DEUTSCHES ELEKTRONEN-SYNCHROTRON DESY

DESY DV-76/02
Dezember 1976

SIMPL11, eine einfache Implementierungssprache für PDP11-Rechner

von

Guido Pfeiffer

NOTKESTRASSE 85 · 2 HAMBURG 52

To be sure that your preprints are promptly included in the
HIGH ENERGY PHYSICS INDEX ,
send them to the following address (if possible by air mail) :

DESY
Bibliothek
Notkestraße 85
2 Hamburg 52
Germany

SIMPL11, eine einfache Implementierungssprache für PDP11-Rechner

von

Guido Pfeiffer

Deutsches Elektronen-Synchrotron DESY, Hamburg

INHALTSVERZEICHNIS

Zusammenfassung

Im Rahmen eines Projektes zur Auswertung von Röntgenbildern wurde die niedere Programmiersprache SIMPL 11 entwickelt. Sie ist eine maschinennahe Sprache mit Eigenschaften, die sonst nur in höheren problemorientierten Programmiersprachen zu finden sind. Die Maschinennähe gewährleistet den Zugriff zu allen Hardware Komponenten, dafür müssen jedoch Eigenheiten der PDP11 Hardware-Konzeption explizit vom Benutzer berücksichtigt werden.

Der vorliegende Bericht enthält eine informelle Beschreibung der SIMPL11-Sprache. Das Sprachkonzept wird ausführlich an Beispielen diskutiert. Es enthält als wichtigste Eigenschaften

- den Begriff der Datentypen, orientiert an der PDP11 Speicherstruktur,
- Ausdrücke zur Formulierung von Rechner-Anweisungen,
- Kontrollstrukturen zur Bildung von Schleifen und bedingten Verzweigungen,
- Benutzerfreundliche Ein-Ausgabe-Anweisungen für die Kommunikation zwischen Benutzer und Maschine, und
- Macro-Definitionsmöglichkeiten, geborgt vom Macro-11 Assembler

Im Vergleich zur Assembler Programmierung lassen sich Programme in SIMPL11 wesentlich kürzer und problemorientierter formulieren, ohne daß Einbußen an Effiziez hingenommen werden müssen. Der Anwendungsbereich umfaßt alle Aufgaben, in denen maschinennahe Programmierung unerläßlich ist, bzw. Probleme, für die höhere Programmiersprachen keine geeigneten Hilfsmittel bieten. Sinnvolle Anwendungen etwa sind System- und Compiler-Implementierung, Textverarbeitung, sowie Prozeßrechneraufgaben insbesondere bei zeitkritischen Datenerfassungs- und Auswertungs-Anwendungen, oder sogar rein numerische Anwendungen, wenn verfügbare höhere Programmiersprachen (z.B. FORTRAN) zu langsam sind.

1. Einleitung

1.1 Übersicht

SIMPL11 (Simple IMPlementation Language for PDP11 Computer) ist eine Programmiersprache für die Familie der PDP11-Rechner zur Erleichterung des maschinennahen Programmierens. SIMPL11 gehört zu einer Klasse von Sprachen, die man allgemein als Niedere Programmiersprachen oder System-Implementierungs-Sprachen (SIL) bezeichnet. Diese nehmen eine Mittelstellung ein zwischen maschinenabhängigen reinen Assemblersprachen und maschinenunabhängigen problemorientierten höheren Programmiersprachen (z.B. FORTRAN, ALGOL), indem sie Konzepte beider Sprachklassen in sich vereinigen. Niedere Programmiersprachen enthalten einerseits die Strukturierungsmöglichkeiten von höheren Programmiersprachen (z.B. Datentypen, Ausdrücke, Schleifen, Bedingungen etc.), erlauben andererseits jedoch eine ähnlich effiziente Ausnutzung spezieller Rechner-Eigenschaften, wie Assemblersprachen.

