Ralf Thoma

HP-Taschenrechner

Programmieren mit RPL

HEISE


HEISE

Ralf Thoma

HP-Taschenrechner

Programmieren mit RPL

Verlag Heinz Heise

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Thoma, Ralf:
HP-Taschenrechner : Programmieren mit RPL / Ralf Thoma. –
Hannover : Heise, 1995

ISBN 3-88229-052-8

Printed in Germany 5 4 3 2 1
1999 98 97 96 95

Umschlaggestaltung: MB-GRAFIK-DESIGN, Hannover
Druck: Kösel GmbH & Co, Kempten/Allgäu

ISBN 3-88229-052-8

Inhaltsverzeichnis

Vorwort

Dieses Buch wendet sich an einen Personenkreis, der die Möglichkeit der freien Programmierung der Taschenrechner von Hewlett-Packard (HP) gerne nutzen würde, sich aber mit der Einführung im Benutzerhandbuch etwas schwer tut.

Ich selbst erwarb meinen ersten HP-Taschenrechner 1989 (einen HP28S) und entwickelte im Lauf meines Studiums einige Hundert (!) Programme, die zur Hälfte allerdings nur an sehr spezielle Problemstellungen angelehnt wurden. Anfang 1994 stellte ich dann eine allgemein gehaltene Programmsammlung für die Taschenrechner vom HP28S bis zum HP48GX zusammen und präsentierte diese in Buch-Form auf der CeBIT '94. Die Resonanz war beeindruckend. Entsprechend wurden dort viele Anfragen nach einem Programmierlehrhandbuch gestellt. Dies war für mich der Anstoß zu diesem Buch. Das Werk soll eine Hilfe für den Einstieg in die Programmierung der Taschenrechner und kein Ersatz für das *Reference Manual* sein. Zudem beinhaltet es eine komplette Programmsammlung für Ingenieursmathematik. Jeden Befehl der Taschenrechner zu erklären, würde den Rahmen des Werks sprengen.

Auf jeden Fall sollte das Benutzerhandbuch aufmerksam gelesen worden sein. Damit ist eine einigermaßen sichere Bedienung des Taschencomputers sichergestellt.

Das Buch basiert auf den Rechnern HP28 und HP48 (S- und G-Modelle), ist aber so konzipiert, daß auch zukünftige Rechner mit Hilfe dieses Buchs zu programmieren sein sollten. Ich habe versucht, auch Programmieranfängern den Einstieg in die Programmiersprache RPL zu erleichtern. Leser, die bereits Erfahrung mit einer Programmiersprache gesammelt haben, werden sich naturgemäß beim Erlernen der Sprache RPL leichter tun.

Zuletzt möchte ich noch Volker Seibt und Rüdiger Hölzel meinen Dank für das Korrekturlesen aussprechen.

Erlangen, im Frühjahr 1995

1 Einleitung

1.1 Ein historischer Abriß

Hewlett-Packard verfügt insbesondere bei den Taschenrechnern über einen guten Ruf. Dies hat, wie die Nutzer der Rechner selbst wissen, seinen guten Grund. Die Geräte sind sehr leistungsfähig und robust. Seit über zwei Jahrzehnten kommt Hewlett-Packard immer wieder mit innovativen Geräten und Produktideen auf den Markt.

Eine komplette Liste aller Innovationen würde sicherlich zu weit führen, dennoch sollen hier einige der wichtigsten genannt werden [6]. Der HP-35A wurde zum ersten Taschenrechner überhaupt, als er 1972 zu einem Preis von 395 US$ auf den Markt kam. Er hatte die damals übliche 7-Segment-Leuchtdiodenanzeige, um seine maximal 15 Dezimalstellen darzustellen, wurde in der RPN-Logik bedient und verfügte über gerade mal 35 Byte Speicherkapazität.

Im Jahre 1975 brachte HP den ersten programmierbaren Taschencomputer zu einem Preis von 795 US$ heraus, den HP-65A. Dieser Rechner hatte immerhin schon 198 Byte Speicher, die ihm maximal 100 Programmschritte ermöglichten. Er hatte ebenfalls noch die 7-Segment-LED-Anzeige, die wiederum über 15 Stellen verfügte.

Mit dem HP-41C, der große Verbreitung mit verschiedenen Untertypen (CX- und CV-Versionen) fand, kam 1979 der erste alphanumerische Taschenrechner auf den Markt. Dieser hatte nun schon eine LCD-Anzeige und verfügte je nach Modell über eine Speicherkapazität zwischen 448 und 3129 Byte, die er in bis zu 1000 Programmschritten nutzen konnte. Das ROM hatte 12 kB, beim später auf dem Markt erschienenen CX sogar schon 24 kB Größe. Die Preise für diesen Computer betrugen je nach Modell zwischen 295 und 325 US$.

Einen weiteren großen Schritt tat HP 1987 mit dem HP28C zu einem Preis von 235 US$. Es wurde der erste Rechner mit symbolischer Algebra. Dieser Taschenrechner verfügte über 2 kB RAM und 64 kB ROM und hatte schon die Programmiersprache RPL (=Reverse Polish Lisp), in der der HP48G/GX auch heute noch programmiert wird. Der Computer verfügte über ein 4-Zeilen-Display mit je 23 Zeichen. Der 4-Bit-Prozessor wurde mit 1 MHz getaktet.

Die Weiterentwicklung des HP28C führte 1988 zum HP28S, der als wesentlichen Unterschied zum HP28C über 32 kB RAM und 128 kB ROM verfügte.

Mit dem HP48S/SX wurde im Jahre 1990 das Manko des kleinen LC-Displays der HP28-Typen beseitigt. Für 250 US$ bekam man nun 32 kB RAM und 256 kB ROM an Speicher, bzw. für 350 US$ mit dem HP48SX einen aufrüstbaren Taschenrechner, der mit RAM- oder ROM-Karten in seinen Fähigkeiten wesentlich erweitert werden konnte. Dieser Taschenrechner hatte zudem eine serielle 4-Bit-Schnittstelle, um Datentransfer zum PC zu ermöglichen.
Mit dem HP48G/GX, bei dem die Grafikfähigkeiten erheblich erweitert wurden, ist nun ein vorläufig letzter Höhepunkt erreicht worden.

1.2 Zur Notation in diesem Buch

In diesem Buch wird bei der Notation eine Konvention verwendet, wie sie ähnlich auch in den Benutzerhandbüchern Anwendung findet.

- Wenn es sich um Befehle handelt (z.B. ROLL), steht das Wort in Großbuchstaben allein da.

- Ist von Funktionsaufrufen die Rede, so sind diese Funktionen in diesem Buch umrandet (z.B. ENTER). Funktionsaufrufe sind eine Abfolge von zu drückenden Tasten. Achtung! Um das vorliegende Buch für alle Leser (vom HP28C/S über den HP48S/SX zu den HP48G/GX) gleich gut lesbar zu machen, wurde darauf verzichtet, für jedes Modell die individuelle Tastenkombination zum Aufruf einer Funktion anzugeben, da dies zu Verwirrungen führen würde.
  Als Beispiel diene hier EDIT. Beim HP28S ist dies über SHIFT (=rote Umschalttaste) und ENTER zu erreichen, während beim HP48G/GX hingegen dies über die LINKS-SHIFT-Taste und die +/−-Taste erreicht wird.

  Hier im Buch würde ich es also bei EDIT belassen. Oft ist die gesuchte Taste gar nicht auf der Tastatur zu finden, sondern man muß die Taste über die sechs Menütasten ansteuern. Im Anhang eines jeden Benutzerhandbuches sind jeder Befehl und jede Funktion mitsamt ihrem Aufruf beschrieben. Sollten Probleme auftreten, so kann ich auf die Benutzerhandbücher verweisen, die die Tastenbelegung und Funktionsaufrufe recht gut erklären.

  Sehr viele Funktionsaufrufe lassen sich auch als Befehle eintippen und in gleicher Form in Programmen nutzen. Als Beispiel sei hier →NUM genannt, welches den Rechner veranlaßt, ein Objekt numerisch auszuwerten. Genauso läßt sich aber über die Tastatur der Befehl →NUM eintippen. Der Effekt ist der gleiche.

- Den Stack in einer übersichtlichen Art und Weise wiederzugeben gestaltet sich insbesondere bei mehreren Elementen im Stack recht schwierig. Ich habe mich hier für zwei verschiedene Darstellungen entschieden, die aber beide recht einfach zu lesen sind.
  Beide sind hier auch als Beispiel aufgeführt. Es soll jeweils ein Stack mit drei Elementen (Par1 bis Par3) in Ebene 1 bis Ebene 3 gezeigt werden.

  Beispiel 1:

Ebene 3	Ebene 2	Ebene 1
Par3	Par2	Par1

  und Beispiel 2:

3:	Par3
2:	Par2
1:	Par1

  Der Grund für die verschiedenen Darstellungen liegt in der unterschiedlich guten Lesbarkeit. Bei Beispiel 2 gewinnt man zwar leichter einen Überblick, allerdings wird bei einer Darstellung eines Stacks mit mehreren Elementen die Lesbarkeit innerhalb eines Textes wieder schlechter. Der Grund liegt im sperrigen, hochgestellten Format. Deshalb wird im Text die Darstellung von Beispiel 1 ab drei Elementen im Stack verwendet.

- Der $\Longrightarrow$ -Pfeil wird in diesem Buch als Abkürzung der Formulierung „führt nach Ausführung durch *Befehl* ENTER zu“ verwendet.

- Programme werden in Courier dargestellt und durch ihre speziellen Anfangs- und Endzeichen zusätzlich gekennzeichnet, damit sie im Lauftext besser auffallen.

  Beispiel: ≪ DUP 1 Speicherbedarf: 32.5 Bytes
  FOR E DUP
  NEXT
  ≫

  Oftmals wird auch der Speicherbedarf eines Programms angegeben. Dieser Speicherbedarf wurde mit einem Namen der lediglich aus einem Buchstaben besteht ermittelt, da die Länge des Namens einen Einfluß auf den Speicherverbrauch hat.

2 Grundlagen

2.1 Aufbau und Arbeitsweise eines Rechners

2.1.1 Die Bauteile eines Computers

Ein Computer, PC oder auch nur Taschenrechner, hat immer dieselben Grundelemente. Dies sind

- Eingabeeinheit
- Ausgabeeinheit

und die

- eigentliche Recheneinheit.

Beim Taschenrechner geschieht die Eingabe über die Tastatur oder, seit dem HP48, über eine Schnittstelle zum PC. Auch eine bidirektionale Datenübertragung zwischen zwei HP48 ist nun möglich. Dies geschieht über eine Infrarotschnittstelle. Die Tastatur kann aber als reguläre Eingabeeinheit betrachtet werden.

Die Ausgabe von Daten wird über das LC-Display, eine Infrarotschnittstelle zum Drucker oder, seit dem HP48, auch über ein Kabel zum PC ermöglicht. Zudem können über die Infrarotschnittstelle Daten an einen weiteren HP48 gesendet werden. Die Ausgabe über das LC-Display ist hierbei die Regel.

Der wichtigste Teil des Geräts, der die eigentlichen Rechnungen vornimmt, ist im Taschencomputer eingebaut. Dieser Teil, den der normale Nutzer ja nie zu Gesicht bekommt, besteht wieder aus Untereinheiten, die ihn zu seinen Funktionen befähigen.

Vereinfacht finden wir dort

- das Rechenwerk (die arithmetisch-logische Einheit = ALU; ***a***rithmetic ***l***ogical ***u***nit), das die eigentlichen Berechnungen anstellt,
- das Steuerwerk, welches sämtliche Funktionen koordiniert und kontrolliert

und

- das Speicherwerk, welches als Gedächtnis und Nachschlagewerk benutzt wird. Es ist gewöhnlich in ROM und RAM unterteilt.
  - Das ROM (=***R***ead ***O***nly-***M***emory; nur-Lese-Speicher) ist das Nachschlagewerk des Rechners für alle seine Funktionen, Befehle, Formeln etc.. Dieser Speicher ist lediglich zum Lesen geeignet und kann nicht modifiziert werden. Die Daten dieses Speicherbausteins sind auch nicht löschbar, so daß seine Daten auch nie verloren gehen können.
  - Das RAM (=***R***andom ***A***ccess-***M***emory; Direkt-Zugriffs-Speicher) ist das für den Nutzer frei verfügbare Gedächtnis für Formeln, Variablen, Programme etc.. Dieser Speicher merkt sich seinen Inhalt nur solange, wie die entsprechenden Bausteine eine Stromversorgung haben. Keine Angst, der Taschenrechner versorgt sein Gedächtnis auch, wenn man ihn ausschaltet. Er wirkt also nur wie ausgeschaltet, ist es aber nicht vollständig. Vergleichbar ist dies mit der Bereitschaft eines Fernsehers, der mit einer Fernbedienung ein- und ausgeschaltet wird.

Zur Verdeutlichung der Verschaltung dieser Bausteine soll Abbildung 2–1 dienen:

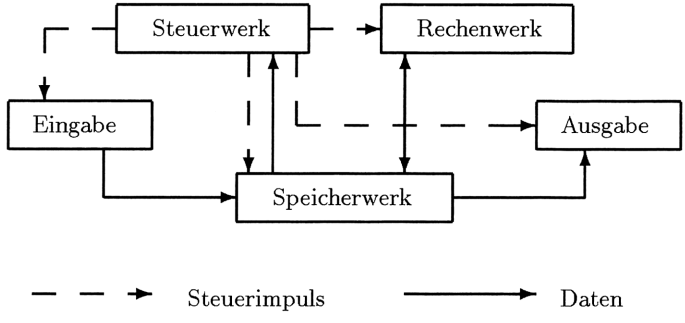


Abb. 2–1: *Aufbau eines Rechners (Prinzipschaltbild)*

Der hardwaretechnische Aufbau soll hier allerdings nicht weiter vertieft werden, da dies zum Erlernen der Softwareentwicklung nichts beiträgt.

Die Funktionen dieser Elemente sollen nun im einzelnen kurz erläutert werden:

- Das Rechenwerk, ALU genannt, nimmt die eigentlichen Berechnungen vor. Hierbei liefert die Eingabe von (i.d.R.) zwei Operanden ein Ergebnis zurück (siehe Abbildung 2–2). Einige Vorgänge benötigen allerdings auch nur einen oder sogar gar keinen Operanden. Bei ersterem könnte als Beispiel das Löschen eines Objekts aus dem Speicher genannt und bei letzterem das Erzeugen einer Zufallszahl angeführt werden.

  Die Operanden werden zuerst im Speicher abgelegt und dann vom Rechenwerk verarbeitet. Das Ergebnis wird nach Abschluß der Berechnung wieder im Stack abgelegt und kann dort zu neuen Berechnungen verwendet werden.

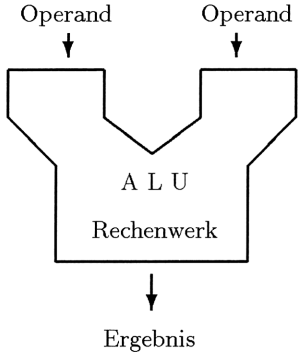


Abb. 2–2: *Funktionsschaltbild der ALU mit zwei Operanden*

- Das Steuerwerk (siehe Abbildung 2–3) koordiniert alle Aktionen des Computers. Es ist dafür zuständig, daß sich immer die richtigen Daten am richtigen Ort zur richtigen Zeit befinden. Zudem steuert es die sukzessive Abarbeitung von Befehlen, für das es über einen Befehlszähler und ein Befehlsregister verfügt. Letzeres enthält immer den gerade auszuführenden Befehl, während der Befehlszähler die Adresse der als nächstes auszuführenden Anweisung bereithält.

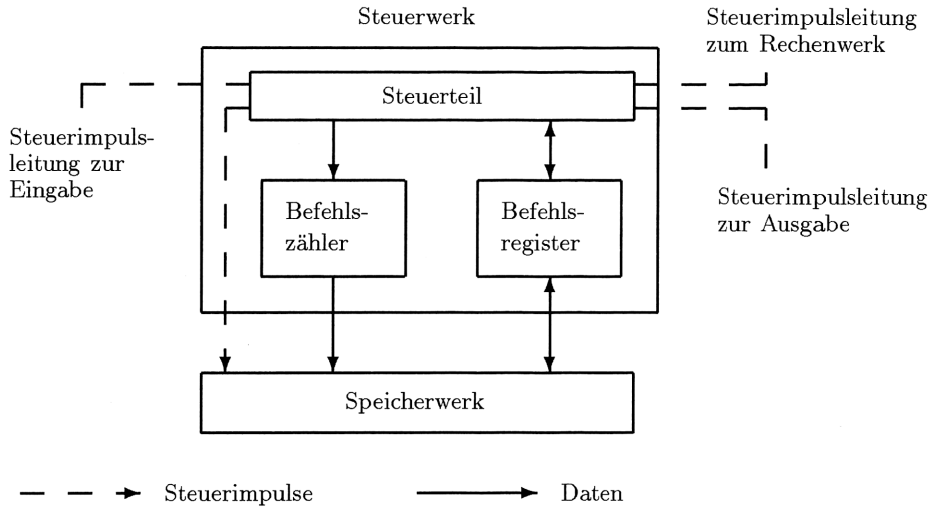


Abb. 2–3: *Prinzipschaltbild zum Steuerwerk*

– Der Speicher ist durchnummeriert, so daß der Computer seine Daten über die Angabe einer sogenannten (Speicher-)Adresse immer wieder finden kann. Auch das Schreiben von Daten geschieht nach diesem Prinzip. Man legt hierzu beim Adreßregister eine Adresse an. Beim Lesen wird der Inhalt der Speicheradresse an das Speicher(daten)register geliefert, beim Schreiben werden die Daten vom Speicherdatenregister an die konkrete Speicheradresse gebracht (siehe Abbildung 2–4).

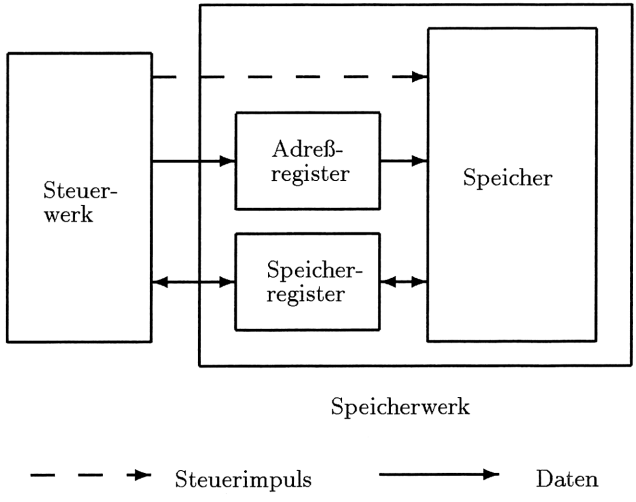


Abb. 2–4: *Funktionsschaltbild zum Speicherwerk*

Eine weitere Aufsplitterung der einzelnen Funktionen der Bausteine eines Computers soll hier nicht vorgenommen werden. Im Rechner sind Steuerwerk und Rechenwerk zu einem Baustein, dem Prozessor (CPU; ***C***entral ***P***rocessing ***U***nit), zusammengefaßt. Der Grund liegt in der sehr verknüpften Wirkungsweise dieser beiden Teile.

2.1.2 Die Darstellung von Daten im Gerät

Betrachten wir nun aber doch einmal, wie ein Computer Daten speichert. Ein Rechner, gleich welcher Art, unterscheidet gerade mal zwei Dinge:

- Strom an (entspricht einer 1 für spätere Zwecke)

und

- Strom aus (entspricht einer 0 für spätere Zwecke)

Dies ist natürlich nicht gerade viel. Es stellt sich also die Frage, wie es der Rechner schafft, mit diesen beschränkten Fähigkeiten solch eine Fülle von Funktionen bereitzustellen. Der Trick ist recht einfach: Man bedient sich der Codierung. Nimmt man zum Beispiel zwei Werte hintereinander, so sieht die Sache schon etwas besser aus. Wir haben nun exakt vier Möglichkeiten bzw. Zustände.

Zustand	1. Wert	2. Wert
0	0	0
1	0	1
2	1	0
3	1	1

Tabelle 2–1: *Tabelle zur Codierung von Zuständen*

Dieses Prinzip führt nun schon auf die Möglichkeit, vier Zustände voneinander trennen zu können. Wir brauchen also nur beliebig viele Zahlen immer zu einem Block (Wort) zusammenfassen, und schon kann der Rechner eine Vielzahl von Funktionen verwalten. Eine einzelne Zahl (0 oder 1) wird als Bit bezeichnet, eine Achtergruppe (acht Bit) als Byte. Bei letzterem stehen dann immerhin 256 Möglichkeiten bereit. Man muß nur Sorge tragen, daß der Anfang der *Wörter* auch immer korrekt gefunden wird. Dies wird durch ein Takten des Prozessors erreicht.

Die Taschenrechner vom HP28 bis HP48G/GX sind 4-Bit-Rechner, die gleichzeitig immer vier Daten auf einmal verarbeiten können. Bei einer Taktfrequenz von 1 MHz bearbeitet ein Taschenrechner also 4 Mio. Bit pro Sekunde.

Wie zählt ein Computer, der nur zwei Zahlen kennt? Mehrere Zahlen werden auch hier einfach hintereinander gestellt. Die Zahlen werden zudem nicht gleich gewichtet,

sondern jeweils mit dem Faktor 2^x multipliziert. An der hintersten Stelle ist x gleich Null, mit jeder weiter links liegenden Position steigt x um Eins an. Die Art der Codierung sieht wie in Tabelle 2–2 gezeigt aus.

...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^x
...	128	64	32	16	8	4	2	1	Wert
...	0	0	0	0	1	1	1	0	Inhalt

Tabelle 2–2: *Codierung einer Zahl*

Dies wäre dann die Zahl 14 im sogenannten Dualsystem (binäres Zahlensystem). Das übliche Zahlensystem ist dezimal, also ein Zehnersystem. Die Zahl 14 aber als binäre Zahl 1110 zu schreiben, benötigt noch ein Kennzeichen, damit sie als Binärzahl erkannt wird. Dies geschieht entweder als $(1110)_2$, oder wie beim Taschenrechner als #1110b. In der Praxis gibt es noch das Oktal- und das Hexadezimalsystem. Hier hat eine Stelle dann acht (von 0 bis 7) bzw. 16 (von 0 bis 9 und von A bis F) Möglichkeiten. Diese Zahlensysteme beherrscht der Rechner ebenfalls.

Zahlen werden allerdings nicht Stelle für Stelle abgespeichert, sondern hier haben sich die Computerentwickler ebenfalls etwas Besonderes einfallen lassen.

Würden wir z.B. die Zahl 3.14159265359 (π) Stelle für Stelle dezimal abspeichern, so bräuchten wir viel zu viel Speicher.

Eine Dezimalzahl beinhaltet zehn Möglichkeiten (0 bis 9). Um diese im Rechner darstellen zu können, brauchen wir 4 Bit.

binäre Zahl	dezimale Zahl
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Tabelle 2–3: *Tabelle zur Zahlendarstellung*

Dabei werden die Zahlen von #0000b bis #1001b benötigt. Die Zahlen von #1010b bis #1111b bleiben allerdings ungenutzt.

Durch Nutzung dieser weiteren Möglichkeiten und einem relativ komplizierten Verfahren kann etwas Speicher gespart werden. Die Methode der (REAL-) Zahlendarstellung zu erklären, bringt jedoch für den Programmierer nichts und soll hier nicht weiter vertieft werden.

Die nächste Frage wäre nun: Wenn der Rechner nur Zahlen kennt – wie schafft er

es dann, Buchstaben zu verarbeiten? Die Antwort lautet: mittels Codierung. Durch einen genormten Code (ASCII), kann man den Computer auch noch dazu bringen, seine „Zahlen“ als Buchstaben zu deuten. Der komplette ASCII-Code ist ganz einfach abrufbar, indem man Zahlen von Null bis 255 in den Stack eingibt und mit dem Befehl CHR das jeweilige Zeichen abruft. Zum Beispiel wird einem A die Zahl 65 zugeordnet, einem Z die 90 usw. Den genormten Teil (Zeichen 0 bis 127) dieses Codes findet man im Anhang auf Seite 269. Damit der Rechner nun Zahlen von Buchstaben unterscheiden kann, wird auch dies verschlüsselt. Zunächst wird der Datentyp festgelegt, und erst dann werden die Daten angeführt.

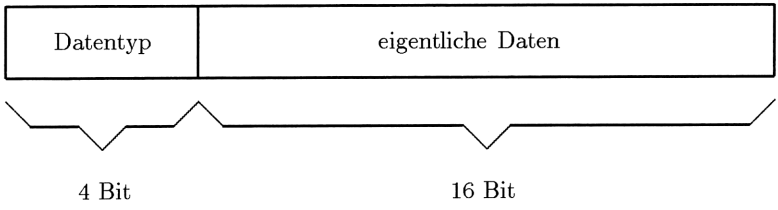


Abb. 2–5: *Codierung von Daten und Datentyp*

Die Länge der eigentlichen Daten ist nicht bei allen Datentypen gleich lang. Dies legt der Taschenrechner anhand des Datentyps fest.

Der Nutzer merkt von all diesen doch recht verwirrenden Vorgängen aber rein gar nichts; und dies ist wohl auch gut so.

Der Taschenrechner kann, wie schon erklärt, nur mit Nullen und Einsen umgehen. Intern werden alle Daten nur als sehr lange Aneinanderreihung von Nullen und Einsen verarbeitet. Da es dem Nutzer aber nicht zuzumuten ist, Dualzahlen einzugeben und abzuschreiben, werden die Daten für die Ein- und Ausgabe (und nur hierfür) konvertiert. Eine Programmiersprache ist selbstverständlich auch nur ein Code für einen Computer, um ihm zukünftige Operationen vorab *beizubringen*. Darauf wollen wir nun endlich unser Augenmerk richten.

2.2 Was ist ein Programm?

Im einfachsten Fall stellt ein Programm eine Abfolge von Anweisungen für Tastenoperationen auf dem Taschenrechner dar. Es wird durch besondere Anfangs- und Endzeichen („≪“- und „≫“-Zeichen) gekennzeichnet, damit der Computer ein Programm auch als solches erkennt. Aufgrund leistungsfähiger Befehle und Strukturen läßt sich der Rechner allerdings weitaus besser nutzen, als obiger Satz dies erwarten läßt.

Da sich (nahezu) jede Funktion, (nahezu) jeder Befehl und jedes schon erstellte Programm (als sogenanntes Unterprogramm) durch Aufruf implementieren läßt, bietet der Taschenrechner unvorstellbare Möglichkeiten.

Das Programm besteht aus einer Abfolge von Funktionen, Operanden und Befehlen, die der Reihe nach ausgeführt werden. Durch besondere Strukturen lassen sich Wiederholungen und Verzweigungen einfach realisieren.

Bei einer Verzweigung wird eine Befehlsfolge nur dann ausgeführt, wenn eine bestimmte Bedingung vorher erfüllt wurde.

Bei Wiederholungsschleifen lassen sich Programmteile, deren Größe frei wählbar ist, beliebig häufig ausführen.

Bei all diesen Strukturen existieren in der Programmiersprache RPL jeweils eine Reihe von Unter-Typen, die den jeweiligen Erfordernissen angepaßt sind.
Bei komplexen Problemen wird es nötig sein, diese Strukturen zu schachteln, d.h. beispielsweise eine Unterverzweigung einzubinden. Auch diese Möglichkeit wird von den Taschenrechnern geboten.

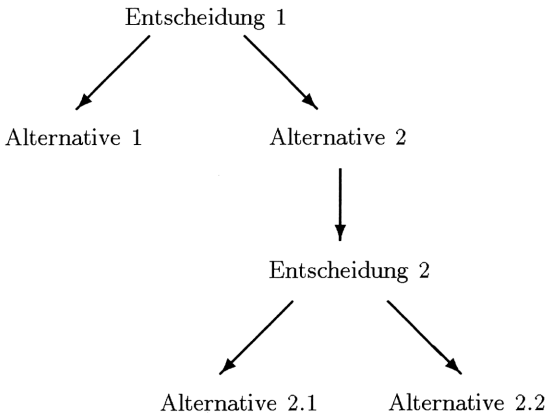


Abb. 2–6: *Unterverzweigungen*

Bei der Programmentwicklung ist es wichtig, ein (komplexes) Problem in immer kleinere Teilprobleme zu zerlegen, bis man auf unterster Ebene nur noch über kleine Bruchstücke aus Anweisungen und einfachen Strukturen verfügt (sogenannte Top-Down-Vorgehensweise). Man entwickelt einen Algorithmus, der das vorliegende Problem vom Ablauf her beschreibt. Nach der Entwicklung des Algorithmus geht man zur Codierung in der eigentlichen Programmiersprache über. Dies geschieht durch die Umsetzung einzelner Algorithmusschritte in die konkreten Befehle der Computersprache. Auch Leser, die bereits Programmiererfahrung haben, werden sich durch die zwangsweise Anwendung des LIFO-Stacks erst an einige Besonderheiten gewöhnen müssen. Aus diesem Grund enthält das Buch aber zahlreiche Problemstellungen und erklärte Musterlösungen, um Übung hierin zu bekommen. Nach dem Umsetzen des

Algorithmus in die rechnerverständliche Sprache sind weitere Schritte nötig. Ist dann die Syntax (Rechtschreibung und Grammatik einer Programmiersprache) überprüft, braucht man sich nur noch mit der Verifikation der Programmfunktion auseinanderzusetzen, was auch später noch die meiste Arbeit und Zeit beansprucht, wenn man in der Programmierung schon recht geübt ist.

2.3 Strukturierte Programmierung

Die Taschenrechner unterstützen strukturierte Programmierung. Ziel dieses Buchs ist

- die Erklärung der Regeln zur Strukturierung des Steuerflusses nach der Top-Down-Vorgehensweise (Damit ist gemeint, daß eine komplexe Problemstellung in immer kleinere Teilprobleme zerlegt wird.)
- die Erklärung der einzelnen Bausteine, die für die Programmierung zur Verfügung stehen

und

- die Übertragung der Strukturen in die Programmiersprache RPL.

Jedes Programm bzw. jede Struktur verfügt über einen fest definierten Anfangs- und Endpunkt. Der HP28 verfügte noch über einen Abbruch-Befehl (ABORT), ist also in diesem Punkt nicht ganz so konsequent gewesen, da dies einen (nicht erlaubten) Seitenausstieg aus einer Struktur ermöglichte.

Die Programme haben keinerlei Adressen, Labels oder Zeilennummern, an die mit Hilfe eines GOTO-Befehls gesprungen werden kann (wie z.B. bei BASIC). Jedes Programm kann als Unterprogramm in einem anderen Programm verwendet werden, d.h. der Name des Programms kann wie ein Befehl des internen Befehlssatzes genutzt werden. Ein Beispiel soll dies verdeutlichen: Man schreibt zunächst ein Programm, welches eine beliebig lange Zeichenkette in einzelne Zeichen zerlegt, und nennt dieses Programm 'ZERLEG'. Nach dem Abspeichern des korrekt funktionierenden Programms kann man nun ZERLEG wie einen Befehl benutzen. Der Rechner würde hierbei die Zeichenkette nach dem Aufruf der Unterroutine zerlegen und anschließend mit dem nächsten Befehl des aufrufenden Programms fortfahren. Dies wäre auch schon das erste Beispiel zur Top-Down-Vorgehensweise. Man hat ein übergeordnetes Problem, stückelte dieses in Teilprobleme und setzt diese als Unterroutinen um.

Auch ein rekursiver Aufruf eines Programms ist beliebig möglich. Hierunter versteht man eine Routine, die sich selbst immer wieder aufruft. Natürlich wird dies

irgendwann durch eine Bedingung unterbrochen, sonst hätten wir ja ein endlos laufendes Programm.

Strukturierte Programmierung ist, einfach ausgedrückt, das Erstellen einer Befehlsabfolge im Baukastenprinzip. Jeder Baustein hat eine genau umrissene Funktion, genau einen Ein- und einen Ausgang und wird immer von oben nach unten abgearbeitet. Ein seitliches „Hereinspringen" oder Abarbeiten in umgekehrter Reihenfolge ist nicht zugelassen. Ebenso ist ein seitliches Aussteigen nicht erlaubt.

Im einzelnen stehen drei grundsätzlich verschiedene Bausteine zur Verfügung. Diese werden als sogenannte Nassi-Shneider-Diagramme dargestellt, da diese am weitesten verbreitet sind.

- Die einfachste Struktur ist ein einzelner Befehl bzw. eine Anweisung (siehe Abbildung 2–7).

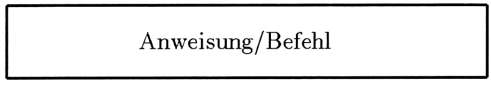


Abb. 2–7: *Struktogramm einer einfachen Anweisung*

Eine Anweisung besteht lediglich aus einem Befehl oder einem sonstigen Objekt. Beim Entwurf eines Struktogramms zu einem konkreten Problem ist es aber durchaus üblich, selbst größere Probleme mit einem Satz zu beschreiben und hierfür nur einen Baustein für eine Anweisung zu verwenden. Als Beispiel soll der Anfang einer Partialbruchzerlegung dienen (siehe Abbildung 2–8).

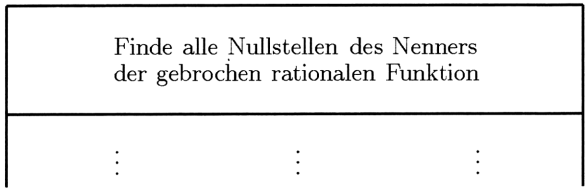


Abb. 2–8: *Kopf eines Struktogramms zur Partialbruchzerlegung*

Bei der Top-Down-Vorgehensweise wird erst später ein weiteres Struktogramm entwickelt, welches Aussage und Vorgehensweise der einzelnen Blöcke aufschlüsselt.

- Bei den Schleifenstrukturen gibt es zwei Untertypen von Strukturen. Bei Abbildung 2–9 wird die Anweisung in jedem Falle mindestens einmal ausgeführt, während bei der Abbildung 2–10 (Schleifenstruktur 2) erst die Schleifenbedingung überprüft wird, bevor die Anweisung überhaupt ausgeführt wird. Schleifenstruktur bezeichnet hier lediglich den Typ der Schleife. Die Wiederholungsstruktur in Abbildung

2–9 wird auch nicht-abweisende Schleife genannt, die andere dementsprechend abweisende Schleife. Hierzu wird allerdings noch auf Seite 73 einiges erläutert werden.

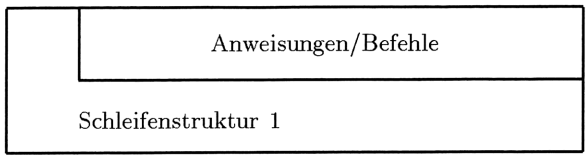


Abb. 2–9: *Schleifenstruktur 1*

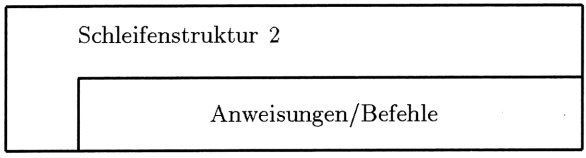


Abb. 2–10: *Schleifenstruktur 2*

– Die dritte Struktur ist die Verzweigung (IF-Struktur; siehe Abbildung 2–11). Hier wird eine Testbedingung überprüft. Ist diese erfüllt bzw. *wahr*, so wird eine THEN-Befehlsfolge ausgeführt. Ist die Bedingung nicht erfüllt, so kann (muß aber nicht) eine als ELSE-Anweisungsfolge bezeichneter Programmblock (Alternativbefehlssequenz) ausgeführt werden. Letztere Struktur wird auch zweiseitige (THEN und ELSE) Alternative (siehe Abbildung 2–12) genannt, während der obere Typ eine einseitige ist. Natürlich ist die einseitige auch nur ein Derivat der zweiseitigen, was die Formulierung *falls Bedingung nicht erfüllt, dann* ***tue nichts*** schon zeigt.

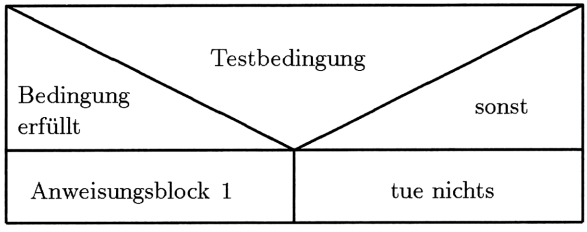


Abb. 2–11: *Einseitige IF-Struktur*

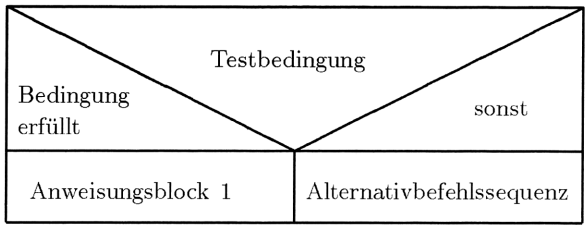


Abb. 2–12: *Zweiseitige IF-Struktur*

Wie man sicherlich schon vermutet, kann eine zweiseitige IF-Struktur in zwei einseitige zerlegt werden. Aus Abbildung 2–13 ganz man ganz einfach Abbildung 2–14 entwickeln.

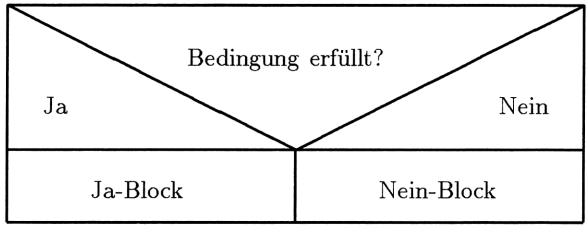


Abb. 2–13: *Zweiseitige Alternative*

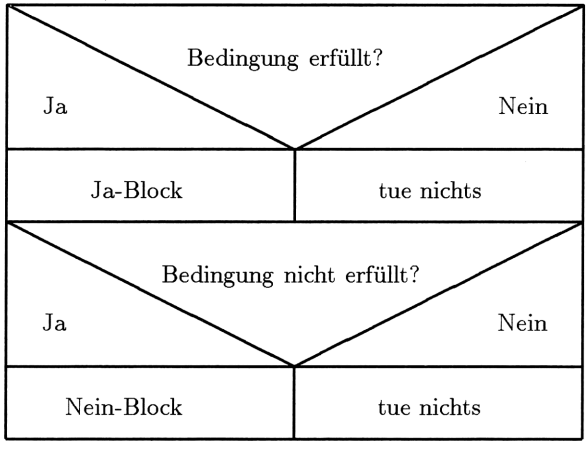


Abb. 2–14: *Zwei einseitige Alternativen*

Es leuchtet ein, daß letzteres aufwendiger in der Programmierung ist, wenn die zweiseitige Struktur zur Verfügung steht.

Aus solchen Bausteinen kann man nun alle seine Programme zusammensetzen. Hierzu ist es natürlich erst einmal nötig, sich mit den Funktionen und Möglichkeiten einer Programmiersprache auseinanderzusetzen.

Als nächstes soll ein Beispiel für ein einfaches Struktogramm gezeigt werden, ohne nun gleich Gedanken über Sinn und Unsinn des Beispiels zu verschwenden.

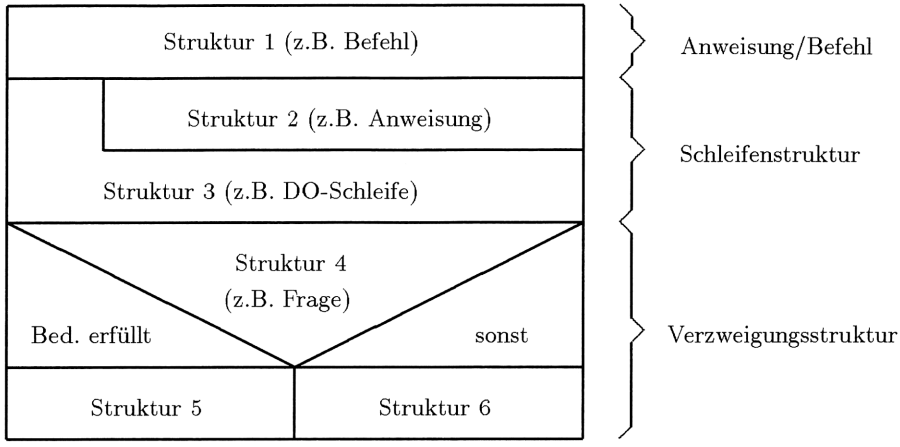


Abb. 2–15: *Beispiel eines Struktogramms*

Der Vorteil der Strukturierung ist eine leicht mögliche Zerlegung des zu programmierenden Problems in kleinere Teile. Zudem ist ein schon gefertigtes Struktogramm recht einfach in beliebige Programmiersprachen übertragbar, da sich die Programme in den einzelnen Sprachen meist nur durch verschiedene Syntax auszeichnen. Der größte Unterschied ist, wie schon erwähnt, die Umstellung auf UPN, mit dem die meisten anderen Programmiersprachen nicht arbeiten. Dies liegt an der Funktion des LIFO-Stacks. Die einzelnen Strukturen sind im Kapitel „Elemente der Programmierung“ (ab Seite 40) nochmals im einzelnen erklärt. Ziel soll es in diesem Buch sein, die Übersetzung eines Problems in die Befehlssequenz der Sprache RPL Schritt für Schritt zu erlernen.

Ein Hinweis auf diese Strukturierung gibt der Rechner auch beim Editieren eines Programms, wobei der Programmcode automatisch mit Einrückungen versehen wird. Hiermit möchte der Taschencomputer die Zusammenhänge der einzelnen Strukturen besser lesbar machen. Gleich weit eingerückte Befehle gehören zusammen. Aufgrund des beschränkten Platzes im Display ist diese Einrückungsfunktion allerdings deutlich eingeschränkt, mit etwas Übung aber doch recht gut erkennbar.

Das Befolgen einiger trivialer Hinweise erleichtert den Umgang mit Struktogrammen:

- Ein Struktogramm sollte übersichtlich sein – nicht verwirrend. Man beschreibe also ruhig größere Probleme mit einem Satz und erläutere diese dann gesondert in einem weiteren Diagramm (Top-Down-Methode).
- Die Bausteine sollen eine verbale Beschreibung enthalten – nicht die Syntax einer Programmiersprache, um eine spätere Übertragung in andere Programmiersprachen möglichst noch offen zu halten.
- Ein Struktogramm sollte im Idealfall eine Abart eines Inhaltsverzeichnisses sein.

2.4 Der Stack

Beim Stack (siehe Abbildung 2–16) handelt es sich um die zentrale Ein- und Ausgabeschnittstelle für Daten. Er besteht aus einer Reihe von Speicherplätzen, die beliebige Objekte aufnehmen können. Ein einzelner Speicherplatz wird als Stackebene bezeichnet. Die Anzahl der Ebenen ist dynamisch, d.h eine Begrenzung der Anzahl von Elementen im Stack ist nur durch den frei verfügbaren Speicher gegeben. Bei einer größeren Anzahl von Elementen im Stack sind zwar immer nur die untersten Objekte zu sehen, die anderen gehen allerdings nie verloren und rücken bei Verarbeitung der unteren Ebenen wieder nach unten nach.

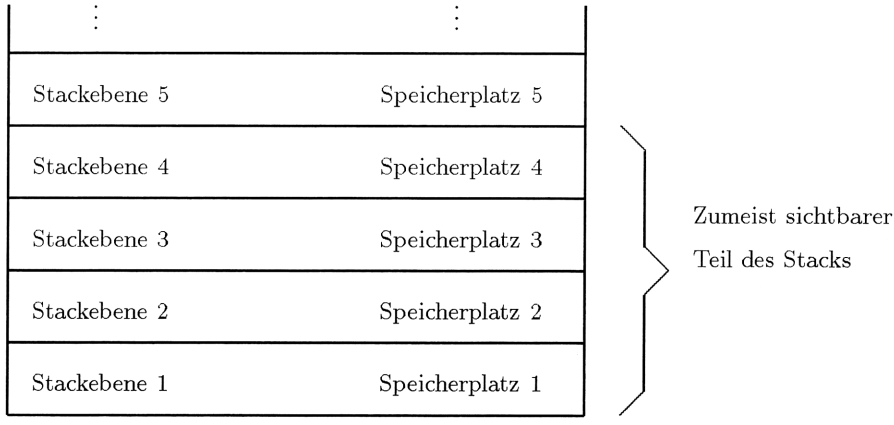


Abb. 2–16: *Darstellung des Stacks*

Der Begriff der unteren Ebenen bezieht sich hier auf das Visuelle, es handelt sich um die Stackebenen mit der kleinsten Nummer.

Die Speichergröße der einzelnen Stackebenen ist ebenfalls nicht begrenzt, es können also beliebig große Objekte in den Stack genommen werden (z.B. Matrizen beliebiger Dimension). Auch hier ist die einzige Begrenzung der noch frei verfügbare (RAM-) Speicher.

Alle Daten werden über diesen Stack verarbeitet. Normalerweise werden immer erst alle Argumente eines Befehls eingegeben und erst anschließend mit der Ausführung des Befehls benutzt. Auch alle Ergebnisse (falls vorhanden) werden erst wieder im Stack abgelegt.

Der Stack ist ein sogenannter LIFO-Stack (***L***ast ***I***n - ***F***irst ***O***ut), d.h., daß die zuletzt eingegeben Daten wieder als erste entnommen bzw. verwendet werden. Vergleichbares Praxisbeispiel ist ein Tellerstapel, bei dem man den zuletzt aufgestapelten Teller ja (i.d.R.) auch wieder als ersten abnimmt. Man sollte sich nicht davon verwirren lassen, daß der Stack *auf dem Kopf* steht. Die Tatsache, daß ein neues Element ganz unten eingefügt und von dort wieder als erstes entnommen wird, ist eine reine Frage der visuellen Darstellung. Die Entwickler bei HP haben sich nun einmal dafür entschieden.

Allein bei einer simplen Addition wird der Stack dreimal benötigt. Zuerst werden die beiden Summanden eingegeben, und nach der Addition steht die Summe in Ebene 1 des Stacks.

Durch den LIFO-Stack ergeben sich einige Besonderheiten, an die man sich erst gewöhnen muß. Da die Rechner auch bei den normalen Rechenoperationen in RPN (***R***everse ***P***olish ***N***otation) bzw. UPN (***u***mgekehrte ***P***olnische ***N***otation) bedient werden, gewöhnt man sich auch bei der Programmierung recht schnell an die *umgedrehte* Eingabe der Argumente.

2.5 RPL im Vergleich zu anderen Sprachen

Der wohl gravierendste Unterschied zwischen den üblichen Programmiersprachen und der als RPL bezeichneten Sprache der HP-Taschenrechner liegt in der zwangsweisen Nutzung des LIFO-Stacks. Nutzer des Rechners, die ihr Gerät *sicher bedienen* können, werden keine Schwierigkeiten haben, sich an das *umgekehrte* Eingeben der Daten zu gewöhnen. Einige andere Programmiersprachen verwenden ebenfalls einen Stack (z.B. Forth [3]).

Die Programmiersprache RPL ist stark an PASCAL angelehnt. Viele Befehle bzw. Strukturen wurden direkt oder mit leichten Abwandlungen von dieser oder anderen

Programmiersprachen übernommen, mit der Ausnahme, daß die Argumente aufgrund des RPN vor dem Befehl stehen. Ein Vorgriff auf eine Programmierstruktur soll den Unterschied kurz zeigen:

In PASCAL könnte eine Wiederholungsschleife wie folgt aussehen:

```
FOR v := 1 TO 10 DO
  BEGIN
    Anweisungsblock
  END
```

Die BASIC-Version wäre hierfür:

```
FOR v = 1 TO 10
  Anweisungsblock
NEXT
```

In RPL wird dies dann folgendermaßen geschrieben:

```
≪ 1 10
  FOR v Anweisungsblock
  NEXT
≫
```

Leser, die PASCAL nicht beherrschen, brauchen sich also nicht beunruhigen. Sofern man über Kenntnisse in BASIC oder FORTRAN verfügt, wird man sicherlich nie vor unlösbaren Rätseln stehen. Sehr von Vorteil sind Kenntnisse in irgendeiner Programmiersprache, da es hierbei leichter fällt, sich in Strukturen und Syntax einzufinden.

3 Die verschiedenen Datentypen

3.1 Vorstellung der Datentypen

Nun haben wir uns das nötigste Basiswissen angeeignet. Bevor wir aber in die Programmierung einsteigen können, sollten wir erst einmal betrachten, welche sogenannten Datentypen der Taschenrechner kennt und, noch wichtiger, wie er sie unterscheidet. Der Rechner kennt eine ganze Anzahl von Datentypen (TYPEs). Die wichtigsten werden in Tabelle 3–1 vorgestellt.

Datentyp	Beispiel
Integer-Zahlen	2
REAL-Zahlen	1.5
komplexe Zahlen	'1+2*i' bzw. (1,2)
binäre Zahlen	#Ah
Zeichenketten	"HALLO"
Vektoren	[1 2 3]
Matrizen	[[1 2][3 4]]
Listen	{ "HALLO" 2 1.5 }
Namen	'PROGRAMMNAME'
Programme	≪ DUP + 2 * ≫
algebraische Ausdrücke	'2*A+B'

Tabelle 3–1: *Übersicht über verschiedene Datentypen*

Je nach Operation sind viele Datentypen miteinander verknüpfbar. Anders als bei anderen Rechnern bzw. Programmiersprachen muß man sich keinerlei Gedanken über die Typanpassung machen. In den meisten Programmiersprachen ist im Programmkopf eine Deklaration der Variablen vorzunehmen, um den Rechner mitzuteilen, wieviel Speicher er für die Variablen bereitstellen soll. Der Taschenrechner nimmt diese Typanpassung automatisch vor. Ebenfalls braucht eine Deklaration der Variablen nie vorgenommen zu werden. Der Rechner erkennt den Datentyp an den verschiedenen Begrenzungszeichen (z.B. [ ] für Vektoren, { } für Listen, # für Binär-Zahlen etc.).

Dies stellt in der Praxis eine ganz erhebliche Entlastung des Programmierers von recht stupiden Aufgaben dar.

Ein simples Beispiel sind Operationen mit den Integer-, REAL- und komplexen Zahlen. Addiert man beispielsweise eine Integer-Zahl (z.B. 8) und eine REAL-Zahl (z.B. 1.5), so ist das Ergebnis (hier: 9.5) ebenfalls eine REAL-Zahl. Um die Operation auszuführen, braucht man also keine manuelle Typanpassung vornehmen. Die Addition einer komplexen Zahl mit einer REAL-Zahl ist wieder eine komplexe Zahl etc. etc.

Eine Besonderheit stellen hierbei die Integer-Zahlen dar, da der Taschenrechner nur ganze Zahlen von -9 bis $+9$ als Integer-Zahlen erkennt. So hat also die Addition der Integer-Zahlen 5 und 6 als Ergebnis die REAL-Zahl 11. Der Nutzer merkt hiervon allerdings nichts. Welche Auswirkungen sich daraus ergeben, ist im Kapitel 6 „Tips und Tricks“ (ab Seite 115) näher erläutert.

Bei manchen Datentypen gibt es allerdings ein paar verblüffende Reaktionen. Addiert man zwei Listen, so werden diese aneinandergehängt. Dies führt zu einer weiteren Besonderheit. Wenn man durch die Operation Plus eine Liste L1 in eine Liste L2 bringen möchte, so muß man L1 zuvor in eine umschließende Liste (durch 1 →LIST) bringen. Man möchte beispielsweise { 1 { 2 } } erreichen. Im Stack stehe

```
2: { 1 }
1: { 2 }
```

Das Plus liefert { 1 2 }, während durch 1 →LIST + das gewünschte { 1 { 2 } } entsteht.
Werden ein beliebiges Objekt und eine Liste addiert, so wird die Liste am Anfang um das Objekt ergänzt. Ist die Reihenfolge umgekehrt, so wird das Element angehängt.

Beispiel 1: 0 [ENTER] { 2 "HALLO" } [ENTER] [+] führt zu { 0 2 "HALLO" }

Beispiel 2: { 2 "HALLO" } [ENTER] 0 [ENTER] [+] ergibt { 2 "HALLO" 0 }

Beim HP48 geht auch: "HALLO" [ENTER] 2 [ENTER] [+], was "HALLO2" liefert.

Dies sind die wichtigsten Operationen, die man nicht automatisch erwartet. Während man noch die Verknüpfung zweier Listen durch [+] erwartet hätte, sind die weiteren Operationen unüblich, aber sehr nützlich. Generell läßt sich folgende Aussage treffen:

Gleichartige Datentypen lassen sich immer verknüpfen, fremdartige eher selten.

Zentrale Bedeutung hat hierbei die Verknüpfung [+]. Die Operation [*] (als Gegenbeispiel) ist nur bei Zahlen (gleich welcher Art), algebraischen Ausdrücken, Vektoren und Matrizen einsetzbar. Multipliziert man zwei Namen, so entsteht ein algebraischer Ausdruck, der in der Regel allerdings recht wenig Sinn macht. Bei allen anderen Datentypen ist ein „*-Error“ und die Meldung „Bad Argument Type“ unvermeidlich, mit der der Rechner sein Mißfallen über die Operation kundtut.

Zu diesem Thema kann nur geraten werden, selbst einmal einige Dinge auszuprobieren, da es sicherlich noch Verknüpfungen gibt, die unerwarteterweise bestimmte Reaktionen zeigen, die auch ich nicht kenne. Insbesondere gilt dies für den HP48, der sicherlich noch einige Überraschungen bereithält.

Über eine solche bin ich selbst einmal gestolpert, als ich meine Programmsammlung für den HP48 umschrieb. Während beim HP28 { 1 } EVAL keine Reaktion zeigt, staunte ich beim HP48 über das Verschwinden der Liste zu einer einfachen 1 im Stack.

3.2 Globale und lokale Variablen sowie Objekte des Stack

Nachdem wir nun alle Datentypen kennen, wenden wir uns den Variablen zu. Variablen sind Objekte, denen ein Name zugeordnet wurde, mit dem man das Objekt ansprechen kann. Der Taschenrechner unterscheidet zwischen globalen und lokalen Variablen, die zudem nicht mit den Objekten im Stack verwechselt werden dürfen.

Globale Variablen sind Objekte (eines beliebigen Datentyps), die nicht nur unter einem Namen abgespeichert sind, sondern auch außerhalb von Programmen existieren. Sie sind, bis zum Löschen, permanent vorhanden. Dies gilt im übrigen auch dann, wenn der Rechner abgeschaltet wurde. Im Gegensatz zu PCs, Workstations etc. verfügen die Taschencomputer über einen permanenten Speicher. Im übertragenen Sinne legt sich der Rechner lediglich Schlafen, schaltet sein Display ab und wartet auf sein Wiedererwecken, bei dem er sich an alle Daten wieder erinnert, solange die Batterien noch ausreichend Ladung hatten.

Die Operation 1.5 ENTER 'X' STO speichert die Zahl 1.5 in die globale Variable mit Namen X. Diese wird am Anfang des aktuellen Verzeichnisses unter ihrem Namen (hier 'X') gespeichert und ist bis zu ihrem Löschen (hier 'X' PURGE) über ihren Namen X, 'X' EVAL oder die entsprechende Menüleistentaste abrufbar.

Ein Überschreiben des alten Inhalts geschieht mit der Abspeicherung neuer Daten in der Variablen automatisch (z.B. "Zeichenkette" 'X' STO). Auch ein Wechsel der Datentypen ist hierbei erlaubt. Dies birgt auf der anderen Seite die latente Gefahr, wichtige Daten zu verlieren, wenn man allzu sorglos mit dem Speichern von Elementen in Variablen umgeht.

Lokale Variablen sind Objekte (eines beliebigen Datentyps), die nur in einem Programm (oder Teil eines Programms) unter einem bestimmten Namen abgespeichert sind. Diese Variablen werden mit dem Pfeil → abgespeichert. Hierbei gibt es zwei

Möglichkeiten:

- die Speicherung in lokalen Variablen und deren direkte Verwendung in einer Formel (Beispiel 1) und
- die Verwendung der lokalen Variablen in einem Teil- bzw. Unterprogramm (Beispiel 2)

Dies soll kurz in der Praxis erläutert werden. Man möchte zwei Zahlen voneinander subtrahieren und anschließend hoch drei nehmen.

Die zu programmierende Formel ist: $(x - y)^3$

Beispiel 1: `≪→ x y '(x-y)^3' ≫`

Beispiel 2: `≪→ x y ≪ x y - 3 ^ ≫ ≫`

Zum Programmieren einfacher Probleme ist Beispiel 1 sicherlich etwas einfacher zu handhaben. Deutlich aufwendiger (und auch langsamer) sind die globalen Variablen. Dies ist bei der Ausführung von Beispiel 3 auch gut zu sehen.

Beispiel 3: `≪ 'Y' STO 'X' STO X Y - 3 ^ ≫`

Hier muß angemerkt werden, daß nach Ausführung von Beispiel 3 die Variablen auch nach der Programmausführung gespeichert bleiben. Nicht verwirren sollte die Reihenfolge des Speicherns. Hier wird Y als erstes abgespeichert, da die untere Zahl Y, die obere im Stack (Ebene 2) X ist. Dies kann man am Ergebnis leicht verifizieren. Bei den lokalen Variablen werden alle Objekte sozusagen in einem Schwung vom Stack genommen. Deshalb ist hier der zuletzt genannte Variablenname, der Name, unter dem das unterste Element (Ebene 1) abgespeichert wird.

Auch der Programmieranfänger sollte für seine ersten Programme die Methode von Beispiel 2 wählen. Nur bei kurzzeitig und kurzfristig benötigten Formeln empfiehlt sich das Vorgehen wie in Beispiel 1. (Näheres hierzu auch bei den „Tips und Tricks“ ab Seite 115).

