Rechenzentrum der Universität Mannheim

S C A L L O P

1. Version, implementiert auf der

Siemens 4004/45

Nr. 17 (1971)

INHALT

1. Einleitung

Die Programmiersprache SCALLOP wurde 1967 von G. Autry und M. Engeli speziell für Probleme der nichtnumerischen Datenverarbeitung entwickelt und am Rechenzentrum der Universität Texas/Austin auf einer CDC 6600 implementiert. Später folgten weitere Implementierungen auf verschiedenen CDC-Rechnern. 1970 wurde SCALLOP am Rechenzentrum der Universität Mannheim der internen Struktur der Siemens 4004/45 angepaßt und auf dieser Maschine implementiert.+ Dabei wurde die Sprache in einigen Punkten stark modifiziert. Die hier vorgelegte Version ist angesichts der geplanten Erweiterungen (vgl. Abschnitt 2.1) als vorläufig zu betrachten.

2. Allgemeine Eigenschaften

2.1 Wesentliche Merkmale der Sprache

Mit SCALLOP sollte eine Sprache geschaffen werden, die speziell Probleme der nicht-numerischen Datenverarbeitung effizient zu bearbeiten gestattet und die zur gleichen Zeit leicht programmierbar, lesbar und erlernbar ist. Aus diesen Forderungen ergab sich das Konzept einer Sprache, die in elementaren Funktionen wie im Ansprechen von Operanden und in den Operationen maschinennah in den komplexen Funktionen und in der Programmverwaltung aber das Niveau höherer Programmiersprachen erreicht.

+ Das Projekt wurde von der Deutschen Forschungsgemeinschaft finanziell unterstützt.
Beteiligt waren: F. Faulbaum, G. Glau, C. Lämmerhold, W. Marschall und N. Schönfelder

Zu den maschinennahen Eigenschaften gehören insbesondere:

- explizite Adressrechnung: SCALLOP gestattet wie die ASSEMBLER-Sprache den Bezug auf Adressen von Speicherbereichen. Adressen werden dabei direkt in Ausdrücken verwendet.
- indirekte Adressierung
- Gerade die explizite Adressrechnung und die indirekte Adressierung bieten die Möglichkeit der Überlagerung beliebiger Daten- und Listenstrukturen über der Speicherstruktur und genügen so wesentlichen Erfordernissen der nicht-numerischen Datenverarbeitung.

- Möglichkeit bitweiser Verknüpfung von Bitmustern durch Bit-für Bit Operationen.

- Verarbeitung von Zeichenketten (Strings): SCALLOP gestattet die Einführung von Zeichenkonstanten, die in Ausdrücken verwendet werden können. Damit ergibt sich die Möglichkeit, direkte Bit-für Bit-Manipulationen an Zeichenketten durchzuführen, sowie beliebige Zeichenketten mittels indirekter Adressierung zu verknüpfen und abzuarbeiten.

- die Möglichkeit, in Abhängigkeit von arithmetischen Überläufen den Programmablauf zu verändern, sowie die Möglichkeit, den Divisionsrest abzufragen.

- Möglichkeit der direkten Verwaltung des freien Kernspeicherbereichs innerhalb von Unterprogrammen ohne vorherige Deklaration unter Zuhilfenahme bestimmter Standardprozeduren (vgl. unten:Standardprozeduren zur Speicherverwaltung).

Problemorientierte Eigenschaften von SCALLOP:

- Anweisungen: Auch die Anweisungen sind von ihrer Struktur her ähnlich denen von ALGOL 60. Neu hinzugekommen ist eine zusätzliche "Fallanweisung" (CASE-Statement). Die Fallanweisung bietet die Möglichkeit, in Abhängigkeit vom Wert eines Ausdruckes eine bestimmte Anweisung aus einer Anweisungsliste auszuwerten.
  Die ALGOL-Schleifenanweisung ist bei SCALLOP auf eine Wiederholungsanweisung ohne Laufvariable reduziert.

- Prozeduren: Das Prozedurkonzept entspricht im wesentlichen den Funktionsprozeduren von ALGOL 60. Rekursive Aufrufe sind gestattet. Neu ist, daß Prozeduren durch ihre Verwendung in beliebigen Ausdrücken aufgerufen werden, sodaß von der Verwendung her kein wesentlicher Unterschied mehr zwischen Prozeduren und Variablen besteht.

Zu den obigen Kennzeichen der Sprache treten noch die folgenden Merkmale des Compilers:

- Strikte von Links- nach Rechtsabarbeitung von Ausdrücken;

- Reine Single-Pass Compilation d.h. Syntax-Analyse und Code-Erzeugung (Ausgabe in ASSEMBLER) geschehen in einem Durchgang.

Geplante Erweiterungen:

- Obige Charakteristika machen SCALLOP in besonderem Maße geeignet als Sprache der syntaktischen Analyse sowie als Supervisor-Sprache innerhalb von Compilern.
  Weitere Planungen sind daher darauf ausgerichtet, SCALLOP mit dem String-Processor G PS P (vgl. G PSP - User Manual) zu verbinden, der ein besonders geeignetes Werkzeug gerade für die Codeerzeugung darstellt.

Die Verbindung beider Sprachen soll so geschehen, daß sowohl maximale Maschinenunabhängigkeit als auch maximale Unabhängigkeit von der zu übersetzenden Eingangssprache gewährleistet ist.

- Deklarationen von Konstanten.

- Ausdehnung der expliziten Adressrechnung auch auf Prozeduradressen.

- Übersetzung von SCALLOP in Maschinencode. Gegenwärtig ist die Ausgangssprache noch ASSEMBLER.

- Linkage Options zur Erleichterung des Anschlusses an unabhängig übersetzte Programme in Form einer programmierbaren Herstellung von Kompatibilität in der Speicherverwaltung.

- Standardprozeduren und -funktionen

  i. zur Ein- Ausgabe;
  ii. zum Übertragen, Löschen und Vergleichen von Bytefolgen, Entfernung von Blanks aus Bytefolgen und Umsetzung von Bytes in Worte.
  iii. zur Konvertierung von Strings in Dualzahlen und von Bytefolgen in Hexadezimalzahlen, sowie zur Umwandlung von numerischen Strings in Dualzahlen.

  vi. zur programmierbaren Alarmbehandlung.

2.2 Speicherstruktur

2.2.1 Der Arbeitsspeicher der Siemens 4004/45

Seien B und C zwei Bitfolgen, dann soll unter D = BC diejenige Folge D von Bits verstanden werden, die durch die Operation der Aneinanderfügung (concatenation) von C an B entsteht.
Wird eine Bitfolge B n-mal an sich selbst angefügt: $\underbrace{BBB...B}_{n\text{-mal}}$

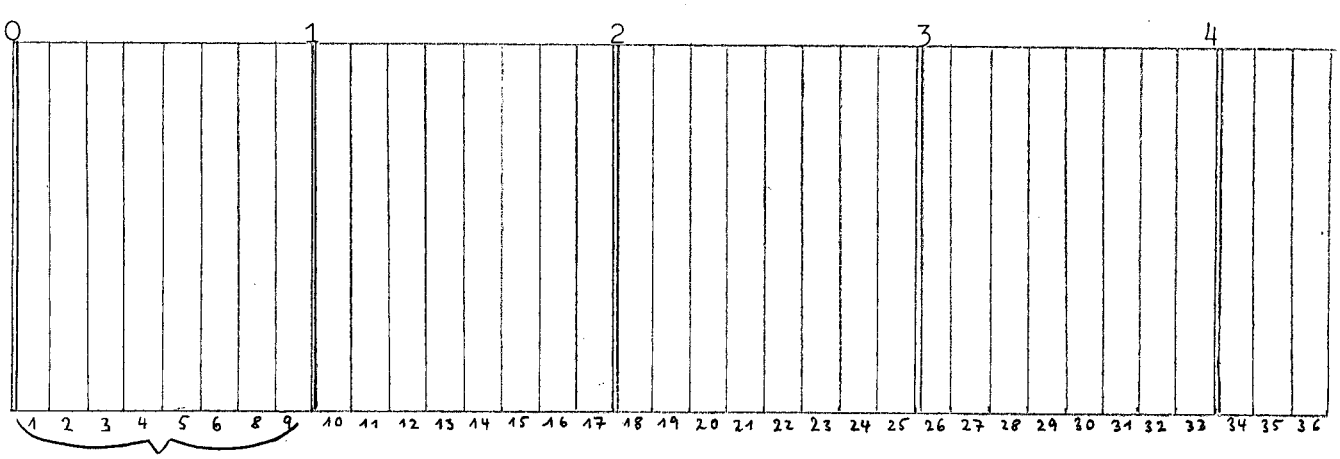
so schreiben wir für die resultierende Bitfolge nB. Unter OB= ε wird die leere Folge verstanden. Ist nun K eine Bitfolge mit einer fest vorgegebenen Anzahl K von Elementen, so lassen sich die Arbeitsspeicheradressen A aufgrund der folgenden primitiv rekursiven Funktionen berechnen:

$$A\,(0K) \equiv 0$$

$$A\,(nK) = A\,((n-1)K) + 1$$

D.h. ausgehend von der Arbeitsspeicheradresse einer 0-mal wiederholten Bitfolge K, bzw. der leeren Bitfolge, erhält man die nächstfolgende dadurch, daß man an diejenige Bitfolge, der die vorangegangene Speicheradresse zugeordnet wurde, die Bitfolge k anfügt.
K heißt kleinste adressierbare Einheit, Speicherstelle oder Speicherplatz.
Der Speicher läßt sich mithin auffassen als eine Folge von kleinsten adressierbaren Einheiten: Bei der Siemens 4004/45 ist die kleinste adressierbare Einheit 1 Byte, wobei unter einem Byte eine Folge von 8 Bit verstanden wird.

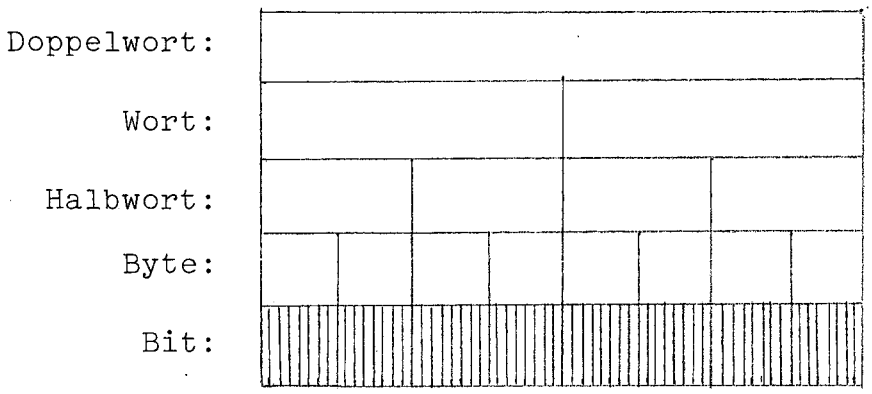
eicher-
ressen:



kleinste adressierbare Einheit

Struktur eines Arbeitsspeichers mit einem Byte=8 Bits als kleinster adressierbarer Einheit

Versteht man unter einem Byte eine Folge von 8 Bit, so versteht man unter einem Halbwort eine Folge von 2 Byte oder 16 Bits, unter einem Wort eine Folge von 2 Halbworten, 4 Bytes oder 32 Bits und unter einem Doppelwort eine Folge von 2 Worten, 8 Bytes oder 64 Bits:



Entsprechend der Definition von Byte, Halbwort, Wort und Doppelwort sowie der Festlegung der kleinsten adressierbaren Einheit können die folgenden Adreßklassen unterschieden werden:

1. Byteadressen: durch 1 teilbare Adressen;
2. Halbwortadressen: durch 2 teilbare Adressen;
3. Wortadressen: durch 4 teilbare Adressen;
4. Doppelwortadressen: durch 8 teilbare Adressen.

