dans 'Chaîne2'.
Pour éviter le problème lié au fait que l'instruction **POS**, qui détecte la présence d'une chaîne dans une autre,n'en détecte en fait que la première occurrence, on détruit un caractère de chaque occurrence détectée de 'Chaîne1' dans 'Chaîne2' au moyen du programme **'DELET'**.

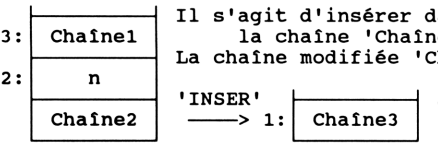
Texte du programme 'OCCUR'	Commentaires
« 0 3 ROLLD	Compteur initialisé à 0, placé au niveau 3.
WHILE DUP2 POS DUP	Tant que 'Chaîne1' est comprise
REPEAT	dans 'Chaîne2':
ROT SWAP DUP	* détruire le premier caractère
DELET	de l'occurrence détectée.
ROT 1 + 3 ROLLD	* incrémenter le compteur.
SWAP	
END	Quand Chaîne1 n'est plus détectée
3 DROPN	dans Chaîne2, on efface la pile
»	à l'exception du compteur.

Remarque:
Le listing ci-dessus est essentiellement le même que celui de **'OCCUR'** (Exercice 3 sur les listes).

Exemple: (en une seconde)



Exercice 8 (**) : Insérer une chaîne dans une autre.



Il s'agit d'insérer dans Chaîne1, à la position n, la chaîne 'Chaîne2'.
La chaîne modifiée 'Chaîne_3' est obtenue au niveau 1 de la pile.
Si n est supérieur à la taille de 'Chaîne1', l'insertion est en fait une concaténation de 'Chaîne2' à 'Chaîne1'.

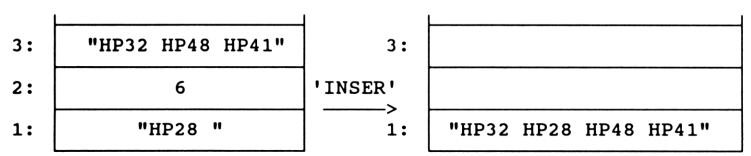
Si n est négatif ou nul, l'effet doit être le même que si n=1.

Le principe de **INSER** est simple: On coupe la chaîne Chaîne1 en 2 parties, au niveau de la position d'insertion souhaitée, on ajoute 'Chaîne2' à la fin de la 1ère partie, avant de compléter par la deuxième partie.

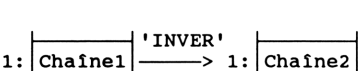
Texte du programme 'INSER'	Commentaires
« → C m a	C:Chaîne1, m:position, a:Chaîne2
« C 1 m 1 - SUB	On crée la sous-chaîne formée des m-1 premiers caractères.
a +	On lui ajoute 'Chaîne2'..
C m C SIZE SUB	On crée la sous-chaîne formée caractères à partir du m-ième.
+	On l'ajoute à la chaîne formée précédemment.
»	
»	

Remarque:
Le listing ci-dessus est essentiellement le même que celui de **'INSER'** (Exercice 4 sur les listes).

Exemple: (En moins d'une seconde)



Exercice 9 (*) : Inverser l'ordre des caractères d'une chaîne.



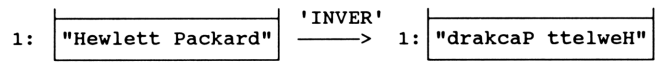
On prend la chaîne 'Chaîne1', et on la remplace par la chaîne 'Chaîne2' formée des mêmes caractères, mais dans l'ordre inverse.

La solution est très simple: on crée une chaîne vide, à laquelle on ajoute successivement tous les caractères de Chaîne1, en commençant par le dernier, et en terminant par le premier.

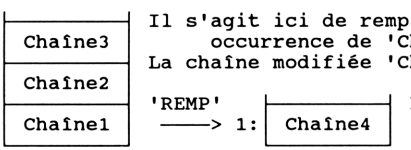
CHAINES: Corrigés des Exercices.

Texte du programme 'INVER'	Commentaires
« DUP SIZE → c n « "" n 1 FOR i c i i SUB + -1 STEP » »	c = Chaîne1, n = sa longueur. "" deviendra Chaîne2. On ajoute chaque caractère de Chaîne1, en partant de la fin.

Exemple: (en moins d'une seconde)



Exercice 10 (***) : Recherche et remplacement dans une chaîne.



Il s'agit ici de remplacer, dans 'Chaîne3', chaque occurrence de 'Chaîne2' par 'Chaîne1'.
La chaîne modifiée 'Chaîne4' est obtenue au niveau 1 de la pile.
Evidemment, si 'Chaîne2' n'est pas détectée dans 'Chaîne3' alors 'Chaîne4'='Chaîne3'.

Dans la solution ci-dessous, Chaîne2 et Chaîne1 sont placées dans deux variables. Seule reste sur la pile la chaîne Chaîne3, au dessus de laquelle on place une chaîne vide, qui petit à petit deviendra Chaîne4.

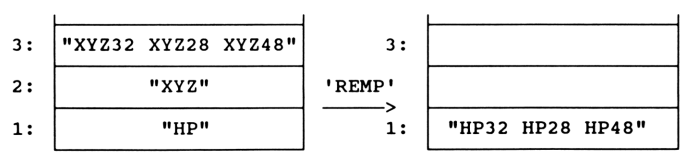
a) Si Chaîne2 est détectée dans Chaîne3, en position n, alors:
On ajoute à Chaîne4 les n-1 premiers caractères de Chaîne3,
On ajoute à Chaîne4 la chaîne de remplacement Chaîne1.
On ne conserve dans Chaîne3 que les caractères qui suivent l'occurrence détectée de Chaîne2.

b) On continue ainsi tant que Chaîne2 est détectée dans ce qu'il reste de Chaîne3.

c) Il suffit alors d'ajouter à Chaîne4 le reste de Chaîne3.

Bien sûr, si Chaîne2 n'a jamais été détectée dans Chaîne3, alors les instructions a) et b) sont ignorées et la chaîne Chaîne4 obtenue est égale à Chaîne3.

Exemple: (En moins d'une seconde)



Texte du programme 'REMP'	Commentaires
« OVER SIZE → ch2 ch1 t	ch1, ch2 = Chaîne1 et Chaîne2, t = taille de Chaîne2.
« "" SWAP	"" va devenir Chaîne4, on la met au dessus de Chaîne3.
WHILE DUP ch2 POS DUP	Tant que Chaîne2 est dans Chaîne3
REPEAT	
DUP2 1 - 1 SWAP	On ajoute à Chaîne4 la partie
SUB 4 ROLL SWAP	de Chaîne3 qui précède cette
+ ch1 + 3 ROLLD	occurrence, puis Chaîne1, et
t + OVER SIZE SUB	on ne conserve dans Chaîne3
END	que ce qui vient après cette occurrence.
DROP +	Maintenant que Chaîne2 n'est plus détectée dans ce qui reste de Chaîne3, on ajoute Chaîne3 à Chaîne4.
»	
»	

Exercice 11 (##) : Cryptage d'une chaîne de caractères.

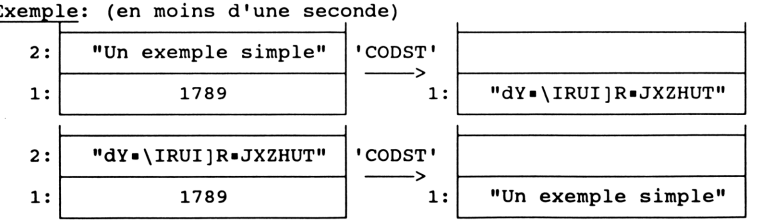
Il s'agit de crypter (coder) une chaîne de caractères 'Chaîne1' en utilisant un code qui peut être un objet quelconque. La chaîne ainsi cryptée est obtenue au niveau 1 de la pile.

Ce codage doit être réversible:si on applique la même opération à 'Chaîne2', et avec le même code,on doit obtenir de nouveau la chaîne initiale 'Chaîne1'.

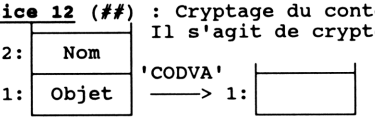
Pour ce faire, il suffit de transformer 'Objet' en chaîne de caractères, de créer une chaîne constante composée de répétitions d' "Objet", et de faire un **XOR** entre cette chaîne constante et la chaîne de caractères 'Chaîne1'.

Texte du programme 'CODST'	Commentaires
« →STR	On transforme 'Objet' en chaîne.
DUP SIZE → c tc	c = cette chaîne, tc = sa taille.
« DUP SIZE tc / CEIL	Création d'une chaîne répétant c,
c SWAP CONST	de longueur ≥ taille(Chaîne1).
1 3 PICK SUB	La chaîne répétitive est tronquée à la longueur de Chaîne1.
XOR	Ou exclusif entre 'Chaîne1' et la chaîne répétitive.
»	
»	

Exemple: (en moins d'une seconde)



Exercice 12 (**##**) : Cryptage du contenu d'une variable.



Il s'agit de crypter le contenu d'une variable dont le nom apparaît au niveau 2 , en utilisant un objet quelconque comme code.

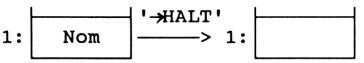
Ce codage doit être réversible:si on applique la même opération à la même variable, et avec le même code, on doit retrouver le contenu initial de cette variable.

Il suffit ici de transformer le contenu du programme en chaîne de caractères, de coder cette chaîne de caractères avec **'CODST'**, et de placer cette chaîne codée dans la variable.

Si la variable dont le nom figure au niveau 2 contient en fait une chaîne de caractères, on considère que cette chaîne résulte d'un codage antérieur d'un programme: on code à nouveau cette chaîne, le résultat est transformé en programme et placé dans la variable, qui retrouve ainsi son contenu initial (à condition toutefois qu'on ait bien utilisé le même code!).

Texte du programme 'CODVA'	Commentaires
« OVER RCL	Rappeler contenu de la variable.
DUP TYPE SWAP	On prend le type de ce contenu.
→STR	Transformer ce contenu en chaîne.
ROT CODST	On code cette chaîne.
IF SWAP 2 == THEN	Si le contenu était une chaîne,
OBJ→	alors on le transforme en la
END	forme initiale.
SWAP STO	Le résultat va dans la variable.
»	

Exercice 13 (**##**) : Placer des **HALT** dans un programme.

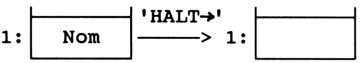


On considère un programme stocké dans une variable dont le nom est placé au niveau 1.
Il s'agit de modifier ce programme en plaçant un **HALT** après chaque caractère « (donc au minimum au début du programme, et plus généralement au début de chaque structure locale).
La solution est très simple en utilisant **'REMP'**: il suffit, dans le corps du programme préalablement transformé en chaîne, de remplacer les occurrences de **"« "** par autant de **"« HALT "**.

Texte du programme '→HALT'	Commentaires
« DUP	On duplique le nom du programme.
RCL	On rappelle le contenu du progr.
→STR	On le transforme en chaîne.
"« " "« HALT "	On remplace tous les « par des
REMP	« HALT .
OBJ→	On revient à la forme 'programme'
SWAP STO	Le résultat va dans la variable.
»	

Remarque: pour que '→**HALT**' fonctionne normalement,il faut que le programme à modifier ne contienne pas le symbole "« " dans une chaîne de caractères (Si on applique '→**HALT**' à lui-même, par exemple, le résultat n'est pas celui souhaité...).

Exercice 14 (**##**) : Enlever les **HALT** d'un programme.



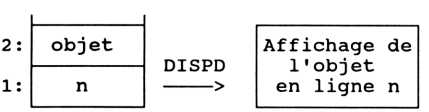
On considère un programme stocké dans une variable dont le nom est placé au niveau 1.
Il s'agit de modifier ce programme en retirant les instructions HALT qui pourraient s'y trouver (le programme **HALT→** défait donc ce qu'a fait le programme →**HALT**).

Là encore, il suffit de remplacer, dans la chaîne de caractères représentant le programme à modifier, toutes les occurrences de **HALT** par des chaînes vides.

Texte du programme 'HALT→'	Commentaires
«	
DUP	On duplique le nom du programme.
RCL	On rappelle le contenu du progr.
→STR	On le transforme en chaîne.
"HALT" "" REMP	On efface les "HALT".
OBJ→	On revient à la forme 'programme'
SWAP STO	Le résultat va dans la variable.
»	

Remarque:
Pour un fonctionnement correct, il est indispensable que le programme à modifier ne comporte pas d'occurrences de HALT autres que celles qui suspendent son éxécution. Il ne faut pas, par exemple, que le nom d'une variable contienne le mot **HALT** , ou que celui-ci apparaisse dans une chaîne de caractères.

Exercice 15 (********) : Affichage cadré à droite d'un objet quelconque.



Il s'agit ici d'afficher un objet quelconque à partir de la ligne n, avec le schéma fonctionnel de l'instruction **DISP**, à ceci près que sur chaque ligne concernée, l'affichage doit être cadré à droite (alors que **DISP** cadre à gauche).

Ici, '**DISPD**' doit se comporter de manière analogue à **DISP**:

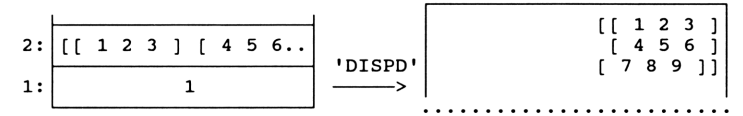
* Si n < 1, on pose n=1.
* Si n > 7, on pose n=7.
* L'objet est affiché en mode multilignes.
* Si l'objet est trop volumineux pour être totalement affiché, seules les premières lignes apparaissent à l'écran.

Texte du programme 'DISPD'	Commentaires

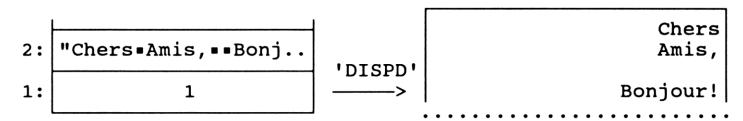
`«  " "  22  CONST`
`  SWAP  1  MAX  7  MIN`
`  →  esp  n`
`  «  →STR`
`  10  CHR  +`
`  WHILE  DUP  10  CHR  POS`
`  DUP  n  7  ≤  AND`
 	
`  REPEAT`
`  → k`
`  « esp  1  23  k  -  SUB`
`  OVER  1  k  1  -  SUB`
`  +`
`  n  DISP`
`  'n'  1  STO+`
`  k  1  +  OVER  SIZE`
`  SUB`
`  »`
`  END  DROP2`
`  3  FREEZE`
`  »`
`»`
	Chaîne de 22 espaces.
	Si n n'est pas compris entre 1 et 7, on le remplace par 1 ou 7.
	Transformer l'objet en chaîne et lui ajouter un 'Retour-chariot'
	Tant que la chaîne contient un 'Retour-chariot' et tant que le numéro de ligne est ≤ 7 :
	
	k = position du retour-chariot.
	Chaîne de (23-k) espaces.
	On y ajoute les caractères qui précèdent ce retour-chariot.
	On affiche en ligne n.
	On incrémente le N° de ligne.
	On détruit ce qui vient d'être affiché.
	
	Terminé: on nettoie la pile.
	On gèle l'affichage.

Exemples:

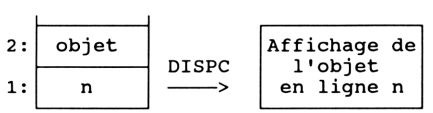
On veut afficher la matrice [[1 2 3 [4 5 6 [7 8 9]] cadrée à droite, en commençant à la ligne 1.



On veut afficher la chaîne "Chers▪Amis,▪▪Bonjour!", où les caractères ▪ désignent des 'retours-chariots'.



Exercice 16 (****) : Affichage centré d'un objet quelconque.



Il s'agit ici d'afficher un objet quelconque à partir de la ligne n, avec le schéma fonctionnel de l'instruction **DISP**, à ceci près que sur chaque ligne concernée, l'affichage doit être centré.

Ici, **'DISPC'** doit se comporter de manière analogue à **DISP**:

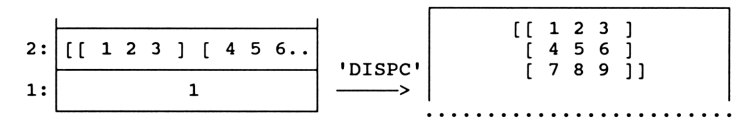
* Si n < 1, on pose n=1, et si n > 7, on pose n=7.
* L'objet est affiché en mode multilignes.
* Si l'objet est trop volumineux pour être totalement affiché, seules les premières lignes apparaissent à l'écran.

Texte du programme 'DISPC'	Commentaires
« " " 22 CONST	Chaîne de 22 espaces.
SWAP 1 MAX 7 MIN	Si n n'est pas compris entre 1 et
→ esp n	7, on le remplace par 1 ou 7.
« →STR	Transformer l'objet en chaîne et
10 CHR +	lui ajouter un 'Retour-chariot'
WHILE DUP 10 CHR POS	Tant que la chaîne contient un
DUP n 7 ≤ AND	'Retour-chariot' et tant que le
	numéro de ligne est ≤ 7 :
REPEAT	
→ k	k = position du retour-chariot.
« esp 1 23 k - SUB	Chaîne de (23-k) espaces.
2 / FLOOR	On en conserve la moitié.
OVER 1 k 1 - SUB	On y ajoute les caractères qui
+	précèdent ce retour-chariot.
n DISP	On affiche en ligne n.
'n' 1 STO+	On incrémente le N° de ligne.
k 1 + OVER SIZE	On détruit ce qui vient d'être
SUB	affiché.
»	
END DROP2	Terminé: on nettoie la pile.
3 FREEZE	On gèle l'affichage.
» »	

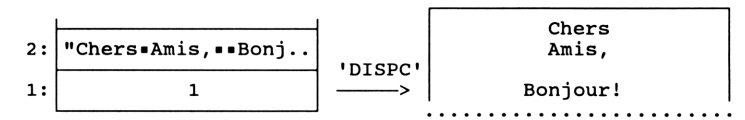
Remarque: les listings de **'DISPC'** et **'DISPD'** sont pratiquement identiques. la seule différence est la présence de l'instruction **2 / FLOOR** dans **'DISPC'**.

Exemples:

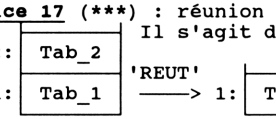
On veut afficher la matrice [[1 2 3 [4 5 6 [7 8 9]] centrée, en commençant à la ligne 1.



On veut afficher la chaîne "Chers▪Amis,▪▪Bonjour!", où les caractères ▪ désignent des 'retours-chariots'.



Exercice 17 (***) : réunion de deux tableaux.



Il s'agit de réunir deux tableaux 'Tab_1' et 'Tab_2', de manière à n'en plus faire qu'un, nommé ici 'Tab_3' et obtenu au niveau 1 de la pile.

Le tableau 'Tab_3' doit être la superposition de 'Tab_2' et de 'Tab_1', comme si on avait posé 'Tab_2' sur 'Tab_1'.
'Tab_1' et 'Tab_2' peuvent être des vecteurs ou des matrices, de nombres réels ou complexes. Il est indispensable que ces deux tableaux aient le même nombre de colonnes.

CHAINES: Corrigés des Exercices.

La solution consiste à transformer Tab_1 et Tab_2 en deux chaînes de caractères, à ajouter ces deux chaînes (après quelques précautions) et à redonner la forme 'tableau' à la chaîne ainsi obtenue.

Les précautions à prendre sont les suivantes:

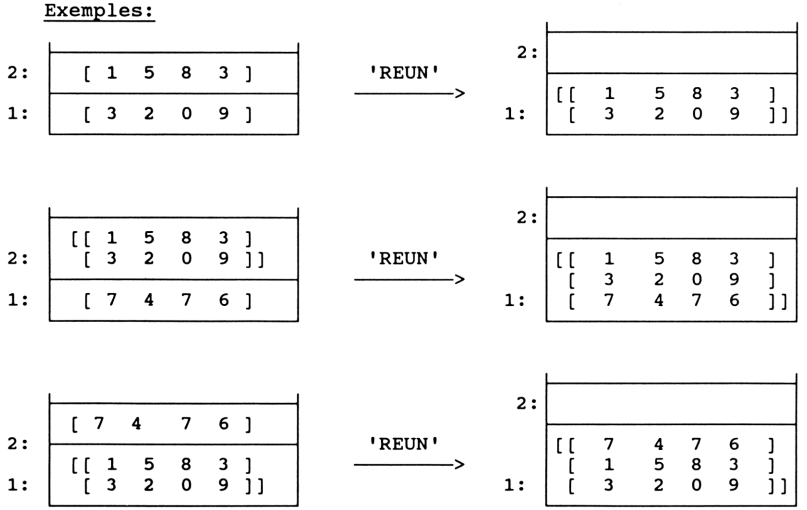
* Si Tab_2 est un vecteur, et non une matrice, ajouter un [au début de la chaîne représentant Tab_2. Ainsi le résultat final sera bien une matrice.

* Eliminer le] qui termine Tab_2 (si Tab_2 est un vecteur, ça n'a pas d'importance, mais c'est utile si Tab_2 est une matrice, pour qu'on puisse lui 'concaténer' Tab_1.

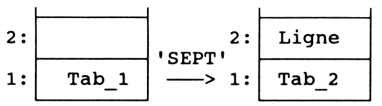
* Si Tab_1 est une matrice, lui retirer le [qui le débute, là encore pour pouvoir juxtaposer Tab_2 et Tab_1.

Texte du programme 'REUT'	Commentaires
`«  →STR  SWAP  →STR`	
`  1  OVER  SIZE  1  -  SUB`	On transforme les tableaux en 2 chaînes de caractères.
`  IF  DUP  1  2  SUB  "[["  ≠`	On supprime le dernier] de Tab_2
`  THEN  "["  SWAP  +  END`	Si les deux premiers caractères de tab_2 ne sont pas [[,alors ajouter "[" au début de Tab_2.
`  SWAP`	
`  IF  DUP  1  2  SUB  "[["  ==`	Si les deux premiers caractères
`  THEN  2  OVER  SIZE  SUB`	de t1 sont "[[", alors on sup-
`  END`	-prime le premier [.
`  +  OBJ→`	Ajouter les deux chaînes obtenues
`»`	et transformer en tableau.

Exemples:



Exercice 18 (***) : séparation d'un tableau en sa ligne supérieure et le reste du tableau.



On part d'une matrice d'au moins 2 lignes, placée au niveau 1 . On en extrait la ligne supérieure, que l'on place au niveau 2. Le reste du tableau est obtenu au niveau 1.

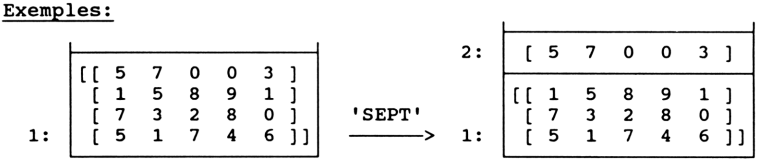
Il suffit ici de transformer la matrice en chaîne de caractères, d'y repérer la première occurrence de] (ce qui désigne la fin de la première ligne).

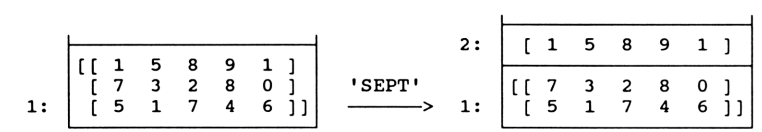
On extrait de cette chaîne la partie correspondant à cette ligne et on lui redonne la forme vecteur.

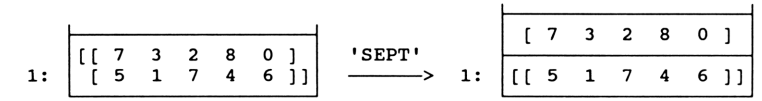
On ajoute un [au début de la chaîne restante (celle qui suivait le premier]) pour en faire une chaîne représentant une matrice, et on lui redonne la forme tableau.

Texte du programme 'SEPT'	Commentaires
« →STR DUP SIZE OVER "]" POS → t n	Transformer la matrice en chaîne, et n est la position du premier caractère] dans cette chaîne.
« DUP 2 n SUB OBJ→	On extrait la première ligne et on lui redonne la forme vecteur
"[" ROT n 1 + t SUB + OBJ→	On ajoute un [au début de la chaîne restante et on lui donne la forme tableau.
»	

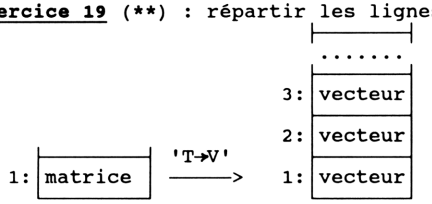
Exemples:







Exercice 19 (**) : répartir les lignes d'une matrice sur la pile.

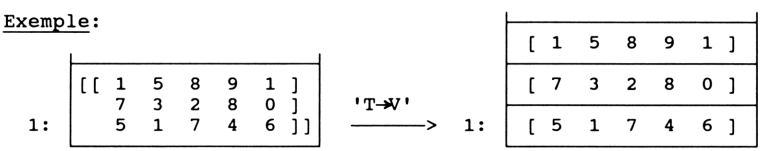


Il s'agit ici de séparer la matrice située au niveau 1 en ses différentes lignes.
Les vecteurs obtenus sur la pile sont placés en conformité avec la position qui était la leur dans la matrice.
Au niveau 1, par exemple, on trouve la ligne inférieure.

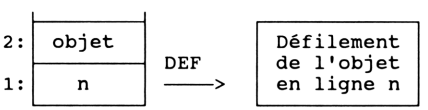
La solution est ici très simple: on transforme la matrice en chaîne de caractères. On élimine le [qui débute cette chaîne, et le] qui la termine. On éxécute ensuite l'instruction **OBJ→** qui répartit les différentes lignes sur la pile.

Texte du programme 'T→V'	Commentaires
« →STR 2 OVER SIZE 1 - SUB OBJ→ »	Transformer la matrice en chaîne, Enlever le premier et le dernier caractère dans cette chaîne. Cette instruction répartit les lignes sur la pile.

Exemple:



Exercice 20 (**) : défilement d'un message à l'écran.



Il s'agit ici de faire défiler un objet quelconque sur la ligne n.
Le défilement est périodique, d'un caractère à la fois de la droite vers la gauche.

L'idée est de modifier l'objet à afficher après chaque mouvement, en déplaçant en dernière position le premier de ses caractères.