Die professionellste Programmierweise ist allerdings das Rechnen mit den Objekten im Stack. Bei einem simplen Beispiel wie diesem ist dies gut zu überschauen und auch für Laien bald anwendbar. Das Programm sieht bei der ausschließlichen Benutzung von Stackbefehlen so aus:

`≪ - 3 ^ ≫`

Man erkennt sofort, daß dies die kürzeste (und damit schnellste) Version ist.
Das Rechnen ohne jegliche globalen oder lokalen Variablen ist sehr kompliziert, wenn einzelne Objekte mehrfach benötigt werden. Die beiden (lokalen und globalen) Variablentypen lassen sich nicht immer vermeiden. Es ist auch nicht immer sinnvoll. Der Nachteil der beiden liegt im höheren Speicherbedarf und der geringeren Verarbeitungsgeschwindigkeit. Dies sollte auch jetzt schon im Hinterkopf behalten werden.

Eine Anzahl von Übungsaufgaben dient zum Erlernen des Rechnens mit Elementen im Stack.

4 Eingabe und Fehlerbehandlung

4.1 Eingabe und Editierung von Programmen

Da dieser Punkt in den Benutzerhandbüchern recht gut beschrieben ist, kann ich mich hier relativ kurz fassen.

Jedes Programm fängt mit dem ≪ an und endet mit dem ≫. Nach dem Anfangszeichen tippt man die Befehle, Zahlen, Felder etc. der Reihe nach ein, die immer durch ein Leerzeichen voneinander getrennt werden müssen. Um die Länge der Zeile braucht man sich keine Gedanken zu machen, da der Taschenrechner selbsttätig Zeilenumbrüche einfügt. Eine besondere Funktion ist das Einrücken für die bessere Übersicht der Strukturierung. So wird zum Beispiel die Eingabe von

```
≪ X SQ IF X 0 ≠ THEN X INV END ≫
```

beim Wiederaufruf (Editierung) als

```
≪ X SQ
  IF X 0 ≠
  THEN X INV
  END
≫
```

wiedergegeben, was etwas übersichtlicher ist. Der HP28 und der HP48 unterscheiden sich bei den Zeilenumbrüchen etwas. Die Zeile eines HP48 ist aufgrund des größeren Zeichensatzes um ein Zeichen kürzer, was aber ansonsten keinerlei Auswirkungen hat. Die Darstellung in diesem Buch bezieht sich auf den Zeilenumbruch des HP48.

Wie man an einzelne Sonderzeichen herankommt, steht in den Benutzerhandbüchern bei der Auflistung aller Befehle, Funktionen und Tastenbelegungen (beim HP48G ist dies der Anhang G).

Zum Übernehmen eines Programms in den Stack (um es abspeichern oder ausführen zu können) drückt man ENTER. Im schlimmsten Fall führt ein sogenannter Syntax-Error dazu, daß dies nicht geschieht. Ein Syntax-Error ist eine Art Grammatik- oder Rechtschreibfehler einer Programmiersprache. Häufigste Fehlerquelle ist hierbei aus eigener Erfahrung eine nicht geschlossene (FOR-NEXT-) Schleife. Der Rechner versucht zu helfen, indem er mit dem Cursor an die Stelle springt, an der er den Fehler vermutet. Diese Funktion arbeitet aber nicht immer zuverlässig, da er bei Schleifen ja nicht wissen kann, wo sie enden sollen. Falsch geschriebene Struktur-Befehle kann er nicht als fehlerhaft erkennen, da er als Unterprogrammnamen nahezu beliebige Namen zuläßt. So ist END ein reservierter Befehl z.B. für das Ende einer Verzweigung. Schreibt man statt dessen ENDE, so interpretiert der Rechner dies als Unterprogramm mit Namen ENDE, und folglich ist für ihn die Verzweigungsstruktur noch nicht geschlossen.

Die korrekte Funktion lautet also:

```
≪
  IF X 0 ==
  THEN DUP
  END
≫
```

Fehlerhaft sieht es wie folgt aus:

```
≪
  IF X 0 ==
  THEN DUP
  ENDE
≫
```

In diesem Falle empfiehlt es sich, den Ausdruck, den der Rechner moniert, zu betrachten. Zeigt er auf ein ≫-Zeichen, so moniert er im vorgehenden Programmblock eine nicht geschlossene Struktur. Erscheint der monierte Ausdruck fehlerfrei oder ist die bemängelte Stelle ein ≫-Zeichen, so sollte man die FOR/START- und NEXT/STEP-Befehle überprüfen. Sind auch diese korrekt, so empfiehlt sich die Kontrolle aller Verzweigungsstrukturen (also alle IF-THEN-(ELSE)-END-Befehle).

Ebenfalls ein Fehler, der häufig auftritt, ist das Vergessen des ≪-Zeichens, nachdem man lokale Variablen vom Stack nahm. Hierbei gilt folgende Regel:

Lokale Variablen erfordern immer dann die Eröffnung eines Teilprogramms, wenn die lokalen Variablen nicht sofort in einem einzigen algebraischen Ausdruck eingesetzt werden sollen.

Erlaubt ist also:

```
≪→ x y 'SQ(x*y)' ≫
≪→ x y ≪ x y * SQ ≫ ≫
```

Nicht erlaubt ist hingegen: ≪→ x x y − ≫

Beim letzten Beispiel kann der Rechner nicht mehr unterscheiden, wann die Aufzählung der lokalen Variablen aufhört und das Programm weitergeht. Hin und wieder ist man mit der Schachtelung der Teilprogramme (Programm in einem Programm) durcheinander geraten. Hier sind die Fehler nur schwer zu finden, wenn man vorher kein Struktogramm angefertigt hat und man anhand dessen nicht zurückverfolgen kann, an welcher Stelle die Strukturen enden sollten.

Bei Verzweigungsstrukturen (IF-THEN-(ELSE)-END) wird automatisch für jedes im Programm vergessene END ein END am Ende des Programms hinzugefügt. Dies gilt nur, wenn am Ende des Programms sämtliche ≫-Zeichen fehlen. Ebenso werden nicht vorhandene ≫-Zeichen selbsttätig ergänzt.

Wenn das Programm frei von Syntax-Fehlern ist, so wird es in den Stack übernommen bzw. wieder unter seinem Namen abgespeichert. Letzteres geschieht beim Beenden des Editierens eines Programms, während beim ersten Eintippen das erstellte Programm erst einmal im Stack abgelegt wird. Um ein Programm, das im Stack abgelegt ist, zu speichern, muß man seinen Namen in '-Zeichen eingeben (z.B. 'NAME') und [STO] drücken. Es wird dann am Anfang des aktuellen Verzeichnisses unter seinem Namen abgespeichert und ist nun auch über die Menüleiste aufrufbar. Gab es schon ein Objekt mit diesem Namen, so wird dieses nun überschrieben. Die alten Daten gehen hierbei verloren, ohne daß der Rechner noch eine Warnung ausgibt. Man ist also auf die eigene Sorgfalt angewiesen.

Ein Programm wird durch Aufruf seines Namens ohne '-Anführungszeichen, Eingeben seines Namens in '-Anführungszeichen und Ausführung von [EVAL] oder Drücken der entsprechenden Menüleistentaste gestartet. Ist das Programm lediglich in der untersten Ebene des Stacks abgelegt, so genügt für den Start das Drücken von [EVAL].

Es folgt ein Hinweis, um sich beim Eintippen von Programmen etwas Zeit und Arbeit zu sparen:

1. Trick: Beim HP28 kann man das Space-Zeichen durch ein anderes Zeichen ersetzen, da der HP28 ein weiteres Zeichen als Trennungszeichen zuläßt. Das Zeichen hängt davon ab, ob der Dezimalpunkt als Trennungszeichen für Vor- und Nachkommazahlen gewählt wurde. Ist dies der Fall, so kann das Komma als Leerzeichen-Ersatz benutzt werden - ansonsten ist dies der Punkt. Dieses zweite Trennungszeichen kennt nur der HP28.

2. Trick: Oftmals benötigt man Befehle, die entweder langwierig über die Tastatur Buchstabe für Buchstabe eingetippt werden müssen oder nur sehr aufwendig über diskrete Verzeichnisse als Befehl abrufbar sind. In gewissem Rahmen gibt es hier eine Hilfe. Man schreibt sich eine Liste mit den wichtigsten (dafür geeigneten) Befehlen und eröffnet ein TMENU (beim HP28 ein MENU). Nun kann auf ganz einfache Weise (ein Tastendruck) der gewünschte Befehl aufgerufen werden. Auch die obligatorischen Leerzeichen, die die Befehle trennen, werden automatisch eingefügt. (Eine Liste hierzu ist bei den Tips auf Seite 121 angegeben.)

4.2 Fehlersuche im Programm

In der Regel liefert ein Programm, das man soeben geschrieben hat, nicht sofort die gewünschten Ergebnisse. Dies liegt nicht daran, daß der Rechner nicht versteht, was man meint, sondern daß das Programm schlichtweg falsch ist. Deshalb sollte man sich auf keinen Fall entmutigen lassen, wenn ein Programm erst nach dem zehnten Versuch richtig läuft. Fatal wäre auch der Glaube, bei den Übungsaufgaben jedes Mal die optimale Version der Musterlösung erstellen zu können. Wenn das Programm seinen Zweck erfüllt, ist seine wichtigste Aufgabe erledigt.

Da sich insbesondere bei komplexen Programmen die Fehlersuche sehr aufwendig gestaltet, bietet der Rechner die Möglichkeit der Einzelschrittausführung, so daß man jeden einzelnen Schritt des Programmablaufs verfolgen kann. Die allgemeinste Art, sich auf die Fehlersuche zu begeben, ist das Einfügen des HALT-Befehls an einer Stelle, ab der die Einzelschrittausführung beginnen soll (i.d.R. am Anfang des Programms). Der HP48 verfügt direkt über diese Funktion des Debuggens. Dies ist im Benutzerhandbuch auch gut beschrieben.

Nach dem Starten des Programms stoppt die Ausführung beim ersten HALT-Befehl, der HALT-Indikator erscheint im Statusbereich des Rechners (oberste Zeile des LC-Displays). Nun kann man mit [SST] (für Single STep) das Programm Schritt für Schritt verfolgen. Die Menüleisten-Taste für [SST] befindet sich beim HP28 unter (*PROGRAM*)[CONTRL] und beim HP48 unter [PRG] [CTRL]. Das Eintippen des Befehls SST zeigt im übrigen nicht die gewünschte Reaktion. Der HP48 kann zusätzlich Unterprogramme mit [SST↓] direkt in Einzelschrittausführung testen, oder aber auch in einem Schritt mit [SST] ausführen. Der HP28 springt generell immer über

Unterprogramme. Wenn man beim HP28 ein Unterprogramm testen möchte, so ist dort separat ein HALT einzufügen.

Editiert wird ein Programm beim HP28 und HP48S/SX mit [VISIT] und beim HP48G/GX mit [EDIT]. Zuvor muß man noch den Namen des zu editierenden Programms in den Stack eingeben. Beim Editieren gibt es insbesondere zwei Modi. Dies ist einerseits der Replace- (Ersetzungs-) und der Insert- (Einsetz-) Modus. Erkennbar sind die Modi jeweils am Cursor, der im Replace-Modus rechteckig und im Insert-Modus ein nach links zeigender Pfeil ist. Selbstverständlich kann man auch einzelne Zeichen löschen. Zusätzlich möglich sind noch Optionen wie das Löschen eines Teils einer Zeile, das Springen an den Anfang oder das Ende eines Programms etc.

Hat man die fehlerhafte Stelle des Programms gefunden, so muß man das Programm editieren und neu starten. Das Editieren startet man durch Eintippen des Namens in '-Zeichen (z.B. 'NAME') und Ausführung von [EDIT] bzw. [VISIT]. Hinzufügung neuer und Löschen von überflüssigen bzw. falschen Anweisungen und Befehlen stellt das eigentliche Editieren dar.

Der Rechner merkt sich in jedem Fall aber noch die letzte HALT-Stelle der alten Einzelschrittausführung der alten Programmversion. Erkennbar ist dies am sogenannten HALT-Indikator (HALT in der Statuszeile des HP48 oder ein ○-Zeichen beim HP28). Das Löschen und Abbrechen der Einzelschrittausführung wird durch den Befehl KILL vorgenommen. Möchte man die Einzelschrittausführung bei der Programmausführung beenden, das Programm aber zu Ende rechnen lassen, so ist [CONT] zu drücken.

5 Elemente der Programmierung

5.1 Die wichtigsten Stackbefehle

5.1.1 Verschieben und Kopieren mit Stackbefehlen

Steigen wir nun in die eigentliche Programmierung ein. Zuerst werden die wichtigsten Befehle zum Verschieben und Kopieren von Objekten des Stacks erklärt und anschliessend die wichtigsten mathematischen Anweisungen.

Mit Hilfe dieser Befehle ist man in der Lage, die ersten Programme zu schreiben. Um dies auch einzuüben, befindet sich im Anschluß daran ein Kapitel mit Aufgaben (und Musterlösungen), mit deren Hilfe zu einer gewissen Sicherheit im Umgang mit Objekten des Stacks und der Anwendung der Befehle gelangt werden soll.

Erst im Anschluß daran folgen die etwas weniger wichtigen Befehle der Programmiersprache RPL. Nachdem man einige Kenntnisse über die einzelnen Anweisungen gewonnen hat, gehe ich zur Erklärung der restlichen Strukturen über, die jeweils auch mit Übungsaufgaben vertieft werden sollen.

Der Inhalt eines Programms besteht aus einer Abfolge von Anweisungen. Die Objekte veranlassen je nach Typ verschiedene Reaktionen des Rechners [1]. Folgende Objekte werden einfach nur in den Stack übernommen:

- Integer-Zahlen: ganzzahlig von -9 bis $+9$
- REAL-Zahlen: reelle Zahlen, die nicht ganzzahlig von -9 bis $+9$ sind
- komplexe Zahlen: Zahlen mit imaginärem Anteil
- Strings: Zeichenketten
- Binärzahlen
- Vektoren
- Matrizen

- Listen
- lokale Variablen in '-Anführungszeichen

und

- globale Variablen in '-Anführungszeichen

Befehle werden direkt ausgeführt.

Bei globalen und lokalen Namen ohne '-Anführungszeichen wird der Inhalt in den Stack geschrieben bzw. ausgeführt. Sind diese Namen in '-Anführungszeichen geschrieben, so wird der Name in den Stack übernommen.

Gehen wir nun zu den Basis-Befehlen über. Prinzipiell kann man mit nur vier Befehlen für alle Stackoperationen der Verschiebung und Duplizierung auskommen, wobei einer dieser vier auch keine besondere Bedeutung hat. Die Befehle DUP, DROP und ROLL sind somit die wichtigsten. DEPTH wird seltener gebraucht.

DUP kopiert das Objekt der (Stack-) Ebene 1 in die Ebene 2. Alle weiter oben liegenden Elemente werden um eine Ebene nach oben geschoben.

Beispiel:

3:	
2:	'B'
1:	'A'

DUP $\Longrightarrow$

3:	'B'
2:	'A'
1:	'A'

DROP löscht das Element der Ebene 1. Alle anderen Objekte rutschen eine Ebene nach unten.

Beispiel:

2:	'A'
1:	'B'

DROP $\Longrightarrow$

2:	
1:	'A'

X **ROLL** erlaubt das zirkulare Wechseln von X Stackebenen, bei der jede Ebene um eine verschoben wird. Hierzu muß mit dem Befehl die Anzahl (hier X) der zu *rollenden* Ebenen angegeben werden.

Beispiel:

5:	'E'
4:	'D'
3:	'C'
2:	'B'
1:	'A'

3 ROLL $\Longrightarrow$

5:	'E'
4:	'D'
3:	'B'
2:	'A'
1:	'C'

(Man kann gut erkennen, daß nur die unteren drei Stackebenen *gerollt* wurden.)

DEPTH gibt die Anzahl der Elemente im Stack zurück.

Beispiel:

3:	
2:	'A'
1:	'B'

DEPTH $\Longrightarrow$

3:	'A'
2:	'B'
1:	2

Alle anderen Stackbefehle sind deshalb noch lange nicht redundant. Sie erleichtern die Programmierarbeit und reduzieren die Länge der Programme ganz erheblich. Die Befehle sind deshalb nachfolgend ebenfalls einzeln erläutert.

X PICK kopiert das Element der Stackebene X in Ebene 1, ohne die Reihenfolge der anderen Elemente im Stack zu verändern. Die Elemente des Stacks sind also um eine Ebene nach oben gerutscht, um Platz für das duplizierte Objekt der Ebene X zu machen.

Beispiel:

4:	
3:	'A'
2:	'B'
1:	'C'

3 PICK $\Longrightarrow$

4:	'A'
3:	'B'
2:	'C'
1:	'A'

OVER ist eine Kurzform für 2 PICK. Dieser Befehl erweist sich in der Praxis als sehr nützlich. Wenn ein Objekt des Stacks mehr als einmal benötigt wird, ist dieser Befehl i.d.R. die eleganteste Kopiermöglichkeit.

Beispiel:

3:	
2:	'A'
1:	'B'

OVER $\Longrightarrow$

3:	'A'
2:	'B'
1:	'A'

DUP2 dupliziert die Objekte der Ebenen 1 und 2 gleichzeitig.

Beispiel:

4:	
3:	
2:	'A'
1:	'B'

DUP2 $\Longrightarrow$

4:	'A'
3:	'B'
2:	'A'
1:	'B'

DROP2 löscht die Elemente der Stackebenen 1 und 2 gleichzeitig.

Beispiel:

3:	'A'
2:	'B'
1:	'C'

DROP2 $\Longrightarrow$

3:	
2:	
1:	'A'

ROT ist eine Kurzform von 3 ROLL.

Beispiel:

4:	'A'
3:	'B'
2:	'C'
1:	'D'

ROT $\Longrightarrow$

4:	'A'
3:	'C'
2:	'D'
1:	'B'

SWAP ist eine Kurzform von 2 ROLL. Es vertauscht die beiden untersten Ebenen.

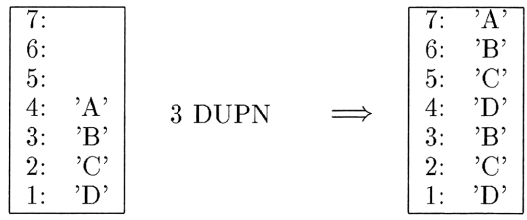
Beispiel:

X **ROLLD** ist das Gegenstück zu ROLL. Hier ist die Richtung des Zirkulierens der des Befehles ROLL entgegengesetzt.

Beispiel:

X **DUPN** dupliziert X Elemente des Stacks auf einmal (Ebenen von 1 bis X).

Beispiel:



X **DROPN** löscht X Elemente des Stacks auf einmal (Ebenen von 1 bis X).

Beispiel:

CLEAR löscht alle Elemente aus dem Stack.

Beispiel:

Man erkennt leicht, daß sich die meisten Stackbefehle auf die ersten zwei, maximal drei, Ebenen beziehen. Dies bedeutet allerdings auch, daß am elegantesten programmiert werden kann, wenn man weitgehend mit den untersten drei Ebenen arbeitet. Alle Daten, die bei der Programmierung in allernächster Zeit gebraucht werden, sollten sich, falls möglich, dort befinden. Objekte, die erst etwas später gebraucht werden, rutschen durch das Verarbeiten von allein nach unten.

Dieser Tip sollte möglichst oft umgesetzt werden. Auf keinen Fall sollte man die Objekte sortieren und dann erst verarbeiten. Ideal ist es, nur die gerade benötigten Elemente in die richtige Reihenfolge zur Abarbeitung zu bringen, diese zu verarbeiten und dann erst fortzufahren. Es empfiehlt sich, erst einmal Überlegungen darüber anzustellen, mit welcher Programmversion ein möglichst reibungsloses Abarbeiten des

Problems möglich ist. Dies erfordert allerdings recht viel Erfahrung und Übung, um zu guten Ergebnissen zu kommen. Es wird aber in den Übungsaufgaben auch noch oft geübt.

5.1.2 Mathematische Stackbefehle

Zudem gibt es noch eine ganze Anzahl von mathematischen Befehlen, die ich nicht alle einzeln erklären muß. Bei den meisten ist der Sinn sofort verständlich (z.B. SIN, COS, *, /, +, − etc.). Trotzdem sind da noch etliche, die bei Programmen oft benötigt werden und die nicht selbstverständlich sind bzw. eine von anderen Programmiersprachen abweichende Notation besitzen. Diese sollen hier aufgeführt werden.

COLCT	faßt einen algebraischen Ausdruck zusammen, soweit dies möglich ist. Beispiel: 'A+B+2*A' COLCT $\Longrightarrow$ '3*A+B'
EXPAN	multipliziert einen algebraischen Ausdruck aus, sofern dies möglich ist. Beispiel: '(A+B)*(C+A)' EXPAN $\Longrightarrow$ '(A+B)*C+(A+B)*A'
EVAL	wertet ein Objekt aus. Ein Variablenname wird durch seinen Inhalt ersetzt, ein Programm ausgeführt und Berechnungen durchgeführt. Beispiel: '4*7+3' EVAL $\Longrightarrow$ 31
→NUM	wertet ein Objekt numerisch aus, d.h. alle Namen werden solange eingesetzt, bis sie ein numerisches Ergebnis liefern. Beispiel: '4*i*π' →NUM $\Longrightarrow$ (0,12.5663706144)
IP	gibt den ganzzahligen Anteil einer Zahl zurück. Beispiel: 3.7 IP $\Longrightarrow$ 3
FP	gibt den Nachkomma-Anteil einer Zahl zurück. Beispiel: 3.9 FP $\Longrightarrow$ .9
NEG	negiert ein Objekt im Stack (wie −1 *). Beispiel: 3.14 NEG $\Longrightarrow$ −3.14
INV	invertiert ein Objekt im Stack (wie 1 SWAP /). Beispiel: 4 INV $\Longrightarrow$ .25
$\sqrt{}$	zieht die Quadratwurzel. Beispiel: 7 $\sqrt{}$ $\Longrightarrow$ 2.64575131106
SQ	quadriert einen Ausdruck (wie 2 ^). Beispiel: 6 SQ $\Longrightarrow$ 36
ABS	bestimmt den Absolutbetrag eines Ausdrucks. Beispiel: −7 ABS $\Longrightarrow$ 7
MAX	liefert die größere zweier Zahlen als Ergebnis. Beispiel: 3 7 MAX $\Longrightarrow$ 7

MIN	liefert die kleinere zweier Zahlen als Ergebnis. Beispiel: 2 8 MIN $\Longrightarrow$ 2
MOD	führt eine Division mit Restangabe durch (Modulo-Operation). Beispiel: 17 5 MOD $\Longrightarrow$ 2 (17/5=3 Rest 2)
FACT	berechnet die Fakultät ganzer Zahlen bzw. $\Gamma(x+1)$ der Gammafunktion, von Werten, die nicht ganzzahlig sind. Beispiel: 1.5 FACT $\Longrightarrow$ 1.32934038818
i	die komplexe Zahl i ($=\sqrt{-1}$); wird vom Rechner auch als (0,1) verwendet. (Notation wie in FORTRAN)
RAND	liefert eine Zufallszahl zwischen Null und Eins.
CEIL	rundet eine Zahl auf den nächsten ganzzahligen Wert auf. Beispiel: 1.6 CEIL $\Longrightarrow$ 2
FLOOR	rundet eine Zahl auf den nächsten ganzzahligen Wert ab. Beispiel: 1.6 FLOOR $\Longrightarrow$ 1
RE	errechnet den Realteil einer komplexen Zahl. Beispiel: (5,9) RE $\Longrightarrow$ 5
IM	bestimmt den Imaginärteil einer komplexen Zahl. Beispiel: (3,7) IM $\Longrightarrow$ 7
CONJ	liefert die konjugiert komplexe Zahl. Beispiel: (2,7) CONJ $\Longrightarrow$ (2,−7)
SIGN	gibt lediglich das Vorzeichen des Arguments zurück. Beispiel: −7 SIGN $\Longrightarrow$ −1
MANT	gibt von einer Zahl die Mantisse zurück. Dies stellt in der Praxis eine Normierung der Zahl (MANT $\cdot 10^{XPON}$) dar, bei der die Mantisse auf einen Wert zwischen 1 und 10 normiert wird (Ausnahme: Null). Beispiel: .76 MANT $\Longrightarrow$ 7.6
XPON	bestimmt den Exponent einer Zahl (MANT $\cdot 10^{XPON}$). Beispiel: .74 XPON $\Longrightarrow$ −1
→Q	(nur HP48); verwandelt eine REAL-Zahl in einen Bruch. Beispiel: 0.25 →Q $\Longrightarrow$ '1/4'
→Qπ	(nur HP48); berechnet aus einer REAL-Zahl einen Ausdruck ($Bruch * \pi$). Beispiel: 0.785398163398 →Qπ $\Longrightarrow$ '1/4*π'

Darüber hinaus verfügt der Rechner noch über eine Vielzahl von weiteren Befehlen, die aber im Rahmen dieses Buchs nicht erklärt werden können. Sie beziehen sich i.d.R. auf spezielle Funktionen wie Matrizenrechnen, den Gleichungslöser etc. Üblicherweise lassen sich diese mit dem Befehl aufrufen, mit dem sie auch bei den normalen Taschenrechnerfunktionen bedient werden.

Sollte man hier einen Befehl vermissen, so ist auf die Kurzerklärung aller Rechner-

Befehle zu verweisen, die sich im Anhang der Benutzerbücher befindet. Nahezu alle aufrufbaren Funktionen lassen sich entsprechend auch in Programmen einsetzen.

5.1.3 Übungsaufgaben zu den Stackbefehlen

Um erst einmal ein Gefühl für die Benutzung von Stackbefehlen zu bekommen, habe ich diesen Abschnitt sehr ausführlich gehalten. Es findet sich hier eine Anzahl von Übungsaufgaben, die nur mit Hilfe von Stackoperationen und den wichtigsten mathematischen Befehlen lösbar sind. Zu programmieren sind hier ausnahmslos Formeln. Ziel ist es, mit minimalem Aufwand (an Anweisungen) das Programm zu erstellen. Als Anweisung ist hier jede Operation (z.B. Lege Zahl in den Stack) einzeln zu zählen. Ebenso soll auch auf vorteilhafte Rechnung hingewiesen werden. Es empfiehlt sich meist, die Formel nicht genau so zu realisieren, wie sie dasteht.

Sinn der möglichst klein zu haltenden Anzahl an Befehlen ist der geringere Bedarf an Speicher und der damit verbundene schnellere Ablauf des Programms.

Man sollte sich durchaus die Mühe machen, die Lösungen erst selbst zu probieren, und braucht auch nicht verzweifeln, wenn meine Musterlösungen kürzer und schneller sind. In diesem Punkt hat der Spruch „Übung macht den Meister“ absolute Gültigkeit. Ich verfüge über einige Hundert Stunden Vorsprung in diesem Metier. Aber auch meine Lösungen sind bestimmt noch weiter optimierbar.

Die Musterlösungen zu den Aufgaben sind alle auf der Diskette unter dem Namen STKAXX zu finden. Sie brauchen also von den Nutzern des HP48 nicht abgetippt werden. Das XX steht für die zweistellige Nummer der Aufgabe (z.B.: Aufgabe 2: STKA02).

Aufgabe 1:

Das erste Beispiel ist absichtlich sehr einfach gehalten. Es soll die Formel $x^2 + x$ programmiert werden.

Der Stack soll zu Beginn x in Ebene 1 beinhalten.

Aufgabe 2:

Welche Funktion hat das Programm: « `ROT + SWAP /` »

Der Stack beinhalte vor Programmbeginn

3:	'a'
2:	'b'
1:	'c'

.

Es sollte erst versucht werden, dies ohne den Taschenrechner herauszufinden.

Aufgabe 3:

Als ein einfach gehaltenes Beispiel soll die Formel $x^3 + x^2 + x$ als Programm geschrieben werden. Die Variable x soll vor Programmstart in Stackebene 1 stehen.

Aufgabe 4:

Als nächstes soll $2x^2 - 3x$ berechnet werden. Auch hier soll vor der Programmausführung x in Ebene 1 des Stacks stehen.

Aufgabe 5:

Als einfaches Beispiel soll die Formel $\frac{x}{y-x}$ programmiert werden.

Der Stack sieht vor Programmstart so aus:

2:	'x'
1:	'y'

Aufgabe 6:

Es sei die Formel $\frac{cb}{a} + b$ zu programmieren.

Der Stack beinhaltet zu Beginn des Programms:

3:	'a'
2:	'b'
1:	'c'

Aufgabe 7:

Zu errechnen ist $\frac{a+b}{a-b}$. Im Stack stehe:

2:	'a'
1:	'b'

Aufgabe 8 :

Zu errechnen ist $\frac{a+b}{a-b} + b$. Im Stack steht:

2:	'a'
1:	'b'

Aufgabe 9:

Es soll $\frac{a+b}{b-a} + a$ programmiert werden. Im Stack steht:

2:	'a'
1:	'b'

Aufgabe 10:

Nun soll $\frac{c-b}{b} + \frac{a}{c-b}$ als Programm verwirklicht werden.

Im Stack steht:

3:	'a'
2:	'b'
1:	'c'

.

Aufgabe 11:

Ein Programm zur Berechnung der Formel $\frac{(x-y)^3}{x}$ soll geschrieben werden. Hierbei soll x in Stackebene 2 und y in Stackebene 1 stehen, bevor das Programm gestartet wird.

Aufgabe 12:

Die Formel $\frac{(x+y-z)^2}{x+z}$ soll nun Gegenstand der Betrachtung sein.

Hier soll der Stack vor Programmstart so aussehen:

3:	'x'
2:	'y'
1:	'z'

.

Aufgabe 13:

Nun soll $a + b + \frac{c}{a+b} + \frac{a+b}{c}$ errechnet werden.

Im Stack steht:

3:	'a'
2:	'b'
1:	'c'

.

Aufgabe 14:

Es soll $\frac{x+y}{x} + 2xy - \frac{x}{y}$ programmiert werden.

Im Stack soll zu Anfang wieder

2:	'x'
1:	'y'

stehen.

Aufgabe 15:

Zu berechnen sei $(xv)^b + \frac{(vw)^c}{ac}$.

Der Stack beinhalte

7:	'a'
6:	'b'
5:	'c'
4:	'v'
3:	'x'
2:	'w'
1:	'v'

.

Aufgabe 16:

Als letztes Beispiel soll nun $\frac{x^2y^2z-v}{z-x} + (x-v)^2 + 2zxv$ programmiert werden.

Der Stack enthält zu Beginn

4:	'v'
3:	'x'
2:	'y'
1:	'z'

.

5.1.4 Musterlösungen der „Übungsaufgaben zu den Stackbefehlen“

Bei all den Übungsaufgaben sind immer eine Vielzahl von Lösungen möglich. Deshalb ist es schwierig, echte Musterlösungen anzubieten. Ein sehr wichtiges Kriterium ist die Kürze der Programme (also der Speicherverbrauch). Der Grund ist einfach: Ein Programm, das wenige Befehle enthält, ist automatisch auch schneller in der Ausführung. Aus diesem Grund ist jeweils der Speicherverbrauch des Programms mit einem einstelligen Namen angegeben (die Länge des Programmnamens hat einen Einfluß auf dessen Speicherverbrauch).

Lösung der Aufgabe 1:

Formel: $x^2 + x$

Musterlösung: `≪ DUP SQ + ≫` Speicherbedarf: 23 Bytes

In der Formel wird x genau zweimal benötigt, deshalb müssen wir x für spätere Berechnungen erst einmal duplizieren.

Aus $\boxed{\begin{array}{ll}1: & \text{'x'}\end{array}}$ wird durch DUP $\boxed{\begin{array}{ll}2: & \text{'x'}\\ 1: & \text{'x'}\end{array}}$.

Jetzt quadrieren wir die Ebene 1 mit SQ und haben damit beide Summanden im Stack stehen: $\boxed{\begin{array}{ll}2: & \text{'x'}\\ 1: & \text{'SQ(x)'}\end{array}}$.

Zuletzt braucht man nur noch die beiden Ausdrücke addieren.

Lösung der Aufgabe 2:

Es errechnet die Formel $\frac{a+c}{b} = \frac{c+a}{b}$.

Lösung der Aufgabe 3:

Formel: $x^3 + x^2 + x$

Sehr einfach ist natürlich die Umsetzung:

```
≪→ x
´x ^ 3 + x ^ 2 + x´
≫
```

Diese Form der Programmierung ist mit 56 Bytes sehr speicheraufwendig. Hingegen hat die Stacklösung nur einen Verbrauch von 33 Bytes. Die Stackversion ist für den Rechner also 41% kürzer.

Musterlösung:

```
≪ DUP SQ OVER 3 ^ +
+
≫
```

Speicherbedarf: 33 Bytes

Da die Programmierung nicht sofort einsichtig ist, soll hier jeder Befehl einzeln mit seiner Wirkung erklärt werden. In dieser Formel taucht x dreimal auf. Es muß Sorge getragen werden, daß x also bis zuletzt allein zur Verfügung steht, um immer wieder darauf zurückgreifen zu können.

Durch DUP wird das x in Ebene 1 dupliziert und erscheint nun als: $\boxed{\begin{array}{ll}2: & \text{'x'}\\ 1: & \text{'x'}\end{array}}$.

Nun quadrieren wir x mit SQ und erhalten somit den zweiten Summanden.

Im Stack haben wir nun: $\boxed{\begin{array}{ll}2: & \text{'x'}\\ 1: & \text{'SQ(x)'}\end{array}}$.

Das x in der zweiten Ebene muß jetzt noch insgesamt einmal dupliziert werden, da noch zwei Summanden berechnet werden sollen. Die eleganteste Kopiermöglichkeit ist fast immer der Befehl OVER.

Durch das OVER entsteht:

Ebene 3	Ebene 2	Ebene 1
'x'	'SQ(x)'	'x'

Der Grund ist, daß OVER das in Stackebene 2 stehende Element in Stackebene 1 herunterkopiert. Das 'x' in Ebene 1 wird nun mit 3 potenziert. Anschließend braucht man nur noch alle drei Summanden aufaddieren.

Selbstverständlich hätte man den ersten Summanden als erstes und erst später den zweiten Summanden berechnen können. Dies wäre sozusagen eine andere Form der Musterlösung.

Lösung der Aufgabe 4:

Formel: $2x^2 - 3x$

Musterlösung: `≪ DUP SQ 2 * SWAP 3 * - ≫` Speicherbedarf: 35.5 Bytes

Auch hier wird x zweimal benötigt. Das Kopieren geschieht mit DUP zu

2:	$'x'$
1:	$'x'$

.

Schauen wir die zu programmierende Formel genauer an. Das Problem ist die Reihenfolge der Berechnung. Sollten wir zuerst den Ausdruck $3x$ bestimmen und erst später $2x^2$, so steht am Schluß das $3x$ über dem $2x^2$. Dies bedeutet einen zusätzlichen Aufwand, um die gewünschte Reihenfolge für die Subtraktion herzustellen. Der Grund ist, daß der zuletzt berechnete Summand immer in Ebene 1 liegt.

Wir berechnen also erst mit SQ 2 * den ersten Ausdruck

2:	$'x'$
1:	'SQ(x)*2'

.

Nun tauschen wir die Ebenen mit SWAP

2:	'SQ(x)*2'
1:	$'x'$

, multiplizieren mit 3 und subtrahieren noch, um das Endergebnis zu erhalten.

Es gibt hier aber noch einen Trick: Die Formel läßt sich ja auch als $2x^2 + (-3x)$ schreiben. Nun haben wir eine Summe, die sich leichter programmieren läßt.

Musterlösung: `≪ DUP −3 * SWAP SQ 2 * + ≫` Speicherbedarf: 35.5 Bytes

Relativ klar ist das DUP, das auch hier dafür sorgt, daß 'x' zweimal zur Verfügung steht. Als nächstes ist die Multiplikation mit -3 nicht zu vergessen, da eine Multiplikation mit 3 bedeuten würde, daß später die Differenz der Ausdrücke gebildet werden

muß. Der Stack beinhaltet nun

2:	'x'
1:	'$x*(-3)$'

.

Durch SWAP SQ 2 * wird nun der andere Term gebildet.

Der Stack zeigt

2:	'$-3x$'
1:	'SQ(x) * 2'

an.

Nun braucht man abschließend nur noch addieren, um das Endergebnis zu erhalten.

Lösung der Aufgabe 5:

Formel: $\frac{x}{y-x}$

Musterlösung: « `OVER - /` » Speicherbedarf: 23 Bytes

In der Formel steht zweimal das x. Aus diesem Grunde duplizieren wir es direkt aus Ebene 2 in Ebene 1.

Nach dem Befehl OVER sind im Stack drei Elemente:

Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'x'

Durch das Minus erhalten wir den Nenner der Formel:

2:	'x'
1:	'y-x'

.

Abschließend brauchen wir nur noch dividieren, um das Ergebnis zu erhalten.

Lösung der Aufgabe 6:

Formel: $\frac{cb}{a} + b$

Musterlösung: « `OVER * ROT / +` » Speicherbedarf: 28 Bytes

Wir brauchen hier als einziges ein zweites b, da alle anderen Variablen lediglich ein einziges Mal benötigt werden. Auch die Position ist zur Berechnung optimal.

Mit OVER kopiert man das b in die unterste Ebene.

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c'	'b'

Nun multipliziert man c mit b und hat damit den Zähler des ersten Summanden errechnet.

Diese Reihenfolge ist günstig, da für die Division das $c*b$ dann schon weiter oben im Stack steht, wenn wir später durch a dividieren wollen.

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c*b'

Das a holen wir nun mit ROT herunter.

Ebene 3	Ebene 2	Ebene 1
'b'	'c*b'	'a'

Nach der Division bekommen wir den ersten Summanden in Ebene 1:

2:	'b'
1:	'(c*b)/a'

Das b kann nun mit einem Plus-Zeichen ganz einfach addiert werden.

Lösung der Aufgabe 7:

Formel: $\frac{a+b}{a-b}$

Musterlösung: `≪ DUP2 + 3 ROLLD - / ≫` Speicherbedarf: 30.5 Bytes

Sicherlich einleuchtend ist hier das DUP2, da wir a und b jeweils zweimal benötigen. Als nächstes errechnen wir den Zähler mit einem Plus.

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'a+b'

Der Vorteil liegt darin, daß im günstigsten Fall der Zähler am Schluß in Ebene 2 stehen bleibt. Dann bildet man den Nenner in Ebene 1 und dividiert nur noch.

Nun stört der Zähler der Formel und wird mit 3 ROLLD weggeschoben.

Ebene 3	Ebene 2	Ebene 1
'a+b'	'a'	'b'

Jetzt können wir den Nenner durch ein Minus-Zeichen bilden und anschließend mit der Division die Berechnung beenden.

Hätten wir erst den Nenner berechnet, so bräuchten wir nun eine weitere Operation, um Zähler und Nenner in die richtige Reihenfolge zu bringen. Es entstünde damit ein längeres und langsameres Programm.

Lösung der Aufgabe 8:

Formel: $\frac{a+b}{a-b} + b$

Musterlösung: ≪ `DUP2 + ROT 3 PICK` Speicherbedarf: 35.5 Bytes
`- / +`
≫

Dieser Fall entspricht nicht dem obigen, da die Variable b insgesamt dreimal verwendet wird. Trotzdem duplizieren wir zunächst die beiden Variablen (mit DUP2). Auch den Zähler des ersten Summanden berechnen wir wieder als erstes, indem wir die Addition ausführen. Der Stack beinhaltet wieder:

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'a+b'

Das a bringen wir nun wieder herunter. Dies geschieht mit dem Befehl ROT.

Ebene 3	Ebene 2	Ebene 1
'b'	'a+b'	'a'

Der Grund für das andere Vorgehen ist recht einfach. Wenn wir das b in Ebene 3 stehen lassen, so stellt es dort ja einzeln betrachtet den zweiten Summanden der Formel dar und kann später einfach addiert werden. Da im Nenner des ersten Summanden $a - b$ steht, haben wir als erstes das a heruntergebracht, damit wir nun durch ein Kopieren von b und Subtraktion den Nenner abschließen können.

Das Kopieren geschieht mit 3 PICK, und die Subtraktion und Division liefert

2:	'b'
1:	'(a+b)/(a-b)'

.

Abschließend brauchen wir nur noch die beiden Ausdrücke zu summieren.

Lösung der Aufgabe 9:

Formel: $\frac{a+b}{b-a} + a$

Musterlösung: ≪ `DUP2 + SWAP 3` Speicherbedarf: 35.5 Bytes
`PICK - / +`
≫

Auch hier starten wir wieder (wie erwartet) mit DUP2 + zu:

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'a+b'

Um den ersten Summanden zu bekommen, wäre als nächster Schritt die Berechnung des Ausdruckes $b-a$ optimal. Das b wird nur noch einmal benötigt und mit SWAP heruntergebracht.

Ebene 3	Ebene 2	Ebene 1
'a'	'a+b'	'b'

Nun kopieren wir a mit 3 PICK herunter und bestimmen den Nenner des ersten Summanden mit dem Minus-Zeichen. Das a bleibt uns somit zusätzlich für die spätere Addition erhalten.

Ebene 3	Ebene 2	Ebene 1
'a'	'a+b'	'b−a'

Abschließend wird dividiert und dann noch beide Summanden addiert.

Lösung der Aufgabe 10:

Formel: $\frac{c-b}{b} + \frac{a}{c-b}$

Musterlösung: `≪ OVER − ROT OVER / SWAP ROT / + ≫` Speicherbedarf: 38 Bytes

Zentraler Punkt dieser Aufgabe ist der Ausdruck $c - b$, der selbstverständlich nicht mehrmals berechnet werden soll. Betrachtet man den zweifach vorkommenden Ausdruck auf diese Weise, so erkennt man, daß c nur einmal einzeln beim Bilden von $c - b$ verwendet wird. Die Variable b wird im Ausdruck $c - b$ und ein zweites Mal allein verwendet. Dies bedeutet, daß sie einmal kopiert werden muß. Bei a ist dies nicht der Fall. Der beste Anfang ist sicherlich das OVER. Hierdurch kopiert man das b. Als Nebeneffekt stehen c und b genau in der richtigen Reihenfolge. Der Stack beinhaltet nach dem OVER:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c'	'b'

Durch Subtraktion erhält man dann:

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c−b'

Für ein weiteres Vorgehen empfiehlt sich nun der Befehl ROT. Zu kopierende Ausdrücke werden am besten aus Ebene 2 kopiert. Dupliziert werden soll $c - b$. Beim Kopieren kommt der Term $c - b$ später in Ebene 1. Für die elegante Weiterbenutzung der Ausdrücke mußte dafür gesorgt werden, daß die Elemente nach dem Duplizieren sofort weiter verwendet werden können. Durch das spätere OVER landet der Term $c-b$ dann in Ebene 1. In Ebene 2 soll dann schon das a stehen, um den zweiten Summanden berechnen zu können.

Durch ROT erhalten wir:

Ebene 3	Ebene 2	Ebene 1
'b'	'c−b	'a'

Nun führen wir OVER aus:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'b'	'c−b'	'a'	'c−b'

Die Division führt dann auch auf den zweiten Summanden:

Ebene 3	Ebene 2	Ebene 1
'b'	'c−b'	'a/(c−b)'

Ab jetzt stört dieser Ausdruck in der weiteren Berechnung. Trotzdem wird der Term nicht mit 3 ROLLD weggeschoben, da die Reihenfolge der beiden anderen Ausdrücke dann falsch wäre. Stattdessen führen wir SWAP ROT aus, so daß jetzt die Reihenfolge stimmt:

Ebene 3	Ebene 2	Ebene 1
'a/(c−b)'	'c−b'	'b'

Die anschließende Division und Addition ist klar und liefert dann auch das gewünschte Ergebnis.

Lösung der Aufgabe 11:

Formel: $\frac{(x-y)^3}{x}$

Musterlösung: `≪ OVER SWAP − 3 ^ SWAP / ≫` Speicherbedarf: 33 Bytes

Nach dem OVER sieht der Stack so aus:

Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'x'

Wir haben nun das x zweimal und das y einmal im Stack stehen, so wie es in der Formel auch benötigt wird. Das folgende SWAP führt zu:

Ebene 3	Ebene 2	Ebene 1
'x'	'x'	'y'

Nun können wir mit einem Minus die Operation für $x - y$ abschließen:

```
2:      'x'
1:  'x-y'
```

Dieses Ergebnis potenzieren wir nun mit 3:

```
2:          'x'
1:  '(x-y)^3'
```

und anschließend wird das x mit einem SWAP heruntergeholt:

```
2:  '(x-y)^3'
1:        'x'
```

Eine Division schließt die Berechnung ab.

Lösung der Aufgabe 12:

Formel: $\frac{(x+y-z)^2}{(x+z)}$

Musterlösung:
```
≪ DUP 4 PICK + 4
ROLLD - + SQ SWAP /
≫
```
Speicherbedarf: 43 Bytes

Hier ergibt sich mit der Reihenfolge der Variablen ein Problem. Klar ist, daß x und z dupliziert werden müssen. Dies muß vor Bildung des Zählers geschehen.

Es macht keinen Unterschied, ob man erst x oder erst z herunterholt. Beides benötigt 3 Befehle. Ich entschied mich für DUP und 4 PICK zu:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'z'	'z'	'x'

Möglich ist nach 3 PICK und OVER aber auch:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'z'	'x'	'z'

Rechnen wir aber mit dem oberen Beispiel weiter. Nach dem Plus haben wir den Nenner, den wir mit 4 ROLLD wegschieben, weil er nur noch stören würde:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'z+x'	'x'	'y'	'z'

Erst jetzt bilden wir den Zähler der Formel mit Minus und Plus zu

```
2:      'z+x'
1:  'x+(y-z)'
```

Den Zähler quadrieren wir nun mit SQ zu

```
2:          'z+x'
1:  'SQ(x+(y-z))'
```

Abschließend wechseln wir noch die beiden Ebenen mit SWAP aus und dividieren, um das Ergebnis zu erhalten.

Lösung der Aufgabe 13:

Formel: $a + b + \frac{c}{a+b} + \frac{a+b}{c}$

Musterlösung: `≪ 3 ROLLD + SWAP OVER / DUP INV + +` `≫` Speicherbedarf: 40.5 Bytes

Auch hier ist wieder zu erkennen, daß $a+b$ immer als ein Block auftaucht. Ebenfalls erkannt werden sollte, daß der zweite Term der inverse Ausdruck des dritten ist.

Es empfiehlt sich, auch hier ein Bilden des Ausdrucks $a + b$ als erste Operation. Dies geschieht mit 3 ROLLD + zu

2:	'c'
1:	'a+b'

.

Nun bringt man das noch zu kopierende Objekt mit SWAP wieder in Ebene 2.

Der Stack sieht nun so aus:

2:	'a+b'
1:	'c'

.

Der Term $a + b$ wird als erster Summand noch gebraucht und kann später unverändert oben stehen bleiben. Das Kopieren geschieht nun mit OVER:

Ebene 3	Ebene 2	Ebene 1
'a+b'	'c'	'a+b'

Jetzt kann man durch die Division den zweiten Summanden bilden:

2:	'a+b'
1:	'c/(a+b)'

Dies dürfen wir aber nur, da der dritte Summand genau der inverse Ausdruck des zweiten ist, d.h. das c und der Term $a + b$ werden nicht mehr einzeln gebraucht. Durch die Befehle DUP INV kann nun auch der dritte Term berechnet werden:

Ebene 3	Ebene 2	Ebene 1
'a+b'	'c/(a+b)'	'INV(c/(a+b))'

Abschließend brauchen die drei Summanden nur noch aufaddiert werden.

Hier soll nun aber noch ein anderer Trick erklärt werden, der sich hin und wieder als praktisch erweist. Das Programm hierfür wäre:

Musterlösung: `≪ 3 DUPN DROP + /` `DUP INV + + +` `≫` Speicherbedarf: 40.5 Bytes

Der Trick besteht darin, eine bestimmte Anzahl von Elementen zu duplizieren und dann gleich wieder einen Teil davon zu löschen. Hier duplizieren wir alle drei Elemente und werfen den untersten der Duplizierten wieder weg.

Mit 3 DUPN DROP entsteht also:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c'	'a'	'b'

Grund hierfür ist, daß wir a und b mehrfach brauchen, c hingegen nicht. Wir duplizieren also mehrere weiter oben liegende Elemente ohne viel Aufwand. Was damit geht, was nicht, wird im Kapitel „Tips und Tricks" ab Seite 127 besser erläutert. Nach der Addition steht folgendes im Stack:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c'	'a+b'

Es stört hier nicht, daß a und b immer noch einzeln im Stack stehen. Sie können aufaddiert als erster Summand betrachtet werden. Nachteil hierbei ist, daß dann einmal häufiger das Plus-Zeichen verwendet werden muß. Der große Vorteil liegt aber darin, daß nun durch Division sofort der zweite Summand gebildet werden kann:

Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c/(a+b)'

Durch Duplizierung und Invertierung erhalten wir auch den dritten Summanden und durch Aufaddition aller Elemente im Stack unser gewünschtes Ergebnis.

Lösung der Aufgabe 14:

Formel: $\frac{x+y}{x} + 2xy - \frac{x}{y}$

Musterlösung: `≪ DUP2 * 2 * 3 ROLLD DUP2 + 3 PICK / 4 ROLLD / - + ≫` Speicherbedarf: 55.5 Bytes

Dieses Problem ist schon recht knifflig. Dementsprechend schwierig ist auch die Erläuterung. Sicherlich leicht verständlich ist der erste Befehl DUP2, da wir drei Summanden haben, die je x und y beinhalten:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'x'	'y'

Ein weiteres DUP2 empfiehlt sich nicht, da viele Elemente im Stack sich nicht mehr so elegant mit den Stackbefehlen verwalten lassen. Auch das $-\frac{x}{y}$ sollte man möglichst zuletzt ausführen, da dann die Subtraktion am einfachsten ausgeführt werden kann.

Es erscheint günstig, nun den zweiten Ausdruck zu errechnen. Nach Multiplikation von x und y sollte man auch gleich die 2 heranmultiplizieren:

Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'x*y*2'

Der Summand ist nun fertig berechnet und wird aus diesem Grunde mit 3 ROLLD ganz nach oben geschoben:

Ebene 3	Ebene 2	Ebene 1
'x*y*2'	'x'	'y'

Jetzt benutzen wir wieder DUP2, um den ersten Summanden zu beginnen.
Nach der folgenden Addition steht im Stack:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'2*x*y'	'x'	'y'	'x+y'

Jetzt kopieren wir x mit 3 PICK herunter (nicht 3 ROLL bzw. ROT, da wir x nochmals brauchen). Nach der Division haben wir:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'2*x*y'	'x'	'y'	'(x+y)/x'

Jetzt schieben wir den ersten Summanden der Formel mit 4 ROLLD nach oben:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'(x+y)/x'	'2*x*y'	'x'	'y'

Ein 3 ROLLD wäre hier natürlich auch möglich gewesen, da wir später sowieso nur summieren. Nun wird $\frac{x}{y}$ durch Division gebildet, dann subtrahiert und anschließend wieder addiert, um das Endergebnis zu erhalten.

Hier stellt sich die Frage nach anderen Programmiermöglichkeiten.

Musterlösung: ≪ `DUP2 + 3 PICK / 3` Speicherbedarf: 55.5 Bytes
`ROLLD DUP2 * 2 * 4`
`ROLDD / - +`
≫

Da das vorzeitige Berechnen von $-\frac{x}{y}$ nicht sinnvoll erscheint (s.o.), berechnen wir also zuerst den ersten Summanden. Dies geschieht mit DUP2 + 3 PICK /:

Ebene 3	Ebene 2	Ebene 1
'x'	'y'	'(x+y)/x'

Mit 3 ROLLD wird dieser Ausdruck dann nach oben geschoben.

Der Stack beinhaltet jetzt:

Ebene 3	Ebene 2	Ebene 1
'(x+y)/x'	'x'	'y'

Nun wird mit DUP2 * 2 * der zweite Summand errechnet:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'(x+y)/x'	'x'	'y'	'x*y*2'

Mit 4 ROLLD (oder 3 ROLLD - Erklärung siehe oben) wird der zweite Summand nun weggeschoben:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'x*y*2'	'(x+y)/x'	'x'	'y'

Mit $/ - +$ schließt man die Berechnung ab. Diese Lösung hat insgesamt exakt gleich viele Anweisungen, ist also ebenso effizient.

Die Frage nach einer Vorgehensmethode, die automatisch zur kürzesten Programmversion führt, kann nicht beantwortet werden. Dies ist im Einzelfall immer genau zu überprüfen. Allein diese 'simple' Formel läßt sich schon in einer Vielzahl von Versionen programmieren. Hier ein Optimum zu finden, gestaltet sich außerordentlich schwierig und ist reine Ausdauer- und Übungssache.

Lösung der Aufgabe 15:

Formel: $(xv)^b + \frac{(vw)^c}{ac}$

Musterlösung: `≪ ROT * 5 ROLL ^ 5` Speicherbedarf: 50.5 Bytes
`ROLLD * OVER ^ 3`
`ROLLD * / +`
`≫`

Im Stack steht zweimal das v. Das x steht in den untersten drei Ebenen und wird nur einmal gebraucht. Deshalb wird man als erstes $x * v$ berechnen. Dies folgt dem Prinzip, möglichst wenig Elemente im Stack zu halten. Nach dem ROT * steht im Stack:

Ebene 6	Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'b'	'c'	'v'	'w'	'v*x'

Da das b nur noch einmal benötigt wird (und dies auch noch beim gerade untenstehenden Ausdruck) holt man das b mit 5 ROLL herunter und potenziert dann:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'a'	'c'	'v'	'w'	'(v*x)^b'

Da der erste Summand nun berechnet ist, schiebt man ihn mit 5 ROLLD weg:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'(v*x)^b'	'a'	'c'	'v'	'w'

Den Zähler des zweiten Summanden zu errechnen ist jetzt mit * OVER ^ möglich:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'(v*x)^b'	'a'	'c'	'(v*w)^c'

Mit 3 ROLLD schieben wir den Zähler des zweiten Summanden nach oben:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'(v*x)^b'	'(v*w)^c'	'a'	'c'

Achtung! Ein 4 ROLLD wäre hier sehr ungünstig, da uns dann wieder das '(v*x)^b' im Wege stehen würde, wenn wir Zähler durch Nenner des zweiten Summanden dividieren wollen. Nun bildet man den Nenner des zweiten Summanden durch eine Multiplikation:

Ebene 3	Ebene 2	Ebene 1
'(v*x)^b'	'(v*w)^c'	'a*c'

Als vorletzten Schritt dividiert man dann Zähler durch Nenner. Zuletzt sind lediglich noch die beiden Summanden zu addieren.

Lösung der Aufgabe 16:

Formel: $\frac{x^2y^2z-v}{z-x} + (x-v)^2 + 2zxv$

Musterlösung: ≪ ROT DUP2 5 ROLL * Speicherbedarf: 88 Bytes

```
SQ * 4 PICK — 3
DUPN DROP - / 4
ROLLD ROT DUP2 — SQ
4 ROLLD * * 2 * + +
≫
```

Dies ist die mir am einleuchtensten erscheinende Lösung. Bei solchen Aufgaben kann man problemlos eine Stunde lang Programmversionen schreiben, die alle auf unterschiedliche Weise das gewünschte Ergebnis liefern.

Dennoch gibt es einige Anhaltspunkte, an die man sich halten kann. Die Variable y wird lediglich ein einziges Mal verwendet. Dadurch empfiehlt es sich, den Ausdruck, der 'y' beinhaltet, als ersten zu berechnen, damit das y bei anderen Berechnungen nicht 'im Wege steht'. Es wurde ja schon erläutert, daß nur bei relativ wenigen Elementen im Stack mit den Befehlen elegant gearbeitet werden kann.

Durch ROT entsteht:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'y'	'z'	'x'

Dies scheint nichts gebracht zu haben. Man darf aber nicht vergessen, daß die untersten drei Ebenen sehr leicht die Berechnung zweier Teil-Terme des ersten Summanden der zu programmierenden Formel beinhalten. Durch DUP2 und 5 ROLL entsteht:

Ebene 6	Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'z'	'x'	'y'

Nun multipliziert man x mit y und quadriert erst dann, um das Quadrieren einmal zu sparen:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'z'	'SQ(x*y)'

Multiplizieren bringt nun:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'z*SQ(x*y)'

Um den Zähler des ersten Summanden zu vervollständigen, kopieren wir v mit 4 PICK herunter und subtrahieren zu:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'z*SQ(x*y)-v'

Jetzt errechnen wir den Nenner des ersten Ausdrucks mit 3 DUPN DROP – und dividieren die beiden Ausdrücke.

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'z*SQ(x*y)-v'	'z-x'

Somit ist der erste Summand berechnet. In den Stackdiagrammen steht vereinfachend nur noch Sum1, entsprechendes gilt für die anderen Summanden, nachdem sie berechnet sind:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
'v'	'z'	'x'	'Sum1'

Mit ROT DUP2 – SQ entsteht nun der zweite Summand:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
Sum1	'z'	'x'	'v'	Sum2

Mit 4 ROLLD schieben wir nun den zweiten Summanden weg:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
Sum1	Sum2	'z'	'x'	'v'

Den letzten Summanden bekommen wir durch * * 2 *, und das Aufsummieren führt zur Berechnung des Ergebnisses.

5.1.5 Vergleichende und logische Operationen

5.1.5.1 Vergleichende Operationen

Sehr wichtige Operationen sind die sogenannten Vergleichsoperationen, bei denen man zwei Objekte in verschiedenster Weise miteinander vergleichen kann.