Jede dieser Adressen definiert eine Bytegrenze, Halbwortgrenze, Wortgrenze oder Doppelwortgrenze, d.h. eine Grenze zwischen 2 Bytes, 2 Halbworten, 2 Worten oder 2 Doppelworten. -
Unter einem Arbeitsspeicherbereich verstehen wir einen an einer festen Byte-, Halbwort-, Wort-, oder Doppelwortadresse beginnende Folge von Bytes, Halbworten, Worten oder Doppelworten. Entsprechend sprechen wir von Byte-, Halbwort-, Wort-, oder Doppelwortbereichen.

2.2.2 Die von SCALLOP induzierte Speicherstruktur

2.2.2.1 Operanden und Operationen

Operanden entsprechen maschinenintern Inhalten von Arbeitsspeicherworten. Speicherworte heißen auch Variablen.

Jede Variable kann im Speicher eindeutig durch ihre Adresse identifiziert werden. Wortadressen werden durch Namen von Variablen bezeichnet. In einer speziellen Deklaration, der NEW-Deklaration, hat der Benutzer die Möglichkeit, Namen für Wortadressen im Arbeitsspeicher explizit zu vergeben.

Den SCALLOP-Operationen entsprechen in der vorliegenden SCALLOP-Version maschinenintern Folgen von Wortbefehlen der Siemens 4004/45 und zwar sowohl Bit-für-Bitbefehle (bitweise verknüpfung von Bitmustern), als auch Festpunktbefehle (arithmetische Operationen). -

Operanden werden im Maschinencode der Siemens 4004/45 fast ausschließlich über ihre Adressen angesprochen (direkte Adressierung), d.h. im Operandenteil der Maschinenbefehle stehen die Adressen der Operanden. Aufgrund dieses Sachverhalts sowie der Zuordnung von SCALLOP-Operationen zu Folgen von Maschinenbefehlen und von Variablennamen zu Adressen ergibt sich, daß dadurch, daß ein Variablenname in einem SCALLOP-Ausdruck erscheint, der Operand direkt angesprochen wird, bzw. der Inhalt durch die dem Namen zugeordnete Adresse bezeichneten Arbeitsspeicherwortes. Konstanten können auch direkt in SCALLOP-Ausdrücken erscheinen. In diesem Fall legt das Übersetzungsprogramm die Konstanten in bestimmte Arbeitsspeicherworte, die dann maschinenintern über Adressen angesprochen werden. -

Operanden können in SCALLOP auch Adressen von Operanden sein. Maschinenintern bedeutet dies, daß als Inhalte von Speicherworten Adressen von Speicherworten entsprechen können.

Tritt die Zeichenkombination'@ Variablenname' auf, so macht der Compiler die Adresse eines Speicherwortes zu einem Operanden. In der vorliegenden Version lassen sich nur Adressen von durch Namen in der NEW-Deklaration bezeichneten Arbeitsspeicherworten als Operanden ansprechen.

Neben der Einführung von Adreßoperanden gibt es die Möglichkeit, sich über Adressen auf den Inhalt von Arbeitsspeicherworten zu beziehen. Diese Möglichkeit wird auch als indirekte Adressierung bezeichnet. -

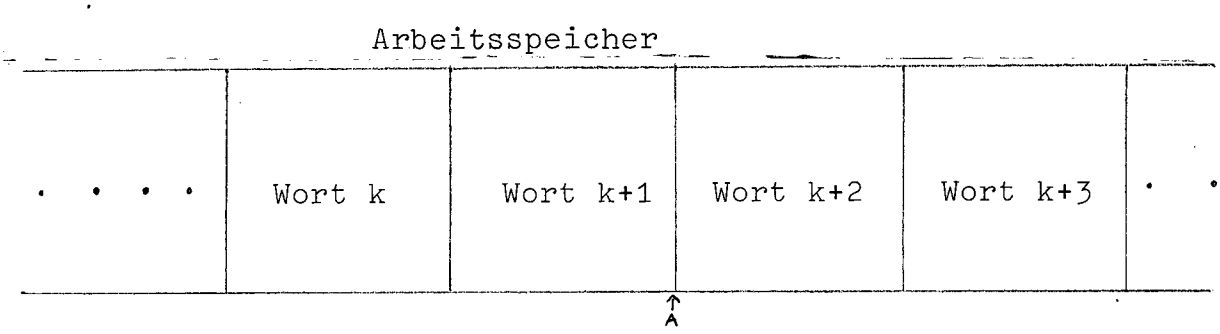
Zusammenfassend läßt sich sagen, daß man bei SCALLOP drei Ebenen unterscheiden kann, auf denen mit Operanden operiert wird:

- Operationen mit Inhalten von Speicherworten;
- Operationen mit Adressen von Arbeitsspeicherworten;
- Operationen mit über Adressen angesprochenen Inhalten von Arbeitsspeicherworten. Diese Ebene verknüpft die beiden anderen, indem sie den Adressierungsvorgang wieder rückgängig macht.

2.2.2.2 Operationen mit Arbeitsspeicherinhalten

Wie bereits ausgeführt wurde, hat der Benutzer die Möglichkeit innerhalb einer speziellen Deklaration, der NEW-Deklaration, Namen für Arbeitsspeicherworte zu vergeben, die auch als NEW-Variable bezeichnet werden.

Beispiel:



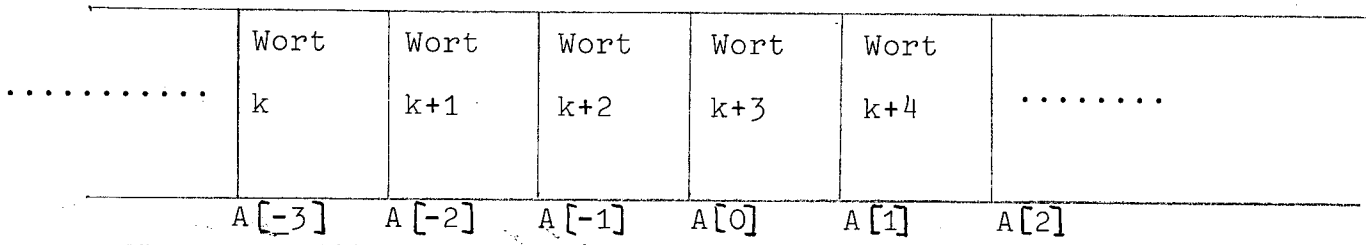
Erklärung:

A stellt den Namen für die Adresse des K+2-ten Wortes des Arbeitsspeichers dar. In einer SCALLOP-Operation würde mit A der Inhalt dieses Wortes angesprochen werden.
Namen von NEW-Variablen können auch indiziert sein (vgl. Abschn. 3.4.2.1 und Abschnitt 3.7.2.1).

Ein indizierter Name bezeichnet die Arbeitsspeicheradresse eines Wortes, das im durch die Adressen der kleinsten adressierbaren Einheiten geordneten Arbeitsspeicher um soviele Worte vor oder hinter dem durch den Namen der NEW-Variablen bezeichneten Arbeitsspeicherwort liegt, wie der Wert des Index angibt. Positive Indexwerte ermöglichen den Bezug auf Worte, die der NEW-Variablen folgen und negative Indexwerte den Bezug auf Worte, die der NEW-Variablen vorangehen.
In dieser Weise können alle Arbeitsspeicherworte, ausgehend von einer NEW-Variablen, angesprochen werden.
Ein Indexwert von 0 bezeichnet dabei die Adresse der NEW-Variablen selbst.

Beispiel:

Arbeitsspeicher



Aus obigen Ausführungen geht hervor, daß auf der Ebene der Operationen mit Arbeitsspeicherinhalten der Bytestruktur des Arbeitsspeichers der Siemens 4004/45 eine Wortstruktur überlagert wird.

2.2.2.3 Operationen mit Adressen von Arbeitsspeicherworten (Adreßrechnung)

Als Adressoperanden sind auch Adressen von Arbeitsspeicherworten zugelassen, die durch indizierte Variablennamen bezeichnet werden: @ A [I] (vgl. Abschnitt 3.4.3)
Durch Verwendung von Adressoperanden in Ausdrücken, kann auf mittels dieser bitweise und arithmetisch operiert werden.

Dabei ist zu bemerken, daß ein Adreßoperand in Ausdrücken zunächst wie jeder andere Operand behandelt wird, also als Bitmuster bei Bit-für-Bitoperationen oder als Festpunktzahl bei arithmetischen Operationen. Der Wert eines Ausdrucks, in dem ein Adreßoperand auftritt, ist zunächst wie der Wert eines jeden anderen Ausdrucks nichts weiter als ein Bitmuster, wenn er nicht durch entsprechende Operationen anders interpretiert wird.
Neben den durch @A[I] angesprochenen Adreßoperanden werden auch die Werte einfacher Ausdrücke als Adressen interpretiert, vorausgesetzt, die einfachen Ausdrücke treten in berechneten Variablen auf, die in SCALLOP indirekt adressierte Operanden bezeichnen. Einfache Ausdrücke bestehen nur aus positiven oder negativen Zahlen oder Namen für NEW-Variablen, Operationssymbolen für Addition oder Subtraktion oder in einer Wertzuweisung des Werts von einfachen Ausdrücken zu Variablen.
Es ist dabei wichtig, zu beachten, daß der Wert eines einfachen Ausdrucks in einer berechneten Variablen als Byteadresse interpretiert wird. Indem nun der Wert eines Ausdrucks, in dem ein Adreßoperand angesprochen wird, einer NEW-Variablen zugewiesen werden kann und diese wiederum ein einfacher Ausdruck ist, kann der resultierende Wert eines Ausdrucks mit Adreßoperanden als Byteadresse interpretiert werden.
Dies bedingt, daß, wenn der Wert eines Ausdrucks, in dem zu einem Adreßoperanden eine Zahl addiert bzw. von ihm eine Zahl abgezogen wird, als Byteadresse interpretiert wird, diese Zahl als Anzahl von Bytes interpretiert werden kann.
Es ist also möglich, durch Addition von Zahlen zu Adreßoperanden bzw. durch Subtraktion von Zahlen von Adreßoperanden, jedes Byte im Arbeitsspeicher zu adressieren.
Aufgrund der auf der Ebene der Adreßrechnung möglichen Adressierung von Bytes kann man sagen, daß dort die Bytestruktur der Siemens 4004/45 realisiert ist, allerdings mit der Einschränkung, daß man sich nur auf Adressen von Arbeitsspeicherbytes beziehen kann, die, ausgehend von einem Arbeitsspeicherwort, angesprochen werden können.

Beispiele:

@ A[1]+2	=	Adresse des dritten Bytes des durch A[1] definierten Wortes
@ A+9	=	Adresse des zweiten Bytes des durch A[2] definierten Wortes
@ A	=	Adresse des ersten Bytes von A.

2.2.2.4 Indirekte Adressierung

Durch indirekte Adressierung ist es in SCALLOP möglich, sich auf Inhalte von Arbeitsspeicheradressen zu beziehen, die als Werte eines einfachen Ausdrucks erst berechnet werden.
Die indirekte Adressierung wird in SCALLOP möglich durch die Einführung sogen. berechneter Variablen, d.h. von Variablen, deren Adresse erst berechnet wird. Sie werden durch Subscripts bezeichnet, die als [A] geschrieben werden. Der Wert von A wird zwar als Byteadresse interpretiert, diese muß aber, da durch A der Inhalt eines Wortes angesprochen wird, zugleich eine Wortadresse sein, d.h. durch 4 teilbar sein.
Die indirekte Adressierung macht so die Bytestruktur der Adreßrechnung wieder rückgängig und verbindet so die Ebene der Arbeitsspeicherinhalte mit der der Arbeitsspeicheradressen.