On donne ainsi l'illusion d'un défilement horizontal,qui est bien entendu périodique.

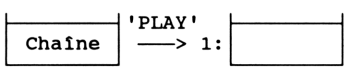
Il faut préalablement transformer l'objet en chaîne, de manière à le compléter par des espaces, et ce pour deux raisons:

* A une étape quelconque de l'affichage, la fin de l'objet doit être clairement séparée de son début.

* L'objet affiché doit être au moins long de 22 caractères, pour que le défilement s'effectue sur toute la largeur de l'écran.

Texte du programme 'DEF'	Commentaires
« → n « →STR 1 OVER SIZE 22 - NEG 4 MAX START " " + NEXT DO DUP n DISP DUP 2 OVER SIZE SUB SWAP 1 1 SUB + .2 WAIT UNTIL KEY END DROP2 » »	n = numéro de ligne d'affichage. On transforme l'objet en chaîne. On complète l'objet par au moins 4 espaces, la longueur totale étant au moins égale à 22. Répéter: On duplique l'objet avant de l'envoyer en ligne n. On déplace le premier caractère de l'objet en fin d'objet. Temporisation de 0.2 secondes. Jusqu'à ce qu'une touche soit appuyée. On nettoie la pile (l'objet et la touche appuyée).

Exercice 21 (****): Jouer un air musical.



Le programme **'PLAY'** joue l'air représenté par la chaîne placée au niveau 1.
Cette chaîne est constituée de "maillons" du type:

* Une lettre de A à L désignant une note, suivie d'un chiffre de 1 à 9 pour indiquer le durée.
* La lettre P, suivie d'un chiffre de 1 à 9 pour indiquer une pause.
* Le signe + pour indiquer qu'on monte d'un octave.
* le signe - pour indiquer qu'on descend d'un octave.

La correspondance entre les lettres et les notes est la suivante:

Lettre	A	B	C	D	E	F	G	H	I	J	K	L
Note	DO	DO#	RE	RE#	MI	FA	FA#	SOL	SOL#	LA	LA#	SI

Les durées des notes sont proportionnelles aux chiffres indiqués après ces notes. L'unité de durée (qui correspond donc à la note la plus rapide) doit être placée dans une variable globale nommée **'TEMPO'** (y mettre par exemple 0.05).
On peut utiliser le contenu de **'TEMPO'** pour jouer l'air plus ou moins rapidement.

Une variable globale nommée **'UT'** doit contenir la fréquence de la note DO. La fréquence du DO 'normal' est de 523.25113 Hz.
Pour jouer une octave plus haut (resp. plus bas), il suffit de multiplier (resp. diviser) cette valeur par 2.

Remarques:
On notera, dans le listing qui suit, comment une liste constituée des différents sous-programmes permet d'éviter le recours à de nombreuses instructions **"IF THEN"** imbriquées.
Avant d'utiliser ce programme, assurez vous que le buzzer de votre calculatrice n'est pas désactivé (activez l'entrée **BEEP** dans le menu **MODES**) sinon aucun son ne sortira.

CHAINES: Corrigés des Exercices.

Texte du programme 'PLAY'	Commentaires

« "ABCDEFGHIJKLP+-"
{
« 1 - 12 / 2 SWAP ^
UT * OVER NUM 48 -
TEMPO * BEEP
2 OVER SIZE SUB »
« DROP DUP NUM 48 -
TEMPO * WAIT
2 OVER SIZE SUB »
« DROP 'UT' 2 STO* »
« DROP 'UT' 2 STO/ »
}
→ ch sp
« DO
DUP 2 OVER SIZE SUB
ch
ROT 1 1 SUB
POS
sp
OVER 12 MAX 11 -
GET EVAL
UNTIL DUP SIZE NOT END
DROP
»
»
	Chaîne des caractères permis.
	Listes des sous-programmes.
	Sous-programme appelé pour jouer une note, avec une certaine durée.
	Sous-programme appelé pour marquer une pose.
	Montée d'un octave.
	Descente d'un octave.
	ch: chaîne, sp: liste.
	Répéter:
	Chaîne privée de son 1er car.
	Caractères autorisés.
	Premier caractère de la chaîne
	Position dans ch.
	Liste des sous-programmes
	Position convertie par rapport à la liste sp.
	On extrait le sous-programme et on l'évalue.
	Jusqu'à ce que la chaîne soit vide.

Un exemple d'utilisation de 'PLAY':

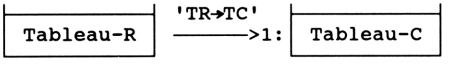
L'exemple suivant permet de jouer un air connu. la chaîne qui contient l'air musical reprend souvent plusieurs fois une même succession de notes. L'idée est donc d'écrire un programme contenant les successions de notes utiles,puis les combinant de manière à former la chaîne complète.

```
« "I4I4G1F1F4E1F1F8E1F1F4E1F1"
  "I4F1I1G8D8D4D4C1D1D4C1D1G8G1F1E1F4I4I4K4D8D4"
  "G1F1D1A1D8B8B4B4A1B1E4D1B1+B8-"
  "B1D1F1I1"
  → a b c d e
  « a b + a + c + d + d + e +
    PLAY
  »
»
```

TABLEAUX : Corrigés des Exercices

Exercice 1 (*) : Transformer un tableau réel en tableau complexe.

Le programme **'TR→TC'** transforme le tableau réel au niveau 1 en un tableau complexe de partie imaginaire nulle. Si le tableau initial est complexe, il est inchangé.



La solution est triviale: multiplier par 1..., mais 1 écrit au format des nombres complexe, c'est-à-dire (1,0)! De cette manière, le tableau reste inchangé s'il était complexe au départ.

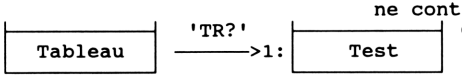
Une autre solution serait de faire: **DUP 0 CON R→C.**
(On crée un tableau constant B de zéros réels, de même format que le tableau de départ A, et on forme un tableau complexe C=A+iB, de partie imaginaire nulle, avec ces deux tableaux réels).

Le défaut de cette deuxième méthode est qu'elle ne fonctionne pas si le tableau initial est complexe.
Elle est cependant un peu plus rapide...

Texte du programme 'TR→TC'	Commentaires
« (1,0) * »	On multiplie par le réel 1 écrit avec le format complexe...

Exercice 2 (*) : Tester si un tableau est formé de nombres réels.

Le programme **'TR?'** teste si le tableau présent au niveau 1 ne contient que des nombres réels. On obtient 1 au niveau 1 si la réponse est oui, 0 sinon.



Là encore la solution est très, très simple: On duplique le tableau, on en prend le conjugué, et on teste si les deux tableaux sont égaux.

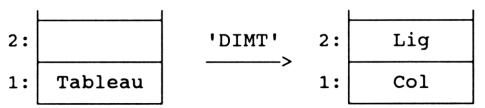
Si le tableau initial C est au format réel l'instruction **CONJ** le laisse inchangé, et == donne le résultat 1.

Autre méthode: **IM** **ABS** **NOT** .On prend la partie imaginaire B du tableau C.On calcule sa norme euclidienne par **ABS** (mais on pourrait tout aussi bien utiliser la norme de ligne **RNRM** ou la norme de colonne **CNRM**) : Si le tableau C est de partie imaginaire nulle, on doit trouver 0. Il suffit alors de faire **NOT** pour obtenir le résultat du test.

Cette méthode est cependant environ deux fois plus lente que la méthode choisie ici.

Texte du programme 'TR?'	Commentaires
« DUP CONJ == »	On duplique le tableau. On en calcule le conjugué. On teste si les deux tableaux sont égaux.

Exercice 3 (*) : Dimensions d'un tableau.



'DIMT' place au niveau 1 le nombre de colonnes et au niveau 2 le nombre de lignes du tableau placé au niveau 1.

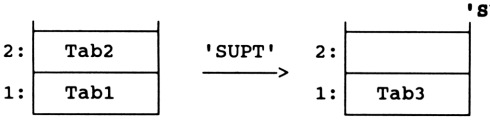
Si 'Tableau' est un vecteur alors 'Col' est égal au nombre de ses coefficients, et 'Lig' est égal à 1.

Aucune difficulté: On prend la dimension par **SIZE** (qui donne une liste à un élément dans le cas d'un vecteur et à deux éléments dans le cas d'une matrice). Si la liste est de longueur 1 , on place 1 au niveau 2.

Texte du programme 'DIMT'	Commentaires
« SIZE OBJ→ IF 1 == THEN 1 SWAP END »	Taille du tableau Libérer cette liste sur la pile. S'il s'agit d'un vecteur, on place 1 au niveau 2.

Remarque:
Le programme « **DIMT** * » calcule le nombre de coefficients du tableau (que ce tableau soit un vecteur, ou une matrice).
On peut obtenir ce résultat directement avec le programme:
« SIZE OBJ→ IF 2 == THEN * END »

Exercice 4 (##) : Superposition de deux tableaux.

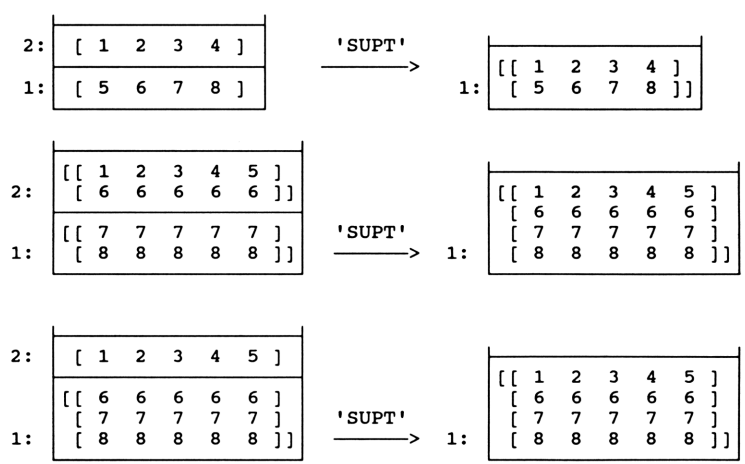


'SUPT' superpose les tableaux Tab1 et Tab2, qui doivent avoir le même nombre de colonnes, en un tableau Tab3 (où Tab2 apparaît au dessus de Tab1).

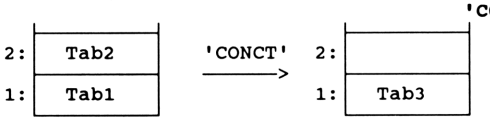
Le programme **'SUPT'** utilise le programme **'DIMT'** de l'exercice précédent, afin de contrôler les dimensions des tableaux 'Tab1' et 'Tab2' (on doit en particulier s'assurer qu'ils ont le même nombre de colonnes, sinon la pile doit garder son aspect initial).

Texte du programme 'SUPT'	Commentaires
« DUP DIMT 4 PICK DIMT → t1 L1 c1 L2 c2 « IF c1 c2 == THEN OBJ→ DROP t1 OBJ→ DROP L1 L2 + c1 2 →LIST →ARRY ELSE t1 END » »	Dimensions de 'Tab1'. Dimensions de 'Tab2'. t1 = Tableau 'Tab1' c1, L1 = nombre de colonnes , et de lignes de 'Tab1' (idem pour c2 ,L2 :dimensions de 'Tab2'). Si même nombre de colonnes: On libère 'Tab2' sur la pile, On libère 'Tab1' sur la pile, On crée la liste { L1+L2,c1 } puis le nouveau tableau. Sinon, on remet la pile dans son état d'origine.

Exemples: (en moins d'une seconde)



Exercice 5 (###) : Concaténation de deux tableaux.



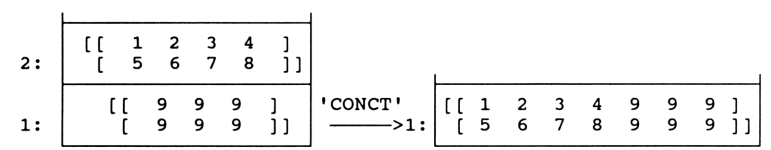
'CONCT' "accole" les tableaux Tab1 et Tab2, qui doivent avoir le même nombre de lignes , en un tableau Tab3 (où Tab2 apparaît à gauche de Tab1).

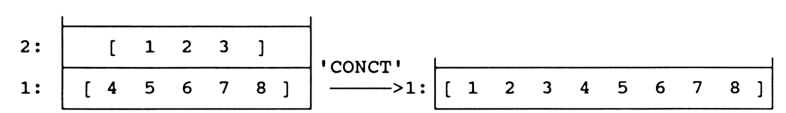
Il suffit ici de transposer les tableaux 'Tab1' et 'Tab2', de placer le transposé de 'Tab2' au dessus du transposé de 'Tab1' (à l'aide du programme **'SUPT'** décrit précédemment) , et de transposer le tableau obtenu.
Néanmoins, si 'Tab1' et 'Tab2' sont des vecteurs, il faut procéder de manière directe (libérer 'Tab2' sur la pile, puis 'Tab1' et en faire une seul vecteur)
Si 'Tab1' et 'Tab2' n'ont pas le même nombre de lignes, on laisse la pile en l'état.

Texte du programme 'CONCT'	Commentaires

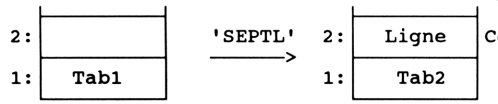
« DUP DIMT
4 PICK DIMT
→ t1 L1 c1 L2 c2
« IF L1 L2 == THEN
IF L1 1 == THEN
OBJ→ DROP
t1 OBJ→ DROP
c1 c2 + →ARRY
 	
ELSE
TRN
t1 TRN
SUPT TRN
END
ELSE t1 END
»
»
	Dimensions de 'Tab1'.
	Dimensions de 'Tab2'.
	Comme dans 'SUPT'.
	Si même nombre de lignes:
	Si ce sont deux vecteurs:
	'Tab2' libéré sur la pile,
	'Tab1' libéré sur la pile,
	On en fait un seul vecteur
	ayant c1+c2 coefficients.
	Sinon:
	On transpose 'Tab2',
	On transpose 'Tab1',
	On superpose et on transpose
	
	Sinon on remet la pile en l'état

Exemples: (en moins d'une seconde)





Exercice 6 (***) : Séparer un tableau de sa ligne supérieure.

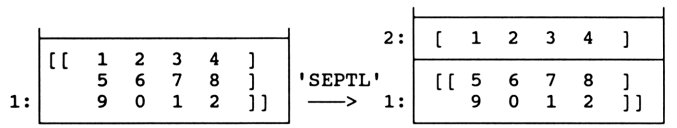


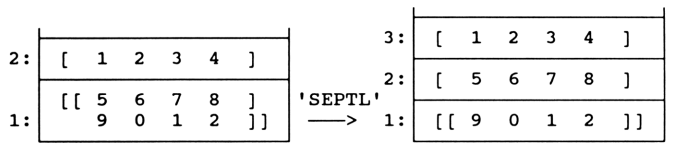
'SEPTL' isole la 1-ère ligne du tableau 'Tab1'. Cette ligne est placée au niveau 2, et ce qui reste du tableau (Tab2) est placé au niveau 1.

Dans le cas où 'Tab1' est une matrice à une ligne, où un vecteur, le schéma fonctionnel ci-dessus est incorrect puisqu'aucun objet n'est placé au niveau 2.
Il convient ici de traiter convenablement les différents cas particuliers ('Tab1' peut être un vecteur, ou une matrice à une seule ligne, ou une matrice à plusieurs lignes).

Texte du programme 'SEPTL'	Commentaires
« OBJ→ OBJ→ IF 1 == THEN →ARRY ELSE → c « 1 - IF DUP THEN c 2 →LIST →ARRY → t « c →ARRY t » ELSE DROP c →ARRY END » END »	Taille de 'Tab1'. S'il s'agissait d'un vecteur, on le remet dans sa forme initiale Sinon: c = nombre de colonnes. Nombre de lignes - 1. S'il y avait au moins 2 lignes on crée la tableau 'Tab2', on en fait la variable t, on crée la ligne sup. et on remet 'Tab2' sur la pile. Sinon, la matrice uni-ligne 'Tab1' devient un vecteur.

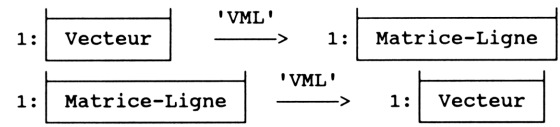
Exemples: (moins d'une seconde à chaque opération)





Une application supplémentaire de **'SEPTL'** transforme alors la matrice uni-ligne du niveau 1 en le vecteur [9 0 1 2]. (et d'autres appels de **'SEPTL'** ne modifieront plus la pile)

Exercice 7 (**): Transformation réversible d'un vecteur en matrice à une seule ligne.



'VML' transforme de façon réversible une matrice-ligne en le vecteur correspondant.

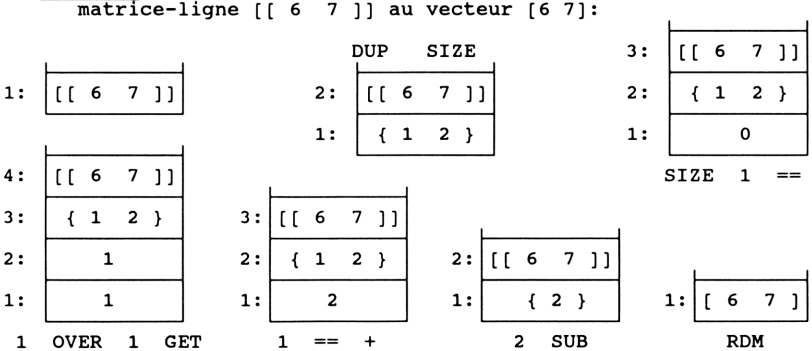
Il suffit ici de considérer la taille du tableau initial, sous forme d'une liste obtenue par **SIZE**:
* Si cette taille est { n }, il s'agit d'un vecteur ayant n coefficients. Il suffit de transformer cette liste en la liste { 1 n }.
* Si cette taille est { n p }, il s'agit d'une matrice ayant n lignes et p colonnes.
  Soit n>1, et alors cette liste ne doit pas être modifiée.
  Soit n=1, et cette liste doit devenir { p }.

On redimensionne alors le tableau initial avec la liste obtenue.

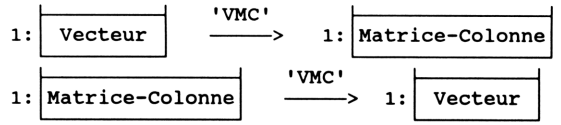
Texte du programme 'VML'	Commentaires
« DUP SIZE IF DUP SIZE 1 == THEN 1 SWAP + ELSE 1 OVER 1 GET 1 == + 2 SUB END RDM »	Liste donnant la taille. S'il s'agit d'un vecteur, on ajoute 1 devant la liste { n } donnant sa taille. Sinon, on transforme la liste { n p } en { p } si n=1. On redimensionne avec la liste ainsi corrigée.

Remarque:
Dans le listing ci-dessus, il y a une petite astuce dans le cas où le tableau initial est une matrice (astuce qui évite une deuxième instruction **IF .. THEN .. END**).

Exemple: Voyons les différentes étapes qui conduisent de la matrice-ligne [[6 7]] au vecteur [6 7]:



Exercice 8 (**): Transformation réversible d'un vecteur en matrice à une seule colonne.



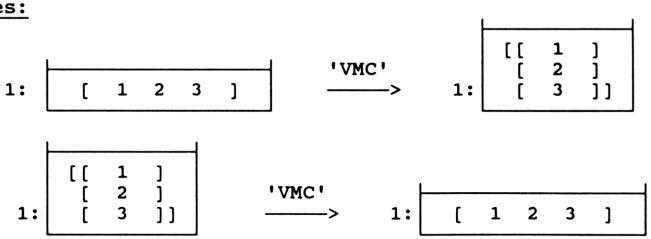
'VMC' transforme de façon réversible une matrice-colonne en le vecteur correspondant.

Comme dans **'VML'**, il faut modifier éventuellement la liste, obtenue par **SIZE**, et qui donne la taille du tableau:
Si cette liste s'écrit { n }, il s'agit d'un vecteur ayant n coefficients, et on transforme cette liste en { n 1 }.
Si cette liste s'écrit { n p }, il s'agit d'une matrice ayant p colonnes, et on transforme cette liste en { n } si p=1.
On redimensionne alors le tableau initial (au moyen de **RDM**) avec la liste ainsi obtenue.

Texte du programme 'VMC'	Commentaires
« DUP SIZE IF DUP SIZE 1 == THEN 1 + ELSE 1 1 3 PICK 2 GET 1 ≠ + SUB END RDM »	Liste donnant la taille. S'il s'agit d'un vecteur, on ajoute 1 en fin de liste. sinon la liste { n p } devient { n } si p=1. On redimensionne le tableau.

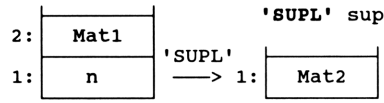
On remarquera là encore la séquence
" 1 1 3 PICK 2 GET 1 ≠ + SUB "
qui modifie la liste { n p } donnant la taille d'une matrice en la transformant en { n } si p=1.
Un autre possibilité aurait été d'écrire:
" IF DUP 2 GET 1 == THEN 1 1 SUB END ".

Exemples:



TABLEAUX: Corrigés des Exercices.

Exercice 9 (***): Supprimer une ligne dans une matrice.

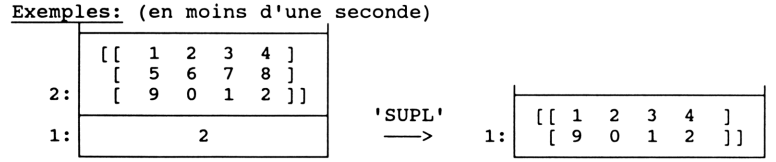


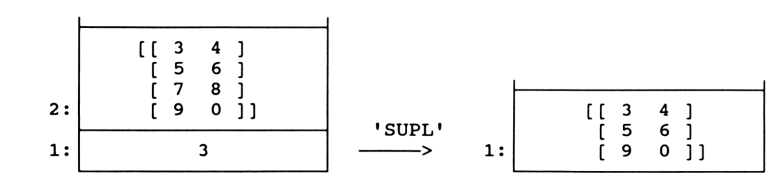
'SUPL' supprime la n-ème ligne de la matrice 'Mat1' placée au niveau 2.
La matrice modifiée, 'Mat2', est obtenue au niveau 1.

Si on note C et L le nombre de colonnes et le nombre de lignes de la matrice (notée A), alors le dernier élément de la n-ème ligne (celle qu'il faut supprimer) est suivi dans A par (L-n)*c éléments.
Si on libère le tableau sur la pile, cet élément se situera donc au niveau (L-n)*c+1.
Il suffira alors de faire descendre cet élément au niveau 1 avant de faire un **'DROP'**.
Sur la pile, l'avant dernier élément de la ligne à supprimer a alors pris la place de celui qui vient d'être détruit.
On continue donc (boucle tournant C fois) la destruction des éléments de la ligne visée.
Il reste alors à reconstituer la tableau avec les éléments qui restent (N-1 lignes et toujours C colonnes).

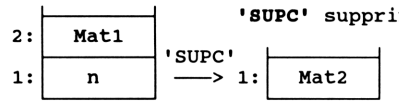
Texte du programme 'SUPL'	Commentaires
« OVER SIZE EVAL → n L c « L n - c * 1 + → h « OBJ→ DROP 1 c START h ROLL DROP NEXT L 1 - c 2 →LIST →ARRY » » »	n = numéro de ligne à détruire. L,c=nombre de lignes,de colonnes de la matrice. h = (L-n)*c+1 = hauteur sur la pile de l'élément à détruire. On libère le tableau sur la pile et on détruit chacun des élts de la ligne considérée. Taille du nouveau tableau. On reconstitue le tableau.

Exemples: (en moins d'une seconde)





Exercice 10 (##): Supprimer une colonne dans une matrice.



'SUPC' supprime la n-ème colonne de la matrice 'Mat1' placée au niveau 2. La matrice modifiée, 'Mat2', est obtenue au niveau 1.

Il y a une solution très simple qui consiste à utiliser **'SUPL'**: Détruire la n-ème colonne de 'Mat1' c'est en effet détruire la n-ème ligne de la transposée de 'Mat1'.

Texte du programme 'SUPC' (Première version)	Commentaires
« SWAP TRN SWAP SUPL TRN »	On place 'Mat1' sous n. On transpose 'Mat1'. On replace n au niveau 1. On efface la n-ème ligne de la transposée de 'Mat1', et on re-transpose.

Autre méthode:

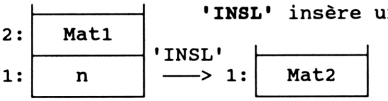
Il est possible d'écrire un programme dans le même esprit que **'SUPL'**. On libère le tableau sur la pile, et on détruit successivement , après les avoir amenés au niveau 1 de la pile, les éléments de la colonne à éliminer.

La différence avec **'SUPL'** est que les éléments à détruire ne sont en principe pas consécutifs sur la pile.

De façon précise, la position sur la pile du k-ème terme de la colonne à éliminer (la n-ème) est (en notant C et L le nombre de colonnes et de lignes de la matrice de départ): (C-n) + (L-k)*C + 1.

Texte du programme 'SUPC' (Deuxième version)	Commentaires
« OVER SIZE EVAL → n L c « OBJ→ DROP 'c-n+(L-1)*c+1' EVAL 'c-n+1' EVAL FOR i i ROLL DROP c NEG STEP L c 1 - 2 →LIST →ARRY » » »	n=numéro de colonne à détruire. L,c=nombre de lignes,de colonnes de la matrice. On libère le tableau sur la pile Position la plus haute. Position la plus basse. On descend l'élément visé au niveau 1 et on le détruit. On diminue de "c" la position. Taille du nouveau tableau. On reconstitue le tableau.

Exercice 11 (***): Insérer une ligne de zéros dans une matrice.

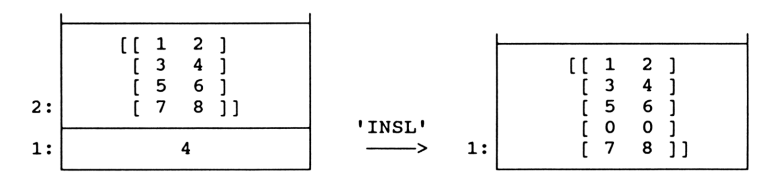


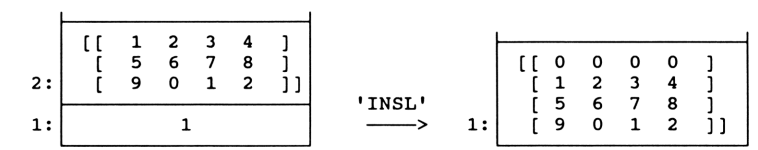
'INSL' insère une ligne de zéros dans la matrice 'Mat1' placée au niveau 2. Cette ligne de zéros devient la n-ème ligne de la matrice modifiée , 'Mat2', obtenue au niveau 1.