Begonnen hat die Entwicklung niederer Programmiersprachen mit PL360, einer Sprache von N. Wirth für die Familie der IBM/360-Maschinen.[1] Inzwischen wurden für viele Rechnertypen Sprachen dieser Art entwickelt, so auch für die PDP11, z.B. PL11 [2] oder LIL [3]. Ausführliche Literaturhinweise zu veröffentlichten niederen Programmiersprachen sind in Ref.(4) zu finden. Die meisten dieser Sprachen sind - ebenso wie das hier beschriebene SIMPL11 - in ihrer Konzeption von PL360 beeinflußt. Ein eingehender Vergleich zwischen den verschiedenen Arten niederer Programmiersprachen ist in einem Artikel von Santo zu finden.[5]

1.2 Ziele

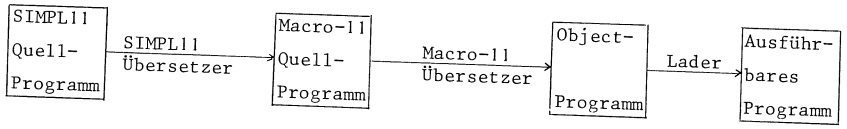
Die Sprache SIMPL11 ist als Teil eines größeren Projektes für die Computer-Angiodensitometrie entstanden, das in einer Zusammenarbeit zwischen dem Deutschen Elektronen-Synchrotron DESY und dem Universitätskrankenhaus Eppendorf (UKE) durchgeführt wird.[6] In diesem Projekt wird unter anderem eine Dialogsprache, bestehend aus einem Interpreter und einem Compiler, für die interaktive Auswertung von Röntgenbildern entwickelt.[7] Die Implementierung des Dialog-Systems sollte in einer hierfür geeigneten Sprache durchgeführt werden. Zur Auswahl standen in den DEC-Betriebssystemen FORTRAN und der Macro-11-Assembler. Während FORTRAN für System-Implementierung keine besonders geeigneten Hilfsmittel bietet, wird die Programmierung in Assembler im Rahmen der Diskussion über "Strukturiertes

Programmieren" allgemein als ungeeignet betrachtet. Das zu dieser Zeit bereits bekannte PL11 dagegen besitzt eine Reihe von unerwünschten Nachteilen und Einschränkungen[4] und war zudem nicht im gewünschten Betriebssystem RT11 verfügbar. Deshalb wurde beschlossen, mit möglichst geringem Aufwand eine einfache Implementierungssprache zu entwickeln mit folgenden Forderungen:

- Möglichst schnelle Verfügbarkeit eines operationellen Compilers
- Entwicklung und Test des Compilers zusammen mit der Implementierung des Dialog-Systems
- Hohe Effizienz des generierten Codes, vergleichbar der Assembler-Programmierung
- Uneingeschränkte Ausnutzung der PDP11-Hardware-Eigenschaften und vorhandener Betriebssystemfunktionen
- Konsistente Einbettung ins Betriebssystem.

1.3 Technik

Um der erstgenannten der obigen Forderungen zu entsprechen, wurde SIMPL11 als Preprocessor implementiert. Hierdurch ließ sich in kürzester Zeit ein funktionsfähiger Compiler entwickeln. Der erzeugte Code ist ein Macro-11-Quellprogramm, das vom Assembler in verschiebbaren Object-Code und vom Lader in ausführbaren Maschinencode umgesetzt wird. Ein SIMPL11-Programm muß also bis zur Ausführung folgende Übersetzungsschritte durchlaufen:



Die Preprocessor-Technik erlaubt es, Teile des nachfolgenden Assemblers in SIMPL11 ohne besonderen Aufwand mitzubenutzen. Hiervon wird insofern Gebrauch gemacht, als syntaktische Konstruktionen, die im Assembler bereits vorhanden sind, während der Übersetzung durch SIMPL11 nicht berücksichtigt werden. In der folgenden Sprachbeschreibung wird auf diese Fälle besonders hingewiesen.

Die wesentlichen Vorteile, die sich aus der Preprocessor-Technik ergeben, lassen sich wie folgt zusammenfassen:

- Schnelle und einfache Implementierung
- Einfache Kontrolle des erzeugten Codes
- Möglichkeit der beliebigen Einfügung von Assembler-Anweisungen
- Benutzbarkeit der Macro-Technik aus dem Assembler.