2.3 Datenrepräsentation

2.3.1 Grundlegendes

Bitmuster:

Unter einem Bit verstanden wir eine Variable mit zwei Zuständen, bezeichnet etwa als 0 und 1.
Sei eine Folge von Bits gegeben, dann wollen wir die dieser Folge zugeordnete Folge von Bitzuständen ein Binärmuster nennen.

Beispiel:

Bitfolge:	B_1	B_2	B_3	B_4	B_5
Bitmuster:	0	0	1	1	0

Ordnen wir den Bits einer Bitfolge von rechts nach links natürliche Zahlen von 0-(K-1) zu, wenn K die Länge der Bitfolge darstellt, so nennen wir diese Zahlen auch Positionsindizes. Die natürliche Zahl 2^i, wobei i=Positionsindex, heißt auch Wertigkeit des betreffenden Bits.

Beispiel:

Bits:	B_1	B_2	B_3	B_4
Positionsindex:	3	2	1	0
Wertigkeit:	2^3	2^2	2^1	2^0

Entsprechend spricht man auch von den höher- oder niederwertigen Bits.

Zahlen:

Natürliche Zahlen lassen sich bekannterweise stets schreiben in der Form $Z=\sum_{0}^{n} a_i B^i$, wobei $B \geq 2$ die Basis des betreffenden Zahlensystems darstellt. Die natürlichen Zahlen a_i müssen dabei der Bedigung $0 \leq a_i < B$ genügen. In einem bestimmten Zahlensystem dargestellte Zahlen werden dabei durch die Folge ihrer Koeffizienten gekennzeichnet. Durch Voranstellen eines negativen Vorzeichens lassen sich auch die negativen ganzen Zahlen darstellen. Die gebräuchlichsten Zahlendarstellungen sind die zur Basis 2 (Dualzahlen), zur Basis 8 (Oktalzahlen) und zur Basis 16 (Hexadezimalzahlen). Jede Dualzahl kann als eine Folge von Binärziffern gekennzeichnet werden. Ordnen wir dieser Folge von links nach rechts die Zahlen von 1-K zu, wobei K=Anzahl der Binärziffern, so bezeichnen wir diese Zahlen auch als (Dual-)stellen.

Beispiel:

Binärziffern:	1	0	0	0	0	1	=	33
Stelle:	1	2	3	4	5	6		

Jede Stelle einer Dualzahl läßt sich als Bit mit den Binärziffern als Zuständen auffassen. Jeder Dualzahl entspricht so ein Binärmuster.

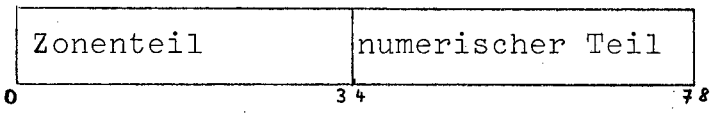
Stellt man einer Dualzahl eine weitere Stelle voran, deren Binärziffern das Vorzeichen der Dualzahl bezeichnen sollen, so nennen wir diese Stelle auch Vorzeichenbit.
Eine Dualzahl, deren erste Stelle als Vorzeichenbit interpretiert wird, bezeichnen wir auch als (duale) Festpunktzahl.
Die Vorzeichenverschlüsselung erfolgt dabei wie folgt:

Vorzeichen	Binärziffer
+	0
-	1

Negative Festpunktzahlen werden als Zweierkomplemente dargestellt. Ist K die Anzahl der Stellen einer Festpunktzahl, so ist die größte positive Zahl, die dargestellt werden kann, ohne, daß das Vorzeichenbit negativ wird: $2^{K-1}-1$. Die höchste negative Zahl jedoch ist 2^{K-1}. Arithmetische Operationen mit Festpunktzahlen werden als bekannt vorausgesetzt.

Zeichen:

Unter einem Zeichen verstehen wir alle alphanumerischen Zeichen und Sonderzeichen.
Die interne Darstellung von Zeichen erfolgt bei der Siemens 4004/45 im EBCDIC (Extended Binary Coded Decimal Interchange Code). Im EBCDID wirt jedem alphanumerischen Zeichen ein Bitmuster in der Länge eines Bytes zugeordnet. Jedes Byte zerfällt in einen Zonenteil (1.Halbbyte) und einen numerischen Teil (2. Halbbyte):



Byte

Durch ein Binärmuster der Länge eines Halbbytes können die Dualzahlen von 0-15 dargestellt werden. Ordnet man den natürlichen Zahlen 0-15 sukzessiv die Zeichen 0,1,2,3,4,5,6,7,8,9,A, B,C,D,E,F zu, so bezeichnet man letztere auch als Hexadezimalziffern.

Jedes Zeichen im EBCDIC kann so durch eine Folge von zwei Hexadezimalziffern dargestellt werden. Dezimalziffern werden im EBCDIC durch F 1 - F 9 dargestellt, der Zonenteil hat also den Wert F. Zu den Interpretationen der übrigen Zeichen im EBCDIC vgl. Heft Siemens System 4004/Zentraleinheiten, Anhang 6.

Binärmusterinterpretationen:

Zunächst wird jede Information in der Maschine grundsätzlich durch ein Binärmuster dargestellt. Ob das Binärmuster dann schließlich als Binärmuster, Zeichen oder Festpunktzahl interpretiert wird, hängt davon ab, durch welche Befehle das Binärmuster interpretiert und verarbeitet wird.
Der Benutzer intendiert also eine bestimmte Interpretation der Information, die er in die Maschine eingibt; ob diese Information in der Maschine in der intendierten Weise verarbeitet wird, hängt von folgenden Gegebenheiten ab:

- Benutzung der richtigen Befehle
- richtige Ausrichtung der Daten im Arbeitsspeicher: Ist in einem Festpunkt-Wortbefehl etwa die Arbeitsspeicheradresse nicht auf Wortgrenze ausgerichtet, ergibt dies eine Fehlermeldung wegen falscher Ausrichtung.

Dadurch wird es möglich, daß man etwa eine Folge von 4 Bytes, die auf Wortgrenze ausgerichtet ist und die eine Folge von Zeichen im EBCDIC darstellt, in einem Festpunkt-Wortbefehl als Festpunktzahl verarbeiten kann, wobei das erste Bit eben als Vorzeichen interpretiert wird. Ebenso wäre es möglich, eine Festpunktzahl in einem logischen Bit-für-Bit Befehl als reines Binärmuster zu behandeln.

Linksbündige und rechtsbündige Darstellung

Sei W ein Wort im Arbeitsspeicher. Dann wollen wir davon sprechen, daß ein Binärmuster rechtsbündig abgespeichert wird, wenn die höchstwertige Binärziffer des Binärmusters in das höchstwertige Bit des Wortes gespeichert wird, die zweithöchstwertige Binärziffer in das zweithöchstwertige usw. Entsprechend ist die linksbündige Darstellung definiert als eine Speicherung der niederwertigsten Binärziffer im niederwertigsten Bit usw.

2.3.2 Datenrepräsentation bei SCALLOP

Bei SCALLOP können folgende Datentypen mittels Programm eingeführt werden:
- Ketten von alphanumerischen- und Sonderzeichen im Literalformat (vgl. Abschnitt 3.4.1.2)
- Ganze Zahlen in der Größenordnung $-2^{31} \leq Z \leq 2^{31}-1$

Zeichen und Zeichenketten

Innerhalb von Literalen auftretende alphanumerische Zeichen und Sonderzeichen werden maschinenintern im EBCDIC-Code (vgl. Abschnitt 2.2.2.1) dargestellt.
Entsprechend der Verwendung der Siemens-Wortbefehle werden Literale auf Wortgrenze ausgerichtet. Die EBCDIC-Bytes der Zeichen in Literalen werden dabei rechtsbündig in Arbeitsspeicherworten abgespeichert.

Zahlen:

Ganze Zahlen können bei SCALLOP in jeder Zahlendarstellung mit einer Basis $2 \leq B \leq 36$ eingeführt werden.
Intern werden diese Zahlen jedoch stets als auf Wortgrenze ausgerichtete Festpunktzahlen von Wortlänge dargestellt.

Dateninterpretation:

Wie Daten maschinenintern weiterverarbeitet werden, hängt von den Befehlen ab, durch die die Daten interpretiert werden. Sie können etwa als Literale in Bit-für-Bit Operationen verarbeitet werden und so als reine Binärmuster interpretiert werden, oder sie können durch Festpunktbefehle als Festpunktzahlen interpretiert werden.

Codierung:

Bezüglich der Codierung eines SCALLOP-Programmes sowie der Eingabe des Programmes über Lochkarten sind folgende Konventionen zu beachten:

1. Das Zeichen Zwischenraum (blank) darf nicht in aus Zeichen zusammengesetzten Grundsymbolen (vgl. Abschnitt 3.2) auftreten. Im übrigen wird es vom Compiler überlesen. Das Zeichen Zwischenraum kann daher in den übrigen Programmteilen nach Belieben zur Verbesserung der Übersichtlichkeit eines Programmes verwendet werden.

2. Die Zeichenpositionen (Spalten) auf der Lochkarte sind von links nach rechts von 1 bis 80 fortlaufend numeriert. Vom Compiler werden die Spalten 1 bis einschl. 72 als Programmtext interpretiert. Die Spalten 73 bis 80 werden überlesen. Das Format der Lochkarte stellt jedoch keine Einschränkung bzgl. der Anzahl der Zeichen einer Anweisung oder einer Deklaration dar.
   Eine Anweisung oder Deklaration kann auf beliebig vielen Lochkarten fortgesetzt werden. Diese Fortsetzungskarten werden nicht besonders gekennzeichnet.
   Ebenso ist es erlaubt, auf einer Lochkarte mehrere Anweisungen oder Deklarationen einzugeben.

Schreibweise:

SCALLOP-Alphabet:

Das SCALLOP-Alphabet setzt sich aus den folgenden Zeichen zusammen:

	Ziffern	Buchstaben		Sonderzeichen		
	1	A	N	=		
	2	B	O	+	(	
	3	C	P	-	)	
	4	D	Q			
	5	E	R	/		
	6	F	S	.	∧	
	7	G	T	;	∨	
	8	H	U		◇	
	9	I	V	&	¬	
	0	J	W	'	*	(Zwischenraum)
		K	X	"		
		L	Y			
		M	Z			

Die Sonderzeichen [,] ,v werden hardwaremäßig dargestellt durch <, >, |. Die übrigen Zeichen stimmen mit ihrer hardwaremäßigen Darstellung überein.

3. Aufbau der Sprache

3.1. Metasprachliche Darstellung

Aufgrund des Deklarationskonzepts und weiterer Eigenschaften erweist sich SCALLOP als nicht kontextfreie Grammatik und analog zu ALGOL als nicht vollständig in der Backus-Naur-Form darstellbar.
Der hier verfolgte Weg besteht darin, die Backus-Naur-Form mangels akzeptabler Alternativen dennoch zu verwenden. Dort, wo sie nicht mehr durchgehalten werden kann, ist dies durch die semantischen Erläuterungen vollkommen klar. -
Es sind bereits Vorschläge für eine günstigere Syn-axdefinition gemacht worden, die wesentlich durch die Forderung bestimmt sind, die Compilerstruktur schon in der Syntax durchsichtiger werden zu lassen.
Die Backus-Naur-Form der Syntaxdarstellung genügt dabei den folgenden Konventionen:

- :: = weist einem syntaktischen Typ die Definitionsnamen zu;
- / trennt alternative Definitionsnamen;
- syntaktische Typen werden durch die Klammern '<' und '>' eingeschlossen;
- die Symbole der formalen Sprache, die erzeugt werden soll, in diesem Falle SCALLOP, werden nicht in Klammern '<' und '>' eingeschlossen.