Dans l'esprit et dans la lettre , la solution est très proche de celle du programme **'SUPL'** (exercice 9).
Si on note L et C le nombre de lignes et le nombre de colonnes de la matrice 'Mat1', la ligne insérée va être suivie d'un nombre de coefficients égal à (L-n+1)*C (c'est-à-dire L-n+1 lignes). Il suffit donc de libérer le tableau 'Mat1' sur la pile, et, C fois de suite, de placer un zéro au niveau (L-n+1)*C+1 de la pile.
On termine en recréant le tableau avec ses nouvelles dimensions (une ligne de plus).

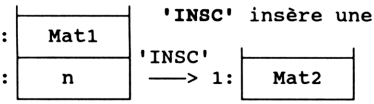
Texte du programme 'INSL'	Commentaires
« OVER SIZE EVAL → n L c « L n - 1 + c * 1 + → h « OBJ→ DROP 1 c START 0 h ROLLD NEXT L 1 + c 2 →LIST →ARRY » » »	n=position de la ligne à insérer. L,c=nombre de lignes, de colonnes h=(L-n+1)*c+1=hauteur sur la pile où mettre des coefficients nuls On libère le tableau initial. C fois de suite, on place 0 au niveau h. Nouvelles dimensions du tableau (L+1 lignes, C colonnes). On le reconstitue.

Exemple: (en moins d'une seconde)





Exercice 12 (##): Insérer une colonne de zéros dans une matrice.



'INSC' insère une colonne de zéros dans la matrice 'Mat1' placée au niveau 2. Cette colonne de zéros devient la n-ème colonne de la matrice modifiée , 'Mat2', obtenue au niveau 1.

Il y a une solution très simple qui consiste à utiliser **'INSL'**: On transpose 'Mat1'. Dans la matrice obtenue, on insère une ligne de zéros en n-ème ligne. On transpose ensuite la matrice trouvée.

Texte du programme 'INSC' (Première version)	Commentaires
« SWAP	On place 'Mat1' sous n.
TRN	On transpose 'Mat1'.
SWAP	On replace n au niveau 1.
INSL	On insère une ligne de zéros à la n-ème ligne de la transpo-sée de 'Mat1'.
TRN	On transpose le tableau obtenu.
»	

Il y a une deuxième méthode comme on l'a vu pour **'INSL'**, qui consiste à libérer 'Mat1' sur la pile, et à placer des zéros aux bonnes places (ici non consécutives comme pour **'INSL'**) avant de former le tableau avec ses nouvelles dimensisons.

Texte du programme 'INSC' (Deuxième version)	Commentaires
« OVER SIZE EVAL	n=position de la colonne insérée.
→ n L c	L,c=nombre de lignes, de colonnes
« OBJ→ DROP	On libère le tableau sur la pile.
'c*L-n+2'	Position la plus haute.
'c-n+2'	Position la plus basse. .
FOR i	On place 0 au niveau calculé,
0 i ROLLD	de la position la plus haute à
c NEG STEP	la position la plus basse.
L c 1 + 2 →LIST	Nouvelles dimensions du tableau
→ARRY	(L lignes, C+1 colonnes). On le
»	reforme sur la pile.
»	

Troisième méthode:
Il y plus original: pour insérer une colonne nulle en n-ème colonne de 'Mat1' (dont on suppose qu'elle est de format { L, C }, il suffit de multiplier 'Mat1' par une matrice B de format { C C+1 } définie de la manière suivante:
B est la matrice identité de format { C , C }, dans laquelle on insère une colonne nulle en n-ème colonne.

Exemple: avec C=3, et n=2 , la matrice B s'écrit:

```
          [[ 1  0  0  0 ]
     B =   [ 0  0  1  0 ]
           [ 0  0  0  1 ]]          [[ 5  4  7 ]
Supposons que L=2 et que  Mat1 =     [ 2  9  1 ]]
```

Alors le produit Mat1*B est égal à:

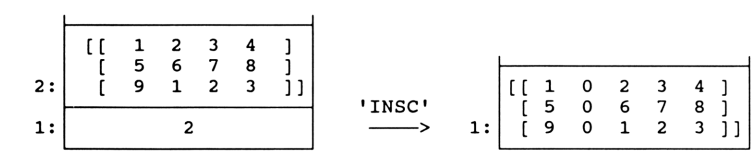
```
[[ 5  0  4  7 ]
 [ 2  0  9  1 ]].
```

On a ainsi inséré une colonne de zéros en deuxième colonne de la matrice Mat1.

Le problème est ici de créer la matrice B. Il suffit ici de créer un tableau constant de format { L C }, rempli de zéros, et de placer des 1 là où il le faut.
Plus précisément, en notant toujours n le numéro de colonne où insérer une colonne nulle, les coefficients égaux à 1 se placent aux positions { i , i + (i≥n) } où "i≥n" vaut 1 si i est supérieur ou égal à n, 0 sinon.
On obtient donc le programme suivant:

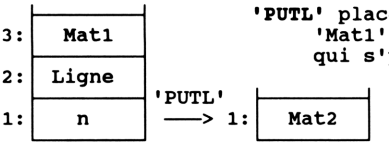
Texte du programme 'INSC' (Troisième version)	Commentaires
`« OVER SIZE EVAL`	n=position de la ligne à insérer.
`→ n L c`	L,c=nombre de lignes, de colonnes
`« c DUP 1 + 2 →LIST`	Format de la matrice à multiplier
`0 CON`	à 'Mat1'.
`1 c FOR i`	Après avoir rempli ce tableau de
`i DUP DUP n ≥ +`	zéros, on place des 1 aux bons
`2 →LIST 1 PUT`	endroits.
`NEXT`	
`*`	Produit de 'Mat1' par la matrice
`»`	ainsi construite.
`»`	

Exemple:



Remarque:
On vérifie en appelant un grand nombre de fois une version donnée de **'INSC'** les temps de calcul suivants (avec le schéma fonctionnel ci-dessus):
Première version : 0.16 s
Deuxième version : 0.14 s
Troisième version : 0.35 s

Exercice 13 (**): Placer une ligne donnée dans une matrice donnée.



'PUTL' place, à la n-ème ligne de la matrice 'Mat1', et en remplacement de la ligne qui s'y trouvait précédemment, la ligne représentée au niveau 2.

La matrice modifiée 'Mat2' est obtenue au niveau 1.

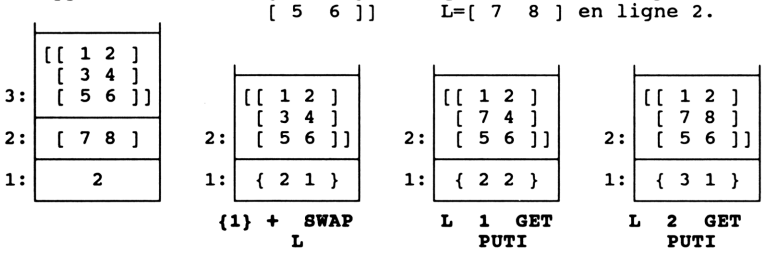
('Ligne' peut être un vecteur ou une matrice ligne).

La solution est assez simple. On place la ligne dans une variable locale L, et on place la liste { n 1 } sous la matrice 'Mat1' (cette liste représente la première position d'écriture).

Il suffit alors d'écrire aux positions successives de 'Mat1' les éléments successifs de la ligne 'L' (utilisation de **PUTI**).

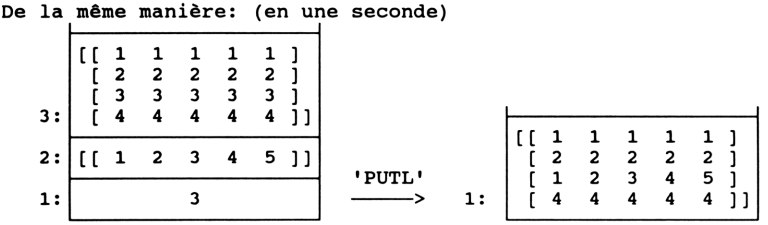
Texte du programme 'PUTL'	Commentaires
« { 1 } + SWAP → L « 1 3 PICK SIZE 2 GET FOR i L i GET PUTI NEXT DROP » »	L = vecteur à placer. Boucle où i prend autant de va--leurs qu'il y a de colonnes. On place l'élément suivant de L Effacer le pointeur désignant la position où on écrit.

Exemples: Supposons 'Mat1' = [[1 2] [3 4] [5 6]] et que l'on veuille placer L=[7 8] en ligne 2.

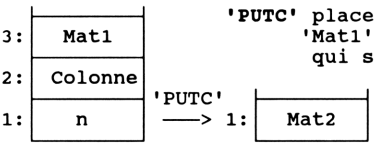


{1} + SWAP L **L 1 GET PUTI** **L 2 GET PUTI**

De la même manière: (en une seconde)



Exercice 14 (##): Placer une colonne donnée dans une matrice donnée.

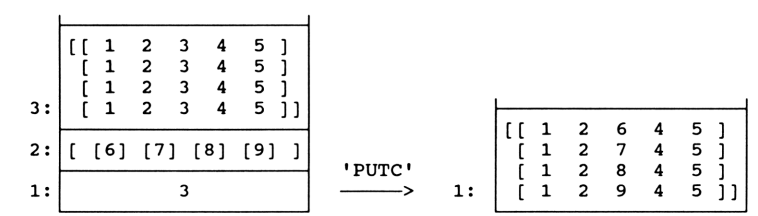


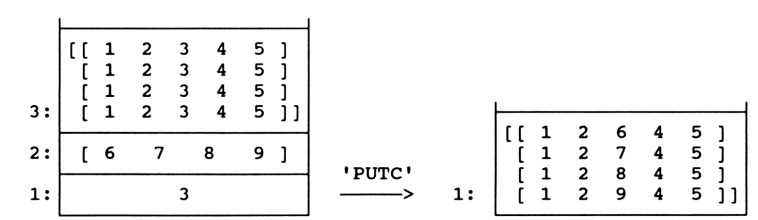
'PUTC' place, à la n-ème colonne de la matrice 'Mat1', et en remplacement de la colonne qui s'y trouvait précédemment, la colonne représentée au niveau 2 (et qui est un vecteur ou une matrice à une colonne).
La matrice modifiée 'Mat2' est obtenue au niveau 1.

Le plus simple est ici d'utiliser le programme **'PUTL'**. Placer une colonne en n-ème position dans 'Mat1', c'est en effet placer une ligne en n-ème position dans la transposée de 'Mat1', avant de transposer cette résultat obtenu.
Il faut également prévoir le cas où la colonne est donnée sous forme de matrice unicolonne. Son format s'écrit alors {n 1}.
Sur ce format il suffit : **1 1 SUB RDM**
pour transformer cette matrice en vecteur à n termes.
Si la colonne est au départ un vecteur (format {n}) la séquence précedente est sans effet, ce qui est correct.

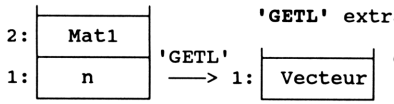
Texte du programme 'PUTC'	Commentaires
« ROT TRN	On place 'Mat1' au niveau 1 et on la transpose.
ROT DUP SIZE 1 1 SUB RDM ROT PUTL	On place la colonne au niveau 1 et on en fait un vecteur. On met n au niveau 1, et on place la colonne dans le tableau.
TRN »	On retranspose.

Exemples:





Exercice 15 (**): Extraire une ligne donnée d'une matrice donnée.



'GETL' extrait la n-ème ligne de la matrice 'Mat1'. Le résultat est un vecteur obtenu au niveau 1.

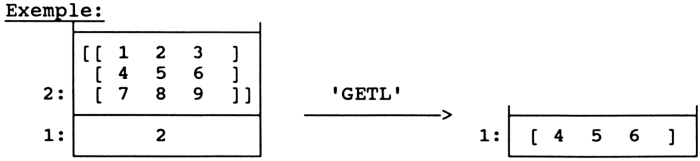
Première solution:
Supposons que 'Mat1' soit de format {L C} (L lignes,C colonnes):
Il suffit de créer un vecteur V de format {L}, dont tous les coefficients soient nuls ,sauf le n-ème (égal à 1).
Alors le produit de la transposée de Mat1 par V fournit le n-ème vecteur colonne de la transposée de Mat1, c'est-à-dire le n-ème vecteur ligne de Mat1 (ce qui est le résultat souhaité).

Texte du programme 'GETL' (Première version)	Commentaires
« → n « TRN DUP SIZE 2 2 SUB 0 CON n 1 PUT * » »	n = numéro de ligne à extraire. On transpose 'Mat1'. Liste { L } Vecteur V,nul, à L coefficients. On place 1 en n-ème position de V et on multiplie la transposée de 'Mat1' par V.

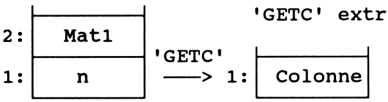
Deuxième solution:
Il suffit de lire un à un les éléments de la n-ème ligne de 'Mat1' (instruction **GETI**) en les laissant sur la pile, puis de former un vecteur avec les éléments ainsi lus.

Texte du programme 'GETL' (Deuxième version)	Commentaires
« { 1 } + OVER SIZE 2 GET → c « 1 c START GETI 3 ROLLD NEXT DROP2 c →ARRY » »	{n 1} = 1ère position de lecture. c = nombre de colonnes de 'Mat1'. De 1 à c, on lit chaque élément de la n-ème ligne et on le met au niveau 3. On élimine 'Mat1' et le pointeur indiquant la position. On forme un vecteur de longueur c avec les éléments lus.

Exemple:



Exercice 16 (**): Extraire une colonne donnée d'une matrice donnée.



'GETC' extrait la n-ème colonne de la matrice 'Mat1'.Le résultat est une matrice uni-colonne, obtenue au niveau 1.

Première solution:
La solution est tout à fait analogue à celle de la première version de **'GETL'**:

Si 'Mat1' est de format { L C } (L lignes, C colonnes), alors on crée une matrice uni-colonne B de format { C 1 }, dont tous les coefficients sont nuls sauf le n-ème (qui vaut 1), et on effectue le produit Mat1*B. On obtient alors la n-ème colonne de 'Mat1', sous forme de matrice colonne format { L 1 }.

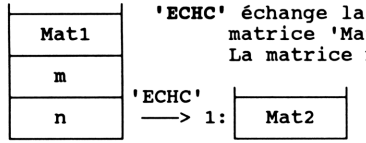
Texte du programme 'GETC' (Première version)	Commentaires
« OVER SIZE 1 1 PUT 0 CON TRN SWAP 1 PUT * »	On crée la liste { 1 C } où C est le nombre de colonnes de 'Mat1' Matrice nulle de format { C 1 } On met à 1 le n-ème coefficient de cette matrice B, et on fait le produit Mat1*B.

Deuxième solution:
On peut utiliser **'GETL'** en se disant qu'extraire la n-ème colonne de 'Mat1', c'est extraire la n-ème ligne de la transposée de 'Mat1' (Le résultat devant cependant passer du format vecteur au format matrice uni-colonne).

Texte du programme 'GETC' (Deuxième version)	Commentaires
« SWAP TRN SWAP GETL DUP SIZE 1 + RDM »	On transpose 'Mat1'. n-ème ligne de la transposée. On la transforme en matrice à une seule colonne.

Exemple:

Exercice 17 (***): Echanger deux colonnes dans une matrice.



'ECHC' échange la n-ème et la m-ème colonnes de la matrice 'Mat1' placée au niveau 3. La matrice modifiée 'Mat2' est obtenue au niveau 1 de la pile.

Il y a ici une méthode originale qui consiste à multiplier 'Mat1' par une certaine matrice J:
Si 'Mat1' est de format { L C } (L lignes, C colonnes), on crée la matrice identité d'ordre C. Dans cette matrice, on annule les deux termes diagonaux d'indices { m m } et { n n }, et on place 1 aux positions d'indice { m n } et { n m } (ce qui revient en fait à échanger, dans la matrice identité, les colonnes d'indice m et n).
On a ainsi obtenu une matrice J, et le produit Mat1*J effectue l'échange des colonnes d'indice m et n dans 'Mat1'.

Si par exemple,on veut échanger les colonnes 2 et 4 de la matrice

```
          [[ 1 2 3 4 ]      alors on crée la      [[ 1 0 0 0 ]
Mat1 =     [ 5 6 7 8 ]      matrice identité  I =  [ 0 1 0 0 ]
           [ 9 0 1 2 ]]                           [ 0 0 1 0 ]
                                                  [ 0 0 0 1 ]]
```

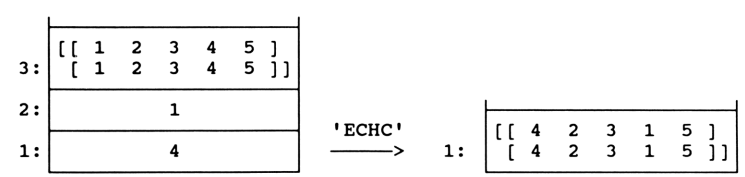
On applique à I les opérations décrites ci-dessus et on obtient J =

```
[[ 1 0 0 0 ]  et le produit Mat1*J  [[ 1 4 3 2 ]
 [ 0 0 0 1 ]       est égal à =      [ 5 8 7 6 ]
 [ 0 0 1 0 ]                         [ 9 2 1 0 ]]
 [ 0 1 0 0 ]]
```

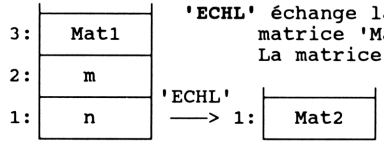
On a ainsi échangé les colonnes 2 et 4 de 'Mat1'.

Texte du programme 'ECHC'	Commentaires
« → m n « DUP SIZE 2 GET IDN m m 2 →LIST 0 PUT n n 2 →LIST 0 PUT m n 2 →LIST 1 PUT n m 2 →LIST 1 PUT * » »	m et n : colonnes à échanger. Nombre C de colonnes de 'Mat1'. Matrice Identité d'ordre C. On y met: 0 à la position { m m }, 0 à la position { n n }, 1 à la position { m n }, 1 à la position { n m }. Et on fait le produit.

Exemple: (en une seconde).



Exercice 18 (##): Echanger deux lignes dans une matrice.

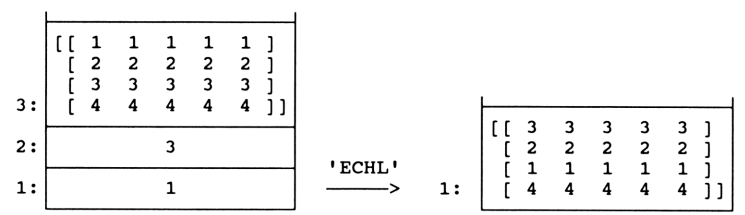


'ECHL' échange la n-ème et la m-ème lignes de la matrice 'Mat1' placée au niveau 3. La matrice modifiée 'Mat2' est obtenue au niveau 1 de la pile.

On pourrait, comme dans **'ECHC'** utiliser un produit matriciel. La même matrice "J" pourrait être utilisée, mais il faudrait faire le produit J*Mat1 au lieu de Mat1*J.
Il est en fait beaucoup plus simple d'utiliser **'ECHC'** en notant qu'échanger les lignes m et n de 'Mat1', c'est échanger les colonnes m et n de la transposée de 'Mat1',avant de transposer à nouveau.

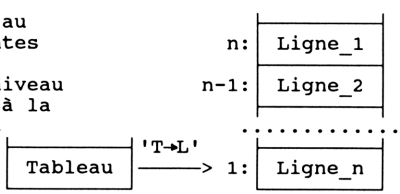
Texte du programme 'ECHL'	Commentaires
« ROT TRN 3 ROLLD ECHC TRN »	'Mat1' au niveau 1. On la transpose. Elle repart au niveau 3. On échange les colonnes m et n. On transpose le résultat obtenu.

Exemple: (en une seconde)



Exercice 19 (***) : Séparer un tableau en ses différentes lignes.

'T→L' sépare le tableau placé au niveau 1 en ses différentes lignes.
Chaque ligne est placée à un niveau de la pile correspondant à la position qu'elle occupait dans le tableau.
Ainsi la première ligne du tableau est placée au plus haut sur la pile.



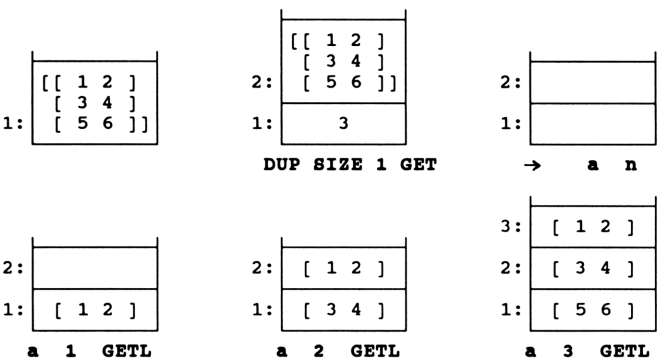
Nous allons voir trois solutions très différentes de ce problème.

Première solution:

C'est la solution du moindre effort: on est capable depuis l'exercice 15 d'extraire une ligne d'un tableau, avec **'GETL'**. Il suffit donc d'extraire les différentes lignes une à une.

Texte du programme 'T→L' (Première version)	Commentaires
« DUP SIZE 1 GET → a n « 1 n FOR i a i GETL NEXT » »	a:le tableau, n:son nb de lignes. De i=1 à i=n, On extrait la i-ème ligne de a.

Voyons sur un exemple comment les choses se passent:



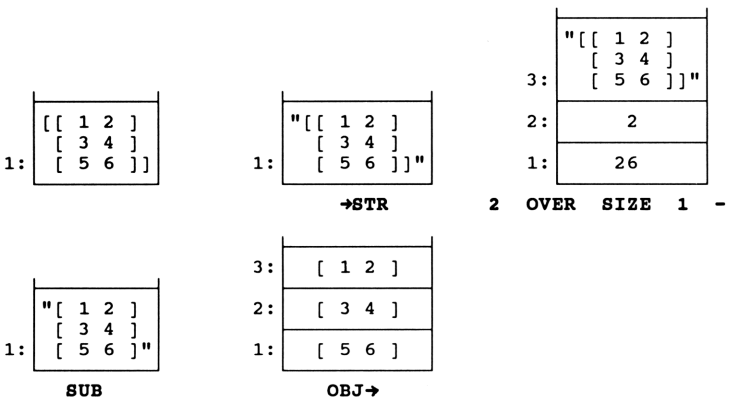
Deuxième solution:

On reprend ici un exercice du chapitre "chaînes de caractères" avec sa solution.

On transforme le tableau en chaîne de caractères (au moyen de l'instruction **→STR**), on efface de cette chaîne le caractère [initial et le caractère] final, avant d'évaluer cette chaîne par **OBJ→** (ce qui a pour effet de libérer les lignes du tableau sur la pile).

Texte du programme 'T→L' (Deuxième version)	Commentaires
« →STR 2 OVER SIZE 1 - SUB OBJ→ »	Tableau transformé en chaîne. On élimine le premier [et le dernier] , avant d'évaluer la chaîne obtenue.

Sur le même exemple que précédemment, on obtient:

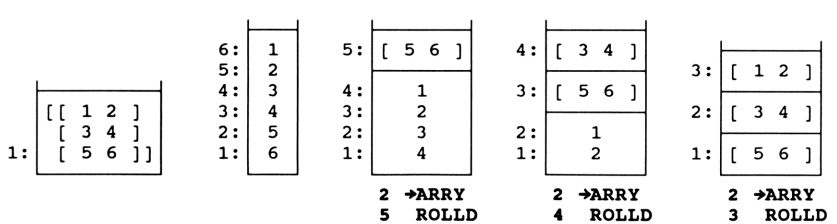


Troisième solution:

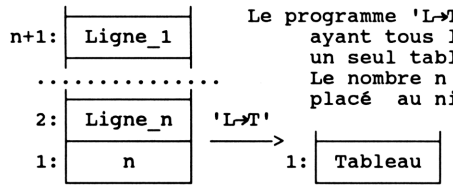
Ici, on commence par libérer le tableau sur la pile.
Si le tableau initial était de format { n p }, les p premiers nombres sur la pile sont les éléments de la dernière ligne.
Il suffit de reconstituer cette ligne par " **p →ARRY** ", avant de remonter cette ligne au dessus des autres coefficients.
On reconstitue chacune des lignes de cette manière, avant de la remonter au dessus des coefficients restants et des lignes déjà reconstituées.

Texte du programme 'T→L' (Troisième version)	Commentaires
« OBJ→ EVAL → n p « 'n*p-p+1' n FOR i p →ARRY i ROLLD 1 p - STEP » »	On libère le tableau. n,p = nb de lignes, de colonnes. Extrémités de la boucle. On recrée chaque ligne avant de la remonter sur la pile.

Avec toujours le même exemple, on obtient:



Exercice 20 (***) : Regrouper n lignes en un seul tableau.



Le programme 'L→T' regroupe n lignes (n vecteurs ayant tous le même nombre de coefficients) en un seul tableau de n lignes.
Le nombre n de lignes à regrouper doit être placé au niveau 1 de la pile.

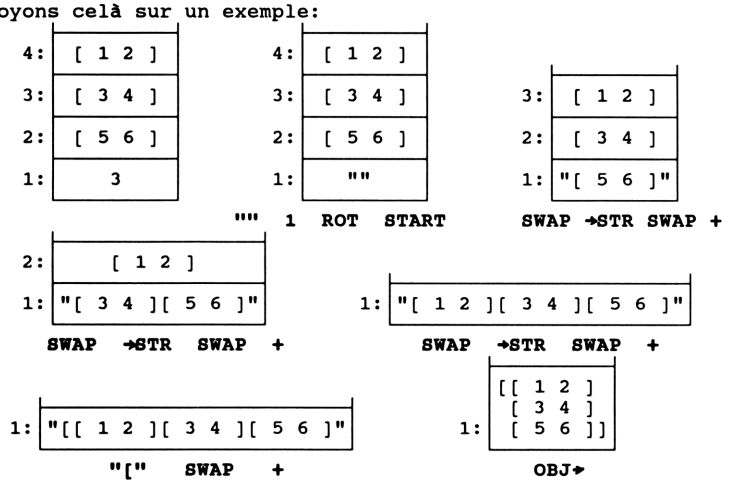
Comme pour l'exercice précédent, nous allons voir trois solutions distinctes.