Diesen offenkundigen Vorteilen stehen Nachteile gegenüber, die typisch für alle Preprocessoren sind:

- Der Übersetzungsvorgang muß zweimal durchgeführt werden. Ein Programm ist erst dann syntaktisch korrekt, wenn es beide Übersetzer fehlerlos durchlaufen hat.
- Der Benutzer muß die Fehlermeldungen aus zwei verschiedenen und voneinander unabhängigen Übersetzern interpretieren können. Dies bedeutet in der Praxis, daß zumindest einige Kenntnisse des Macro-11 Assemblers in jedem Falle erforderlich sind.

1.4 Betriebssystem

Die Implementierung von SIMPL11 wurde im Betriebssystem RT11 durchgeführt. Die angegebenen Beispiele beziehen sich - soweit Betriebssystemfunktionen verwendet werden - auf RT11.

Die Benutzung von SIMPL11 ist jedoch auch in allen anderen PDP11-Betriebssystemen, die den Macro-11 Assembler unterstützen, möglich, sofern die Ein/Ausgabe-Macros im Compiler entsprechend abgeändert werden.

2. Allgemeine Grundlagen

2.1 Einführendes Beispiel

Zur Demonstration einiger typischer Sprachelemente von SIMPL11 sei ein einfaches Programmbeispiel angegeben. Das erzeugte Assembler Programm ist im Anhang D.1) enthalten. Die ersten 25 Glieder der Fibonacci-Folge sollen in einem Array berechnet werden, und die Ausgabe soll in Zeilen zu je 5 Zahlen erfolgen. Die Zahlen F_u der Fibonacci-Folge sind definiert als:

$$F_{u+1} = F_u + F_{u-1} \quad \text{für } u > \emptyset$$

$$F_\emptyset = \emptyset$$

$$F_1 = 1$$

```
        // SIMPL11 PROGRAMM BEISPIEL:
        // BERECHNUNG UND AUSGABE DER FIBONACCI ZAHLEN

array   word FIBON[25.];                        // DEKLARATIONEN
word    IX=2;
lit     TAB=11;

FIBO:   read '1. ZAHL = ', %FIBON[1]            // ZWEI ANFANGSZAHLEN EINLESEN
        read '2. ZAHL = ', %FIBON[2],

        repeat                                  // FIBONACCI ZAHLEN BERECHNEN
        begin
          FIBON[IX]=>push+FIBON[IX=>R0-]
          pop=>FIBON[IX+]
        end until IX eq size(FIBON)

        0=>IX
LOOP:   for 5=>R0 do                            // RESULTAT AUSGEBEN
        begin
          if IX lt size(FIBON) then print /%FIBON[IX+], TAB else so ENDE
        end
        print; so LOOP

ENDE:   .END FIBO
```

```
 R FIBO
1. ZAHL = 0
2. ZAHL = 1
0       1       1       2       3
5       8       13      21      34
55      89      144     233     377
610     987     1597    2584    4181
6765    10946   17711   28657   -19168
```

Das Programm besteht aus folgenden Schritten:

1. Deklaration des Arrays FIBON, bestehend aus 25 Worten und des Index IX . IX wird mit 2 initialisiert.

2. Die ersten beiden Zahlen des Arrays FIBON sind vom Benutzer einzugeben. Die Strings in den READ-Anweisungen werden auf dem Terminal ausgedruckt. Das Kontrollzeichen '%' bedeutet, daß die einzulesende Zahl dezimal zu interpretieren ist.

3. Die bedingte Schleife REPEAT ... UNTIL berechnet die Fibonacci-Zahlen gemäß der Definition und füllt den Array FIBON.

4. Für die Berechnung wird der Stack als temporärer Speicher benutzt. PUSH und POP sind Stack Operanden. Mit PUSH wird der Wert FIBON [IX] auf den Stack geladen. Die folgende Addition addiert den Wert FIBON [IX→RØ-] auf den obersten Stack Eintrag. Dabei wird zuerst der Index Ausdruck berechnet. [IX→RØ-] repräsentiert die Gleichung RØ = IX-1. Die neu berechnete Fibonacci-Zahl befindet sich auf dem Stack. POP → FIBON [IX+] entfernt das oberste Stack-Element mit Index IX, wobei dieser zuvor incrementiert wurde.