Ein besonderer Status in der Syntax kommt dem Symbol "empty" zu. Es dient als leeres Wort und vereinfacht als solches an vielen Stellen die syntaktische Schreibweise.

Beispiel:

<C> :: = <A>|<B>

〈A〉 :: = <D> |empty

〈B〉 :: = a

Erklärung:

Da <A> auch das leere Wort sein kann, kann <C> auch aus a allein bestehen.

3.2. Grundsymbole

Syntax:

<basic symbol> :: = <digit> | <letter> |<delimiter>

Die Grundsymbole, aus denen sich die Worte der Sprache SCALLOP zusammensetzen, bestehen entweder aus einzelnen alphanumerischen Zeichen, bzw. Sonderzeichen oder aus mit Hilfe dieser Zeichen zusammengesetzten Zeichenfolgen wie z.B. die Deklaratoren.

3.2.1 Ziffern

Syntax:

<digit> :: = ϕ | 1 |2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.2.2 Buchstaben

Syntax:

<letter> :: = 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

3.2.3 Delimiter

Syntax:

< delimiter > :: = < operator >| < separator >| < bracket >| < specificator >| < declarator >

3.2.3.1 Operatoren

Syntax:

< operator > :: = < basic operator >| < relational operator >|
< sequential operator > |<logical operator>

< basic operator > :: = < arithmetic operator >| < bit by bit operator >

Arithmetische Operatoren:

Syntax:

<arithmetic operator> :: = +|-|*|/|/.

Die arithmetischen Operatoren stellen in der Reihenfolge der obigen Aufzählung die Addition, Subtraktion, Multiplikation, Division und Abfrage des Divisionsrestes dar. Es handelt sich um Operatoren für Festpunktarithmetik. Gleitpunktarithmetik ist in dieser SCALLOP-Version nicht vorgesehen.

Bit für Bit Operatoren:

Syntax:

<bit by bit operator> :: = ∧|∨|◇|∧¬|∨¬|◇¬|.+|.-

Bit für Bit Operatoren dienen der bitweisen Verknüpfung von Bitmustern. In der Reihenfolge der obigen Aufzählung stellen die Operatoren die folgenden Bitoperationen dar:
und, oder, exklusives oder, und nicht, oder nicht, exklusives oder nicht, Verschiebung nach links und Verschiebung nach rechts. Die beiden Verschiebungsoperatoren entsprechen den Hardwarebefehlen SLL und SLR.

Logische Operatoren:

Syntax:

<logical operator>:: = 'AND'|'OR'

Relational Operatoren:

Syntax:

<relational operator> :: = 'EQUAL'|'NOTEQUAL'|'GREATER'|
'NOTGREATER'|'LESS'|'NOTLESS'

Sequentielle Operatoren:

Syntax:

<sequential operator> :: = 'OVERFLOW'|'GOTO'|'TIMES'
'ELSE'|'OF'|'OTHERWISE'

Trennsymbole:
Syntax:
<separator> :: = ,|;|:|=|=:

Klammern:
Syntax:
<bracket> :: = (|)|[|]|"| |'('|')'|
'BEGIN'|'END'

Spezifikatoren:
Syntax:
<specificator> :: =@|'IF'|'CASE'|'DO'|&

Deklaratoren:
Syntax:
declarator :: ='PROGRAM'|'PROCEDURE'|'NEW'|
'FOREWARD'|'EXTERNAL'

3.3 Namen

Syntax:
<identifier> :: = <letter>|<identifier><letter>|
<identifier><digit>

Namen dienen zur Bezeichnung von Programmen, Prozeduren, Marken und Arbeitsspeicheradressen. Alle Namen außer denjenigen von Marken müssen vor ihrer Verwendung in einer Deklaration erklärt worden sein. Die Anzahl der in einem Namen vorkommenden Buchstaben und Ziffern ist nicht beschränkt. Unterscheidende Bedeutung haben jedoch nur die ersten sechzehn Zeichen.
Blanks können beliebig zur besseren Lesbarkeit eingeschoben werden. Sie werden vom Compiler überlesen und haben keine unterscheidende Bedeutung.

Beispiel:
zulässige Namen:
I D 8 5 V 6
K V R S 6 0 7 B
M O 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L
M O 1 2 3 4 5 6 7 8 9 A B C D E F G 1 2 3 4
Die beiden letzten Namen werden als gleich interpretiert, da sie in den ersten sechzehn Zeichen übereinstimmen.

unzulässige Namen:

1 A B Beginnt mit einer Ziffer

A B " D E } Enthalten Sonderzeichen
A B & C S }

3.4 Operand

Syntax:

<operand> :: = <constant>|<variable>|<adress variable>

Operanden sind Sprachelemente, die mittels Operatoren zu Ausdrücken verknüpft werden.

3.4.1 Konstanten

Syntax:

<constant> :: = <number>|<literal>

3.4.1.1. Zahlen

Syntax:

<number> :: = <decimal number>|
<base number>

Zahlen können zu jeder Basis B mit $2 \leq B \leq 36$ angegeben werden. Ihr Wert muß zwischen -2^{31} und $2^{31}-1$ liegen. Die Zahl Null wird in jedem Zahlensystem durch Ø dargestellt. Blanks können zur besseren Lesbarkeit einer Zahl an beliebiger Stelle eingefügt werden.

Dezimalzahlen:

Syntax:

<decimal number> :: = 1|2|3|4|5|6|7|8|9|
<decimal number><digit>

Die erste Ziffer einer Dezimalzahl darf abgesehen von der Zahl Null keine Null sein.

Beispiele:

zulässig: 1 2 4
9 8 7 6 5 4 3 2 1

unzulässig: Ø 1 3 7 führende Null
7 A B 6 7 enthält Zeichen ungleich Ziffer

Basiszahlen:
Syntax:

<base number> :: = ϕ <base number> <extended digit>
<extended digit> :: = <digit>|<letter>

Basiszahlen sind Zahlen zu einer vom Benutzer vorgegebenen Basis zwischen 2 und 36. Als erweiterte Ziffern stehen die Ziffern ϕ, 1, ..., 9 sowie die Buchstaben A, B, C,..., X, Y, Z zur Verfügung. Entsprechend ihrer Stellung im Alphabet werden den Buchstaben in ihrer Eigenschaft als erweiterte Ziffern die Werte zwischen 1ϕ und 35 zugeordnet. So hat z.B. der Buchstabe A den Wert 1ϕ, der Buchstabe P den Wert 25 und der Buchstabe Z den Wert 35.
Die erste Ziffer einer Basiszahl muß eine Null sein.
Z.Zt. sind als Basiszahlen nur Sedezimalzahlen zugelassen.

Beispiele:
zulässig: ϕ 1 A B 2 ϕ C
ϕ F A C 1 3

unzulässig: ϕ A & B 1 enthält Zeichen ungleich erweiterte Ziffer
D 1 B keine führende Null.

3,4.1.2 Literale
Syntax:

<literal> : = <left bounded literal> |<right bounded literal>

<left bounded literal> :: = "<any sequence of characters not containing'"'> "

<right bounded literal> :: = #<any sequence of characters not containing '#'>#

Literale dienen zur Einführung von nicht-numerischen Daten in Programme. Sie werden z. B. benutzt,um ausdruckbare Zeichenfolgen zu definieren. Literale dürfen innerhalb der Spezifikationszeichen maximal 4 Zeichen enthalten, die ungleich diesem Spezifikationszeichen sind. Rechtsbündige Literale werden rechtsbündig (S. 2.3.1) gespeichert und mit führenden binären Nullen auf Wortlänge aufgefüllt. Linksbündige Literale werden linksbündig (S. 2.3.1) gespeichert und mit Leerzeichen auf Wortlänge aufgefüllt.

Beispiele:
zulässige Literale
1. "1+-"-
2. ": & "

3. "Ø =:C"
4. ✱3v¬✱
5. ✱: & "✱
6. ✱Ø =:C✱

unzulässig:

7. "1"34" enthält "
8. ✱1 2 3 4 5✱ enthält mehr als 4 Zeichen

3.4.2 Variablen

Syntax:

```
<variable> :: = <declared variable>|<procedure value>
                <computed variable>
```

Ihrer Semantik nach ist eine Variable ein Wort im Arbeitsspeicher.

3.4.2.1 Deklarierte Variable

Syntax:

```
<declared variable> :: = <new variable>|<procedure variable>
```

NEW-Variable:

Syntax:

```
<new variable> :: = <new identifier> |
                    <new identifier><subscript>
<new identifier> :: = <identifier>
<subscript> :: =[<simple expression>]
<simple expression>:: =<simple operand>|+<simple operand>|
-<simple operand>|<simple expression>+<simple operand>|
                 <simple expression>= : <new variable>
<simple operand> :: =<constant>|<new variable>|
                     <computed variable>
```

NEW-Variable sind Arbeitsspeicheradressen, für die der Benutzer in einer besonderen Deklaration, der NEW-Deklaration, Namen vergeben kann. NEW-Variable müssen in einer NEW-Deklaration erklärt werden. Durch den Namen der NEW-Variablen wird symbolisch eine Wortadresse im Arbeitsspeicher angesprochen.
Jede NEW-Variable kann indiziert werden, selbst wenn sie ohne Freihalteanweisung (S. 3.7.2.1) erklärt ist. Als Index wird ein "einfacher Ausdruck" verwendet, in dem als Operationen nur Addition, Subtraktion und Zuweisen eines Wortes an eine NEW-Variable erlaubt ist.

Zur Zeit darf nur der erste Operand in einem einfachen Ausdruck eine subskribierte NEW-Variable oder eine berechnete Variable (vgl. Abschnitt 3.4.2.3) sein. Ist k der Wert des einfachen Ausdruckes, A eine NEW-Variable, so bezeichnet man A{k}die Adresse des k-ten Wortes, das auf das Wort mit der symbolischen Adresse A folgt.

Beispiele:
zulässig:
A [17]
A [ϕ B 6]
A [B [C [D [E+3] +2] +1] = : C + ϕ B]

unzulässig:
A [5 + B[6] = : C [12] + U] Verletzung der Konvention
A [B * C] unzulässiger Operator

Prozedurvariable:
Syntax:
<procedure variable> :: = <procedure identifier>
<procedure identifier> :: = <identifier>

Eine Prozedurvariable wird durch einen Namen gekennzeichnet, der in einer Prozedurerklärung definiert ist.
Durch den Prozedurnamen wird erstens ein Wort im Arbeitsspeicher angesprochen, über das die Prozedur mit ihrer Umgebung kommuniziert.
Zweitens wird durch Verwendung eines Prozedurnamens in einem Ausdruck (s. 3.5) die Prozedur aufgerufen.

3.4.2.2 Prozedurwert

Syntax:
<procedure value> :: = & <procedure identifier>

Die Variable Prozedurwert ist eine lokale Variable, mit deren Hilfe die Prozedur mit ihrer Umgebung kommuniziert.
Durch & <procedure identifier> wird innerhalb der Prozedur, die durch den Prozedurnamen bezeichnet wird, das Wort im Arbeitsspeicher angesprochen, das durch den Prozedurnamen allein ebenfalls angesprochen wird. (s 3.4.2.1)

Der Unterschied zwischen den beiden Variablen Prozedurwert und Prozedurvariable besteht darin, daß bei Verwendung der Variablen Prozedurwert in einem Ausdruck die Prozedur selbst nicht durchlaufen wird.