Première solution:

On transforme chaque ligne en chaîne (instruction →STR) et on concatène ces chaînes les unes aux autres (en respectant leur ordre sur la pile). A la chaîne résultante, on ajoute le caractère "[" (en tête). Il suffit alors d'évaluer cette chaîne par OBJ→ pour obtenir le tableau souhaité.

Texte du programme 'L→T' (Première version)	Commentaires
« "" 1 ROT START SWAP →STR SWAP + NEXT "[" SWAP + OBJ→ »	Chaîne vide. Autant de fois que de lignes: Transformer une ligne en une chaîne et l'ajouter. Mettre un "[" en début de chaîne et évaluer cette chaîne.

Voyons celà sur un exemple:



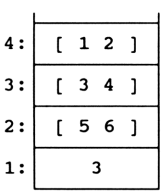
Deuxième solution:

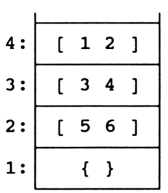
L'idée est de transformer chacune des lignes en la liste de ses éléments, d'ajouter ces listes entre elles, en respectant l'ordre des lignes sur la pile, et de créer le tableau final à partir des éléments regroupés dans la liste obtenue.

Comme pour la première solution, l'avantage est ici que tout le travail se fait pratiquement au niveau 1 de la pile, ce qui libère l'utilisateur du souci de comprendre où il doit lire telle ou telle ligne sur la pile, et où il doit libérer tels ou tels éléments (à ce sujet voir la solution suivante).

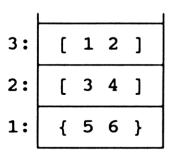
Texte du programme 'L→T' (Deuxième version)	Commentaires
« OVER SIZE 1 GET → n p « { } 1 n START SWAP OBJ→ DROP p →LIST SWAP + NEXT EVAL n p 2 →LIST →ARRY » »	n = nombre de lignes, p = nombre de colonnes. Liste vide. Autant de fois que de lignes: On libère une ligne sur la pile on en fait une liste que l'on ajoute en tête de la liste qui est sur la pile. On libère cette liste et on fait un tableau.

Voyons comment fonctionne cette deuxième solution, en reprenant l'exemple qui a servi à illustrer la première méthode:

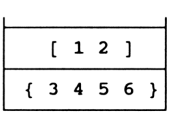




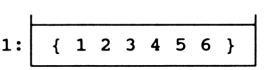
{ } 1 3
START



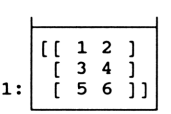
SWAP OBJ→ DROP
2 →LIST SWAP +



SWAP OBJ→ DROP
2 →LIST SWAP +



SWAP OBJ→ DROP
2 →LIST SWAP +



EVAL 3 2
2 →LIST
→ARRY

Troisième solution:

L'idée est ici de libérer un à un le contenu de chaque ligne sur la pile, pour regrouper ensuite tous les éléments obtenus en un seul tableau.

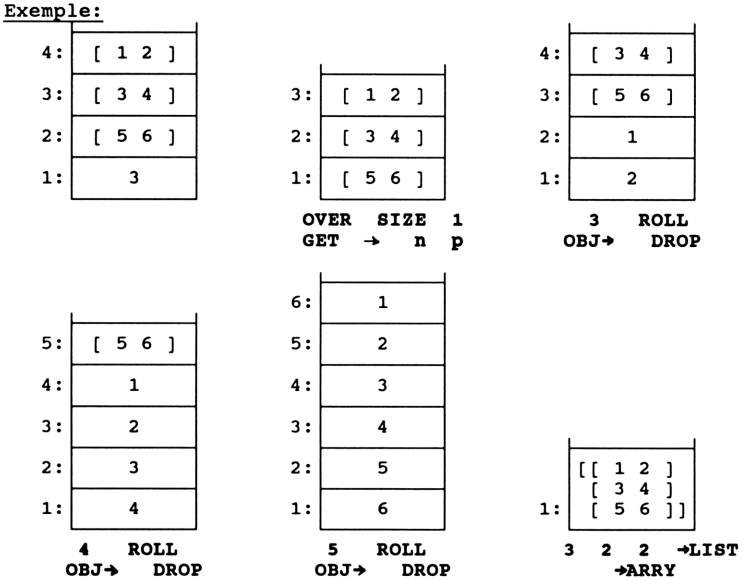
Il faut cependant penser à ce que les éléments ainsi déposés sur la pile doivent l'être en respectant l'ordre qui doit être le leur dans le tableau final.

Celà impose que l'on doit d'abord déposer sur la pile les éléments de la première ligne (qui au départ se trouve au niveau le plus haut), puis ceux de la deuxième, etc....

Il est donc important de bien comprendre à quel niveau se trouve la ligne à aller chercher (car ce niveau évolue avec la libération des lignes précédentes).

Texte du programme 'L→T' (Troisième version)	Commentaires
« OVER SIZE 1 GET → n p « n 'n*p-p+1' FOR i i ROLL OBJ→ DROP p 1 - STEP n p 2 →LIST →ARRY » »	n = nombre de lignes, p = nombre de colonnes. Extrémités de la boucle. On va chercher la ligne la plus haute, on la libère sur la pile. On forme le tableau.

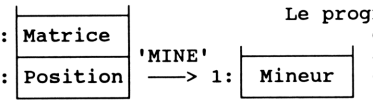
Exemple:



TABLEAUX: Corrigés des Exercices.

Exercice 21 (##): Mineur d'un coefficient d'une matrice carrée.

```
2: Matrice
                'MINE'
1: Position  ---->  1: Mineur
```



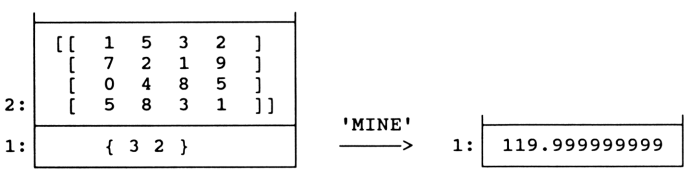
Le programme **'MINE'** calcule le mineur du coefficient de la matrice donnée au niveau 2, la position de ce coefficient dans cette matrice étant placée au niveau 1 (sous la forme d'une liste { n p } de 2 entiers: n pour le numéro de ligne et p pour le numéro de colonne).

La solution est ici extrêmement simple: grâce aux programmes **'SUPL'** et **'SUPC'** (voir exercices 9 et 10), on supprime la n-ème ligne et la p-ème colonne de la matrice. On calcule ensuite le déterminant de la matrice obtenue.

Texte du programme 'MINE'	Commentaires
« SWAP	La liste au-dessus de la matrice.
OVER 1 GET SUPL	On supprime la n-ème ligne.
SWAP 2 GET SUPC	On supprime la p-ème colonne.
DET	On calcule le déterminant.
»	

Exemple:

```
     [[  1  5  3  2  ]
      [  7  2  1  9  ]
      [  0  4  8  5  ]
2:    [  5  8  3  1  ]]
                          'MINE'
1:        { 3 2 }        ------>   1: 119.999999999
```



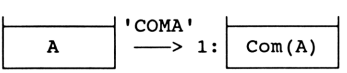
Cet exemple signifie que l'on a, dans la matrice initiale, détruit la 3-ème ligne et la 2-ème colonne, pour obtenir la matrice extraite:

```
        [[  1  3  2  ]
B =      [  7  1  9  ]
         [  5  3  1  ]]
```

On a ensuite calculé le déterminant (120) de B.

Exercice 22 (**): Comatrice d'une matrice carrée.

```
                'COMA'
1:    A      ---->  1: Com(A)
```



Le rôle du programme **'COMA'** est de calculer la comatrice de la matrice carrée A.

On calcule chacun des mineurs de la matrice avec le programme **'MINE'** de l'exercice précédent. Chacun de ces mineurs est éventuellement changé en son opposé,en fonction de la position correspondante dans la matrice. On regroupe ensuite les résultats en un seul tableau de même format que la matrice initiale.

Texte du programme 'COMA'	Commentaires
« DUP SIZE 1 GET → n	n=nombre de lignes (de colonnes)
« 1 n FOR i	De la ligne 1 à la ligne n,
1 n FOR j	De la colonne 1 à la colonne n,
DUP	Dupliquer la matrice,
i j 2 →LIST	Prendre le mineur du terme d'indice i,j et multiplier par (-1)^(i+j).
MINE	
-1 i j + ^ *	
SWAP	Placer ce mineur au-dessus de la matrice.
NEXT	
NEXT	
SIZE	Taille de la matrice.
→ARRY	C'est la taille de la comatrice.
»	
»	

Exemples: En 13 secondes:

```
     [[ 1  -1  2  1 ]                    [[ -6  0  3 -3  ]
      [ 0   2  1  1 ]                     [ -6 -2  3 -2  ]
      [-1   1  0  2 ]     'COMA'          [  6  1 -3  1  ]
1:    [ 1   0  2  0 ]]    ------>    1:   [  9  1 -6  4  ]]

     [[ 1   0  1  0 ]                    [[ -2  1  2  1  ]
      [ 1  -1  1  1 ]                     [  2 -1 -2 -1  ]
      [ 1   1  0  1 ]     'COMA'          [  2 -1 -2 -1  ]
1:    [ 1   0  0  2 ]]    ------>    1:   [ -2  1  2  1  ]]

     [[ 1   0  1  0 ]                    [[  0  0  0  0  ]
      [ 1  -1  1  1 ]                     [  0  0  0  0  ]
      [ 0   1  0 -1 ]     'COMA'          [  0  0  0  0  ]
1:    [ 2  -1  2  1 ]]    ------>    1:   [  0  0  0  0  ]]
```

Remarque pour les matheux:

Sur l'exemple 2, la matrice Com(A) est de rang 1 (ses colonnes sont proportionnelles entre elles), ce qui traduit le fait que la matrice A est de rang 4-1=3.

Sur l'exemple 3, la matrice Com(A) est nulle ce qui traduit le fait que la matrice A est de rang < 3. A est en fait de rang 2.

Autre possibilité:

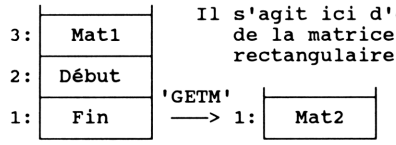
Si A est une matrice carrée et si A est sa comatrice, alors on a l'égalité: **t(A)*Com(A)=Det(A)*I** , où t(A) désigne la transposée de A et où I est la matrice identité d'ordre n.

On en déduit que, si A est inversible:

Com(A)=Det(A)*[t(A)]^(-1).

Le programme **« DUP DET SWAP TRN INV * »** permet donc de calculer la comatrice d'une matrice carrée A, quand A est inversible (il ne fonctionne donc pas sur les deux derniers exemples ci-dessus).

Exercice 23 (***): Extraire une sous-matrice d'une matrice.



Il s'agit ici d'extraire une sous-matrice 'Mat2' de la matrice 'Mat1'. 'Mat2' est le tableau rectangulaire délimité dans 'Mat1' par les positions 'Debut' et 'Fin'. Ces deux positions ont la forme d'une liste {n p} de 2 entiers (n pour l'indice de colonne et p pour l'indice de ligne).

'Début' désigne le coin supérieur gauche de la matrice à extraire et 'Fin' la coin inférieur droit.

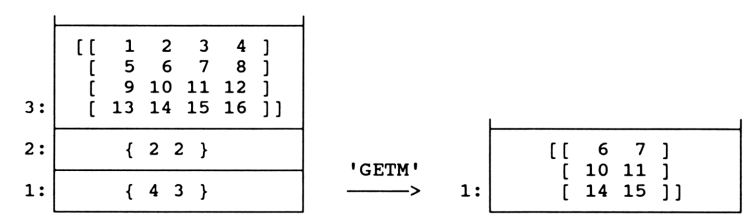
Première méthode:

On se contente de lire, aux positions adéquates, les éléments de la matrice initiale Mat1, avant de placer les éléments lus dans un tableau représentant la sous-matrice cherchée.

Texte du programme 'GETM' (Première version)	Commentaires

`«  +  OVER  SIZE  +`
`  EVAL`
`  → b  d  h  g  lig  col`
`  « h  b  FOR  i`
`  g  d  FOR  j`
`  DUP`
`  i  j  2  →LIST  GET`
`  SWAP`
`  NEXT`
`  NEXT`
`  DROP`
`  b  h  -  1  +`
`  d  g  -  1  +`
`  2  →LIST  →ARRY`
`  »`
`»`
	b,d= position du point bas,droit.
	h,g=position du point haut,gauche
	lig,col= dimensions de 'Mat1'.
	De la ligne h à la ligne b,
	De la colonne g à la colonne d,
	On duplique Mat1, avant d'en lire le terme d'indice {i,j} et on le place au dessus de Mat1.
	
	On détruit Mat1.
	Nb de lignes de la sous-matrice.
	Nb de colonnes de la sous-matrice
	On forme cette sous-matrice.

Exemple:



Deuxième méthode:
Il y a une solution qui consiste à effectuer un produit du type:
P*Mat1*Q, où P et Q sont deux matrices dont le rôle est le suivant: pour P: sélectionner les lignes voulues dans Mat1.
pour Q: sélectionner les colonnes voulues dans Mat1.

Si par exemple, Mat1 est égale à et si on veut obtenir la sous-matrice définie par le coin supérieur gauche en { 3 2 } et le coin inférieur droit en { 5 5 }, ce qui signifie qu'on choisit les lignes 3 à 5 et les colonnes 2 à 5, alors il faut multiplier Mat1 à gauche par P et à droite par Q, avec:

```
[[  1  2  3  4  5  6 ]
 [  7  8  9 10 11 12 ]
 [ 13 14 15 16 17 18 ]
 [ 19 20 21 22 23 24 ]
 [ 25 26 27 28 29 30 ]]
```

```
      [[ 0 0 1 0 0 ]
P =    [ 0 0 0 1 0 ]
       [ 0 0 0 0 1 ]]
```

```
                [[ 0 0 0 0 ]
                 [ 1 0 0 0 ]
et  Q =          [ 0 1 0 0 ]
                 [ 0 0 1 0 ]
                 [ 0 0 0 1 ]
                 [ 0 0 0 0 ]]
```

Le résultat de P*Mat1 est alors

```
[[ 13 14 15 16 17 18 ]
 [ 19 20 21 22 23 24 ]
 [ 25 26 27 28 29 30 ]
```

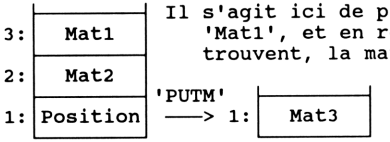
et le résultat de P*Mat1*Q est

```
[[ 14 15 16 17 ]
 [ 20 21 22 23 ]
 [ 26 27 28 29 ]]
```

Texte du programme 'GETM' (Deuxième version)	Commentaires
`« + OVER SIZE +`	b,d= position du point bas,droit.
`EVAL`	h,g=position du point haut,gauche
`→ b d h g lig col`	lig,col= dimensions de 'Mat1'.
`« b h - 1 + lig`	Tableau constant nul de format
`2 →LIST 0 CON`	{ b-h+1,lig }
`h b FOR i`	On y écrit des 1 aux positions
`i h - 1 + i`	{ i-h+1,i } avec h ≤ i ≤ b.
`2 →LIST 1 PUT`	On obtient ainsi la matrice P des
`NEXT`	explications ci-dessus.
`SWAP *`	On effectue la produit Mat1*P.
`col d g - 1 +`	Tableau constant nul de format
`2 →LIST 0 CON`	{ col, d-g+1 }
`g d FOR j`	On y écrit des 1 aux positions
`j j g - 1 +`	{ j, j-g+1 } avec g ≤ j ≤ d.
`2 →LIST 1 PUT`	On obtient ainsi la matrice Q des
`NEXT`	explications ci-dessus.
`*`	Produit P*Mat1*Q.
`»`	
`»`	

Remarque:
Sur un exemple comme celui donné page précédente, les temps de calculs sont les suivants:
* 0.6 seconde pour la première version de **'GETM'**.
* 1.1 seconde pour la deuxième version de **'GETM'**.
La 1ère version est donc plus courte et plus rapide, ce qui n'enlève rien à l'intérêt pédagogique de la 2ème version.

Exercice 24 (***): Placer une matrice à l'intérieur d'une autre.

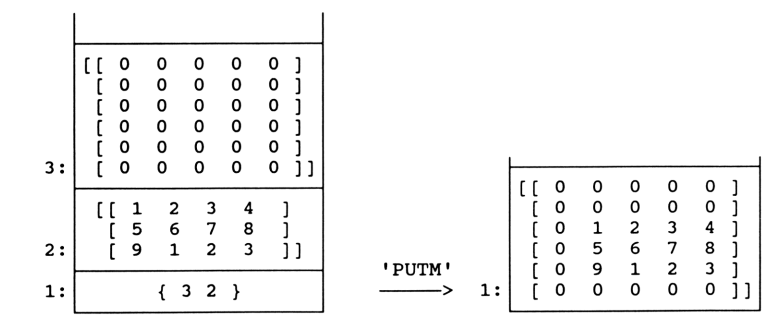


Il s'agit ici de placer, à l'intérieur de la matrice 'Mat1', et en remplacement des termes qui s'y trouvent, la matrice 'Mat2', en positionnant le coin supérieur gauche de 'Mat2' à la position de 'Mat1' donnée par une liste placée au niveau 1 (et qui a la forme { n p }, où n et p représentent respectivement un indice de ligne et un indice de colonne) .
La matrice modifiée, notée ici 'Mat3', est obtenue au niveau 1.
'Mat2' doit alors apparaître comme une sous-matrice de 'Mat3'.

On va ici utiliser la méthode la plus simple qui soit:
Lire chaque élément de 'Mat2', et le placer à la bonne position dans 'Mat1'.

Texte du programme 'PUTM'	Commentaires
« OVER SIZE + EVAL → mat2 h g L2 c2 « 1 L2 FOR i 1 c2 FOR j i h + 1 - j g + 1 - 2 →LIST mat2 i j →LIST GET PUT NEXT NEXT » »	mat2 = Matrice 'Mat2', de format { L2 c2 }. { h g }= position où on commence à écrire dans Mat1. De ligne en ligne sur Mat2, de colonne en colonne sur Mat2, ligne où écrire dans Mat1, colonne où écrire dans Mat1, on en fait une liste. Mat2 sur la pile, on y lit le coefficient placé en { i j }, et on le place dans Mat1. Colonne suivante de Mat2... Ligne suivante de Mat2...

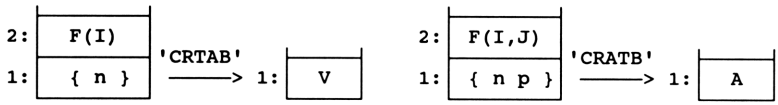
Exemple: en deux à trois secondes:



Exercice 25 (***): Créer un tableau dont le terme général répond à une formule donnée.

Premier cas: Création d'un vecteur V de longueur n dont le terme général Vi obéit à une formule Vi = F(I).

Deuxième cas: Création d'une matrice A de format {n p} dont le terme général Aij obéit à une formule Aij = F(I,J).



Il y a dans la solution ci-dessous une astuce qui permet d'éviter d'avoir à créer, pour l'occasion, deux variables globales I,J.

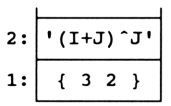
Il suffit de créer un sous programme, que l'on met dans une variable locale, qui prend deux valeurs I et J sur la pile, et qui calcule F(I,J).

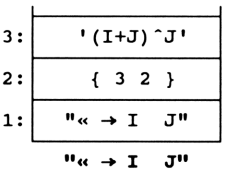
On se rend compte qu'il est nécessaire d'inclure la formule donnant F(I,J), dans ce sous-programme, en travaillant avec des chaînes de caractères et les instructions **→STR** et **OBJ→**.

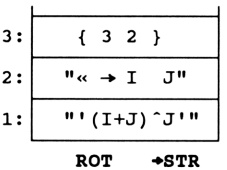
Reportez vous au listing ci-dessous, et à l'exemple détaillé qui suit pour bien comprendre comment fonctionne ce programme.

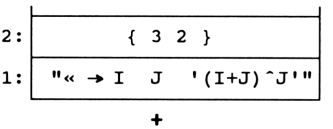
Texte du programme 'CRTAB'	Commentaires
`« "« → I J" ROT →STR + OBJ→ → d f « d OBJ→ DUP 1 ≠ DROPN → L c « 1 L FOR i 1 c FOR j i j f EVAL NEXT NEXT d →ARRY » » »`	Début du sous-programme à créer. On lui ajoute la chaîne représen--tant F,avant d'évaluer cette chaîne pour lui donner le forme programme. d = dimension du tableau à créer, f = formule (sous-programme). On place sur la pile le nombre de lignes et de colonnes. (dans le cas d'un vecteur, L = nb de coeff, et c = 1) De la 1ère à la dernière ligne, de la 1ère à la dernière colonne évaluer f pour les valeurs i,j Colonne suivante... Ligne suivante... On forme le tableau obtenu.

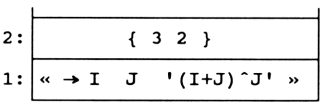
Supposons que nous voulions créer la matrice de format { 3 2 } (trois lignes et deux colonnes), dont le terme général soit donné par (I+J)^J (formule donnant le terme situé en ligne I et en colonne J).
La pile, pendant l'éxécution du programme **'CRTAB'**, a l'aspect suivant:

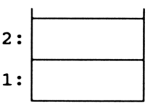


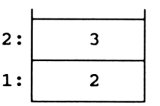


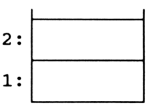


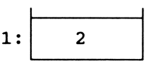


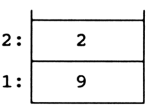


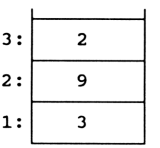


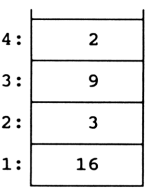


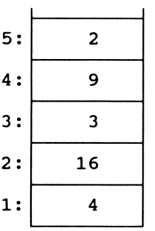


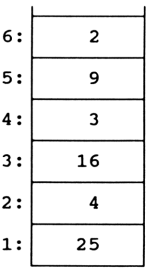


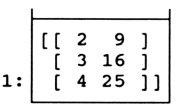




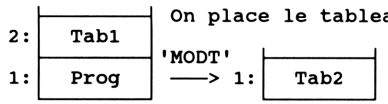








Exercice 26 (***): Application d'une même procédure à tous les éléments d'un tableau.

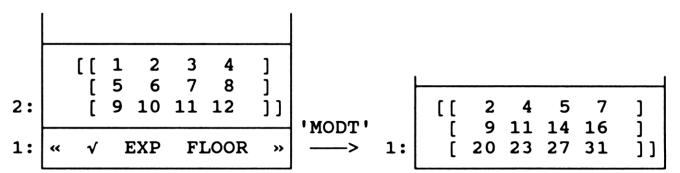


On place le tableau à modifier ('Tab1') au niveau 2. 'Prog' est ici un programme (en notation **RPN**) qui est destiné à s'appliquer à chaque élément de 'Tab1', et à fournir ainsi les éléments du tableau modifié 'Tab2' (de même format que 'Tab1'). 'Prog' doit s'appliquer au nombre placé au niveau 1 de la pile, et doit le remplacer par un nouveau (en principe) nombre.

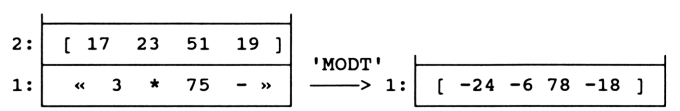
Texte du programme 'MODT'	Commentaires

`« → p`
`« OBJ→ DUP → d`
`« OBJ→`
`IF 2 == THEN`
`*`
`END`
`→ n`
`« 1 n START`
`p EVAL`
`n ROLLD`
`NEXT`
`d →ARRY`
`»`
`»`
`»`
`»`
	p = la procédure à appliquer.
	d = liste donnant le format.
	
	n = nombre d'éléments du tableau
	De 1 à n:
	On évalue p sur le terme placé au niveau 1, et on remonte le résultat au-dessus des autres.
	
	On forme le tableau obtenu.

Exemples:



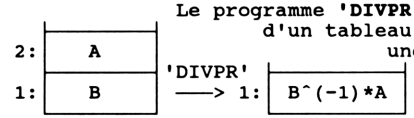
Sur cet exemple, on a remplacé chaque terme x de la matrice initiale par la partie entière de EXP(√(x)).



Sur cet exemple, chaque terme x du vecteur a été remplacé par 3x-75.

Exercice 27 (***): Division d'un tableau par une matrice en précision accentuée.

Le programme **'DIVPR'** permet d'effectuer la division d'un tableau A par une matrice carrée B, avec une précision améliorée par rapport à l'utilisation de /, et ceci grâce à l'utilisation de **RSD**.



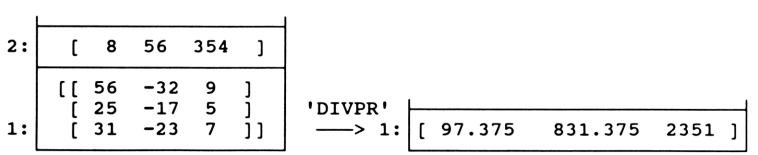
La solution reprend pas à pas les explications qui ont été données dans le chapitre consacré aux tableaux.
Nous allons reprendre les notations alors utilisées:
On suppose donc que l'on veut diviser un tableau A par une matrice carrée B.
Soit D la solution théorique, c'est-à-dire D=B^(-1)*A.
Soit C le résultat obtenu en fait par **A B /** ,et qui est une première approximation de D.
Soit E=D-C le tableau mesurant l'erreur commise.
Soit R=A-B*C, le résidu des tableaux A,B,C, obtenu par **RSD**.
Soit E' le résultat de **R B /** , et qui est une approximation de E.
Alors D'=C+E' est une meilleure approximation de D=B^(-1)*A: C'est le résultat que doit obtenir **'DIVPR'**.

Texte du programme 'DIVPR'	Commentaires
« DUP2	On duplique A et B.
/	On calcule C.
3 DUPN	On duplique A,B,C.
RSD	On calcule le résidu R=A-B*C.
ROT /	On obtient E' par R B / .
+	On trouve D' = C + E'.
SWAP DROP	On élimine A.
»	

Exemple:

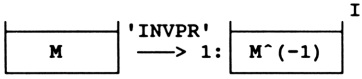
Reprenons les données qui sont utilisées dans la description de l'instruction **RSD** (chapitre consacré aux tableaux).