5. Die Ausführung der Schleife ist beendet, wenn Index IX gleich der Array-Größe SIZE(FIBON), also = 25 geworden ist.

6. Die Schleife FOR ... DO ... druckt jeweils 5 Zahlen des Arrays in eine Zeile. Das Ausdrucken geschieht mit dem Kommando PRINT ... . Das Zeichen '/' verhindert die Ausgabe einer neuen Zeile.

7. Nach Beendigung der FOR-Schleife gibt das Kommando PRINT (ohne Argumente) eine neue Zeile aus.

Das Beispiel enthält folgende Sprachkomponenten:

- Deklaration von Daten
- Schleifen
- Bedingte Ausdrücke
- Dateneingabe, Datenausgabe

Die Kleinschreibung im Programmausdruck dient lediglich dem Zweck, reservierte Symbole von SIMPL11 besonders hervorzuheben!

2.2 Syntax-Beschreibung

Die Syntax von SIMPL11 wird in diesem Bericht i.a. verbal beschrieben und an Beispielen demonstriert. Einzelne Sprachkonzepte sind der Einfachheit halber in der üblichen Backus Naur Form (BNF) angegeben. Folgende Zusatzregeln werden verwandt:

i) Einfache Optionen sind durch Klammerungen der Art [...] beschrieben.

ii) End-Symbole und End-Zeichen sind unterstrichen.

Eine vollständige BNF-Syntax Beschreibung ist im Anhang F.1) enthalten.

2.3 Vokabular

Das SIMPL11-Vokabular besteht aus:

- Buchstaben: A,..., Z
- Zahlen: 0,..., 9
- Reservierten Symbolen (End Symbolen)
- Spezialzeichen
- Identifikatoren (Namen)

2.3.1 Reservierte Symbole

Folgende Symbole haben in SIMPL11 eine feste Bedeutung:

ABS	ABSF	ACO	AC1	AC2	AC3	AC4
AC5	ADC	ARRAY	ASH	ASHC	ASL	ASR
BEGIN	BIC	BIS	BIT	BYTE	CALL	CARRY
COM	DO	DREG	DSTACK	DUMP	DUMPS	ELSE
END	ENDM	EQ	FALSE	FOR	GE	GO
GT	HIGHER	HISAME	IF	JSR	LDEXP	LDFPS
LE	LIMIT	LIST	LIT	LOSAME	LOWER	LT
MACRO	MODF	NE	NEG	NLIST	OFLOW	POP
PRINT	PUSH	RADIX	READ	REAL	REF	REPEAT
RETURN	ROL	ROR	RO	R1	R2	R3
R4	R5	SBC	SIZE	SP	STACK	STEXP
STFPS	STRING	STST	SWAB	SXT	THEN	TOP
TRAP	TRUE	UNTIL	WHILE	WORD	XOR	

Diese Symbole dürfen nicht als benutzereigene Namen verwendet werden. In den Beispielen sind Begrenzungssymbole (IF, THEN, ELSE, BEGIN...) zur besseren Lesbarkeit unterstrichen, in Programm-Ausdrucken sind alle reservierten Symbole durch Kleinschreibung hervorgehoben.

2.3.2 Spezialzeichen

Folgende Spezialzeichen werden benutzt:

```
+    -    *    /    //   .    ,
;    :    (    )    [    ]    >
<>   %    %%   =    =>   !    &
^    $    '    "    ..   #    @
```

2.3.3 Identifikatoren

Identifikatoren sind Namen zur Bezeichnung von Variablen, Marken (Labels) und Registern. Namen beginnen mit einem Buchstaben. Es darf eine beliebig lange Kombination von Buchstaben und Zahlen folgen, jedoch werden nur die ersten 6 Zeichen beachtet.

Marken (Labels):
Marken dienen als Referenz für einen Sprung oder Subroutinen Aufruf. Marken dürfen vor jedem ausführbaren Kommando stehen. Sie sind gekennzeichnet durch Namen, gefolgt von einem Doppelpunkt:

Beispiele:

```
ABC: <COMMAND>

ABC: DEF: <COMMAND>

ABC: DEF:
LAB1: LAB2: <COMMAND>
```

Variable:
Variable sind vom Benutzer gewählte Namen zur Bezeichnung von Objekten mit einem Datentyp. Reservierte Symbole sind verboten.