Die Bedeutung dieser Anweisungen liegt in ihrer universellen Einsetzbarkeit. Wichtig sind sie bei den Verzweigungen, wenn also aufgrund eines bestimmten Ergebnisses verschiedene Fortführungen im Programm nötig sind.

X Y $\geq$	größer gleich – gibt als Ergebnis eine Eins (sonst Null) zurück, wenn X größer gleich Y ist. Beispiel: 5 7 $\geq$ $\Longrightarrow$ 0 8 4 $\geq$ $\Longrightarrow$ 1
X Y $\leq$	kleiner gleich – gibt eine Eins (sonst Null) zurück, wenn X kleiner gleich Y ist. Beispiel: 7 9 $\leq$ $\Longrightarrow$ 1 9 2 $\leq$ $\Longrightarrow$ 0
X Y <	kleiner – gibt eine Eins (sonst Null) zurück, wenn X kleiner als Y ist. Beispiel: 2 6 < $\Longrightarrow$ 1 7 1 > $\Longrightarrow$ 0
X Y >	größer – gibt eine Eins (sonst Null) zurück, wenn X größer als Y ist. Beispiel: 2 6 > $\Longrightarrow$ 0 7 1 > $\Longrightarrow$ 1
X Y $\neq$	ungleich – liefert eine Eins (sonst Null), wenn X ungleich Y ist. Beispiel: 6 3 $\neq$ $\Longrightarrow$ 1 7 7 $\neq$ $\Longrightarrow$ 0
X Y ==	gleich – ergibt eine Eins (sonst Null), wenn X und Y gleich sind. Wie bei der Programmiersprache C ist die Abfrage des *gleich-seins* mit einem Doppel-Ist-gleich-Zeichen zu tätigen, da das einfache Ist-gleich-Zeichen nur Gleichungen vorbehalten ist. Beispiel: 3 4 == $\Longrightarrow$ 0 3 3 == $\Longrightarrow$ 1
X Y **SAME**	hat dieselbe Funktion wie das ==-Zeichen. Das SAME ist hierbei die bessere Alternative, da dieser Befehl immer eine Eins oder Null liefert. Beispiel: 3 'A' SAME $\Longrightarrow$ 0 3 'A' == $\Longrightarrow$ '3==A'

5.1.5.2 Logische Operationen

Von ähnlichem Interesse bei Verzweigungen sind die logischen Operationen, um Entscheidungen treffen zu können. Bei Fällen wie *...wenn dieser* ***oder*** *jener Fall auftritt, dann mache...* werden diese eingesetzt. Ihre Herkunft ist die Boolesche Algebra, bei der die Rechenoperationen mit Hilfe dieser Befehle definiert sind.

AND	für das logische UND. Hierbei gibt der Rechner eine Eins als Ergebnis zurück, falls beide Argumente ungleich Null waren, sonst wird eine Null ausgegeben. Beispiel: 1 0 AND $\Longrightarrow$ 0 1 1 AND $\Longrightarrow$ 1
OR	für das logische ODER. Beim OR wird nur dann eine Null zurückgegeben, wenn beide Argumente Null waren, ansonsten wird eine Eins geliefert. Beispiel: 1 0 OR $\Longrightarrow$ 1 0 0 OR $\Longrightarrow$ 0
NOT	für die logische Negation. Aus einem Wert ungleich Null wird eine Null und umgekehrt aus einer Null eine Eins gemacht. Beispiel: 0 NOT $\Longrightarrow$ 1 1 NOT $\Longrightarrow$ 0
XOR	für das EXCLUSIV-ODER dar. Es funktioniert aber nur im Zusammenhang mit Binärzahlen (#-Darstellung) und ist eine Abwandlung des OR-Befehls. Der Unterschied zum OR besteht darin, daß auch eine Null zurückgegeben wird, wenn beide Argumente Eins waren. Beispiel: #1b #1b XOR $\Longrightarrow$ #0b

Diese Operationen werden auch in vielen anderen Programmiersprachen verwendet. Es sind die wichtigsten Operationen der Booleschen Algebra, bei der bitweise Zahlen miteinander verglichen werden. Sinnvoll können diese Operationen nur bei Binärzahlen ausgeführt werden. Damit sie aber auch bei Verzweigungen eingesetzt werden können, wurde ihre Anwendbarkeit auch auf die normalen Zahlen ausgedehnt. Dies betrifft die Befehle AND, OR und NOT, die auch bei den *normalen Zahlen* korrekte logische Ergebnisse liefern.

5.1.6 Weitere wichtige Befehle

Es folgt eine Aufstellung weiterer wichtiger Befehle, die häufig gebraucht werden, aber sich von ihrer Bedeutung her nicht den anderen Anweisungen zuordnen lassen.

O 'N' **STO**	speichert das Objekt O unter dem Namen 'N' im aktuellen Verzeichnis ab. Beispiel: 1.5 'X' STO
'N' **PURGE**	löscht die Variable bzw. das Programm mit Namen 'N' aus dem aktuellen Verzeichnis. Es tritt kein Fehler auf, wenn dieser Name nicht existiert!
'N' **RCL**	ruft den Inhalt, des unter dem globalen Namen 'N' gespeicherten Objekts in den Stack. Beispiel: 'X' RCL
→STR	verwandelt ein Objekt in eine Zeichenkette. Beispiel: 2 →STR ⟹ "2"
STR→	verwandelt eine Zeichenkette zurück in ihr ursprüngliches Objekt. Beispiel: "23" STR→ ⟹ 23
TYPE	gibt den Typ eines Objekts als Kennzahl an. Beispiel: "HALLO" TYPE ⟹ 2 '2*A' TYPE ⟹ 9 Eine Tabelle mit allen TYPEs befindet sich im Anhang auf Seite 265.
OBJ→	(nur HP48); zerlegt ein Objekt in seine Bestandteile. Beispiel: '7*3+5' OBJ→ ⟹ 4: '7*3' 3: 5 2: 2 1: +

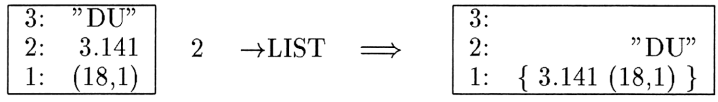
5.1.7 Listen-Befehle

Als sehr vielseitig verwendbar zeigen sich die Listen. Hierfür stellt der Rechner eine große Anzahl an Befehlen bereit, die ich im folgenden als Listen-Befehle bezeichnen werde.

Ihre Bedeutung erschließt sich dem Programmieranfänger in diesem Moment noch nicht. Spätestens bei den „Tips und Tricks" ab Seite 122 erkennt man ihre Bedeutung.

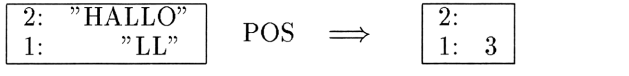
LIST→ löst eine Liste in ihre einzelnen Objekte auf, schreibt sie in den Stack und gibt die Anzahl der Elemente in Ebene 1 zurück.
Beispiel:

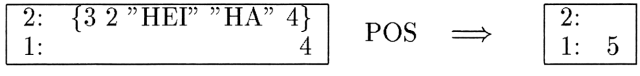
X →LIST schreibt X Elemente in eine Liste.
Beispiel:



SIZE gibt die Anzahl der Elemente einer Liste, die Dimension eines Vektors bzw. eines Feldes, die Länge eines Strings oder die Anzahl der Operanden eines algebraischen Ausdrucks an.
Beispiele: "HALLO" SIZE ⟹ 5
'5*X+3*B' SIZE ⟹ 7
{ 3 2 } SIZE ⟹ 2
[[2 2][2 3][3 1]] SIZE ⟹ { 3 2 }
[1 2 3] SIZE ⟹ {3}

POS gibt die Position eines Teilstrings im String oder die Position eines Elements in einer Liste zurück.
Ist das Element der Stackebene 1 nicht enthalten, so wird eine Null ausgegeben.
Beispiele:





NUM berechnet den Wert eines Charakters im ASCII-Code.
Beispiel: "?" NUM ⟹ 63

CHR verwandelt eine Zahl in den entsprechenden ASCII-Character.
Beispiel: 57 CHR ⟹ "9"

X GET dient zum Extrahieren von Objekten aus Listen oder Zahlen aus Feldern. Hierbei darf je nach Anwendung X auch eine Liste mit einer oder zwei Zahlen sein.

Beispiel:

```
1: [[1 2][3 4]]   { 2 1 }  GET  ⟹  1: 3
1: [6 5 4]        { 2 }    GET  ⟹  1: 5
1: {"DU" "ER" "S"}  2 GET  ⟹  1: "ER"
```

X PUT erlaubt das Ersetzen eines Objekts bzw. einer Zahl in einem Array oder einer Liste. Die Operation muß selbstverständlich erlaubt sein, was zum Beispiel bei [1 2] 1 'A' PUT nicht der Fall wäre, da Vektoren keine Variablen enthalten dürfen.

Beispiel:

```
2: [[1 2][3 4]]
1:     {2 1}      6 PUT  ⟹  1: [[1 2][6 4]]

2: [ 1 2 3 ]
1:   { 2 }        3 PUT  ⟹  1: [ 1 3 3 ]

2: { 7 "ES" }
1:    "ER"        2 PUT  ⟹  1: { 7 "ER" }
```

X Y SUB erzeugt einen Teilstring einer Zeichenkette oder eine Teil-Liste einer Liste von Position X bis Y.

Beispiele:

```
1: {2 "H" 4 7 8}  2 4 SUB  ⟹  1: {"H" 4 7}
1: "HALLO"        2 4 SUB  ⟹  1: "ALL"
```

X Y PUTI bringt das Objekt Y an die Stelle X eines Objekts, sofern das Objekt ein Feld bzw. eine Liste ist, X die Position angibt (als Zahl oder Liste) und Y die Operation zuläßt. Letzteres beruht darauf, daß PUT(I) manchmal nicht zugelassen ist (z.B. ein algebraischer Ausdruck in einer Matrix).

Beispiele:

```
2:
1: { 6 5 4 }      1 'A' PUTI  ⟹  2: { 'A' 5 4 }
                                 1:       2

2:
1: [ 4 2 3 ]      {1} 2 PUTI  ⟹  2: [ 2 2 3 ]
                                 1:   { 2 }

2:
1: [[1 2][3 4]]   { 1 1 } 6 PUTI  ⟹  2: [[6 2][3 4]]
                                      1:   { 1 2 }
```

X GETI extrahiert das X-te Element einer Liste oder, wenn X eine Liste mit ein oder zwei Zahlen ist, eine Zahl aus einem Feld. Hierbei bleibt aber die Liste erhalten, sowie der Zähler X für die Entnahme um Eins erhöht und ebenfalls im Stack abgelegt.
Beispiele:

```
3:
2:
1:  { 'A' 'B' 'C' }      2 GETI  ⟹   3:  { 'A' 'B' 'C' }
                                     2:              3
                                     1:            'B'
```

```
3:
2:
1:  [ 4 5 6 ]      {2} GETI  ⟹   3:  [ 4 5 6 ]
                                 2:       {3}
                                 1:         5
```

```
3:
2:
1:  [[1 2][3 9]]      {1 2} GETI  ⟹   3:  [[1 2][3 9]]
                                      2:        {2 1}
                                      1:           2
```

An diesem Beispiel wird klar, daß der „Zähler" zeilenweise orientiert ist.

Dies sind die wichtigsten Befehle, die im Zusammenhang mit Listen benötigt werden. Aufgrund der sehr universellen Einsetzbarkeit der Listen sind auch die dazugehörigen Befehle von großer Bedeutung.

Zudem verfügt der HP48G/GX zusätzlich noch über eine Vielzahl weiterer nützlicher Befehle zur Handhabung von Listen, die bei den Vorgängern, dem HP28 und dem HP48S/SX, noch nicht zum Befehlssatz gehörten.

ADD addiert zwei Listen elementweise miteinander.
Beispiel:

```
2:  { 1 2 }
1:  { 3 7 }      ADD  ⟹   2:
                          1:  { 4 9 }
```

REVLIST kehrt die Reihenfolge der Objekte in einer Liste um.
Beispiel: { 1 2 3 } REVLIST ⟹ { 3 2 1 }

HEAD nimmt das erste Element aus einer Liste heraus. Kurzform für 1 GET.
Beispiel: { A B C } HEAD ⟹ 'A'

TAIL schneidet das erste Element einer Liste von der Liste ab.
Beispiel: { 1 2 3 4 } TAIL ⟹ { 2 3 4 }

SORT sortiert alle Objekte in einer Liste in aufsteigender Reihenfolge.
Beispiel: { B F A } SORT ⟹ { A B F }

∑LIST addiert alle Objekte einer Liste auf.
Beispiel: { 1 3 7 } ∑LIST ⟹ 11

ΠLIST multipliziert alle Elemente einer Liste.
Beispiel: { 'A' 2 3 } ΠLIST ⟹ 'A*2*3'

ΔLIST gibt eine Liste zurück, bei der beginnend von hinten jedes Element vom jeweils vorherigen Objekt der Liste abgezogen wurde.
Beispiel: { A B C D } ΔLIST ⟹ { 'B-A' 'C-B' 'D-C' }

X P DOLIST wendet eine Funktion bzw. ein Programm auf X Listen an.
Beispiel: Gegeben seien drei Listen L1, L2 und L3. Nun soll (L1 * L2 + L3) ausgeführt werden. Das P steht hier für das Programm, das mit den einzelnen Listenelementen diese Operation ausführen kann (P= ≪ * + ≫).
Beispiel:

L3	3:	{4 7 0}
L2 =	2:	{0 1 3}
L1	1:	{4 1 1}

3 ≪ * + ≫ DOLIST ⟹ {4 8 3}

A P DOSUBS wendet eine Funktion bzw. Programm sequentiell auf einen Teil einer Liste an.
Beispiel:
{ 1 2 3 4 } 3 ≪ + + 2 / ≫ DOSUBS ⟹ { 3 4.5 }
Es wurde also (1+2+3)/2 und (2+3+4)/2 berechnet. Hierbei war die Zahl 3 die Anzahl der jeweils zu verknüpfenden Elemente und das Programm die Vorschrift, welche auf diese Elemente angewendet wurde.

STREAM wendet ein Programm (zwei Ein- und ein Ausgabeparameter) rekursiv auf alle Elemente einer Liste an, wobei der Ausgabewert des Programms einer der beiden neuen Startwerte für den nächsten Durchlauf ist.
Beispiel:

2:	{ 1 2 3 4 }
1:	≪ * 2 / ≫

STREAM ⟹ 3

Es wurde also: (1*2)/2=1
↓
(1*3)/2=1.5
↓
(1.5*4)/2=3 berechnet.

5.1.8 Flags

Die unterschiedlichen Taschenrechnermodelle verfügen noch über sogenannte Flags, die je Flag ein Bit Speicherkapazität haben, d.h. sich eine Null oder Eins merken können. Mit ihnen werden in erster Linie Systemeinstellungen festgelegt. Ein Beispiel wäre hier die Festlegung über die Verwendung des Punkts oder des Kommas als Dezi-

malzeichen. Beim HP48 steuert dies der Flag −51. Ist dieser gelöscht, wird der Punkt als Trennungszeichen von Integer- und Nachkomma-Anteil einer Zahl gewertet. Ist dieser gesetzt, wird das Komma als Trennungszeichen verwendet. Beim HP28S war der Flag 48 hierfür zuständig.

Zu den Flags, mit denen die Systemeinstellungen festgelegt werden, kommen noch Benutzer-Flags, über die der Programmierer frei verfügen kann. Er hat die Möglichkeit, bei seinen Programmen Systemeinstellungen einzufügen oder die Flags für 1-Bit-Informationen intern zu verwenden, sofern er dies möchte. Hierzu hat er eine Vielzahl von Anweisungen zum Prüfen, Löschen und Setzen zur Verfügung.

RCFL	ruft den Inhalt der Flags als Binärwert ab. Um hier aussagekräftige Werte zu bekommen, sollte 64 STWS vorher ausgeführt werden, um 64-Bit-Binärzahlen zuzulassen. Das Ergebnis ist in den 64 Bits des Binärwertes codiert. Beim HP28 wird lediglich ein Binärwert zurückgegeben, während der HP48 eine Liste mit zwei Binärwerten für die Systemeinstellungen und Benutzerflags ausgibt.
X **STOF**	speichert den Binärwert X als Code für alle Flags. Beim HP28 handelt es sich bei X lediglich um eine (Binär-)Zahl, während der HP48 auch eine Liste mit zwei Binärwerten zuläßt. Beim HP28 werden mit dieser Binärzahl gleichzeitig die System- und die Benutzerflags festgelegt. Beim HP48 werden mit einer einzelnen (Binär-) Zahl nur die Systemflags gesetzt, bei der Liste mit zwei Werten mit der zweiten Zahl auch noch die Benutzerflags.
X **SF**	setzt Flag Nummer X auf den Wert 1.
X **CF**	löscht Flag Nummer X auf den Wert 0.
X **FS?**	liefert eine Eins, wenn das Flag Nummer X gesetzt war (ansonsten Null). Der Inhalt des Flags wird nicht verändert.
X **FC?**	liefert eine Eins, wenn das Flag Nummer X gelöscht war (ansonsten Null). Der Inhalt des Flags wird hierbei nicht geändert.
X **FS?C**	arbeitet wie FS?, nur daß das Flag am Schluß immer gelöscht wird.
X **FC?C**	arbeitet wie FC?, nur daß das Flag am Schluß immer gelöscht wird.

Die frei verfügbaren (Benutzer-) Flags sind je nach Taschenrechnermodell unterschiedlich:

- Flag 1 bis 64 beim HP48
- Flag 1 bis 30 beim HP28

Die Bedeutung der restlichen (System-) Flags erschließt sich aus den Benutzerhandbüchern.

5.1.9 Interaktive Befehle

Bisher haben wir eine große Anzahl von Befehlen kennengelernt, die alle lediglich über den Stack Ein- und Ausgaben zuließen. Der Taschenrechner hält zudem aber noch eine Anzahl von weiteren Anweisungen bereit, die zusätzliche Möglichkeiten erschließen. Es handelt sich um Befehle, die interaktives Arbeiten ermöglichen.

HALT	unterbricht die Programmausführung, und der HALT-Indikator in der Status-Zeile geht an. Der Rechner wartet auf CONT oder SST bzw. SST↓ zur Weiterführung des Programms. Dieser Befehl erlaubt zwischenzeitliche Eingaben während des Ablaufs des Programms oder dient der Fehlersuche (siehe Seite 38). Durch das Anhalten im Programmablauf sind Eingaben von Daten in den Stack möglich, die bei einem Fortfahren der Ausführung verarbeitet werden können.
s **WAIT**	führt eine Pause von s Sekunden aus.
KEY	liefert einen Tastenstring (HP28) bzw. eine Code-Nummer (HP48) der Taste zurück, wenn eine Taste gedrückt wurde. Eine Tabelle der Code-Nummern des HP48 findet sich auf Seite 270.
CLLCD	löscht das LC-Display.
X N **DISP**	zeigt das Objekt X in Zeile N des Displays an.
F S **BEEP**	gibt ein Akustiksignal der Frequenz F und der Dauer S Sekunden aus, wenn der Akustik-Flag Töne zuläßt (HP48: Flag −57 und HP28: Flag 51)
CLMF	(nur beim HP28) – zum Wiederherstellen der normalen Anzeige nach Anwendung von DISP-Befehlen. Der HP48 führt diese Funktion automatisch immer aus.
L **TMENU**	eröffnet eine benutzerdefinierte Menüleiste, über die man Werte abspeichern oder abrufen kann. Das L steht für eine Liste mit beliebig vielen Namen von Variablen oder Programmen. Beispiel: { DE INPOL MMUL } TMENU ⟹ eine Menüleiste mit den drei Einträgen Beim HP28 hieß der Befehl nur MENU. Dieser Befehl existiert beim HP48 ebenfalls, hat aber als unangenehmen Nebeneffekt nun das Speichern der Liste L in der Variablen CST, was meist nicht erwünscht ist.

X Y **INPUT**	(nur HP48G/GX) tätigt eine Eingabe aus einem Programm heraus und schreibt gleichzeitig eine Meldung ins Display. Das X und das Y stehen hier jeweils für einen String, wobei X im Display weiter oben geschrieben wird, während Y als String direkt über der Menüleiste ausgegeben und nachher zur Eingabe hinzugenommen wird. Beispiel: "HALLO" "EINGABE=" INPUT ENTER 89 ENTER $\Longrightarrow$ "EINGABE=89" **Hinweis**: Drückt man nach Ausführung des INPUT-Befehls die ON-Taste, so verschwindet die untere Meldung (also der String Y).

5.2 Schleifenstrukturen

5.2.1 Die Vorstellung der Schleifenstrukturen

Bei den Schleifenstrukturen unterscheidet man prinzipiell zwei Typen von Schleifen. Diese beiden bezeichnet man als bestimmte und unbestimmte Schleifen. Der Unterschied zwischen beiden basiert darauf, daß bei einer bestimmten Schleife das Anfangs- und das Endargument vor dem Start der Wiederholung angegeben werden müssen.

5.2.1.1 Bestimmte Schleifen

Der Typ der bestimmten Schleifen kann als Struktogrammbaustein durch

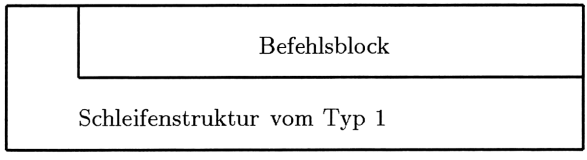


Abb. 5-1: *Schleifenstruktur vom Typ 1*

dargestellt werden.

Dem Taschenrechner bekannte bestimmte Schleifen sind die

- START-NEXT-
- START-STEP-
- FOR-NEXT-

und die

- FOR-STEP-Schleife

die im folgenden erklärt werden.

Die START-NEXT-Schleife

Die einfachste Schleife ist die START-NEXT-Konstruktion, bei der der Rechner den Befehlsblock zwischen dem START- und dem NEXT-Befehl wiederholt. Die Anzahl der Wiederholungen richtet sich nach dem Anfangs- und dem Endwertargument der Schleife.

Beispiel:

```
≪ 2 7
  START Anweisungen
  NEXT
≫
```

Das obige Programm erzeugt im Rechner einen Zähler, der als Anfangswert den Wert 2 besitzt. Während der Anweisungen bleibt der Wert des Zählers gleich. Ist bei Erreichen des NEXT-Befehls der Zähler ≤ 7, so wird der Anweisungsblock wiederholt und der interne Schleifenzähler um Eins erhöht (inkrementiert).

In obigem Beispiel werden die *Anweisungen* also sechsmal ausgeführt. Erst danach führt der Taschenrechner die dem NEXT-Befehl folgende Anweisung aus (hier nichts).

Bei Fällen wie

```
≪ 1 2.5
  START Anweisungen
  NEXT
≫
```

, bei der der Zähler ja nie den Endwert erreicht, wird die Ausführung der Schleife dann abgebrochen, wenn der (interne) Zähler den Endwert überschritten hat. In diesem Beispiel wird die Schleife also genau drei mal ausgeführt.

Basierend auf diesem Wissen ist die Funktion der anderen Schleifen relativ leicht erklärt.

Die START-STEP-Schleife

Bei der START-STEP-Schleife kann man die sogenannte Schrittweite, um die der (interne Schleifen-) Zähler modifiziert wird, frei wählen.

Beispiel:
```
≪ 1 5
  START Anweisungen
  .5 STEP
≫
```

Obiges Beispiel wird genau 10 mal ausgeführt, da der Zähler erst nach zehn Schleifendurchläufen den Wert 5 erreicht bzw. überschritten hat.

Die FOR-NEXT-Schleife

Die FOR-NEXT-Schleifen erlauben die Benutzung des internen Zählers.

Das Programm
```
≪ 1 3
  FOR E Anweisungen
  NEXT
≫
```
führt die *Anweisungen* dreimal aus und erlaubt innerhalb der Befehle FOR und NEXT die Benutzung der Schleifenvariable E, die den aktuellen Stand des Zählers beinhaltet. Selbstverständlich darf jeder beliebige Name als Schleifenzähler verwendet werden, der in einer noch offenen Schleife noch nicht benutzt wurde. Nicht erlaubt sind auch Namen, die Befehle sind. Die Schrittweite des Zählers ist immer Eins.

Die FOR-STEP-Schleife

Bei der FOR-STEP-Schleife ist zudem noch erlaubt, die Schrittweite des Schleifenzählers flexibel zu ändern. Das Prinzip gleicht dem der START-STEP-Schleife.

Die FOR-STEP-Schleife ist mit Sicherheit die wichtigste, da allgemeinste, der sogenannten bestimmten Schleifenstrukturen. Ebenfalls häufig Verwendung findet auch noch die FOR-NEXT-Schleife, deren konstante Schrittweite meist auch ausreicht.

5.2.1.2 Unbestimmte Schleifen

Die Verwendung von DO und UNTIL

Zudem gibt es noch eine unbestimmte Schleifenstruktur des Typs 1. Es ist die DO-UNTIL-END-Schleife. Diese wird als

```
≪ DO Anweisungen
  UNTIL Bedingung
  END
≫
```

verwendet.

Hierbei wird ein Anweisungsblock ausgeführt, bis die Bedingung erfüllt (ungleich Null) ist. Aus dem Satz geht schon hervor, daß der Anweisungsblock mindestens einmal ausgeführt wird. Dieser Typ wird deshalb auch nicht-abweisende Schleife genannt.

Die WHILE-REPEAT-END-Struktur

Die Schleifenstruktur des zweiten Typs läßt sich als

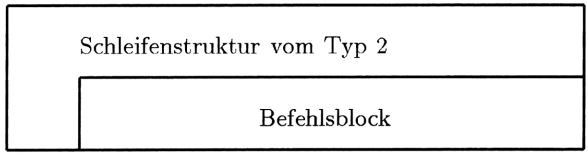


Abb. 5–2: *Schleifenstruktur vom Typ 2*

darstellen.

Vom Typ 2 der Schleifenstrukturen kennt der Taschencomputer nur die WHILE-REPEAT-END-Schleife, die als

```
≪ WHILE Bedingung erfüllt
  REPEAT Anweisungsblock
  END
≫
```

verwendet wird. Sie ist wie die DO-UNTIL-Konstruktion ebenfalls eine unbestimmte Schleifenstruktur. Hier wird aber erst eine Schleifenbedingung geprüft und erst dann die *Anweisung* ausgeführt, wenn diese Bedingung erfüllt (ungleich Null) war. Aus diesem Grund heißt sie auch abweisende Schleife. Die Wiederholung wird solange ausgeführt, bis die Bedingung nicht mehr erfüllt ist.

Achtung! Es soll nochmals darauf hingewiesen werden, daß eine FOR/START-NEXT/STEP-Konstruktion immer mindestens einmal durchlaufen wird. Die Schleife

```
3 1
  FOR E Anweisungen
  NEXT
```

wird mit E=3 einmal durchlaufen. Diese Konstruktionen (mit Ausnahme der WHILE-REPEAT-END-Schleife) sind also alle nicht-abweisende Schleifen.

5.2.2 Übungsaufgaben zu den Schleifenstrukturen

Naturgemäß lernt man beim Üben immer noch am meisten. Aus diesem Grunde folgen wieder einige Aufgaben, bei denen auch hier wieder versucht werden sollte, sie allein zu lösen. Das wichtigste hierbei ist, zu erkennen, welcher Schleifentyp der günstigste (einfachste) ist.

Auch diese Musterlösungen sind allesamt auf der Diskette gespeichert. Die Notation entspricht der bei den Musterlösungen zu den Stackaufgaben, mit dem Unterschied, daß die Notation hier SLAXX ist (z.B. für Aufgabe 2: SLA02).

Aufgabe 1:

Zu Schreiben ist ein Programm, das 17 mal die Zahl 3.141 in den Stack schreibt.

Aufgabe 2:

Gesucht wird ein Programm, das die Zahlen von -5 bis 7 in Zehntel-Schritten in den Stack schreibt.

Aufgabe 3:

Welche Funktion hat folgendes Programm:

```
≪ 0 1 3                    Speicherbedarf: 92 Bytes
  FOR E 0 -3 3
    FOR N N E FACT
/ E N  ^  * +
    NEXT E SQ / +
  NEXT
≫
```

Es liefert als Ergebnis: 5.95466820988

Aufgabe 4:

Welche Funktion hat folgendes Programm:

```
≪ 0 4 9                    Speicherbedarf: 91.5 Bytes
  FOR N 0 8 17
    FOR E E DUP LN
2 / SWAP / +
    NEXT N SQ / +
  NEXT
≫
```

Es liefert als Ergebnis: 0.18452386202

Aufgabe 5:

Es soll ein Programm geschrieben werden, das fünfmal die Zahlen von 3 bis 7 in eine Liste schreibt.

Aufgabe 6:

Es soll $\sum_{E=3}^{8} \frac{1}{E}(\frac{E+2}{ln(E)})$ berechnet werden.

Aufgabe 7:

Gesucht wird ein Programm, das $\sum_{N=2}^{7}(\frac{1}{N}(\sum_{E=8}^{18}(E^2+E)))$ berechnet.

Aufgabe 8:

Gesucht sei ein Programm, das von einer beliebigen Zahl, die vor Programmstart im Stack steht, den ganzzahligen Anteil bildet und dann rückwärts zählend jeweils das Quadrat dieses Zählers im Stack ablegt. Die Abbruchbedingung sei dann erfüllt, wenn der Zähler nicht mehr größer als Null ist.

Aufgabe 9:

Es soll ein Programm geschrieben werden, das eine im Stack stehende Zahl dupliziert und dann quadriert. Nachdem das Ergebnis dieser Rechnung im Stack abgelegt wurde, soll eine Zufallszahl (zwischen Null und Eins) an die duplizierte Zahl heranmultipliziert werden. Das ganze soll solange durchgeführt werden, bis die durch die Multiplikationen immer kleiner gewordene Zahl kleiner als Eins wurde.

5.2.3 Musterlösungen der „Übungsaufgaben zu den Schleifenstrukturen“

Lösung der Aufgabe 1:

Musterlösung: Speicherbedarf: 44 Bytes

```
≪ 1 17
  START 3.141
  NEXT
≫
```

Es genügt eine START-NEXT-Schleife, da hierbei weder der (Schleifen-) Zähler noch die von 1 abweichende Schrittweite des Zählers benötigt werden. Als Anweisungsblock in der Schleife steht nur die Zahl 3.141, die durch die Schleife 17 mal in den Stack geschrieben wird.

Lösung der Aufgabe 2:

Musterlösung: Speicherbedarf: 45 Bytes

```
≪ -5 7
  FOR E E .1
  STEP
≫
```

Hier wird eine FOR-STEP-Schleife benötigt, da man den Zähler, der in Zehntel-Schritten erhöht wird, benutzt, um die gewünschten Zahlen in den Stack zu schreiben. Die Version mit STEP wird benötigt, um die Zehntel-Schritte auszuführen, und das FOR, da der Schleifenzähler eingesetzt werden muß.

Natürlich kann man auch mit einer FOR-NEXT-Schleife arbeiten. Hierzu sind allerdings erst ein paar weitergehende Überlegungen nötig.

Wir können genausogut eine Wiederholung kreieren, bei der man von -50 bis 70 zählt und dann einfach nur ein Zehntel des Schleifenzählers in den Stack schreibt. Dadurch haben wir die Schrittweite und die Argumente verzehnfacht, gleichen dies aber durch eine Division durch 10 wieder aus.

Musterlösung: Speicherbedarf: 63.5 Bytes

```
≪ -50 70
  FOR E E 10 /
  NEXT
≫
```

Ein Vergleich des Speicherverbrauchs weißt wohl schon auf die günstigere Lösung hin.

Lösung der Aufgabe 3:

Es berechnet die Formel: $\sum_{E=1}^{3}(\frac{1}{E^2}\sum_{N=-3}^{3}\frac{N}{E!}E^N)$

Lösung der Aufgabe 4:

Es berechnet die Formel: $\sum_{N=4}^{9}(\frac{1}{N^2}\sum_{E=8}^{17}\frac{ln(E)}{2E})$

Lösung der Aufgabe 5:

Musterlösung:
```
≪ 1 5
  START 3 7
    FOR E E
    NEXT 5 →LIST
  NEXT
≫
```
Speicherbedarf: 49.5 Bytes

Die Aufteilung in Teilprobleme ergibt zuerst die Suche nach einem Programm, das die Zahlen von 3 bis 7 in eine Liste schreibt. Dies erledigt:

```
3 7
  FOR E E
  NEXT 5 →LIST
```

Die Schleife muß von 3 bis 7 laufen und jeweils den Schleifenzähler im Stack ablegen. Sind die Zahlen im Stack, so führt man nur noch ein 5 →LIST aus, um diese fünf Zahlen in eine Liste zu übernehmen.

Da dies insgesamt fünfmal geschehen soll, führt man dies mit einer START-NEXT-Schleife fünf Mal aus.

Lösung der Aufgabe 6:

Musterlösung:
```
≪ 0 3 8
    FOR E E DUP 2 +
  OVER LN / SWAP / +
    NEXT
≫
```
Speicherbedarf: 59.5 Bytes

Ergebnis: 5.47495244273

Hier wird nur eine FOR-NEXT-Schleife benötigt, die von 3 bis 8 zählt. Es ist eine Summe zu programmieren. Jeder Wert, der errechnet wurde, wird also zum bisherigen Summen-Wert addiert. Beim ersten Ergebnis muß man allerdings sicherstellen, daß die Plus-Operation auch durchführbar ist. Die Null zu Beginn des Programms dient also lediglich als Startwert für die Aufsummation.

Etwas komplizierter ist die Wiederholungssequenz. Das E wird mehrfach gebraucht, also wird es nach dem Aufruf gleich dupliziert zu

2:	E
1:	E

.

Das E steht hier nur für den Inhalt des Schleifen-Zählers.

Als nächstes addieren wir die Zahl Zwei, um den Zähler des zweiten Bruchs zu erhalten:

2:	E
1:	E+2

Nun holen wir mit dem OVER das E aus Ebene 2 und ziehen den Logarithmus naturalis:

Ebene 3	Ebene 2	Ebene 1
E	E+2	LN(E)

Nun dividieren wir Zähler und Nenner zu

2:	E
1:	(E+2)/LN(E)

.

Als letzten Schritt werden die beiden Ebenen getauscht und dividiert. Zum Aufsummieren fehlt jetzt nur noch die Addition.

Lösung der Aufgabe 7:

Musterlösung:

```
≪ 0 2 7
  FOR N 0 8 18
    FOR E E DUP SQ
+ +
    NEXT N / +
  NEXT
≫
```

Speicherbedarf: 81.5 Bytes

Ergebnis: 3364.11428571

Auch hier muß man das Problem erst in zwei kleinere Probleme auftrennen. Zuerst sucht man ein Programm, das

$\sum_{E=8}^{18}(E^2+E)$

errechnet. Schnell erkannt ist hier, daß man eine FOR-NEXT-Schleife benötigt, da der Schleifenzähler in der Formel benötigt wird. Die Schleife läuft von 8 bis 18 und berechnet jeweils `E DUP SQ + +`. Dieser Wert wird immer aufsummiert. Dies geschieht,

indem man vor Beginn der Schleife eine Null in den Stack schreibt, auf die man die Summanden der Reihe nach aufsummiert. Das erste Teil-Programm ist also:

```
0 8 18
  FOR E E DUP SQ
+ +
  NEXT
```

Als nächstes muß man die äußere Summe um dieses Teil-Programm herumlegen. Bei jedem Wert der inneren Schleife muß durch N dividiert und dann der Wert aufsummiert werden. Also wird um das Teil-Programm nun folgende Schleife herumgelegt:

```
≪ 0 2 7
  FOR N
    1. Teil-Programm
    N / +
  NEXT
≫
```

Lösung der Aufgabe 8:

Musterlösung:

```
≪ IP
  WHILE DUP 0 ≥
  REPEAT DUP SQ
SWAP 1 -
  END DROP
≫
```

Speicherbedarf: 53 Bytes

Die erste Operation ist selbstverständlich das IP, das zum Bilden des ganzzahligen Anteils gebraucht wird. Als Schleife ist dann eine abweisende Schleife notwendig, da nicht ausgeschlossen werden kann, daß eine negative Zahl vor Programmstart im Stack steht. Das Struktogramm in Abbildung 5–3 zeigt den Aufbau des Programms. Wir brauchen also eine WHILE-REPEAT-END-Konstruktion.

Die Bedingung der Schleife ist der Test, ob der Zähler größer gleich Null ist. Ist dies erfüllt, so wird die Zahl erst dupliziert und dann quadriert. Das DUP sichert uns also den Zähler, der sonst ja verloren ginge, beim Test darauf, ob er nun noch größer gleich Null ist.

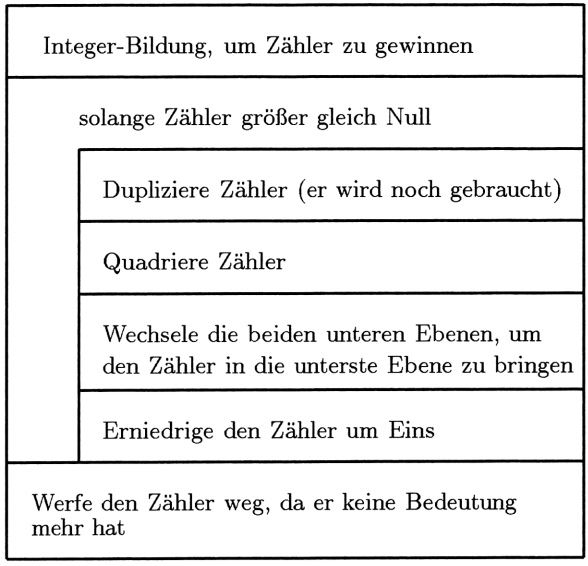


Abb. 5–3: *Struktogramm zu Aufgabe 8*

Das Programm soll auch gleich mit den Zahlen -6 und 7.5 getestet werden. Im ersten Fall scheint das Programm nichts zu tun, beim zweiten wird es wunschgemäß ausgeführt. Dies ist aber die korrekte Arbeitsweise.

Lösung der Aufgabe 9:

Musterlösung: ≪ Speicherbedarf: 45.5 Bytes

```
  DO DUP SQ SWAP
RAND *
  UNTIL DUP 1 ≤
  END DROP
≫
```

Die erste Operation ist selbstverständlich das IP, das zum Bilden des ganzzahligen Anteils gebraucht wird. Als Schleife ist dann eine abweisende Schleife notwendig, da nicht ausgeschlossen werden kann, daß eine negative Zahl vor Programmstart im Stack steht. Auch hier soll ein Struktogramm die Funktion des Programms erläutern:

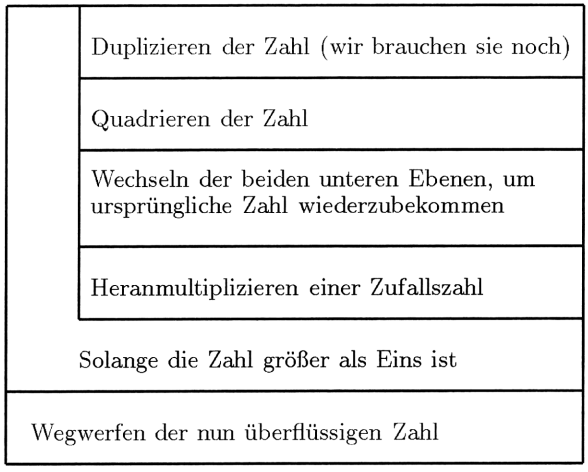


Abb. 5–4: *Struktogramm zu Aufgabe 9*

Aus Formulierung und Struktogramm geht hervor, daß das Quadrieren mindestens einmal vorgenommen werden soll. Es muß also mit einer DO-UNTIL-END-Schleife gearbeitet werden, da der Anweisungsblock mindestens einmal ausgeführt werden soll.

Die Anweisungen DUP SQ SWAP RAND * gehen in ihrer Bedeutung schon aus dem Struktogramm hervor. Nötig ist das erste DUP, da sonst die Originalzahl durch die anderen Operationen verschwindet. Hinter dem SQ steht im Stack

2:	Org-Zahl
1:	SQ(Org-Zahl)

.

Durch das SWAP holen wir die Original-Zahl wieder herunter:

2:	SQ(Org-Zahl)
1:	Org-Zahl

.

Dann multiplizieren wir die Zufallszahl.

Jetzt geht es an den Test. Unsere Problemstellung wird bei der DO-UNTIL-END-Schleife wie folgt formuliert:

Führe obige Rechnung durch, bis modifizierte Original-Zahl kleiner gleich Eins ist. Um die modifizierte Zahl beim Test nicht zu verlieren, muß sie erst dupliziert werden. Nach dem DUP kommt der Test 1 $\leq$. Dies ist die *solange bis*-Bedingung. Sollte die Zahl kleiner als Eins sein, springt der Rechner aus der Schleife und wirft die nun überflüssige Zahl mit DROP weg.

5.3 Verzweigungsstrukturen

5.3.1 Die Verzweigungstypen

Ein weiteres Element der Programmierung ist die sogenannte Verzweigungsstruktur.

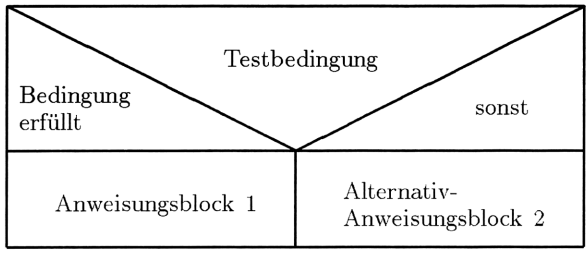


Abb. 5–5: Zweiseitige Verzweigungsstruktur

Der Taschenrechner kennt davon eine ganze Anzahl:

- IF-THEN-END
- IF-THEN-ELSE-END
- IFT
- IFTE
- IFERR-THEN-END
- IFERR-THEN-ELSE-END

und

- CASE-THEN-END-END (ab HP48)

Am gebräuchlichsten sind die ersten beiden IF-Verzweigungen. Dennoch werden auch hier alle einzeln beschrieben. Man unterscheidet zwischen einseitigen und zweiseitigen Verzweigungen. Einseitige Verzweigungen sind dadurch gekennzeichnet, daß eine Befehlssequenz nur dann ausgeführt wird, wenn eine Bedingung erfüllt wird. Ist diese nicht erfüllt, so wirkt es nach außen hin, als ob diese Struktur gar nicht vorhanden sei.

Bei der zweiseitigen Konstruktion wird bei dem Nicht-erfüllt-sein der Bedingung eine Alternativsequenz ausgeführt.

5.3.1.1 Einseitige Verzweigungsstrukturen

Die IF-THEN-END-Struktur

Die einseitige Struktur

```
≪
  IF Bedingung
  THEN Anweisungsblock
  END
≫
```

Hierbei wird zwischen dem IF- und dem THEN-Befehl eine Bedingung gestellt und geprüft. Der *Anweisungsblock* wird nur dann ausgeführt, wenn sie *wahr* (also Eins) war. Beim Taschenrechner wurde diese Funktion sogar erweitert. Der *Anweisungsblock* wird immer dann ausgeführt, wenn als geprüfte Bedingung ein Wert ungleich Null herauskommt. Dies soll an zwei Beispielen verdeutlicht werden.

Beispiel 1:

```
≪
  IF 1 0 ==
  THEN "WAHR"
  END
≫
```

Beispiel 2:

```
≪
  IF .1
  THEN "WAHR"
  END
≫
```

Beispiel 1 prüft ab, ob Eins gleich Null ist. Dies ist natürlich nicht erfüllt. Der Rechner gibt deshalb eine Null an den THEN-Befehl zurück, die bei der Verzweigungsstruktur eine Nichtausführung des *Anweisungsblocks* bewirkt. Bei Beispiel 2 hingegen wird dem THEN-Befehl ein Wert ungleich Null geliefert. Der Taschenrechner interpretiert diese Zahl als eine (hier nur simulierte) wahre Bedingung und führt die Befehlssequenz zwischen dem THEN- und dem END-Befehl aus. Das Ergebnis ist das Wort "WAHR" im Stack.

Der IFT-Befehl

Die IFT-Verzweigung ist eine andere Form der IF-THEN-END-Struktur. Der Unterschied liegt darin, daß man die Bedingung und den ausgeführten Befehlsblock vor dem Erreichen des IFT-Befehls im Stack ablegen muß.

Es soll wieder dieselbe Aufgabe wie bei der IF-THEN-END-Sequenzerklärung gelöst werden.

Die Routine sähe so aus: `≪ 1 0 == "WAHR" IFT ≫`

Der Vorteil liegt bei dieser zweiten IF-THEN-END-Befehlsmöglichkeit darin, daß Speicher gespart werden kann, wenn der THEN-Befehlsblock nur aus einem Objekt besteht, welches kein Befehl ist. In der Praxis ist dies aber recht selten.

Der gravierende Nachteil liegt in der nicht möglichen Schachtelung und dem meist höheren Speicheraufwand, wenn mehrere Befehle als THEN-Befehlssequenz ausgeführt werden sollen. Zudem ist der Programmablauf langsamer, da der THEN-Block immer ausgeführt wird, auch wenn er gar nicht erfüllt ist.

Die IFERR-THEN-END-Sequenz

Eine Besonderheit der Programmiersprache RPL sind die IFERR-Anweisungen. Diese Option erlaubt das Abfragen von Rechnerabstürzen sowie die Behandlung möglicher Fehlerquellen.

Die einfache Art ist die IFERR-THEN-END-Sequenz. Hier wird ein Anweisungsblock zwischen dem IFERR- und dem THEN-Befehl ausgeführt. Sollte in diesem Block ein Fehler auftreten, der zu einer Fehlermeldung des Rechners führen würde, so wird diese unterdrückt und stattdessen die Sequenz zwischen THEN- und END-Befehl ausgeführt.

Ein Beispiel soll dies erläutern: Gesucht sei ein Programm, das an ein Objekt die Zahl 5 heranmultipliziert. Da diese Operation nicht mit allen Objekten erlaubt ist, werden bei Eingabe von falschen Objekttypen Fehler zu erwarten sein. Als Meldung soll der Rechner bei Erkennen eines solchen Fehlers den Stack löschen und die Meldung "FALSCHER TYP" ausgeben.

Lösung:
```
≪
  IFERR 5 *
  THEN CLEAR
"FALSCHER TYP"
  END
≫
```

Getestet werden kann dieses Programm mit einer Zahl und einer Zeichenkette als Eingabe.

5.3.1.2 Zweiseitige Verzweigungen

Die IF-THEN-ELSE-END-Sequenz

Die zweiseitige IF-THEN-ELSE-END-Struktur erlaubt eine Erweiterung der einseitigen IF-THEN-END-Konstruktion. Ist hier die Testbedingung nicht erfüllt, so wird dann ein Alternativanweisungsblock ausgeführt. Ansonsten gilt hier genau dasselbe wie bei der IF-THEN-END-Bedingung.

Als Beispiel soll ein Programm geschrieben werden, das eine Zahl vom Stack nimmt und prüft, ob diese Zahl kleiner als Null ist. Das Ergebnis soll das Wort "NEGATIV" sein, wenn die Zahl kleiner Null war, ansonsten soll "POSITIV" im Stack abgelegt werden. (Null soll hier also als positive Zahl angesehen werden.)

Lösung:
```
≪
  IF 0 <
  THEN "NEGATIV"
  ELSE "POSITIV"
  END
≫
```

Selbstverständlich kann man auch schreiben:

```
≪
  IF 0 ≥
  THEN "POSITIV"
  ELSE "NEGATIV"
  END
≫
```

Hier wurde also nur die Testbedingung umgedreht und der THEN-Befehlsblock mit der ELSE-Anweisungssequenz vertauscht. Zu beachten ist hier, daß beim unteren Beispiel nun abgefragt werden muß, ob die eingegebene Zahl größer gleich Null ist. Vergißt man das Gleich-Zeichen, würde der Rechner die Zahl Null beim oberen Beispiel als positiv, beim unteren hingegen als negativ bewerten. Der Leser möge das durch Abändern des $\geq$-Zeichens durch ein $>$-Zeichen bei letzterem Beispiel überprüfen.

Die IFTE-Anweisung

Die IFTE-Anweisung ist die Übertragung der IF-THEN-ELSE-END-Struktur auf dieselbe Weise wie beim IFT-Befehl. Die Vor- und Nachteile entsprechen sich ebenfalls. Die Abänderung des Beispiels von oben soll die Anwendung erläutern:

```
≪ 0 ≥ "POSITIV"
"NEGATIV" IFTE
≫
```

Diese Version hat also die gleiche Funktion wie das Beispiel bei der Erklärung der IF-THEN-ELSE-END-Struktur.

Generell kann getrost vom IFT- und vom IFTE-Befehl abgeraten werden, da sie viel zu selten Vorteile bringen und recht unflexibel sind.

Die IFERR-THEN-ELSE-END-Befehlsfolge

Eine zusätzliche Erweiterung zur einseitigen IFERR-Konstruktion stellt IFERR-THEN-ELSE-END dar. Hier ist es möglich, zusätzlich Befehle nur auszuführen, wenn zwischen dem IFERR- und dem THEN-Befehl kein Fehler auftrat.

Als Ergänzung zu unserem vorhergehenden Beispiel bei der Erklärung der IFERR-THEN-END-Sequenz wird man noch die frei gewählte Meldung "KEIN FEHLER" erwarten, wenn kein Fehler auftrat.

Lösung: ≪

```
  IFERR 5 *
  THEN CLEAR
"FALSCHER TYP"
  ELSE
"KEIN FEHLER"
  END
```

≫

Dieses Programm sollte ebenfalls mit einer Zeichenkette und einer Zahl getestet werden.

5.3.1.3 Die mehrseitige Verzweigung

Der HP48 verfügt über eine weitere Verzweigungsstruktur. Durch CASE-THEN-END-END läßt sich eine größere Anzahl von Verzweigungen in einem Stück verwirklichen. Verwendet wird dies als

```
CASE
  Bedingung 1 THEN Wahr-Sequenz 1 END
  Bedingung 2 THEN Wahr-Sequenz 2 END
  Bedingung 3 THEN Wahr-Sequenz 3 END
  ⋮
  Bedingung N THEN Wahr-Sequenz N END
  Alternativsequenz
END
```

Die Besonderheit besteht hier darin, daß nacheinander die Bedingungen getestet werden, bis eine erfüllt ist und die entsprechende *Wahr-Sequenz* ausgeführt wird. War eine Bedingung erfüllt, so wird die ***gesamte*** CASE-END-Struktur nach Ausführung der Wahr-Sequenz verlassen. War hingegen keine Bedingung erfüllt, so kann (muß aber nicht) eine *Alternativsequenz* ausgeführt werden.

Beispiel soll hier ein Programm sein, das ein Objekt vom Stack in der globalen Variable 'X' speichert und dann überprüft, ob in X eine 1, 2 oder 3 gespeichert wurde.

War dies der Fall, so wird die Zahl als String ausgegeben. Ist dem nicht so, wird das Wort "UNBEKANNT" im Stack abgelegt.

Lösung:
```
≪ 'X' STO
  CASE X 1 SAME
    THEN "EINS"
    END X 2 SAME
    THEN "ZWEI"
    END X 3 SAME
    THEN "DREI"
    END "UNBEKANNT"
  END
≫
```

Auch dieses Programm sollte mit verschiedenen Eingaben getestet werden.

5.3.2 Übungsaufgaben zu den Verzweigungsstrukturen

Auch hier sollen wieder Übungsaufgaben der Vertiefung dienen. Die Musterlösungen sind auf der Diskette unter VZWAXX zu finden, wobei hier XX wieder die zweistellige Aufgabennummer ist.

Aufgabe 1:

Es soll eine Password-Abfrage programmiert werden. Das Programm soll einen String vom Stack nehmen und ihn mit einem frei wählbaren (vierstelligen) Code vergleichen. Wurde der richtige Code eingegeben, so soll die Meldung "OKAY" ausgegeben werden. Ist dies nicht der Fall, soll der Rechner 2 Sekunden mit einer Frequenz von 1000 Hertz pfeifen (Befehl:BEEP) und das Wort "FALSCH" ausgeben.

Aufgabe 2:

Gesucht ist ein Programm, das an ein Objekt im Stack eine Fünf heranmultipliziert, falls dies erlaubt ist. Hierzu soll eine Abfrage der TYPEs vorgenommen werden, mit dem geprüft wird, ob dies erlaubt ist. Es soll keine CASE-END-Struktur verwendet werden. Es soll hier nur Bezug auf die TYPEs des HP28 genommen werden! Die Liste der TYPEs befindet sich im Anhang.

Aufgabe 3:

Es soll ein Programm geschrieben werden, welches eine Zahl daraufhin prüft, ob sie negativ, positiv oder Null ist. Das Ergebnis soll als Wort im Stack abgelegt werden (Lösung mit IF-THEN-(ELSE)-END-Struktur).

Aufgabe 4:

Es soll ein Programm geschrieben werden, das ein Objekt als Zahl von Eins bis Vier im Stack erkennt und entsprechend das Zahlwort im Stack ablegt. Ist das Objekt mit diesen vier Möglichkeiten nicht zu beschreiben, so soll das Wort "UNBEKANNT" in den Stack geschrieben werden. (Nur für HP48 — CASE-END-Struktur)

5.3.3 Musterlösungen der „Übungsaufgaben zu den Verzweigungsstrukturen“

Lösung der Aufgabe 1:

Musterlösung:
```
≪
  IF "CODE" SAME
  THEN "OKAY"
  ELSE 1000 2 BEEP
"FALSCH"
  END
≫
```

Speicherbedarf: 82.5 Bytes (HP28)
78.5 Bytes (HP48)

Dies war recht einfach. Es gibt zwei Alternativen. Damit reicht also eine IF-THEN-ELSE-END-Konstruktion.

Zwischen IF und THEN wird getestet, ob der schon im Stack stehende String identisch mit dem "CODE" ist. Ist dies der Fall, so bekommt das THEN eine Eins und führt den THEN-Block aus. Andernfalls wird der ELSE-Block ausgeführt.

Lösung der Aufgabe 2:

Musterlösung:
```
≪ DUP
  IF TYPE DUP 2 ==
OVER 5 == ROT 8 ==
OR OR NOT
  THEN 5 *
  END
≫
```

Speicherbedarf: 73 Bytes (HP28)
68 Bytes (HP48)

Man sollte sich hier erst einmal eine Aufstellung machen, welche TYPEs erlaubt sind und welche nicht (siehe Tabelle 5–1).

nicht erlaubt		**erlaubt**	
Objekt	TYPE	Objekt	TYPE
String	2	Zahl	0
Liste	5	komplexe Zahl	1
Programm	8	reeller Vektor/Matrix	3
		komplexer Vektor/Matrix	4
		Name	6
		lokaler Name	7
		algebraischer Ausdruck	9
		Binärzahl	10

Tabelle 5–1: Erlaubte TYPEs zur Aufgabenstellung 2

Man erkennt sofort, daß es nur drei nicht erlaubte Objekttypen gibt. Folgerichtig sollten wir lieber prüfen, ob ein nicht erlaubter TYPE vorliegt, da hier weniger Fälle auftreten. Wir drehen deshalb die Aussage logisch um: Wenn kein nicht erlaubter TYPE vorliegt, darf die Operation ausgeführt werden.

Nach Duplizierung bestimmt man den TYPE des Objekts. Da aber drei Tests durchgeführt werden, wird diese Kennzahl erst nochmals dupliziert zu:

Ebene 3	**Ebene 2**	**Ebene 1**
OBJEKT	TYPE	TYPE

(TYPE steht hier für die Kennzahl des TYPEs. Das „TYPE=X?“ steht für eine Eins, falls die Bedingung erfüllt ist, sonst für eine Null.)

Nun testen wir der Reihe nach die TYPEs ab. Mit 2 == entsteht

Ebene 3	**Ebene 2**	**Ebene 1**
OBJEKT	TYPE	TYPE=2?

Mit OVER 5 == bekommen wir

Ebene 4	Ebene 3	Ebene 2	Ebene 1
OBJEKT	TYPE	TYPE=2?	TYPE=5?

Mit ROT 8 == entsteht

Ebene 4	Ebene 3	Ebene 2	Ebene 1
OBJEKT	TYPE=2?	TYPE=5?	TYPE=8?

Wenn keine der unteren drei Ebenen eine Eins enthält, ist die Operation erlaubt. Mit OR OR wird getestet, ob eine der drei Ebenen eine Eins enthält. Ist dies der Fall, so würde nun eine Eins in Stackebene 1 stehen. Ist dort eine Eins, so darf das Fünf-Heranmultiplizieren nicht ausgeführt werden. Deshalb müssen wir nun mit NOT die Aussage des *kein nicht erlaubter TYPE* vervollständigen.

Ist nun die Abfrage positiv ausgefallen und die Operation erlaubt, so wird das 5 * ausgeführt. War dies nicht der Fall, bleibt das zu Anfang duplizierte Objekt allein im Stack stehen.

Lösung der Aufgabe 3:

Musterlösung:

```
≪ DUP
  IF 0 ==
  THEN DROP "NULL"
  ELSE
    IF 0 >
    THEN "POSITIV"
    ELSE "NEGATIV"
    END
  END
≫
```

Speicherbedarf: 103.5 Bytes (HP28)
93.5 Bytes (HP48)

Die wenigsten dürften diese Lösung in dieser kurzen Form gefunden haben. Das Hauptproblem ist hierbei der sinnvollste Umgang mit der eingegebenen Zahl.

Sicherlich am einfachsten gestaltet sich das Problem, wenn man die Zahl in einer Variable speichert und diese dann bei jeder Bedingungsabfrage beliebig zur Verfügung steht. Ein Vergleich des Speicherverbrauchs mit meiner Lösung dürfte wohl jeden Leser überzeugen. Zum besseren Verständnis sei deshalb auch das Struktogramm angegeben.