En une seconde:



On obtient ainsi le résultat exact de la division, alors qu'une utilisation simple de l'opérateur / donne:
[97.3750000122 831.375000059 2351.00000014].

Exercice 28 (##): Inversion d'une matrice carrée, en précision accentuée.



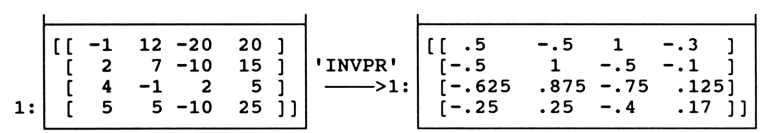
Il s'agit ici de calculer l'inverse d'une matrice carrée M, avec une précision supérieure à celle qu' on obtient par **INV**, et ce en utilisant le programme **'DIVPR'**.

L'idée est simple: on effectue la division en précision accentuée, grâce au programme **'DIVPR'**, de la matrice identité (de même format que A) et de la matrice A.

Texte du programme 'INVPR'	Commentaires
« DUP IDN SWAP DIVPR »	On duplique la matrice A et on crée la matrice identité de même format que A. On place I au dessus de A et on on divise I par A en précision accentuée.

Exemple:

en trois secondes,

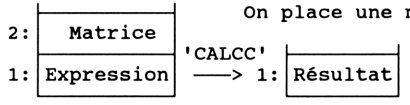


ce qui est le résultat exact de l'inversion.

Une utilisation simple de INV aurait donné:

```
[[ .499999999996 -.499999999992  .999999999995 -.300000000002   ]
 [-.499999999996  .999999999993 -.499999999998 -9.99999999975E-2]
 [-.624999999995  .874999999991 -.749999999996  .125000000003   ]
 [-.249999999998  .249999999996 -.399999999998  .170000000001   ]]
```

Exercice 29 (####): Calculs sur les colonnes d'une matrice.



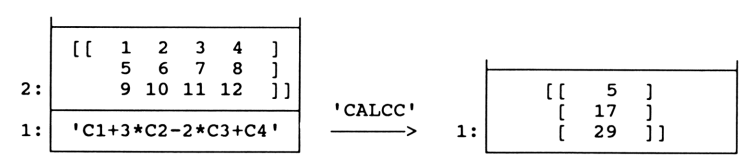
On place une matrice A au niveau 2 de la pile. Le problème est ici d'évaluer une expression algébrique ou un programme où apparaissent éventuellement les noms C1, C2 C3, ... etc, représentant les différentes colonnes de A (numérotées de gauche à droite, et considérées comme des matrices à une colonne).
Le résultat de l'évaluation est placé au niveau 1 de la pile.

L'idée est d'extraire une à une les différentes colonnes de la matrice (à l'aide du programme **'GETC'**), d'en faire des variables globales nommées C1, C2, C3, etc...., d'évaluer l'expression, puis de purger chacune des variables Ci créées.

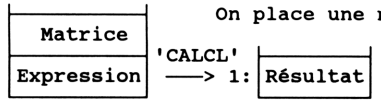
Le problème est dans la formation des noms 'C1', 'C2', 'C3', nécessaires à la création des variables globales.
On confie ce travail à un sous-programme qui prend un entier i comme argument et qui crée le nom 'Ci'. Il suffit de travailler sur des chaînes de caractères et d'utiliser les instructions **→STR** et **OBJ→**.

Texte du programme 'CALCC'	Commentaires
« OVER SIZE 2 GET	Nombre de colonnes.
« "'C" SWAP →STR + OBJ→	Sous-programme formant les noms C1, C2, C3, etc...
»	
→ p C	
« SWAP	On place la matrice au niveau 1.
1 p FOR i	Autant de fois que de colonnes:
DUP i GETC	on extrait la i-ème colonne,
i C EVAL STO	on la place dans la variable globale Ci.
NEXT	
DROP	Effacer la matrice.
EVAL	Evaluer l'expression.
1 p FOR i	Autant de fois que de colonnes:
i C EVAL PURGE	on purge la variable Ci.
NEXT	
»	
»	

Exemple: en sept secondes,



Exercice 30 (####): Calculs sur les lignes d'une matrice.



On place une matrice A au niveau 2 de la pile. Le problème est ici d'évaluer une expression algébrique ou un programme où apparaissent éventuellement les noms L1, L2 L3, L4 ... etc, représentant les différentes lignes de A (numérotées de haut en bas , et considérées comme des vecteurs).
Le résultat de l'évaluation est placé au niveau 1 de la pile.

On procède de la même manière que pour **'CALCC'**:

* Extraire une à une les différentes lignes de la matrice, à l'aide du programme **'GETL'**.

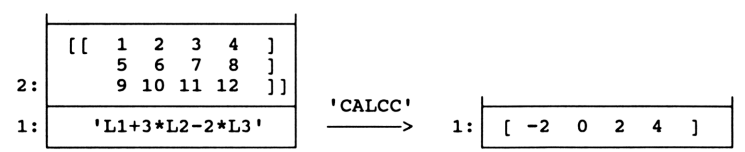
* Placer ces lignes dans des variables globales nommées 'L1', 'L2', 'L3', etc..., la création de ces noms étant assurée par un sous-programme.

* Evaluer l'expression.

* Purger les différentes variables 'Li'.

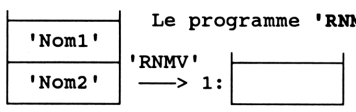
Texte du programme 'CALCL'	Commentaires
« OVER SIZE 1 GET	Nombre de lignes.
« "'L" SWAP →STR +	Sous-programme formant les noms
OBJ→	L1, L2, L3, etc...
»	
→ p L	
« SWAP	On place la matrice au niveau 1.
1 p FOR i	Autant de fois que de lignes:
DUP i GETL	on extrait la i-ème ligne,
i L EVAL STO	on la place dans la variable
NEXT	globale Li.
DROP	Effacer la matrice.
EVAL	Evaluer l'expression.
1 p FOR i	Autant de fois que de lignes:
i L EVAL PURGE	on purge la variable Li.
NEXT	
»	
»	

Exemple: en six secondes,



VARIABLES ET REPERTOIRES : Corrigés des Exercices

Exercice 1 (***) : Renommer une variable.

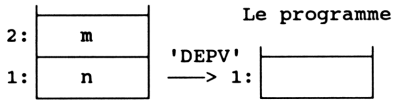


Le programme **'RNMV'** doit renommer la variable dont le nom actuel est 'NOM1', et lui donner le nom 'NOM2'. La position de la variable dans le menu **VAR** doit rester la même. (Cette variable peut très bien désigner un répertoire).

Il faut tout d'abord placer le contenu de 'Nom1' sur la pile, puis purger 'Nom1' (Cette opération s'effectuant au moyen de **PGDIR** ou de **PURGE**, selon que 'Nom1' est ou n'est pas un nom de répertoire)
On place ensuite l'ancien contenu de 'Nom1' dans une variable nommée 'Nom2'.
Reste alors à placer 'Nom2' à l'endroit où se trouvait 'Nom1' dans le menu **VAR** initial (instruction **ORDER**).

Texte du programme 'RNMV'	Commentaires
« → n1 n2	n1: 'Nom1', n2: 'Nom2'.
« VARS	Liste des entrées de VAR.
DUP n1 POS	Place de 'Nom1' dans cette liste
IF DUP THEN	Si 'Nom1' est dans cette liste,
n1 RCL	contenu de 'Nom1' sur la pile.
n1	
IF OVER TYPE 15 ==	Si 'Nom1' est le nom d'un réper-
THEN PGDIR	-toire on utilise PGDIR, sinon
ELSE PURGE	on utilise PURGE.
END	
n2 STO	On met le contenu dans 'Nom2'.
1 SWAP 1 - SUB	Entrées de VAR qui étaient
	avant 'Nom1'.
n2 +	On ajoute 'Nom2' à cette liste
ORDER	et on réordonne le menu VAR.
ELSE DROP n1 n2	Sinon, on remet la pile dans son
END	état initial.
»	
»	

Exercice 2 (***) : Déplacer une entrée du menu **VAR**.



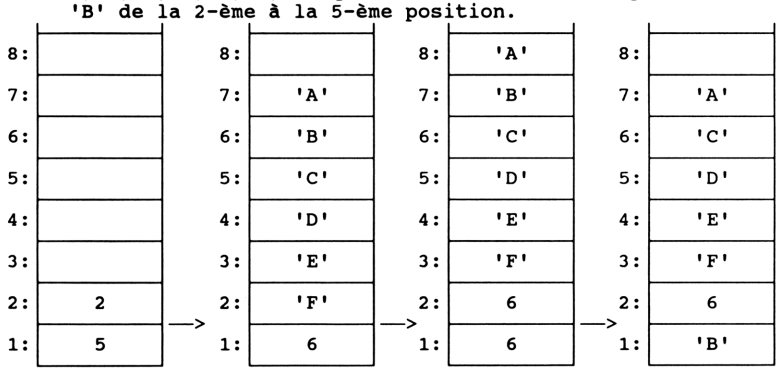
Le programme **'DEPV'** doit déplacer le nom dont la position est la m-ième dans le menu **VAR**, et le placer en n-ème position dans ce menu.
Ce nom peut aussi bien désigner une variable qu'un répertoire.

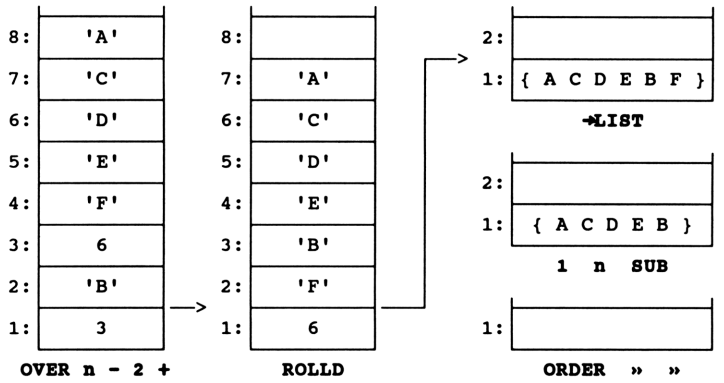
Il suffit de modifier la liste,donnée par **VARS**, de toutes les entrées du menu **VAR**, de telle manière que le nom qui s'y trouvait en m-ème position soit placé en n-ème position.
On utilise alors l'instruction **ORDER** pour réordonner les entrées de **VAR** (il suffit de limiter la liste donnée à **ORDER** aux n premières entrées).

Dans le listing ci-dessous, le déplacement du nom dans la liste donnée par **VARS** se fait après avoir libéré cette liste sur la pile (et on utilise les instructions **ROLL** et **ROLLD**).

Texte du programme 'DEPV'	Commentaires
« → m n	
« VARS	Liste des entrées du menu VAR.
OBJ→	On la libère sur la pile.
DUP m - 2 +	Niveau où chercher le nom.
ROLL	Ce nom descend au niveau 1.
OVER n - 2 +	Niveau où le placer.
ROLLD	Ce nom remonte sur la pile.
→LIST	On reconstitue la liste.
1 n SUB	On en garde la partie utile.
ORDER	On réordonne le menu VAR.
»	
»	

Exemple:
Supposons que les entrées du menu **VAR** soient,dans cet ordre, A, B, C, D, E, F, et que l'on veuille faire passer le nom 'B' de la 2-ème à la 5-ème position.





La dernière instruction **ORDER** réordonne le menu **VAR**, dont la liste des entrées devient { A C D E B F }.

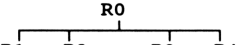
Exercice 3 (*****) : Affichage de l'arborescence.

Le programme **'TREE→'** crée une liste dont le contenu représente l'arborescence de répertoires à partir du répertoire actuel considéré comme racine de cet arbre.

Dans cette liste, chaque nom de répertoire est suivi par la liste des noms de ses sous-répertoires.

Supposons dans un premier temps que le répertoire où on se trouve contienne éventuellement des sous-répertoires mais qu'eux mêmes ne contiennent pas de sous-répertoire.
Il suffit alors de créer une liste de deux objets:
* le 1er objet est le nom du répertoire.
* le deuxième objet est la liste de ses répertoires-fils.

Si par exemple on se trouve dans le répertoire 'R0' et que l'arborescence y est :
alors la liste que doit donner 'TREE→' est:
{ R0 { R1 R2 R3 R4 } }.

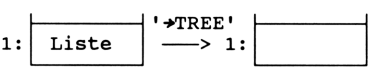


Le problème se complique du fait que, pour reprendre les notations ci-dessus, R1, R2, R3, et R4 peuvent eux-mêmes posséder des sous-répertoires.

La solution passe ici par la récursivité (possibilité qu'a un programme de s'appeler lui-même).
Le schéma précédent reste valable mais devra être appelé à l'intérieur de chacun des sous-répertoires passés en revue.
Il faudra donc rentrer à l'intérieur de chacun de ces sous-répertoires , y effectuer le schéma précédent, en descendant dans l'arborescence autant que nécessaire, sans jamais s'y perdre !

Texte du programme 'TREE→'	Commentaires
« PATH DUP SIZE DUP SUB VARS DUP SIZE → v n « IF n THEN { } 1 n FOR i v i GET IF DUP VTYPE 15 == THEN EVAL TREE→ + UPDIR ELSE DROP END NEXT IF DUP SIZE THEN 1 →LIST END + END » »	Liste { R }, où 'R' est le nom du répertoire courant. v: entrées de VAR. n: nombre d'entrées. S'il y a au moins une entrée: Liste vide L. De i=1 à i=n: i-ème entrée 'S' de VAR. Si 'S' est le nom d'un répertoire: On entre dans S, On appelle TREE-, Ajouter le résultat à L Retour répertoire-père. Sinon effacer le contenu de la variable 'S'. Entrée suivante de VAR... Si la liste L est non vide, on crée la liste { L }. Ajouter { R } à cette liste. Fin du traitement de VAR.

Exercice 4 (*****): Créer une arborescence.



Le programme **'→TREE'** crée une arborescence, en partant du répertoire actuel, et à l'aide d'une liste donnée au niveau 1

Cette liste est constituée des noms des sous-répertoires qu'il faut créer dans le répertoire de départ, chacun de ces noms 'R' étant suivi par la liste des sous-répertoires qu'il faut éventuellement créer dans le sous-répertoire 'R'.

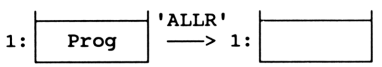
Supposons que l'on soit dans un répertoire 'R0' et que la liste soit { R1 R2 R3 R4 R5 }:
Il faut placer cette liste sur la pile,et créer successivement les sous-répertoires R5, R4, R3, R2, et R1, à l'intérieur de 'R0'. L'ordre de ces créations est important pour que dans le menu **VAR** de 'R0', les noms R1, R2, R3, R4, et R5 apparaissent dancréations est important pour que dans le menu **VAR** de 'R0', les noms R1, R2, R3, R4, et R5 apparaissent dans le même ordre que dans la liste initiale.
On voit qu'il faut ici utiliser une boucle traitant chaque élément de la liste initiale, de la fin vers le début.

Supposons que l'on soit dans un répertoire 'R0' et que la liste soit { R1 R2 R3 { R4 R5 } R6 R7 }:
Après avoir créé, comme ci-dessus, les sous-répertoires 'R7' et 'R6' de 'R0', on lit la liste { R4 R5 }. On va alors chercher l'objet précédent 'R3': On crée le sous-répertoire 'R3' de 'R0', on y descend et on y crée les sous-répertoires 'R5' et 'R4', avant de remonter (**UPDIR**) dans 'R0'.
La situation peut être encore plus compliquée mais ces explications donnent l'essentiel de la version récursive ci-dessous.

Texte du programme '→TREE'	Commentaires

« OBJ→
IF DUP THEN
1 FOR i
 	
IF DUP TYPE 5 ==
THEN
'i' 1 STO-
SWAP
DUP CRDIR
EVAL
→TREE
 	
UPDIR
ELSE
CRDIR
 	
END
-1 STEP
END
»
	Libérer la liste L sur la pile.
	Si ce n'est pas la liste vide:
	Boucle parcourant cette liste de la fin vers le début.
	Si c'est une sous-liste L2,
	
	Avancer compteur de boucle
	Entrée 'S' de répertoire.
	Créer le répertoire 'S'.
	Descendre dans 'S'.
	Traiter la sous-liste L2 dans le répertoire 'S'.
	Monter au répertoire-père.
	
	Sinon c'est un répertoire sans descendants et on se contente de le créer.
	On va traiter l'entrée suivante.
	Fin du traitement de la liste.

Exercice 5 (****): Exécuter un programme dans toute une arborescence.

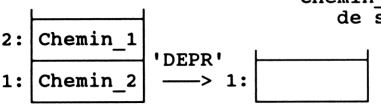


'ALLR' permet d'éxécuter le programme Prog (donné par un nom de variable ou par un contenu) dans le répertoire où **'ALLR'** est appelé et dans toute sa descendance de sous-répertoires.

Voici une solution possible:

Texte du programme 'ALLR'	Commentaires
« → p « p EVAL VARS DUP SIZE → v n « IF n THEN 1 n FOR i v i GET IF DUP VTYPE 15 == THEN EVAL p ALLR UPDIR ELSE DROP END NEXT END » » »	On met le programme dans une une variable locale 'p'. On évalue ce programme. v: liste des entrées de VAR, n: nombre de ces entrées. Si n>0, De i=1 à i=n, i-ème entrée de VAR. Si 'S' est le nom d'un répertoire: On entre dans S, On évalue le programme. Retour répertoire-père. Sinon effacer le contenu de la variable 'S'. Entrée suivante de VAR... Fin du traitement de VAR.

Exercice 6: (*****) Déplacer une arborescence.



Chemin_1 et Chemin_2 sont deux chemins de sous-répertoires tels qu'on peut en obtenir par l'instruction **PATH**. Soit 'Rep1' le dernier nom de la liste Chemin_1, et 'Rep2' le dernier nom de Chemin_2:

'DEPR' transporte dans 'Rep2' le contenu complet de 'Rep1', avec toute l'arborescence de variables et de sous-répertoires qui peut s'y trouver.

Lors du parcours dans l'arborescence décrit par Chemin_2, il peut arriver qu'un nom de répertoire soit inconnu. Dans ce cas, ce répertoire est créé.

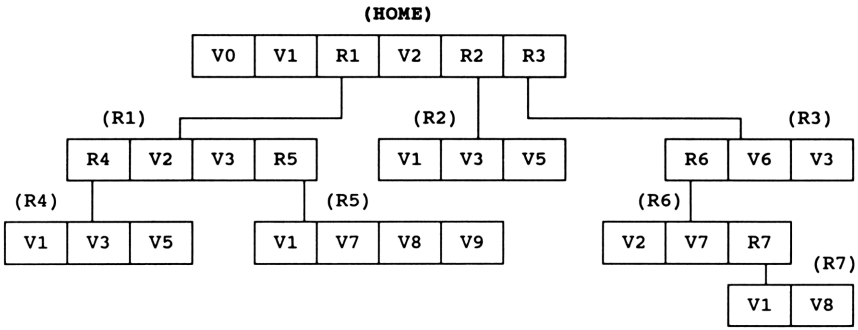
On réalise successivement les opérations suivantes:
- On parcourt le chemin 'Chemin1', jusqu'au répertoire père du répertoire source 'S'.
- On en rappelle le contenu sur la pile (**RCL**).
- On détruit le contenu de 'S' (**PGDIR**).
- On parcourt le chemin 'Chemin2', jusqu'au répertoire père du répertoire but 'B'.
- On place dans 'B' ce qui était le contenu de 'S' (**STO**)

Remarques:
- Le répertoire source 'S' ne peut pas être **HOME** (car le contenu de **HOME** ne peut pas être rappelé par **RCL**).
- Le répertoire but 'B' ne doit pas exister (même vide), avant qu'on lance 'DEPR' (car le **STO** final serait illégal).
- La commande **LAST** doit être active (utilisation de **IFERR**).

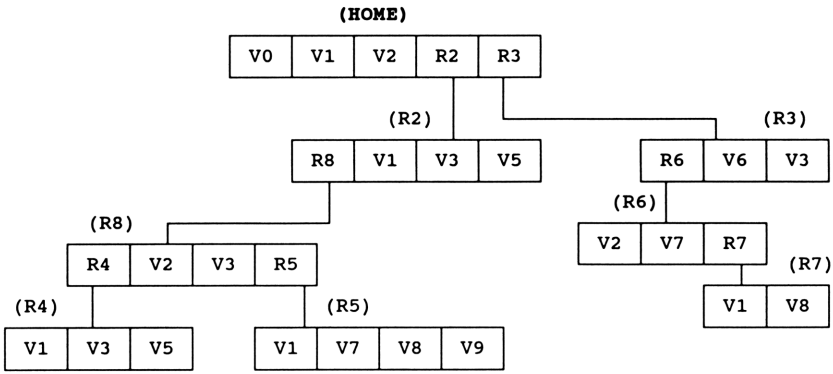
Texte du programme 'DEPR'	Commentaires
`« « 1`	Début ss-programme de parcours.
`DO`	Répéter:
`GETI`	Lire nom du chemin.
`IFERR DUP CRDIR THEN`	En faire un sous-répertoire
`DROP`	s'il est inconnu.
`END`	
`EVAL`	Descendre dans sous-répert.
`UNTIL -64 FS? END`	Jusqu'à chemin lu en entier.
`DROP2`	Effacer liste et index.
`»`	
`→ c1 c2 p`	'p' = ss-programme de parcours.
`« c1 EVAL`	On parcourt Chemin1, jusqu'au
`UPDIR`	répertoire père de 'S'.
`c1 DUP SIZE GET`	On rappelle le contenu de 'S'
`DUP RCL SWAP`	sur la pile, puis on purge
`PGDIR`	ce répertoire source 'S'.
`c2`	On crée le chemin allant au
`1 OVER SIZE 1 - SUB`	répertoire-père de 'B'.
`p EVAL`	On fait le déplacement.
`c2 DUP SIZE GET STO`	On place l'ancien contenu de
	'S' dans le but 'B'.
`END`	C'est fini.
`»`	
`»`	

Exemple:

Supposons que l'arborescence de la HP48SX soit:



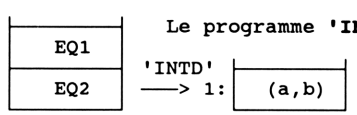
Si on éxécute la séquence **{ HOME R1 } { HOME R2 R8 } DEPR**, alors on obtient l'arborescence:



On remarque que le nom 'R1' est supprimé dans **HOME**, et que l'ancien contenu de 'R1' est passé dans le répertoire 'R8' créé pour l'occasion dans le répertoire 'R2'.

EXPRESSIONS : Corrigés des Exercices

Exercice 1: (***) Point d'intersection de deux droites.

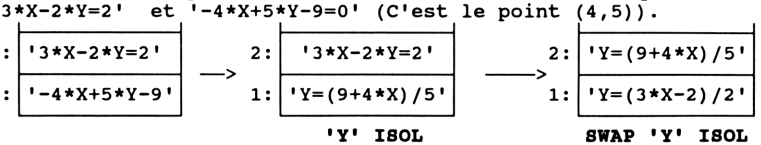


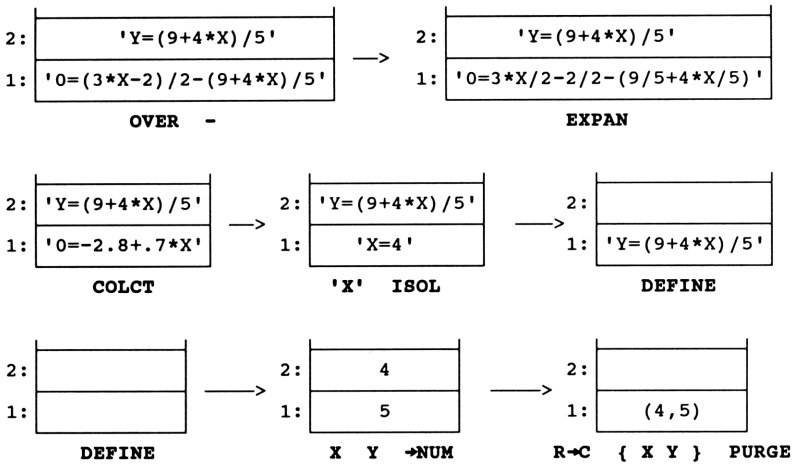
Le programme **'INTD'** donne le point d'intersection A(a,b) des deux droites D1 et D2 dont les équations 'EQ1' et 'EQ2' sont placées aux niveaux 1 et 2. EQ1 et EQ2 sont des expressions du type 'αX+βY+σ', ou 'αX+βY=σ', ou 'βY+αX=σ' ou 'βY+αX+σ' etc... (bref toute forme usuelle d'une équation de droite).
On se limitera au cas où Y est réellement présent dans les équations, et où les deux droites ne sont pas parallèles.

La solution décrite ici consiste à isoler l'expression de Y en fonction de X, dans chacune des deux équations, puis à faire la différence des deux expressions obtenues.
On obtient ainsi une expression en X seul. On y isole X, ce qui donne la valeur de X annulant cette expression (et donc l'abscisse du point d'intersection).
Il suffit alors de créer la variable 'X' avec cette valeur (à l'aide de l'instruction **DEFINE**), de stocker dans 'Y' (toujours à l'aide de **DEFINE**) l'une des 2 expressions Y=f(X) obtenues avant.

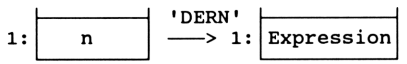
Texte du programme 'INTD'	Commentaires
«	
'Y' ISOL	Isole Y=g(X) dans l'équation EQ2
SWAP 'Y' ISOL	Isole Y=f(X) dans l'équation EQ1
OVER -	Equation 0=f(X)-g(X).
EXPAN COLCT	Réduit l'expression f(X)-g(X).
'X' ISOL	Extrait X de cette expression.
DEFINE	Place la valeur trouvée dans 'X'
DEFINE	Stocke dans Y l'expression g(X).
X Y →NUM R→C	Crée le complexe (x,y).
{ X Y } PURGE	Purge les variables X et Y.
»	

Voyons comment les choses se passent sur un exemple.
On veut trouver le point d'intersection des droites d'équations:
'3*X-2*Y=2' et '-4*X+5*Y-9=0' (C'est le point (4,5)).





Exercice 2: (**) Dérivée n-ème de **'X^(n-1)*LN(X)'**



'DERN' calcule la dérivée n-ème de la fonction 'X^(n-1)*LN(X)' par rapport à la variable for--melle 'X'.Le résultat est une expression (simple) placée au niveau 1.