3.4.2.3 Berechnete Variable

Syntax:

<computed variable> ::= <subscript>

Die Verwendung der berechneten Variablen entspricht der indirekten Adressierung, während mit dem Namen einer New-Variablen symbolisch adressiert wird.
Im Gegensatz zur subskribierten New-Variablen (s. 3.4.2.1) bezieht sich der Wert des einfachen Ausdruckes bei der berechneten Variablen auf Adressen von Bytes.
Da in Ausdrücken (s. 3.5) nur mit Inhalten oder Namen von Arbeitsspeicherworten gearbeitet wird, muß bei der Verwendung einer berechneten Variablen in einem Ausdruck der Wert des einfachen Ausdruckes durch 4 teilbar d.h. auf Wortgrenze ausgerichtet sein. Mit der Variablen wird dann das Wort im Arbeitsspeicher angesprochen, dessen erstes Byte durch den Wert des einfachen Ausdruckes adressiert wird.

Beispiele:

[864] Es wird das Wort mit den Byteadressen 864, 865, 866 und 867 angesprochen.

[865] Adressierungsfehler. Der Index ist nicht auf Wortadresse ausgerichtet.

3.4.3 Adresse

Syntax:

<address> ::= <address variable>|<string>

3.4.3.1 Adressvariable

Syntax

<addressvariable> ::= <new identifier>
| <new identifier>| <subscript>

Eine Adressvariable gibt die Adresse einer NEW-Variablen an.

Beispiele:

@A	absolute Adresse der NEW-Variablen A
@A [17]	Adresse der NEW-Variablen A [17]

3.4.3.2 StString

Syntax:

<string> :: = '(' <proper string> ')'

<proper string> :: = empty |<character>|
<proper string> <character>

Wird ein String einer Variablen zugeordnet, so wird in der von der Variablen bezeichneten Stelle die Adresse gespeichert, die den Speicherplatz des Strings angibt. Aus diesem Grunde wird hier der String zu den Adressen gerechnet. Die maximale Zeichenwahl in einem String beträgt 255.

Beispiel:

'(' ')'

'(' A*@" ';')'

Das Konzept des Strings soll in allernächster Zukunft mit dem Strinkonzept von GPSP kompatibel gemacht werden. Aus diesem Grunde wird auf eine Beschreibung des Strings verzichtet.

3.5 Ausdrücke

Syntax:

<expression> :: = <operand> |+ <operand> |- <operand> |
<expression>< bit by bit operator> <operand>|
<expression> <arithmetic operator> <operand>|
<expression> <arithmetic operator> <operand>
'OVERFLOW' <identifier> /
<expression> = : <variable>

Ausdrücke werden in SCALLOP strikt von links nach rechts abgearbeitet. Es gibt keine Priorität unter Basisoperatoren (s.3.2.3.1) Die Verwendung von Klammern ist in Ausdrücken nicht erlaubt.

3.5.1 Ausdrücke mit Bit für Bit Operatoren

Logische Bit für Bit Operatoren:

Wird ein Ausdruck durch einen logischen Bit für Bit Operator mit einem Operanden verknüpft, so wird das den Wert des Ausdruckes darstellende Bitmuster bitweise nach folgender Wahrheitstabelle verknüpft:

E	OP	$\wedge$	v	$\diamond$	$\wedge_\neg$	$v_\neg$	$\diamond_\neg$
1	1	1	1	Ø	Ø	1	1
1	Ø	Ø	1	1	1	1	Ø
Ø	1	Ø	1	1	Ø	Ø	Ø
vØ	Ø	Ø	Ø	Ø	Ø	1	1

Beispiele:

E = 1 1 Ø 1 Ø 1 ....
OP = Ø 1 1 Ø Ø Ø ....
E $\wedge$ OP = Ø 1 Ø Ø Ø Ø ....
E v OP = 1 1 1 1 Ø 1 ....
E $\diamond$ OP = 1 Ø 1 1 Ø 1 ....
E $\wedge_\neg$ OP = 1 Ø Ø 1 Ø 1 ....
E $v_\neg$ OP = 1 1 Ø 1 1 1 ....
E $\diamond_\neg$ OP = Ø 1 Ø Ø 1 Ø ....

In diesen Beispielen sind nur 6 Bits der 32 Bit langen Wörtern E und OP angegeben.

Verschiebungsoperatoren:

Wird der Ausdruck durch einen Verschiebungsoperator (Schriftoperator) mit dem Operanden verbunden, so wird das Bitmuster, das den Wert des Ausdruckes im Rechner darstellt um so viele Stellen verschoben, wie die Verschiebezahl im Operanden nann- gibt.

Die Verschiebezahl ist durch die 6 niederwertigsten Bits des Operanden gegeben, d.h. ihr Dezimalaquivalent liegt zwischen 0 und $2^6-1 = 63$.
Wird der Operand als Dezimalzahl angegeben, so liefern die Zahlen N und N+k*64 mit k=0,±1, ... die gleiche Verschiebezahl. Bei der Links-Verschiebung gehen die nach links hinausgeschobenen Stellen verloren. Rechts werden binäre Nullen nachgezogen.
Bei der Rechts-Verschiebung gehen die nach rechts hinausgeschobenen Stellen verloren. Von links werden binäre Nullen nachgezogen. Es gibt keine Sonderbehandlung für das Vorzeichen.

3.5.2 Arithmetische Ausdrücke und Abfrage von Überlauf
Im folgenden sind als Beispiel einige mathematische Relationen sowie ihr Pendant in SCALLOP aufgeführt.

mathematische Relationen	entsprechender Ausdruck in SCALLOP
A + B*C	B*C + A
(A + B)*C	A + B*C
(A + B*C)*C	B*C + A*C
(A*C + C)*B	A*C + C*B

Da nur Festpunktarithmetik vorgesehen ist, erhält man bei Division nur ganzzahlige Ergebnisse. Der Ausdruck 17/5 liefert z.B. den Wert 3.
Den Divisionsrest 2 erhält man durch den Ausdruck 17 /. 5.
Wie bereits vorher gesagt wurde, werden Operanden im Arbeitsspeicher in einem Wort gespeichert. Wird bei arithmetischen Ausdrücken das Ergebnis länger als ein Wort, d.h. der Wert des Ergebnisses ist kleiner als -2^{31} bzw. größer als $2^{31}-1$, so sprechen wir von Überlauf. Eine Programmunterbrechung findet nicht statt. Mit dem Operator'OVERFLOW' wird abgefragt, ob Überlauf eingetreten ist. Ist Überlauf eingetreten, so wird die Marke angesprungen, die durch den auf 'OVERFLOW' folgenden Namen gekennzeichnet wird.

Folgt auf Multiplikation sofort Division bzw. Abfrage des Divisionsrestes, so wird beim Eintreten von Überlauf aufgrund der Multiplikation der richtige Wert des Produktes als Dividend benutzt, da intern Multiplikand und Dividend auf einem Doppelwort stehen und das Produkt rechtsbündig in einem Doppelwort gespeichert ist (vgl. Zentraleinheit).
Tritt in einem Ausdruck Division durch Null auf, so wird diese nicht ausgeführt. Es gibt die folgende Regelung:
A / ϕ = : C bewirkt, daß der Wert von A auf C steht.
In Abhängigkeit von der Maschine gilt:
1. A /. ϕ =: C
Ist der Wert von A kleiner als Null, so enthält C den Wert -1, sonst den Wert ϕ.

2. A*B /. ϕ =: C
C enthält die 32 höherwertigsten Bits des Produktes.

3.5.4 Wertzuweisung

Mit dem Ausdruck <expression> =: <variable>
wird der Variablen der Wert des Ausdruckes zugewiesen.

Beispiel:
A, B, C, D seien NEW-Variablen. Der Ausdruck A+B =: C*C =: D bewirkt, daß der Wert der Summe in C gespeichert wird. Weiter wird C mit sich selbst multipliziert und das Ergebnis nach D gespeichert. Damit enthält D das Quadrat der Summe A+B.

3.6 Anweisung

Syntax:

```
<statement> :: =  empty|<expression>|
                  <compound statement>|
                  <if statement>|<case statement>|
                  <loop statement>|<goto statement>|
                  <label definition><statement>
```

Das leere Statement ist nur in der Statementliste des Case-Statement relevant.

3.6.1 Ausdrücke

vgl. 3.5

3.6.2 Zusammengesetzte Anweisung

Syntax:

<compound statement> :: = 'BEGIN' <statement -list> 'END'
<statement-list> :: = <statement> |
<statement list> ; <statement>

Beispiele:

1) 'BEGIN'
A+B =: C*D+E*C =: C
'END'

2) 'BEGIN'
A+B =: C;
'BEGIN'
D+E =: F+C*C =: E
'END';
C+E*E =: C
'END'

3.6.3 IF Anweisung

<if statement> ::= 'IF' <condition> 'THEN'
<statement> 'ELSE' <statement>
'IF' <condition> 'ELSE' <statement>

<condition> ::= <expression> | <relation> |
<condition> <logical operator> <expression> |
<condition> <logical opreator> <relation>

<relation> ::= <expression> <relational operator> <expression> |
<relation> <relational operator> <expression>

[illegible] wenn eine weitere Abfrage [illegible] wenn kann.

Beispiele:

1. 'IF' N 'NOTGREATER' 5 'AND' B
'LESS' 0 'THEN'
<statement>
'ELSE' <statement>

2. 'IF' A 'EQUAL' B 'OR' B 'NOTEQUAL' C
'ELSE' <statement>

3.6.4 CASE - Anweisung

Syntax:

<case statement> ::='CASE'<expression>'OF'
<statement-list>'OTHERWISE'<statement>

Entsprechend dem Wert des Ausdruckes wird ein Statement in der Statement-Liste angesprungen. Ist z.B. der Wert des Ausdruckes gleich k und enthält die Statement-Liste n Statements mit

$$\phi = < k \leq n,$$

so wird das k-te Statement der Statement-Liste angesprungen. Nachdem dieses Statement abgearbeitet ist, wird die Abarbeitung bei dem auf das CASE-Statement folgende Statement fortgesetzt. Gilt für den Wert k des Ausdruckes

$$\phi \geq k \quad \text{oder } n < k,$$

so wird das Statement angesprungen, das auf'OTHERWISE' folgt. Ein leeres Statement in der Statement-Liste oder nach 'OTHERWISE' bewirkt, daß beim Ansprechen dieses Statements die Abarbeitung des Programmes sofort bei dem Statement fortgesetzt wird, das auf das CASE-Statement folgt. Es muß beachtet werden, daß auf das letzte Statement in der Statement-Liste vor dem 'OTHERWISE' kein Semikolon folgt. Tritt vor'OTHERWISE'ein Semikolon auf, so bedeutet das, daß noch ein leeres Statement auf das Semikolon folgt.

Beispiel:

Die n Statements der Statement-Liste werden im folgenden symbolisch mit LSTMn, das auf'OTHERWISE' folgende Statement mit OSTM und das auf das CASE-Statement folgende Statement mit STM bezeichnet.
Nimmt der Ausdruck E den Wert 5 an, so soll LSTM1, für E=6 LSTM2, für E=8 LSTM4 ausgeführt werden. Für E=7 soll mit STM die Abarbeitung fortgesetzt werden. Für alle anderen Werte von E soll das Programm mit OSTM fortgesetzt werden.
Die Aufgabe kann folgendermaßen gelöst werden:

```
'CASE' E  -4  'OF'
LSTM1;
LSTM2;
;
LSTM4 'OTHERWISE' OSTM;
STM
```

3.6.5 Schleifenanweisung

Syntax:

<loop statement> ::= 'DO' <expression> 'TIMES'
<statement>

Das auf 'TIMES' folgende Statement wird dem Wert des Ausdruckes entsprechend oft wiederholt. Das Loop-Statement kann beliebig geschachtelt werden. Vom Compiler her sind keine Einschränkungen gegeben. Im Gegensatz zu dem Loop-Statement in ALGOL oder FORTRAN muß der Benutzer die Indexfortschaltung in SCALLOP selber vornehmen. Das hat den Vorteil, daß ein effizienterer Code erzeugt werden kann. Weiter werden Probleme vermieden, die bei Rekursion in der Schleife auftauchen.