Register:
Eine Reihe von Identifikatoren sind in SIMPL11 vordefiniert und haben eine feste Bedeutung:

a) Register: RØ, R1, R2, R3, ..., R5, SP
   SP = Register 6 = Stack Pointer

b) Float-Register: AC0, AC1, ..., AC5

c) Stack Operanden: PUSH, POP, TOP

2.4 RADIX Kontrolle

Eine oktale oder dezimale Basis des Zahlensystems kann durch eine RADIX-Anweisung angegeben werden (vgl. auch RADIX-Direktive im MACRO-11 Assembler). Es gelten die Regeln:

a) Umschaltung zur Dezimal-Basis durch:
   . RADIX 10

b) Umschaltung zur Oktal-Basis durch:
   . RADIX

c) Als Initial-Zustand ohne RADIX-Anweisung wird oktale Basis angenommen.

Alle Zahlenangaben innerhalb eines Programms werden gemäß der jeweils eingestellten Zahlenbasis interpretiert. Diese kann jedoch für einzelne Zahlenwerte lokal übergangen werden (vgl. 2.5.1 Numerische Konstante).

2.5 Konstante

Konstanten werden unterteilt in numerische-, string-, und RAD50 Konstanten.

2.5.1 Numerische Konstante

Numerische Konstante werden nach folgender Syntax - ohne Berücksichtigung des Vorzeichens - gebildet:

```
<NUM CONST>     := <ZAHL> / <DEC ZAHL> / <OCT ZAHL>
<DEC ZAHL>      := <ZAHL>. / ^D<ZAHL>
<OCT ZAHL>      := ^O<ZAHL>
<ZAHL>          := <DIGIT> / <ZAHL><DIGIT>
<DIGIT>         := 0/1/2/3/4/5/6/7/8/9
```

BEISPIELE:

1234 1234. ^D1234 ^O1234

Falls die Basis des Zahlensystems nicht explizit durch ↑D, ↑O, . angegeben ist, richtet sich der interne Wert nach der in einer RADIX-Anweisung angegebenen Basis. Der Wert einer Zahl muß innerhalb des durch ein 16 Bit-Wort gegebenen Bereichs von (-32768, 32767) liegen. Numerische Konstante sind nur für Worte und Bytes zulässig.

2.5.2 String Konstante

String Konstanten existieren für Bytes und Worte. Sie sind gekennzeichnet durch die Zeichen ' bzw. ", gefolgt von einem bzw. zwei Zeichen.

Beispiele:

Byte String: 'A '?
Word String: "AB "X+

2.5.3 RAD50-Konstante

Die RAD50-Darstellung erlaubt, Konversion der Charakter in ein Zahlensystem mit der Basis 50 durch 3 Zeichen in einem 16 bit-Wort zu speichern (für Einzelheiten vgl. DEC-Handbücher). Folgende Zeichen sind in der RAD50-Darstellung erlaubt:

A...Z Ø...9 $.

RAD50-Konstanten sind für die Typen Word und Real zugelassen. Die Kennzeichnung erfolgt durch das Zeichen '$'.

Beispiele:

WORD (maximal 3 Zeichen): $ABC $RAD
REAL (maximal 6 Zeichen): $ABCD $RAD50 $ABCDEF

RAD50-Konstante sind nur in Deklarationen zulässig, nicht innerhalb von Ausdrücken!

2.5.4 Numerische reelle Konstante

In SIMPL11 können innerhalb von Deklarationen reelle numerische Konstanten verwendet werden (nicht in Ausdrücken!).

Beispiele:

1.414 123.45 E-12 0.00123

Die Syntax-Analyse wird jedoch nicht in SIMPL11, sondern erst im nachgeschalteten MARCO-11 Assembler durchgeführt. Dieser enthält bereits die syntaktische Analyse reeller Zahlen, so daß diese Aufgabe für SIMPL11 überflüssig ist.
Die BNF-Syntax für numerisch reelle Konstanten ist im Anhang F.3) angegeben.