On doit tout d'abord purger la variable 'X' de manière à être certain d'obtenir des dérivées symboliques.

On procède ensuite à n dérivations successives; après chacune d'elle, il faut réduire l'expression par **COLCT** sans quoi les dérivées successives iraient en se compliquant inutilement.

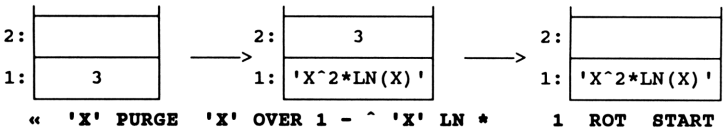
Texte du programme 'DERN'	Commentaires
« 'X' PURGE 'X' OVER 1 - ^ 'X' LN * 1 ROT START 'X' ∂ COLCT NEXT »	'X' variable formelle. Crée l'expression X^(n-1)*LN(X) n fois de suite: On dérive par rapport à X. On réduit l'expression.

On montre que la dérivée n-ème de **X^(n-1)*LN(X)** est **(n-1)!/X.**
C'est effectivement l'expression obtenue par le programme **DERN.**

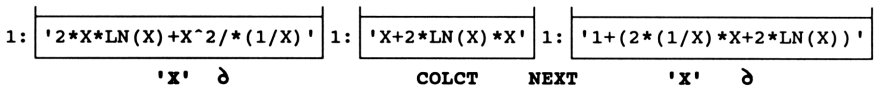
Les temps de calcul sont les suivants:

n	1	2	3	4	5	7
Temps	1s	2s	5s	9s	15s	26s

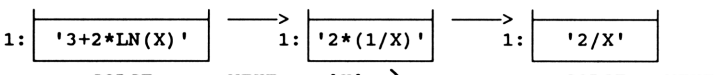
Examinons les différentes étapes du calcul, dans le cas n=3.



« 'X' PURGE 'X' OVER 1 - ^ 'X' LN * 1 ROT START



'X' ∂ COLCT NEXT 'X' ∂

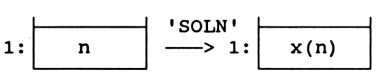


COLCT NEXT 'X' ∂ COLCT NEXT »

Si on ne place pas l'instruction **COLCT** après chaque dérivation, voici ce qu'on obtient:

```
Après la première dérivation: '2*X*LN(X)+X^2*(1/X)'.
Après la 2ème: '2*LN(X)+2*X*(1/X)+(2*X*(1/X)+X^2*-(1/X^2))'.
Après la 3ème: '2*(1/X)+(2*(1/X)+2*X*-(1/X^2))+(2*(1/X)+
               2*X*-(1/X^2)+(2*X*-(1/X^2)+X^2*(2*X/X^2^2)))'.
```

Exercice 3: (***) Calculer une solution particulière de **TAN(X)=X.**



'SOLN' calcule la solution x(n) de l'équation TAN(X)=X, telle que $n\pi-\pi/2 \leq X \leq n\pi+\pi/2$. (n entier positif ou négatif).

On se placera en mode radians.

On peut remarquer que le système $\begin{cases} n\pi-\pi/2 < x(n) < n\pi+\pi/2 \\ TAN(x) = x \end{cases}$

équivaut à l'unique équation:
'ATAN(X)=X-nπ, qui admet une solution unique sur R.

On peut donc résoudre le problème en utilisant l'une ou l'autre des deux équations, ce qui donne deux versions de la solution.

Texte du programme 'SOLN' (Première version)	Commentaires
« → n « « X DUP TAN - » 'X' n π →NUM * ROOT » »	Ss-prog. calculant X-TAN(X). Variable de recherche. nπ: point de départ recherche. On lance la recherche.

Texte du programme 'SOLN' (Deuxième version)	Commentaires
« → n « « X DUP ATAN - π →NUM n * - » 'X' n π →NUM * ROOT » »	Sous-programme calculant: X-ATAN(X)-n*π. Variable de recherche. nπ: valeur initiale de recherche On lance la recherche.

D'une manière générale, la 2ème version est bien plus rapide que la première. celà tient sans doute à ce que la dérivée de ATAN(X) au voisinage de x(n) tend vers zéro quand n augmente (alors qu'au contraire la dérivée de TAN(X) est, au voisinage de x(n), à peu près proportionnelle au carré de x(n)).

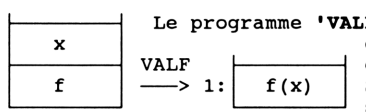
La deuxième version est également souvent plus précise:

Ainsi pour n=100, on trouve:

avec la 1ère version: x(n)≈315.609633869 (en 2 secondes)
et TAN(315.609633869) ≈ 8.26 ....

avec la 2ème version: x(n)≈315.726894402 (en 1 à 2s)
et TAN(315.726894402) ≈ 315.726890101 ....

Exercice 4: (*) Evaluer un programme ou une expression.



Le programme **'VALF'** évalue au point x l'expression ou le programme f, de manière à obtenir un résultat noté ici f(x). Si f est un programme, il est tout simplement évalué.

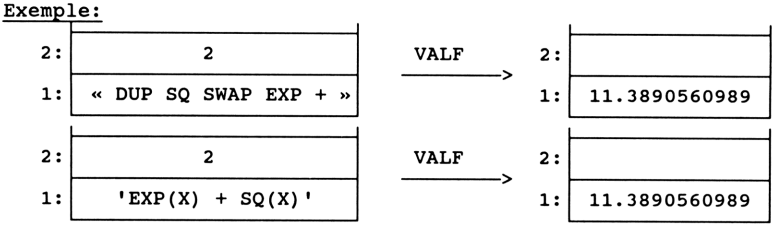
Si f est une expression, la valeur x est placée dans la variable 'X' puis l'expression f est évaluée (f est donc en principe une expression par rapport à la variable X)

Exemple: avec x=5 au niveau 2, le résultat doit être EXP(5) aussi bien si f est le programme « EXP » que l'expression 'EXP(X)'.

Çà ne présente aucune difficulté...

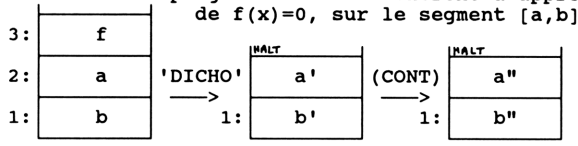
Texte du programme 'VALF'	Commentaires
« IF DUP TYPE 9 == THEN SWAP 'X' STO END EVAL »	Si f est une expression, on met la valeur dans 'X'. Dans les deux cas, on évalue f.

Exemple:



Exercice 5: (***) Résolution de f(X)=0 par **dichotomie.**

Le programme **'DICHO'** cherche à approcher la solution x de f(x)=0, sur le segment [a,b], par la méthode de la dichotomie.



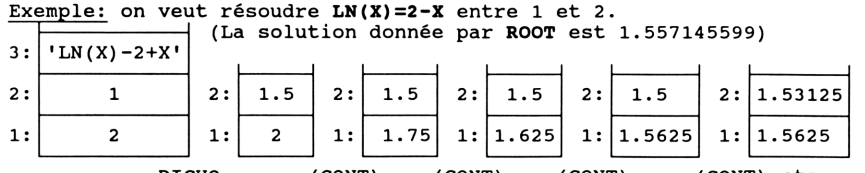
Le programme est suspendu à chaque calcul d'un nouveau segment [a',b'], [a",b"], etc...

f peut aussi bien être une expression qu'un programme (cf ex5).

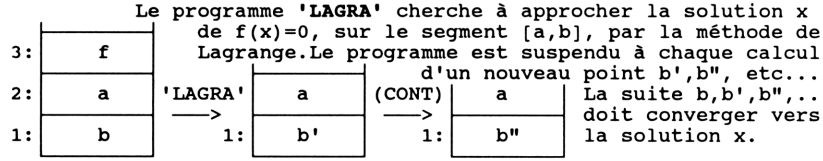
Rappelons que la méthode consiste à remplacer chaque segment (le premier étant [a,b]), par la moitié de ce segment qui contient la solution x (on suppose que cette racine existe et que la fonction f est continue).
Plus précisément: Si on part d'un segment [a,b], on considère le milieu c=(a+b)/2 de ce segment et on évalue f(c).
Si f(c)*f(a)<0, le nouveau segment est [a,c]. Sinon c'est [c,b].
Le défaut de cette méthode est sa relative lenteur. Sa qualité est de fournir un encadrement (deux fois plus précis à chaque étape) de la solution x.

Texte du programme 'DICHO'	Commentaires
« ROT → f « DO DUP2 + 2 / DUP f VALF ROT f VALF IF * 0 < THEN DUP 2 * ROT - END HALT UNTIL 0 END » »	Placer f dans variable locale. Répéter: a,b,c aux niveaux 3,2,1. c au dessus de f(c). f(b) au niveau 1. Si f(b)*f(c)<0 alors remplacer a par b. a' et b' aux niveaux 2 et 1. Programme suspendu. Ad Aeternam....

Exemple: on veut résoudre **LN(X)=2-X** entre 1 et 2.
(La solution donnée par **ROOT** est 1.557145599)



Exercice 6: (***) Résolution de f(X)=0 par la méthode de **Lagrange.**



Le programme **'LAGRA'** cherche à approcher la solution x de f(x)=0, sur le segment [a,b], par la méthode de Lagrange.Le programme est suspendu à chaque calcul d'un nouveau point b',b", etc... La suite b,b',b",.. doit converger vers la solution x.

Dans ce schéma fonctionnel, a est la valeur fixe de la méthode, et b', b" etc...sont les valeurs successives calculées.

Le fait que "a" apparaisse à chaque **HALT** permet de le modifier avant de faire **CONT.**

f peut aussi bien être une expression qu'un programme (cf ex5).

Rappelons le principe de la méthode:

On part du segment [a,b].
On trace la corde reliant les points d'abscisse a et b de la courbe représentative de f.
Cette corde rencontre l'axe Ox, au point d'abscisse

$$c = a - \frac{(b-a)f(a)}{f(b)-f(a)} = \frac{af(b)-bf(a)}{f(b)-f(a)} = b - \frac{(b-a)f(b)}{f(b)-f(a)}$$

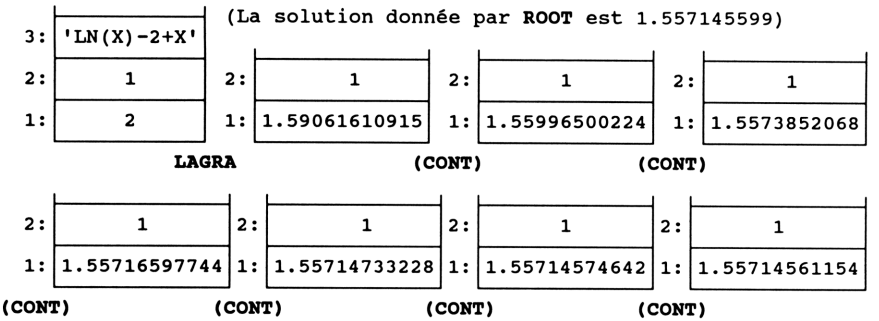
On remplace alors b par c, et on recommence les étapes précédentes sur le segment [a,c].

Remarque:

Le choix du point a (qui reste fixe) est important. Un choix incorrect fait diverger la méthode. En principe on a avantage à choisir le point a tel que f(a)*f"(a) > 0.
Si la méthode semble diverger, revenez à la pile de départ et inversez l'ordre respectif de a et b sur cette pile.

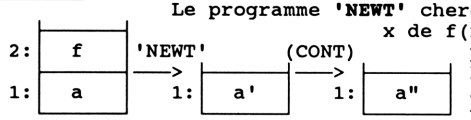
Texte du programme 'LAGRA'	Commentaires
« ROT → f « DO OVER DUP2 - SWAP f VALF 3 PICK f VALF DUP ROT - / * - HALT UNTIL 0 END » »	f dans une variable locale. Répéter: a,b,a,b-a aux niveaux 4,3,2,1. a,b,b-a,f(a) ..... a,b,b-a,f(a),f(b) ..... a,b,b-a,f(b),f(b),f(a) ..... a, c=b-(b-a)*f(b)/(f(b)-f(a)). Programme suspendu. Ad Aeternam.

Exemple: on veut résoudre **LN(X)=2-X** entre 1 et 2.



Exercice 7: (***) Résolution de f(X)=0 par la méthode de Newton.

Le programme **'NEWT'** cherche à approcher la solution x de f(x)=0, en partant du point a, par la méthode de Newton. Le programme est suspendu à chaque calcul d'un nouveau point: a',a".



La suite a,a',a" doit converger vers la solution. Le fait que le programme soit suspendu permet de relancer la recherche à partir d'un nouveau point a.
f doit être une expression algébrique par rapport à X (car f doit être dérivée). f ne peut donc ici pas être un programme.

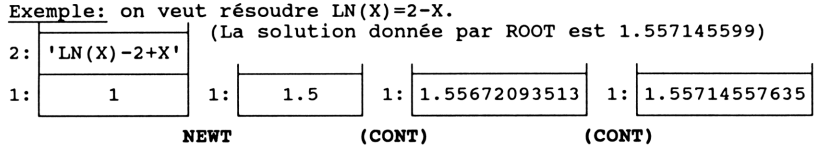
Le principe de la méthode est le suivant: On considère la tangente à la courbe y=f(x), au point d'abscisse a.
L'équation de cette tangente est: y = f(a) + (x-a)f'(a).
Elle rencontre l'axe Ox au point d'abscisse: a'=a-f(a)/f'(a).

On recommence les mêmes opérations à partir de a', et ainsi de suite. Les points obtenus a,a',a",etc... forment une suite qui doit converger vers la solution x.

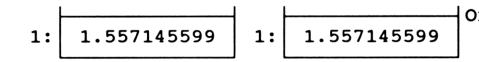
Il se peut qu'il y ait divergence (notamment si le point initial a est trop éloigné de la solution x).
Dans ce cas, on peut modifier la valeur placée au niveau 1 lors de la suspension du programme **'NEWT'** par **HALT**.

Texte du programme 'NEWT'	Commentaires
« 'X' PURGE SWAP DUP 'X' ∂ → f df « DO DUP 'X' STO f EVAL df EVAL / - HALT UNTIL 0 END » »	Purger X pour dérivée symbolique Calcule f'(X). f et f' dans variables locales. Répéter: Placer a dans la variable 'X' Calcule f(a) et f'(a). a' = a-f(a)/f'(a). Programme suspendu. Ad aeternam...

Exemple: on veut résoudre LN(X)=2-X.

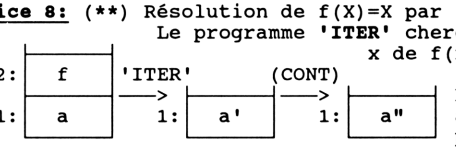


(La solution donnée par ROOT est 1.557145599)



On constate que la solution exacte est obtenue à la quatrième itération.
En général la méthode de Newton est effectivement la plus rapide.

Exercice 8: (**) Résolution de f(X)=X par itérations.



Le programme **'ITER'** cherche à approcher la solution x de f(x)=x, en partant du point a, par itérations.
Le programme est suspendu à chaque calcul d'un nouveau point: a'=f(a), a"=f(a'),...

La suite a,a',a" peut converger vers la solution. Le fait que le programme soit suspendu permet de relancer la recherche à partir d'un nouveau point a.
La convergence est probable si a est "suffisamment" proche de la solution x et si la dérivée de f, en x, est inférieure à 1 en valeur absolue (convergence d'autant plus rapide que f'(x) est proche de zéro).

La solution est d'une simplicité....

Texte du programme 'ITER'	Commentaires
« SWAP → f « DO f VALF HALT UNTIL 0 END » »	f dans une variable locale. Répéter: Calcule a'=f(a). Programme suspendu. Ad aeternam...

Exemple: on veut résoudre l'équation **LN(X)=2-X.**
(La solution donnée par **ROOT** est 1.557145599)

Cette équation peut être écrite: **2-LN(X)=X.**

Si on part de la pile: '2-LN(X)' au niveau 2, et 1 au niveau 1, on obtient successivement:

2	1.58617428463	1.55917135423
1.30685281944	1.5386749934	1.55584550305
1.73237818116	1.56907834782	1.55798087027
1.4505048643	1.54951159262	1.55660933102
1.62808832189	1.56206022026	1.55749004999
1.51259348212	1.55399439602	1.55692441679

On constate que la convergence est extrêmement lente.

Cette situation est due au fait que la dérivée de '2-LN(X)' est '-1/X' qui vaut ≈ -0.64 au voisinage de la solution:

* Dérivée comprise entre -1 et 1 : convergence.
* Dérivée négative: valeurs successives encadrant la solution.
* Dérivée pas assez proche de zéro: convergence lente.

On peut modifier l'expression initiale de manière à accélérer la convergence. L'équation 'LN(X)=2-X' est en effet équivalente à toute équation s'écrivant: **'α*(LN(X)-2+X)+X=X'**, (α réel).

On choisit α pour minimiser la dérivée de g(X)=α*(LN(X)-2+X)+X au voisinage de la solution x.
Dire que g'(x)=0 c'est dire que α*(1/x+1)+1=0 cad: α=-x/(x+1).

En choisissant α≈-x/(x+1), où x est la solution encore inconnue, la convergence doit donc être rapide.
Puisque x≈1.56, prenons α≈-1.56/2.56≈-.61.

On est donc amené à utiliser **'ITERA'** avec, sur la pile: **'.61*(2-X-LN(X))+X'** au niveau 2 et 1 au niveau 1.

on trouve alors successivement:

1.61	1.55714516865	1.557145599
1.55739715081	1.5571459975	1.557145599

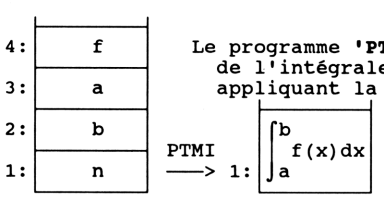
Le résultat exact est donc obtenu ici à la 5ème itération.

Remarque: utilisation de **'ISOL'**.

L'instruction **'ISOL'** de la HP48SX permet de transformer une expression 'F(X)' en une expression 'X=G(X)' équivalente.
(Il faut que la première occurence de X puisse être isolée dans l'expression F, ce qui n'est pas possible pour toutes les fonctions de la HP48SX).

On peut donc utiliser l'instruction **'ISOL'** conjointement au programme **'ITERA'** pour résoudre l'équation 'F(X)=0'.

Exercice 9: (***) Calcul d'une intégrale (méthode du **point-milieu**)



Le programme **'PTMI'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode du point milieu.
Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

$$\int_{\alpha}^{ß} f(x)dx = (ß-\alpha)f((\alpha+ß)/2).$$

Appelons h=(b-a)/n la longueur de chacun des sous-intervalles.
La valeur obtenue s'écrit:
I = h*(f(a+h/2) + f(a+3*h/2) + f(a+5*h/2) +...+ f(b-h/2)).

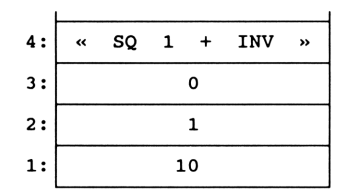
Le texte du programme s'en déduit immédiatement:

Texte du programme 'PTMI'	Commentaires

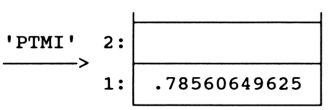
`« → f a b n`
`  « b a - n / → h`
`  « 0`
`  a h 2 / + b`
`  FOR x`
`  x f VALF +`
`  h STEP`
`  h *`
`  »`
`  »`
`»`
	Crée les variables locales.
	C'est le "pas" de la méthode.
	Initialise l'intégrale I.
	De x = a+h/2 à x=b , avec un incrément égal à h.
	Ajoute f(x) à I.
	Valeur suivante de x...
	On multiplie I par h.

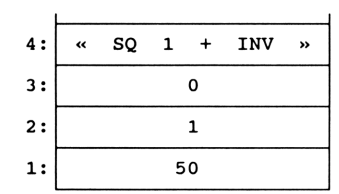
Remarque: le calcul de l'intégrale se fait de a vers b. Il se peut très bien que a>b.
On obtient ainsi deux résultats opposés si on "swappe" les objets a et b des niveaux 2 et 3.

Exemple: Calculons l'intégrale de $1/(1+X^2)$ entre 0 et 1.
(le résultat exact est $\pi/4 \approx .785398163398$).

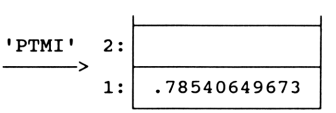


En une seconde:
(erreur ≈ 2E-4)

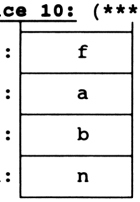




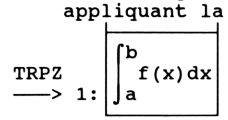
En 3 secondes:
(erreur ≈ 8E-6)



Exercice 10: (***) Calcul d'une intégrale (méthode des **trapèzes**)



Le programme **'TRPZ'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode des trapèzes.



Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

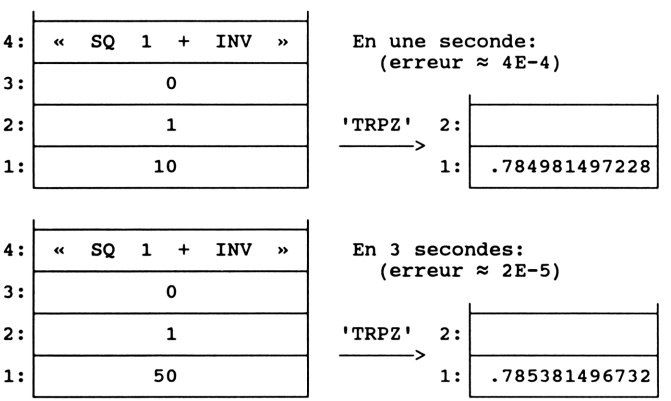
$$\int_{\alpha}^{\beta} f(x)dx = (\beta-\alpha)(f(\alpha)+f(\beta))/2.$$

Appelons h=(b-a)/n la longueur de chacun des sous-intervalles.
La valeur obtenue s'écrit:
I = h*(f(a)/2 + f(a+h) + f(a+2*h) + ... + f(b-h) + f(b)/2).

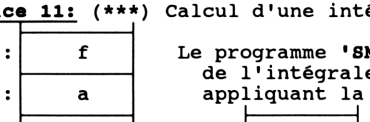
On commence donc par initialiser I avec I=(f(a)+f(b))/2, puis on ajoute f(a+h), f(a+2*h), ... dans une boucle.
Il reste à la fin à multiplier par h.
Comme dans **'PTMI'**, le calcul de l'intégrale se fait de a vers b, même si a>b.

Texte du programme 'TRPZ'	Commentaires
« → f a b n « b a - n / → h « a f VALF b f VALF + 2 / a h + b h 2 / - FOR x x f VALF + h STEP h * » » »	Crée les variables locales. C'est le "pas" de la méthode. On initialise l'intégrale I avec la valeur (f(a)+f(b))/2. De x = a+h à x=b-h/2 , avec un incrément égal à h. Ajoute f(x) à I. Valeur suivante de x... On multiplie I par h.

Exemple: Calculons l'intégrale de $1/(1+X^2)$ entre 0 et 1.
(le résultat exact est $\pi/4 \approx .785398163398$).



Exercice 11: (***) Calcul d'une intégrale (méthode de **Simpson**)



Le programme **'SMPS'** calcule une valeur approchée de l'intégrale de f sur le segment [a,b], en appliquant la méthode de Simpson.

Le segment [a,b] est divisé en n sous-intervalles égaux [α,ß] et sur chacun d'eux on pose:

$$\int_{\alpha}^{\beta} f(x)dx = (\beta-\alpha)\left[\frac{f(\alpha)+4f((\alpha+\beta)/2)+f(\beta)}{6}\right]$$

EXPRESSIONS: Corrigés des Exercices.

Première méthode:

La valeur de I obtenue en sommant les approximations effectuées sur chacun des sous-intervalles est:

$$I = \frac{h}{6} * \left[f(a) + 4*f(a+h/2) + 2*f(a+h) + 4*f(a+3*h/2) + 2*f(a+2*h) + \ldots + 2*f(b-h) + 4*f(b-h/2) + f(b) \right]$$

La solution choisie ici consiste à:

* Initialiser I = (f(a)-f(b)) / 2. (pourquoi - ? ...)
* Ajouter à I : 2*f(a+h/2) et f(a+h),
  2*f(a+3*h/2) et f(a+2*h),.....,
  2*f(b-h/2) et f(b). (voilà pourquoi -)
* A multiplier le résultat obtenu par h/3.

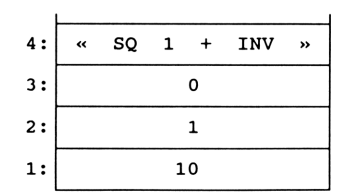
Comme dans **'PTMI'** et **'TRPZ'** le calcul se fait de a vers b dans tous les cas (que a<b ou que a>b).

Texte du programme 'SMPS' (Première version)	Commentaires
« → f a b n « b a - n / → h « a f VALF b f VALF - 2 / a h 2 / + b FOR x x f VALF 2 * + x h 2 / + f VALF + h STEP h * 3 / » » »	Crée les variables locales. C'est le "pas" de la méthode. On initialise l'intégrale I avec la valeur (f(a)-f(b))/2. De x = a+h/2 à x=b , avec un incrément égal à h. Ajoute 2*f(x) à I. Calcule x'=x+ h /2. Ajoute f(x') à I. Valeur suivante de x... On multiplie I par h /3.

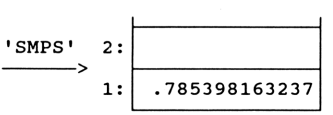
Remarque:

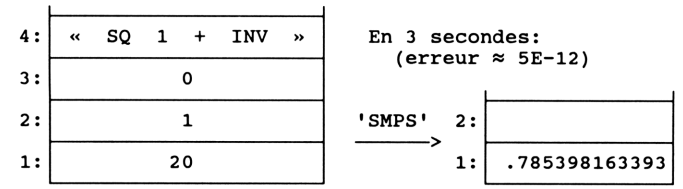
Comme dans les programmes **'PTMI'** et **'SPMS'**, on utilise ici une boucle dont les valeurs extrêmes peuvent surprendre.
Théoriquement, x doit varier de a+h/2 à b-h/2, avec un pas de h.
Dans le listing les extrémités de boucles sont:
de x=a+h/2 à x=b, avec un incrément de h.
Ce choix ne modifie pas le nombre des valeurs prises par x, ni ces valeurs; il sert uniquement à éviter que des erreurs d'arrondi (dues à l'accumulation du pas de la boucle) ne conduisent à perdre la dernière valeur x=b-h/2.