Beispiel:
Es sollen zwei Felder mit den Namen A und B auf Gleichheit geprüft werden. Sobald zwei ungleiche Worte gefunden sind, wird der Vergleich abgebrochen und eine Meldung durch die Prozedur MESSAGE ausgedruckt. Im ersten Wort von A und B ist die Länge des jeweiligen Feldes angegeben.

```
'IF' A 'NOTEQUAL' B 'THEN'
     'GOTO' ESCAPE
                    'ELSE'
'BEGIN'
   ϕ =: N;
  'DO' A 'TIMES'
         'BEGIN'
          N+1 =: N;
         'IF' A  N 'EQUAL' B  N
         'ELSE'    'GOTO' ESCAPE
         'END'
   'GOTO' CONTINUE
   ESCAPE: MESSAGE
  'END';
   CONTINUE: ....
```

3.6.6 Sprunganweisung

Syntax:

<goto statement> ::='GOTO'<identifier>

Das GOTO-Statement ist eine Sprunganweisung. Der Name bezeichnet die Marke, die angesprungen wird.
Es kann nicht aus einer Prozedur heraus- oder in eine Prozedur hineingesprungen werden. Weiter darf nicht in eine Schleife hineingesprungen werden.

3.6.7 Anweisungen mit Marken

Syntax:

<label definition> ::= <identifier>:

Jede Anweisung kann mit einer Marke versehen werden. Marken werden nicht deklariert. Sie sind prozedurspezifisch d.h. sie können nicht von einer anderen Prozedur aus angesprochen werden.

3.7) Programmstruktur

3.7.1 Allgemeiner Aufbau

Die Programmstruktur ist durch den folgenden Syntaxteil definiert:

```
<program> ::= <program head> <body>
<program head> ::= 'PROGRAM' <identifier>
<body> ::= <compound statement> |
           <declaration list> <compound statement>
<declaration list> ::= <simple declaration list> |
                       <procedure declaration list> |
                     <simple declaration list> <procedure
                                                declaration list>
<simple declaration list> ::= <simple declaration> |
                              <simple declaration list>
                                <simple declaration>
<simple declaration> ::= <external declaration> |
                         <forward declaration> | <new declaration>
<procedure declaration list> ::= <procedure> |
                                 <procedure declaration list>
                                 <procedure>
```

Ein Programm besitzt also die folgende Struktur:

'PROGRAM' Programmname	Programmkopf
Deklarationsliste 'BEGIN' <Anweisungsliste> 'END'	Programmrumpf

Es ist zu beachten, daß die Deklarationsliste auch leer sein kann, d.h. der Programmrumpf nur aus der zusammengesetzten Anweisung besteht.

3.7.2 Einfache Deklarationen

Einfache Deklarationen unterscheiden sich dadurch von den übrigen, daß sie keine Beschreibungen von Unterprogrammen beinhalten.

3.7.2.1 'NEW-Deklaration

Syntax:

<new-declaration> ::= 'NEW' <new element> | <new declaration> <new element>

<new element> ::= <new identifier> <reservation factor> | <new identifier> <reservation factor> <fill list>

<reservation factor> ::= empty | <number>

<fill list> ::= <fill element> | <fill list> <fill element>

<fill element> ::= <repetition factor> <constant> | <address constant>

<repetition factor> ::= empty | <number>

<new identifier> ::= <identifier>

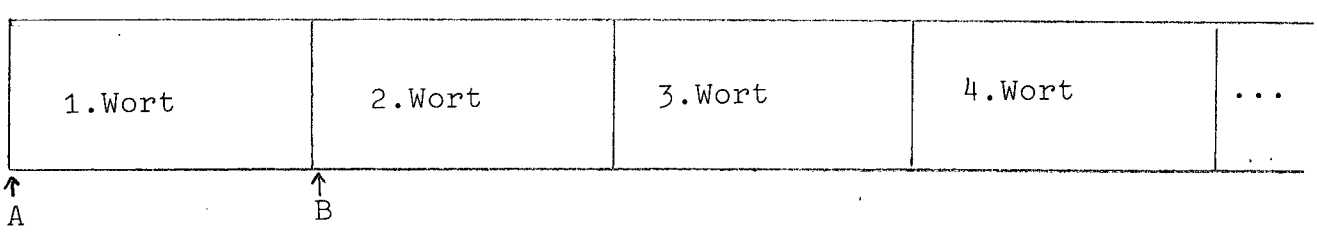
<address constant> ::= @ <new identifier> <address addend>

<address addend> ::= empty | <decimal number>

Die NEW-Deklaration dient der Zuweisung von Namen an Wortadressen im Arbeitsspeicher. Diese Namen heißen auch "new identifier" (NEW-Namen).
Die den NEW-Namen zugeordneten Worte heißen auch NEW-Variablen.
Die nach NEW erfolgten Zuweisungen behalten in allen untergeordneten Prozeduren ihre Gültigkeit, sofern dort nicht über die Namen anders verfügt wird, d.h. eine neue Zuordnung der Namen an Wortadressen erfolgt.

Beispiel: 'NEW' A, B

Speicherbereich



Hinweis: Durch Indizierung ist es möglich, sich direkt auf jedes Wort zu beziehen, das auf A oder B folgt, bzw. vorangeht (vgl. Abschnitt 2.2.2)

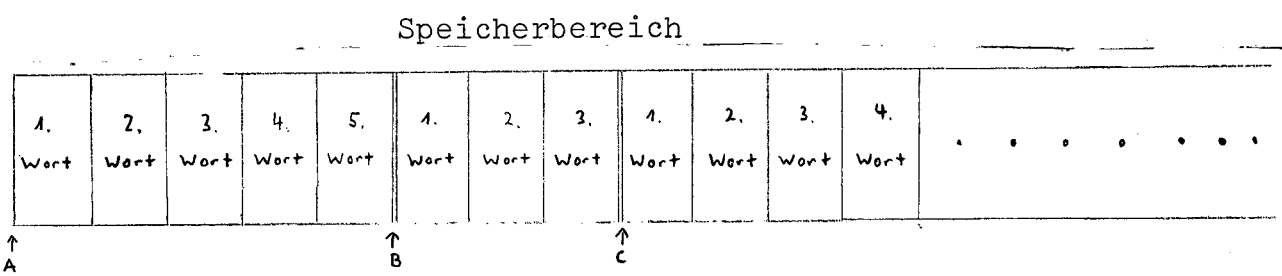
```
A [0] = A
A[1]  = B
A [2] = B [1]
A [B[1]+3]
usw.
```

Über die einfache Deklaration von Variablen hinaus bietet die NEW-Deklaration folgende Möglichkeiten:

- Angabe eines Reservierungsfaktors (reservation factor): Es kann durch die Angabe einer Zahl in eckigen Klammern hinter dem Variablennamen angegeben werden, wieviele Worte, beginnend an der durch den Variablennamen definierten Arbeitsspeicheradresse für die entsprechende Variablen freigehalten werden sollen, bei der der nächste Name vergeben werden soll. Ist keine Zahl angegeben, so beträgt die Länge des zugeordneten Bereichs von Worten genau ein Wort. Ist ein Reservierungsfaktor n angegeben, so beträgt die Anzahl der reservierten Worte n+1.

Beispiel:

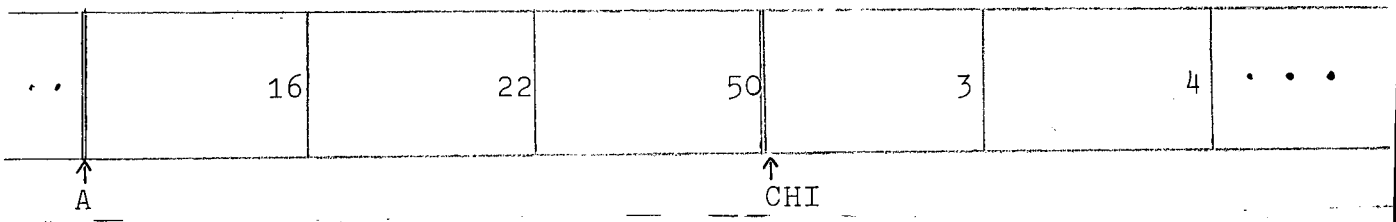
'NEW' A [4], B [2] , C [3]



- Angabe einer Füll-Liste (fill list). Durch Angabe einer Füll-Liste können den Worten von Wortbereichen im Arbeitsspeicher, die mit dem NEW-Namen zugeordneten Wort beginnen, Anfangswerte zugewiesen werden. Letztere müssen, durch Kommata getrennt, hinter den Reservierungsfaktor geschrieben werden, wobei zwischen Reservierungsfaktor und Anfangswerten ein Gleichheitszeichen stehen muß. Abgespeichert werden die Werte rechtsbündig in den Arbeitsspeicherworten, linksbündig werden binäre Nullen nachgezogen.

Beispiel:

'NEW' A [2] = 1 6, 22, 5o, CHI [1] = 3,4

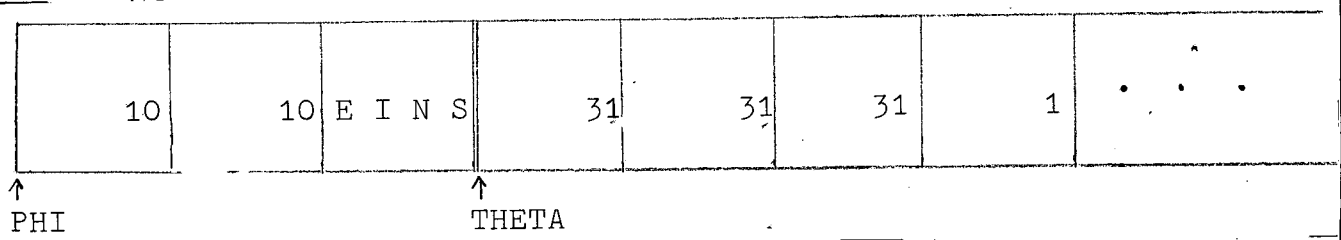


Durch Angabe eines Wiederholungsfaktors kann man es sich ersparen, identische Anfangswerte von Variablen immer wieder erneut hinzuschreiben. Der Wiederholungsfaktor besteht in einer Zahl, die vor den betreffenden Anfangswert geschrieben wird, wobei zwischen Anfangswert und Wiederholungsfaktor das Zeichen * geschoben werden muß

Beispiel:

'NEW' PHI [2]= 2*1o, # EINS# , THETA [3] = 3 * 31,1

Speicherbereich



Ist der Reservierungsfaktor größer als die Anzahl der zugewiesenen Anfangswerte, so wird der Inhalt der Worte, denen kein Wert zugewiesen wurde, gleich 0 gesetzt.
Ist die Länge der Füll-Liste, d.h. die Anzahl der zugewiesenen Werte größer als der Reservierungsfaktor, so werden diese dennoch sukzessiv den folgenden Arbeitsspeicherworten zugewiesen. Der folgende NEW-Name wird dann dem Wort zugewiesen, dem als erstes kein Anfangswert mehr zugewiesen wurde.

Hinweis: Als Anfangswerte dürfen mehr Konstanten und Adresskonstanten zugewiesen werden. Adresskonstanten sind dabei definiert als Adressen fester Arbeitsspeicherbytes, die von einer gegebenen Wortadresse aus durch Addition einer festen Anzahl von Bytes (des sogen. address addend) berechnet werden können (vgl. Abschn. 2.2.2.3) Eine Adresskonstante wird dadurch symbolisiert, daß das Zeichen @ vor einen NEW-Namen gesetzt wird, der durch eine feste Konstante indiziert sein kann.