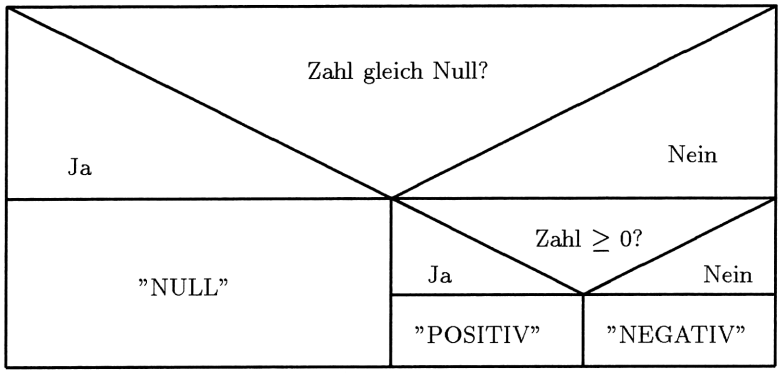


Abb. 5–6: *Struktogramm zu Aufgabe 3*

Nun beleuchten wir erst einmal die Vorgehensweise. Festzustellen ist, daß wir drei mögliche Lösungen haben:

- NULL
- POSITIV
- NEGATIV

Aus diesem Grunde werden wir mit einer einfachen IF-THEN-ELSE-END-Sequenz nicht auskommen, da diese nur zwei Möglichkeiten offen läßt. Da wir durch die zwei IFs auch zwei Bedingungen haben, brauchen wir unsere Zahl also logischerweise doppelt, da sie beim Test selbst verloren geht. Dafür steht das DUP gleich zu Beginn.

Der nächste Schritt ist fast beliebig. Ich entschied mich, die Zahl erst auf Null zu prüfen, deshalb das `IF 0 == THEN...` .

Jetzt wird es schwerer verständlich. Das DROP nach dem THEN wird benötigt, um die nach erfolgreichem Test nun überflüssige (weil bekannte) Zahl wieder zu entfernen. Tut man dies nicht, so bleibt sie einfach stehen. Der Ausdruck "NULL" braucht nicht weiter erklärt zu werden. Ist die Zahl ungleich Null, so wird der ELSE-Block der ersten Verzweigung ausgeführt. Als nächstes wird geprüft, ob die Zahl größer Null ist. Ist dies der Fall, so wird "POSITIV" im Stack abgelegt. Ist sie allerdings nicht positiv, so ist sicher, daß sie negativ sein muß. Also kann der zweite ELSE-Block ohne IF-Abfrage gleich "NEGATIV" in den Stack schreiben.

Eine Bemerkung noch zur Schachtelung der IFs: Etliche, die nach einer eigenen Lösung gesucht haben, werden vermutlich die Abfragen hintereinander gesetzt haben. Den Effekt möchte ich kurz erläutern. Nehmen wir an, der Rechner prüft erst die Zahl daraufhin ab, ob sie Null ist. Ist sie dies, so wird "NULL" in den Stack geschrieben. Nach dem END wird entweder getestet, ob die Zahl positiv (größer Null) oder negativ (kleiner Null) ist. Gestaltet man das letzte IF mit einer ELSE-Anweisung, so wird bei

der Eingabe von Null auch wieder ein Ergebnis ausgegeben, welches dann natürlich nicht stimmt.

Mit minimalstem Aufwand läßt sich dieses Problem nur lösen, wenn man soweit Schachtelungen vornimmt, bis nur noch zwei Alternativen übrig bleiben.

Lösung der Aufgabe 4:

Musterlösung: ≪ Speicherbedarf: 130.5 Bytes

```
CASE DUP 1 SAME
  THEN "EINS"
  END DUP 2 SAME
  THEN "ZWEI"
  END DUP 3 SAME
  THEN "DREI"
  END DUP 4 SAME
  THEN "VIER"
  END "UNBEKANNT"
END SWAP DROP
```

≫

Die CASE-END-Struktur war in der Aufgabenstellung schon vorgegeben. Die Grundstruktur ist recht einfach, dennoch muß einiges beachtet werden. Vor jedem Test muß das Objekt dupliziert werden, um nicht verloren zu gehen. Sollte ein Test positiv ausfallen, wird das entsprechende Wort im Stack abgelegt und die CASE-END-Struktur verlassen. Nun steht aber noch das Objekt in Ebene 2 des Stacks. Dieses entfernen wir durch das SWAP DROP. Sollten alle vier Tests fehlschlagen, wird das Wort "UNBEKANNT" abgelegt. Das Programm läßt sich auch wie folgt darstellen:

≪ Speicherbedarf: 145.5 Bytes

```
  CASE DUP 1 SAME
    THEN DROP
"EINS"
    END DUP 2 SAME
    THEN DROP
"ZWEI"
    END DUP 3 SAME
    THEN DROP
"DREI"
    END 4 SAME
    THEN "VIER"
    END "UNBEKANNT"
  END
```

≫

Es braucht allerdings 15 Bytes mehr und ist somit weniger günstig. In diesem Fall wäre das Programm in der zweiten Version allerdings einige Hundertstel Sekunden schneller, da das SWAP nicht vorhanden ist und damit auch nicht ausgeführt werden braucht. Dies ist also ein Gegenbeispiel dafür, daß kürzere Programme immer schneller laufen.

5.4 Gemischte Aufgaben

Nun kennen wir alle mehr oder weniger notwendigen Befehle, um nahezu beliebige Probleme zu lösen. Der Anfänger mag sich vielleicht angesichts der immer noch großen Zahl an Befehlen noch etwas erschlagen fühlen. Solange man aber Schritt für Schritt Probleme löst, werden sich nach gewissenhaftem Durcharbeiten der Aufgaben doch schon bald Erfolge zeigen. Insbesondere nach dem Lesen der „Tips & Tricks“ wird man Appetit auf das Programmieren verspüren. Mir selbst erging es seinerzeit nach dem Lesen der Handbücher meines HP28S genauso. Nach zwei Tagen schrieb ich meine ersten Programme. Mittlerweile existieren von den Programmen oftmals die vierten oder fünften Versionen, was von einer Entwicklung meiner Programmierfähigkeiten in RPL und meinem Eifer zeugt. Der Taschenrechner wurde zu meinem Hobby.

In diesem Kapitel ist eine große Anzahl an etwas komplexeren Aufgaben zusammengestellt. Oftmals werde ich Befehle des Taschenrechners als Programmieraufgabe stellen, bei denen gezeigt werden soll, wie simpel manche Befehle zu realisieren sind, ohne daß der eigentliche Befehl benutzt wird.

Die Musterlösungen findet man auf der Diskette unter GEMAXX, wobei XX wieder die zweistellige Nummer der Aufgabe ist.

Aufgabe 1:

Es soll eine alternierende Summe programmiert werden, deren Länge frei wählbar sein soll.

Die Formel der alternierenden Summe ist: $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \ldots \pm \frac{1}{x}$

Für x sollen Werte von 1 bis unendlich zugelassen werden. Der Wert für x soll bei Programmstart in Ebene 1 des Stacks stehen.

Aufgabe 2:

Es soll ein Programm geschrieben werden, das die Reihenfolge der Objekte in einer Liste genau umdreht. Das Programm soll also die selbe Funktion wie der Befehl REVLIST (HP48G/GX) haben, ohne den Befehl zu nutzen.

Eine leere Liste ist als Eingabe nicht zugelassen und braucht im Programm auch nicht beachtet werden.

Aufgabe 3:

Ein Programm, das die Objekte einer Liste nach der Wertigkeit ihres TYPEs sortiert, wird gesucht.

Das Programm soll die TYPEs der Objekte bestimmen und die Reihenfolge so neu ordnen, daß die Objekte mit „höherem TYPE-Wert" weiter hinten in der Liste zu stehen kommen.

Elemente gleichen TYPEs können in beliebiger Reihenfolge angeordnet werden.

Aufgabe 4:

Es soll ein Programm geschrieben werden, welches eine Dezimalzahl in Stunden, Minuten und Sekunden umrechnet. Der Vorkommaanteil der Zahl soll die Stunden, der Nachkommaanteil den dezimalen Bruchteil in Stunden darstellen (5.5 = 5 Stunden 30 Minuten).

Das Ergebnis soll als Liste mit 3 Zahlen im Stack abgelegt werden.

Beispiel: 5.5 $\Longrightarrow$ { 5 30 0 }

Die Uhrzeit soll auf eine Zeit zwischen 0 und 24 Uhr normiert werden, falls die Stunden mehr als 24 betragen.

Der Befehl →HMS darf nicht verwendet werden.

Aufgabe 5:

Es soll die quadratische Gleichung gelöst werden.

Die Gleichung $ax^2 + bx + c$ steht dem Programm als { a b c } zur Verfügung. Der Koeffizient a soll hierbei als ungleich Null angenommen werden (also keine Fallunterscheidung), aber es sollen beide Lösungen $x_{1/2}$ bestimmt werden.

Als Hilfe sei hier noch eine Umformung angegeben, die ein effektives Programmieren wesentlich erleichtert.

$$x_{1/2} = \tfrac{1}{2a}(-b \pm \sqrt{b^2 - 4ac}) \qquad = -\tfrac{b}{2a} \pm \sqrt{\tfrac{b^2-4ac}{4a^2}}$$

$$= -\tfrac{b}{2a} \pm \sqrt{(\tfrac{b}{2a})^2 - \tfrac{c}{a}} \qquad = -\tfrac{b}{2a} \pm \sqrt{(-\tfrac{b}{2a})^2 - \tfrac{c}{a}}$$

Aufgabe 6:

Es soll eine Erweiterung zur Lösung der quadratischen Gleichung geschrieben werden. Ausgehend vom alten Programm, nennen wir es QUA, soll eine Version geschrieben werden, die QUA als Unterprogramm nutzt und den Spezialfall beachtet, daß bei $ax^2 + bx + c$ der Koeffizient a Null sein kann.

Als Eingabe sei hier wieder eine Liste zu erwarten, diesmal aber mit zwei oder drei Werten. Da auch { 0 b c } zugelassen ist, muß auch dieses erkannt und die korrekte Lösung der linearen Gleichung ausgegeben werden. Bei zwei Werten in der Liste soll der Rechner ebenfalls die lineare Gleichung lösen.

Aufgabe 7:

Es soll ein Programm geschrieben werden, das einen String beliebiger Länge mit lauter Nullen schreibt.

Es soll angenommen werden, daß das Argument des Programms (also die Länge des Strings) eine ganze Zahl größer gleich Null ist.

Beispiel: 0 $\Longrightarrow$ ""
5 $\Longrightarrow$ "00000"

Aufgabe 8:

Ein Programm, das eine Liste mit x Nullen erzeugt, wird gesucht.

Als x darf eine Integer-Zahl von Null bis unendlich angenommen werden, d.h. für x=0 soll eine leere Liste geliefert werden.

Aufgabe 9:

Ausgehend vom vorherigen Programm, das den Namen LI (für Liste) bekommen soll, ist ein Programm zu schreiben, das die Länge zweier Listen überprüft und die kürzere von vorn mit Nullen auffüllt. Hierbei soll die Reihenfolge der Listen im Stack nicht verändert werden.

Achtung! Es wurde nichts darüber ausgesagt, welche der beiden die längere Liste ist. Dies muß das Programm selbst erkennen. Bei gleich langen Listen soll nichts verändert werden.

Das Programm LI läßt sich hier hervorragend nutzen.

Aufgabe 10:

Es soll das Skalarprodukt im n-dimensionalen Raum programmiert werden. Die Vektoren sollen in Listen geschrieben sein.

Also wird für $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ dann einfach { 1 2 3 } eingegeben.

Das Programm soll selbsttätig die Dimension der Vektoren erkennen und elementweise die Werte ausmultiplizieren und dann aufsummieren.

Beispiel: $\begin{pmatrix} A \\ B \\ C \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \quad 'A + 2 * B + 3 * C'$

Aufgabe 11:

Es soll ein Programm entwickelt werden, das das Löschen von mehreren Programmen bzw. Variablen komfortabel gestaltet.

Diese Routine wird eine abgemagerte Version der Verwendung sogenannter Meta-Zeichen, wie sie zum Beispiel unter DOS oder UNIX verwendet werden. Es soll zum Beispiel mit der Eingabe von "HA" und dem Aufruf des Programms erreicht werden, daß dann alle Programme gelöscht werden, die mit HA beginnen.

Die Anzahl der Buchstaben ist bei der Eingabe frei wählbar, mindestens ein Buchstabe ist aber vorgeschrieben.

Hierfür soll ein Befehl erklärt werden, der hierzu gebraucht wird. Die Anweisung heißt VARS und gibt in einer Liste sämtliche Namen von Programmen bzw. Variablen des aktuellen Verzeichnisses an.

Aufgabe 12:

Es soll ein Programm geschrieben werden, das alle Elemente einer Liste aufsummiert. Zusätzlich soll aber bei einer leeren Liste die Aufsummation eine Null ergeben. Der Befehl $\sum$LIST des HP48G/GX soll nicht benutzt werden.

5.5 Musterlösungen der „Gemischten Aufgaben“

Lösung der Aufgabe 1:

Musterlösung: ≪ 0 1 ROT
FOR E E INV -1 E
^ * -
NEXT
≫

Speicherbedarf: 54 Bytes

Hier sind wieder viele Tricks des vorteilhaften Rechnens versteckt. Der Rechner nimmt den Wert für x vom Stack und ist somit in der Lage, mit einer FOR-NEXT-Schleife (den Anfangs- und Endwert hat er ja nun) die Reihe zu berechnen.

Wie schon bei früheren Aufgaben summieren wir hier einfach auf. Deshalb benötigen wir einen Startwert für die Reihe, auf den die einzelnen Summanden addiert werden können. Durch das Schreiben der Null haben wir nun

2:	x
1:	0

.

Die Schleife läuft von 1 bis x. Hierzu muß die 1 in Ebene 2 und die Zahl x in Ebene 1 des Stacks stehen. Deshalb schreiben wir erst die 1 in den Stack:

Ebene 3	Ebene 2	Ebene 1
x	0	1

Nun bringen wir mit ROT die Objekte in die richtige Reihenfolge. (Ich hoffe, es ist zu erkennen, weshalb ich zuerst die Null in den Stack schrieb!).

Die Sequenz FOR E nimmt nun die Argumente vom Stack und es bleibt unsere „Summations“-Null zurück.

Die Formel läßt sich sicherlich als $\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \ldots \pm \frac{1}{x}$ schreiben. Somit ist der Anfangswert Eins kein Sonderfall.

Achtung! Was auf keinen Fall passieren darf, ist, daß man die Eins statt der Null als Startwert nimmt und die Schleife von 2 bis x laufen läßt.

In der Aufgabenstellung ist zu lesen, daß x Eins sein darf. Eine Schleife, deren Anfangsargument größer als ihr Endwertargument ist, wird trotzdem immer einmal durchlaufen (siehe Seite 76). Ein falsches Ergebnis wäre die Folge.

Die einzelnen Werte bekommt man sehr einfach durch E INV.

Nun brauchen wir uns nur noch Gedanken darüber machen, wie man das wechselnde Vorzeichen realisiert.

Hierzu soll uns ein alter Programmiertrick helfen: Der Ausdruck −1 E ^ liefert bei laufendem E mal eine 1 (gerades E), mal eine −1 (ungerades E). Genau dieses

wechselnde Vorzeichen können wir hier gebrauchen. Wenn wir das Ergebnis dieser Operation heranmultiplizieren, so hat jeder Summand genau das *falsche* Vorzeichen. Dies könnte wir durch −1 E 1 + ^ ändern. Nun wäre das heranmultiplizierte Ergebnis genau der gesuchte Faktor. Aber dies wäre dumm, da erstens sehr häufig zusätzlich die Operation 1 + ausgeführt werden müßte (genau x mal) und zweitens zwei Operationen zusätzlich gebraucht würden (Speicherbedarf: 5 Bytes). Einfach zu umgehen ist zumindest das x-malige Ausführen von 1 +, indem man nach Berechnung der alternierenden Summe das falsche Vorzeichen ganz einfach mit −1 * ausgleicht. Man berechnet also einfach

$$-1 * (-\tfrac{1}{1} + \tfrac{1}{2} - \tfrac{1}{3} + \tfrac{1}{4} - \tfrac{1}{5} \ldots \mp \tfrac{1}{x})$$

Aber auch dies ist nicht die optimale Lösung. Viel einfacher läßt es sich doch realisieren, wenn die Formel als

$$0 - (-\tfrac{1}{1}) - (+\tfrac{1}{2}) - (-\tfrac{1}{3}) - (+\tfrac{1}{4}) - (-\tfrac{1}{5}) \ldots - (\mp\tfrac{1}{x})$$

geschrieben wird. Wir bilden also jeweils die Differenz. Die Werte in den Klammern haben wir ja schon durch das E INV −1 E ^ * erhalten.

Lösung der Aufgabe 2:

Musterlösung: Speicherbedarf: 52 Bytes

```
≪ DUP SIZE 1
  FOR E DUP E GET
SWAP -1
  STEP SIZE →LIST
≫
```

Wie geht man so ein Programm an? Es gibt immer eine Unzahl von Möglichkeiten. Das macht es immer schwer, den ersten Schritt zu erklären. Meine erste Version dieses Problems hatte 57 Bytes. Das sollte ein Trost für die sein, die vielleicht 60 oder gar 70 Bytes für die Lösung des Problems brauchten.

Am vorteilhaftesten ist die Überlegung, welche Schleife wohl am ehesten unser Problem lösen könnte. Wir müssen, beginnend beim letzten Objekt der Liste, ein Element nach dem anderen aus der Liste entnehmen. Dies entspricht einer rückwärts zählenden Schleife (FOR-(−1 STEP)-Schleife).

Wie bekommen wir die Elemente in eine Liste? Sehr einfach: x →LIST bringt x Objekte in eine Liste, wobei das in Ebene x stehende Element das erste Objekt der Liste wird. Wir brauchen also bloß die Objekte der Liste in umgekehrter Reihenfolge in den Stack legen und am Schluß x →LIST durchführen. Die Anzahl der Elemente der alten und der neuen Liste ist identisch.

Zurück zur Schleife. Die Argumente der Schleife sind recht klar. Startwert ist die Länge der Liste (SIZE) und Endwert die Eins für die Entnahme des ersten Objekts der ursprünglichen Liste. Um aber unsere Liste nicht zu verlieren, werden wir sie mit DUP erst duplizieren.

Mit dem anschließenden SIZE 1 erhalten wir:

Ebene 3	Ebene 2	Ebene 1
Liste	Länge d. Liste	1

Das FOR E nimmt die Argumente vom Stack. Jetzt steht nur noch die (ursprüngliche) Liste im Stack. Da diese für spätere Entnahmen gesichert bleiben muß, duplizieren wir sie wieder.

Als nächstes entnehmen wir mit E GET das E-te Element

2:	Liste
1:	E-tes Element

.

Beim nächsten Schleifendurchlauf muß die Liste wieder in Ebene 1 stehen.

Wir tauschen mit SWAP zu

2:	E-tes Element
1:	Liste

.

Bei der nächsten Entnahme aus der Liste wird das (E−1)-te Element entnommen und genauso behandelt, bis nach Ende der Schleife alle Elemente der Liste (in umgekehrter Reihenfolge) und die ursprüngliche Liste selbst im Stack stehen. Die SIZE der ursprünglichen Liste ist die der neuen. Wir führen also mit SIZE →LIST die beiden letzten Schritte durch.

Lösung der Aufgabe 3:

Musterlösung:

```
≪ 0 10
  FOR E 1 OVER SIZE
    FOR F DUP F GET
DUP
      IF TYPE E
SAME
      THEN SWAP
      ELSE DROP
      END
    NEXT
  NEXT SIZE →LIST
≫
```

Speicherbedarf: 101.5 Bytes (HP28)
96.5 Bytes (HP48)

Ein Struktogramm soll der Verdeutlichung dienen:

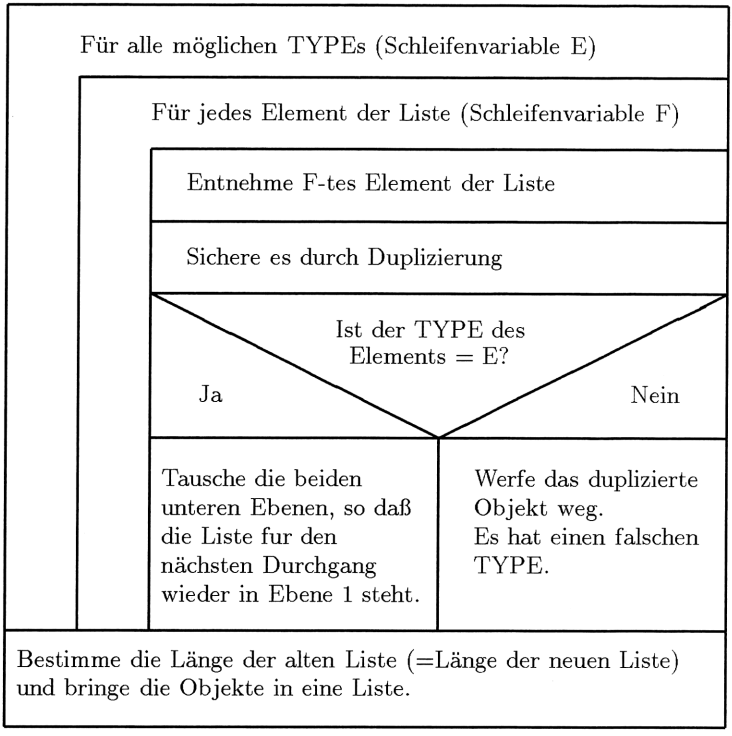


Abb. 5–7: *Struktogramm zur Aufgabe 3*

Die Aufgabenstellung hört sich recht kompliziert an. Nach Zerlegung in kleinere Probleme bleibt aber nichts schwieriges übrig.

Die Top-Down-Methode ergibt eine Zusammensetzung zweier geschachtelter Schleifen. Die äußere Schleife führt ein Zählen der aktuellen TYPE-Zahl durch. Die innere Schleife prüft jeweils alle Elemente der Liste darauf, ob sie dem aktuellen TYPE entsprechen. Ist dies der Fall, so wird das gerade gewählte Objekt im Stack abgelegt.

Die Bedeutung der Schleifen geht aus obiger Erklärung und dem Struktogramm klar hervor. Betrachten wir nun die restlichen Operationen. In der inneren Schleife prüfen wir Objekt für Objekt. Hierzu entnehmen wir das laufende Element der Liste, sichern es durch ein DUP und prüfen den TYPE. Ist dieser gleich E (also gleich dem aktuellen TYPE aus der äußeren Schleife), so tauschen wir lediglich die Ebenen 1 und 2. Der Grund wird klar, wenn wir uns überlegen, daß das (zuvor gesicherte) Objekt nun um eine Ebene nach oben gerutscht ist und nun die Original-Liste für den nächsten Schleifendurchlauf in Ebene 1 steht. Beim nächsten Durchlauf (und allen späteren) wird nicht mehr auf Elemente des Stacks zugegriffen, die oberhalb der Liste stehen. Diese rutschen mit jedem neu erkannten und sortierten Objekt immer eine Ebene nach oben.

Fiel der Test negativ aus, so wird das Element einfach weggeworfen. Dies geschieht nun Element für Element und TYPE für TYPE. Am Schluß stehen die nun sortierten Objekte im Stack und in Stackebene 1 die „alte" Original-Liste. Diese gebrauchen wir jetzt noch als Lieferanten der Anzahl der Elemente für die neue Liste. Der letzte Befehl sorgt noch dafür, daß die neue Liste gebildet wird.

Lösung der Aufgabe 4:

Musterlösung: `≪ DUP IP 24 MOD SWAP FP 60 * DUP IP SWAP FP 60 * 3 →LIST ≫`

Speicherbedarf: 79.5 Bytes

Hier brauchen wir lediglich die üblichen Stackverschiebungs- und -duplizierungsbefehle, sowie einige mathematische Befehle.

Zuerst duplizieren wir (wie fast immer) unsere Zahl:

2:	Zahl
1:	Zahl

Da die Stunden vor dem Komma stehen, wird der ganzzahlige Anteil mit IP genommen. Da wir auf eine 24-Stunden-Uhr normieren sollen, führen wir 24 MOD aus. Dies erfüllt genau diesen Zweck.

Im Stack steht nun:

2:	Zahl
1:	Stunden

Nun tauschen wir die Ebenen Eins und Zwei und nehmen mit FP den Nachkommaanteil der Zahl — der Rest wurde ja schon verarbeitet. Um die Minuten zu bekommen, multiplizieren wir mit 60. Der Vorkommaanteil ist nun die gesuchte Anzahl der Minuten. Da der Nachkommaanteil aber auch die Sekunden beinhaltet, müssen wir erst die Zahl duplizieren, bevor wir mit IP die Anzahl der Minuten bilden

Ebene 3	**Ebene 2**	**Ebene 1**
Stunden	Min.Sek	Minuten

Diese schieben wir wieder mit SWAP weg

Ebene 3	**Ebene 2**	**Ebene 1**
Stunden	Minuten	Min.Sek

Da der Nachkommaanteil nur noch interessiert, bilden wir diesen mit FP. Um die Anzahl der Sekunden zu bekommen (im Moment sind es noch dezimale Bruchteile von Minuten), multiplizieren wir abermals mit 60. Nun steht in der Regel immer noch ein Nachkommaanteil bei den Sekunden. Hierüber wurde in der Aufgabenstellung nichts ausgesagt. Wir lassen ihn also als Bruchteil von Sekunden stehen.

Als letztes faßt man mit 3 →LIST die Werte in einer Liste zusammen. Die Probe fällt also wie folgt aus: 16.011968 sind 16 Stunden 0 Minuten 43.0848 Sekunden.

Lösung der Aufgabe 5:

Musterlösung: `≪ LIST→ PICK / SWAP ROT 2 * / NEG DUP SQ ROT - √ DUP2 - SWAP ROT + ≫` Speicherbedarf: 63 Bytes

Dieses Problem ist recht knifflig. Ich habe deshalb schon die optimal programmierbare Formel angegeben. Entscheidend hierbei war die Reduktion der zu berechnenden Ausdrücke. Das $(-\frac{b}{2a})$ braucht, wie schon früher erklärt, dann nur noch einmal berechnet zu werden und kann als ganzer Ausdruck dupliziert werden.

Schauen wir die Formel genauer an. Die beiden Werte von b und c werden jeweils, mit obiger Prämisse, ein einziges Mal verwendet, das a zweimal.
Da die drei Zahlen in der Liste stehen, aber von ihr extrahiert benötigt werden, lösen wir die Liste mit LIST→ auf:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
a	b	c	3

Jetzt heißt es clever sein! Die Zahl 3 stört eigentlich und müßte gelöscht werden. Besonders schlau ist aber, wenn man sie gleich mitverwenden kann. Als Faktor wird sie nicht gebraucht, aber eine andere Möglichkeit tut sich hier auf. Wir brauchen doch das a doppelt. Würde nur

Ebene 3	Ebene 2	Ebene 1
a	b	c

im Stack stehen, so würden wir a mit 3 PICK holen. Die hierfür nötige 3 ist aber schon da. Nach dem PICK sieht der Stack so aus:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
a	b	c	a

Jetzt sollten wir dies nutzen, um $\frac{c}{a}$ mit einer Division zu bilden,

Ebene 3	Ebene 2	Ebene 1
a	b	'c/a'

Ideal wäre nun ein Wechsel der Stackebenen bis b in Ebene 2 und a in Ebene 1 stehen bleibt. Dies liefert SWAP ROT zu

Ebene 3	Ebene 2	Ebene 1
'c/a'	b	a

Nun multiplizieren wir das a mit Zwei, dividieren und bekommen nach der Negation:

2:	'c/a'
1:	'$-(b/(a*2))$'

Diesen Ausdruck brauchen wir so, wie er dasteht, und nochmals quadriert.

Ein DUP SQ liefert:

Ebene 3	Ebene 2	Ebene 1
'c/a'	'$-(b/(a*2))$'	'$SQ(-(b/(a*2)))$'

Um die Diskriminante zu bekommen, führen wir ROT $-$ $\sqrt{}$ aus:

2:	'$-(b/(a*2))$'
1:	'$\sqrt{}(SQ(-(b/(a*2))) - c/a)$'

Da wir beide Lösungen der quadratischen Gleichung angeben sollen, muß man beide duplizieren. Auch hier gilt wieder ein früher schon erklärtes Prinzip: Summen lassen sich einfacher bilden. Nachdem die beiden untersten Ebenen ideal stehen, um die Subtraktion auszuführen, tun wir dies nach dem DUP2 mit dem Minus-Zeichen auch.

Den Abschluß bildet ein SWAP ROT + (oder auch ein 3 ROLLD +).

Lösung der Aufgabe 6:

Musterlösung:

```
≪ DUP SIZE
    IF 2 SAME OVER 1
  GET 0 SAME OR
    THEN LIST→
      IF 3 ==
      THEN ROT DROP
      END NEG SWAP /
    ELSE QUA
    END
≫
```

Speicherbedarf: 104.5 Bytes (HP28)
94.5 Bytes (HP48)

Ein Struktogramm soll uns wiederum helfen, einen Überblick zu gewinnen:

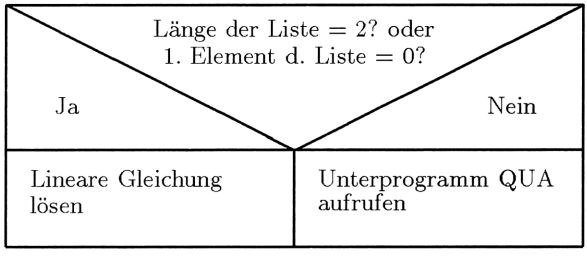


Abb. 5–8: *Struktogramm zu Aufgabe 6*

Hier muß zuerst einmal die Liste dupliziert werden, um sie überprüfen zu können. Ist die Länge (SIZE) Zwei, oder das erste Element eine Null, so haben wir eine lineare Gleichung.

Nach dem Kopieren der Liste bestimmen wir mit SIZE die Länge der Liste und fragen ab, ob sie zwei ist.

Im Stack steht nun:

2:	Liste
1:	Länge=2?

Die Notation Länge=2? ist nur ein Platzhalter für eine Eins, falls die Länge zwei ist, bzw. ein Platzhalter für eine Null, falls dies nicht der Fall ist.

Dann kopieren wir die Liste abermals mit OVER, entnehmen das erste Element und prüfen, ob es eine Null ist. Wenn der erste Fall oder der zweite zutrifft, lösen wir das Problem als lineare Gleichung. Sonst (ELSE) rufen wir einfach das Programm QUA auf.

Beim THEN-Block wissen wir nun nicht (mehr), welcher der beiden Fälle vorliegt, d.h. es können zwei oder drei Objekte in der Liste stehen. Deshalb führen wir LIST→ aus und bekommen entweder

4:	
3:	b
2:	c
1:	2

oder

4:	0
3:	b
2:	c
1:	3

Durch die Abfrage IF 3 == können wir herausfinden, ob die zusätzliche Null vorhanden ist und sie mit ROT DROP entfernen, falls dies der Fall ist.

Nun steht nur noch

2:	b
1:	c

im Stack.

Die Lösung $-\frac{c}{b}$ bekommen wir ganz einfach durch NEG SWAP /.

Lösung der Aufgabe 7:

Musterlösung: Speicherbedarf: 56.5 Bytes

```
≪ ""
  WHILE OVER
  REPEAT "0" + SWAP
1 - SWAP
  END SWAP DROP
≫
```

Die Art der hier zu wählenden Schleife hat eine abweisende Struktur. Hiervon kennt der Rechner bekanntermaßen lediglich die WHILE-REPEAT-END-Konstruktion.

In jedem Fall müssen wir einen String als Ergebnis abliefern. Deshalb setzen wir vor Beginn der Schleife einen leeren String in den Stack, der nun Schritt für Schritt erweitert wird, solange der Zähler größer Null ist.

2:	Zähler
1:	""

Ist nun der Zähler größer Null, dürfen wir eine Null an den String anhängen. Deshalb eröffnen wir nun die WHILE-Schleife. Solange nun der Zähler (durch Herunterholen

mit OVER) ungleich Null ist (kleiner Null kann er nicht werden, da wir bei Null abbrechen), dürfen wir eine Null an den String anhängend den Zähler um Eins erniedrigen.

Weshalb benötigen wir keinen Test, ob die Zahl größer Null ist? Der Grund liegt auf der Hand: Der Rechner interpretiert jede Zahl als *erfüllt* bzw. *wahr*, wenn sie ungleich Null ist. Sie kann aber nur größer Null sein (also *wahr*) oder Null. Ist sie Null, so interpretiert der Rechner dies als *unwahr* und wiederholt die REPEAT-END-Anweisungssequenz nicht mehr. In diesem Fall sparte man also das $0 \neq$, was dem Rechner absolut gleichgültig ist.

Nun dürfen wir aber nach positivem „Test“ den String erweitern und den Zähler erniedrigen. Dies geschieht mit "0" + SWAP 1 − SWAP. Das "0" + hängt eine Null an den String. Um den Zähler zu erniedrigen, müssen wir ihn mit SWAP herunterholen, ihn dekrementieren und wieder in die Ausgangsposition zurückschieben.

Dies geschieht solange, bis der Zähler Null ist. Der String hat die gewünschte Länge, aber der Zähler in Ebene Eins ist nun überflüssig und stört noch. Mit SWAP DROP verschwindet aber auch dieser.

Lösung der Aufgabe 8:

Musterlösung: Speicherbedarf: 51.5 Bytes

```
≪ DUP
  WHILE DUP
  REPEAT 1 - 0 3
ROLLD
  END DROP →LIST
≫
```

Hier haben wir wieder eine abweisende Schleife zu wählen, was an der Aufgabenstellung daran erkannt werden kann, daß für x=0 die leere Liste zurückgegeben werden soll.

Wie schon früher, können wir beim Test, ob die Schleifenvariable noch größer als Null ist, auf einen echten Test verzichten. Ist der (Schleifen-)Zähler größer Null, so wird fortgefahren. Ist dies nicht der Fall, so wird der Zähler, der ja nun Null ist, selbst als *unwahre* Bedingung erkannt.

Vermieden werden soll ein unnötiges Herumschieben der Objekte im Stack. In diesem Fall heißt das, daß man sogar auf das Mitführen einer Liste, die Schritt für Schritt gefüllt wird, verzichtet. Dies ist möglich, da mit dem Befehl →LIST beliebig viele Elemente mit einem Schlag in eine Liste übernommen werden können. Ein Gegenbeispiel stellen die Zeichenketten dar (siehe Aufgabe 7).

Aus dieser Überlegung heraus werden wir (wieder mal) unser Anfangsargument (also die Länge der Liste) kopieren, um es für spätere Zwecke zu verwenden. Diesen

Original-Zähler (OrgZähl) behalten wir zum aktuellen Zähler (aktZähl) im Stack:

2:	OrgZähl
1:	aktZähl

.

Ist nun der Zähler ungleich Null, es sind ja nur positive Zahlen zugelassen, so wird man eine Null in den Stack legen. Diese Null darf natürlich nicht stören und muß nach oben verschoben werden. Ebenfalls muß der Zähler um Eins erniedrigt werden. Da er in Ebene 1 steht, werden wir deshalb auch mit dem Dekrementieren beginnen.

Das 1 − vermindert unseren Zähler

2:	OrgZähl
1:	aktZähl-1

.

Nun schreiben wir die Null in den Stack:

Ebene 3	**Ebene 2**	**Ebene 1**
OrgZähl	aktZähl-1	0

Jetzt brauchen wir die Null nur noch wegschaffen. Was erwartet das Programm im späteren Ablauf? In der untersten Ebene muß sich der aktuelle Stand des Zählers befinden.

Wenn wir nun die Null nur mit SWAP tauschen, würde im Laufe des Programms folgendes passieren: Der Original-Zähler-Stand wandert mit jeder weiteren Null um eine Position nach oben. Springt der Rechner aus der Schleife, steht der Zähler in der höchsten Stackebene und unter ihm eine ganze Anzahl von Nullen. Dies ist natürlich Unfug.

Wenn wir stattdessen mit 3 ROLLD die Null(en) über den Original-Zähler schieben, so bleibt er während des Ablaufs immer in Ebene 2, d.h. wir können am Ende des Programms gezielt auf ihn zurückgreifen, da wir wissen, in welcher Ebene er sich befindet.

Was soll dieser Aufwand? Nun, wenn wir am Ende der Schleife zwar eine ganze Anzahl an Nullen im Stack haben, aber nicht wissen, wieviele es genau sind, können wir diese Nullen auch nicht in eine Liste bringen. Haben wir aber den ursprünglichen Zählerstand aufgehoben (dieser ist ja die Anzahl der Nullen), so können wir problemlos die Liste bilden.

Der auf Null gesunkene aktuelle Zählerstand ist wegzuwerfen (DROP). Nun stehen die Nullen im Stack und in der untersten Ebene die Anzahl der Nullen. Mit →LIST schreiben wir sie abschließend in eine Liste.

Lösung der Aufgabe 9:

Musterlösung: Speicherbedarf: 79.5 Bytes

```
≪ OVER SIZE OVER
SIZE DUP2 - ABS LI
SWAP ROT
  IF ≥
  THEN ROT + SWAP
  ELSE SWAP +
  END
≫
```

Wir brauchen hier, für spätere Zwecke, die Listen (ListeX) selbst, sowie die Länge beider Listen (Länge Liste X). Diese Aufgabe erledigt am schnellsten OVER SIZE OVER SIZE:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
Liste 1	Liste 2	Länge Liste 1	Länge Liste 2

Was brauchen wir noch? Das Programm LI liefert eine Liste der Länge x, die mit Nullen aufgefüllt ist. Diese Liste sollte in diesem Anwendungsfall exakt die Differenz der beiden Längen, also die Länge ABS(Länge Liste 1 − Länge Liste 2), haben. Prüfen wir also, ob wir die Länge im Verlauf des Programms noch brauchen. Wenn wir später bestimmen müssen, an welche Liste die von LI erzeugte Liste angefügt werden muß, benötigen wir die Länge der Liste nochmals, um zu ermitteln, an welche Liste die vorderen Nullen angehängt werden müssen. Also duplizieren wir erst mit DUP2 und bilden dann den Absolutbetrag der Differenz. Im Stack steht nun:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
Liste 1	Liste 2	Länge Liste 1	Länge Liste 2	ABS(Differenz)

Nun soll die Unterroutine LI die zu ergänzende Liste erzeugen (durch den Aufruf von LI):

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
Liste 1	Liste 2	Länge Liste 1	Länge Liste 2	LI-Liste

Jetzt müssen wir prüfen, welche der Listen die längere ist. Hierfür gibt es mehrere Möglichkeiten, um die beiden Längen der Listen herunterzuholen. Ich entschied mich für SWAP ROT (es ist ebenfalls 3 ROLLD oder ROT ROT möglich — die Testbedingung muß dem nur jeweils angepaßt werden):

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
Liste 1	Liste 2	LI-Liste	Länge Liste 1	Länge Liste 2

Mit IF ≥ wird nun geprüft, ob hier die Länge der Liste 2 größer oder gleich der Länge der Liste 1 ist. Ist dies der Fall, so wird mit ROT + SWAP die LI-Liste vor die Liste 1 gebracht und diese beiden verknüpft (ROT +). Um die korrekte Reihenfolge wiederherzustellen, wird ein Tausch der untersten beiden Ebenen vorgenommen (SWAP).

Ist der Test negativ ausgefallen, wird mit SWAP + die LI-Liste vor die Liste 2 gebracht und verknüpft. Die Reihenfolge ist hier schon korrekt und braucht nicht mehr verändert zu werden.

Lösung der Aufgabe 10:

Musterlösung:
```
≪ 0 1 3 PICK SIZE
  FOR E OVER E GET
4 PICK E GET * +
  NEXT ROT ROT
DROP2
≫
```

Speicherbedarf: 76.5 Bytes (HP28)
71.5 Bytes (HP48)

Hier sollen natürlich die Vektoren nicht erst elementweise multipliziert und später alle aufaddiert werden, sondern alles in einem Schritt mit einer Schleife erledigt werden.

Wir müssen, wie bei einer Reihe, aufsummieren, d.h. wir brauchen wieder eine Null im Stack als Start-Summationswert. Die Schleife geht von Eins bis zur Dimension der Vektoren. Zur Zeit steht im Stack:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
Vektor 1	Vektor 2	0	1

Aus Ebene 3 (oder 4) holen wir jetzt einen Vektor herunter und bestimmen seine Dimension durch SIZE. Das FOR E nimmt nun die Schleifen-Argumente vom Stack.

Ebene 3	Ebene 2	Ebene 1
Vektor 1	Vektor 2	0

Jetzt muß das E-te Element der beiden Vektoren miteinander multipliziert werden. Da Vektor 2 in Ebene 2 steht, wird wieder elegant mit OVER Vektor 2 kopiert. (Es ist ja nicht von Belang, welche Reihenfolge die Elemente bei der Multiplikation haben). Es wird nun das E-te Element (Element E(VekX)) mit E GET entnommen.

Ebene 4	Ebene 3	Ebene 2	Ebene 1
Vektor 1	Vektor 2	0	Element E(Vek2)

Nun kopieren wir mit 4 PICK den Vektor 1 herunter, entnehmen mit E GET das E-te Element und multiplizieren dieses mit dem E-ten Element des Vektors 2. Jetzt braucht dieser Wert nur noch aufsummiert werden (mit Plus).

Hätten wir zuerst den Vektor 1 geholt und erst dann den Vektor 2, so wäre ein Befehl mehr benötigt worden, da das OVER nicht mehr hätte eingesetzt werden können. Dies geschieht nun Element für Element der Vektoren.

Nach Ende der Schleife stehen noch immer die Vektoren in Ebene 1 und 2. Um diese zu entfernen, benutzt man ROT ROT (oder 3 ROLLD) und DROP2.

Lösung der Aufgabe 11:

Musterlösung:
```
≪ VARS 1 OVER SIZE
  FOR E DUP E GET
DUP →STR 4 PICK POS
    IF 2 ==
    THEN PURGE
    ELSE DROP
    END
  NEXT DROP2
≫
```

Speicherbedarf: 84.5 Bytes (HP28)
79.5 Bytes (HP48)

Wir beginnen mit dem Befehl VARS. Nun stehen unser String und die Variablenliste im Stack. Die Schleife dient dem Entnehmen von Elementen aus der Liste, ist also eine FOR-NEXT-Konstruktion. Die Argumente erhalten wir durch 1 OVER SIZE, die das FOR E dem Stack sofort wieder entnimmt. Da wir die Liste noch brauchen, kopieren wir sie, bevor wir mit E GET das E-te Element aus der Variablenliste (VARS-Liste) entnehmen.

Ebene 3	Ebene 2	Ebene 1
String	VARS-Liste	E-tes Element

Wenn wir dieses Element später aus dem Verzeichnis löschen müssen, so brauchen wir es nochmals. Nach dem Duplizieren haben wir:

Ebene 4	Ebene 3	Ebene 2	Ebene 1
String	VARS-Liste	E-tes Element	E-tes Element

Das E-te Element ist kein String. Deshalb verwandeln wir den Namen mit →STR in eine Zeichenkette. Nun stören die '-Zeichen etwas. Das braucht uns aber nicht zu kümmern. Wir kopieren den String aus Ebene 4 herunter, um den Vergleich zu ermöglichen:

Ebene 5	Ebene 4	Ebene 3	Ebene 2	Ebene 1
String	VARS-Liste	E-tes Element	„E-tes Element“	String

Durch POS wird nun bestimmt, ob der String der Ebene 1 im String der Ebene 2 enthalten ist. Ist dies der Fall, wird die Stelle der ersten Übereinstimmung zurückgegeben.

Betrachten wir ein Beispiel: Das E-te Element sei 'PAPI', der String also " 'PAPI' " und unser (Such-)String "PA". Bei POS wird eine 2 zurückgegeben. Wäre nun " 'PIPA' " in Ebene 2, wäre das Ergebnis von POS die Zahl 4. Letzteres ist kein zu löschendes Objekt.

Ist also das Ergebnis der Ausführung von POS eine 2 (mit IF 2 == getestet), so dürfen wir getrost PURGE zum Löschen aufrufen. Ansonsten darf das E-te Element nicht gelöscht werden, der Name wird also mit DROP weggeworfen. Dies wird Name für Name der Variablen-Liste durchgeführt. Zuallerletzt werden mit DROP2 noch der String und die Variablenliste gelöscht.

Lösung der Aufgabe 12:

Musterlösung: `≪ LIST→ 0 0 0 4` Speicherbedarf: 40 Bytes

```
ROLL
  START +
  NEXT
≫
```

Dies ist wieder eine sehr trickreiche Konstruktion. Wir stellen folgende Überlegungen an:

- Es langt eine START-NEXT-Schleife, da der Zähler nicht gebraucht wird.
- Wir benötigen eine Null zum Aufsummieren wie bei einer Reihe - diese Null können wir auch gleich benutzen, um die Null des Ergebnisses einer leeren Liste zu liefern.
- Wir müssen Sorge tragen, daß nicht zu viele Elemente addiert werden. (Drei Elemente erfordern lediglich zwei Additionen)

Die letzte Aussage erscheint trivial. Sie ist es aber nur in gewissen Rahmen, da wir mit möglichst wenig Aufwand an Befehlen operieren wollen.

Die kürzeste Version mutet in der Tat seltsam an. Als erstes löse ich die Liste auf. Das ist sicherlich das vernünftigste, da wir so die Länge der Liste bekommen und auch alle Objekte im Stack landen, wo wir sie einfach addieren können.

Nach LIST→ sieht der Stack exemplarisch so aus:

4:	a		4:			4:				4:					
3:	b		3:	a		3:				3:					
2:	c	oder	2:	b	oder	2:	a	oder		2:					
1:	3		1:	2		1:	1			1:	0				

Laut unserer Vorüberlegung dürfen wir nicht zu oft addieren. In unseren Beispielen sehen wir aber, daß die Länge der Liste immer um Eins länger ist, als Additionen nötig sind. Startwert müßte also die Zwei sein. Große Probleme treten aber bei den letzten beiden Beispielen auf:

- Bei einer Liste mit einem Element dürfte die START-NEXT-Schleife gar nicht durchlaufen werden.
- Bei einer leeren Liste müßte sogar die Null unangetastet bleiben.

Deshalb behelfen wir uns mit einem Trick: Wir liefern dem Rechner einfach ein paar Elemente, die er addieren darf. Dies können natürlich nur Nullen sein.

In Ebene 1 steht immer die (Anzahl − 1) der Additionen, darüber die zu addierenden Elemente. Ausnahme bildet die leere Liste, bei der keine Elemente über der Anzahl sind. Eine START-NEXT-Schleife wird mindestens einmal durchlaufen, d.h. auch eine Liste mit nur einem Element wird Probleme bereiten, da die Anzahl der

nötigen Additionen eigentlich Null wäre. Liefert man nun dem Stack zwei Nullen, so sind immer ausreichend Elemente vorhanden, die von der Schleife addiert werden können. Die Anzahl der Additionen steigt aber selbstverständlich ebenfalls um zwei an. Die Länge der eingegebenen Liste ist also um Eins niedriger als die Anzahl der nun nötigen Summenbildungen. Jetzt wäre es natürlich Unfug, die Schleife von Eins bis (Anzahl+1) laufen zu lassen, da die Schleife von Null bis Anzahl ebenso viele Durchläufe ergibt. Hierbei brauchen wir aber keine zusätzlichen Operationen anstellen, um die Schleifenargumente zu bekommen.

Fassen wir zusammen: Nach dem LIST→ brauchen wir zwei Nullen, die zusätzlich aufaddiert werden. Danach brauchen wir die Argumente der Schleife. Dies ist eine weitere Null, und mit 4 ROLL holen wir die Länge der Liste. Die Stacks würden nach LIST→ 0 0 0 4 ROLL wie folgt aussehen:

7:	a		7:			7:			7:			7:			
6:	b		6:	a		6:			6:			6:			
5:	c		5:	b		5:	a		5:			5:			
4:	0	oder	4:	0	oder	4:	b	oder	4:	0		4:	0		
3:	0		3:	0		3:	0		3:	0		3:	0		
2:	0		2:	0		2:	0		2:	0		2:	0		
1:	3		1:	2		1:	1		1:	0		1:	0		

Die untersten beiden Ebenen sind jeweils die Argumente der Schleife (Anzahl der Additionen).

6 Tips und Tricks

6.1 Einfache Tips für effektive Programme

Dieses Kapitel beinhaltet eine Anzahl von Hinweisen darauf, wie man bei Programmen Speicher sparen kann, ohne daß hierbei die geringsten Kenntnisse vorausgesetzt werden. Die Hinweise sind somit für jeden nutzbar. Sie sind insbesonders auch nachträglich auf Programme anwendbar, da sie nicht in ihre Struktur eingreifen.

Nichtsdestotrotz ist der Effekt überraschend groß, wie wir bei den einzelnen Punkten noch sehen werden.

6.1.1 Der IF-THEN-Tip

Mit einem ganz simplen Trick läßt sich unwahrscheinlich viel Speicher sparen. Leider funktioniert dieser Trick nur beim HP28, da der HP48 diesen Trick immer automatisch nutzt.
Wenn zwischen einem IF- und einem THEN-Befehl lediglich genau eine Anweisung (bzw. Operand etc.) steht, so minimiert dies den Speicherverbrauch enorm. Der Grund ist, daß die IF-THEN-Struktur vom Betriebssystem des Taschenrechners daraufhin ausgelegt ist, daß zwischen den Befehlen nur exakt eine Anweisung steht.

Da es sich um einen UPN-Rechner handelt, kann dieser Tip immer angewendet werden, da es für den Taschencomputer keine Rolle spielt, ob eine Anweisung vor oder nach dem IF-Befehl ausgeführt wird.

Ein triviales Beispiel soll dies zeigen. Zuerst nehmen wir ein normales Programm:

```
≪                Speicherbedarf: 40 Bytes
  IF X 2 ==
  THEN DROP
  END
≫
```

Und nun betrachten wir das selbe Programm mit der vorgenommenen Änderung:

```
≪ X 2            Speicherbedarf: 35 Bytes
  IF ==
  THEN DROP
  END
≪
```

Die Umsetzung dieses Tips ist einfach und gelingt ohne Probleme. Man schreibt das IF ganz einfach vor die letzte Anweisung vor dem THEN-Befehl. Das war's. Dieser Umbau bringt jedes Mal fünf Bytes Gewinn an verfügbarem Speicher.

Es sei noch darauf hingewiesen, daß dieser Tip bei den Musterlösungen in diesem Buch noch nicht einmal umgesetzt wurde, um Verwirrungen zu vermeiden. Hieraus resultiert der unterschiedliche Verbrauch an Speicher der verschiedenen Taschenrechnermodelle bei den Musterlösungen.

Einziger Nachteil ist das etwas unübersichtlichere Listing des Programms, bei dem man nun genau hinsehen muß, um den eigentlichen Test zu erkennen.

6.1.2 Die Integer-Zahlen

Schon viel früher in diesem Buch haben wir gehört, daß der Rechner intern zwischen Integer- und REAL-Zahlen unterscheidet, was der normale Nutzer aber nicht bemerkt.

Was kümmert es mich? Falsch! Damit läßt sich ebenfalls eine Menge Speicher sparen. Dies funktioniert sehr einfach.

Der Rechner erkennt die ganzen Zahlen von -9 bis 9 als Integer-Zahlen, alle anderen Zahlen werden als REAL-Zahlen abgespeichert. Dies hat gravierende Folgen. Während eine Integer-Zahl lediglich 2.5 Bytes benötigt, bringt es jede REAL-Zahl auf ganze 10.5 Bytes. Das ist 4.2 mal so viel!

Wie läßt sich das nutzen? Stellen wir uns vor, wir brauchen die Zahl 100 im Programm. Allein diese Zahl kostet nun 10.5 Bytes. Wenn wir aber im Programm 2 ALOG ausführen, so steht ebenfalls eine 100 im Stack. Diese wurde zwar berechnet, aber sie ist gleichwertig zur ersten. Hierfür haben wir nun eine (Integer-)Zahl und einen Befehl gebraucht.

Wie sieht die Rechnung beim Speicherverbrauch aus? Ein Befehl kostet uns 2.5 Bytes, die ganze Zahl ebenfalls. Das sind zusammen genau fünf Bytes. Das bedeutet also, daß wir ganze 5.5 Bytes eingespart haben.

Zeigen wir dies noch an einem Beispiel. Das zu schreibende Programm soll eine Zahl mit 100 multiplizieren.

Erste Version: `≪ 100 *` Speicherbedarf: 29.5 Bytes
`≫`

Zweite Version: `≪ 2 ALOG *` Speicherbedarf: 24 Bytes
`≫`

Der Speicherverbrauch ist also tatsächlich gesunken, obwohl die erste Version des Programms eigentlich kürzer aussieht. Man darf sich hier also nicht täuschen lassen. Einzig entscheidend ist der Verbrauch für den Rechner.

Hier schleicht sich allerdings ein Nachteil ein. Die Berechnungen müssen ja durchgeführt werden, und dies kostet leider Zeit. Sollte man also eine REAL-Zahl sehr häufig gebrauchen (mindestens einige Hundert Mal), so verringert sich die Geschwindigkeit des Programms in einem merklichen Rahmen. Allerdings sind selbst dann die Auswirkungen i.d.R. zu vernachlässigen, zumindest im Vergleich zur restlichen Laufzeit des Programms.

Betrachten wir nochmals die Sachlage. Alle Zahlen, die sich mit Hilfe von Integer-Zahlen und Operationen (beides kostet jeweils 2.5 Bytes) darstellen lassen und hierbei maximal vier Befehle oder (Integer-)Zahlen benötigen, sparen Speicher. Daß das Programm vom Code, den der Benutzer lesen kann, länger wird, tut nichts zur Sache.

Die Umsetzung dieses Tips geschieht durch Ersetzen von anderweitig darstellbaren REAL-Zahlen durch Operationen, die diese Zahl ebenfalls erzeugen.

Um erstens ein paar Tips an möglichen Operationen zu geben und zweitens die Leser etwas zu entlasten, habe ich im Anhang auf Seite 266 eine Tabelle zusammengestellt, die Anregungen und Beispiele enthält, was sich hierbei mit welcher Vorgehensweise gewinnen läßt. Diese Tabelle ist selbstverständlich nicht komplett (damit ließe sich ein eigenes Buch füllen), sondern nur als Anregung zu verstehen, was auch die Pünktchen auszudrücken versuchen.

6.1.3 Der Tip mit der START-Schleife

Auch dieser Tip dient nur dem Speichersparen. Bei der Erklärung der Schleifen-Strukturen habe ich darauf hingewiesen, daß die meisten Wiederholungen mit der FOR-NEXT-Schleife getätigt werden. Als Beweis dienen hier all die Übungsaufgaben. Es ist allerdings so, daß die START-NEXT-Konstruktionen weniger Speicher verbrauchen.

Hat man also eine Schleifen-Struktur, in der die Schleifenvariable nicht benutzt wird, so kann man ohne Frage das FOR *Schleifenvariable* durch ein START ersetzen. Dieser Umbau bringt immerhin 4.5 Bytes Gewinn an frei verfügbarem Speicher. Beispiele:

```
≪ 1 9          Speicherbedarf: 32.5 Bytes
  FOR E 1
  NEXT
≫
```

Zum Vergleich folgt das umgestaltete Programm:

```
≪ 1 9          Speicherbedarf: 28 Bytes
  START 1
  NEXT
≫
```

All diese Speicherersparnisse erscheinen manch einem Nutzer eines HP48GX mit 128kBytes (oder mehr) vielleicht lächerlich. Das zeugt aber bestenfalls von Ignoranz, da nur konsequentes Sparen etwas bringt. Mein HP28S hat (immerhin) 32 kBytes, ein HP48S und HP48G ebenfalls. Hätte ich keinen meiner Tips (und der noch kommenden Tricks) je beherzigt, so würde meine Programmsammlung mit ihren 150 Programmen sicherlich nicht nur 21 kBytes brauchen, sondern (geschätzt) 40 bis 50 kBytes. Die Sammlung wäre somit nicht mehr lauffähig, da sie gar nicht ins RAM paßte. Ein Auslagern von Daten (wie beim HP48) kennt mein HP28S ja noch nicht. Die Folgen des langwierigen Eintippens (z.B. in einer Klausur) wären, obwohl ich schon unwahrscheinlich flott tippen kann, absolut fatal.

Dieser Tip kann noch ausgedehnt werden. Immer wenn sich eine Konstruktion durch eine einfachere ersetzen läßt, ist ein Fortschritt erreicht. Eine Tabelle auf Seite 267 zeigt den Speicherbedarf der einzelnen Strukturen auf.

6.1.4 Keine unnötigen Kommentare

Sehr viel Speicher belegt das Aufnehmen von (möglicherweise) langen Kommentaren in Programme. Was bringt das beste und kürzeste Programm, wenn ich als Einleitungszeile Meldungen wie: „Sie haben gerade das Programm ... aufgerufen“ einfüge? Nichts! Die besten Programme sind die, die völlig ohne irgendwelche Kommentare auskommen, solange ihre Bedienung von Grund auf klar ist.