Exemple: Calculons l'intégrale de $1/(1+X^2)$ entre 0 et 1.
(le résultat exact est $\pi/4 \approx .785398163398$).



En 2 secondes:
(erreur ≈ 2E-10)





On voit que la méthode de Simpson donne des résultats excellents. A titre de comparaison, le calcul de cette intégrale en utilisant la touche d'intégration de la HP48SX, avec la même précision que ci-dessus, dure 8 secondes!

Deuxième méthode:

Sur chacun des sous-segments [α,ß] du segment [a,b], la méthode de Simpson consiste à utiliser l'approximation:

$$Is = \int_{\alpha}^{ß} f(x)dx \approx (ß-\alpha) * \frac{f(\alpha)+4f((\alpha+ß)/2)+f(ß)}{6} .$$

La méthode du point milieu utilise l'approximation:

$$Ip = \int_{\alpha}^{ß} f(x)dx = (ß-\alpha)\ f((\alpha+ß)/2).$$

Enfin la méthode des trapèzes utilise:

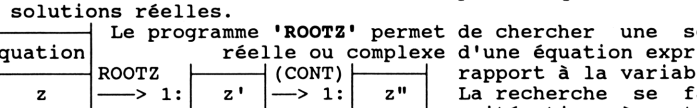
$$It = \int_{\alpha}^{ß} f(x)dx = (ß-\alpha)(f(\alpha)+f(ß))/2.$$

On se rend compte que Is = (It + 2*Ip)/3.
On en déduit une solution très simple pour le programme **'SPMS'**.

Texte du programme 'SMPS' (Deuxième version)	Commentaires
« 4 DUPN TRPZ 5 ROLLD PTMI 2 * + 3 / »	Duplique f,a,b,n. Applique méthode des trapèzes. Résultat au-dessus de f,a,b,n. Méthode du point milieu. Combine les deux résultats pour obtenir la méthode de Simpson.

Exercice 12: (*****) Chercher une solution complexe d'une équation.

L'instruction **ROOT** de la HP48SX ne permet que d'obtenir des solutions réelles.



Le programme **'ROOTZ'** permet de chercher une solution réelle ou complexe d'une équation exprimée par rapport à la variable 'Z'. La recherche se fait par itérations à partir d'un point initial z placé au niveau 1.

Supposons par exemple que l'on veuille trouver un zéro complexe de l'expression F(Z).
Le choix de l'algorithme de Newton nous conduit à considérer que F est en fait une fonction des deux variables réelles X, et Y, avec Z=X+iY.
Il faudra donc commencer par placer 'X+i*Y' dans la variable 'Z', avant de dériver l'expression F(Z) (c'est à dire F(X+iY)) par rapport à X et à Y.

Appelons, en un point z=x+iy donné:
f la valeur de F(Z)=F(X+iY),
dfx la valeur de la dérivée de F par rapport à X, au point z.
dfy la valeur de la dérivée de F par rapport à Y, au point z.

Des calculs qu'il est inutile de développer ici conduisent à créer une suite de nombres complexes z,z',z", etc... (z étant la valeur initiale que vous avez placée au niveau 1) de la façon suivante:
La formule qui donne z' en fonction de z est (mais la formule est la même pour passer de z' à z", etc...) est:

$$z' = z + \frac{\text{IM(f * CONJ(dfy))} + i*\text{IM(CONJ(f) * dfx)}}{\text{IM(CONJ(dfx) * dfy)}}$$

où **IM**, **CONJ** sont les instructions de la HP48SX (partie imaginaire , conjugué d'un nombre complexe), les valeurs de f,dfx,dfy étant calculées au point z.

Le texte du programme **'ROOTZ'** utilise une astuce pour calculer la quantité à ajouter à z pour passer de z à z'.
Posons par exemple f=(a,b), dfx=(t,u), dfy=(v,w).
Alors IM(f*CONJ(dfy)) = -aw+bv; IM(CONJ(f)*dfx) = au-bt
et IM(CONJ(dfx)*dfy) = tw-uv.
Si on place f,dfx,dfy dans un vecteur complexe, et que l'on en sépare la partie réelle et la partie imaginaire, on obtient les deux vecteurs réels [a t v] et [b u w].
Si on effectue leur produit vectoriel, on obtient:

$$\begin{bmatrix} a \\ t \\ v \end{bmatrix} \wedge \begin{bmatrix} b \\ u \\ w \end{bmatrix} = \begin{bmatrix} tw-uv \\ bv-aw \\ au-bt \end{bmatrix}$$ vecteur que nous notons $$\begin{bmatrix} \alpha \\ \beta \\ \sigma \end{bmatrix}$$

On se rend compte alors que le passage de z à z' s'écrit, avec ces notations: $z' = z + (\beta+i\sigma)/\alpha$.

Il ne reste plus alors qu'à écrire le programme (les commentaires utilisent les notations ci-dessus).

Remarque:

Ce programme est suspendu à chaque calcul d'un nouveau point.
Ceci permet en particulier de modifier ce point si on veut démarrer la recherche d'un autre endroit (notamment si la convergence laisse à désirer).
D'autre part, la meilleure façon de quitter proprement le programme est de vider la pile avant de faire CONT. **'ROOTZ'** comprend alors que vous avez terminé et purge les 3 variables 'X','Y' et 'Z', crées pour l'occasion.

Texte du programme 'ROOTZ'	Commentaires
« { X Y } PURGE	X et Y variables formelles.
'X+i*Y' 'Z' STO	On place 'X+i*Y' dans 'Z'.
SWAP DUP	z au dessus de F(Z) dupliqué.
'X' ∂	Dérivée de F par rapport à X.
OVER 'Y' ∂	Dérivée de F par rapport à Y.
→ f dfx dfy	Crée trois variables locales.
« DO	Répéter:
DUP	Dupliquer z.
C→R	Le séparer en x et y.
'Y' STO 'X' STO	Actualise contenus de 'X','Y'.
f →NUM	Valeur de f en z.
dfx →NUM	Valeur de dfx en z.
dfy →NUM	Valeur de dfy en z.
3 →ARRY	Tableau [f dfx dfy].
C→R	Séparé en [a t v] et [b u w].
CROSS	Produit vectoriel [α ß σ].
OBJ→ DROP	z,α,ß,σ aux niveaux 4,3,2,1.
R→C	z,α,(ß+i*σ) aux niveaux 3,2,1.
SWAP / +	z' = z + (ß+i*σ)/α.
HALT	Programme suspendu.
UNTIL DEPTH NOT	Jusqu'à ce que la pile soit vide
{ X Y Z } PURGE	Purger 'X','Y' et 'Z'.
»	
»	

<u>Exemple:</u> on veut trouver une solution pour **'SIN(Z)=Z²+1'**.

On suppose qu'on est en mode "radians". On part du point (1,1).

```
                                (8 sec.)                  (2 sec.)
   2: 'SIN(Z)=Z*Z+1'  ROOTZ  ┌──────────────────┐ CONT  ┌──────────────────┐
   1:      (1,1)        ──>  1: (.637460498577,  ──>  1: (.488531625213,
                                 .758681431203)           .769546966652)
(2 sec.)                (2 sec.)                  (2 sec.)

CONT  ┌──────────────────┐ CONT  ┌──────────────────┐ CONT  ┌──────────────────┐
──>  1: (.488337691548,  ──>  1: (.488374313022,  ──>  1: (.488374325324,
      .785655378062)            .785481427782)            .785481409895)
```

Les résultats restent alors stables (à la dernière décimale près).

On a donc trouvé la solution z=(.488374325324,.785481409895).

Effectivement, on vérifie, pour cette valeur z:

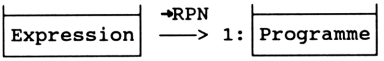
SIN(z) = (.621528436345,.767217907225)

z*z+1 = (.621528436345,.767217907224)

et donc SIN(z)-z*z-1 = (0, .000000000001).

EXPRESSIONS: Corrigés des Exercices.

Exercice 13: (*****) Transformer une expression en programme RPN.



'→RPN' transforme l'expression algébrique (ou équation) placée au niveau 1 en un programme en notation polonaise inverse, équivalent à cette expression.

Pour prendre un exemple:

'EXP(5^SQ(Z))+X/(Y-1)^A'
devient: **« 5 Z SQ ^ EXP X Y 1 - A ^ / + »**

Commençons par dégager les idées directrices qui conduisent à la solution, les détails techniques seront réglés petit à petit.

Le mieux pour "collecter" les différents objets qui figurent dans l'expression initiale est d'utiliser une liste, quitte ensuite à transformer cette liste en programme avec les instructions **→STR** et **OBJ→**.

Par exemple l'expression 'X*SIN(Y+Z)' est transformée en la liste { X Y Z + SIN * } avant de devenir « X Y Z + SIN * »

Dans toute expression algébrique contenant au moins un opérateur, c'est-à-dire ne se réduisant pas à un seul objet (nombre ou nom), il y a un et un seul opérateur au niveau logique 0 (à la racine de l'arbre).

Cet opérateur "racine" figurera en dernière position dans le programme **RPN** que nous devons former.

La première étape consiste à extraire cet opérateur racine.

L'instruction **OBJ→**, appliquée à l'expression, nous le permet. Mieux encore, elle nous donne le nombre d'arguments de cet opérateur, ces arguments étant eux-même placés sur la pile.

On sait que l'opérateur racine viendra se placer en fin du programme **RPN**. Il suffit maintenant de décomposer le ou les arguments de cet opérateur racine.

On s'oriente de manière naturelle vers un algorithme récursif, car ces arguments sont des expressions.

Comme dans tout algorithme récursif, il faut se préoccuper des conditions d'arrêt de la "plongée récursive".
On finit à un moment ou à un autre par avoir à considérer un objet élémentaire de l'expression initiale (un nombre ou un nom). Il suffira alors de placer ce nombre ou ce nom dans une liste à un élément.

La solution ci-dessous utilise un programme principal (dont le rôle est de traiter les opérations non récursives comme la transformation de la liste finale en programme **RPN**) et un sous-programme destiné à s'auto-appeler (et placé dans la variable locale 'sp') . Il est nécessaire de donner au départ un contenu fictif à la variable 'sp' pour pourvoir y placer ensuite le sous-programme, si on veut qu'il puisse lui même s'appeler...

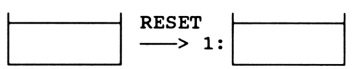
Texte du programme '→RPN'	Commentaires
`« 0 → sp`	Contenu fictif dans 'sp'.
`  «`	Début du programme principal.
`  «`	Début du sous-programme 'sp'.
`  { 0 1 6 }`	Expression 'Expr' au niveau 2.
`  IF OVER TYPE POS`	Si c'est un nombre ou un nom,
`  THEN 1 →LIST`	On le met dans une liste.
`  ELSE`	Sinon:
`  OBJ→`	On extrait les sous-expressions
`  1 ROT START`	successives jusqu'à obtenir l'
`  SWAP`	expression complète.On a alors
`  sp EVAL`	au niveau 1 la position "n" de
`  SWAP`	l'opérateur racine.
`  +`	Extrait opérateur racine: {op}
`  NEXT`	{op},'Expr',n sur la pile.
`  END`	Si 'op' pas remplaçable par =,
`  »`	
`  'sp' STO`	On le met dans variable 'sp'.
`  sp EVAL`	Appelle le sous-programme.
`  →STR`	Transforme la liste en chaîne.
`  2 OVER SIZE 1 - SUB`	Enlève les caractères { et }.
`  "«" SWAP + "»" +`	Les remplace par « et ».
`  OBJ→`	Donne la forme "programme".
`  »`	
`»`	

Exemple: (en 4 secondes)

```
1: 'EXP(5*SQ(Z))+X/(Y-1)^A'   →RPN
                              —> 1: « 5 Z SQ * EXP X Y 1 - A ^ / + »
```

GRAPHISME : Corrigés des Exercices

Exercice 1 (*): revenir au paramètres graphiques par défaut.



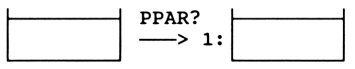
Le programme **'RESET'** doit placer dans **'PPAR'** la liste des para-mètres graphiques par défaut.

Il y a une première possibilité qui consiste à écrire:
« { (-6.5,-3.1) (6.5,3.2) X 0 (0,0) FUNCTION Y }
'PPAR' STO »

mais il y a plus malin:
* On commence par purger **'PPAR'**.
* On éxécute la séquence **1 *W**.

Texte du programme 'RESET'	Commentaires
« 'PPAR' PURGE 1 *W »	Purge éventuellement 'PPAR'. La recrée avec ses valeurs par défaut.

Exercice 2 (#): être certain de la présence d'un **'PPAR'** valable.



Le programme **'PPAR?'** doit tester la présence de **'PPAR'** dans le répertoire et la créer si elle ne s'y trouve pas.

Le programme **'PPAR?'** doit également recréer **'PPAR'** si son contenu n'est pas conforme à ce qu'attend la HP48SX.

On utilise là encore l'instruction **1 *W** (on pourrait tout aussi bien utiliser **1 *H**):

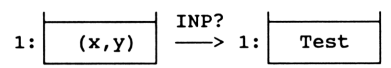
Si **'PPAR'** est absente du répertoire, cette instruction la crée avec ses valeurs par défaut (voir exercice précédent).
Si **'PPAR'** est présente et si son contenu est correct, alors **'PPAR'** est inchangée.
Si **'PPAR'** est présente mais si son contenu est incorrect, cette instruction provoque une erreur (**"Invalid PPAR"**).
Il suffit pour ce cas de prévoir un test **IFERR...THEN**.

Texte du programme 'PPAR?'	Commentaires
« IFERR 1 *W THEN 55 NEG FC? DROPN RESET END »	Si 1 *W est impossible, (mauvais 'PPAR'), alors: Effacer 1 si LASTARG active. Purger et recréer 'PPAR'.

On remarquera ici l'intruction **55 NEG FC? DROPN**, qui élimine la présence du chiffre 1 sur la pile, dans le cas où **1 *W** a provoqué une erreur et si **LASTARG** est active (indicateur -55 désarmé).

Dans le cas où **LASTARG** est inactive, le chiffre 1 disparaît avec l'erreur commise, et **55 NEG FC? DROPN** se réduit ici à **0 DROPN**, ce qui est parfait.
(voir l'ex.1 du chapitre consacré à la pile opérationnelle)

Exercice 3 (**##**): Tester si un point est dans la fenêtre graphique.



Il s'agit de tester si le point de coordonnées (x,y) du plan est ou n'est pas à l'intérieur de la fenêtre graphique que **PICT** ouvre sur le plan.
S'il y est, Test vaut 1, sinon Test vaut 0.

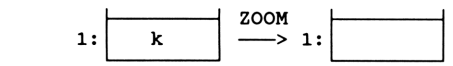
On doit tout d'abord appeler **'PPAR?'** pour être certain de la présence, dans le répertoire, d'un **'PPAR'** correct.

Si on note (Xmin,Ymin) les coordonnées dans le plan du coin inférieur gauche de **PICT**, et (Xmax, Ymax) celles du coin supérieur droit de **PICT**, alors le point de coordonnées (x,y) est intérieur à la fenêtre si:

(Xmin-x)*(Xmax-x)≤0, et si (Ymin-y)*(Ymax-y)≤0.

Texte du programme 'INP?'	Commentaires
« PPAR? PPAR 1 GET OVER - OBJ→ PPAR 2 GET 4 ROLL - OBJ→ ROT * 0 ≤ 3 ROLLD * 0 ≤ AND »	S'assurer d'un PPAR correct. (Xmin, Ymin). Xmin-x au dessus de Ymin-y. (Xmax, Ymax). Xmax-x au dessus de Ymax-y. 1 si (Ymin-y)*(Ymax-y)≤0. Remonte ce premier test. 1 si (Xmin-x)*(Xmax-x)≤0. 1 si le point (x,y) est dans la fenêtre, 0 sinon.

Exercice 4 (***): Zoom sur la fenêtre graphique.



Il s'agit ici d'effectuer un zoom sur la fenêtre graphique.
k (réel > 0) est le coefficient de ce zoom (zoom avant si k<1, zoom arrière si k>1).

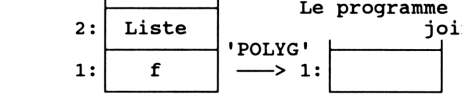
Le centre de la fenêtre graphique doit être invariant dans cette opération (qui consiste donc à multiplier par le coefficient k l'amplitude de l'intervalle des 2 coordonnées,en conservant le centre de la fenêtre).

Si on appelle A le point inférieur gauche de la fenêtre, B le point supérieur droit, Ω le centre du segment AB, et v le vecteur d'origine A et d'extrémité Ω, on doit remplacer A par A', et B par B', avec: A' = Ω -k*v, B' = Ω + k*v.

On utilise ici les instructions **PMIN** et **PMAX** (qui ne figurent dans aucun menu de la HP48SX)

Texte du programme 'ZOOM'	Commentaires
«	
PPAR?	Etre sûr d'un 'PPAR' correct.
PPAR OBJ→ 6 DROPN	Restent A=Pmin, B=Pmax.
→ k a b	k: le coeff. du zoom.
« a b + 2 /	Le point Ω, centre de [A,B].
DUP a -	Le vecteur v joignant A à Ω.
DUP2	
k * + PMAX	B' = Ω + k*v.
k * - PMIN	A' = Ω - k*v.
»	
»	

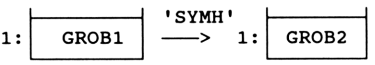
Exercice 5 (**): Tracer un polygône.



Le programme **'POLYG'** trace la ligne brisée qui joint les points (a,b),(c,d), etc.... qui figurent dans la liste placée au niveau 2.
Au niveau 1 le réel f vaut zéro si on ne veut pas que la ligne brisée se referme sur elle-même. Si f est non nul, le dernier segment tracé est celui qui joint le dernier point (y,z) de la liste au premier point (a,b): on obtient ainsi un polygône fermé.

Texte du programme 'POLYG'	Commentaires
«	
NOT	0 si on veut fermer, 1 sinon.
1 3 PICK SIZE ROT -	Extrémités de boucle.
FOR i	Pour l'indice i:
DUP i GETI	Le i-ème point,
3 ROLLD	monté au niveau 3.
GET	Le point suivant.
LINE	On trace le segment.
NEXT	Au suivant...
DROP	Effacer la liste.
»	

Exercice 6 (**): Symétrie horizontale sur un objet-graphique.

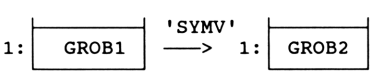


'SYMH' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une symétrie d'axe horizontal.

Le plus simple est de créer un objet blanc de mêmes dimensions que **GROB2** (instruction **BLANK**), puis de placer les lignes successives de **GROB1** (extraites par **SUB**), de la ligne la plus haute à la ligne la plus basse, dans les lignes successives de **GROB2** (placées par **REPL**) de la ligne la plus basse à la ligne la plus haute.

Texte du programme 'SYMH'	Commentaires
« DUP SIZE → g w h « w h BLANK 0 h 1 - B→R FOR i g {#0} i R→B + w 1 - i R→B 2 →LIST SUB {#0} h 1 - i - + SWAP REPL NEXT » »	g = GROB1, w = largeur de GROB1, h = hauteur de GROB1. GROB2 "blanc" de taille w x h. De i=0 à i=(h-1). GROB1 Liste { #0 #i } Liste { #(w-1) #i } On extrait la i-ème ligne de GROB1. Liste { #0 #(h-1-i) } On place la ligne extraite en ligne (h-1-i) de GROB2. Ligne suivante de GROB1.

Exercice 7 (**): Symétrie verticale sur un objet-graphique.

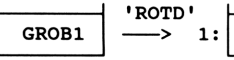


'SYMV' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une symétrie d'axe vertical.

On applique la même méthode que précédemment, en agissant cette fois-ci sur les colonnes de pixels.

Texte du programme 'SYMV'	Commentaires
« DUP SIZE → g w h « w h BLANK 0 w 1 - B→R FOR i g i R→B {#0} + i R→B h 1 - 2 →LIST SUB w 1 - i - {#0} + SWAP REPL NEXT » »	g = GROB1, w = largeur de GROB1, h = hauteur de GROB1. GROB2 "blanc" de taille w x h. De i=0 à i=(w-1). GROB1 Liste { #i #0 } Liste { #i #(h-1) } On extrait la i-ème colonne de GROB1. Liste { #(w-1-i) #0 } On place la colonne extraite en colonne w-1-i de GROB2. Colonne suivante de GROB1.

Exercice 8 (***): Rotation de 90° à droite sur un objet-graphique.



'ROTD' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une rotation de 90° dans le sens des aiguilles d'une montre (donc "vers la droite").

Ici, il n'est pas possible de travailler sur les lignes ou les colonnes de pixels de **GROB1** (tout simplement parce qu'une ligne de pixels de **GROB1** doit devenir une colonne de pixels de **GROB2**, et qu'une telle transformation n'est pas possible d'un coup).

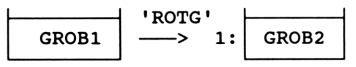
On doit donc travailler "pixel par pixel".
Le seul moyen de le faire est d'utiliser les instructions **PIX?** (pour savoir si un pixel est allumé dans **GROB1**) et **PIXON** (pour allumer un pixel dans **GROB2**).

Il se trouve que les instructions **PIX?** et **PIXON** ne s'appliquent qu'au contenu de **PICT**.
On est donc amené à sauvegarder le contenu actuel de **PICT**, avant de redimensionner **PICT** (ce qui le nettoie) de façon à ce qu'il puisse contenir simultanément **GROB1** et **GROB2**.
On place **GROB1** dans le coin supérieur de **PICT**, la place de **GROB2** étant située, par exemple, au dessous de celle de **GROB1**.
On lit successivement tous les pixels de la partie de **PICT** qui contient **GROB1**, et on allume dans celle contenant **GROB2** les pixels correspondant à ceux qui étaient allumés dans **GROB1**.

Il suffit ensuite d'extraire de **PICT** l'objet-graphique **GROB2** ainsi constitué, avant de rétablir l'ancien contenu de **PICT**.

Texte du programme 'ROTD'	Commentaires
`« DUP SIZE`	
`→ w h`	w = largeur de GROB1.
`« PICT RCL`	h = hauteur de GROB1.
`w h + DUP PDIM`	Sauvegarde PICT sur la pile.
`PICT`	Définit un PICT blanc, carré de
`{ #0 #0 } 4 ROLL REPL`	coté w+h.
`0 w 1 - B→R FOR i`	Place GROB1 en haut à gauche.
`0 h 1 - B→R FOR j`	De i=0 à i=(w-1)
`IF i R→B j R→B`	De j=0 à j=(h-1)
`2 →LIST PIX?`	Si le point { #i #j } est
`THEN`	allumé dans GROB1
`h 1 - j -`	Alors on allume le point qui
`h i + 2 →LIST`	correspond dans GROB2
`PIXON`	(coordonnées #(h-1-j) et
`END`	#(h+i) dans PICT)
`NEXT`	Descend d'un pixel dans GROB1.
`NEXT`	Passe colonne suivante de GROB1.
`PICT`	Extrait GROB2 de PICT (rectangle
`{ #0 } h +`	d'extrémités {#0 #h} et
`h 1 - DUP w +`	{ #(h-1) #(h-1+w) }
`2 →LIST SUB`	
`SWAP PICT STO`	Rétablit le contenu initial de
`»`	PICT.
`»`	

Exercice 9 (***): Rotation de 90° à gauche sur un objet-graphique.



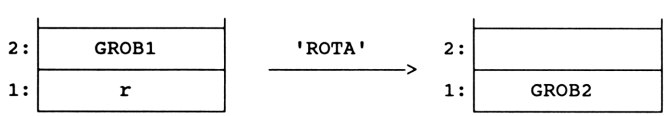
'ROTG' transforme l'objet-graphique **GROB1** en son image **GROB2** dans une rotation de 90 degrés dans le sens inverse des aiguilles d'une montre (sens "trigonométrique").

La méthode est semblable à celle de l'exercice 8, la seule différence résidant dans le calcul des coordonnées du point qui dans **GROB2** correspond à un point donné de **GROB1**.

On a souligné ci-dessous le seul passage différentiant **'ROTD'** et **'ROTG'**.

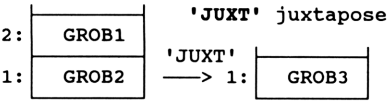
Texte du programme 'ROTG'	Commentaires
« DUP SIZE → w h « PICT RCL w h + DUP PDIM PICT { #0 #0 } 4 ROLL REPL 0 w 1 - B→R FOR i 0 h 1 - B→R FOR j IF i R→B j R→B 2 →LIST PIX? THEN j R→B 'h+w-1-i' →NUM 2 →LIST PIXON END NEXT NEXT PICT { #0 } h + h 1 - DUP w + 2 →LIST SUB SWAP PICT STO » »	w = largeur de GROB1. h = hauteur de GROB1. Sauvegarde PICT sur la pile. Définit un PICT blanc, carré de coté w+h. Place GROB1 en haut à gauche. De i=0 à i=(w-1) De j=0 à j=(h-1) Si le point { #i #j } est allumé dans GROB1 Alors on allume le point qui correspond dans GROB2 (coordonnées #j et #(h+w-1-i) dans PICT) Descend d'un pixel dans GROB1. Passe colonne suivante de GROB1. Extrait GROB2 de PICT (rectangle d'extrémités {#0 #h} et { #(h-1) #(h-1+w) } Rétablit le contenu initial de PICT.

Je vous laisse imaginer un programme permettant d'effectuer la rotation de 90° dans un sens ou dans l'autre, suivant le signe du réel placé au niveau 1:



Rotation de 90° dans le sens positif (trigonométrique) si le réel r est ≥0, et dans le sens négatif (sens des aiguilles d'une montre) sinon.

Exercice 10 (**): Juxtaposition de deux objets-graphiques.



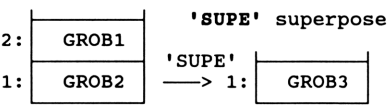
'JUXT' juxtapose deux objets-graphiques **GROB1** et **GROB2** (ayant la même hauteur), pour en faire un objet-graphique **GROB3** (**GROB1** et **GROB2** y sont donc placés côte à côte, **GROB1** à gauche de **GROB2**).