——— Beispiel:

@B [2]

Hierdurch würde die Adresse des ersten Bytes des dritten Wortes des durch B definierten Speicherbereichs bezeichnet werden. Durch Addition einer Zahl, die die Anzahl der Bytes angibt, kann man sich auf bestimmte Bytes in Worten des durch eine Variable definierten Speicherfeldes beziehen.

Beispiel:

@B [2] + 2

Hierdurch würde die Adresse des dritten Bytes des dritten Wortes bezeichnet werden, das auf das durch B bezeichnete Wort folgt.

Ist die Adresskonstante so beschaffen, daß hinter dem Variablen-Namen in Klammern eine Literalkonstante steht, so wird diese Literalkonstante als Festpunktzahl interpretiert. Die Bedeutung dieser Festpunktzahl entspricht dann den obigen Erläuterungen.

Beispiel:

@TAB [#ABC#] + 2

Diese Adresskonstante bezeichnet die Adresse des dritten Bytes des Wortes, das durch Addition der dem Binärmuster ABC entsprechenden Festpunktzahl und der Adresse von TAB berechnet wurde.

3,7.2.2 FOREWARD - Deklaration

Syntax:

⟨foreward declaration⟩ ::= "FOREWARD" ⟨procedure identifier⟩ |
⟨foreward declaration⟩,⟨procedure identifier⟩

Der SCALLOP Compiler verlangt, daß jeder Name vor seiner Verwendung deklariert worden ist. Ausgenommen von dieser Regelung sind Markennamen und Namen von Standardprozeduren. Da es möglich ist, daß in Prozeduren andere Prozeduren aufgerufen werden, die zu diesem Zeitpunkt noch nicht deklariert sind, muß dem Compiler in einer vorausgegangenen FOREWARD-Deklaration mitgeteilt werden, daß es sich um einen Namen handelt, der interpretiert werden kann und dessen Deklaration noch erfolgen wird.

Eine FOREWARD-Deklaration muß also immer dann erfolgen, wenn ein Prozedurname benutzt wird, bevor er deklariert ist. Dabei gelten folgende Konventionen:

- FOREWARD-Deklarationen müssen in der Deklarationsliste vor den Prozedurdeklarationen stehen.
- FOREWARD-Deklarationen können mehrere Prozedurnamen enthalten, die durch Kommata zu trennen sind.
- Der Geltungsbereich von FOREWARD-Deklarationen erstreckt sich nur auf ein bestimmtes Niveau der Schachtelungstiefe.

Beispiel:

falsch

```
'PROGRAM' FOR
'NEW' A, B
'PROCEDURE' P 1
'FOREWARD' P 2
'BEGIN'
   .
   .
   .
P 2;
   .
   .
   .
'END'
'PROCEDURE' P 2
'BEGIN'
   .
   .
   .
'END'
'BEGIN'
   .
   .
'END'
```

richtig

```
'PROGRAM' FOR
'NEW' A, B
'FOREWARD' P 2
'PROCEDURE' P 1
'BEGIN'
   .
   .
   .
 P 2;
   .
   .
   .
'END'
'PROCEDURE' P 2
'BEGIN'
   .
   .
   .
'END'
'BEGIN'
   .
   .
'END'
```

Im linken Beispiel steht FOREWARD auf einem höheren Niveau der Schachtelungstiefe als P 2.

3.7.2.3 Externe Prozeduren

Syntax:

⟨external declaration⟩ ::= 'EXTERNAL' ⟨external procedure identifier⟩ |
⟨external declaration⟩ ,
⟨external procedure identifier⟩

⟨external procedure identifier⟩ ::= ⟨identifier⟩

Externe Prozeduren entsprechen etwa den CODE-Prozeduren bei ALGOL. Es handelt sich bei ihnen um getrennt übersetzte Programmabschnitte, die auch in anderen Programmiersprachen programmiert sein können.

Anschlußkonventionen müssen über eigene Programmierung realisiert werden. - Die Regeln für ihre Verwendung im Programm sind die gleichen wie für die gewöhnlichen Prozeduren, mit der einzigen Ausnahme, daß sie nicht innerhalb des Programms beschrieben werden und ihre Eingabe getrennt erfolgt.

3.7.3 Prozeduren

Jede SCALLOP-Prozedur läßt sich auffassen als ein Programmteil mit allen Eigenschaften eines Hauptprogramms und den zusätzlichen Eigenschaften des rekursiven Aufrufs und der Aufrufsschachtelung, der in die Umgebung der restlichen Programmteile eingebettet ist und mit einigen dieser restlichen Programmteile kommuniziert.

Prozedurerklärung:

Die Definition einer Prozedur vollzieht sich in der Prozedurerklärung (Prozedurdeklaration). Bei SCALLOP hat die Prozedurerklärung folgende Form:

Syntax:

⟨procedure declaration list⟩ ::= ⟨procedure⟩ | ⟨procedure declaration list⟩ ⟨procedure⟩

⟨procedure⟩ ::= ⟨procedure head⟩ ⟨body⟩

⟨procedure head⟩ ::= 'PROCEDURE' ⟨procedure identifier⟩

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨body⟩ ::= ⟨compound statement⟩ |
⟨declaration list⟩ ⟨compound statement⟩

Prozeduren werden also deklariert, indem man ihren Namen hinter die Zeichenkombination 'PROCEDURE' schreibt und diesem Namen eine zusammengesetzte Anweisung oder eine Deklarationsliste, gefolgt von einer zusammengesetzten Anweisung, folgen läßt.

```
'PROCEDURE' ZAHL
'BEGIN'
 statement list
'END'
```

Wie die formale Syntax weiter zeigt, können mehrere Prozedurdeklarationen ineinandergeschachtelt sein:

```
'PROCEDURE' IDENT
'NEW' A 3, B 5
    'PROCEDURE' CENT
    'NEW'  K[3]= @Z, 3 * 24
    'FOREWARD' SYS 1, SYS 2
    'BEGIN'
     statement list
    'END'
'BEGIN'
 statement list
'END'
```

1 2

Obiges Beispiel zeigt zwei ineinandergeschachtelte Prozedurdeklarationen, die jede für sich Deklarationslisten enthält, wobei die Deklarationsliste der äußersten Prozedur (1) die innere Prozedur (2) enthält.

Prozeduraufrufe:
Eine Prozedur wird in SCALLOP aufgerufen, indem ihr Name in einem Ausdruck verwendet wird.
Es können mehrere Prozeduraufrufe ineinandergeschachtelt sein: A ruft B auf, B ruft C auf, usw.
Die Anzahl der Schachtelungen ist allein durch die Arbeitsspeicherkapazität und die maximale Anzahl der symbolischen Adressen bestimmt, die durch den ASSEMBLER verarbeitet werden kann.

Rekursive Aufrufe sind gestattet. Sie werden in der gleichen Weise verarbeitet wie andere Aufrufe auch. Wird eine Prozedur rekursiv aufgerufen, so wird für die aufgerufene Prozedur ein neues Wort für den Prozedurwert zugeordnet, d.h. der Prozedurwert der aufgerufenen Prozedur geht bei wiederholtem Durchlauf nicht verloren (vgl. nächster Abschnitt).

Parameterübergabe:
Für die Kommunikation einer Prozedur mit ihrer Umgebung stellt der Compiler ein spezielles Speicherwort zur Verfügung, dessen Inhalt sichergestellt wird, sobald eine andere Prozedur aufgerufen wird. Den Inhalt dieses Wortes bezeichnen wir auch als den Wert der Prozedur. Der Inhalt dieses Wortes wird auch als Wert der Prozedur bezeichnet. Außerhalb einer Prozedur wird das Wort, das den Prozedurwert enthält, durch den Namen der Prozedur angesprochen. Bei Auftreten eines Prozedurnamens wird außerdem die entsprechende Prozedur durchlaufen (vgl. Abschnitt 3.4.2.1).

Beispiel:

A - P + C wobei A und C Variablenname und P Prozedurname

In diesem Ausdruck wird P durchlaufen und der Wert von P von A abgezogen. Anschließend wird C addiert.
Soll eine Prozedur mit bestimmten Anfangswerten durchlaufen werden, so kann dies dadurch geschehen, daß der Prozedur in folgendem Ausdruck ein Wert zugewiesen wird:

⟨expression⟩ =: P

Dabei wird P der Wert des linksstehenden Ausdrucks als Prozedurwert zugewiesen. Ferner wird die Prozedur durchlaufen. Auf diesem Prozedurwert kann man sich dann innerhalb einer Prozedur beziehen, indem man vor den Prozedurnamen das Sonderzeichen & schreibt: Durch &P wird innerhalb von P das Wort angesprochen, das den Prozedurwert enthält. Dabei wird P jedoch nicht durchlaufen, im Unterschied zu einem rekursiven Prozeduraufruf mit dem Namen P.

Hinweis:
& P darf nur innerhalb einer Prozedur P verwendet werden.

Beispiel:

```
'PROGRAM' K WERT
'PROCEDURE' A
'NEW' C = 3, D
'BEGIN'
 statement list
C + & A = D
 statement list
'END'
'BEGIN'
 13 =: A
'END'
```

Erklärung:

Im Hauptprogramm wird in der Anweisung '13 = A' die Prozedur A aufgerufen, wobei ihr der Parameter 13 als Wert zugewiesen wird. Innerhalb von A wird der Wert 13 dann durch &A angesprochen.

Auf ähnliche Weise wie Werte in die Prozedur eingeführt werden, können aus der Prozedur Werte an die Umgebung übergeben werden: Ist innerhalb einer Prozedur P dem Wort, das den Prozedurwert enthält, durch ⟨expression⟩ =: &P der Wert eines Ausdrucks zugewiesen worden, so kann dieser Wert außerhalb der Prozedur durch den Prozedurnamen P angesprochen werden.
Insbesondere wird bei einer Anweisung P =: A der Variablen A der Prozedurwert übergeben, nachdem P durchlaufen wurde. Mit Hilfe rekursiver Aufrufe kann man erreichen, daß eine Prozedur ihren eigenen Anfangswert schrittweise verändert. Das Abbrechen der rekursiven Schleife ist etwa mit der IF-Anweisung zu erreichen: IF &P ⟨relational operator⟩ ⟨expression⟩ 'THEN' ...
Bisher wurde nur der Fall behandelt, daß in eine Prozedur nur einzelne Werte hereingegeben werden oder einzelne Werte herausgegeben werden. Da man nun die Möglichkeit hat, einen Prozedurnamen in einer Wertzuweisung auch Adressen zuzuweisen, ist es möglich, dem Prozedurwert Adressen von Speicherbereichen zu übergeben, auf deren Inhalte dann innerhalb der Prozedur durch indirekte Adressierung Bezug genommen werden kann.
Beispiel:

```
        @A =: P
```

Durch diesen Ausdruck wird P die Adresse von A als Prozedurwert zugewiesen und P durchlaufen.

Beispiel:

```
'PROGRAM' Name
'NEW' A [1] = 3, 4
'PROCEDURE' ADR
'NEW' C, D, E
'BEGIN'
& ADR =: C ;
C  +4 =: D ;
[D] +2 =: E
'END'
'BEGIN'
@A =: ADR
'END'
```

Erklärung:

Im Hauptprogramm wird @ A der Prozedur ADR als Prozedurwert zugewiesen. In ADR wird diese Adresse nach C gebracht. Dann wird die um 1 erhöhte Adresse nach D gebracht. Anschließend wird der Inhalt der um 1 erhöhten Adresse um 2 erhöht und nach E gebracht; d.h. in E steht dann der Wert 6.
In dieser Weise hätte innerhalb der Prozedur jedes Wort des durch A adressierten Bereichs angesprochen werden können.