Betrachten wir die Matrizenmultiplikation MMUL aus der Programmsammlung (Seite 193). Man findet keinerlei Kommentare. Natürlich wäre es nützlich, wenn sich der Rechner bei Fehlern des Programms mit Klartext melden würde, aber was kostet es an Speicher? Das angesprochene Programm MMUL hat in der vorliegenden Version 193 Bytes Länge. Es beherrscht die Matrizenmultiplikation von quadratischen und nicht-quadratischen Matrizen, die in meiner Konvention mit geschweiften Matrizen eingegeben werden, ohne Rückmeldung. Sollte natürlich jemand versuchen, folgendes zu berechnen:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 4 & 5 \end{pmatrix},$$

braucht er sich über einen Fehler nicht zu wundern. Es geht nämlich einfach nicht. Das Betriebssystem des Taschenrechners leistet sich den Luxus, den Grund (hier die falschen Dimensionen) zu monieren, wenn man die Matrizen in den eckigen Klammern eingegeben hat und die Multiplikation ausführen möchte. Ich rate dringend davon ab, solche Meldungen in Programme einzubinden.

Programme sollten so aufgebaut werden, daß sie ohne lange Erklärung sofort verständlich sind. Werden sie falsch bedient, erscheint ein Fehler; was sonst? Hilfestellungen sind beim vorhandenen Speicher leider zu aufwendig.

6.1.5 Die Verwendung von ROT ROT

Von der Funktion her sind die Befehlssequenzen 3 ROLLD und ROT ROT absolut gleichwertig, leider aber nicht in ihrer Laufzeit. Meine Messungen (auf meinem HP28S) ergaben für 100 Durchläufe von

– ROT ROT durchschnittlich 1.4 Sekunden

und für

– 3 ROLLD durchschnittlich 1.696 Sekunden

Daher empfiehlt es sich, alle 3 ROLLD-Sequenzen durch ROT ROT zu ersetzen.

6.1.6 Keine Bandwürmer als Namen

Die Länge eines Namens hat ebenfalls Einfluß auf den Speicherverbrauch, und dies gleich zweifach.

- Jeder Buchstabe kostet ein Byte mehr. Tip von mir: Programme mit möglichst kurzen Namen ausstatten - aber nicht zu kurz - man sollte mit etwas Phantasie die Namen noch entschlüsseln können. Ich benütze in meiner Programmsammlung u. a. die in Tabelle 6–1 aufgeführten.

zu kurz	mein Name	zu langer Name	Funktion
MM	MMUL	MATMULT	Matrizenmultiplikation
MA	MAD	MATADD	Matrizenaddition
MD	MDI	MATDIFF	Differenz zweier Matrizen
P	PART	PARTIALBRUCH	Partialbruchzerlegung
N	NULL	NULLSTELLE	Nullstellensuche bei Polynomen

Tabelle 6–1: *Beispiele für Programmnamen*

- Wenn eine Routine in einem anderen Programm als Unterprogramm aufgerufen wird, so ruft man es mit seinem Namen auf. Jeder Buchstabe des Aufrufens kostet abermals Speicher, weshalb allzu lange Namen nicht empfehlenswert erscheinen.

6.1.7 Einbuchstabige Variablennamen

Dies habe ich fast vollständig verwirklicht.
Programmnamen sind mindestens zweibuchstabig und Variablennamen bestehen immer nur aus einem Buchstaben. Dies verhindert zum einen Rätselraten darüber, ob bei der Bezeichnung 'E' von einem Programm oder einer Variable die Rede ist, und zum anderen spart dies ebenfalls Speicher, da Variablennamen erheblich häufiger als Namen von Unterroutinen verwendet werden.

Zudem habe ich in meinen Programmen noch eine weitere Auftrennung vorgenommen, die auch von Hewlett Packard vorgeschlagen wird. Sie bringt zwar keinen Speichergewinn, macht den Rechner auch nicht schneller, ist aber übersichtlicher. Globale Variablen bezeichne ich mit (einstelligen) Großbuchstaben und lokale Variablen immer mit (einstelligen) Kleinbuchstaben. Der Taschenrechner unterscheidet nämlich zwischen einem kleinen und einem großen Buchstaben.

6.1.8 Eine Erleichterung beim Eintippen von Programmen

Oftmals benötigt man Befehle, die entweder langwierig über die Tastatur Buchstabe für Buchstabe eingetippt werden müssen oder nur sehr aufwendig über diskrete Verzeichnisse als Befehl abrufbar sind. In gewissem Rahmen gibt es hier eine Hilfe. Man schreibt sich eine Liste mit den wichtigsten (dafür geeigneten) Befehlen und eröffnet ein MENU. Allerdings reagieren der HP48 und der HP28 auf dieses MENU mit leichten Unterschieden. Das MENU ist bei beiden Modellen immer wieder über das Custom-MENU (CST) aufrufbar, wird beim HP48 aber zusätzlich in der Variablen 'CST' gesichert. Nach Verwendung dieses Tips ist also ein Löschen von 'CST' meist ratsam.

Nach der Eröffnung des MENUs kann auf ganz einfache Weise (per Tastendruck) der gewünschte Befehl aufgerufen werden. Auch die obligatorischen Leerzeichen, die die Befehle trennen, werden automatisch eingefügt.

Leider geht dies nur mit Befehlen wie:

DUP	DUP2	DROP	DROP2	PUT	GET
PUTI	GETI	OVER	ROT	ROLL	ROLLD
LIST→	→LIST	POS	SUB	SIZE	EVAL
→NUM	→STR	STR→	CHR	NUM	DISP
NEG	FACT	RAND	ABS	SIGN	MANT
XPON	etc.				

Ebenfalls sind noch die meisten Befehle für Matrizen, Vektoren, etc. zugelassen, sowie selbst geschriebene (Unter-)Programme.

Nicht erlaubt sind Befehle von Strukturen wie:

IF	THEN	ELSE	END	DO	UNTIL
WHILE	REPEAT	FOR	START	NEXT	STEP

Gibt man die Befehle hingegen als Zeichenketten ein (z.B. { "UNTIL" }), so sind beliebige Ausdrücke im MENU zugelassen. Der Nachteil hierbei ist aber, daß dann beim HP48 die Leerzeichen nicht mehr eingefügt werden. Der HP28 hingegen zeigt die Befehle im MENU als Strings, reagiert ansonsten aber wie oben beschrieben.

6.1.9 Ein Hinweis zum Entwickeln etwas größerer Programme

Aus meiner Sicht erscheint es am günstigsten, ein Programm nach Anfertigung eines Struktogramms (wenn auch nur im Kopf), in einzelne Teil-Probleme zu zerlegen und diese getrennt zu entwickeln. Sobald einzelne Stücke funktionieren, sollte man diese als Unterprogramme in das steuernde Hauptprogramm integrieren. Erst wenn das gesamte Programm mit allen Spezialfällen und Unterroutinen korrekt läuft, sollte man sich Gedanken darüber machen, welche Unterprogramme man als solche behalten möchte. Damit kann man diese bei anderen Problemen ebenfalls einsetzen. Hierbei sind vier Überlegungen anzustellen:

- Ein Unterprogramm (UP) läßt sich leicht auch von anderen Programmen nutzen.
- Ein UP-Aufruf kostet mehr Speicher als die einmal direkt vorhandene Routine.
- Ein Aufruf einer Unterroutine ist in der Bearbeitung deutlich langsamer.
- Je mehr Unterprogramme sich im Speicher befinden, desto leichter verliert man den Überblick.

Hierzu noch ein paar Anregungen: Es ist sicherlich speichersparender, wenn man eine Routine, die man häufiger braucht, als Unterprogramm umsetzt. Der Verlust an Rechengeschwindigkeit ist i.d.R. zu vernachlässigen. Wird aber eine Routine nur in einem einzigen Programm gebraucht, so ist es eigentlich ratsam, dieses in das Hauptprogramm als Quellcode einzubinden. Sollte eine Routine in einem einzelnen Programm mehrfach benötigt werden, so kann man dieses mit einem Trick (siehe Seite 131) ebenfalls ins Hauptprogramm aufnehmen.

6.2 Raffinierte Tricks für Fortgeschrittene

In diesem Abschnitt habe ich alle von mir jemals erarbeiteten Kniffe zusammengestellt, die allesamt Vorteile bringen, sich aber meist nicht trivial umsetzen lassen.

Niemand braucht sich Sorgen machen, wenn er diese Tricks nicht umzusetzen weiß. Das wichtigste an einem Programm ist immer noch seine korrekte Funktion. Ein optimal entwickeltes Programm hat zwar noch zusätzliche Vorzüge, aber die Funktion steht immer im Vordergrund.

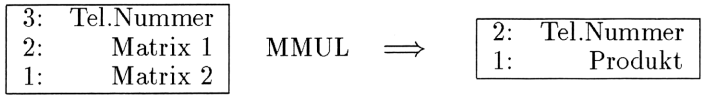
Ich habe versucht, die Tricks der Umsetzbarkeit nach zu ordnen, ich hoffe dies ist gelungen.

6.2.1 Kein Verwenden von CLEAR

Nur ein schlechter Programmierer braucht den Befehl CLEAR. Ein guter Programmentwickler muß immer wissen, welche und insbesondere wie viele Elemente im Stack stehen.

Dies hat selbstverständlich noch einen Hintergrund. Ich möchte dies sehr praxisnah erklären. Man traf in der Cafeteria ein hübsches Mädchen/einen hübschen Jungen und hat sich die Telefonnummer in Ermangelung von Stift und Papier als String im Stack gespeichert. In der Vorlesung/im Beruf berechnet man etwas mit Hilfe eines Programms. Nehmen wir an, das Programm enthält den Befehl CLEAR - futsch - weg ist sie, die Telefonnummer.

Ein sauber entwickeltes Programm darf nur die erwarteten Objekte vom Stack nehmen! Wieder zurück in die Praxis: die Matrizenmultiplikation MMUL darf zwei Matrizen vom Stack nehmen, der Rest der Stacks darf nicht angetastet werden.



6.2.2 Der Befehl DEPTH ist unnötig

Ähnlich wie bei CLEAR liegt der Fall bei DEPTH. Wozu diesen Befehl in einem Programm nutzen? Wenn ich schon nicht weiß, wie viele Elemente im Stack herumschwirren, dann kann ich doch auch nicht wissen, was überhaupt im Stack steht. Auch hier lautet mein Urteil: Nur ein schlechter Programmierer braucht diesen Befehl.

Es gibt nur eine sinnvolle Anwendung: Wenn man den ganzen Stack sichern möchte (z.B. in eine Liste), bevor man irgendwelche Berechnungen anstellt, kann man damit die unbekannte Anzahl der Objekte des Stacks bestimmen.

6.2.3 Keine Sicherheitsabfragen

Unsinn sind Sicherheitsabfragen am Anfang eines Programms. Erstens kostet eine Abfrage, ob wirklich der richtige TYPE der richtigen Größe vorliegt, viel Zeit und zweitens auch noch viel Speicher. Beides haben wir beim Taschenrechner (leider) nicht. Mir wären solche Abfragen als Sicherheitsfreak und Perfektionist ja auch lieber, aber der Preis ist viel zu hoch, um dies umsetzen zu können.

6.2.4 Die Flags

Bei den Flags gibt es ebenfalls einen Tip zur Nutzung. Wie wir ja schon wissen, brauchen Integer-Zahlen weniger Speicher. Aus diesem Grund lassen sich die Flags von Eins bis Neun logischerweise mit weniger (Speicher-)Aufwand nutzen.

Nutzer des HP48 stehen vor einer weiteren Entscheidung. Die Flags Eins bis einschließlich Fünf werden in der Status-Zeile angezeigt (die jeweilige Zahl ist zu sehen, wenn der Flag gesetzt ist). Das läßt sich hervorragend nutzen. Wenn Programme sogenannte Systemeinstellungen haben, sollten die Flags Eins bis Fünf gewählt werden. Unter Systemeinstellungen ist eine unterschiedliche Verfahrensweise des Programms zu verstehen, die über Flags gesteuert werden kann (soll). Man hat zum Beispiel ein Programm geschrieben, das die Nullstellen eines Polynoms mit zwei verschiedenen Algorithmen suchen kann. Durch den Flag könnte dem Programm die Wahl des zu wählenden Algorithmus' vorgeschrieben werden.

Wer sich mit dem Gedanken trägt, meine Programmsammlung zu nutzen, der sei darauf hingewiesen, daß ich die Flags Eins und Zwei für Systemeinstellungen verwendet habe. Die Flags Drei, Vier und Fünf sind frei, während die Flags Sechs bis Neun in manchen Programmen intern verwendet werden.

6.2.5 Effektives Löschen

Oftmals sind in einem Programm doch globale Variablen nötig geworden. Diese möchte man ohne viel Aufwand am Ende der Berechnungen löschen. Schreibt man die Namen der zu löschenden Programme und Variablen in eine Liste und führt den Befehl PURGE aus, so werden sämtliche in der Liste enthaltenen Namen aus dem aktuellen Verzeichnis gelöscht.

Beispiel: { A B C } PURGE löscht die Programme bzw. Variablen mit den Namen 'A', 'B' und 'C'.

6.2.6 Die Vermeidung von −1 STEP

Bei manchen Programmen muß man einfach rückwärts zählen. Da ist nicht immer etwas zu ändern. Manchmal ist es aber gleichgültig, ob nun vorwärts oder rückwärts gezählt wird. Dann gilt folgendes: Man kann hier Speicher sparen, und oft springt zusätzlich eine etwas schneller laufende Version heraus. Der Umbau gestaltet sich meist einfach.

Wurde vorher von B nach A rückwärts mit

```
≪ B A
  FOR E Anweisungen
  −1 STEP
≫
```

gezählt, so heißt es nun:

```
≪ A B
  FOR E Anweisungen
  NEXT
≫
```

Für A=1, B=100 und keinerlei Anweisungen (um das Ergebnis nicht zu verwässern) ergaben sich beim oberen Beispiel auf meinem HP28S durchschnittlich 0.722 Sekunden und beim unteren Beispiel 0.646 Sekunden. Das sind immerhin 10.5 % weniger.

Der Grund ist, daß beim oberen Beispiel immer erst die −1 in den Stack gelegt wird, ehe sie dann gleich wieder von STEP verwendet wird. Das umgeht der NEXT-Befehl.

6.2.7 GET und PUT bei den Vektoren

Beim Entnehmen von Zahlen aus Vektoren kann man auf die umschließende geschweifte Klammer (Akkolade) bei der Beschreibung der Position verzichten, d.h. es ist auch [4 5 6] 2 GET ⟹ 5 erlaubt.

Gleiches gilt im übrigen auch für den Befehl PUT, bei dem auch auf die geschweifte Klammer bei der Positionsangabe verzichtet werden kann. Das Problem ist aber dann, daß man sehr leicht die Position und das neue Element von der Reihenfolge der Eingabe her verwechselt.

6.2.8 Die Anweisung SUB

Öfter braucht man den hinteren Teil einer Liste. Der HP48G/GX hält hierfür einen Spezialbefehl zur Verfügung, der lediglich das erste Objekt einer Liste wegschneidet:

{ A B C } TAIL ⟹ { B C }

Manchmal möchte man aber die Liste vom x-ten bis zum letzten Element. Penibel programmiert wäre das dann x OVER SIZE SUB.

Es geht aber auch anders. Wenn der Endwert eines SUB größer als die Länge der Liste selbst ist, dann stört dies den Rechner auch nicht. Unter der Voraussetzung, daß die Liste weniger als eine Milliarde Elemente enthält, tut es also auch x 9 ALOG SUB, was schon schneller ausgeführt wird.

Speicher sparen läßt sich dann, wenn man sicher weiß, daß maximal neun Elemente in der Liste stehen. In diesem Fall würde dann auch ein x 9 SUB ausreichen.
Das gleiche gilt im übrigen für Zeichenketten, die dadurch ebenfalls gekürzt werden können.

Als Erweiterung funktioniert dies auch mit dem Anfang einer Liste oder Zeichenkette. Ein −9 3 SUB schneidet alle Elemente einer Liste nach dem dritten Objekt ab. Dies läßt sich in der Praxis aber kaum nutzen, da man ja weiß, daß die Liste mit Objekt Eins anfängt. Theoretisch Verwendung finden kann es nur dann, wenn durch irgendwelche Berechnungen eine Zahl kleiner als Eins im Stack stehen geblieben ist und diese hierfür verarbeitet werden kann, da man sie anderweitig nicht mehr braucht.

Deutlich praxisrelevanter ist aber folgendes: Der Befehl verdaut auch REAL-Zahlen. Die Zahlen werden immer auf den zunächst liegenden ganzzahligen Wert gerundet.

{ 1 2 3 } 1 1.6 SUB $\Longrightarrow$ { 1 2 }

Entsprechendes gilt auch bei den Zeichenketten.

6.2.9 Die Vermeidung globaler und lokaler Variablen

Auch diesen Punkt habe ich schon früher angesprochen. Die Vermeidung von globalen Variablen ist fast immer sinnvoll, da erstens viel Speicher gespart werden kann und zweitens auch noch ein Zeitgewinn möglich ist. Absolut äquivalent lassen sich die lokalen Variablen nutzen, die erheblich schneller verarbeitet werden. So braucht:

```
≪ 'X' STO X SQ X - X      Speicherbedarf: 48.5 Bytes
/
≫
```

im Mittel 0.61 Sekunden (auf meinem HP28S) zur Ausführung, während

```
≪ → X                     Speicherbedarf: 48.5 Bytes
  ≪ X SQ X - X /
  ≫
≫
```

durchschnittlich 0.196 Sekunden benötigt. Die Ergebnisse sprechen für sich.

Die lokalen Variablen sollten, falls andere Konstruktionen ohne viel Aufwand denkbar sind, ebenfalls vermieden werden. Sind alle Operationen im Stack durchführbar, so ist ein weiterer Gewinn von Geschwindigkeit und auch Speicher möglich, und meist sogar leicht zu realisieren. Die Handhabung von Objekten im Stack wurde bei den Übungsaufgaben ausreichend eingeübt.

6.2.10 Der DUPN-DROP-Trick

Diese manchmal sehr geschickte Vorgehensweise lohnt sich immer, wenn mehrere weiter oben im Stack stehende Elemente kopiert werden müssen.

Im Stack stehe:

4:	A
3:	B
2:	C
1:	D

Gesucht werde:

6:	A
5:	B
4:	C
3:	D
2:	A
1:	B

Mit 4 DUPN DROP2 ist das mit einem Aufwand von 7.5 Bytes geschehen.

Die Anwendung lohnt sich i.d.R. nur, wenn maximal die beiden unteren Ebenen nicht kopiert werden müssen und mindestens zwei Elemente dupliziert werden sollen. Die untersten Ebenen lassen sich nach dem DUPN mit DROP oder DROP2 problemlos entfernen. Bei mehr Elementen muß mit X DROPN gearbeitet werden, was 2.5 Bytes für die Integer-Zahl X benötigt. In der Praxis sind dann andere Vorgehensweisen oftmals günstiger.

Dieser Kniff ist umso wirkungsvoller, je mehr Objekte dadurch kopiert werden können, was aber nicht dazu führen darf, daß man in seiner Euphorie gleich alles vervierfacht, obwohl man die Elemente eigentlich gar nicht bräuchte.

6.2.11 Der Trick mit dem verschobenen Befehl

Sehr einfach läßt sich folgendes umsetzen: Wir nehmen eine IF-THEN-ELSE-END-Struktur an, die beim THEN- und ELSE-Anweisungsblock mit dem (den) selben Befehl(en) endet:

```
≪
  IF Bedingung
  THEN Anweisungen 1
Befehl 1
  ELSE Anweisungen 2
Befehl 1
  END
≫
```

Obiges läßt sich ohne Frage auch als

```
≪
  IF Bedingung
  THEN Anweisungen 1
  ELSE Anweisungen 2
  END Befehl 1
≫
```

schreiben, da *Befehl 1* in jedem Fall am Schluß ausgeführt wird. Jeder Befehl, der dadurch nach hinten gezogen werden kann, spart 2.5 Bytes.

Ganz ausgefuchste Programmierer werden versuchen, die Anweisungen so zu stellen, daß gleiche Befehle am Schluß zu stehen kommen. **Achtung!** Das ist nichts für Anfänger.

Ebenfalls schwierig ist es, (einen) Befehl(e), die zu Anfang des THEN- und des ELSE-Blockes ausgeführt werden müssen, vorher auszuführen. Dies gelingt in der Praxis allerdings recht selten.

Zuletzt soll noch darauf hingewiesen werden, daß die Struktur von IF-THEN-(ELSE)-END so ausgelegt ist, daß am meisten Speicher gespart wird, wenn jeweils zwischen dem IF-, THEN-, ELSE- und dem END-Befehl nur eine einzige Anweisung steht. Sind dort mehr Befehle, so kostet allein diese Tatsache fünf Bytes extra.

6.2.12 Der optimale Nutzen von LIST→

Wie schon bekannt, werden beim Befehl LIST→ nicht nur die Elemente der Liste selbst sondern auch die Länge der Liste in den Stack geschrieben. Oftmals ist diese Zahl nicht nötig, oder sogar störend. Man kann sie sicherlich wegwerfen. Optimal genutzt, kann sie aber sogar Vorteile bringen. Häufig kann die Zahl bei Stackverschiebungen oder dem Kopieren von Objekten im Stack verwendet werden.

Ein Beispiel soll dies zeigen: Aus { A B } soll die Formel $\frac{A}{B} + A$ berechnet werden. Ein konventionelles Programm sieht wie folgt aus:

```
≪ LIST→ DROP OVER      Speicherbedarf: 30.5 Bytes
SWAP / +
≫
```

Durch optimale Nutzung von LIST→ entsteht:

```
≪ LIST→ PICK SWAP /    Speicherbedarf: 28 Bytes
+
≫
```

Man muß hierzu sicherlich ein neues Programm schreiben. Viele werden den Umbau einer alten Routine scheuen. Bei einer Neuentwicklung ist es hingegen immer ratsam, sich an diesen Trick zu erinnern.

Beim HP48 läßt sich aber etwas anderes nutzen. Wenn man weder die Länge der Liste braucht, noch sie irgendwie günstig verarbeiten kann, führt der Befehl EVAL exakt die gewünschte Funktion aus.

Beispiel (nur für HP48): { A B C } EVAL ⟹

3:	A
2:	B
1:	C

im Vergleich zu: { A B C } LIST→ ⟹

4:	A
3:	B
2:	C
1:	3

6.2.13 Die Verwendung von PUTI und GETI

Wenn man es schafft, die Befehle PUTI und GETI optimal einzusetzen, kann man nicht so aufwendige Schleifenkonstruktionen, die weniger Speicher brauchen, wählen. Trotzdem ist der Vorteil schwierig zu nutzen, da durch die Befehle auch mehr Elemente im Stack stehen und verwaltet werden müssen. Ein weiterer Vorteil der Verwendung ist die Tatsache, daß das Objekt beim Anwenden von GETI nicht gelöscht wird und deshalb auch nicht dupliziert werden muß, was einen Gewinn an Speicher bedeutet. Eine perfekte Nutzung ist mir allerdings bis heute auch noch nicht gelungen.

6.2.14 Bedingungen, ohne einen Test durchzuführen

Dieser Trick wurde schon bei einigen Aufgaben eingesetzt und beruht im Wesentlichen darauf, daß der Taschenrechner erstens ein UPN-Rechner ist und zweitens jede Zahl ungleich Null als eine wahre Aussage interpretiert.

Das bedarf natürlich noch einiger Erklärungen: Bei Entscheidungen, die es zu treffen gilt, kann beim Computer beliebig viel mit Daten manipuliert werden. Wichtig für die Verzweigung ist lediglich das Ergebnis, das er bei Erreichen des THEN-Befehls in der Stackebene 1 stehen hat.
Ist dieser Wert ungleich Null, so interpretiert der Rechner dies als erfüllte Bedingung. Die Nutzung sieht so aus, daß der Taschencomputer nicht auf Konstruktionen wie

```
≪
  IF Zahl 0 ≠
  THEN Anweisungen
  END
≫
```

angewiesen ist, sondern akzeptiert mit gleicher Funktion auch:

```
≪
  IF Zahl
  THEN Anweisungen
  END
≫
```

6.2.15 Modifizierung einer globalen Variablen

Angenommen, man möchte ein Programm schreiben, welches mit globalen Variablen arbeitet, diese während des Ablaufs aber (unerwünschterweise) verändert. Dies ist durch Berechnungen durchaus die Regel. Da man aber die Ausgangsdaten exakt wieder am Ende haben möchte, ärgert man sich jedes Mal, wenn man nach Programmaufruf die alten Daten wieder eingeben bzw. die in einer anderen Variable gespeicherten alten Daten „umladen" muß.

Hier gibt es einen simplen Trick. Zu Beginn des Programms ruft man den Inhalt der globalen Variable auf und nimmt diesen in eine lokale Variable gleichen Namens. Während des Ablaufs ist diese Variable exakt gleich einer globalen, mit dem Unterschied, daß nach Beendigung des Programms die alten Daten wieder da sind.

Beispiel: Es soll eine Liste mit { 1 2 3 4 5 6 7 } in 'V' abgespeichert werden. Diese soll Stück um Stück im Programm um ein Element gekürzt werden. Das jeweilige Ergebnis soll in der Variablen selbst abgespeichert und zur Kontrolle in Zeile 1 des Displays ausgegeben werden. Am Ende des Programms soll die alte Liste wieder in V stehen.

Lösung:

```
≪ V → V
  ≪ 1 7
    START V 2 9 SUB
DUP 1 DISP 'V' STO
1 WAIT
    NEXT
  ≫
≫
```

Das 1 WAIT dient nur dem leichteren Verfolgen.

6.2.16 Der Trick mit den Unterprogrammen

Hin und wieder benötigt man ein Unterprogramm mehrfach in nur einem einzelnen Programm. Dummerweise läßt sich das Problem aber nicht immer so lösen, daß man eine Schleife zum mehrfachen Nutzen des Programmteils verwenden kann. Trotzdem läßt sich mit einem Trick ein Unterroutinenaufruf verwirklichen, ohne daß wirklich ein separates Unterprogramm geschrieben wird.

Der Trick beruht darauf, daß auch ein Programm innerhalb eines Programms in den Stack gelegt werden kann. Solange man weiß, in welcher Stackebene es sich befindet, wenn man es benötigt, kann man es später mit X PICK (oder X ROLL) aus Ebene

X herunterholen und mit EVAL ausführen. Auch ein Speichern eines Programms in einer lokalen Variable ist möglich:

```
≪                                   Speicherbedarf: 62.5 Bytes
  ≪ P1
  ≫ 1 9
  FOR E E OVER EVAL
DROP2
  NEXT DROP
≫
```

Der Vorteil liegt in der Ersparnis, ein Unterprogramm für diesen Fall auszuführen (hier wäre der Trick selbstverständlich unnötig gewesen).

Ein Gegenbeispiel sieht wie folgt aus: ≪ P1 ≫ 'A' STO Speicherbedarf: 21 Bytes

und:
```
≪ 1 9                    Speicherbedarf: 44 Bytes
    FOR E E + A DROP2
    NEXT
  ≫
```

Letzteres benötigt also mindestens 65 Bytes. Der günstigste Fall des einbuchstabigen Unter-Programmnamens (hier: 'A') ist aber nicht ratsam, da er leicht mit Variablennamen verwechselt wird.

6.2.17 Optimale Programmierung

Es dürfte wohl keinem Leser verborgen geblieben sein, daß vorteilhafte Programmierung in allen Fällen das mit Abstand wichtigste ist, wenn man clevere Programme schreiben möchte.
Dummerweise ist diese nicht einfach mit einigen wenigen Regeln zu beschreiben, da sie sehr komplex ist. Ich habe in all den Übungsaufgaben versucht, die Vorgehensweise so ausführlich wie irgend möglich zu gestalten, um ein Nachvollziehen zumindest zu erleichtern, und hoffe, daß dies wenigstens ansatzweise gelungen ist.

6.2.18 Anlegen der Verzeichnisse

Dies hat eigentlich gar nichts mit der Programmierung zu tun, trotzdem möchte ich auch hier einige Hinweise geben: Wenn man in einem Verzeichnis ein Programm (oder eine Variable) aufruft, so führt der Rechner den Inhalt aus, wenn es (bzw. sie) im aktuellen Verzeichnis ist. Befindet es (bzw. sie) sich dort nicht, so durchsucht der Rechner das Directory, das dem HOME-Verzeichnis um eine Ebene näher liegt. Ist der Name dort zu finden, wird das Programm (bzw. die Variable) abgerufen, ansonsten die Suche wieder einen Schritt Richtung HOME fortgesetzt. Dies wird solange fortgeführt, bis man im HOME-Directory angekommen ist.

Dies hat aber zur Folge, daß man immer auf beliebige Programme (und Variablen) zurückgreifen kann, wenn sich diese alle im HOME-Verzeichnis befinden, da sie sich (nach kurzer Suche des Rechners) immer im HOME-Directory finden lassen. Niemals braucht eine Suche gestartet werden, wo sich nun dieses oder jene Programm nun gerade wieder befindet. Es gibt dann ebenfalls keine Schwierigkeiten wegen nicht vorhandener Unterprogramme, die allein betrachtet auch als selbständiges Programm fungieren, in manchen Fällen aber auch als Unterprogramm.

Der Nachteil liegt auf der Hand. Sind zu viele Programme in HOME abgespeichert, so verliert man schnell den Überblick. Trotzdem sind alle 150 Programme meiner Programmsammlung bei meinem HP28S im HOME-Verzeichnis abgelegt, um sie immer abrufbar zu haben. Zum Aufruf der Programme tippe ich einfach den Namen ein. Das setzt aber deren Kenntnis voraus, aber über diese verfüge ich. Wer es also schafft, sollte alle seine „wichtigen" Programme nach oben ins HOME-Verzeichnis schaffen und dann nur mit dem Eintippen über die Tastatur alles erledigen. Sicherlich habe ich es hier als Nutzer eines HP28S mit seinen zwei Tastatur-Feldern deutlich leichter. Die Entscheidung ist selbstverständlich freigestellt.

6.2.19 Warum sind Listen so wichtig?

Aus eigener Erfahrung weiß ich, daß es erhebliche Vorteile bringt, wenn man alle seine Programme mit Hilfe der Listen schreibt. Dies hat einen einfachen Grund. Der Rechner verdaut damit alle nur erdenklichen TYPEs, die er kennt.

Der HP48G hat eine Nullstellensuche, die zwar schnell ist, aber bei mehrfachen Nullstellen reichlich miese Ergebnisse liefert. Das ganze Problem wurde numerisch gelöst. Aufgrund einer Studienarbeit mit gleichem Thema habe ich ebenfalls eine Nullstellensuche programmiert, die diese Probleme nicht hat. Geschwindigkeitsmäßig kann ich mit der in Maschinensprache geschriebenen Version des HP48G natürlich nicht mithalten, wohl aber bei den Ergebnissen. Die Eingabe tätigt man beim HP48G als „Vektor", wobei z.B. [1 2 3] das Polynom $x^2 + 2x + 3$ darstellt.

Ein paar Jahre früher kam ich wohl auf dieselbe Idee, ich schreibe mein Polynom

bei diesem Beispiel als { 1 2 3 }. Man sollte nicht darauf hereinfallen, daß das doch nichts ausmache. Irrtum! In meiner Programmsammlung sind noch Routinen zur Polynommultiplikation, -addition, -subtraktion, Partialbruchzerlegung etc. enthalten. Da diese allesamt mit Variablen rechnen können, mußte ich die Listen wählen, da die „Vektoren" nur Zahlen annehmen.

Das Ganze geht also symbolisch durch Verwendung von Listen. So wird das Polynom ax^2+bx+c als { a b c } eingegeben, das Polynom $ax+b$ als { a b } geschrieben. Möchte man diese beiden multiplizieren, so tippt man MUL ein, und das von mir geschriebene Programm MUL liefert { 'a^2' '2*a*b' 'a*c+b^2' 'b*c' } zurück. Es wäre also Unfug, andere TYPEs zu verwenden, nur weil auch meine Nullstellensuche rein numerisch ist. Wer das Programm mit einem symbolischen Polynom startet, der bekommt nach kurzer Zeit einen „*-Error" „Undefined Name" geliefert, was wohl deutlich genug ist.

Beim Taschenrechner dachte sich HP natürlich auch etwas. Wieso sollte man einen neuen TYPE einführen, wenn ein alter diesen Dienst übernehmen kann? Die Vektoren lassen lediglich numerische Werte zu, das Programm zur Nullstellensuche braucht keine zusätzliche Sicherheitsabfrage.

6.2.20 Ein Vorschlag zur Nomenklatur

Auch dies ist (natürlich) nur aus meiner Programmsammlung entnommen. Warum auch nicht? Mein Vorschlag richtet sich an jene, die meine Konvention einfach übernehmen wollen.

- Ein Polynom schreibe ich (wie schon beschrieben) als Liste, wobei ich ja lediglich die Koeffizienten angebe. Nach kurzer Eingewöhnungszeit liest sich das sogar sehr schnell.

  Beispiel: Das Polynom ax^3+bx^2+d wird als { a b 0 d } eingetippt.

  Achtung! Nicht den linearen Koeffizienten unterschlagen!

- Die Matrix

  $$\begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix}$$

  wird als { { a b c d } { e f g h } } eingegeben.

- Die gebrochen rationalen Funktionen werden als Liste zweier „Polynom-Listen" geschrieben:

  { { *Zählerpolynom* } { *Nennerpolynom* } }

  Beispiel: Die Funktion $\frac{ax^2+bx}{bx^3-cx+d}$ wird als { { a b 0 } { b 0 '-c' d } } eingegeben.

Um ein Wechseln von meinen Formaten in die rechnerübliche nicht zur Strafarbeit verkommen zu lassen, habe ich mir Konversionsprogramme geschrieben, die von einem ins andere Format die Daten transformieren können, soweit ich dies brauche. So ist gewährleistet, daß ich die eingebauten Funktionen des Taschenrechners ebenfalls voll nutzen kann.

7 Die Diskette

7.1 Die Programme auf der Diskette

Wie schon mehrfach erwähnt, sind alle Musterlösungen der Übungsaufgaben sowie die Routinen der Programmsammlung auf der beigelegten Diskette zu finden. Nutzer des HP48 mit einem Personal-Computer und dem notwendigen Übertragungskabel können sich hiermit die Arbeit des Abtippens der Lösungen und der Programme sparen.

Da verschiedene Versionen von Sende- und Empfängerprogrammen der PCs existieren, kann hier kein Hinweis gegeben werden, mit welchen Anweisungen des PCs die Daten kopiert werden können. Hier ist auf die Benutzeranleitung der Programme zu verweisen. Beim Taschenrechner ist darauf zu achten, daß seine Übertragung bei den I/O-Parametern auf **wire** steht. Hierzu ruft man das Menü IOPAR auf und drückt die IR/W-Taste, bis **wire** in der Anzeige erscheint. Gesendet werden können die Daten erst, wenn der Taschenrechner schon empfangsbereit ist.

Die Diskette enthält zwei Verzeichnisse für die beiden verschiedenen Modelle des HP48. Im jeweiligen Directory befinden sich die entsprechenden Daten für die Taschenrechner.

Die Daten unterteilen sich jeweils nochmals in fünf Verzeichnisse. Hierbei sind die Programme in den Verzeichnissen mit den Namen PRG1 bis PRG4 und die Musterlösungen der Übungsaufgaben im Verzeichnis LSG zu finden. Letzteres unterteilt sich wieder in die Lösungen für die Stackaufgaben, die Schleifen- und Verzweigungsaufgaben und zuletzt die gemischten Aufgaben. Die Zuordnung der einzelnen Programme bzw. Lösungen ist in den entsprechenden Abschnitten erläutert.

Achtung! Beim Laden der Daten ist darauf zu achten, daß der empfangende Taschenrechner auch über genug Speicher verfügt, um die Daten zu empfangen.

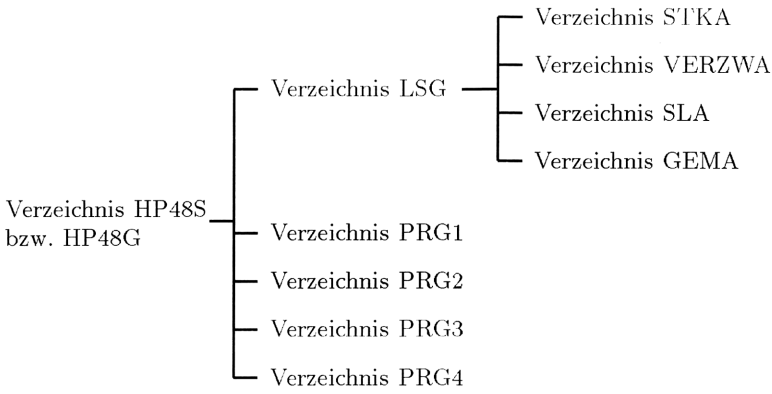


Abb. 7–1: *Struktur der Verzeichnisse auf der Diskette*

8 Programmsammlung

In diesem Abschnitt soll anhand der Programmsammlung die Leistungsfähigkeit des Rechners gezeigt werden. Die Zusammenstellung ist das Ergebnis jahrelanger Programmiertätigkeit auf einem HP28S. Die Routinen entstanden, als ich mich mit diversen mathematischen Problemen im Rahmen meines Studiums der Elektrotechnik beschäftigte und mir die stupide Rechenarbeit ersparen wollte. Da in letzter Zeit die Rechner HP48S/SX und HP48G/GX große Verbreitung gefunden haben, habe ich alle Programme (falls sinnvoll) auf die HP48-Modelle umgeschrieben. Nahezu alle Programme erlauben Variablen bei der Eingabe. Die Bedienung der Programme ist einfach und schnell erlernt. Die Programme zeigen immer ein Beispiel, das die Bedienung verdeutlichen soll.

Alle Programme dieser Sammlung benötigen lediglich 21.0 kBytes RAM. Der Leser wird aber sicherlich selbst eine Auswahl treffen.

8.1 Anmerkungen

Zuerst noch einige Anmerkungen, die spätere Mißverständnisse verhindern sollen: Alle meine Programme sind sehr stark auf speichersparende Programmierung getrimmt, was so manches mal zu einigen sehr seltsamen Konstruktionen geführt hat, die ich hier anspreche, um zu verhindern, daß manch Programmierer durch eine vermeintliche Kürzung der Programme plötzlich mehr Speicher braucht.

Paradebeispiel ist '6 6 +'. Nahezu jeder wird sich dabei fragen, weshalb hier nicht einfach die Zahl 12 steht. Ganz einfach: Die Integer-Zahlen von -9 bis 9 brauchen nur 2.5 Bytes, alle anderen 10.5 Bytes. Das heißt für dieses Beispiel, daß '6 6 +' insgesamt 7.5 Bytes (2.5 Bytes jeweils für die Zahl 6 und 2.5 Bytes für das Plus-Zeichen) braucht und die Zahl 12 satte 10.5 Bytes, eine Ersparnis von immerhin 3 Bytes. Aber zugegebenermaßen ist das in der Programmausführung etwas lang-

samer, da die Rechenoperation 6+6 durchgeführt werden muß. Ich habe mir bei allen meinen Programmen oft genug stundenlang den Kopf zerbrochen, wann ich lieber Speicher und wann ich lieber Zeit sparte und habe, so glaube ich, einen guten Kompromiß gefunden.

Gleich am Anfang habe ich ein Programm SYS für den HP28S (Modell 2BB), das ein Programm FA installiert. Dieses Programm SYS muß exakt so abgetippt werden, wie es dasteht, sonst droht ein Rechnerabsturz!!! Also unbedingt vorher die Prüfsumme kontrollieren. Wenn das Programm FA erst einmal existiert, kann man den HP28S durch Aufruf dieses Programms auf doppelte Geschwindigkeit beschleunigen. Alle Programme, bei denen die erhöhte Geschwindigkeit sinnvoll ist, rufen dieses Programm auf.

Beim HP48 ist mir ähnliches schon vom 48S bekannt, der durch das Bildschirmabschalten etwa 10 Prozent beschleunigt werden kann. Aus rechtlichen Gründen konnte es hier nicht aufgenommen werden. Beim HP28S ist mir durch das Bildschirmabschalten eine Beschleunigung nicht gelungen. Für den HP48 habe ich deshalb gleich zu Beginn dieser Sammlung einen Dummy geschaffen, welcher einen Platzhalter darstellt, um ein Beschleunigungsprogramm schnell hinzufügen zu können.

Die Programm-Listings wurden von mir im Dezember 1993 auf den Zeilenumbruch des HP48 umgeschrieben, um dem größten Anwenderkreis der Programme gerecht zu werden. Beim Aufruf auf einem HP28S können somit leichte Abweichungen auftreten. Sind spezielle Versionen für den HP28S vorgesehen, so ist der Zeilenumbruch natürlich dort dem des HP28S angepaßt.

Das Programm(paket) DE stellt eine große Hilfe dar (Funktion siehe dort). Wer trotzdem darauf verzichten will, sollte ≪≫ 'DE' STO eintippen, um einen Platzhalter zu schaffen.

Um mögliche Fehler in den Programmen nicht zu provozieren, sollte man einige Grundeinstellungen von mir übernehmen:

- 1/0 sollte 9.99999999999E499 als Ergebnis liefern (und keinen Fehler) (HP28S: 59 CF / HP48: -22 SF)
- der '.' sollte als Dezimalpunkt gewählt werden (HP28S: 48 CF / HP48: -51 CF)
- das symbolische Rechnen muß eingeschaltet sein, da sonst symbolisches Rechnen nicht möglich ist (HP28S: 36 SF / HP48: -3 CF)
- es empfiehlt sich den Konstantenmodus einzuschalten (HP28S: 35 SF / HP48: -2 CF)
- beim HP48 muß zur Kontrolle der Checksum-Zahlen die Zahlenbasis HEX sein (HP48: -11 SF und -12 SF)

8.1.1 Die Aufteilung eines Programms

Die Programmbeschreibungen haben alle folgenden Aufbau:

Programmname		
Version:	Rechnertypen	≪ Programmcode ≫
Checksum(HP28S):	Prüfsumme des HP28S	
Checksum(HP48):	Prüfsumme des HP48S/SX bzw. HP48G/GX	
Bytes:	benötigter Speicher	

Inhalt des Stacks vorher	Inhalt des Stacks nachher

Erläuterung:	Erläuterung zum Sinn und zur Funktion des Programms.
Beispiel:	Ein einfaches Test-Beispiel.
benötigte UP:	benötigte Unterprogramme

Zu den Rechnertypen sei noch folgendes angemerkt:

Wenn vom HP48 die Rede ist, so sind immer alle Modelle S, SX, G und GX gemeint. Steht z.B. HP48S bzw. HP48G dort, so ist jedes Mal auch die SX- bzw. GX-Version gemeint.

8.1.2 Hinweis zum Eingabeformat der Programme

In dieser Programmsammlung ist häufig von der Listen-Schreibweise die Rede, was daran erinnern soll, daß z. B. das Polynom $x^2 + 2x + 3$ als { 1 2 3 }, die gebrochen rationale Funktion $\frac{x^3-2x^2-4x+6}{x^4-4x-5}$ als {{1 -2 -4 6}{1 0 0 -4 -5}} und die Matrix $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$ als {{a b c}{d e f}{g h i}} einzugeben ist. Dies hat den entscheidenden Vorteil, daß dadurch ein Rechnen mit Variablen überhaupt erst möglich wird. Um aber auch auf die implementierten Routinen des Rechners zurückgreifen zu können, habe ich Konversionsprogramme für die Polynome und die Matrizen geschrieben, so daß man ohne Aufwand von einem ins andere Format wechseln kann.

8.1.3 Die benutzten Flags

Einige Programme benutzen Flags. Dies sind insbesondere die Flags 1, 2, 6, 7, 8 und 9. Diese Belegung wurde erst spät geändert, um den Usern des HP48 entgegenzukommen. Der HP48 zeigt in der obersten Zeile seines Displays an, wann einer der ersten fünf Flags gesetzt ist. Deshalb habe ich die beiden ersten (sehr wichtigen) Flags für zwei Systemeinstellungen meiner Programme benutzt.

- Flag 1 schaltet das Programm EXCO ein und aus.
- Flag 2 schaltet die Langversion von DE ein und aus.

Man kann also in der ersten Zeile sofort erkennen, welche Systemeinstellung gewählt ist. Einige Programme schalten die Flags während des Ablaufs allerdings selbsttätig an und aus. Am Ende dieser Programme wird der Flag 2 immer gesetzt (also die Langversion von DE aktiviert). Diese Praxis hat sich bei mir am ehesten bewährt. Dies ist bei EXCO nicht der Fall. EXCO stört insbesondere dann, wenn sehr viel rein numerisch gerechnet wird oder unsinnig oft ausmultipliziert werden muß. In der Regel sollte man aber auch EXCO einschalten (1 SF).

Die anderen Flags werden von einigen Programmen intern genutzt. Die Vorbelegung ist hierbei egal. Allerdings ist der Endzustand nicht definiert.

8.1.4 Weitere Benutzerhinweise

Um die Programme optimal nutzen zu können, muß zuallererst ein gewisser Grundstock vorhanden sein. Auf jeden Fall eingegeben werden sollten alle Programme im Abschnitt „Basis-Hilfsprogramme“. Auch die Programme aus der „allgemeinen Ingenieursmathematik“, sowie "XPAC" und "dc" aus dem Abschnitt „Kompressions-Programme“, sind empfehlenswert. Wenn diese Programme alle im HOME-Directory abgespeichert sind, ist damit gewährleistet, daß alle Programme in allen Unter-Verzeichnissen genutzt werden können.

Dringend empfohlen wird das Arbeiten in einem leeren Arbeits-Verzeichnis. Der Grund liegt darin, daß der normale User bei Wahl eines Variablennamen u.U. ein Programm überschreibt. Dies ist in leeren Unterverzeichnissen nicht möglich. Gewarnt wird davor, Großbuchstaben als Variablennamen zu wählen, da etliche Programme (z.B. PART) diese überschreiben. Sicher ist die Wahl einer Kombination aus Buchstaben und Zahlen, da diese nur als 'uX' bei LINA verwendet wurden. X ist hierbei eine beliebige Zahl. Es kann keine Gewährleistung für Datenverlust übernommen werden, wenn die Programme nicht bestimmungsgemäß eingesetzt werden.

8.2 Programme

8.2.1 Basis-Hilfsprogramme

SYS

Version:	HP28S
Checksum(HP28S):	1517891 (bei Basis HEX)
Checksum(HP48):	entfällt
Bytes:	220.5

1: —	1: —

Erläuterung: Dieses Programm ist nur für den HP28S bestimmt. Das Programm installiert ein Programm 'FA', das den Rechner beim Aufruf des Programms doppelt so schnell macht, als dieser üblicherweise wäre. Nach einiger Zeit hebt der Rechner durch interne Routinen die Wirkung des Programms wieder auf, so daß er wieder normal langsam wird. Alle Programme sind genau so ausgelegt, daß das Programm aufgerufen wird, wenn es sinnvoll erscheint, und zwar nur dann.

Beispiel: entfällt

benötigte UP: keine

```
≪ HOME HEX
"962A60C9022300103"
"3F100FF3FF0510D31"
"11246184800CF902" +
+ "" 1 5 SQ
  FOR i "#" 3 PICK i
2 * DUP 1 - SWAP SUB
+ STR→ B→R CHR +
  NEXT ´μ´ DUP DUP
PURGE 4 ROLLD STO
DROP # CFFCEh
SYSEVAL 1 GET ´FA´
STO PURGE
≫
```

FA

Version: HP48 und HP28S
Checksum(HP28S): 441
Checksum(HP48): FCEE
Bytes: 16.5

≪ ≫

1: nichts	1: nichts

Erläuterung: Stellt einen Platzhalter für ein softwaremäßiges Beschleunigungsprogramm dar. Beim HP28S ist das Programm für den Ablauf des Programmes CHK nötig.
Beispiel: entfällt
benötigte UP: keine

CHK

Version: HP28S
Checksum(HP28S): 209565
Checksum(HP48): entfällt
Bytes: 79.0

```
≪ FA RCL →STR 0 3 3
PICK SIZE
  FOR i OVER i DUP
SUB NUM i * +
  NEXT SWAP DROP
≫
```

1: Programmname des zu prüfenden Programms	1: Prüfsumme

Erläuterung: Ist ein simples Prüfsummenprogramm, das ein schnelleres Auffinden von Eingabefehlern beim Abtippen ermöglichen soll. Die im Buch angegebenen Prüfsummen beziehen sich für den HP28S auf dieses Programm. Die Prüfsummen des HP48 (S und G) wurden mit der internen Routine des HP48 ermittelt.
Beispiel: 'CHK' CHK $\Longrightarrow$ 209565
benötigte UP: FA

GE

Version: HP28S
Checksum(HP28S): 73589
Checksum(HP48): entfällt
Bytes: 49.0

```
≪ 4 FACT 7 + SF
  IFERR GET
  THEN DROP2 0
  END
≫
```

1: —	1: —

Erläuterung: Ähnlich wie GET, nur als Hilfsprogramm für ROU gedacht.
Beispiel: entfällt
benötigte UP: keine

GE

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): EE9
Bytes: 49.5

```
≪ -55 CF
  IFERR GET
  THEN DROP2 0
  END
≫
```

1: —	1: —

Erläuterung: Ähnlich wie GET, nur als Hilfsprogramm für ROU gedacht.
Beispiel: entfällt
benötigte UP: keine

cnr

Version: HP48 und HP28S
Checksum(HP28S): 46552
Checksum(HP48): 638C
Bytes: 40.0

```
≪ LIST→ 0 0 0 4
ROLL
  START +
  NEXT
≫
```

1: Liste mit Elementen	1: Summe der Listenelemente

Erläuterung: Addiert alle Elemente einer Liste auf. Achtung! Dieses Programm hat nicht ganz die gleiche Funktion wie ΣLIST beim HP48G.
Beispiel: { 1 2 0 -4 } cnr ⟹ -1
benötigte UP: keine

cnrm

Version: HP48 und HP28S
Checksum(HP28S): 71822
Checksum(HP48): BC06
Bytes: 46.0

```
≪ LIST→ 0 0 0 4
ROLL
  START SWAP ABS +
  NEXT
≫
```

1: Liste mit Elementen	1: Summe der Beträge der Listenelemente

Erläuterung: Addiert die Beträge aller Elemente einer Liste auf.
Beispiel: { 1 2 0 -4 } cnrm ⟹ 7
benötigte UP: keine

KO

Version: HP48 und HP28S
Checksum(HP28S): 53982
Checksum(HP48): 49AE
Bytes: 44.5

```
≪ LIST→ DUP DUP 2 +
ROLLD 1 + TA →LIST
≫
```

1: Liste	1: Liste mit verdrehten Elementen

Erläuterung: Wendet die Elemente einer Liste von hinten nach vorn.
Beispiel: { 1 2 3 4 } KO ⟹ { 4 3 2 1 }
benötigte UP: TA

LI

Version: HP48 und HP28S
Checksum(HP28S): 113927
Checksum(HP48): 3314
Bytes: 51.5

```
≪ DUP
  WHILE DUP
  REPEAT 1 - 0 ROT
ROT
  END DROP →LIST
≫
```

1: Zahl	1: Liste mit Nullen

Erläuterung: Erzeugt aus einer Zahl eine Liste mit ebensovielen Nullen.
Beispiel: 5 LI ⟹ { 0 0 0 0 0 }
benötigte UP: keine

WA

Version: HP48 und HP28S
Checksum(HP28S): 49227
Checksum(HP48): 3164
Bytes: 36.5

```
≪
  DO
  UNTIL KEY
  END 2 ALOG 1 BEEP
≫
```

1: nichts	1: Name bzw. Code der gedrückten Taste

Erläuterung: Wartet auf einen Tastendruck, piepst und gibt den Namen der gedrückten Taste an.

Beispiel: Den Befehl WA eingeben, dann ENTER für die Befehlsübergabe und anschließend noch ein zweites Mal ENTER drücken ⟹ "ENTER"
Der HP48 liefert hier nur eine Zahl, die der Taste entspricht (hier 51).

benötigte UP: keine

pos

Version: HP48 und HP28S
Checksum(HP28S): 41904
Checksum(HP48): 5A9D
Bytes: 35.0

```
≪ POS DUP
  IF NOT
  THEN INV
  END
≫
```

1: —	1: —

Erläuterung: Wie der Befehl „POS", nur daß statt einer Null ein 9.99999999E499 zurückgegeben wird. Ist nur als Unterprogramm konzipiert.

Beispiel: 2: { 4 3 2 1 }
1: 2 pos ⟹ 3

benötigte UP: keine

SS

Version: HP48 und HP28S
Checksum(HP28S): 30399
Checksum(HP48): A717
Bytes: 40.0

```
≪ 8 SQ + CHR "'"
SWAP + STR→
≫
```

1: Zahl von 1 bis 26	1: zugehöriger Buchstabe des ABCs

Erläuterung: Verwandelt eine Integer-Zahl in den entsprechenden Buchstaben des Alphabets.
Beispiel: 1 SS $\Longrightarrow$ 'A'
benötigte UP: keine

ST

Version: HP48 und HP28S
Checksum(HP28S): 125612
Checksum(HP48): 6338
Bytes: 57.5 bzw. 60.0

```
≪ ""
  WHILE OVER
  REPEAT "0" + SWAP
1 - SWAP
  END SWAP DROP
≫
```

1: Zahl	1: String mit Nullen

Erläuterung: Erzeugt einen String mit Nullen.
Beispiel: 5 ST $\Longrightarrow$ "00000"
benötigte UP: keine

TA

Version: HP48 und HP28S
Checksum(HP28S): 33899
Checksum(HP48): D050
Bytes: 38.0

```
≪ 1 SWAP
  FOR f f ROLL
  NEXT
≫
```

2-n:	n-1 Elemente	1-(n-1):	n-1 Elemente,
1:	Anzahl der zu stürzenden Elemente		nur gestürzt

Erläuterung: Erlaubt das Auf-den-Kopf-Stellen beliebig vieler Elemente im Stack.

Beispiel:

5: 'D'		
4: 'C'		4: 'D'
3: 'B'		3: 'A'
2: 'A'		2: 'B'
1: 3 TA	$\Longrightarrow$	1: 'C'

benötigte UP: keine

AU

Version: HP48 und HP28S
Checksum(HP28S): 249291
Checksum(HP48): C2AD
Bytes: 79.5

```
≪ OVER SIZE OVER
SIZE DUP2 - ABS LI
SWAP ROT
  IF ≥
  THEN ROT + SWAP
  ELSE SWAP +
  END
≫
```

2:	Liste	2:	aufgefüllte Liste
1:	Liste	1:	aufgefüllte Liste

Erläuterung: Prüft die Länge zweier Listen und füllt die kürzere von vorn mit Nullen auf (wurde als Hilfsprogramm konzipiert).

Beispiel:

2: { 1 2 3 }	
1: { 1 2 3 4 5 } AU	$\Longrightarrow$
2: { 0 0 1 2 3 }	
1: { 1 2 3 4 5 }	

benötigte UP: LI

EK

Version: HP48 und HP28S
Checksum(HP28S): 55843
Checksum(HP48): 2F8F
Bytes: 39.5

1: Liste	1: Liste, deren hintere Nullen abgeschnitten wurden

Erläuterung: Schneidet in einer Liste alle sich am Schluß befindenden Nullen ab. Achtung! Zahlen, deren Betrag kleiner als .0001 ist, werden ebenfalls abgeschnitten.
Beispiel: { 1 2 3 0 0 } EK ⟹ { 1 2 3 }
benötigte UP: ES

```
≪
  DO DUP ES DUP ROT
  UNTIL SAME
  END
≫
```

ES

Version: HP48 und HP28S
Checksum(HP28S): 653751
Checksum(HP48): 5674
Bytes: 109.0

1: Liste	1: Liste, gekürzt um evtl. vorhandene Null am Ende

Erläuterung: Schneidet eine evt. vorhandene Null am Ende der Liste ab. Es bleibt aber immer mindestens noch eine Zahl/ein Ausdruck in der Liste stehen. Achtung! Zahlen, deren Betrag kleiner als .0001 ist, werden ebenfalls abgeschnitten!
Beispiel: { 1 2 3 0 0 } ES ⟹ { 1 2 3 0 }
benötigte UP: keine

```
≪ LIST→
  IF DUP
  THEN OVER TYPE 1
    IF ≤
    THEN OVER ABS
-4 ALOG < OVER 1 ≠
      IF AND
      THEN + ABS 1
-
      END
    END
  ELSE 1
  END →LIST
≫
```

8.2.2 Anwenderorientierte Hilfsprogramme

EXCO

Version: HP48 und HP28S
Checksum(HP28S): 650467
Checksum(HP48): 40DE
Bytes: 96.5

1: ausführlicher Term	1: zusammengefaßter Term

Erläuterung: Faßt einen mathematischen Ausdruck vollständig zusammen, nachdem das Programm alles ausmultipliziert hat.
1 CF schaltet diese Funktion ab.
1 SF schaltet diese Funktion an.
Dieses Programm wird von vielen anderen Programmen aufgerufen. Damit man erkennt, daß der Rechner sich nicht aufgehängt hat, erzeugt er bei jedem Ausführen von EXCO ein Ticken. Dies kann bei Nichtgefallen aus dem Programm entfernt werden.
Beispiel: '4*A+B*B-4*A' EXCO ⟹ B ^ 2
benötigte UP: FA

```
≪ FA 7 EXP DUP INV
BEEP 1
  IF FS?
  THEN COLCT
    DO DUP EXPAN
    UNTIL DUP ROT
SAME
    END
    DO DUP COLCT
    UNTIL DUP ROT
SAME
    END
  END
≫
```

CD

Version: HP28S
Checksum(HP28S): 35435
Checksum(HP48): entfällt
Bytes: 31.5

1: —	1: —

Erläuterung: Wechselt beim HP28S exakt um ein Directory nach oben.
Beispiel: entfällt
benötigte UP: keine

```
≪ 1 PATH LIST→ ROT
EVAL DROPN
≫
```

LOE

Version: HP48 und HP28S
Checksum(HP28S): 141444
Checksum(HP48): CD44
Bytes: 147.5

```
≪ FA { A B C D E F
G H I J K L M N O P
Q R S T U V W X Y Z
} PURGE
≫
```

1: —	1: —

Erläuterung: Löscht alle Programme und Variablen, die aus nur einem Buchstaben bestehen aus dem aktuellen Directory. Dies ist praktisch, da einige Programme manchmal ziemlich viele solcher Variablen hinterlassen.
Beispiel: entfällt
benötigte UP: FA

EX

Version: HP48 und HP28S
Checksum(HP28S): 122022
Checksum(HP48): 3A10
Bytes: 54.0

```
≪
  DO DUP EVAL DUP
ROT
  UNTIL SAME
  END EXCO 6 FACT 1
BEEP
≫
```

1: ausführlicher mathematischer Term	1: eingesetzter mathematischer Term

Erläuterung: Setzt zuerst alle Variablen in einen mathematischen Term ein, faßt dann zusammen und pfeift ganz am Schluß. Ideal um längere Terme ausrechnen zu lassen.
Beispiel: zuerst 2 'B' STO 'A' PURGE
'B*B+A' EX $\Longrightarrow$ '4+A'
benötigte UP: EXCO

EVV

Version: HP48 und HP28S
Checksum(HP28S): 1590340
Checksum(HP48): DC32
Bytes: 165.0

1: Matrix oder Polynom in Listenschreibweise	1: Matrix oder Polynom mit eingesetzten Variablen

Erläuterung: Setzt so viele Elemente in die Matrix (oder das Polynom) in Listen-Schreibweise ein, wie nur irgendwie möglich, bzw. es verwandelt die Schreibweise von DE (siehe dort) in die rechnerinterne REAL-Zahl-Darstellung. Letzteres hat folgenden Sinn: Multipliziert man z.B. Polynome oder Matrizen miteinander, werden Ausdrücke wie '1/2' sehr langsam verarbeitet, was durch vorherige Behandlung mit EVV verhindert werden kann.