La méthode est simple:

* Soit h la hauteur commune de GROB1 et GROB2.
  Soit w1 la largeur de GROB1 et w2 celle de GROB2.
* On crée un objet-graphique blanc de dimension (w1+w2) x h.
* On y place **GROB1** au point { #0 #0 }.
* On y place **GROB2** au point { #(w1) #0 }.

Texte du programme 'JUXT'	Commentaires
« DUP2 SIZE ROT SIZE → w2 h2 w1 h1 « IF h1 h2 == THEN w1 w2 + h1 BLANK w1 { #0 } + ROT REPL { #0 #0 } ROT REPL END » »	w1 x h1 = dimensions de GROB1. w2 x h2 = dimensions de GROB2. Si hauteurs égales, alors: GROB blanc de dim. (w1+w2)xh1 On y place GROB2 en #w1,#0. On y place GROB1 en #0,#0.

Exercice 11 (**): Superposition de deux objets-graphiques.

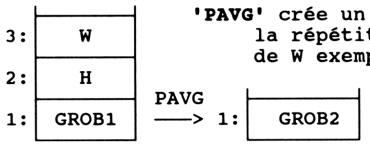


'SUPE' superpose deux objets-graphiques **GROB1** et **GROB2** (ayant la même largeur), pour en faire un objet-graphique **GROB3** (**GROB1** et **GROB2** y sont donc placés l'un au dessus de l'autre, **GROB1** au-dessus de **GROB2**).

La méthode est la même que dans l'exercice précédent.

Texte du programme 'SUPE'	Commentaires
« DUP2 SIZE ROT SIZE → w2 h2 w1 h1 « IF w1 w2 == THEN w1 h1 h2 + BLANK { #0 } h1 + ROT REPL { #0 #0 } ROT REPL END » »	w1 x h1 = dimensions de GROB1. w2 x h2 = dimensions de GROB2. Si largeurs égales, alors: GROB blanc de dim. w1x(h1+h2) On y place GROB2 en #0,#h1. On y place GROB1 en #0,#0.

Exercice 12 (##): Pavage obtenu à partir d'un objet-graphique.



'PAVG' crée un objet graphique **GROB2** constitué de la répétition de **GROB1** (**GROB2** est un pavage de W exemplaires de **GROB1** en largeur sur H exemplaires de **GROB1** en hauteur).

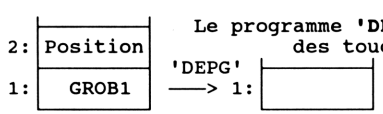
On utilise les programmes **'JUXT'** et **'SUPE'**.

'JUXT' nous permet de créer la première ligne du pavage, en juxtaposant W exemplaires de **GROB1**.

'SUPE' nous permet d'obtenir le pavage voulu, en superposant H exemplaires de la première ligne obtenue ci-dessus.

Texte du programme 'PAVG'	Commentaires
« → w h g « g w WHILE 2 ≥ REPEAT g JUXT 'w' DECR END DUP h WHILE 2 ≥ REPEAT OVER SUPE 'h' DECR END SWAP DROP » »	g = GROB1. Place GROB1 sur la pile Tant qu'il faut continuer à juxtaposer g à la ligne en cours, on le fait. On a obtenu la 1ère ligne. On la duplique. Tant qu'il faut la superposer au pavage en cours de forma-tion, on le fait. Elimine exemplaire de la ligne du pavage, devenue inutile.

Exercice 13 (*****): Déplacements d'un objet-graphique dans **PICT**.



Le programme **'DEPG'** permet de déplacer, à l'aide des touches de curseur, l'objet-graphique GROB1 sur le contenu affiché de PICT.

Dans la solution, on utilise une structure **WHILE..REPEAT..END** dont le sens est à peu près le suivant:

Tant que:
- On affiche **GROB** (à sa position initiale, ou à sa nouvelle position s'il y a eu déplacement).
- On attend qu'une touche soit appuyée.
- On calcule le déplacement correspondant.
- Ce déplacement est non nul.

Alors:
- On efface **GROB**.
- On calcule la nouvelle position.
- On retourne après le "Tant que:"

L'attente de la frappe d'une touche se fait par l'instruction **0 WAIT** qui stoppe l'éxécution du programme jusqu'à ce qu'une touche soit actionnée et qui renvoie un nombre décimal T.P où:

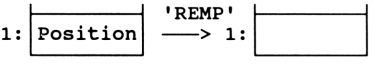
T:entier définissant la position de la touche (10*n° de ligne + n° de colonne).
La touche **VAR** a par exemple le code T = 10*2+4 = 24.
P:entier définissant le préfixage éventuel:
Pas de préfixe: P=1.
Préfixes <⌐, ╔>, α, α <⌐, α ╔>: P vaut respectivement 2, 3, 4, 5, 6.

On calcule le déplacement (a,b) résultant de la touche de direction qui a été actionnée (Ex: (0,-1) si c'est la touche Δ car il faut diminuer d'une unité la hauteur en pixels.
On multiplie ce déplacement (a,b) par 2^(P-1), où P a le sens ci-dessus: La séquence α Δ donne donc le déplacement (0,-8).

Si la touche actionnée n'est pas une touche de direction, on définit un déplacement nul, ce qui permet d'effectuer le test de continuation du programme.
Sinon, on efface **GROB** de sa position actuelle (l'affichage et l'effacement s'effectuent à l'aide deux **GXOR** successifs), et on calcule la nouvelle position (avant de retourner à la nouvelle scrutation du clavier, précédée de l'affichage de **GROB** dans sa nouvelle position).

Texte du programme 'DEPG'	Commentaires
`« SWAP`	'Position' au niveau 1.
`IF DUP TYPE 1 == THEN`	Si elle est en unité-utilisateur
`C→PX`	revenir aux unités-pixels.
`END`	
`→ g p`	g = GROB, p = Position.
`« { #0 #0 } PVIEW`	Visualiser PICT.
`WHILE`	Tant que:
`PICT p g GXOR`	On affiche GROB.
`0 WAIT`	Code T.P de la touche appuyéé
`DUP FP 10 *`	Entier P (1,2,3,4,5,ou 6).
`SWAP IP`	Entier T.
`{ 0 (0,-1) (-1,0)`	Liste des déplacements.
`(0,1) (1,0) }`	
`{ 25 34 35 36 }`	Liste des entiers P possibles
`ROT POS 1 + GET`	Déplacement choisi (a,b).
`2 ROT 1 - ^ *`	On muliplie par 2^(P-1).
`DUP ABS`	Déplacement non nul ?
`REPEAT`	Répéter:
`PICT p g GXOR`	Effacer GROB.
`p EVAL B→R SWAP B→R`	Ajouter le déplacement, qui
`SWAP R→C`	est un nombre complexe,à la
`+`	liste de 2 entiers binaires
`OBJ→ R→B SWAP R→B`	donnant la position 'p' et
`SWAP 2 →LIST 'p' STO`	placer le résultat dans 'p'
`END`	Repartir après le WHILE.
`DROP`	Nettoyer pile avant de sortir.
`»`	
`»`	

Exercice 14 (*****): remplissage d'un contour fermé.



'REMP' remplit le contour fermé entourant le point A dont la position figure au niveau 1.

Le remplissage se fait dans **PICT**, qui est affiché pour la circonstance.

Texte du programme 'REMP'	Commentaires
« PICT SIZE 0	w, h: largeur et hauteur de PICT
→ w h sp	0 contenu fictif pour 'sp'.
«	Début du sous-programme 'sp'.
« IF DUP PIX? NOT	Si le point n'est pas allumé:
THEN	.
EVAL OVER	. b=ordonnée du point.
→ ag b ad	. ag=ad=abscisse du point.
« WHILE	. Tant que:
ag #0 >	. le point à gauche existe et
ag 1 - b 2 →LIST	. est éteint, on dimininue ag
PIX? NOT AND	. de 1.
REPEAT	.
'ag' 1 STO-	.
END	.
WHILE	. Tant que:
ad w <	. le point à droite existe et
ad 1 + b 2 →LIST	. est éteint, on augmente ad
PIX? NOT AND	. de 1.
REPEAT	.
'ad' 1 STO+	.
END	.
ag b 2 →LIST	. On trace le segment joignnt
ad b 2 →LIST	. les points { #ag #b } et
LINE	. { #ad #b }.
IF b #0 > THEN	. Si b>0, on appelle 'sp' pour
ag B→R ad B→R	. tous les points {#i #(b-1)}
FOR i	. où i varie de ag à ad.
i R→B b 1 -	. (C'est-à-dire tous ceux qui
2 →LIST	. sont juste au-dessus du
sp EVAL	. segment tracé par LINE).
NEXT	.
END	.
IF b h < THEN	. Si b<h, on appelle 'sp' pour
ag B→R ad B→R	. tous les points {#i #(b+1)}
FOR i	. où i varie de ag à ad.
i R→B b 1 +	. (C'est-à-dire tous ceux qui
2 →LIST	. sont juste au-dessous du
sp EVAL	. segment tracé par LINE)
NEXT	.
END	.
»	.
ELSE DROP END	Sinon on nettoie la pile.
» 'sp' STO	Place sous-programme dans 'sp'.
IF DUP TYPE 1 == THEN	Si le point initial est donné en
C→PX	unités-utilisateur, le passer
END	en unités-pixels.
{ #0 #0 } PVIEW	Visualiser PICT.
sp EVAL	Appeler le sous-programme 'sp'
» »	avec le point initial.

Dans la solution présentée ici, on adopte une démarche récursive:

Un sous programme (nommé ici 'sp' et qui s'appelera lui même) prend comme argument un point.

Si ce point est déjà allumé, 'sp' ne fait rien.

Si ce point A est éteint, le sous programme 'sp' explore les points situés à droite et à gauche de A (sur une même horizontale), jusqu'à rencontrer des points allumés ou la frontière de **PICT**.
On peut alors tracer le segment horizontal CD, où CD est le plus large segment horizontal, contenant A, et formé de points éteints.

Il suffit alors d'appeler le sous programme 'sp' pour tous les points situés immédiatement au dessus ou en dessous du segment CD.

Cette solution peut paraître lente à l'éxécution (notamment parce qu'elle conduit à examiner de nombreux points qui en fait viennent d'être précédemment allumés).
Cette lenteur (toute relative) est d'autant plus sensible sur des contours simples (rectangulaires).

Par contre cette méthode a l'avantage de s'appliquer à tous les contours fermés entourant le point initial A (aussi tortueux soient-ils).

INDEX

% 112
+/- 95, 290
π (=3.14...) 105
+/- (Dans GRAPHICS) 284
* (Dans GRAPHICS) 284
+ (Dans GRAPHICS) 284
- (Dans GRAPHICS) 284
! (factorielle) 116
Σ (Sommations) 238
| (WHERE) 236
'→HALT' 153, 378
'ΣNK' 125
'ΣNPK' 346
'ΣP3' 124, 341
'ΣPK' 125, 345
'→RPN' 257, 446
'→TREE' 223, 426
'→VL' 90, 295
'AIRE' 122, 332
'ALIG' 123, 336
'ALLR' 223, 427
'ANGL' 121, 331
→ARRY 175
'BISS' 122, 335
'CALCC' 206, 420
'CALCL' 206, 421
%CH 112
'COCYC' 123, 337
'CODST' 153, 377
'CODVA' 153, 378
+COL 161
-COL 161
'COMA' 204, 410
'CONCT' 201, 390
'CONST' 151, 369
'CRTAB' 205, 415
'CVGLE' 121, 331
'CYCST' 93, 322
2D 114, 194
3D 194
2D-vecteurs 163
3D-vecteurs 164
'DC135' 124, 342
'DEF' 155, 384
→DEL 20
'DELET' 136, 152, 352, 373
'DELN' 91, 309
'DEPG' 293, 456
'DEPR' 223, 428
'DEPV' 222, 424
'DERN' 254, 432
'DICHO' 255, 435
'DIFF' 138, 357
'DIMT' 200, 388
'DISPC' 154, 380
'DISPD' 154, 379
'DIVPR' 205, 418
'DRTE' 122, 334
'E250' 125, 348
'ECHAN' 136, 152, 351, 373
'ECHC' 203, 403
'ECHL' 203, 404
'ECHST' 90, 299
'GEB' 124, 343
'GETC' 203, 402
'GETL' 203, 401
'GETM' 204, 412
'GETST' 92, 320
→GROB 268
*H 281
'HALT→' 379
'HALT→' 154
→HMS 120
'IDEM' 139, 360
'INL' 138, 360
'INP?' 291, 450
'INSC' 202, 397
'INSER' 137, 152, 353, 375
'INSL' 202, 396
'INST' 91
'INTD' 254, 431
'INTER' 137, 356
'INVER' 153, 375
'INVPR' 206, 419
'INVST' 306
'ITER' 256, 438
'JUXT' 292, 455
'KAPR1' 123, 338
'KAPR2' 123, 339
'L→T' 204
'L→T' 407
'LAGRA' 255, 436
→LCD 69, 270
'LDIV' 124, 340
→LIST 79, 130
'LSL' 139, 362
'MAJU' 151, 370
'MDTR' 122, 335
'MINE' 204, 410
'MINU' 152, 371
'MJMI' 152, 372

'MODT' 205, 417
'MORE' 139, 361
'N→ST' 123, 338
'NEWT' 255, 437
→NUM 106
'NUML' 137, 356
'OCCUR' 137, 152, 352, 374
'PARF?' 124, 340
'PART' 125, 347
'PAVG' 292, 456
'PBIH' 91, 314
'PERI' 122, 332
'PERP' 122, 334
'PICKB' 92, 318
'PLAY' 155, 385
'POLYG' 291, 451
'PPAR?' 291, 449
'PROJ' 122, 333
'PTMI' 256, 439
'PUTC' 203, 400
'PUTL' 202, 399
'PUTM' 205, 414
→Q 114
→Qπ 114
'RANDL' 139, 364
'REDUC' 137, 355
'REMP' 153, 293, 376, 458
'RENV' 137, 354
'RESET' 291, 449
'REUN' 138, 359
'REUT' 154, 381
'RNDST' 91, 305
'RNMV' 222, 423
'ROLDB' 92, 317
'ROLLB' 92, 316
'ROOTZ' 257, 443
'ROTD' 292, 453
'ROTG' 292, 454
+ROW 161
-ROW 161
'SEPT' 154, 383
'SEPTL' 201, 391
→SKIP 20
'SMPS' 256, 441
'SOLN' 254, 433
'ST→N' 123, 338
→STK 161, 248
↑STK 20, 69, 161
→STR 97, 148
'SUPC' 202, 395
'SUPE' 292, 455
'SUPL' 202, 394
'SUPT' 201, 389
'SYM' 122, 333
'SYMH' 292, 452
'SYMV' 292, 452
%T 112
'T.BUL' 365
'T.BULL' 140
'T.INS' 140, 366
'T→L' 404
'T.SEL' 141, 367
'T→V' 155, 384
'TR?' 200, 387
'TR→TC' 200, 387
'TRCRT' 93, 326
'TREE→' 222, 425
'TRPZ' 256, 440
'TRSTK' 93, 324
'TYPST' 91, 315
'UN' 124, 344
'UNIQ' 139, 362
→V2 113, 114, 193
→V3 194
'VALF' 254, 434
'VMC' 202, 393
'VML' 201, 392
*W 281
→WID 161
'XORL' 138, 358
'ZOOM' 291, 451

A

ABS 110, 189
ACOS 119
ACOSH 118
ALG 17, 25, 226
ALG PRG 17
ALGEBRA 229, 248
ALOG 117
AND 32, 146
ARC 266
AREA 287
ARG 83, 111
Argument 101
ARRY 174
ASCII 144
ASIN 119
ASINH 118
ATAN 119
ATANH 118
AUTO 276
AXES 280

B

Bad Guesse(s) 252
BAR 272
BEEP 26
BLANK 268
BOX 266, 289
BYTES 215

C

C$ 144
C→R 113, 176
Calcul différentiel 225
Calcul intégral 225
CASE...END 38
CAT 248, 273
Catalogue des expressions 248
CEIL 112

CENT 279, 285
CF 106
Chaînes comptées 144
Chaînes de caractères 143
CHR 151
CIRCL 289
CLEAR 72
CLR 72
CLR (Dans GRAPHICS) 284
CLVAR 220
CNCT 275
CNRM 190
COLCT 229
COMB 116
Comparaison de deux chaînes 147
CON 182
Concaténation de deux chaînes 145
Concaténation de deux listes 128
CONIC 272, 273
CONJ 111, 194, 196
Constant? 252
CONT 58, 60, 82
COORD 285
Coordonnées polaires 164
Coordonnées sphériques 164
COS 119
COSH 118
CRDIR 210, 217
CROSS 191
CST 25
Curseur 16, 19

D
D→R 121
DBUG 61
DECR 220
DEFINE 221
DEG 102, 119, 165, 240
DEL 19, 20, 290
DEL→ 20
DEL (Dans GRAPHICS) 284
Déplacements 20
DEPN 278
DEPTH 76
Dérivation 239
Dérivation numérique 240
Dérivation symbolique 240
DET 187
DISP 69, 144, 259, 268
DO...UNTIL...END 46
Données 21, 28
DOT 192
DOT+ 288
DOT- 288
DRAW 272, 276
DRAX 280
DROP 71
DROP2 77
DROPN 78
DRPN 79
DUP 72
DUP2 76
DUPN 77, 79

E
e (=EXP(1)) 105
ECHO 79
Echo 162
EDEQ 247, 273
EDIT 19, 161, 163, 248
EDIT (Menu) 16, 69
EEX 96
ENG 100
ENTER 18, 27, 68, 72
ENTER (Dans GRAPHICS) 283
Entiers binaires 11
ENTRY 17
EQ 245, 272
EQ+ 248
EQ- 238
Equation-writer 16, 208
Equations 225
ERASE 275
ERRM 50
ERRN 50
EVAL 27, 29, 106
EVAL (Sur une liste) 129
Evaluation des objets 21
Evaluer un Programme 27
EXIT 287
EXP 117
EXPAN 230
EXPM 117
Exposant 96
EXPR= 250
Expression en cours 273
Expressions 12, 225
Extension .EQ 246
EXTR 287
Extremum 252

F
F' 288
F(X) 287
Factorielle 116
FAST 249
Faux (sur la pile) 31
FCN 285
FIX 98, 114, 149, 244
FLOOR 112
FM 18, 26, 95
FOR...NEXT 41
FOR...STEP 41
FP 112
FREEZE 68, 260, 265, 271, 283
FUNCTION 272, 273

G
Gamme de valeurs 277
Gamme horizontale 276
Gamme verticale 276
GET 132, 135, 179, 197
GETI 133, 135, 181, 197
GO→ 161
GO↓ 161
GOR 268
GRAD 102, 119, 165, 240
GRAPH 260, 283
GRAPH (Dans GRAPHICS) 283
GRAPHICS 260, 282
Graphisme 259
GROB 261
GXOR 269

H
HALT 58, 60, 69, 81, 260
HISTOGRAM 272
HMS+ 120
HMS- 120
HMS→ 120
HOME 24, 209

I
i (=(0;1)) 105
IDN 183, 196
IERR 245
IF..THEN..ELSE..END 33
IFERR..THEN..(ELSE)..END 49, 59
IFERR et LASTARG 87
IFT 37
IFTE 37
IM 111, 195
INCR 220
INDEP 277
Indicateur -1 232
Indicateur -2 106, 233, 240
Indicateur -3 105, 106, 221, 241
Indicateur -16 102, 166
Indicateur -17 103, 119, 165, 166
Indicateur -18 103, 119, 165
Indicateur -19 114, 194
Indicateur -20 104
Indicateur -21 105
Indicateur -22 105
Indicateur -23 105
Indicateur -24 105
Indicateur -25 105
Indicateur -26 105
Indicateur -30 274
Indicateur -31 275
Indicateur -52 70
Indicateur -55 83
INS 16
INS▪ 20
Insertion 16
Intégration 241
Intégration numérique 244
Intégration sur la pile 242
Intégration symbolique 243
INV 109, 170
IP 112
ISECT 286
ISOL 231, 232

K
KEEP 79
KEYS 290
KILL 58

L
LABEL 281, 285
LAST ARG 23
LAST CMD 18
LAST STACK 18, 80
LASTARG 31, 51, 83, 199, 209
LCD→ 261, 270
LEFT: 250
LEVEL 79
Ligne de commande 16, 68
LINE 265, 289
Listes 12, 22, 127
LN 117
LNP1 117
LOG 117

M
MANT 112
Mantisse 96
MARK 289
Matrices 12, 158
Matrix-writer 16, 160, 167, 208
MAX 111
MAXR 104, 115
MEM 215
MEMORY (Menu) 213, 215
Menus 25
MIN 111
MINR 104, 115
Mise au point d'un programme 57
ML 70, 158, 166
MOD 111
Mode algébrique 17
Mode alpha 17, 143
Mode compact 70
Mode multi-lignes 70
Mode programme 17
MODES (Menu) 18, 70, 80, 83, 95, 97, 102, 119, 158, 165, 166, 221, 275
Module 101

MTH HYP (Menu) 117, 118
MTH MATR (Menu) 182
MTH PARTS (Menu) 110, 111, 115, 194
MTH PROB (Menu) 116
MTH VCTR (Menu) 102, 121, 166, 191

N

NEG 173
NEW 246, 273
NEXT 25
Nombres réels ou complexes 95
Noms 12, 22
Noms sans cotes 22
NOT 31, 145
Notation polaire 163
Notation RPN 14
NUM 150
NXEQ 249, 288

O

OBJ→ 130, 148, 174, 193, 237
Objets 11
Objets-graphiques 13, 260
Objets-préfixés 13
Objets-répertoires 214
Objets-sauvegardes 13
Objets-unités 13
ON 68
Opérandes 225
Opérateurs 225
Opérateurs binaires 227
Opérateurs unaires 227
Opération ^ 110
Opérations + - * / +/- 108
Opérations sur les tableaux 168, 169, 171, 172
OR 32, 146
ORDER 215, 249
OVER 73
Overflow 97, 104

P

PARAMETRIC 272, 274
Partie imaginaire 101
Partie réelle 101
PATH 217
PC→X 262, 267
PDIM 281
PERM 116
PGDIR 213
PICK 75, 79
PICT 260, 264, 275, 276, 280, 281
Pile interactive 14, 69, 78, 162
Pile opérationnelle 13, 67
PIX? 267
Pixel 259
PIXOFF 267
PIXON 267
PLOT (Menu) 272
PLOTR 248
PLOTR (Menu) 275
PMAX 263, 451
PMIN 263, 451
POLAR 102, 166, 272, 274
POS 131, 149
PPAR 272
PREV 25
PRG 17, 25, 27, 127
PRG ALG 226
PRG BRCH (Menu) 34, 40, 43, 47, 49, 51
PRG CTRL (Menu) 58
PRG DSPL (Menu) 262, 264
PRG OBJ→ (Menu) 148
PRG OBJ (Menu) 113, 130, 174
PRG STK (Menu) 72
Programme 27
Programmes 12, 22
PROMPT 61
PTYPE 273, 278
PURG 249
PURGE 23, 53, 209
PURGE et LASTARG 87
PUT 132, 177, 199
PUTI 133, 180, 199
PVIEW 69, 260, 264
PX→C 262, 267

Q

QUAD 231, 233

R

R∠∠ 102, 166
R→C 113, 176
R→D 121
R∠Z 102, 166
Racine carrée √ 109
RAD 102, 119, 165, 240
RCEQ 282
RDM 185, 196
RDZ 116
RE 111, 195
Recouvrement 16
Rectangulaire (forme) 101
Récursivité 62
Répertoires 24, 207, 209
REPL 131, 149, 269, 289
RES 279
RESET 280
Réticule 283
REVIEW 221, 246, 275
REVIEW (Dans GRAPHICS) 283

RIGHT: 250
RND 115, 116
RNRM 189
ROLL 74, 79
ROLLD 75, 79
ROOT 253, 286
ROT 73
RSD 187

S

SCALE 279
SCATTER 272
SCI 99
SCONJ 219
Séparateurs 18
SF 106
SHOW 234
SIGN 110
Sign Reversal 251
SIN 119
SINH 118
SINV 219
SIZE 131, 149, 177, 238, 267
SKIP→ 20
SLOPE 287
SNEG 219
Solution générale 231
SOLVE (Menu) 245
Solving For ... 250
SOLVR 248
SOLVR (Menu) 245, 249
SQ 109, 173
SST 59
SST↑ 59
START...NEXT 41
START...STEP 41
STD 98, 114, 244
STEQ 247, 273, 282
STK 80
STO 23, 207, 209
STO* 219
STO+ 218
STO- 219
STO/ 219
STO (Dans GRAPHICS) 283
STORE (Menu) 208, 218
STR→ 97
Structures locales 52
SUB 134, 150, 270, 290
SWAP 71
SYM 105, 221, 232, 241
Syntax Error 228

T

Tableaux 157
TAN 119
TANH 118
TAYLR 53, 235
TEXT 271
TIME (Menu) 120
TLINE 266, 289
Tracé d'un programme 274
Tracé d'une équation 274
Tracé d'une liste 274
TRN 184, 196
TRNC 115
TRUTH 272, 274
TVARS 218

U

Underflow 97, 104
UP 25, 212
UPDIR 25, 212

V

V→ 113, 193
Valeur principale 231
VAR 23, 25, 207
Variables 23
Variables globales 207
Variables locales 52
VARS 215
VEC 161
Vecteurs 12, 157
VIEW 24, 79, 163, 208, 248
Visibilité des variables locales 54
VISIT 19, 24, 208
Vrai (sur la pile) 31
VTYPE 221

W

WHERE (|) 236
WHILE...UNTIL...END 46
WID→ 161

X

X (Dans ZOOM) 284
XAUTO (Dans ZOOM) 284
XOR 32, 146
XPON 112
XRNG 276
XROOT 109
XY (Dans ZOOM) 284
XYZ 102, 166

Y

Y (Dans ZOOM) 284
YRNG 276

Z

Z-BOX 285
Zéro 251
ZOOM 284

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

Achevé d'imprimer par

31240 L'UNION (Toulouse)
Tél. (16) 61.74.27.67
Dépôt légal : Novembre 1990

TABLE DES MATIERES DU TOME 1

INTRODUCTION

PRINCIPES DE BASE

ELEMENTS DE PROGRAMMATION

LA PILE OPERATIONNELLE

NOMBRES REELS OU COMPLEXES

LES LISTES

LES CHAINES DE CARACTERES

LES TABLEAUX

VARIABLES GLOBALES ET REPERTOIRES

EXPRESSIONS, CALCUL DIFFERENTIEL ET INTEGRAL

GRAPHISME, CLAVIER, SON

INDEX

ISBN 2-908791-02-1

Prix 180 F TTC