Eine Übergabe von Werten an und von Prozeduren braucht aber nicht unbedingt über den Prozedurwert erfolgen. Vielmehr wäre es ebenso denkbar, daß Werte mittels globaler NEW-Variablen übergeben werden, da durch 'NEW' deklarierte Speicherbereiche während des Programmablaufs in untergeordnetenProzeduren bestehen bleiben und nach erfolgter Deklaration in jeder Prozedur angesprochen werden können, sofern dort über den betreffenden Variablennamen nicht anders verfügt wird.

Beispiel:

```
'PROGRAM'SIN
'NEW' A, B
'PROCEDURE' ZAHL
'NEW' D, E, F
'BEGIN'
 statement list;
A + 5 =: D ;
 statement list
'END'
'BEGIN'
 statement list
7 =: A;
ZAHL ;
 statement list
'END'
```

Erklärung

Mittels der durch 'NEW' deklarierten Variablen A wird an die Prozedur ZAHL die Dezimalzahl 7 übergeben.

Auch auf die Adressen von NEW-Variablen kann man sich in der Prozedur stets beziehen:

4. Standardprozeduren zur Speicherverwaltung

In SCALLOP stehen bestimmte Standardprozeduren zur Verwaltung des noch freien Arbeitsspeicherbereichs zur Verfügung. Derartige Standardprozeduren erweisen sich insbesondere bei rekursiven Prozeduren als sinnvoll, wenn etwa bei einem Prozedurdurchlauf die Werte bestimmter Variablen verändert werden, während bei erneutem Durchlauf die alten Werte dieser Variablen benötigt werden. In diesem Fall müssen die alten Werte vorübergehend in einem Speicherbereich gerettet werden.

STACK:

Durch Aufruf des Standardunterprogrammes STACK ist es möglich, sukzessiv Werte von Ausdrücken in den noch nicht belegten Arbeitsspeicherbereich zu übertragen. Die zu übertragenden Werte werden rechtsbündig in Arbeitsspeicherworte gelegt und zwar so, daß dem Wort des unmittelbar vorher übertragenen Wertes im Arbeitsspeicher unmittelbar folgt.

Der Aufruf, mit dem eine solche Übertragung möglich ist, hat die Form:

⟨expression⟩ =: STACK

Durch einen Aufruf von STACK kann man auch erreichen, daß der zuletzt übertragene Wert an einer Variablen zugewiesen wird. Dieser Aufruf hat die Form:

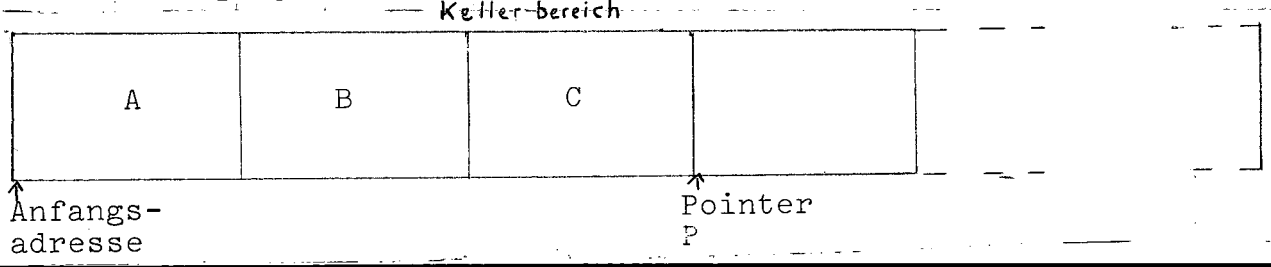
STACK =: ⟨variable⟩

Einen Speicherbereich, der in der Weise verwaltet wird, daß der zuletzt in den Bereich übertragene Wert zuerst wieder aus dem Speicher geholt wird, nennt man auch Keller, Kellerbereich (vgl. stack, pushdown store). Es ist zu beachten, daß in der vorliegenden Version das aus dem Keller geholte Wort nicht gelöscht wird.

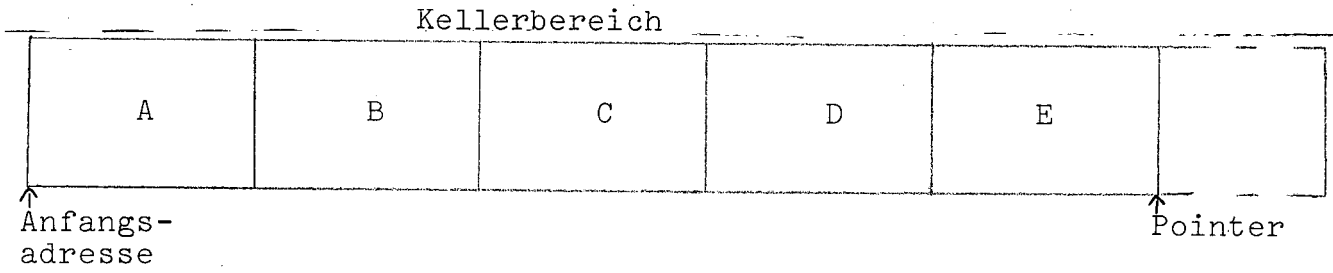
Im einzelnen geschieht die Verwaltung eines Kellers in der Weise, daß ein Zeiger (Pointer), der zunächst auf die Anfangsadresse des Speicherbereichs weist, nach Übertragung einer festen Anzahl von Wortinhalten weiterrückt. Bei Übertragung aus dem Keller wird der Zeiger um die Anzahl der übertragenen Wortinhalte zurückgesetzt. Der Zeiger weist also immer auf das erste freie Wort im Arbeitsspeicher.

Beispiel:

Vor Übertragung zweier Werte D, E:



Nach Übertragung zweier Worte D, E



Durch die Zuweisung von Werten an STACK erhöht sich der Wert des Pointers um ein Wort. Bei Übertragung eines Wertes aus dem Keller erniedrigt sich der Pointer um ein Wort. Durch Aufruf von STACK kann nur ein Wortinhalt zur Zeit übertragen werden. Die Übertragung einer Folge von Werten kann etwa geschehen unter Verwendung der Schleifenanweisung.

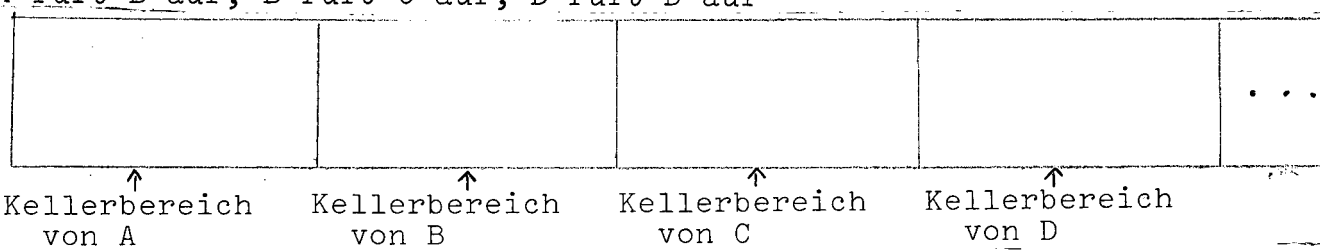
Beispiel:

```
- 1 =: J
'DO' 3'TIMES'  A  [J+1 =: J]  =: STACK
```

Durch diese beiden Anweisungen würden, beginnend beim durch den NEW-Namen A bezeichneten Arbeitsspeicherwort, die Inhalte von drei Arbeitsspeicherworten sukzessiv in den Keller übertragen werden. Tritt im Programm eine Aufrufschachtelung auf und wird für mehrere Prozeduren ein Kellerbereich aufgeklappt, so wird der Kellerbereich der aufgerufenen Prozedur unmittelbar an den der aufrufenden Prozedur gelegt.

Beispiel:

A ruft B auf, B ruft C auf, D ruft D auf



Bei Verlassen einer Prozedur wird der Pointer wieder auf den Anfangswert des Kellerbereichs der betreffenden Prozedur zurückgesetzt.

POINTER:

Durch Aufruf der Standardprozedur POINTER ist es möglich, den Zeiger herauf- und herabzusetzen, sowie seinen Wert zu erfragen. Die dazu notwendigen Anweisungen haben die Form:

⟨expression⟩ =: POINTER
POINTER =:⟨variable⟩

Aufgrund der ersten Anweisung wird dem Pointer der Wert des links stehenden Ausdrucks zugewiesen. Durch POINTER + number =: POINTER etwa könnte man den Pointer um einen bestimmten Zahlenwert erhöhen. Durch die zweite der obigen beiden Anweisungen kann der Wert des Pointers einer Variablen zugewiesen werden. Dabei wird der Pointer selbst aber nicht verändert.

MEMORY:

Mit Hilfe der Standardprozedur MEMORY ist es möglich, die Größe des noch freien Arbeitsspeichers,d.h. die Anzahl der noch freien Arbeitsspeicherworte abzufragen.

Beispiele:

```
MEMORY =: A
```

Der Variablen A wird die Größe des noch freien Arbeitsspeichers als Wert zugewiesen.

```
'IF' MEMORY 'LESS' K
     'THEN'
     'GOTO' NOT ENOUGH MEMORY
```

In Abhängigkeit von der Größe des noch freien Arbeitsspeichers wird die Ablauffolge der Anweisungen geändert.

Aufgrund der Möglichkeit der Abfrage des noch freien Arbeitsspeicherbereichs ist es möglich zu verhindern, daß der Keller den Arbeitsspeicher sprengt und die Maschine das Programm abbricht. Die Abfrage ist besonders dann sinnvoll, wenn

- die Anzahl der in den Kellerbereich zu übertragenden Werte sehr groß ist oder
- wenn die Anzahl der in einer Aufrufschachtelung auftretenden Prozeduren sehr groß wird. Wenn jede der aufgerufenen Prozeduren einen Keller definiert, so wird die Größe des belegten Arbeitsspreichers immer unübersichtlicher.

Fehlermeldungen

Fehler Nr.	Bedeutung
E 1	kein Identifier nach (
E 2	) fehlt
E 3	kein Identifier nach
E 4	kein + oder - nach .
E 5	unzulässiger Charakter
E 6	fehlerhaftes Wortsymbol
E 7	Dezimalzahl zu lang
E 8	Basiszahl zu lang
E 9	String zu lang
E 10	String verstümmelt
E 31	'PROGRAM' oder 'PROCEDURE' erwartet
E 32	'NEW', 'FOREWARD', 'PROCEDURE' oder 'BEGIN' erwartet
E 33	'PROCEDURE' oder 'BEGIN' erwartet
E 34	'PROCEDURE-IDENTIFIER' fehlt (kein Eintrag)
E 35	'PROGRAM-IDENTIFIER' fehlt (kein Eintrag)
E 36	New-Deklaration beginnt falsch
E 37	Syntax-Fehler in der New-Deklaration an der bezeichneten Stelle
E 38	Adresse in der New-Deklaration nicht erklärt
E 39	Syntax-Fehler in der Foreward-Deklaration
E 40	Procedure-Name doppelt
E 41	Procedure Body endet nicht mit 'END'
E 42	Label doppelt
E 43	Goto Label fehlt
E 44	'TIMES' erwartet
E 45	'OF' erwartet
E 46	'OTHERWISE' erwartet
E 47	'ELSE' oder 'THEN' erwartet
E 48	unzulässiger Operand
E 49	auf 'OVERFLOW' folgt kein Name
E 50	'OVERFLOW' folgt falsch

Fehler Nr.	Bedeutung
E 51	Operator erwartet
E 52	Operand falsch
E 53	unzulässiger Operand im Subscript
E 54	unzulässiger Operator
E 55	Label Identifier erwartet
E 56	Externer Identifier erwartet
E 57	doppelt definiert External
E 58	Komma erwartet im External
E 59	(Prozedur) ungleich der laufenden Prozedur