Beispiel: { { '1/2' '1/4' } } EVV ⟹
{ { .5 .25 } }

benötigte UP: keine

```
≪ DUP → m
  ≪ TYPE 5
    IF ≠
    THEN 1 CHR m
      IFERR →NUM
      THEN
        DO
        UNTIL 1 CHR
SAME
        END m EVAL
      ELSE SWAP
DROP
      END
    ELSE 1 m SIZE
      FOR j m j GET
EVV
      NEXT m SIZE
→LIST
    END
  ≫
≫
```

RAUS

Version:	HP48 und HP28S
Checksum(HP28S):	252450
Checksum(HP48):	61B5
Bytes:	69.0

1:	nichts	1-n:	Programme und Variablen(namen)

Erläuterung: Legt alle Programme und Variablen eines Directory in den Stack, wobei das Ablegen in den Stack so erfolgt, daß erst das Programm (oder die Variable) und dann als zweites der Name in den Stack geschrieben wird. Sinn des Programms ist eine Vereinfachung des Transports von Variablen von einem Directory in ein anderes.
Achtung! Dieses Programm bricht sofort ab, wenn ein nicht leeres Directory verlegt werden soll.
Der HP48G verfügt aber mittlerweile über ein sehr ausgereiftes Management der Dateien, so daß dieses Programm nicht unbedingt nötig ist.

Beispiel: entfällt

benötigte UP: FA

```
≪ FA VARS 1 1 3
PICK SIZE
  START GETI DUP
RCL OVER PURGE 4
ROLLD ROT ROT
  NEXT DROP2
≫
```

OFF

Version: HP28S
Checksum(HP28S): 14946 (bei Basis HEX)
Checksum(HP48): entfällt
Bytes: 33.0

```
≪ # 18E58h SYSEVAL
≫
```

1: —	1: —

Erläuterung: Schaltet den Taschenrechner aus. Ist ganz sinnvoll, wenn der Computer sich nach längerer Rechenzeit selbst ausschalten soll. Das Ausschalten ist also softwaremäßig steuerbar.
Achtung! Das Programm läuft nur auf einem HP28S.
Ach ja - ein kleiner Gag: wenn man eine Schleife programmiert, die den Rechner mehrere Male ausschaltet, muß man den Rechner erst mehrere Male einschalten, bevor er wieder normal angeht.
z. B.:

```
≪ 1 10
    START OFF
    NEXT
≫
```

Beispiel: entfällt
benötigte UP: keine

SEE

Version: HP48 und HP28S
Checksum(HP28S): 601309
Checksum(HP48): 4E88
Bytes: 110.5

1: Liste mit den zu durchsuchenden Programmen	1-n: n verschiedene Programmnamen

Erläuterung: Stellt eine Erleichterung dar, wenn man in Programmen (oder Variablen) nach einem bestimmten Ausdruck sucht.
Man muß im Programm 'SEE' jeweils den Suchstring so ändern, daß der gewünschte Ausdruck zwischen den beiden "-Zeichen liegt. Im hier vorliegenden Fall wäre dies FA.
Vor dem Start muß man in einer Liste die Namen der Programme schreiben, die durchsucht werden sollen. (Tip: Der Befehl VARS erleichtert diese Arbeit ganz massiv). In der Liste dürfen folgende Namen nicht auftauchen: 'FA' und alle Namen von Directorys. Beim HP48 werden sonst diese Verzeichnisse und deren Unterverzeichnisse ebenfalls durchsucht, was recht zeitaufwendig ist; und selbst beim Finden des Ausdrucks wird nur das Directory angezeigt, in dem der Rechner den Begriff fand, was ja nicht sehr aufschlußreich ist.

Beispiel: entfällt

benötigte UP: FA

```
≪ 1 DUP 3 PICK SIZE
  START FA DUP2 GET
DUP 1 DISP OVER 2
DISP RCL →STR "FA"
    IF POS
    THEN DUP2 GET
ROT ROT EX
    END 1 +
  NEXT DROP2
≫
```

TI

Version:	HP28S
Checksum(HP28S):	Angabe nicht möglich
Checksum(HP48):	eingebaute Funktion des HP48
Bytes:	70.5

```
≪ # 11CAh SYSEVAL
B→R 29491200 / →HMS
XX.XXXXX HMS+ 4 FACT
MOD
≫
```

1: nichts	1: Uhrzeit

Erläuterung: Gibt die jeweils gültige Uhrzeit an. Diese Uhr muß allerdings noch gestellt werden. Das XX.XXXX ist eine Uhrzeit von 0 bis 24 Uhr, die den Fehler, um den die Rechneruhr falsch geht, ausgleicht. Das Eingabeformat ist z.B. 12.3442, wenn die Uhr um 12 Std 34 Min und 42 Sek nachgeht.
Zum Stellen der Uhr setzt man XX.XXXX zuerst auf den Wert 0, errechnet sich dann den Fehler und ersetzt den Wert 0 wieder durch den errechneten Fehler. Ab diesem Zeitpunkt müßte die Uhr korrekt gehen (auch nach dem Abschalten des Rechners).

Beispiel: entfällt

benötigte UP: keine

8.2.3 Allgemeine mathematische Programme

LD

Version: HP48 und HP28S
Checksum(HP28S): 26810
Checksum(HP48): F13F
Bytes: 43.0

```
≪ → x
  ≪ x LOG 2 LOG /
  ≫
≫
```

1: REAL-Zahl	1: Logarithmus Dualis aus der REAL-Zahl

Erläuterung: Berechnet den Logarithmus Dualis aus einer REAL-Zahl.
Für Programmierer: das Programm ist etwas kompliziert programmiert, um eine Konstruktion 'LD(24)' zuzulassen. Man kann also den Log. dualis genauso wie Sinus, Cosinus etc in Formeln einsetzen.
Beispiel: 100 LD $\Longrightarrow$ 6.64385618977
benötigte UP: keine

QUERG

Version: HP48 und HP28S
Checksum(HP28S): 145766
Checksum(HP48): 50B5
Bytes: 85.0

```
≪ LIST→ PICK / SWAP
3 PICK 2 * / DE 'X'
OVER + SQ ROT ROT
SQ - DE + *
≫
```

1: Polynom 2. Grades	1: Polynom in quadratisch ergänzter Darstellung

Erläuterung: Führt die quad. Ergänzung mit einem Polynom zweiten Grades (in Listen-Schreibweise) durch.
Beispiel: { 1 2 3 } QUERG $\Longrightarrow$ 'SQ(X+1)+2'

benötigte UP: DE

Das Programmpaket DE (mit den Unterprogrammen FE, de, De, dE, DEZ und SQR) gehört mit zu den durchdachtesten Programmen, die in diesem Buch enthalten sind. Es ist durch jahrelange Verbesserungen zu einem recht leistungsfähigen Programm geworden. Entstanden ist es, da ich mich früher ziemlich ärgerte, daß der HP28S nicht mit Brüchen rechnen kann, was schon bei einfachen Zahlen wie $\frac{3}{7}$ zu häßlichen Ergebnissen führt.

Das Programm führt nach einem bestimmten Algorithmus eine Suche durch, nach dem für eine REAL-Zahl eine entsprechende Schreibweise als Bruch, Wurzel aus einem Bruch, Summe aus Bruch und Wurzel etc. gesucht wird. Dem Programm wurde beigebracht, kleine Rundungsfehler nicht zu berücksichtigen. Welche Auswirkung das hat, können insbesondere HP48-User mit .42857142856 →Q ausprobieren und das Ergebnis mit DE nachprüfen.

Das Programm DE erkennt selbsttätig, ob es sich um eine Zahl, ein Polynom, eine Matrix oder eine gebrochen rationale Funktion (jeweils in Listen-Schreibweise) handelt und ist auch ansonsten relativ fehlerresistent. Einen Nachteil hat das Programm gegenüber dem Befehl →Q: Ausdrücke wie zum Beispiel '.25*A' können nicht direkt vom Programm bearbeitet sondern müssen erst zerlegt werden.

Dieses Programm ist durch das Setzen des Flags 2 verschieden intelligent. In der Kurzversion erkennt es nur einfache Brüche, Wurzeln aus Brüchen, LN von Wurzel aus einem Bruch etc. Dafür ist es dann sehr schnell beendet.

DE

Version:	HP48
Checksum(HP28S):	entfällt
Checksum(HP48):	FB26
Bytes:	159.0

1: REAL-Zahl	1: Wurzel-Bruch-Darstellung

Erläuterung: Siehe Erläuterung für die Version des HP28S.

Beispiel:
.701560760018 DE ⟹ '3*√7/(8*√2)'
.129436325678 DE ⟹ '62/479'
1.60943791243 DE ⟹ 'LN(5)'
.261799387799 DE ⟹ '1/12*π'
-.629960524948 DE ⟹ '-(1/4)^(1/3)'
25.2477084857 DE ⟹ '1/113+7*√13'

benötigte UP: FE

```
≪ DUP → m
  ≪ TYPE 5
    IF ≠
    THEN m DUP TYPE
2
      IF <
      THEN RE FE m
IM FE i * +
      END
    ELSE 1 m SIZE
      FOR j m j GET
DE
      NEXT m SIZE
→LIST
    END
  ≫
≫
```

DE

Version: HP28S
Checksum(HP28S): 969233
Checksum(HP48): entfällt
Bytes: 161.5

1: REAL-Zahl	1: Wurzel-Bruch-Darstellung

Erläuterung: Stellt REAL-Zahlen als Brüche, Wurzeln und deren Summe dar. Zusätzlich werden noch einige andere Darstellungsmöglichkeiten überprüft (siehe Beispiele).
Das Programm teilt sich in zwei Bereiche. Der erste Teil sucht nach etwas simpleren Ausdrücken und ist sehr schnell fertig, der zweite sucht nach einem Algorithmus nach der Summe aus zwei Ausdrücken, was etwas länger dauert. Aus diesem Grunde ist die Lang-Version mit 2 CF ausschaltbar (mit 2 SF anschaltbar).

Beispiel:
.701560760018 DE ⟹ '3*√7/(8*√2)'
.129436325678 DE ⟹ '62/479'
1.60943791243 DE ⟹ 'LN(5)'
.261799387799 DE ⟹ '1/12*π'
-.629960524948 DE ⟹ '-(1/4)^(1/3)'
25.2477084857 DE ⟹ '1/113+7*√13'

benötigte UP: FE

```
≪ EVAL DUP → m
  ≪ TYPE 5
    IF ≠
    THEN m DUP TYPE
2
      IF <
      THEN RE FE m
IM FE i * +
      END
    ELSE 1 m SIZE
      FOR j m j GET
DE
      NEXT m SIZE
→LIST
    END
  ≫
≫
```

de

Version: HP48 und HP28S
Checksum(HP28S): 191392
Checksum(HP48): FFCF
Bytes: 74.5

1: —	1: —

Erläuterung: Reines Hilfsprogramm für DE.
Beispiel: entfällt
benötigte UP: FE

```
≪ SWAP FE →STR DUP
NUM 6 SQ 3 +
  IF ==
  THEN 2 4 FACT SUB
  END + STR→ OVER
≫
```

SQR

Version: HP48 und HP28S
Checksum(HP28S): 1411569
Checksum(HP48): DD91
Bytes: 162.5

1: REAL-Zahl	1: Wurzel-Darstellung

Erläuterung: Stellt eine REAL-Zahl als Wurzel aus einem Bruch dar.
Beispiel: .261861468284 SQR ⟹ '2*√3/(5*√7)'
benötigte UP: dE, DEZ

```
≪ DUP SIGN SWAP SQ
dE 1 2
  START 7 SQ OVER √
IP MIN 1 MAX
    WHILE DUP2 SQ
MOD OVER 2 ≥ AND
    REPEAT 1 -
    END SWAP OVER
SQ / DUP 1
    IF ==
    THEN /
    ELSE "'√" SWAP
→STR + STR→ *
    END SWAP
  NEXT SWAP / *
≫
```

DEZ

Version: HP48 und HP28S
Checksum(HP28S): 313001
Checksum(HP48): EE83
Bytes: 87.5

1: REAL-Zahl	1: Bruch-Zahl

Erläuterung: Stellt eine REAL-Zahl als Bruch dar.
Beispiel: .3333333333333 DEZ $\Longrightarrow$ '1/3'
benötigte UP: dE

```
≪ DUP SIGN SWAP ABS
dE ROT * "´" SWAP
→STR + "/" + OVER
→STR + STR→ SWAP 1
  IF ==
  THEN EVAL
  END
≫
```

De

Version: HP48 und HP28S
Checksum(HP28S): 418509
Checksum(HP48): D520
Bytes: 99.5

1: —	1: —

Erläuterung: Reines Hilfsprogramm für DE.
Beispiel: entfällt
benötigte UP: dE, XPAC

```
≪ SWAP →NUM SQ dE
XPON SWAP XPON +
ABS 5
  IF <
  THEN 6
    IF FC?
    THEN XPAC EVAL
SWAP ROT 6 SF
    END
  END DROP
≫
```

FE

Version: HP48 und HP28S
Checksum(HP28S): 27850481
Checksum(HP48): BB23
Bytes: 814.5

1: —	1: —

Erläuterung: Reines Hilfsprogramm für DE.
Beispiel: entfällt
benötigte UP: DEZ, de, De, FA, SQR

```
≪ FA →NUM
  IF DUP
  THEN DUP DEZ →STR
SIZE 6 6 +
    IF >
    THEN DUP SQ DEZ
→STR SIZE 6 6 +
      IF >
      THEN 6 CF DUP
SIGN SWAP ABS DUP
LN
        ≪ LN
"´EXP(" de
        ≫ De DUP
EXP
        ≪ EXP
"´LN(" de
        ≫ De DUP π
* "ZCNZDH" De DUP π
/ "ZDNZCH" De DUP π
SQ / "ZQDNZQCH" De
DUP ALOG
        ≪ ALOG
"´LOG(" de
        ≫ De * 2
        IF FS?
        THEN 0
          DO FA 4
INV + DUP2 INV -
"IKBNHNA" De DUP2
INV + "IKANHNB" De
DUP2 √ - "[KBNHNA"
De DUP2 √ +
"[KANHNB" De
          UNTIL DUP
1 ALOG ≥ 6 FS? OR
          END DROP
1
          DO FA 1 +
DUP2 √ MOD
```

```
"[K]PHBNHNA" De
DUP2 √ SWAP OVER
MOD - "[K]]KANHNB"
De
          UNTIL DUP
4 SQ == 6 FS? OR
          END DROP
0
          WHILE DUP
1 ALOG ≤ 6 FC? AND
          REPEAT FA
4 INV + DUP2 -
"KBNHNA" De DUP2 +
"KANHNB" De DUP2
INV √ - "I[KBNHNA"
De DUP2 INV √ +
"I[KANHNB" De
          END DROP
DUP ABS 2 5
          FOR w DUP
w ^ "HOHbFNbINFC"
De
          NEXT DROP
        END
      ELSE SQR
      END
    ELSE DEZ
    END
  END
≫
```

dE

Version:	HP48 und HP28S
Checksum(HP28S):	486021
Checksum(HP48):	3B89
Bytes:	99.5

1: —	1: —

Erläuterung:	Reines Hilfsprogramm für DE.
Beispiel:	entfällt
benötigte UP:	keine

```
≪ 9 OVER XPON 1 MAX
- ALOG SWAP OVER *
DUP2
  DO OVER MOD SWAP
  UNTIL OVER 50 <
  END SWAP DROP ROT
OVER / IP ROT ROT /
IP
≫
```

NULL

Version: HP48 und HP28S
Checksum(HP28S): 18210330
Checksum(HP48): A840
Bytes: 580.5

1: Polynom in Listen-Schreibweise	1-n: Faktoren ersten Grades des Polynoms

Erläuterung: Berechnet die Nullstellen (bzw. Faktorenzerlegung) eines Polynoms (in Listen-Schreibweise) beliebigen Grades.
Achtung! Leider nur ein rein numerisches Programm.
Für User des HP48G! Dieser Algorithmus ist insbesondere dann dem im HP48G implementierten Verfahren überlegen, wenn mehrere Nullstellen gleich sind (siehe auch Beispiel).

Beispiel: { 2 17 27 -26 -44 24 } NULL ⟹
{ 1 -.5 }
{ 1 2 }
{ 1 -1 }
{ 1 2 }
{ 1 6 } in beliebiger Reihenfolge

benötigte UP: FA, ABL, FUNC, NU, DIV, QU, TA

```
≪ 2 CF DUP SIZE 4
  IF ≥
  THEN 1 OVER SIZE
3 -
    START FA DUP
DUP ABL DUP ABL
RAND
      WHILE 4 PICK
OVER FUNC DUP ABS 2
INV ≥
      REPEAT 3 DUPN
DROP FUNC 5 PICK 4
PICK FUNC OVER /
NEG DUP SQ 4 ROLL 4
ROLL / 2 * - √ DUP2
- SWAP ROT + OVER
ABS OVER ABS ≥ SWAP
ROT IFTE +
      END DROP DUP
5 ROLLD
      DO
.0000000001 6 ROLLD
        WHILE FA 4
PICK OVER FUNC DUP
ABS 8 ROLL 3 * DUP
9 ROLLD ≥
        REPEAT 3
DUPN DROP FUNC 5
PICK 4 PICK FUNC
OVER / NEG DUP SQ 4
ROLL 4 ROLL / 2 * -
√ DUP2 - SWAP ROT +
OVER ABS OVER ABS ≥
SWAP ROT IFTE +
        END DROP
      UNTIL 3 PICK
OVER FUNC ABS .05 ≤
3 PICK SIZE 1 >
        IF AND
        THEN 6 TA 3
DROPN SWAP DUP ABL
```

```
4 PICK 0
       ELSE 1
       END
     END 6 ROLLD 5
DROPN NU DIV ROT
DROP
   NEXT
 END DUP SIZE 3
 IF ==
 THEN QU
 END 2 SF
≫
```

FUNC

Version: HP48 und HP28S
Checksum(HP28S): 204741
Checksum(HP48): FE07
Bytes: 77.5

```
≪ 0 ROT LIST→ 3 +
DUP ROLL 0 ROT 4
  FOR i i ROLL +
OVER * -1
  STEP SWAP DROP +
≫
```

2: Polynom in Listen-Schreibweise 1: Argument	1: Funktionswert an der Argumentenstelle

Erläuterung: Berechnet an einem Argumentenwert den Funktionswert eines Polynoms (in Listen-Schreibweise).

Beispiel: 2: { 1 2 3 }
1: 2 FUNC $\Longrightarrow$ 11

benötigte UP: keine

NU

Version: HP48 und HP28S
Checksum(HP28S): 32362
Checksum(HP48): 676
Bytes: 37.0

```
≪ DE →NUM NEG 1
SWAP 2 →LIST
≫
```

1: beliebiger Ausdruck 'AUSDR'	1: { 1 '-AUSDR' }

Erläuterung: Verwandelt die Darstellung. Dafür gedacht, daß eine Nullstelle als Faktor geschrieben wird.
Beispiel: 3 NU ⟹ { 1 -3 }
benötigte UP: DE

DUAL

Version: HP48 und HP28S
Checksum(HP28S): 948554
Checksum(HP48): 8069
Bytes: 165.0

```
≪ DUP "." POS "#0"
3 PICK +
  IF OVER
  THEN 1 3 PICK 1 +
SUB "b" + STR→ B→R
ROT ROT 1 + OVER
SIZE SUB "#" OVER
"b" + + STR→ B→R 2
ROT SIZE ^ / DE
  ELSE SWAP ROT
DROP
  END +
≫
```

1: Binär-Zahl als String	1: Dezimal-Zahl

Erläuterung: Rechnet eine binäre Zahl (mit Nachkommastellen) in eine Dezimal-Zahl um.
Beispiel: "1001.1001011"DUAL ⟹ '9+75/128'
Achtung! Vor und nach dem Kommapunkt muß jeweils mindestens eine Zahl stehen!
benötigte UP: DE

TAY

Version:	HP48 und HP28S
Checksum(HP28S):	632056
Checksum(HP48):	F004
Bytes:	138.5

4: zu entwickelnde Funktion 3: unabh. Variable 2: Grad bis zu dem entwickelt werden soll 1: Stelle an der entwickelt werden soll	1: entwickelte Funktion

```
≪ FA
  IF DUP
  THEN ´μ´ OVER + 4
PICK STO 4 ROLL
EVAL ´μ´ 4 ROLL
TAYLR 3 PICK ROT -
´μ´ STO EVAL { μ }
ROT + PURGE
  ELSE DROP TAYLR
  END
≫
```

Erläuterung: Führt die Taylor-Entwicklung in einem beliebigen Punkt durch. Dies ist einfacher zu handhaben, als die Erklärung im Buch des HP48G nachzurechnen.

Beispiel: RAD [ENTER]
4: 'SIN(X)'
3: 'X'
2: 2
1: 3 TAY ⟹
'.14112000806-.9899924966*(X-3)-
.07056000403*(X-3)^2'

benötigte UP: FA

PART

Version:	HP28S
Checksum(HP28S):	20714795
Checksum(HP48):	entfällt
Bytes:	730.5

1: —	1: —

Erläuterung: Führt die Partialbruchzerlegung durch. Dieses Programm verfügt über eine kleine Menüsteuerung. Beim Aufruf erscheint ein User-Menü in der untersten Zeile. Die Ein- und Ausgaben erfolgen jeweils als Polynome (in Listen-Schreibweise).
In die Variable Z speichert man den Zähler der zu zerlegenden Funktion. In A wird der erste Nennerfaktor, in B der zweite etc. abgespeichert. Jeder Nennerfaktor darf nur einmal in A, B, C etc. gespeichert sein. Die Vielfachheit der Nennerfaktoren wird dem Programm über V mitgeteilt. Nennerfaktoren, die die Buchstaben E, F, G, H etc. benötigen, müssen per Hand abgespeichert werden, da in der Menüleiste nur vier Nennerfaktoren berücksichtigt werden konnten.
Die zu zerlegende Funktion darf Variablen enthalten, allerdings nur Variablennamen, die auf keinen Fall einen einzelnen Großbuchstaben als Namen tragen.

Beispiel: siehe unten

benötigte UP: SS, MUL, DIV, QUA, FUNC, DE, DI, EXCO

```
≪ { STO Z A B C D V
} MENU HALT V 1 + Z
→ V Z
  ≪ Z { 1 } V SIZE
DUP2 SS STO 1 SWAP 1
-
    FOR i 1 V i GET
      START i SS
EVAL MUL
      NEXT
    NEXT DIV SWAP
DROP SWAP 'Z' STO
HALT 1 V SIZE 1 -
    FOR i 1 V i GET
      START i SS
EVAL DUP SIZE 3
        IF ==
        THEN QUA
DROP
        ELSE 2 GET
NEG
        END V i 0
PUT { 1 } OVER SIZE
1 - 1 SWAP
        FOR e OVER e
GET
          WHILE DUP
          REPEAT e
SS EVAL ROT MUL SWAP
1 -
          END DROP
        NEXT SWAP
DROP SWAP DUP2 FUNC
Z 3 PICK FUNC SWAP /
EXCO i SS EVAL SIZE
3
        IF ==
        THEN DUP IM
3 PICK IM / SWAP RE
OVER 4 ROLL RE * - 2
        ELSE SWAP
```

```
DROP 1
        END →LIST
SWAP ´Y´ STO DUP ´W´
STO DE i SS EVAL DE
V i GET HALT V i GET
2
        IF ≥
        THEN V Z W Y
MUL DI i SS EVAL DIV
´Z´ STO DROP2 i DUP2
GET 1 - PUT ´V´ STO
        END
      NEXT
    NEXT
  ≫
≫
```

Beispiel: Es soll die Zerlegung von $\frac{s^8}{(s+1)(s+2)^2(s^2+2s+3)(s+i)(s-i)}$ gezeigt werden.

Erster Aufruf: PART [ENTER]. Es erscheint die Store-Menü-Leiste.
Eingabe: { 1 0 0 0 0 0 0 0 0 } 'Z'-STO-Taste

{ 1 1 } 'A'-STO-Taste
{ 1 2 } 'B'-STO-Taste
{ 1 2 3 } 'C'-STO-Taste
{ 1 (0,1) } 'D'-STO-Taste
{ 1 (0,-1) } 'E' STO [ENTER]
{ 1 2 1 1 1 } 'V'-STO-Taste für die Vielfachheiten. Hier ist der zweite Wert 2, da der Faktor 'B' quadratisch im Nenner vorkommt.

Nun [CONT] ausführen. Nach einiger Rechenzeit erscheint { 1 (-7,0) } im Stack, und der Rechner wartet auf ein weiteres [CONT]. Dieses Ergebnis ist der abdividierte Rest, also ein ganz normales Polynom (in Listen-Schreibweise). Nun wieder [CONT]. Das nächste Ergebnis sieht wie folgt aus:

3: { '1/4' }
2: { 1 1 }
1: 1

Das muß interpretiert werden. In der untersten Zeile erscheint die Vielfachheit des Nenners. In der vorletzten der Nenner selbst und in der obersten der Zähler des Partialbruchs. Unser bisheriges Ergebnis sieht also wie folgt aus:

$s - 7 + \frac{\frac{1}{4}}{(s+1)^1}$

Nun sind die folgenden Ergebnisse genauso zu interpretieren wie die bisherigen. Bei

jedem Anhalten des Rechners ist nach dem Abschreiben ein CONT auszuführen. Das Gesamtergebnis sollte dann wie folgt aussehen:

$$s-7+\frac{\frac{1}{4}}{(s+1)^1}+\frac{-\frac{256}{15}}{(s+2)^2}+\frac{\frac{5888}{225}}{(s+2)^1}+\frac{\frac{11}{18}s-\frac{73}{12}}{s^2+2s+3}+\frac{-\frac{3}{200}-\frac{1}{50}*i}{(s+i)^1}+\frac{-\frac{3}{200}+\frac{1}{50}*i}{(s-i)^1}$$

PART

Version:	HP48
Checksum(HP28S):	entfällt
Checksum(HP48):	44D1
Bytes:	728.0

1: —	1: —

Erläuterung: Führt die Partialbruchzerlegung durch. Dieses Programm verfügt über eine kleine Menüsteuerung. Beim Aufruf erscheint ein User-Menü in der untersten Zeile. Die Ein- und Ausgaben erfolgen jeweils als Polynome (in Listen-Schreibweise).
In die Variable Z speichert man den Zähler der zu zerlegenden Funktion. In A wird der erste Nennerfaktor, in B der zweite etc. abgespeichert. Jeder Nennerfaktor darf nur einmal in A, B, C etc. gespeichert sein. Die Vielfachheit der Nennerfaktoren wird dem Programm über V mitgeteilt. Nennerfaktoren, die die Buchstaben E, F, G, H etc. benötigen, müssen per Hand abgespeichert werden, da in der Menüleiste nur vier Nennerfaktoren berücksichtigt werden konnten.
Die zu zerlegende Funktion darf Variablen enthalten, allerdings nur Variablennamen, die auf keinen Fall einen einzelnen Großbuchstaben als Namen tragen.

Beispiel: siehe unten

benötigte UP: SS, MUL, DIV, QUA, FUNC, DE, DI, EXCO

```
≪ { Z A B C D V }
TMENU HALT V 1 + Z
→ V Z
  ≪ Z { 1 } V SIZE
DUP2 SS STO 1 SWAP
1 -
    FOR i 1 V i GET
      START i SS
EVAL MUL
      NEXT
    NEXT DIV SWAP
DROP SWAP 'Z' STO
HALT 1 V SIZE 1 -
    FOR i 1 V i GET
      START i SS
EVAL DUP SIZE 3
        IF ==
        THEN QUA
DROP
        ELSE 2 GET
NEG
        END V i 0
PUT { 1 } OVER SIZE
1 - 1 SWAP
        FOR e OVER
e GET
          WHILE DUP
          REPEAT e
SS EVAL ROT MUL
SWAP 1 -
          END DROP
        NEXT SWAP
DROP SWAP DUP2 FUNC
Z 3 PICK FUNC SWAP
/ EXCO i SS EVAL
```

```
SIZE 3
        IF ==
        THEN DUP IM
3 PICK IM / SWAP RE
OVER 4 ROLL RE * -
2
        ELSE SWAP
DROP 1
        END →LIST
SWAP ´Y´ STO DUP
´W´ STO DE i SS
EVAL DE V i GET
HALT V i GET 2
        IF ≥
        THEN V Z W
Y MUL DI i SS EVAL
DIV ´Z´ STO DROP2 i
DUP2 GET 1 - PUT
´V´ STO
        END
      NEXT
    NEXT
  ≫
≫
```

Beispiel: Es soll die Zerlegung von $\frac{s^8}{(s+1)(s+2)^2(s^2+2s+3)(s+i)(s-i)}$ gezeigt werden.

Erster Aufruf: PART ENTER. Es erscheint die Store-Menü-Leiste.
Eingabe: { 1 0 0 0 0 0 0 0 0 } 'Z'-STO-Taste

{ 1 1 } 'A'-STO-Taste
{ 1 2 } 'B'-STO-Taste
{ 1 2 3 } 'C'-STO-Taste
{ 1 (0,1) } 'D'-STO-Taste
{ 1 (0,-1) } 'E' STO ENTER
{ 1 2 1 1 1 } 'V'-STO-Taste für die Vielfachheiten. Hier ist der zweite Wert 2, da der Faktor 'B' quadratisch im Nenner vorkommt.

Nun CONT ausführen. Nach einiger Rechenzeit erscheint { 1 (-7,0) } im Stack, und der Rechner wartet auf ein weiteres CONT. Dieses Ergebnis ist der abdividierte Rest, also ein ganz normales Polynom (in Listen-Schreibweise). Nun wieder CONT. Das nächste Ergebnis sieht wie folgt aus:

3: { '1/4' }
2: { 1 1 }
1: 1

Das muß interpretiert werden. In der untersten Zeile erscheint die Vielfachheit des Nenners. In der vorletzten der Nenner selbst und in der obersten der Zähler des Partialbruchs. Unser bisheriges Ergebnis sieht also wie folgt aus:

$$s - 7 + \frac{\frac{1}{4}}{(s+1)^1}$$

Nun sind die folgenden Ergebnisse genauso zu interpretieren wie die bisherigen. Bei jedem Anhalten des Rechners ist nach dem Abschreiben ein [CONT] auszuführen. Das Gesamtergebnis sollte dann wie folgt aussehen:

$$s - 7 + \frac{\frac{1}{4}}{(s+1)^1} + \frac{-\frac{256}{15}}{(s+2)^2} + \frac{\frac{5888}{225}}{(s+2)^1} + \frac{\frac{11}{18}s - \frac{73}{12}}{s^2+2s+3} + \frac{-\frac{3}{200} - \frac{1}{50}*i}{(s+i)^1} + \frac{-\frac{3}{200} + \frac{1}{50}*i}{(s-i)^1}$$

rot

Version:	HP48 und HP28S
Checksum(HP28S):	494747
Checksum(HP48):	81C0
Bytes:	125.0

4: erster Ausdruck	
3: zweiter Ausdruck	3: Wert der x-Koordinate
2: dritter Ausdruck	2: Wert der y-Koordinate
1: Liste mit den Variablennamen	1: Wert der z-Koordinate

```
≪ LIST→ 1 + PICK 3
PICK ∂ 6 PICK 3
PICK ∂ - EXCO 7
PICK ROT ∂ 5 ROLL 5
PICK ∂ - EXCO 5
ROLL 5 ROLL ∂ 5
ROLL 5 ROLL ∂ -
EXCO
≫
```

Erläuterung: Berechnet die Rotation eines Feldes im kartesischen Koordinatensystem.

Beispiel: RAD [ENTER] 1 SF [ENTER]
4: '2*X*Z*SIN(Y)+Y'
3: 'X^2*Z*COS(Y)+X+Z^2'
2: 'X^2*SIN(Y)+2*Z*Y'
1: { X Y Z } rot ⟹
3: 0
2: 0
1: 0

benötigte UP: EXCO

ABSCH

Version: HP48 und HP28S
Checksum(HP28S): 729913
Checksum(HP48): BB17
Bytes: 149.0

2: Liste mit abschnittsweise definierter Funktion 1: Name der unabh. Variablen	1: abschnittsweise definierte Funktion (für Rechner verständlich)

```
≪ OVER 1 GET 1 4
PICK SIZE 2 -
  FOR e 3 PICK e 2
+ GET 4 PICK e GET
- "'G1(" 4 PICK 6
PICK e 1 + GET -
→STR 2 9 ALOG SUB +
STR→ * + 2
  STEP ROT ROT
DROP2
≫
```

Erläuterung: Erzeugt aus einer Darstellung einer abschnittsweise definierten Funktion eine Funktion, die der Rechner tatsächlich versteht (z. B. beim Plotten).
Zur Beachtung! Das Programm läuft nur, wenn in der Eingabeliste mindestens drei Elemente stehen, was bei einer sinnvollen Nutzung des Programms auch nötig ist.

Beispiel: Die Funktion soll 'SIN(X)' von Minus unendlich bis 0 sein, dann bis 3 '.5*COS(X)-.5' sein und ab 3 (bis unendlich) '-.995'
Eingabe:
2: {'SIN(X)' 0 '.5*COS(X)-.5' 3 -.995}
1: 'X' ABSCH $\Longrightarrow$
'SIN(X)+(.5*COS(X)-
.5-SIN(X))*G1(X)+(
-.995-(.5*COS(X)-.5
))*G1(X-3)'

benötigte UP: G1

QUA

Version: HP48 und HP28S
Checksum(HP28S): 144165
Checksum(HP48): EBDC
Bytes: 65.0

```
≪ LIST→ PICK / SWAP
ROT 2 * / NEG DUP
SQ ROT - √ DUP2 -
SWAP ROT +
≫
```

	2: Nullstelle 2 des Polynoms
1: Polynom 2. Grades	1: Nullstelle 1 des Polynoms

Erläuterung: Löst die quadratische Gleichung eines Polynoms in Listen-Schreibweise.
Beispiel: { 2 -5 3 } QUA ⟹ 2: 1
1: 1.5
benötigte UP: keine

QU

Version: HP48 und HP28S
Checksum(HP28S): 12015
Checksum(HP48): C4D1
Bytes: 36.5

```
≪ QUA NU SWAP NU
≫
```

	2: Polynom 1. Grades
1: Polynom 2. Grades	1: Polynom 1. Grades

Erläuterung: Zerlegt ein Polynom 2. Grades (in Listen-Schreibweise) in das Produkt zweier Polynome 1. Grades und gibt diese aus.
Beispiel: { 1 2 1 } QU ⟹ 2: { 1 1 }
1: { 1 1 }
benötigte UP: NU, QUA

PHY

Version: HP48 und HP28S
Checksum(HP28S): 1955857
Checksum(HP48): AEA2
Bytes: 318.0

1: —	1: —

Erläuterung: Schreibt in das aktuelle Directory 13 wichtige physikalische Konstanten. Nachdem der HP48 sehr viele Konstanten im Konstanten-Speicher hat, ist dieses Programm am ehesten für User des HP28S interessant. Es funktioniert aber auch auf einem HP48.
Beispiel: einfach das Programm aufrufen
benötigte UP: keine

```
≪ 8.31441 'R' STO
6.022045E23 'NA'
STO 22.41383 'VMOL'
STO 6.672E-11 'GG'
STO 9.80665 'G' STO
1.6605655E-27 'U'
STO 6.626176E-34
'H' STO 299792458
'C' STO
1.380662E-23 'K'
STO 9.109534E-31
'ME' STO
1.6021892E-19 'E'
STO 8.854187818E-12
'E0' STO
1.256637061E-6 'μ0'
STO
≫
```

CROS

Version: HP48 und HP28S
Checksum(HP28S): 459706
Checksum(HP48): 2FE7
Bytes: 123.5

2: Vektor der Dimension 3 1: Vektor der Dimension 3	1: Kreuzprodukt der Vektoren

Erläuterung: Berechnet das Kreuzprodukt zweier Vektoren (in Listen-Schreibweise).
Beispiel: 2: { 1 2 3 }
1: { 7 4 2 } CROS ⟹
3: -8
2: 19
1: -10
benötigte UP: EXCO

```
≪ + LIST→ DROP 5
PICK OVER * 5 PICK
4 PICK * - EXCO 5
ROLL 5 PICK * ROT 7
PICK * - EXCO ROT 6
ROLL * 5 ROLL 5
ROLL * - EXCO
≫
```

dot

Version: HP48 und HP28S
Checksum(HP28S): 182732
Checksum(HP48): 5BEB
Bytes: 73.5

```
≪ 0 1 3 PICK SIZE
  FOR e OVER e GET
4 PICK e GET * +
  NEXT ROT ROT
DROP2
≫
```

2: 2. Vektor 1: 1. Vektor	1: Skalarprodukt der Vektoren

Erläuterung: Berechnet das Skalarprodukt aus zwei Vektoren.

Beispiel: (4) (0)
(2)*(2)=7
(3) (1)
Eingabe: 2: { 4 2 3 }
1: { 0 2 1 } dot ⟹ 7

benötigte UP: keine

INPOL

Version: HP48 und HP28S
Checksum(HP28S): 1414145
Checksum(HP48): DFBA
Bytes: 155.0

```
≪ DUP SIZE SWAP
MTRN LIST→ ROLL 1
OVER SIZE
  FOR i DUP i GET 4
PICK 1 - 0
    FOR j DUP j ^ 5
ROLLD -1
    STEP DROP
  NEXT DROP LIST→ 1
→LIST →ARRY OVER SQ
2 + ROLLD DUP 2
→LIST →ARRY / ARRY→
1 GET →LIST
≫
```

1: Liste mit zu interpolierenden Punkten	1: interpoliertes Polynom

Erläuterung: Interpoliert eine beliebige Anzahl von Punkten und gibt ein Polynom (in Listen-Schreibweise) aus, das diese Punkte als Stützwerte hat. Zugelassen sind auch komplexe Funktionswerte an komplexen Stützstellen, aber keine Variablen.

Beispiel: { { 1 2 }{ 3 6 }{ 4 2 } } INPOL ⟹
{ -2 10 -6 }

benötigte UP: MTRN

BES

Version: HP48 und HP28S
Checksum(HP28S): 92227
Checksum(HP48): 86C
Bytes: 66.0

```
≪ 2 / 'V' DUP ROT
OVER ^ SWAP FACT /
SQ SWAP 0 6 REI
≫
```

1: REAL-Zahl	1: Wert der Bessel-Funktion

Erläuterung: Berechnet den Funktionswert der Bessel-Funktion. Dieses Programm ist allerdings nur als Hilfsprogramm konzipiert.
Beispiel: 1.2 BES $\Longrightarrow$ 1.3937255841
benötigte UP: REI

DRW

Version: HP28S
Checksum(HP28S): 1222044
Checksum(HP48): entfällt
Bytes: 153.0

```
≪ 'PPAR' PURGE
# 2484Bh SYSEVAL
DROP2 ROT ROT DROP2
SWAP CLΣ CLLCD FA
LIST→ DROP DUP2 -
NEG 137 / ROT ROT
  FOR e e DUP 4 PICK
STO RCEQ →NUM 2
→ARRY Σ+ DUP
  STEP SCLΣ DRWΣ
DROP PURGE DGTIZ
≫
```

1: Liste mit minimalem und maximalem Wert auf der X-Achse	1: nichts

Erläuterung: Zeichnet eine beliebige Funktion mit automatischer Skalierung. Diese Option beinhalten die Rechner des HP48 auch ohne dieses Programm.
Beispiel: RAD ENTER
'SIN(Y)' STEQ ENTER
{ -2 3 } DRW
(Zeichnet die Funktion SIN(Y) von -2 bis 3 unter optimaler Ausnutzung des Displays).
benötigte UP: FA

BODE

Version: HP48 und HP28S
Checksum(HP28S): 31019
Checksum(HP48): 9366
Bytes: 44.0

```
≪ ROT ABS LOG -4 5
* * STEQ LG
≫
```

3: Start- und Endwert in einer Liste 1: unabh. Variable 1: zu zeichnende Funktion	1: nichts

Erläuterung: Zeichnet den Betragsfrequenzgang einer Funktion logarithmisch.

Beispiel: 3: '(0,1)*W+3'
2: { .1 128 }
1: 'W' BODE ENTER zeichnet nach einiger Rechenzeit den Betragsfrequenzgang logarithmisch

benötigte UP: LG

BOOL

Version: HP48 und HP28S
Checksum(HP28S): 1242548
Checksum(HP48): 1FA3
Bytes: 154.0

3: erster Ausdruck 2: zweiter Ausdruck 1: Liste mit den Variablen	1: Vergleichsergebnis

Erläuterung: Vergleicht zwei boolesche Ausdrücke miteinander. Sind sie gleich, liefert das Programm eine Eins, ansonsten eine Null.

Beispiel: 3: 'X'
2: 'NOT (NOT(X))'
1: { X } BOOL ⟹ 1

benötigte UP: keine

```
≪ 8 SF BIN 0 OVER
SIZE SQ 1 -
  FOR i 1 R→B i R→B
3 PICK SIZE 1
    FOR j DUP2 AND
B→R 4 PICK j GET
STO SR -1
    STEP DROP2 OVER
→NUM 4 PICK →NUM
    IF ≠
    THEN 8 CF
    END
  NEXT PURGE DROP2
8 FS?
≫
```

LG

Version: HP28S
Checksum(HP28S): 1119537
Checksum(HP48): entfällt
Bytes: 162.0

2: Start- und Endwert in einer Liste 1: Name der unabh. Variable	1: nichts

Erläuterung: Ermöglicht das logarithmische Zeichnen einer Funktion. Das Programm ermöglicht nach dem Plotten der Funktion ein Abfragen der einzelnen Punkte (wie beim Befehl DGTIZ), die gelieferten X-Werte stimmen allerdings gar nicht, da sie umgerechnet sind.

Beispiel: Es soll die Funktion 'LOG(ABS(i*W+3))*(-20)' von .1 bis 128 logarithmisch gezeichnet werden. (Bode-Diagramm des Betrags-Frequenzganges).
'LOG(ABS(i*W+3))*(-20)' STEQ
ENTER
1: { .1 128 }
2: 'W' LG ENTER berechnet erst alle Punkte, speichert sie in ΣDAT ab und zeichnet anschließend die Funktion.

benötigte UP: keine

```
≪ FA CLΣ SWAP LIST→
PICK LD OVER LD -
ABS CEIL 137 0 OVER
CLLCD
  FOR x 3 PICK OVER
/ x * 2 x 4 PICK / 5
PICK * ^ 6 PICK * 7
PICK STO RCEQ →NUM 2
→ARRY Σ+
  NEXT 5 DROPN SCLΣ
DRWΣ DGTIZ
≫
```

LG

Version: HP48S
Checksum(HP28S): entfällt
Checksum(HP48): 6CBA
Bytes: 159.5

2: Start- und Endwert in einer Liste 1: Name der unabh. Variable	1: nichts

Erläuterung: Ermöglicht das logarithmische Zeichnen einer Funktion. Das Programm ermöglicht nach dem Plotten der Funktion ein Abfragen der einzelnen Punkte (wie beim Befehl DGTIZ), die gelieferten X-Werte stimmen allerdings gar nicht, da sie umgerechnet sind.

Beispiel: Es soll die Funktion 'LOG(ABS(i*W+3))*(-20)' von .1 bis 128 logarithmisch gezeichnet werden. (Bode-Diagramm des Betrags-Frequenzganges).
'LOG(ABS(i*W+3))*(-20)' STEQ
ENTER
1: { .1 128 }
2: 'W' LG ENTER berechnet erst alle Punkte, speichert sie in ΣDAT ab und zeichnet anschließend die Funktion.

benötigte UP: keine

```
≪ FA CLΣ SWAP LIST→
PICK LD OVER LD -
ABS CEIL 137 0 OVER
ERASE
  FOR x 3 PICK OVER
/ x * 2 x 4 PICK /
5 PICK * ^ 6 PICK *
7 PICK STO RCEQ
→NUM 2 →ARRY Σ+
  NEXT 5 DROPN
SCATRPLOT GRAPH
≫
```

LG

Version: HP48G
Checksum(HP28S): entfällt
Checksum(HP48): 6CBA
Bytes: 159.5

2: Start- und Endwert in einer Liste 1: Name der unabh. Variable	1: nichts

Erläuterung: Ermöglicht das logarithmische Zeichnen einer Funktion. Das Programm ermöglicht nach dem Plotten der Funktion ein Abfragen der einzelnen Punkte (wie beim Befehl DGTIZ), die gelieferten X-Werte stimmen allerdings gar nicht, da sie umgerechnet sind.

Beispiel: Es soll die Funktion 'LOG(ABS(i*W+3))*(-20)' von .1 bis 128 logarithmisch gezeichnet werden. (Bode-Diagramm des Betrags-Frequenzganges).
'LOG(ABS(i*W+3))*(-20)' STEQ
ENTER
1: { .1 128 }
2: 'W' LG ENTER berechnet erst alle Punkte, speichert sie in ΣDAT ab und zeichnet anschließend die Funktion.

benötigte UP: keine

```
≪ FA CLΣ SWAP LIST→
PICK LD OVER LD -
ABS CEIL 137 0 OVER
ERASE
  FOR x 3 PICK OVER
/ x * 2 x 4 PICK /
5 PICK * ^ 6 PICK *
7 PICK STO RCEQ
→NUM 2 →ARRY Σ+
  NEXT 5 DROPN
SCATRPLOT PICTURE
≫
```

REI

Version: HP48 und HP28S
Checksum(HP28S): 204009
Checksum(HP48): 92B3
Bytes: 72.0

```
≪ FA 0 ROT ROT
  FOR i 3 PICK i 4
PICK STO EVAL +
  NEXT SWAP PURGE
SWAP DROP
≫
```

4: Funktion 3: unabh. Variable 2: Anfangswert der Reihe 1: Endwert der Reihe	1: Wert der Reihe

Erläuterung: Berechnet den Wert einer Reihe.
Beispiel: 4: 'X-.5'
3: 'X'
2: 2
1: 8 REI ⟹ 31.5
benötigte UP: FA

HORN

Version: HP48 und HP28S
Checksum(HP28S): 294448
Checksum(HP48): 986B
Bytes: 78.5

```
≪ SWAP 1 2 3 PICK
SIZE
  START GETI 4 PICK
* 3 ROLLD DUP2 GET
4 ROLL + PUTI 1 -
  NEXT ROT DROP2
≫
```

2: Polynom in Listen-Schreibweise 1: Argumenten-Stelle	1: Polynom, mit dem das Horner-Schema durchgeführt wurde

Erläuterung: Führt das Horner-Schema mit einem Polynom (in Listen-Schreibweise) durch.
Beispiel: 2: { 1 2 3 4 }
1: -1 HORN ⟹
{ 1 1 2 2 }
benötigte UP: keine

DEFI

Version: HP28S
Checksum(HP28S): 2532438
Checksum(HP48): entfällt
Bytes: 259.5

1: Matrix in Listen-Schreibweise	1: Kurzinfo über die Definitheit der Matrix

Erläuterung: Gibt über ein Kurzinfo Auskunft über die Definitheit einer Matrix (in Listen-Schreibweise).
Achtung! Nur für numerische Matrizen geeignet.

Beispiel: { { 2 3 8 }{ 2 5 7 }{ 3 2 1 } } DEFI
$\Longrightarrow$ 'INDEF'

Kurzinfos: 0 'posi' → positiv semidefinit
0 'neg' → negativ semidefinit
1 → positiv definit
-1 → negativ definit
'INDEF' → indefinit

benötigte UP: DE, MO, SHO, carp, DEF, MAL, KO

```
≪ 1 2
  FOR i -1 MAL DUP {
} OVER SIZE 1
    FOR e OVER MO
DET + SWAP e DUP SHO
SWAP -1
    STEP + DE DUP 0
POS SWAP →STR "-"
POS OR
    IF NOT
    THEN DROP -1 i ^
ABORT
    END
  NEXT DUP carp KO 1
  IF GET
  THEN DROP INDEF
  ELSE 0 { } ROT DEF
→STR "-" POS NOT
posi neg IFTE
  END
≫
```

DEFI

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): EF9
Bytes: 259.5

1: Matrix in Listen-Schreibweise	1: Kurzinfo über die Definitheit der Matrix

Erläuterung: Gibt über ein Kurzinfo Auskunft über die Definitheit einer Matrix (in Listen-Schreibweise).
Achtung! Nur für numerische Matrizen geeignet.
Falls das Programm mit 'HALT' stoppt, so ist die Berechnung abgeschlossen und 'KILL' einzutippen.

Beispiel: { { 2 3 8 }{ 2 5 7 }{ 3 2 1 } } DEFI ⟹ 'INDEF'

Kurzinfos: 0 'posi' → positiv semidefinit
0 'neg' → negativ semidefinit
1 → positiv definit
-1 → negativ definit
'INDEF' → indefinit

benötigte UP: DE, MO, SHO, carp, DEF, MAL, KO

```
≪ 1 2
  FOR i -1 MAL DUP
{ } OVER SIZE 1
    FOR e OVER MO
DET + SWAP e DUP
SHO SWAP -1
    STEP + DE DUP 0
POS SWAP →STR "-"
POS OR
    IF NOT
    THEN DROP -1 i
^HALT
    END
  NEXT DUP carp KO
1
  IF GET
  THEN DROP INDEF
  ELSE 0 { } ROT
DEF →STR "-" POS
NOT posi neg IFTE
  END
≫
```

DEF

Version: HP48 und HP28S
Checksum(HP28S): 363761
Checksum(HP48): 61A0
Bytes: 116.0

1: —	1: —

Erläuterung: Reines Hilfsprogramm für DEFI.
Beispiel: entfällt
benötigte UP: MO, SHO, DE

```
≪ DUP → u
  ≪ MO DET + u SIZE
1
    IF ≠
    THEN 1 u SIZE
      FOR l u l DUP
SHO DEF
      NEXT
    END
  ≫ DE
≫
```

div

Version: HP48 und HP28S
Checksum(HP28S): 4284915
Checksum(HP48): DFF2
Bytes: 288.0 bzw. 290.5

2: Zahl als String	
1: Zahl als String	1: dividierte Zahl

Erläuterung: Dividiert zwei REAL-Zahlen mit beliebiger Genauigkeit. Die Zahlen werden als Strings eingegeben.
Achtung! Es ist nur erlaubt, die Zahl auszuschreiben (also nicht 1E+3). Es muß immer der Dezimalpunkt geschrieben werden (also nicht "7", sondern "7." etc.).
Man muß die Anzahl der zu berechnenden Stellen im Programm immer selbst ändern. Die Standardeinstellung beträgt 30 Stellen Genauigkeit. Im Programm ist dazu die Zahl '30' zu ändern.

Beispiel: 2: "1.2345"
1: "1.634965843" div ⟹
"0.755061645651761790353194552 94"

benötigte UP: mu, mul, di

```
≪ DUP2 mu 5 DROPN
SWAP DROP ROT ROT 8
9
  FOR e e DUP 2 -
SWAP
    IF FS?
    THEN SF 2 OVER
SIZE SUB
    ELSE CF
    END SWAP
  NEXT DUP2 "" ROT
ROT 1 30
  FOR i OVER STR→
OVER STR→ / IP →STR
4 ROLL OVER + 4
ROLLD "." + OVER
mul ROT SWAP di
"10." mul SWAP
  NEXT DROP2 ROT
STR→ ROT STR→ / IP
→STR SIZE OVER SIZE
- NEG ROT EVAL 6
FS? 7 FS?
  IF ≠
  THEN "-" SWAP +
  END
≫
```

mul

Version: HP48 und HP28S
Checksum(HP28S): 285074
Checksum(HP48): CD00
Bytes: 93.0

2: Zahl als String 1: Zahl als String	1: multiplizierte Zahl

Erläuterung: Multipliziert Zahlen beliebig genau. Eingabeformat wie bei 'div'!
Beispiel: 2: "1.235"
1: "1.0122" mul ⟹ "1.250067"
benötigte UP: MUL, mu

```
≪ mu MUL ROT DROP
ROT EVAL SWAP 2 *
ROT EVAL SWAP EVAL
8 FS? 9 FS?
  IF ≠
  THEN "-" SWAP +
  END
≫
```

ad

Version: HP48 und HP28S
Checksum(HP28S): 1382408
Checksum(HP48): 1D5E
Bytes: 175.0

2: Zahl als String 1: Zahl als String	1: addierte Zahl

Erläuterung: Addiert zwei REAL-Zahlen mit beliebiger Genauigkeit. Die Zahlen werden als Strings eingegeben.
Achtung! Eingabeformat wie bei „div"!

Beispiel: 2: "1.2156412254"
1: "3.1021045469" ad ⟹
"4.3177457723"
benötigte UP: MUL, mu, AD

```
≪ mu 4 ROLL EVAL 8
FS? 9 FS?
  IF ≠
  THEN { -1 } MUL
  END AD ROT EVAL
SWAP ROT EVAL SWAP
EVAL 9 FS? 8 FS?
DUP2 AND OVER 4
PICK NOT AND 6 PICK
AND OR ROT ROT NOT
AND 4 ROLL NOT AND
  IF OR
  THEN "-" SWAP +
  END
≫
```

di

Version: HP48 und HP28S
Checksum(HP28S): 148671
Checksum(HP48): 44F
Bytes: 78.0

2: Zahl als String 1: Zahl als String	1: subtrahierte Zahl

Erläuterung: Subtrahiert zwei REAL-Zahlen in beliebiger Genauigkeit. Die Zahlen werden als Strings eingegeben.
Achtung! Eingabeformat wie bei „div“!

Beispiel: 2: "3.14159265358979323"
1: "2.3125647243113" di ⟹
"0.82902792927849323"
benötigte UP: ad

```
≪ DUP NUM 9 5 *
  IF ==
  THEN 2 9 ALOG SUB
  ELSE "-" SWAP +
  END ad
≫
```

mu

Version: HP48 und HP28S
Checksum(HP28S): 23698144
Checksum(HP48): B286
Bytes: 698.5 bzw. 701.0

1: —	1: —

Erläuterung: Reines Hilfsprogramm für mul, div, ad und di.
Beispiel: entfällt
benötigte UP: AU, KO

```
≪ 8 9
  FOR e e OVER 1 1
SUB "-"
    IF ==
    THEN SF 2 OVER
SIZE SUB
    ELSE CF
    END DUP SIZE
SWAP DUP "." POS
DUP2 1 SWAP 1 - SUB
ROT 3 PICK 1 + 5
PICK SUB + ROT ROT
- ROT
  NEXT DUP2 MAX ROT
ROT - ROT OVER ABS
ST ROT 0
  IF >
  THEN 4 ROLL SWAP
+
  ELSE + ROT
```

```
  END 1 2
  START DUP SIZE 1
    FOR e DUP e 4 -
e SUB STR→ SWAP -5
    STEP SIZE 5 /
CEIL →LIST KO SWAP
  NEXT
  ≪ DUP2 DI 1 GET 0
    IF <
    THEN SWAP 1
    ELSE 0
    END 7 ROLLD
  ≫
  ≪ "" 0 0 4 ROLL +
+ DUP SIZE 2
    FOR j DUP j GET
DUP 5 ALOG MOD DUP2
- ROT DROP 5 ALOG /
SWAP →STR DUP SIZE
5 - NEG ST SWAP + 4
ROLL + ROT j 1 -
DUP2 GET 5 ROLL +
PUT -1
    STEP DROP
  ≫≪ OVER SIZE - NEG
DUP2 1 SWAP SUB "."
+ ROT ROT 1 + OVER
SIZE SUB +
  ≫
  ≪
    WHILE DUP DUP
SIZE DUP SUB "0" ==
    REPEAT 1 OVER
SIZE 1 - SUB
    END
    WHILE DUP 1 1
SUB "0" == OVER 2 2
SUB "." ≠ AND
    REPEAT 2 OVER
SIZE SUB
    END
  ≫ 7 TA SWAP ROT
≫
```

EXY

Version: HP48 und HP28S
Checksum(HP28S): 68860
Checksum(HP48): 543
Bytes: 45.0

```
≪ 2 →LIST RCLΣ DUP
TRN SWAP * SWAP GET
NΣ /
≫
```

2: Kennzahl 1 1: Kennzahl 2	1: Erwartungswert

Erläuterung: Ermöglicht die Berechnung der Erwartungswerte E(x^2), E(x*y), E(z^2) etc. Diese Funktion hat der HP48 mittlerweile implementiert!

Beispiel: [2.3 1.2] ENTER
[1.7 2.8] ENTER
[1.1 2.3] ENTER , nun in das STAT-Menu gehen und dreimal die Σ+ -Taste drücken.
Es ist E(x^2)=3.13,
E(x*y)=3.35 und
E(y^2)=4.856666666666.
für 1 1 EXY bekommt man E(x^2)
für 1 2 EXY bekommt man E(x*y)
für 2 2 EXY bekommt man E(y^2)
etc.

benötigte UP: keine

EF

Version: HP28S
Checksum(HP28S): 46437
Checksum(HP48): entfällt
Bytes: 42.0

```
≪ 8 ENG DUP 1 DISP
STD WA DROP CLMF
≫
```

1: REAL-Zahl	1: REAL-Zahl

Erläuterung: Schreibt eine REAL-Zahl im technischen Format.

Beispiel: 1E8 EF $\Longrightarrow$ schreibt 100.000000E6 in die erste Bildschirmzeile und wartet auf Tastendruck.
Sinn des Programms ist ein schneller Überblick über die Größenordnung einer Zahl, ohne in das ENG-Format wechseln zu müssen.

benötigte UP: WA

EF

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): 68F4
Bytes: 39.5

```
≪ 8 ENG DUP 1 DISP
STD WA DROP
≫
```

1: REAL-Zahl	1: REAL-Zahl

Erläuterung: Schreibt eine REAL-Zahl im technischen Format.

Beispiel: 1E8 EF $\Longrightarrow$ schreibt 100.000000E6 in die erste Bildschirmzeile und wartet auf Tastendruck.
Sinn des Programms ist ein schneller Überblick über die Größenordnung einer Zahl, ohne in das ENG-Format wechseln zu müssen.

benötigte UP: WA

8.2.4 Matrizenrechnen

MMUL

Version: HP48 und HP28S
Checksum(HP28S): 1336210
Checksum(HP48): EB64
Bytes: 193.0

2: Matrix in Listen-Schreibweise 1: Matrix in Listen-Schreibweise	1: Produkt der beiden Matrizen

Erläuterung: Führt die Matrizenmultiplikation zweier Matrizen in Listen-Schreibweise aus.

Beispiel: 2: { { 3 3 3 }{ 4 1 2 }{ 0 2 1 } }
1: { { 0 1 0 }{ 0 1 2 }{ 3 4 5 } } MMUL
$\Longrightarrow$ {{ 9 18 21 }{ 6 13 12 }{ 3 6 9 }}

benötigte UP: MTRN, EXCO

```
≪ DUP MTRN SWAP
SIZE → m w t
  ≪ 1 m SIZE
    FOR p { } 1 w
SIZE
      FOR q m p GET
w q GET 0 1 t
        FOR r OVER
r GET 4 PICK r GET
* +
        NEXT ROT
ROT DROP2 EXCO +
      NEXT
    NEXT m SIZE
→LIST
  ≫
≫
```

MAD

Version: HP48 und HP28S
Checksum(HP28S): 164760
Checksum(HP48): 7DFB
Bytes: 69.0

2: Matrix in Listen-Schreibweise 1: Matrix in Listen-Schreibweise	1: Summe der Matrizen

Erläuterung: Berechnet aus zwei Matrizen (in Listen-Schreibweise) die Summe.

Beispiel: 2: { { 2 3 } { 5 6 } }
1: { { 1 0 } { 0 1 } } MAD $\Longrightarrow$
{ { 3 3 }{ 5 7 } }

benötigte UP: AD

```
≪ 1 OVER SIZE
  FOR i i DUP2 GET
4 PICK i GET AD PUT
  NEXT SWAP DROP
≫
```

MDI

Version: HP48 und HP28S
Checksum(HP28S): 6793
Checksum(HP48): EBF4
Bytes: 33.0

```
≪ -1 MAL MAD
≫
```

2: Matrix in Listen-Schreibweise 1: Matrix in Listen-Schreibweise	1: Differenz der Matrizen

Erläuterung: Berechnet aus zwei Matrizen (in Listen-Schreibweise) die Differenz.
Beispiel: 2: { { 2 3 } { 5 6 } }
1: { { 1 0 } { 0 1 } } MDI ⟹
{ { 1 3 }{ 5 5 } }
benötigte UP: MAL, MAD

MAL

Version: HP48 und HP28S
Checksum(HP28S): 294374
Checksum(HP48): D0F8
Bytes: 71.5

```
≪ 1 →LIST SWAP 1
DUP 3 PICK SIZE
  START GETI 4 PICK
MUL 4 ROLLD
  NEXT ROT DROP2
SIZE →LIST
≫
```

2: Matrix in Listen-Schreibweise 1: skalarer Faktor	1: Produkt von Faktor und Matrix

Erläuterung: Multipliziert einen skalaren Faktor an eine Matrix in Listen-Schreibweise.
Beispiel: 2: { { 2 3 } }
1: 3 MAL ⟹ 1: { { 6 9 } }
benötigte UP: MUL

MINV

Version: HP48 und HP28S
Checksum(HP28S): 66981
Checksum(HP48): 2978
Bytes: 69.0

```
≪ FADD 2 GET ca KO
1 GET INV MAL 'ca'
PURGE
≫
```

1: Matrix in Listen-Schreibweise	1: Inverse dieser Matrix

Erläuterung: Berechnet die Inverse einer Matrix (in Listen-Schreibweise).

Beispiel: 1: {{ 1 2 3}{0 5 6}{7 8 9}} MINV

$\Longrightarrow$

{{.125 −.25 .125}
{−1.75 .5 .25}
{1.458333 −.25 −.208333}}

benötigte UP: FADD, KO

MINVV

Version: HP48 und HP28S
Checksum(HP28S): 31231
Checksum(HP48): 2A1D
Bytes: 48.5

```
≪ DUP MADJ SWAP
MDET INV MAL
≫
```

1: Matrix in Listen-Schreibweise	1: Inverse dieser Matrix

Erläuterung: Berechnet die Inverse einer Matrix (in Listen-Schreibweise).
Dieses Programm ist bei Matrizen, die Variablen enthalten, deutlich schneller als das Programm MINV, ansonsten aber langsamer.

Beispiel: 1: {{1 2 3}{0 5 6}{7 8 9}} MINVV

$\Longrightarrow$

{{.125 −.25 .125}
{−1.75 .5 .25}
{1.458333 −.25 −.208333}}

benötigte UP: MADJ, MDET, MAL

MDET

Version: HP48 und HP28S
Checksum(HP28S): 1423031
Checksum(HP48): C240
Bytes: 228.0

1: Matrix in Listen-Schreibweise	1: Determinante dieser Matrix

Erläuterung: Berechnet die Determinante aus einer Matrix (in Listen-Schreibweise).
Beispiel: { { 3 1 2 }{ 4 2 9 }{ 0 1 1 } } MDET
⟹ -17
benötigte UP: EXCO, SHO

```
≪ DUP SIZE → m s
  ≪ 0 s 1
    IF ≠
    THEN 1 s
      FOR i 1 m i
GET s GET -1 s i +
^ * DUP 0
        IF SAME
        THEN 0 *
        ELSE * EXCO
m i s SHO MDET
        END * +
      NEXT
    ELSE m 1 GET 1
GET +
    END EXCO
  ≫
≫
```

MADJ

Version: HP48 und HP28S
Checksum(HP28S): 444845
Checksum(HP48): 80B2
Bytes: 133.5

1: Matrix in Listen-Schreibweise	1: Adjunkte dieser Matrix

Erläuterung: Berechnet aus einer Matrix (in Listen-Schreibweise) die Adjunkte dieser Matrix.
Beispiel: { { 1 5 3 }{ 2 1 5 }{ 2 2 1 } } MADJ
⟹
{ { -9 1 22 }{ 8 -5 1 }{ 2 8 -9 } }
benötigte UP: SHO, MDET

```
≪ DUP SIZE → m s
  ≪ 1 s
    FOR i 1 s
      FOR j -1 i j
+ ^ m j i SHO MDET
*
      NEXT s →LIST
    NEXT s →LIST
  ≫
≫
```

MTRN

Version: HP48 und HP28S
Checksum(HP28S): 503718
Checksum(HP48): FD0A
Bytes: 125.5

```
≪ FA DUP 1 GET SIZE
OVER SIZE → m s z
  ≪ 1 s
    FOR i 1 z
      FOR j m j GET
i GET
      NEXT z →LIST
    NEXT s →LIST
  ≫
≫
```

1: Matrix in Listen-Schreibweise	1: transponierte Matrix

Erläuterung: Transponiert eine Matrix (in Listen-Schreibweise).
Beispiel: { { A B }{ 0 0 } } MTRN ⟹
{ { A 0 }{ B 0 } }
benötigte UP: FA

MIDN

Version: HP48 und HP28S
Checksum(HP28S): 3756
Checksum(HP48): 7329
Bytes: 26.5

```
≪ IDN MI
≫
```

1: Zahl	1: Einheitsmatrix (in Listen-Schreibweise) der Dimension Zahl

Erläuterung: Erzeugt eine Einheitsmatrix in der Listen-Schreibweise.
Beispiel: 2 MIDN ⟹ { { 1 0 }{ 0 1 } }
benötigte UP: MI

KOND

Version: HP48S und HP28S
Checksum(HP28S): 32437
Checksum(HP48): 3227
Bytes: 39.0

```
≪ MO DUP CNRM SWAP
INV CNRM *
≫
```

1: Matrix in Listen-Schreibweise	1: Konditionszahl

Erläuterung: Bestimmt den Wert der Konditionszahl einer Matrix (in Listen-Schreibweise). Achtung! Funktioniert nur bei reinen Zahlenmatrizen!
Beispiel: { { 2 3 }{ 3 2 } } KOND ⟹ 5
benötigte UP: MO

KOND

Version: HP48G
Checksum(HP28S): entfällt
Checksum(HP48): 90FB
Bytes: 29.5

```
≪ MO COND
≫
```

1: Matrix in Listen-Schreibweise	1: Konditionszahl

Erläuterung: Bestimmt den Wert der Konditionszahl einer Matrix (in Listen-Schreibweise). Achtung! Funktioniert nur bei reinen Zahlenmatrizen!
Beispiel: { { 2 3 }{ 3 2 } } KOND ⟹ 5
benötigte UP: MO

GJ

Version: HP48 und HP28S
Checksum(HP28S): 1772061
Checksum(HP48): 5E4A
Bytes: 198.5

3: Matrix in Listen-Schreibweise 2: Zeile des Pivotelementes 1: Spalte des Pivotelementes	1: Matrix nach Durchführung eines Schrittes der GJ-Elimination

Erläuterung: Berechnet einen Schritt der Gauß-Jordan-Elimination. Das Programm notiert automatisch die Pivotelemente in der Variablen 'Y' fortlaufend mit (zur späteren Kontrolle, sowie zur Benutzung durch ein anderes Programm [KBM]).

Beispiel: 3: { { 1 2 4 }{ 3 4 2 }{ 4 2 1 } }
2: 1
1: 2 GJ ⟹
{ { 1 2 4 }{ 1 0 -6 }{ 3 0 -3 } }
Hier wurde das Pivot-Element (1,2) gewählt, also die 2 in der ersten Zeile.

benötigte UP: MUL, AD

```
≪ 3 DUPN DROP GET 4
ROLL 1 OVER SIZE
  FOR i i 5 PICK
    IF ≠
    THEN i 3 DUPN
GET 6 PICK DUP2 GET
4 PICK ROT GET /
NEG 1 →LIST ROT MUL
AD PUT
    END
  NEXT SWAP DROP Y
DUP TYPE 5
  IF ≠
  THEN DROP { }
  END 4 ROLL 4 ROLL
R→C + 'Y' STO
≫
```

ZMUL

Version: HP48 und HP28S
Checksum(HP28S): 26070
Checksum(HP48): 6511
Bytes: 37.5

```
≪ 3 DUPN DROP GET
MUL PUT
≫
```

3: Matrix in Listen-Schreibweise 2: Zeile, an die der Faktor heranmultipliziert werden soll 1: skalarer Faktor in geschweiften Klammern geschrieben	1: modifizierte Matrix

Erläuterung: Multipliziert an eine Zeile einer Matrix einen skalaren Faktor.
Beispiel: 3: { { 1 2 }{ 3 4 } }
2: 1
1: { 3 } ZMUL $\Longrightarrow$ {{ 3 6 }{ 3 4 }}
benötigte UP: MUL

EI

Version: HP48 und HP28S
Checksum(HP28S): 15824
Checksum(HP48): 71B6
Bytes: 35.5

```
≪ DUP SIZE MIDN ANH
≫
```

1: Matrix in Listen-Schreibweise	1: Matrix mit angehängter Einheitsmatrix

Erläuterung: Hängt an eine Matrix (in Listen-Schreibweise) eine Einheitsmatrix an.

Beispiel: { { 2 3 }{ 2 3 } } EI $\Longrightarrow$
{ { 2 3 1 0 }{ 2 3 0 1 } }
benötigte UP: MIDN, ANH

RANG

Version: HP48 und HP28S
Checksum(HP28S): 2261353
Checksum(HP48): F0
Bytes: 241.5

1: Matrix in Listen-Schreibweise	1: Rang der Matrix

Erläuterung: Bestimmt den Rang einer Matrix (in Listen-Schreibweise).
Achtung! Funktioniert nur sinnvoll, wenn es sich um eine rein zahlengefüllte Matrix handelt.

Beispiel: { { 1 2 3 }{ 4 5 6 }{ 7 8 9 } } RANG
$\Longrightarrow$
2: { { 1 0 -1 }{ 0 -3 -6 }{ 0 0 0 } }
1: 2

benötigte UP: LI, MA, MTRN, GJ, PIV, DE, EVV

```
≪ 'Y' PURGE DUP
SIZE OVER 1 GET
SIZE DUP2 SWAP MA
SWAP LI ROT LI 4
ROLL DUP MTRN
  DO PIV SWAP GJ
DUP 1 Y SIZE
    FOR i Y i GET
RE 5 PICK PUT
    NEXT MTRN 1 Y
SIZE
    FOR i Y i GET
IM 4 PICK PUT
    NEXT
  UNTIL DE EVV DUP
6 PICK SAME
  END DROP 4 ROLLD
3 DROPN Y SIZE
≫
```

SPUR

Version: HP48 und HP28S
Checksum(HP28S): 72425
Checksum(HP48): D464
Bytes: 52.5

1: Matrix in Listen-Schreibweise	1: Spur dieser Matrix

Erläuterung: Berechnet die Spur einer Matrix (in Listen-Schreibweise).

Beispiel: { { 1 2 3 }{ 4 5 6 }{ 7 8 9 } } SPUR
$\Longrightarrow$ 15

benötigte UP: keine

```
≪ LIST→ 0 SWAP 1
  FOR i SWAP i GET
+ -1
  STEP
≫
```

PIV

Version: HP48 und HP28S
Checksum(HP28S): 2669443
Checksum(HP48): 716A
Bytes: 223.5

	2: Zeile des Pivot-Elementes
1: Matrix in Listen-Schreibweise	1: Spalte des Pivot-Elementes

Erläuterung: Bestimmt ein mögliches Pivotelement aus einer Matrix (in Listen-Schreibweise).

Beispiel: { { 0 2 0 }{ 0 0 0 }{ 0 0 0 } } PIV

$\Longrightarrow$

2: 1
1: 2

benötigte UP: keine

```
≪ 1 OVER 1 GET SIZE
  FOR e 1 OVER SIZE
    FOR f DUP f GET
e GET 8 SF DUP TYPE
2
      IF <
      THEN ABS -4
ALOG
        IF <
        THEN 8 CF
        END
      ELSE DROP
      END 8
      IF FS?
      THEN f e ROT
DUP 1 GET SIZE 'e'
STO SIZE 'f' STO
      END
    NEXT
  NEXT 8
  IF FC?
  THEN DROP 0 DUP
  END
≫
```

KBM

Version: HP48 und HP28S
Checksum(HP28S): 9022129
Checksum(HP48): 2961
Bytes: 449.5

1: Matrix in Listen-Schreibweise	1: Kern-Basis-Matrix

Erläuterung: Bestimmt die Kern-Basis-Matrix einer Matrix (in Listen-Schreibweise).

Beispiel: Hierbei gibt es zwei Wege: Erstens gibt man die bekannten Daten direkt ein, oder zweitens macht man die vorhergehenden Schritte ebenfalls mit dem Rechner (Normalfall)!
Beispiel zur ersten Methode:
{{ 1 0 -1 }{ 0 -3 -6 }{ 0 0 0 }} im Stack und { (1,1) (2,2) } in der Variablen 'Y' gespeichert (Pivotelemente)
KBM $\Longrightarrow$ {{ 1 }{ -2 }{ 1 }}
Beispiel zur zweiten Methode:
{{ 1 2 3 }{ 4 5 6 }{ 7 8 9 }} 1 1 GJ $\Longrightarrow$
{{ 1 2 3 }{ 0 -3 -6 }{ 0 -6 -12 }}
dann 2 2 GJ $\Longrightarrow$
{{ 1 0 -1 }{ 0 -3 -6 }{ 0 0 0 }}
dann KBM $\Longrightarrow$ {{ 1 }{ -2 }{ 1 }}

benötigte UP: ZMUL, LI, MA

```
≪ DUP 1 GET SIZE →
s
  ≪ Y
    IF SIZE
    THEN 1 Y SIZE
      FOR i Y i GET
DUP2 RE GET OVER IM
GET INV 1 →LIST
SWAP RE SWAP ZMUL
      NEXT
    END { } 1 s
    FOR e Y →STR
"," e →STR ")" + +
POS
      IF NOT
      THEN e +
      END
    NEXT DUP 'A'
STO
    IF SIZE
    THEN 1 s
      FOR k Y DUP
→STR "," k →STR ")"
+ + POS 6 /
        IF DUP
        THEN GET RE
{ } 1 A SIZE
          FOR l 2 k
+ PICK 3 PICK GET A
l GET GET NEG +
          NEXT SWAP
DROP
        ELSE DROP2
A SIZE LI A k POS 1
PUT
        END
      NEXT s →LIST
    ELSE s 1 MA
    END SWAP DROP
  ≫
≫
```

EV

Version: HP48 und HP28S
Checksum(HP28S): 85390
Checksum(HP48): CFB9
Bytes: 61.5

```
≪ OVER SIZE IDN *
SWAP MO SWAP - MI
RANG DROP KBM
≫
```

2: Matrix in Listen-Schreibweise 1: Eigenwert der Matrix	1: Eigenvektor der Matrix

Erläuterung: Errechnet aus einer Matrix (in Listen-Schreibweise) und einem Eigenwert einen Eigenvektor.
Achtung! Läuft nur bei Zahlenmatrizen!
Beispiel: 2: { { 1 2 3 }{ 4 5 6 }{ 7 8 9 } }
1: 0 EV ⟹ { { 1 }{ -2 }{ 1 } }
benötigte UP: MO, MI, RANG, KBM

FADD

Version: HP48 und HP28S
Checksum(HP28S): 1930799
Checksum(HP48): 8844
Bytes: 237.0

```
≪ DUP SIZE DUP MIDN
{ 1 } OVER 1 →LIST
1 5 PICK
  FOR q DUP q GET 6
PICK MMUL 4 PICK
OVER SPUR q / NEG
DE 5 ROLL OVER + 5
ROLLD MAL MAD 1
→LIST +
  NEXT KO 5 ROLL
DROP 2 DUP2 GET -1
DUP 8 ROLL ^ * MAL
PUT SWAP -1 4 ROLL
SIZE ^ 1 →LIST MUL
´ca´ STO
≫
```

1: —	1: —

Erläuterung: Führt den Algorithmus nach Faddejew zur Bestimmung der Spektralanteile einer Matrix (in Listen-Schreibweise) durch.
Achtung! Dieses Programm wurde nur als Hilfsprogramm konzipiert.
Beispiel: entfällt
benötigte UP: MIDN, MMUL, SPUR, DE, MAL, MAD, KO, MUL

CARP

Version: HP48 und HP28S
Checksum(HP28S): 25991
Checksum(HP48): 6263
Bytes: 47.0

```
≪ FADD DROP ca 'ca'
PURGE
≫
```

1: Matrix in Listen-Schreibweise	1: charakteristisches Polynom der Matrix

Erläuterung: Berechnet aus einer Matrix (in Listen-Schreibweise) das charakteristische Polynom.
Beispiel: { { 1 2 }{ 5 6 } } CARP ⟹ { 1 -7 -4 }
benötigte UP: FADD

carp

Version: HP48 und HP28S
Checksum(HP28S): 1518838
Checksum(HP48): 5C84
Bytes: 205.0

```
≪ DUP MO DUP { } 1
5 PICK SIZE
  START OVER MI
SPUR + SWAP 3 PICK
* SWAP
  NEXT { 1 } 1 3
PICK SIZE
  FOR i 0 1 i
    FOR j OVER i j
- 1 + GET 4 PICK j
GET * -
    NEXT i / +
  NEXT 5 ROLLD 3
DROPN SIZE -1 SWAP
^ 1 →LIST MUL
≫
```

1: Matrix in Listen-Schreibweise	1: charakteristisches Polynom der Matrix

Erläuterung: Berechnet aus einer Matrix (in Listen-Schreibweise) das charakteristische Polynom.
Das Programm ist sehr schnell, läuft aber nur mit reinen Zahlenmatrizen.
Beispiel: { { 1 2 }{ 5 6 } } carp ⟹ { 1 -7 -4 }
benötigte UP: MO, MI, SPUR, MUL

DIAG

Version: HP48 und HP28S
Checksum(HP28S): 278198
Checksum(HP48): F412
Bytes: 80.0

1: Liste	1: Diagonalmatrix mit den Listenelementen

Erläuterung: Erzeugt aus einer Liste eine Matrix mit den Listenelementen als Diagonalmatrixelemente.
Beispiel: { 1 2 3 } DIAG ⟹
{ { 1 0 0 }{ 0 2 0 }{ 0 0 3 } }
benötigte UP: MIDN

```
≪ DUP SIZE DUP MIDN
1 ROT
  FOR j j DUP2 GET
OVER 5 PICK OVER
GET PUT PUT
  NEXT SWAP DROP
≫
```

ANH

Version: HP48 und HP28S
Checksum(HP28S): 213223
Checksum(HP48): F390
Bytes: 75.5

2: Matrix in Listen-Schreibweise 1: Matrix in Listen-Schreibweise	1: aneinandergehängte Matrizen

Erläuterung: Hängt zwei Matrizen (in Listen-Schreibweise) aneinander.
Beispiel: 2: { { 2 3 }{ 2 3 } }
1: { { 4 5 }{ 4 5 } } ANH ⟹
{ { 2 3 4 5 }{ 2 3 4 5 }}
benötigte UP: keine

```
≪ SWAP 1 OVER SIZE
  FOR f DUP f GET 3
PICK f GET + f SWAP
PUT
  NEXT SWAP DROP
≫
```

MI

Version: HP48 und HP28S
Checksum(HP28S): 500233
Checksum(HP48): C64C
Bytes: 120.5

1: Matrix in der [ ]-Schreibweise	1: Matrix in Listen-Schreibweise

Erläuterung: Wandelt Matrizen vom rechnerinternen [ ]-Format in die Listen-Schreibweise um.
Beispiel: [[1 2]] MI ⟹ { { 1 2 } }
benötigte UP: FA

```
≪ FA ARRY→ LIST→
DROP DUP2 → u w
  ≪ * →LIST 0 u 1 -
    FOR k DUP k w *
DUP 1 + SWAP w +
SUB SWAP
    NEXT DROP u
→LIST
  ≫
≫
```

MO

Version: HP48 und HP28S
Checksum(HP28S): 552164
Checksum(HP48): 6465
Bytes: 124.0

1: Matrix in Listen-Schreibweise	1: Matrix in der [ ]-Schreibweise

Erläuterung: Wandelt eine Matrix (in Listen-Schreibweise) in das rechnerinterne [ ]-Format um.
Achtung! Funktioniert nur, wenn alle evtl. vorhandenen Variablen als Zahlenwerte definiert (also abgespeichert) sind.
Beispiel: { { 1 2 }{ 2 3 } } MO ⟹
[[1 2][2 3]]
benötigte UP: FA

```
≪ FA DUP 1 GET SIZE
→ m u
  ≪ 1 m SIZE
    FOR k 1 u
      FOR l m k GET
l GET →NUM
      NEXT
    NEXT m SIZE u 2
→LIST →ARRY
  ≫
≫
```

SHO

Version: HP48 und HP28S
Checksum(HP28S): 806832
Checksum(HP48): 47DD
Bytes: 156.0

3: Matrix in Listen-Schreibweise 2: zu streichende Zeile 1: zu streichende Spalte	1: „reduzierte“ Matrix

Erläuterung: Ermöglicht das Streichen einer Spalte und einer Zeile einer Matrix (in Listen-Schreibweise).
Beispiel: 3: { { 1 2 3 }{ 4 5 6 }{ 7 8 9 } }
2: 2
1: 3 SHO ⟹ { { 1 2 }{ 7 8 } }
benötigte UP: keine

```
≪ 2 - 3 PICK 1 GET
SIZE → m c d t
  ≪ 1 m SIZE
    FOR l l c
      IF ≠
      THEN m l GET
LIST→ t d - ROLL
DROP 1 - →LIST
      END
    NEXT m SIZE 1 -
→LIST
  ≫
≫
```

MA

Version: HP48 und HP28S
Checksum(HP28S): 114058
Checksum(HP48): C0C5
Bytes: 54.5

2: Zahl 1 1: Zahl 2	1: Matrix der Dimension (Zahl 1) x (Zahl 2) mit Nullen gefüllt

Erläuterung: Erzeugt aus zwei Zahlen eine Null-Matrix mit der Dimension der beiden Zahlen.
Beispiel: 2: 2
1: 3 MA ⟹
{ { 0 0 0 }{ 0 0 0 } }
benötigte UP: LI

```
≪ LI 1 →LIST { } 1
4 ROLL
  START OVER +
  NEXT SWAP DROP
≫
```

DYA

Version: HP48 und HP28S
Checksum(HP28S): 70006
Checksum(HP48): 13D6
Bytes: 55.0

```
≪ 1 →LIST SWAP 1
→LIST MTRN SWAP
MMUL MTRN
≫
```

2: Vektor in Listen-Schreibweise 1: Vektor in Listen-Schreibweise	1: dyadisches Produkt der Vektoren

Erläuterung: Berechnet aus zwei Vektoren (Listen-Schreibweise) das dyadische Produkt.

Beispiel:
2: { 1 2 3 }
1: { 1 2 3 } DYA ⟹
{ { 1 2 3 }{ 2 4 6 }{ 3 6 9 } }

benötigte UP: MTRN, MMUL

qr

Version:	HP48 und HP28S
Checksum(HP28S):	4391774
Checksum(HP48):	C1DA
Bytes:	381.0

	2: Q-Matrix der QR-Zerlegung
1: Matrix in Listen-Schreibweise	1: R-Matrix der QR-Zerlegung

Erläuterung: Führt die QR-Zerlegung einer Matrix (in Listen-Schreibweise) durch.
A = **QR** (ohne Permutationsmatrix)

Beispiel: {{4 8 2 6 0}{2 4 2 4 8}{-2 -4 -4 -6 -2}{-1 -2 -5 -6 -5}} qr ⟹
2: {{4 -2 -2}{2 0 6}{-2 -2 2}{-1 -4 0}}
1: {{1 2 1 2 1}{0 0 1 1 1}{0 0 0 0 1}}

benötigte UP: MTRN, PIV, dot, DYA, MAD, MA, KO, MAL

```
≪ 0 →LIST 1 →LIST
SWAP DUP2 DUP MTRN
SWAP SIZE OVER SIZE
0 → m v u s d r
  ≪
    DO u PIV DROP
´r´ STO { } 1 d
      FOR k u r GET
DUP u k GET dot
SWAP DUP dot / +
      NEXT DUP u r
GET v OVER 1 →LIST
+ ´v´ STO DYA -1
MAL u MTRN MAD MTRN
´u´ STO 1 →LIST + u
    UNTIL d s MA
SAME
    END KO LIST→
SWAP DROP 1 - →LIST
KO v KO LIST→ SWAP
DROP 1 - →LIST KO
MTRN SWAP
  ≫
≫
```

CHOL

Version:	HP48 und HP28S
Checksum(HP28S):	9386126
Checksum(HP48):	64CC
Bytes:	593.5

	2: linke Matrix der Cholesky-Zerlegung
1: Matrix in Listen-Schreibweise	1: rechte Matrix der Cholesky-Zerlegung

Erläuterung: Führt mit einer Matrix (in Listen-Schreibweise) die Cholesky-Zerlegung durch.

Beispiel: { { 1 a 1 }{ a 'a^2+1' '2*a' }{ 1 '2*a' 'a^2+10' } } CHOL ⟹
2: { { 1 0 0 }{ a 1 0 }{ 1 a 3 } }
1: { { 1 a 1 }{ 0 1 a }{ 0 0 3 } }

benötigte UP: MTRN, PIV, MUL, DYA, MAL, MAD, cnrm, KO, DIAG, MMUL, DE

```
≪ 0 OVER SIZE OVER
4 PICK MTRN { }
OVER SIZE OVER 1
→LIST → m p z q u x
s h
  ≪
    DO h 1 DUP2 GET
m PIV DUP2 'p' STO
'q' STO R→C + PUT
'h' STO m q GET DUP
p GET INV 1 →LIST
MUL DUP u p GET h
OVER 1 →LIST + 'h'
STO DYA -1 MAL m
MAD DUP 'm' STO
MTRN 'u' STO
    UNTIL 0 1 z
      FOR k m k GET
cnrm +
      NEXT DE 0
SAME
    END z →LIST h
KO LIST→ SWAP 'h'
STO 1 - →LIST KO
'm' STO 1 h SIZE
    FOR v m h v GET
RE GET DUP h v GET
IM GET DUP √ x +
'x' STO INV 1 →LIST
MUL
    NEXT h SIZE
→LIST MTRN x KO
DIAG MMUL SWAP x KO
DIAG SWAP MMUL
  ≫
≫
```

8.2.5 Polynome

ABL

Version: HP48 und HP28S
Checksum(HP28S): 399299
Checksum(HP48): 5294
Bytes: 104.0

1: Polynom in Listen-Schreibweise	1: Ableitung des Polynoms

Erläuterung: Berechnet die Ableitung eines Polynoms (in Listen-Schreibweise).
Beispiel: { 2 2 -3 -7 } ABL ⟹ { 6 4 -3 }
benötigte UP: keine

```
≪ LIST→ 1 - → u
  ≪ DROP
    IF u
    THEN 1 u
      FOR i i * u
ROLLD
      NEXT u →LIST
    ELSE { 0 }
    END
  ≫
≫
```

MUL

Version: HP48 und HP28S
Checksum(HP28S): 839073
Checksum(HP48): FCD1
Bytes: 145.0

2: Polynom in Listen-Schreibweise 1: Polynom in Listen-Schreibweise	1: Produkt der Polynome

Erläuterung: Multipliziert zwei Polynome (in Listen-Schreibweise) miteinander.
Beispiel: 2: { 1 4 -3 -5 }
1: { 1 0 2 } MUL ⟹
{ 1 4 -1 3 -6 -10 }
benötigte UP: EXCO, LI

```
≪ DUP2 SIZE DUP ROT
SIZE + LI 1 ROT
  FOR k 1 4 PICK
SIZE
    FOR f k f +
DUP2 GET 5 PICK f
GET 5 PICK k GET *
+ EXCO PUT
    NEXT
  NEXT SWAP ROT
DROP2 2 3 ALOG SUB
≫
```

DIV

Version: HP48 und HP28S
Checksum(HP28S): 2248525
Checksum(HP48): 2F04
Bytes: 232.0

	3: Rest der Division
2: Polynom in Listen-Schreibweise	2: Nenner des Restes
1: Polynom in Listen-Schreibweise	1: Abdividierter Teil der Division

Erläuterung: Dividiert zwei Polynome (in Listen-Schreibweise) miteinander und gibt den Rest, den Nenner des Restes und den abdividierten Teil aus.

Beispiel: 2: { 1 3 -5 2 }
1: { 1 0 -1 } DIV $\Longrightarrow$ 3: { -4 5 }
2: { 1 0 -1 }
1: { 1 3 }

Das Ergebnis stellt

$$x + 3 + \frac{-4x + 5}{x^2 - 1}$$

dar.

benötigte UP: EXCO, KO, EK

```
≪ OVER SIZE OVER
SIZE - DUP 0
  IF ≥
  THEN 1 + { } 1
ROT
    FOR i 3 PICK 1
GET 3 PICK 1 GET /
ROT 4 ROLL 1 3 PICK
SIZE
    FOR j j DUP2
GET 5 PICK 5 PICK j
GET * - EXCO PUT
    NEXT 2 3 ALOG
SUB SWAP 4 ROLL 4
ROLL +
  NEXT ROT KO EK
KO ROT ROT
  ELSE DROP { 0 }
  END
≫
```

AD

Version: HP48 und HP28S
Checksum(HP28S): 196658
Checksum(HP48): D1B0
Bytes: 78.0

```
≪ AU 1 OVER SIZE
  FOR i i DUP2 GET
4 PICK i GET + EXCO
PUT
  NEXT SWAP DROP
≫
```

2: Listen-Polynom 1: Listen-Polynom	1: Summe der Listen-Polynome

Erläuterung: Berechnet die Summe zweier Polynome (in Listen-Schreibweise).
Beispiel: 2: { 1 2 3 4 }
1: { 1 2 0 0 } AD ⟹ { 2 4 3 4 }
benötigte UP: AU, EXCO

DI

Version: HP48 und HP28S
Checksum(HP28S): 22242
Checksum(HP48): 794F
Bytes: 52.5

```
≪ { -1 } MUL AD KO
EK KO
≫
```

2: Listen-Polynom 1: Listen-Polynom	1: Differenz der Listen-Polynome

Erläuterung: Berechnet die Differenz zweier Polynome (in Listen-Schreibweise).
(Element der Stack-Ebene 2 minus Element der Stack-Ebene 1)
Beispiel: 2: { 1 2 3 0 }
1: { 2 3 5 } DI ⟹ { 1 0 0 -5 }
benötigte UP: MUL, AD, KO, EK

NO

Version: HP48 und HP28S
Checksum(HP28S): 30216
Checksum(HP48): DE81
Bytes: 38.0

```
≪ DUP 1 GET INV 1
→LIST MUL
≫
```

1: Polynom in Listen-Schreibweise	1: normiertes Polynom

Erläuterung: Normiert ein Polynom (in Listen-Schreibweise).
Beispiel: { 2 4 6 } NO ⟹ { 1 2 3 }
benötigte UP: MUL

PO

Version: HP48 und HP28S
Checksum(HP28S): 109793
Checksum(HP48): 13F9
Bytes: 77.0

```
≪ LIST→ 0 SWAP 1 +
2
  FOR e e ROLL 'X'
e 2 - ^ * + -1
  STEP
≫
```

1: Polynom in Listen-Schreibweise	1: Polynom in üblicher Schreibweise

Erläuterung: Stellt ein Polynom (in Listen-Schreibweise) in der üblichen Darstellungart dar.
Beispiel: { 1 2 3 } PO ⟹ 'X^2+2*X+3'
benötigte UP: keine

PI

Version: HP48 und HP28S
Checksum(HP28S): 532850
Checksum(HP48): 8CED
Bytes: 99.0

2: Polynom in üblicher Schreibweise 1: Name der unabh. Variable	1: Polynom in Listen-Schreibweise

Erläuterung: Stellt das Gegenstück zu „PO“ dar. Es werden hierbei die Koeffizienten eines Polynoms bestimmt und geordnet in eine Liste eingetragen.

Beispiel: 2: 'X^2-X+R+3*X'
1: 'X' PI $\Longrightarrow$ { 1 2 R }

benötigte UP: EXCO

```
≪ { } SWAP ROT
  DO DUP2 0 ROT STO
EVAL 4 PICK SIZE
FACT / EXCO 4 ROLL
+ ROT ROT OVER DUP
PURGE ∂
  UNTIL DUP 0 SAME
  END DROP2
≫
```

8.2.6 Gebrochen rationale Funktionen

Bei den hier aufgeführten Programmen werden einige ein Programm zum Dividieren zweier gebrochen rationaler Zahlen vermissen. Dies ist aber nicht notwendig, da es durch die Befehle KO und FMUL ersetzt werden kann. Wer will, kann sich ein Programm ≪ KO FMUL ≫ 'FDIV' STO abspeichern, was ich aber für unnötig halte.
Ein Befehl '1 durch gebrochen rationale Funktion' ist durch das Programm KO überflüssig. KO aufzurufen hat die gleiche Funktion.

FMUL

Version: HP48 und HP28S
Checksum(HP28S): 145643
Checksum(HP48): D16F
Bytes: 61.5

```
≪ LIST→ DROP ROT
LIST→ ROT 5 ROLL
MUL ROT 4 ROLL MUL
ROT →LIST
≫
```

2: gebrochen rationale Funktion in Listen-Schreibweise	
1: gebrochen rationale Funktion in Listen-Schreibweise	1: miteinander multiplizierte gebrochen rationale Funktion

Erläuterung: Multipliziert zwei gebrochen rationale Funktionen (in Listen-Schreibweise) miteinander.
Beispiel: 2: { { 1 2 }{ 1 2 3 } }
1: { { 2 4 }{ 1 2 3 } } FMUL ⟹
{ { 2 8 8 }{ 1 4 10 12 9 } }
benötigte UP: MUL

FABL

Version: HP48 und HP28S
Checksum(HP28S): 171667
Checksum(HP48): 70D5
Bytes: 81.5

```
≪ LIST→ PICK ABL
OVER MUL OVER ABL 4
ROLL MUL DI SWAP
DUP MUL 2 →LIST
≫
```

1: gebrochen rationale Funktion in Listen-Schreibweise	1: abgeleitete Funktion

Erläuterung: Leitet eine gebrochen rationale Funktion (in Listen-Schreibweise) ab.
Beispiel: { { 1 2 3 }{ 3 2 1 } } FABL ⟹
{ { -4 -16 -4 }{ 9 12 10 4 1 } }
benötigte UP: ABL, MUL, DI

FAD

Version: HP48 und HP28S
Checksum(HP28S): 5405
Checksum(HP48): 5E0C
Bytes: 34.5

```
≪ 'AD' adD
≫
```

2: gebrochen rationale Funktion in Listen-Schreibweise 1: gebrochen rationale Funktion in Listen-Schreibweise	1: addierte gebrochen rationale Funktion

Erläuterung: Addiert zwei gebrochen rationale Funktionen (in Listen-Schreibweise).
Beispiel: 2: { { 3 0 0 }{ 1 2 3 } }
1: { { 2 1 }{ 1 2 3 } } FAD ⟹
{ { 3 2 1 }{ 1 2 3 } }
benötigte UP: adD

FDI

Version: HP48 und HP28S
Checksum(HP28S): 5442
Checksum(HP48): C99F
Bytes: 34.5

```
≪ 'DI' adD
≫
```

2: gebrochen rationale Funktion in Listen-Schreibweise	
1: gebrochen rationale Funktion in Listen-Schreibweise	1: subtrahierte, gebrochen rationale Funktion

Erläuterung: Subtrahiert zwei gebrochen rationale Funktionen (in Listen-Schreibweise).
Beispiel: 2: { { 3 2 1 }{ 1 2 3 } }
1: { { 3 0 0 }{ 1 2 3 } } FDI ⟹
1: { { 2 1 }{ 1 2 3 } }
benötigte UP: adD

adD

Version: HP48 und HP28S
Checksum(HP28S): 434082
Checksum(HP48): 11D8
Bytes: 101.0

```
≪ ROT LIST→ DROP 4
ROLL LIST→ OVER 6
ROLL MUL 4 ROLL 5
PICK MUL 6 ROLL
EVAL 3 PICK 5 ROLL
MUL ROT →LIST SWAP
KUE
≫
```

1: —	1: —

Erläuterung: Reines Hilfsprogramm für FAD und FDI.
Beispiel: entfällt
benötigte UP: MUL, KUE

KUE

Version: HP48 und HP28S
Checksum(HP28S): 1025691
Checksum(HP48): 71C2
Bytes: 151.5

2: gebrochen rationale Funktion in Listen-Schreibweise 1: zu kürzendes Polynom in Listen-Schreibweise	1: gekürzte, gebrochen rationale Funktion, falls Kürzen möglich war

Erläuterung: Versucht aus einer gebrochen rationalen Funktion ein Polynom herauszukürzen. Ist dies möglich, wird die gekürzte Funktion zurückgegeben, ansonsten die Originalfunktion.

Beispiel: 2: { { 1 2 1 }{ 1 5 7 3 } }
1: { 1 2 1 } KUE $\Longrightarrow$
{ { 1 }{ 1 3 } }

benötigte UP: DIV, DE

```
≪ 2 CF SWAP LIST→
OVER 5 ROLL DIV ROT
DE { 0 }
  IF ==
  THEN 5 PICK ROT
DIV ROT DE { 0 }
    IF ==
    THEN ROT ROT
DROP ROT 5 ROLL 5
ROLL
    ELSE DROP
    END
  END DROP2 →LIST 2
SF
≫
```

8.2.7 Allgemeine Programme für die Elektrotechnik

HVS

Version: HP48 und HP28S
Checksum(HP28S): 95097
Checksum(HP48): 479E
Bytes: 79.5

```
≪ CC AA PSI 3 ROLLD
BB MMUL MMUL SWAP
"+" DD 2 →LIST
≫
```

	3: Zähler der Übertragungsfunktion
	2: Nenner der Übertragungsfunktion in Listen-Schreibweise
1: nichts	1: { { + DD } }

Erläuterung: Berechnet aus der A-, B-, C- und D-Matrix (der Zustandsdarstellung) die Übertragungsfunktion.

Beispiel: Die A-, B-, C- und D-Matrix muß als AA-, BB-, CC- und DD-Matrix abgespeichert sein.

{ { 1 2 }{ 2 1 } } 'AA' STO
{ { 1 }{ 0 } } 'BB' STO
{ { 1 0 } } 'CC' STO
{ { 1 } } 'DD' STO HVS $\Longrightarrow$
3: { { '-1+S' } }
2: { 1 -2 -3 }
1: { "+" { { 1 } } } für

$$H(S)=\frac{-1+S}{S^2-2S-3}+1$$

benötigte UP: MMUL, PSI

GLZ

Version: HP48 und HP28S
Checksum(HP28S): 845964
Checksum(HP48): 9084
Bytes: 159.0

```
≪ 'W' i →NUM OVER *
ROT 'D'
  IF SAME
  THEN EXP
  END 'X' STO STO
"XVJ√'W'EHTHUCaSUPTCBRWQD"
XPAC OVER 2 GET
OVER EVAL ROT 1 GET
ROT EVAL - EVAL { W
X } PURGE
≫
```

3: gebrochen rationale Funktion in Listen-Schreibweise 2: Wert der Frequenz Omega 1: 'D' für diskretes System, sonst kontinuierliches System	1: Gruppenlaufzeit

Erläuterung: Berechnet die Gruppenlaufzeit für kontinuierliche und diskrete Systeme.
Beispiel: 3: { { 1 2 3 }{ 4 0 0 8 2 } }
2: 0
1: 'D' GLZ $\Longrightarrow$ 1.04761904762
Wenn man in der untersten Stackzeile etwas anderes als 'D' schreibt, wird die Gruppenlaufzeit für ein kontinuierliches System berechnet (hier 3.33333333333).
benötigte UP: PO, XPAC

DB

Version: HP48 und HP28S
Checksum(HP28S): 10111
Checksum(HP48): C4F6
Bytes: 31.5

```
≪ ABS LOG 4 5 * *
≫
```

1: REAL-Zahl	1: REAL-Zahl in dB umgerechnet

Erläuterung: Rechnet eine Zahl um in dB.
Beispiel: .001 DB $\Longrightarrow$ -60
benötigte UP: keine

DB→

Version: HP48 und HP28S
Checksum(HP28S): 8184
Checksum(HP48): EB70
Bytes: 30.0

```
≪ 4 5 * / ALOG
≫
```

1: REAL-Zahl	1: REAL-Zahl umgerechnet aus dB

Erläuterung: Berechnet den Wert einer Zahl, die in dB eingegeben wurde.
Beispiel: -60 dB ⟹ .001
benötigte UP: keine

LOGS

Version: HP48 und HP28S
Checksum(HP28S): 5072
Checksum(HP48): 9ACD
Bytes: 26.0

```
≪ 5 / ALOG
≫
```

1: Abstand in Zentimetern	1: Angabe der Frequenz

Erläuterung: Ist stark auf die Nutzung in der Regelungstechnik ausgelegt. Zum Zeichnen von Bode-Diagrammen empfiehlt sich ein Abstand von 5 cm pro Dekade für die Frequenzachse. Dadurch folgt, daß ziemlich genau im Abstand von 1.5 cm einer Frequenz die doppelte (bzw. die halbe) Frequenz liegt. Für die Errechnung von Zwischenwerten eignet sich dieses Programm.
Beispiel: Hier gibt man den Abstand in cm von der Freq 1 (Hz) ein und erhält die Frequenz (Abstände links der 1 (Hz) als negative Zahlen eingeben).
1.2 LOGS ⟹ 1.73780082875
benötigte UP: keine

WLOG

Version: HP48 und HP28S
Checksum(HP28S): 3813
Checksum(HP48): DDDA
Bytes: 26.0

```
≪ LOG 5 *
≫
```

1: Frequenz	1: Abstand in Zentimetern

Erläuterung: Ist stark auf die Nutzung in der Regelungstechnik ausgelegt. Zum Zeichnen von Bode-Diagrammen empfiehlt sich ein Abstand von 5 cm pro Dekade. Dadurch folgt, daß ziemlich genau im Abstand von 1.5 cm einer Frequenz die doppelte (bzw. die halbe) Frequenz liegt. Um Zwischenwerte zu errechnen eignet sich dieses Programm.

Beispiel: Bei WLOG gibt man die Frequenz ein, die man einzeichnen möchte und erhält den Abstand in cm von der Frequenz 1 (Hz).
Negative Zahlen kommen heraus, wenn der gesuchte Punkt links der 1 (Hz) liegt.
1.3 WLOG $\Longrightarrow$ .569716761535

benötigte UP: keine

G1

Version: HP48 und HP28S
Checksum(HP28S): 29067
Checksum(HP48): 1DA9
Bytes: 45.5

```
≪ → x
  ≪ x 0 ≥ 1 0 IFTE
  ≫
≫
```

1: REAL-Zahl	1: Wert der Sprungfunktion

Erläuterung: Simuliert die Sprungfunktion, d.h. es liefert bei negativen Argumenten eine Null als Ergebnis, ansonsten eine Eins.

Beispiel: -2 G1 $\Longrightarrow$ 0
.1 G1 $\rightarrow$ 1
benötigte UP: keine

G0

Version: HP48 und HP28S
Checksum(HP28S): 15769
Checksum(HP48): B722
Bytes: 35.5

```
≪ → x
  ≪ x NOT
  ≫
≫
```

1: REAL-Zahl	1: Wert der Impulsfunktion

Erläuterung: Berechnet den Wert der Impulsfunktion an der Argumentenstelle, d.h. das Programm liefert eine Null, wenn das Argument ungleich Null ist, ansonsten eine Eins.

Beispiel: .001 G0 $\Longrightarrow$ 0
0 G0 $\rightarrow$ 1
benötigte UP: keine

D→S

Version: HP48 und HP28S
Checksum(HP28S): 306326
Checksum(HP48): 5937
Bytes: 105.0

```
≪ 3 DUPN + + 4 PICK
4 PICK * OVER /
EXCO 5 ROLL 4 PICK
* 3 PICK / EXCO 5
ROLLD 5 ROLLD INV *
* EXCO
≫
```

3: Element zwischen Knoten 1 und 2	3: Element Sternmittelpunkt zu Knoten 1
2: Element zwischen Knoten 1 und 3	2: Element Sternmittelpunkt zu Knoten 2
1: Element zwischen Knoten 2 und 3	1: Element Sternmittelpunkt zu Knoten 3

Erläuterung: Führt eine Dreieck-Stern-Umwandlung durch.

Beispiel:
3: 'R'
2: '2*R'
1: '3*R' D→S ⟹
3: '.3333333333333*R'
2: '.5*R'
1: 'R'

benötigte UP: EXCO

S→D

Version: HP48 und HP28S
Checksum(HP28S): 386075
Checksum(HP48): 750F
Bytes: 112.5

3:	Element Sternmittelpunkt zu Knoten 1	3:	Element zwischen Knoten 1 und 2
2:	Element Sternmittelpunkt zu Knoten 2	2:	Element zwischen Knoten 1 und 3
1:	Element Sternmittelpunkt zu Knoten 3	1:	Element zwischen Knoten 2 und 3

Erläuterung: Führt eine Stern-Dreieck-Umwandlung durch.

Beispiel:
3: '.33333333333333*R'
2: '.5*R'
1: 'R' S→D ⟹
3: '.999999999996*R'
2: '2*R'
1: '3*R'

benötigte UP: EXCO

```
≪ INV ROT INV ROT
INV 3 DUPN + + 3
DUPN INV * * INV
EXCO OVER 6 PICK 6
ROLL * / EXCO ROT 5
ROLL 5 ROLL * /
EXCO
≫
```

ROU

Version:	HP28S
Checksum(HP28S):	11091983
Checksum(HP48):	entfällt
Bytes:	485.5

1: Polynom in Listen-Schreibweise	1: durchgeführter Routh-Test

Erläuterung: Führt den Routh-Test mit einem Polynom (in Listen-Schreibweise) durch.
Das Programm beherrscht alle möglichen Sonderfälle, wie z. B. das Auftreten einer ganzen Zeile Nullen (siehe Beispiel) oder einer einzigen Null am Anfang einer Zeile.
Der Routh-Test dient der Bestimmung der Lage der Nullstellen in der linken oder rechten (komplexen) Halbebene.

Beispiel: { 6 6 22 19 24 15 8 2 } ROU ⟹
Ergebnis: (die Stackzeilennummern stehen hier auf dem Kopf)
1: { 6 22 24 8 }
2: { 6 19 15 2 }
3: { 3 9 6 }
4: { 1 3 2 }
5: { 0 0 }
6: 'end'
7: 'X4+3*X2+2'
8: { 1 3 2 }
9: { 4 6 }
10: { '3/2' 2 }
11: { '2/3' }
12: { 2 }
Der Routh-Test bricht zwischenzeitlich ab, wird aber automatisch mit dem Rest-Polynom P+P' wieder weitergeführt.

benötigte UP: EK, ES, ABL, PO, AD, GE, EXCO, DE, TA

```
≪ { } 1 3 PICK SIZE
  FOR e OVER e GET +
2
  STEP SWAP { } OVER
SIZE 1
  IF ≠
  THEN 2 3 PICK SIZE
    FOR e OVER e GET
+ 2
    STEP SWAP DROP
    DO DUP DUP 1 GET
0
      IF SAME
      THEN EK SIZE 1
        IF ==
        THEN end { }
1 5 PICK SIZE
          FOR f 4
PICK f GET + 0 +
          NEXT ES
DUP PO SWAP DUP ABL
AD ROU
        ELSE 1 'e'
PUT
        END
      ELSE DROP
      END { } 2 4
PICK SIZE
      FOR e OVER 1
GE 4 PICK e GE * 4
PICK 1 GE 4 PICK e
GE * - 3 PICK 1 GE /
EXCO DE +
      NEXT
    UNTIL OVER SIZE
1 == OVER DUP SIZE 1
== SWAP 1 GET 0 SAME
NOT AND AND
    END
  ELSE SWAP DROP 0 +
  END DEPTH TA ABORT
≫
```

ROU

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): 7592
Bytes: 485.5

1: Polynom in Listen-Schreibweise	1: durchgeführter Routh-Test

Erläuterung: Führt den Routh-Test mit einem Polynom (in Listen-Schreibweise) durch.
Wenn das Programm mit 'HALT' stoppt, ist noch der Befehl 'KILL' einzugeben.
Das Programm beherrscht alle mögliche Sonderfälle, wie z. B. das Auftreten einer ganzen Zeile Nullen (siehe Beispiel) oder einer einzigen Null am Anfang einer Zeile.
Der Routh-Test dient der Bestimmung der Lage der Nullstellen in der linken oder rechten (komplexen) Halbebene.

Beispiel: { 6 6 22 19 24 15 8 2 } ROU ⟹
Ergebnis: (die Stackzeilennummern stehen hier auf dem Kopf)
1: { 6 22 24 8 }
2: { 6 19 15 2 }
3: { 3 9 6 }
4: { 1 3 2 }
5: { 0 0 }
6: 'end'
7: 'X^4+3*X^2+2'
8: { 1 3 2 }
9: { 4 6 }
10: { '3/2' 2 }
11: { '2/3' }
12: { 2 }
Der Routh-Test bricht zwischenzeitlich ab, wird aber automatisch mit dem Rest-Polynom P+P' wieder weitergeführt.

benötigte UP: EK, ES, ABL, PO, AD, GE, EXCO, DE, TA

```
≪ { } 1 3 PICK SIZE
  FOR e OVER e GET
+ 2
  STEP SWAP { }
OVER SIZE 1
  IF ≠
  THEN 2 3 PICK
SIZE
    FOR e OVER e
GET + 2
    STEP SWAP DROP
    DO DUP DUP 1
GET 0
      IF SAME
      THEN EK SIZE
1
        IF ==
        THEN end {
} 1 5 PICK SIZE
          FOR f 4
PICK f GET + 0 +
          NEXT ES
DUP PO SWAP DUP ABL
AD ROU
        ELSE 1 'e'
PUT
        END
      ELSE DROP
      END { } 2 4
PICK SIZE
      FOR e OVER 1
GE 4 PICK e GE * 4
PICK 1 GE 4 PICK e
GE * - 3 PICK 1 GE
/ EXCO DE +
      NEXT
    UNTIL OVER SIZE
1 == OVER DUP SIZE
1 == SWAP 1 GET 0
SAME NOT AND AND
    END
```

```
  ELSE SWAP DROP 0
+
  END DEPTH TA HALT
≫
```

PSI

Version:	HP48 und HP28S
Checksum(HP28S):	243927
Checksum(HP48):	6C7B
Bytes:	105.5

1: Matrix in Listen-Schreibweise	2: Übergangsmatrix 1: charakteristisches Polynom der Matrix

Erläuterung: Berechnet die Übergangsmatrix einer Matrix (in Listen-Schreibweise), sowie deren charakteristisches Polynom.

Beispiel: { { 1 2 }{ 3 2 } } PSI ⟹
2: { { '-2+S' 2 }{ 3 '-1+S' } }
1: { 1 -3 -4 }

benötigte UP: MAL, MIDN, MAD, MADJ, CARP, MUL

```
≪ DUP -1 MAL OVER
SIZE MIDN ´S´ MAL
MAD MADJ SWAP DUP
CARP -1 ROT SIZE ^
1 →LIST MUL
≫
```

OK

Version:	HP28S
Checksum(HP28S):	826365
Checksum(HP48):	entfällt (eingebaute Funktion des HP48)
Bytes:	135.0

1: —	1: —

Erläuterung: Malt eine in 'EQ' abgespeicherte Ortskurve in Abhängigkeit der Variablen 'W'.
Dieses Programm ist eine einfachere Version des Zeichnens als mit dem Mode PARAMETRIC beim HP48.

Beispiel: '(0,1)*W/(1+(0,1)*W))' STEQ
ENTER OK ENTER
Das Malen des Bildes braucht etwas Zeit. Es wird zuerst ein Halbkreis (ausgehend von [0,0]) gemalt. Dieser entspricht dem Durchlauf mit positivem W, anschließend wird noch der Durchlauf mit negativem W durchgerechnet und ganz am Schluß die beiden Teilbilder aufaddiert.

benötigte UP: FA

```
≪ 1 DUP 2
  START -2 ALOG *
´W´ STO CLLCD FA
    DO W 2 ACOSH *
´W´ STO RCEQ i + i -
→NUM PIXEL
    UNTIL W ABS 7
FACT ≥
    END LCD→ -1
  NEXT DROP OR →LCD
DGTIZ
≫
```

MODAL

Version: HP48 und HP28S
Checksum(HP28S): 639063
Checksum(HP48): 9372
Bytes: 142.5

	3: A-Matrix in Listen-Schreibweise 2: Liste mit den Nullstellen des char. Polynoms
1: A-Matrix in Listen-Schreibweise	1: Modal-Matrix in Listen-Schreibweise

```
≪ DUP carp 1 2 3
PICK SIZE
  FOR i ROLL NULL 2
GET NEG i
  NEXT 1 - →LIST
DUP2 1 GET EV 2 4
PICK SIZE
  FOR i 3 PICK 3
PICK i GET EV ANH
  NEXT
≫
```

Erläuterung: Berechnet aus der A-Matrix einer Zustandsdarstellung die Modal-Matrix.

Beispiel: { { 2 2 0 }{ 2 0 2 }{ 0 2 2 } } MODAL
$\Longrightarrow$
3: { { 2 2 0 }{ 2 0 2 }{ 0 2 2 } }
2: { -2 4 2 }
1: { { 1 -1 1 }{ -2 0 1 }{ 1 1 1 } }

Vorsicht! Es sind mehrere Darstellungen der Modal-Matrix möglich, da diese nicht eindeutig ist! Das Programm liefert jeweils eine gültige Form!

benötigte UP: carp, NULL, EV, ANH

8.2.8 Netzwerk-Analyse-Programm

Die folgenden Programme stellen ein simples Netzwerk-Analyse-Programm dar. Um dieses nutzen zu können ist eine relativ aufwendige Erklärung notwendig, um die Möglichkeiten des Programmpakets zu erläutern. Da aufgrund von Speicherschwierigkeiten das Programmpaket sehr kurz gehalten wurde, sind die Möglichkeiten relativ beschränkt. In den Netzwerken sind lediglich Stromquellen, Widerstände, Spulen und Kondensatoren zugelassen, die allerdings beliebig miteinander verknüpft werden dürfen. Eine Begrenzung der Knoten oder Netzwerkelemente gibt es nicht - lediglich Speichergröße und Rechenzeit stellen eine Grenze dar. Das Programmpaket führt eine rein numerische Auswertung des Netzwerks durch.

Die Bedienung soll anhand eines Beispiels erläutert werden. Gegeben ist das Netzwerk aus Abbildung 8–1, das nun analysiert werden soll:

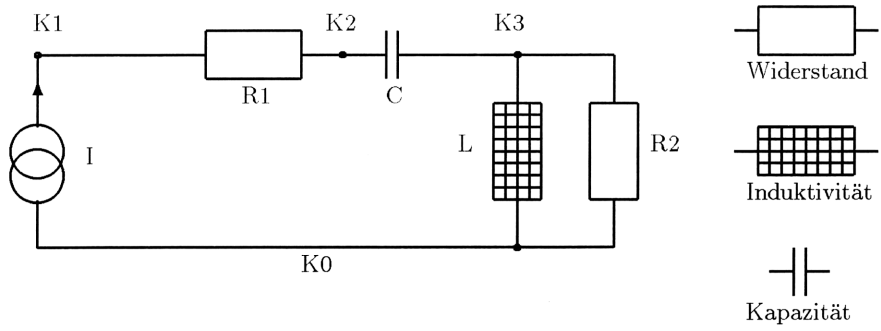


Abb. 8–1: *Netzwerk zur Übungsaufgabe*

Das Netzwerk hat 4 Knoten(K_X), eine Stromquelle und 4 Bauelemente. Das Aussehen des Netzwerkes gibt man dem Programm über 2 Matrizen an. Diese sind die Impedanzen-Matrix und die Stromquellen-Matrix.

Die Impedanzen-Matrix enthält alle Bauelemente, die je nach Lage im Netzwerk in die Matrix eingetragen werden. Das Schema ist folgendermaßen:

$$\begin{pmatrix} K0-K0 & K0-K1 & K0-K2 & \dots \\ 0 & K1-K1 & K1-K2 & \dots \\ 0 & 0 & K2-K2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Also hier gilt für die Impedanzen-Matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 'L+R_2' \\ 0 & 0 & 'R_1' & 0 \\ 0 & 0 & 0 & C \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Wenn sich mehrere Elemente zwischen zwei Knoten befinden, braucht man sich keine großen Gedanken machen, wie dies geschrieben werden muß. Angenommen zwischen zwei Knoten wären eine Spule, ein Kondensator und ein Widerstand: In der Matrix schreibt man trotzdem 'R+C+L'. Das Programmpaket ordnet aufgrund der Buchstaben R, C und L die Frequenzabhängigkeit korrekt zu. Die Stromquellenmatrix sieht so aus:

$$\begin{pmatrix} 0 & I & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Nun gibt man also die Matrizen so ein:

2: {{0 0 0 'L+R2'}{0 0 R1 0}{0 0 0 C}{0 0 0 0}}
1: {{0 I 0 0}{0 0 0 0}{0 0 0 0}{0 0 0 0}} LINA ENTER

Nach einer kurzen Weile ist der Rechner fertig. Nun speichert man die Bauelementewerte ab:

Hier 1 'I' STO 1 'C' STO 1 'L' STO 1 'R1' STO 2 'R2' STO.

Als nächsten Schritt gibt man den zu untersuchenden Frequenzbereich und ein Programm zum Bestimmen der gesuchten Werte ein. Dies geschieht folgendermaßen:

1. Man entscheidet sich für zwei Frequenzen, zwischen denen das Netzwerk untersucht werden soll. Das Programm erwartet zwei Werte in einer Liste. Als Option verdaut es auch noch einen dritten Wert, der dann als Schrittweite interpretiert wird. Gibt man diesen dritten Wert nicht ein, so wählt der Rechner 137 Zwischenschritte.

2. Um Daten zu erhalten, muß man ein simples Programm selbst schreiben. Das Programm bestimmt an diskreten Frequenzwerten alle Knotenspannungen. Möchte man z.B. einen Strom (z.B. den durch den Kondensator im Beispiel oben), so muß man etwas mehr Aufwand treiben. Eine Spannung erhält man dagegen recht einfach. Ich gebe stellvertretend nun ein Programm an, das den Betrag der Spannung über die Spule und das Verhältnis aus dem Betrag des Stroms durch den Kondensator zum Betrag der Spannung am Eingang des Netzwerkes berechnet. Dieser Strom ergibt sich aus dem Spannungsabfall am Kondensator „u3 u2 -“ (in UPN), multipliziert mit jwC „(0,1) W C * *“ und „* ABS“ für die Multiplikation und die Betragswertbildung. Das Verhältnis zur Eingangsspannung erhält man durch anschließende Division durch den Betrag von u1 „u1 ABS /“.

```
≪ u3 u2 - (0,1) W C * * * ABS            /* Betrag des Stromes durch C */
u1 ABS /                          /* durch den Betrag der Eingangsspannung */
u3 ABS                                   /* Betrag der Spannung über L */
2 ≫                                  /* Anzahl der zu berechnenden Werte */
```

Zu beachten sind folgende Einzelheiten:

1. Die Spannungen fangen mit einem kleinen u an und sind durchnummeriert (nach der Knotennummer). Sie werden immer als Spannung vom Knoten X zum Knoten 0 aufgefaßt.
2. Am Ende des Programms zur Datenbestimmung muß eine Zahl stehen, die genau der Anzahl der zu berechnenden Werte entspricht.
3. Man kann auf Omega zugreifen (Variable W) und kann Omega ganz normal einrechnen.

Zum Einsatz im Programm soll das obige Beispiel fortgeführt werden.

2: { .1 1 .015 }
1: ≪ ...siehe obiges Beispiel... ≫ GO ENTER

Nun dauert es zugegebenerweise recht lange (HP48G: 150 sec), dann erscheint ein Menü: { FDRW FMIT WERT }.

FDRW erwartet eine Zahl (Kennzahl des Datensatzes) und malt dann die entsprechende Funktion .

1 FDRW zeichnet also im Beispiel das Verhältnis aus Strom in Kondensator und Eingangsspannung und 2 FDRW malt den Betrag der Ausgangsspannung u3.

FMIT erwartet die Werte PMIN und PMAX (als Bildschirmausschnitt [Eckdaten]) und dann noch die Kennzahl des Datensatzes (in dieser Reihenfolge).

WERT erwartet nur die Kennzahl des Datensatzes. Dieses Unterprogramm gibt dann die berechneten Werte als Zahlen am Bildschirm aus. Die ausgegebenen Werte sind: in der ersten Zeile die Kreisfrequenz, in der zweiten die Datensatznummer, in der dritten das Ergebnis der Berechnung und in der vierten bzw. fünften die Frequenz. Die Tastensteuerung erlaubt:

– „4“ Datensatznummer + 1

– „1“ Datensatznummer - 1

– „5“ Datensatznummer + 10

– „2“ Datensatznummer - 10

– „0“ Ende des Unterprogramms

LINA

Version: HP48 und HP28S
Checksum(HP28S): 2800994
Checksum(HP48): FBBA
Bytes: 318.0

2: Impedanzenmatrix 1: Stromquellenmatrix	1: nichts

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: MTRN, MAD, MAL, cnr, KN

```
≪ OVER MTRN OVER -1
MAL ROT MTRN MAD
ROT ROT MAD OVER
SIZE ROT i →NUM 'W'
* 'S' STO { } 2 4
PICK
  FOR e OVER e GET
cnr +
  NEXT 'V' STO DROP
'T' STO 2 T
  FOR e { } 2 T
    FOR q e PICK q
e
      IF ≠
      THEN e GET q
GET KN NEG
      ELSE q GET
cnr KN
      END +
    NEXT
  NEXT T 1 - →LIST
'T' STO DROP
≫
```

KN

Version:	HP28S
Checksum(HP28S):	1867174
Checksum(HP48):	entfällt
Bytes:	198.5

1: —	1: —

Erläuterung:	siehe oben
Beispiel:	entfällt
benötigte UP:	EXCO

```
≪ DUP
  IF TYPE
  THEN 0 'Y' STO DUP
SIZE 1 + 2 / 1 SWAP
    START DUP 1
EXGET DUP DUP →STR 2
DUP SUB DUP "R"
      IF ≠
      THEN SWAP S *
SWAP "C"
        IF ≠
        THEN INV
        END
      ELSE DROP INV
      END Y + 'Y'
STO - EXCO
    NEXT DROP Y
  END
≫
```

KN

Version:	HP48
Checksum(HP28S):	entfällt
Checksum(HP48):	B39A
Bytes:	231.0

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: EXCO

```
≪ DUP
  IF TYPE
  THEN 0 'Y' STO 1
SF EXCO DUP SIZE 1
+ 2 / 1 SWAP
    START DUP
      IFERR OBJ→
      THEN DROP 0
SWAP
      ELSE DROP2
ROT DROP
      END DUP →STR
2 DUP SUB DUP "R"
      IF ≠
      THEN SWAP S *
SWAP "C"
        IF ≠
        THEN INV
        END
      ELSE DROP INV
      END Y + 'Y'
STO
    NEXT DROP Y
  END
≫
```

GO

Version:	HP28S
Checksum(HP28S):	4957578
Checksum(HP48):	entfällt
Bytes:	409.0

2: Liste mit den Frequenzeckpunkten 1: Programm zum Berechnen der Daten	1: Menüleiste

Erläuterung:	siehe oben
Beispiel:	entfällt
benötigte UP:	MUL, FA, MO, FDRW, FMIT, WERT

```
≪ 2 π * →NUM 'P' STO
'j' STO 'f' STO CLΣ
f P 1 →LIST MUL
LIST→ 2
  IF ==
  THEN DUP 3 PICK -
137 /
  END 'ST' STO 'FX'
STO -9 ALOG + DUP
'FN' STO
  DO 'W' STO W P /
DUP 1 DISP FA V SIZE
1 OVER
    FOR q V q GET
→NUM
    NEXT V SIZE 1
→LIST →ARRY T MO / 1
ROT
    FOR w DUP w GET
"'u" w →STR + STR→
STO
    NEXT DROP j EVAL
1 + →ARRY Σ+ W ST +
  UNTIL FX W ≤
  END DROP CLMF {
FDRW FMIT WERT }
MENU
≫
```

GO

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): AA0
Bytes: 406.5

2: Liste mit den Frequenzeckpunkten 1: Programm zum Berechnen der Daten	1: Menüleiste

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: MUL, FA, MO, FDRW, FMIT, WERT

```
≪ 2 π * →NUM ´P´
STO ´j´ STO ´f´ STO
CLΣ f P 1 →LIST .MUL
LIST→ 2
  IF ==
  THEN DUP 3 PICK -
137 /
  END ´ST´ STO ´FX´
STO -9 ALOG + DUP
´FN´ STO
  DO ´W´ STO W P /
DUP 1 DISP FA V
SIZE 1 OVER
    FOR q V q GET
→NUM
    NEXT V SIZE 1
→LIST →ARRY T MO /
1 ROT
    FOR w DUP w GET
"´u" w →STR + STR→
STO
    NEXT DROP j
EVAL 1 + →ARRY Σ+ W
ST +
  UNTIL FX W ≤
  END DROP { FDRW
FMIT WERT } TMENU
≫
```

WERT

Version:	HP28S
Checksum(HP28S):	5561436
Checksum(HP48):	entfällt
Bytes:	344.5

1: —	1: —

Erläuterung:	siehe oben
Beispiel:	entfällt
benötigte UP:	WA

```
≪ ΣDAT SIZE LIST→
DROP 0
  DO DUP 2 DISP ΣDAT
OVER 4 PICK * 1 +
GET ΣDAT 3 PICK 5
PICK * 1 + 7 PICK +
GET ´erg´ = 3 DISP
DUP 2 π * →NUM * ´w´
= 1 DISP ´f´ = 4
DISP WA STR→ DUP 1
    IF ==
    THEN DROP 1 -
DUP
    ELSE DUP 4
      IF ==
      THEN DROP 1 +
DUP
      ELSE DUP 5
        IF ==
        THEN DROP 1
ALOG + DUP
        ELSE DUP 2
          IF ==
          THEN DROP
1 ALOG - DUP
          END
        END
      END
    END
  UNTIL NOT
  END CLMF 4 DROPN
≫
```

WERT

Version:	HP48
Checksum(HP28S):	entfällt
Checksum(HP48):	B0B2
Bytes:	384.5

1: —	1: —

Erläuterung:	siehe oben
Beispiel:	entfällt
benötigte UP:	WA

```
≪ CLLCD ΣDAT SIZE
LIST→ DROP 0
  DO DUP 2 DISP
ΣDAT OVER 4 PICK *
1 + GET ΣDAT 3 PICK
5 PICK * 1 + 7 PICK
+ GET 'erg' = 3
DISP DUP 2 π * →NUM
* 'w' = 1 DISP 'f'
= 5 DISP WA DUP 82
    IF ==
    THEN DROP 1 -
DUP
    ELSE DUP 72
      IF ==
      THEN DROP 1 +
DUP
      ELSE DUP 73
        IF ==
        THEN DROP 1
ALOG + DUP
        ELSE DUP 83
          IF ==
          THEN DROP
1 ALOG - DUP
          END
        END
      END
    END
  UNTIL 92 ==
  END 4 DROPN
≫
```

FDRW

Version: HP28S
Checksum(HP28S): 246103
Checksum(HP48): entfällt
Bytes: 68.5

```
≪ CLLCD 1 SWAP 1 +
COLΣ
  IFERR SCLΣ
  THEN 9 FACT NEG
DUP R→C PMIN
  END DRWΣ DGTIZ
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

FDRW

Version: HP48S
Checksum(HP28S): entfällt
Checksum(HP48): 820F
Bytes: 43.5

```
≪ ERASE 1 SWAP 1 +
COLΣ SCATTER AUTO
DRAW GRAPH
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

FDRW

Version: HP48G
Checksum(HP28S): entfällt
Checksum(HP48): 820F
Bytes: 43.5

```
≪ ERASE 1 SWAP 1 +
COLΣ SCATTER AUTO
DRAW PICTURE
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

FMIT

Version:	HP28S
Checksum(HP28S):	75371
Checksum(HP48):	entfällt
Bytes:	43.5

```
≪ CLLCD 1 SWAP 1 +
COLΣ PMAX PMIN DRWΣ
DGTIZ
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

FMIT

Version:	HP48S
Checksum(HP28S):	entfällt
Checksum(HP48):	76ED
Bytes:	46.0

```
≪ ERASE 1 SWAP 1 +
COLΣ PMAX PMIN
SCATTER DRAW GRAPH
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

FMIT

Version:	HP48G
Checksum(HP28S):	entfällt
Checksum(HP48):	76ED
Bytes:	46.0

```
≪ ERASE 1 SWAP 1 +
COLΣ PMAX PMIN
SCATTER DRAW
PICTURE
≫
```

1: —	1: —

Erläuterung: siehe oben
Beispiel: entfällt
benötigte UP: keine

8.2.9 Programme zur Netzwerk-Synthese

ABSP

Version: HP48 und HP28S
Checksum(HP28S): 2804682
Checksum(HP48): 119C
Bytes: 247.5

	2: Rest der abgespaltenen Funktion
1: gebrochen rationale Funktion in Listen-Schreibweise	1: abgespaltete Funktion

Erläuterung: Spaltet (falls möglich) einen Pol im Unendlichen ab. Falls dies nicht möglich ist, wird versucht, einen Pol in Null abzuspalten.

Beispiel: { { 2 3 4 5 }{ 1 2 3 } } ABSP ⟹
2: { { -1 -2 5 }{ 1 2 3 } }
1: { 2 0 } für

$$\frac{2s^3+3s^2+4s+5}{s^2+2s+3} =$$

$$= \frac{-s^2-2s+5}{s^2+2s+3} + 2s$$

benötigte UP: ES, KO, MUL, DI

```
≪ LIST→ PICK SIZE
OVER SIZE - 1
  IF ==
  THEN OVER 1 GET
OVER 1 GET / 0 2
→LIST DUP2 MUL 4
ROLL SWAP DI ROT 2
→LIST SWAP
  ELSE DUP KO 1 GET
-5 ALOG
    IF <
    THEN OVER KO 1
GET OVER KO 2 GET /
1 →LIST SWAP ES
DUP2 MUL 4 ROLL
SWAP DI ES SWAP 2
→LIST SWAP { 1 0 }
2 →LIST
    ELSE DROP2
    END
  END
≫
```

ZPF

Version: HP48 und HP28S
Checksum(HP28S): 1387384
Checksum(HP48): 2D4D
Bytes: 182.0

1: gebrochen rationale Funktion in Listen-Schreibweise	1: errechnete Bedingung

Erläuterung: Prüft bei einer rationalen Funktion zweiten Grades die Bedingung, ob die Funktion eine Zweipolfunktion ist.

Beispiel: {{1 2 3}{3 2 1}} ZPF $\Longrightarrow$ "4≥4"
Da diese Bedingung erfüllt ist, ist die Funktion eine Zweipolfunktion.

benötigte UP: KO, AU, DE

```
≪ 2 CF LIST→ DROP
AU KO DUP SIZE 3
  IF ==
  THEN 1 OVER SIZE
    FOR j DUP2 j
GET SWAP j GET * 3
ROLLD
    NEXT DROP2 √
ROT √ - SQ SWAP DE
EVAL →STR "≥" +
SWAP DE EVAL →STR +
  ELSE KO SWAP 2
→LIST
  END 2 SF
≫
```

RAB

Version: HP48 und HP28S
Checksum(HP28S): 689864
Checksum(HP48): 227F
Bytes: 120.5

1: gebrochen rationale Funktion in Listen-Schreibweise	2: Rest der abgespaltenen Funktion 1: Realteilminimum

Erläuterung: Spaltet das Realteilminimum einer gebrochen rationalen Funktion (in Listen-Schreibweise) ab.

Beispiel: { { 1 2 3 }{ 1 5 2 } } RAB $\Longrightarrow$
2: { { .6 0 2.2 }{ 1 5 2 } }
1: .4 für

$$.4 + \frac{.6s^2 + 2.2}{s^2 + 5s + 2}$$

benötigte UP: FDI, AU

```
≪ LIST→ ALOG ALOG
ROT ROT AU 1 OVER
SIZE
  FOR j OVER j GET
OVER j GET / 4 ROLL
MIN ROT ROT
  NEXT 2 →LIST OVER
1 →LIST { 1 } 2
→LIST FDI SWAP
≫
```

TB

Version: HP48 und HP28S
Checksum(HP28S): 191867
Checksum(HP48): 2B24
Bytes: 84.5

2: gebrochen rationale Funktion in Listen-Schreibweise 1: gerade oder ungerade Zahl	1: gerader oder ungerader Teil der gebrochen rationalen Funktion

Erläuterung: Berechnet den geraden oder den ungeraden Teil einer gebrochen rationalen Funktion (in Listen-Schreibweise).

Beispiel:
2: { { 1 2 3 }{ 3 2 1 } }
1: 1 TB ⟹
{ { 8 0 -8 0 }{ 18 0 4 0 2 } }
oder
2: { { 1 2 3 }{ 3 2 1 } }
1: 0 TB ⟹
{{6 0 12 0 6 }{ 18 0 4 0 2}}

benötigte UP: MINX, FDI, FAD, FMUL

```
≪ OVER MINX SWAP 2
  IF MOD
  THEN FDI
  ELSE FAD
  END { { 1 } { 2 }
} FMUL
≫
```

MINX

Version:	HP48 und HP28S
Checksum(HP28S):	938692
Checksum(HP48):	869A
Bytes:	150.0

1: F(x): Polynom oder gebrochen rationale Funktion in Listen-Schreibweise	1: F(-x)

Erläuterung: Dieses Programm macht aus einer Funktion F(x) (Polynom oder gebrochen rationale Funktion (in Listen-Schreibweise)) F(-x).

Beispiel: $F(x) = x^2 + 2x + 3$
Eingabe als: { 1 2 3 } MINX $\Longrightarrow$
{ 1 -2 3 }, was $F(-x) = x^2 - 2x + 3$ entspricht

benötigte UP: KO

```
≪
  ≪ LIST→ 1 2 ROT 1
+
    FOR i ROLL -1 i
^ * i
    NEXT 1 - →LIST
KO
  ≫ OVER 1 GET TYPE
5
  IF ==
  THEN OVER 1 GET
OVER EVAL ROT 2 GET
ROT EVAL 2 →LIST
  ELSE EVAL
  END
≫
```

8.2.10 Programme zur digitalen Signalverarbeitung

ZYKFAL

Version: HP48 und HP28S
Checksum(HP28S): 1103197
Checksum(HP48): 89EF
Bytes: 159.5

3: Liste mit Werten 2: Liste mit Werten 1: Länge der Listen	1: zyklische Faltung der beiden Listen

Erläuterung: Führt die zyklische Faltung zweier Listen (mit diskreten Funktionswerten) durch.
Der jeweils erste Wert der Liste steht für den Wert zum Zeitpunkt Null, dann kommt der erste Wert...

Beispiel:
3: { 1 1 0 0 0 0 0 }
2: { 1 0 1 0 1 0 1 }
1: 7 ZYKFAL ⟹
{ 2 1 1 1 1 1 1 }

benötigte UP: KO

```
≪ SWAP LIST→ 1 -
→LIST KO + SWAP
OVER SIZE OVER → z
a
  ≪ 1 SWAP
    FOR i LIST→
ROLL a →LIST 0 1 a
      FOR j OVER j
GET 4 PICK j GET *
+
      NEXT 3 ROLLD
    NEXT DROP2 z
→LIST KO
  ≫
≫
```

YKMAT

Version:	HP28S
Checksum(HP28S):	3508756
Checksum(HP48):	entfällt
Bytes:	346.0

1: —	1: —

Erläuterung: Ermöglicht die Berechnung der y(k)- und x(k)-Werte in der digitalen Signalverarbeitung, wenn dem Programm x(0), v und die A-, B-, C- und D-Matrizen der Zustandsdarstellung gegeben werden. Das Programm verfügt über eine Store-Menü-Steuerung.

Beispiel: A sei { { 0 1 }{ -1 '√2' } }, B sei { { 0 }{ 1 } }, C sei { { 0 '√2' } } und D sei { { 1 } }. Die Matrizen sind als AA, BB, CC und DD abzuspeichern und dann das Programm YKMAT aufzurufen. Das Programm fragt nun nach x(0), V, VWERT und END.
END ist der Wert für k, bis zu dem berechnet werden soll (hier 10).
x(0) ist der Anfangswert für x(k). Für ein energieloses System ist {{ 0 }{ 0 }} einzugeben.
V ist die Art der Erregung. Hier sind sprungförmige und diracförmige Erregungen simulierbar. Eine Null steht für diracförmige Erregung und eine Eins für sprungförmige Erregung (hier {0} also diracförmig).
VWERT ist ein Faktor, der bei der erregenden Gammafunktion steht (hier { 1 }).
Nun den Befehl 'CONT' ausführen. Nach einer ganzen Weile kann man die abgespeicherten Werte in den Variablen nachschauen (z.B. y(10)=2).

benötigte UP: MMUL, MAD, DE

```
≪ { STO X0 V VWERT
end } MENU HALT 0
end
  FOR k AA "X" k
→STR + STR→ MMUL BB
1 V SIZE
    FOR j V j GET
DUP NOT k NOT AND
      IF OR
      THEN VWERT j
GET
      ELSE 0
      END 1 →LIST
    NEXT V SIZE
→LIST DUP 'va' STO
MMUL MAD "'X" k 1 +
→STR + STR→ STO CC
"X" k →STR + STR→
MMUL DD va MMUL MAD
DE "'Y" k →STR +
STR→ STO
  NEXT
≫
```

YKMAT

Version:	HP48
Checksum(HP28S):	entfällt
Checksum(HP48):	CB9C
Bytes:	343.5

1: —	1: —

Erläuterung: Ermöglicht die Berechnung der y(k)- und x(k)-Werte in der digitalen Signalverarbeitung, wenn dem Programm x(0), v und die A-, B-, C- und D-Matrizen der Zustandsdarstellung gegegeben werden. Das Programm verfügt über eine Store-Menü-Steuerung.

Beispiel: A sei { { 0 1 }{ -1 '√2' } }, B sei { { 0 }{ 1 } }, C sei { { 0 '√2' } } und D sei { { 1 } }. Die Matrizen sind als AA, BB, CC und DD abzuspeichern und dann das Programm YKMAT aufzurufen. Das Programm fragt nun nach x(0), V, VWERT und END.
END ist der Wert für k, bis zu dem berechnet werden soll (hier 10).
x(0) ist der Anfangswert für x(k). Für ein energieloses System ist {{ 0 }{ 0 }} einzugeben.
V ist die Art der Erregung. Hier sind sprungförmige und diracförmige Erregungen simulierbar. Eine Null steht für diracförmige Erregung und eine Eins für sprungförmige Erregung (hier {0} also diracförmig).
VWERT ist ein Faktor, der bei der erregenden Gammafunktion steht (hier { 1 }).
Nun den Befehl 'CONT' ausführen. Nach einer ganzen Weile kann man die abgespeicherten Werte in den Variablen nachschauen (z.B. y(10)=2).

benötigte UP: MMUL, MAD, DE

```
≪ { X0 V VWERT end
} TMENU HALT 0 end
  FOR k AA "X" k
→STR + STR→ MMUL BB
1 V SIZE
    FOR j V j GET
DUP NOT k NOT AND
      IF OR
      THEN VWERT j
GET
      ELSE 0
      END 1 →LIST
    NEXT V SIZE
→LIST DUP 'va' STO
MMUL MAD "'X" k 1 +
→STR + STR→ STO CC
"X" k →STR + STR→
MMUL DD va MMUL MAD
DE "'Y" k →STR +
STR→ STO
  NEXT
≫
```

DFT

Version: HP48 und HP28S
Checksum(HP28S): 489194
Checksum(HP48): D007
Bytes: 128.0

3: Funktion 2: Name der unabh. Variable 1: Länge der DFT	1: Liste mit den DFT-Werten

```
≪ { } 0 3 PICK 1 -
  FOR e 3 DUPN DROP
/ 2 * π * i * e *
NEG EXP 5 PICK * 4
PICK 0 5 PICK 1 -
REI →NUM +
  NEXT 4 ROLLD 3
DROPN
≫
```

Erläuterung: Berechnet die DFT beliebiger Länge einer beliebigen Funktion.

Beispiel: 3: 'G0(K)'
2: 'K'
1: 8 DFT $\Longrightarrow$ { 1 1 1 1 1 1 1 1 }

benötigte UP: REI

8.2.11 Hochfrequenztechnik

Dies sind Umrechnungsprogramme für das Leitungsdiagramm zweiter Art (Smith Chart).

Unter dem „Winkel l" ist hier immer das Verhältnis 'l/λ' zu sehen, das im Smith-Chart wie ein Winkel angetragen wird.

P→R

Version: HP48
Checksum(HP28S): entfällt
Checksum(HP48): 2B81
Bytes: 40.0

```
≪ C→R DUP2 COS *
ROT ROT SIN * R→C
≫
```

1: komplexe Zahl in polaren Koordinaten	1: komplexe Zahl in kartesischen Koordinaten

Erläuterung: Führt die Umrechnung polare in rechtwinklige Koordinaten um.
Beispiel: (1,1) P→R ⟹ (.540302305868,.841470984808)
benötigte UP: keine

R→Z

Version: HP48 und HP28S
Checksum(HP28S): 19223
Checksum(HP48): A59A
Bytes: 37.5

```
≪ DUP 1 + SWAP NEG
1 + /
≫
```

1: Wert des Reflexionsfaktors	1: Wert der Impedanz

Erläuterung: Berechnet aus einem Reflexionsfaktor den Wert der Impedanz.
Beispiel: (.2,.4) R→Z ⟹ (1,1)
benötigte UP: keine

Z→R

Version: HP48 und HP28S
Checksum(HP28S): 8193
Checksum(HP48): 3572
Bytes: 32.5

```
≪ 1 - DUP 2 + /
≫
```

1: Wert der komplexen Impedanz	1: Wert des Reflexionsfaktors

Erläuterung: Berechnet aus der Impedanz den Reflexionsfaktor.
Beispiel: (1,1) Z→R ⟹ (.2,.4)
benötigte UP: keine

L→Z

Version: HP48 und HP28S
Checksum(HP28S): 356884
Checksum(HP48): 9B87
Bytes: 87.5 bzw. 91.5

```
≪ 2 INV MOD 6 FACT
* π R→D →NUM SWAP -
SWAP EVAL SWAP R→C
RCLF SWAP DEG P→R
DUP 1 + 1 ROT - /
SWAP STOF
≫
```

2: Betrags-Wert vom Mittelpunkt aus 1: Wert des Winkels l	1: Wert des komplexen Widerstandes Z

Erläuterung: Ermöglicht die Umrechnung vom Wert des Betrages und des Winkels l auf den komplexen Wert der Impedanz Z.
Beispiel: 2: .4472135955
1: .161895904412 L→Z ⟹
HP28S: (1,.9999999999999)
HP48: (.9999999999998,1)
benötigte UP: HP48: P→R
HP28S: keine

Z→L

Version: HP48 und HP28S
Checksum(HP28S): 345601
Checksum(HP48): 6625
Bytes: 92.5

```
≪ RCLF DEG SWAP 1 -
DUP 2 + / DUP ABS
→STR "'ABS(" SWAP +
STR→ π R→D →NUM ROT
ARG - 6 FACT / ROT
STOF
≫
```

1: Wert des komplexen Widerstandes	2: Betrags-Wert vom Mittelpunkt aus 1: Wert des Winkels l

Erläuterung: Ermöglicht die Umrechnung vom Wert des komplexen Widerstandes auf den Wert des Betrages vom Mittelpunkt und den Wert des Winkels l.

Beispiel: (1,1) Z→L ⟹
2: 'ABS(.4472135955)'
1: .161895904412
Das ABS steht in der zweiten Zeile nur, um Verwechslungen zu vermeiden.

benötigte UP: keine

R→S

Version: HP48 und HP28S
Checksum(HP28S): 13528
Checksum(HP48): E1F8
Bytes: 35.5

```
≪ R→Z ZMM DROP RE
≫
```

1: Wert des Reflexionsfaktors	1: Wert des Stehwellenwerts

Erläuterung: Berechnet aus einem Reflexionsfaktor den Stehwellenwert.

Beispiel: (.2,.4) R→S ⟹ 2.61803398875

benötigte UP: R→Z, ZMM

S→R

Version: HP48 und HP28S
Checksum(HP28S): 26603
Checksum(HP48): 9D7D
Bytes: 40.0

```
≪ DUP 1 - NEG SWAP
1 + / ABS
≫
```

1: Wert des Stehwellenwertes	1: Wert des Reflexionsfaktors

Erläuterung: Berechnet den Reflexionsfaktor aus dem Stehwellenwert.
Beispiel: 2.61803398875 S→R ⟹ .4472135955
benötigte UP: keine

ZMM

Version: HP48 und HP28S
Checksum(HP28S): 50269
Checksum(HP48): 9BDA
Bytes: 52.0

```
≪ Z→L DROP DUP 4
INV L→Z SWAP 0 L→Z
≫
```

	2: maximaler Widerstand
1: Wert der komplexen Impedanz	1: minimaler Widerstand

Erläuterung: Zieht gedanklich einen Kreis um den Mittelpunkt des Smith-Charts, bestimmt den Betrag der Impedanz und gibt den maximalen und den minimalen Wert der Impedanz aus.
Beispiel: (1,1) ZMM ⟹
2: (2.61803398875,0)
1: (.38196601125,0)
benötigte UP: Z→L, L→Z

8.2.12 Kompressionsprogramme

PAC

Version:	HP48 und HP28S
Checksum(HP28S):	4083838
Checksum(HP48):	94A6
Bytes:	265.0 bzw. 270.0

1: zu komprimierendes Programm	1: komprimiertes Programm

Erläuterung: Ein großes Problem der Rechner ist der doch begrenzte Speicherplatz, der sich besonders dann ungünstig auswirkt, wenn man Programme im Speicher hat, die man zwar nur selten braucht, die man aber trotzdem nicht gleich löschen möchte. Ein Ausweg stellt ein (relativ simples) Kompressionsprogramm dar. Das Programm PAC komprimiert die Programme. Sie werden dann als String dargestellt und sind nicht mehr direkt ausführbar, da sie erst dekomprimiert werden müssen. Diesen String speichert man unter dem normalen Programmnamen ab.
Achtung! Es empfiehlt sich nicht, die kleinen Unterprogramme zu komprimieren, da es zu Problemen kommt, wenn ein Hauptprogramm ein komprimiertes Unterprogramm aufruft.

Beispiel: ≪ INV + DUP2 ≫ PAC ⟹ "IAK"

benötigte UP: FA, pos, dc

```
≪ DUP TYPE 8
  IF ==
  THEN →STR 3 OVER
SIZE 2 - SUB ""
SWAP 5 5 + CHR
    DO FA DUP2 pos
3 PICK " " pos MIN
DUP 1
      IF -
      THEN ROT DUP2
1 ROT 1 - SUB dc
OVER POS
        IF DUP
        THEN CHR
SWAP
        ELSE OVER
SIZE 2 7 ^ + CHR
ROT + SWAP
        END DROP 5
ROLL SWAP + 4 ROLLD
      ELSE ROT
      END SWAP 1 +
4 ALOG SUB SWAP
OVER ""
    UNTIL ==
    END DROP2
  END
≫
```

XPAC

Version:	HP48 und HP28S
Checksum(HP28S):	1686212
Checksum(HP48):	D554
Bytes:	208.0

1: zu dekomprimierendes Programm	1: dekomprimiertes Programm

Erläuterung: Dekomprimiert die Programme, die mit PAC komprimiert wurden. XPAC läuft viel schneller als PAC, um einen schnellen Einsatz auch komprimierter Programme zu ermöglichen.

Beispiel: "IAK" XPAC $\Longrightarrow$ $\ll$ INV + DUP2 $\gg$

benötigte UP: dc, FA

```
≪ DUP TYPE 2
  IF ==
  THEN DUP "≪" 1 3
PICK SIZE
    FOR i FA " " +
OVER i DUP SUB NUM
DUP 2 7 ^
      IF ≥
      THEN 129 -
SWAP 3 PICK i 1 + 4
PICK OVER + SUB +
SWAP 2 +
      ELSE dc SWAP
GET + 1
      END
    STEP STR→ ROT
ROT DROP2
  END
≫
```

dc

Version:	HP48 und HP28S
Checksum(HP28S):	8971761
Checksum(HP48):	8F19
Bytes:	780.5

1: —	1: —

Erläuterung: Das „Wörterbuch“ für die Kompressionsprogramme. Diese Liste darf nicht verändert werden!

Beispiel: entfällt

benötigte UP: keine

{ "≪" "≫" "STO"
"END" "IF" "THEN"
"NEXT" "DO" "FOR"
"GET" "SIZE" "≠"
"==" "FA" "STEP"
"DE" "ELSE" "→NUM"
"→STR" "STR→"
"SAME" "→LIST"
"LIST→" "→" "<" ">"
"≤" "≥" "DIV" "MUL"
"AD" "DI" "{" "}"
"START" "UNTIL"
"TYPE" "SUB" "DISP"
"AU" "REPEAT"
"WHILE" "MO" "MI"
"EK" "EXCO" "MTRN"
"MAL" "MDET" "AND"
"ROLLD" "STEQ"
"RCEQ" "MMUL" "GE"
"CHR" "POS" "OR"
"IP" "FP" "SS"
"cnr" "MIDN" "MAD"
"+" "-" "*" "/" "∂"
"^" "NEG" "OVER"
"INV" "DUP" "DUP2"
"DROP" "DROP2" "FE"
"SIGN" "ROT" "SQ"
"SWAP" "PICK" "RE"
"IM" "EVAL" "ABS"
"PO" "ROLL" "π" "√"
"HALT" "MOD" "FACT"
"BEEP" "NUM" "3"
"w" }

8.2.13 Biorhythmus

BIO

Version:	HP48 und HP28S
Checksum(HP28S):	12555641
Checksum(HP48):	B49A
Bytes:	558.0

	3: erstes Datum/ Form: { TT MM JJ Wochentag }
2: erstes Datum/ Form: {TT MM JJ}	2: zweites Datum/ Form: { TT MM JJ Wochentag Tage }
1: zweites Datum/ Form: {TT MM JJ}	1: Biorhythmuswerte

Erläuterung: Stellt ein simples Programm zur Biorhythmusbestimmung dar. Berechnet zusätzlich die Wochentage der eingegebenen Daten. Das Programm erkennt alle(!) Schaltjahre, also auch, daß 1900 keines war, 2000 eines ist etc.
Jahreszahlen unter 100 werden als Daten nach 1900 interpretiert.
Die Berechnung von Wochentagen vor 1583 hat keinen Sinn, da in diesem Jahr der Gregorianische Kalender eingeführt wurde.

Beispiel:
2: { 16 1 68 }
1: { 13 5 1968 } BIO ⟹
3: { 16 1 1968 "Di" }
2: { 13 5 1968 "Mo" 118 }
1: { K 73 s S 97 s G -45 f }
Zum Ergebnis: Der 16. Jan. 1968 war ein Dienstag, der 13. Mai 1968 war ein Montag. Zwischen den beiden Daten liegen 118 Tage. Jemand, der am 16.1.68 geboren wurde, fühlte sich am 13.5.68 K(örperlich) 73 steigend, S(eelisch) 97 steigend und G(eistig) -45 fallend.

```
≪ RAD
"SaSoMoDiMiDoFr" 0
4 FACT 7 + 4 7 *
DUP2 2 + DUP2 OVER
5 DUPN DROP 6 6 +
→LIST 1 2
  START 4 ROLL
LIST→ DROP DUP LOG
2
    IF <
    THEN 1900 +
    END 4 PICK OVER
DUP 4 MOD SWAP 2
ALOG MOD
    IF NOT
    THEN 3 PICK 2
ALOG / 4 MOD OR
    END
    IF NOT
    THEN 3 5 SQ 4 +
PUT
    END 2 4 PICK
SUB cnr 4 PICK +
OVER 365 * + OVER 1
- DUP 4 / IP OVER 2
ALOG / IP - SWAP 4
5 * SQ / IP + + 6
PICK OVER 7 MOD 2 *
1 + DUP 1 + SUB
SWAP 8 ROLLD 4
→LIST ROT ROT
  NEXT DROP2 ROT 4
ROLL - SWAP OVER +
´K´ ´S´ ´G´ 5 ROLL
2 π * →NUM * 6 SQ 3
- DUP 1 ALOG -
  FOR i { } ROT +
OVER i / DUP SIN 2
```

Die Extremwerte sind -100 und +100, zwischen denen die Werte schwanken.

benötigte UP: cnr

```
ALOG * IP ROT SWAP
+ SWAP COS 0 < f s
IFTE + 4 ROLLD
-5
  STEP DROP + +
≫
```

8.2.14 Master Mind

MIND

Version:	HP48 und HP28S
Checksum(HP28S):	3191168
Checksum(HP48):	6AD7
Bytes:	282.0

1: —	1: —

Erläuterung: MasterMind, welches auch als Superhirn bekannt ist. Das Programm denkt sich 4 Zahlen (von 1 bis 7) aus, die nicht doppelt vorkommen dürfen. Der Rechner hält jeweils pro Runde an und erwartet einen Tip in Form einer Liste mit vier Zahlen. Beim nächsten Schritt gibt er die Anzahl der schwarzen und weißen Zahlen an. Eine weiße Zahl ist eine Zahl, die in diesen 4 Zahlen ebenfalls vorkommt, die aber nicht an der richtigen Stelle steht. Eine schwarze Zahl steht außerdem an der richtigen Stelle.

Beispiel: Man startet MIND ganz einfach. Nach einer kurzen Weile hält das Programm an und erwartet den ersten Tip. Man gibt vier Zahlen in einer Liste ein und führt [CONT] aus. Bei der nächsten Runde wird das Ergebnis des letzten

```
≪ 0 'T' STO { } 1 4
  START 0
    DO DROP RAND 7
* 1 + IP DUP2 POS
    UNTIL NOT
    END +
  NEXT 'μ' STO
  DO HALT 0 1 4
    FOR i OVER μ i
GET POS NOT NOT +
    NEXT 1 'T' STO+
OVER μ { -1 } MUL
AD 0 1 4
    FOR i OVER i
GET NOT +
    NEXT SWAP DROP
SWAP OVER - R→C +
  UNTIL DUP 5 GET
RE 4 ==
  END T trys { T μ
} PURGE
≫
```

Tips ausgegeben und dann erwartet der Rechner den nächsten Tip. Bei Eingabe des richtigen Tips gibt der Rechner die Anzahl der Versuche aus.
Zu suchende Kombination: {3 5 2 1}
abgegebener Tip: { 6 4 3 1 }
Der Tip enthält eine schwarze und eine weiße Zahl, d.h. der Rechner gibt folgendes aus { 6 4 3 1 (1,1) }. Die erste Zahl der komplexen Zahl ist die Anzahl der schwarzen Zahlen, die zweite die der weißen.

benötigte UP: MUL, AD

8.3 Kurzübersicht über die Programme

Polynome:

AD	addiert zwei Polynome
DI	bildet die Differenz zweier Polynome
MUL	Polynommultiplikation
DIV	Polynomdivision
ABL	Ableiten eines Polynoms
NULL	Polynom in Faktoren ersten Grades zerlegen
QUA	quadratische Gleichung lösen
QUERG	quadratische Ergänzung
PO	wandelt Listen-Schreibweise in rechnerübliche um
PI	Gegenstück zu PO
ROU	Routh-Test

Matrizen:

MMUL	Matrizenmultiplikation
MAD	Summe zweier Matrizen
MDI	Differenz zweier Matrizen
MTRN	transponieren einer Matrix
MAL	skalaren Faktor an eine Matrix ranmultiplizieren
MADJ	Adjunkte einer Matrix
MDET	Determinante einer Matrix
MIDN	Einheitsmatrix

MINV	Inverse einer Matrix
MINVV	Inverse einer Matrix
CARP	charakteristisches Polynom einer Matrix berechnen
MI	Matrix von [[ ]]-Format in {{ }}-Format verwandeln
MO	Matrix von {{ }}-Format in [[ ]]-Format verwandeln
EV	Eigenvektor zur Matrix bestimmen
ANH	Anhängen einer Matrix an eine andere
GJ	Gauß-Jordan-Algorithmus
KBM	Kern-Basis-Matrix
RANG	Rang einer Zahlenmatrix
MODAL	Modalmatrix
DEFI	Definitheit einer Matrix bestimmen
CHOL	Cholesky-Zerlegung
QR	QR-Zerlegung

Gebrochen rationale Funktionen:

FAD	Addieren zweier gebrochen rationaler Funktionen
FDI	Differenz aus zwei gebrochen rationalen Funktionen
FMUL	Produkt zweier gebrochen rationaler Funktionen
KUE	Kürzen aus einer gebrochen rationalen Funktion
ABSP	Abspalten eines Pols in Null oder Unendlich
TB	(un-)gerader Teil berechnen
FABL	Ableiten einer gebrochen rationalen Funktion
PART	Partialbruchzerlegung

Programme aus der Elektrotechnik:

PSI	Übergangsmatrix einer Matrix bestimmen
HVS	H(s) (Übertragungsfunktion) bestimmen
RAB	Widerstand abspalten
ZYKFAL	zyklisches Falten einer Funktion
LINA	Netzwerkanalyseprogramm
DFT	diskrete Fourier-Transformierte
GLZ	Gruppenlaufzeit
DB	Zahl in dB umrechnen
DB→	Zahl von dB umrechnen
G1	Sprungfunktion
G0	Impulsfunktion
LOGS	Abstand für Omega auf log. Skala bestimmen
WLOG	Omega auf log. Skala berechnen

Anderweitige Programme:

FA	(Platzhalter für) Beschleunigungsprogramm
SYS	Installationsprogramm für das Beschleunigungsprogramm (HP28S)
TI	(nur HP28S) Uhrzeit

EXCO	Zusammenfassungsprogramm
EX	wie EXCO, setzt aber Werte in Variablen ein und pfeift am Schluß
EVV	wie der Befehl →NUM - allerdings für Polynome, Matrizen etc. in Listenschreibweise
DE	Wurzel-Bruch-Suchprogramm
ABSCH	abschnittsweise definierbare Funktion
TAY	Taylorentwicklung
OK	(HP28S) Ortskurven malen
LG	logarithmisches zeichnen
BOOL	boolesche Ausdrücke miteinander vergleichen
INPOL	Interpolation
rot	Rotation eines Feldes bestimmen
EXY	Erwartungswerte bestimmen
LD	dualer Logarithmus
REI	Wert einer Reihe
MINX	macht aus F(x) F(-x)
PHY	physikalische Konstanten
MIND	Master Mind
BIO	Biorhythmus
CD	(HP28S) Wechseln um ein Directory nach oben
LOE	löscht alle einbuchstabigen Variablen und Programme
mul	Multiplikation mit bel. Nachkommastellenanzahl
div	Division mit bel. Nachkommastellenanzahl
ad	Addition mit bel. Nachkommastellenanzahl
di	Differenz mit bel. Nachkommastellenanzahl
PAC	Kompressionsprogramm
XPAC	Dekompressionsprogramm
CHK	(HP28S) Prüfsummenprogramm
OFF	(HP28S) softwaremäßiges Ausschalten des Rechners

9 Anhang

9.1 Tabellen

TYPE	Bezeichnung	Beispiel	Bemerkung
0	Integer- oder REAL-Zahl	1.51	
1	komplexe Zahl	(1,2)	
2	Zeichenkette	"HALLO"	
3	Matrix/Vektor mit reellen Zahlen	[1 2 3]	
4	Matrix/Vektor mit komplexen Zahlen	[[(1,2) (3,2)]]	
5	Liste	{ "HALLO" 3 }	
6	Name	'NAME'	
7	lokaler Name	→ NAME	
8	Programm	≪ DUP ≫	
9	algebraischer Ausdruck	'2*A'	
10	Binärzahl	#Ah	
11	graphisches Objekt	Graphic ...	(nur HP48)
12	benamtes Objekt	a: 2	(nur HP48)
13	Einheiten-Objekt	55_cm	(nur HP48)
14	XLIB Library	XLIB 645 2	(nur HP48)
15	Verzeichnis	DIR Name	(nur HP48)
16	Library	Library 645: ...	(nur HP48)
17	Backup-Objekt	Backup HOMEDIR	(nur HP48)
18	implementierte Funktion	COS	(nur HP48)
19	implementierter Befehl	ROT	(nur HP48)
20	Adresse	<28EFh>	(nur HP48)
21	lange reelle Zahl	entfällt	(nur HP48)
22	lange komplexe Zahl	entfällt	(nur HP48)
23	angehängtes Feld	entfällt	(nur HP48)
24	Character	entfällt	(nur HP48)
25	Code	entfällt	(nur HP48)
26	Library-Daten	entfällt	(nur HP48)
27	Externes	entfällt	(nur HP48)

Tabelle 9–1: *TYPE-Zuordnung von Objekten*

Zahl	Operation in UPN	Ersparnis/Bytes
0.0000000001	-9 ALOG	5.5
0.000000001	-8 ALOG	5.5
⋮	⋮	⋮
0.1	-1 ALOG	5.5
0.05	4 5 * INV	.5
⋮	⋮	⋮
0.5	2 INV	5.5
⋮	⋮	⋮
10	1 ALOG	5.5
100	2 ALOG	5.5
⋮	⋮	⋮
1000000000	9 ALOG	5.5
16	4 SQ	5.5
⋮	⋮	⋮
81	9 SQ	5.5
20	4 5 *	3
⋮	⋮	⋮
1096	7 EXP IP	3
⋮	⋮	⋮
24	4 FACT	5.5
⋮	⋮	⋮
362880	9 FACT	5.5
27	4 FACT 3 +	.5
⋮	⋮	⋮
39	6 SQ 3 +	.5
⋮	⋮	⋮

Tabelle 9–2: *Darstellung von REAL-Zahlen mit Hilfe von Integer-Zahlen*

Struktur	Beispiel	Speicherverbrauch
Integer-Zahl	2	2.5 Bytes
REAL-Zahl	1.5	10.5 Bytes
komplexe Zahl	(1,2)	18.5 Bytes
String	""	2.5 Bytes + 1 Byte pro Buchstabe
Vektor	[0]	12.5 + 8 Bytes pro Zahl
Matrix	[[0]]	15 Bytes + 8 Bytes pro Zahl
Vektor mit komplexen Zahlen	[(1,1)]	12.5 Bytes + 16 Bytes pro komplexe Zahl
Matrix mit komplexen Zahlen	[[(1,1)]]	15 Bytes + 16 Bytes pro komplexe Zahl
Liste	{ }	5 Bytes + der Bedarf der Objekte
Name in '-Zeichen	'X'	9.5 Bytes + 1 Byte pro Buchstabe
Name ohne '-Zeichen	X	4.5 Bytes + 1 Byte pro Buchstabe
Programm	≪ ≫	15.5 Bytes + der Verbrauch an Anweisungen
algebraischer Ausdruck	'2*X'	Summe aus den Einzelteilen
Binärzahl	#Ah	13 Bytes
Befehl	DUP	2.5 Bytes
Aufruf eines Unterprogramms	QUA	3.5 Bytes + 1 Byte pro Buchstabe

Tabelle 9–3: *Speicherverbrauch einzelner Objekte*

Struktur	Speicherverbrauch
IF *Befehl* THEN *Befehl* END	12.5 Bytes
IF *Befehl* THEN *Befehl* ELSE *Befehl* END	17.5 Bytes
START NEXT	5 Bytes
FOR E NEXT	9.5 Bytes + 1 Byte pro weiteren Buchstaben der Variable
START X STEP	5 Bytes + Verbrauch von X
FOR E X STEP	9.5 Bytes + 1 Byte pro weiteren Buchstaben der Variable + Verbrauch für X
IFERR *Befehl* THEN *Befehl* END	12.5 Bytes
IFERR *Befehl* THEN *Befehl* ELSE *Befehl* END	17.5 Bytes
DO UNTIL END	7.5 Bytes
WHILE REPEAT END	12.5 Bytes

Tabelle 9–4: *Speicherverbrauch einzelner Strukturen mit jeweiligem Speicherverbrauch*

Falls bei einer Struktur der Begriff *Befehl* steht, so ist im angegebenen Speicherverbrauch je eine Anweisung hierfür eingerechnet.

CHR	Zeichen	CHR	Zeichen	CHR	Zeichen	CHR	Zeichen
0	NUL	32	Space	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	–	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Tabelle 9–5: *Darstellung der Zeichen nach dem ASCII-Code*

Achtung! Abweichungen vom eigentlichen ASCII-Code sind die Zeichen von 0 bis 31, sowie das Zeichen 127. Die Zeichen von 0 bis 31 werden als kleines Quadrat dargestellt und das Zeichen 127 ist durch ein Rechteck ersetzt worden, bei dem abwechselnd ein Punkt seiner Darstellungsmatrix an und aus ist.

Die Zeichen ab Character 128 sind nicht genormt und wurden von HP für den internen Befehlssatz genutzt. Dieser differiert auch zwischen den einzelnen Modellen.

Dies ist die Aufstellung der Tastatur-Code-Nummern des HP48:

11	12	13	14	15	16
21	22	23	24	25	26
31	32	33	34	35	36
41	42	43	44	45	46
51		52	53	54	55
61	62	63	64	65	
71	72	73	74	75	
81	82	83	84	85	
91	92	93	94	95	

Das Schema ist sehr leicht zu durchschauen. Beim HP48 ist die Code-Nummer einer Taste immer als zweistelllige Zahl XY angebbar, bei der X die Zeile (von oben her) ist und Y die Spalte in der Zeile (von links her) ist.

Der HP28 hingegen gibt seinen Tasten Namen, die der jeweiligen Aufschrift entsprechen (also „ENTER“ für die ENTER-Taste).

vorhanden	Ziel	Umsetzung in RPL
3: A 2: B 1: C	3: C 2: B 1: A	SWAP ROT
3: A 2: B 1: C	3: C 2: A 1: B	ROT ROT
3: 2: A 1: B	3: B 2: A 1: B	SWAP OVER
3: A 2: B 1: C	3: B 2: A 1: C	ROT SWAP
4: 3: 2: A 1: B	4: A 3: B 2: Größe(A) 1: Größe(B)	OVER SIZE OVER SIZE
5: 4: 3: A 2: B 1: C	5: B 4: A 3: C 2: A 1: B	3 DUPN DROP
6: 5: 4: A 3: B 2: C 1: D	6: A 5: B 4: C 3: D 2: A 1: B	4 DUPN DROP2

Tabelle 9–6: *Optimale Umformungen von Objekten des Stacks*

9.2 Literaturverzeichnis

[1] Benutzerhandbuch HP28S,
Hewlett-Packard, Bestellnummer 00028-90072

[2] Serie HP48G Benutzerhandbuch
Hewlett-Packard, Bestellnummer 00048-90127

[3] Forth 83
von R. Zech, Franzis', München, 1987

[4] Skript: Einführung in die Programmierung (FORTRAN)
von M. Abel, 1989, Friedrich-Alexander-Universität Erlangen-Nürnberg

[5] Skript zur Vorlesung: Informatik für Elektroingenieure
von Prof. Niemann, 1990, Friedrich-Alexander-Universität Erlangen-Nürnberg

[6] A Detailed Description of the Hewlett-Packard Calculator Products
von Craig Finseth, 1991, St Paul MN 55104-2437 USA

9.3 Stichwortverzeichnis

Neues aus erster Hand

Physikalische Abmessungen und Kodierungen der Magnetkarten werden erläutert, vorhandene oder sich in Entwicklung befindliche Hard- und Software vorgestellt.

1. Auflage 1995
Gebunden, 117 Seiten
mit Diskette
DM 68,–/öS 530,–/sfr 68,–
ISBN 3-88229-027-7

Schwierigkeiten mit Akronymen in der Netzwerk-Welt? Ein Experte gibt zuverlässig und knapp Auskunft beim Abkürzungs-wirrwarr.

1. Auflage 1995
Broschur, ca. 340 Seiten
DM 48,–/öS 374,–/sfr 48,–
ISBN 3-88229-032-3

Setzen Sie die Konzepte der objektorientierten Programmierung beim Erstellen kommerzieller Software um.

1. Auflage 1994
Gebunden, 288 Seiten
DM 78,–/öS 608,–/sfr 78,–
ISBN 3-88229-040-4

Die X.25-Paketvermittlungstechnik und Folgetechnologien der Hochgeschwindigkeitsdatenkommunikation, wie Frame Relaying und ATM werden vorgestellt.

1. Auflage 1995
Gebunden, ca. 400 Seiten
ca. DM 100,–/öS 780,–/sfr100,–
ISBN 3-88229-048-x

Im Buch- und Fachhandel erhältlich

C.4.2/94 1/2q

Verlag
Heinz Heise
GmbH & Co KG
Postfach 610407
D-30604 Hannover

Netzwerke

Alle sieben Ebenen in Theorie und Praxis vorgestellt. Für Informatikstudenten, Techniker und Anwender, deren Computer miteinander kommunizieren müssen.

1. Auflage 1994
Broschur, 176 Seiten
DM 28,–/öS 218,–/sfr 28,–
ISBN 3-88229-045-5

Kompakt alles über Transmission, Control Protocol/Internet Protocol (TCP/IP): Geschichte und Komponenten, Netzwerke und Protokolle...

1. Auflage 1994
Broschur, 111 Seiten
DM 28,–/öS 218,–/sfr 28,–
ISBN 3-88229-042-0

Hardwareperspektiven und -architektur, Virtual Storage, Session und Links, VTAM-Grundlagen und APPN prägnant und kompetent beschrieben.

1. Auflage 1994
Broschur, 144 Seiten
DM 28,–/öS 218,–/sfr 28,–
ISBN 3-88229-043-9

Binden Sie ohne Probleme die ungleichen Umgebungen TCP/IP und SNA aneinander an. Ein Glossar erläutert die beiden Begriffswelten.

1. Auflage 1994
Broschur, 224 Seiten
DM 28,–/öS 218,–/sfr 28,–
ISBN 3-88229-044-7

Im Buch- und Fachhandel erhältlich

C.4.1/94 1/2q

Verlag
Heinz Heise
GmbH & Co KG
Postfach 610407
D-30604 Hannover

Der ideale Einstieg in die Programmierung der Taschenrechner von Hewlett-Packard vom HP 28 bis zum HP 48.

Ralf Thoma bietet eine Einführung in die Programmiersprache RPL, die auch Anfängern nach kurzer Zeit die optimale Ausnutzung ihrer Rechner ermöglicht. Der fortgeschrittene Nutzer findet neben Insidertips ausgereifte Programme. Folgende Schwerpunkte werden gesetzt:

- Eingabe und Fehlersuche;
- Polynome;
- Netzwerkanalyse;
- digitale Signalverarbeitung;
- Hochfrequenztechnik.

Die vorgestellten Programme werden auf der beigefügten Diskette mitgeliefert.

Zur thematischen Ergänzung vom gleichen Verlag: *Rau/Gießelink, Programmsammlung HP 28S/48*

ISBN 3-88229-052-8