2.5.5 Zusammenfassung der Konstanten

In Tab.1) sind die erlaubten Konstanten und ihre Zuordnung zu Datentypen in Beispielen zusammengestellt:

Tab.1: Zuordnung von Konstanten zu Datentypen

	BYTE	WORD	REAL
NUMERISCH BEISPIELE	* 123	* 123 123. ^D123 ^O123	(*) 12.34E-10
STRING BEISPIELE	* 'X	* "XY	
RAD50 BEISPIELE		* $ABC	* $ABCDEF

2.6 Kommentar

Zwei aufeinanderfolgende Schrägstriche // definieren den Beginn eines Kommentars. Der Kommentar wird durch das Ende der Zeile abgeschlossen.

Beispiel:

```
// Kommentar-Zeile
```


3. Daten

Ein zentraler Begriff höherer Programmiersprachen ist das Konzept der Datentypen. Ein Datentyp definiert eine Klasse von Werten, die eine Variable annahmen kann. Während in höheren Programmiersprachen wie FORTRAN oder ALGOL die Datentypen problemorientiert sind, besitzt SIMPL11 als niedere Programmiersprache Datentypen, die der Speicherstruktur der PDP11 angepaßt sind. Generell sind auf Maschinenebene zu unterscheiden:

- Speicherbezogenene Daten (Bytes, Worte)
- Register (allgemeine Register, Float-Register)

Zusätzlich gibt es Objekte, die

- zur Übersetzungszeit bekannte Daten definieren. (Literale)

Nach diesen Objekten sind die Datentypen bei SIMPL11 orientiert. Es gibt die einfachen Typen BYTE, WORD, REAL, LIT und außerdem die strukturierten Typen ARRAY, STACK.

3.1 Datentypen BYTE, WORD, REAL, LIT

Ein BYTE ist die kleinste adressierbare Einheit der PDP11. WORD's bestehen aus 2 Bytes und REAL's aus 4 Bytes bzw. 2 Worten. Das erste Byte von WORD- oder REAL-Objekten muß eine gerade Adresse haben, und alle Bytes müssen aufeinanderfolgen. Für Objekte der Art BYTE, WORD, REAL muß explizit Speicherplatz reserviert werden.

Der Datentyp LIT (= Literal) bezeichnet Übersetzerkonstante, d.h. Daten, die nur zur Übersetzungszeit definiert sind und für die kein Speicherplatz explizit reserviert werden muß. Literale werden i.a. benutzt, um Byte- oder Word-Konstanten (vgl. Tab.1) durch Identifikatoren zu bezeichnen. Intern sind Literale wieder als BYTE oder WORD, repräsentiert. Durch Benutzung von Literalen können Programme weitgehend von Konstanten freigehalten werden.

Die zulässigen Operationen für die Datentypen sind durch die Hardware der PDP11 vorgegeben. In Tab.2 sind die grundsätzlichen Operationen zusammengefaßt, ohne an dieser Stelle auf maschinenabhängige Details einzugehen.

Tab.2: Datentypen und zugehörige Operationen (Klammern bedeuten, daß die Zulässigkeit von der speziellen Operation abhängt)

	Arithmetische Operationen	Logische Operation	Zuweisung	Bit-Test	Bit-Schieben
BYTE		+	+	+	(+)
WORD	+	+	+	+	+
REAL	+	+	+		
LIT	+	+	+	+	(+)

Float-Register

Den Float-Registern AC0, AC1, ... AC5 ist fest der Datentyp REAL zugeordnet.

3.2 Datentypen ARRAY, STACK

Ein ARRAY besteht aus einer festen vorgegebenen Anzahl von Einzelkomponenten, die vom Typ BYTE, WORD oder REAL sein müssen. In BYTE- bzw. WORD-Arrays sind auch Literale zulässig; diese nehmen dann konkret den Typ BYTE bzw. WORD an. Der Zugriff zu den Komponenten erfolgt durch Indizes. Die Zählung von Indices beginnt mit 1. ARRAYs sind in SIMPL11 grundsätzlich linear (eine Dimension).

Ein STACK besteht aus einem festen, durch Deklaration definierten Speicherbereich. Die Stack-Komponenten können jeden der einfachen Datentypen annehmen. Die Anzahl der vorhandenen Komponenten ist dynamisch veränderlich und wird durch einen vom Benutzer zu definierenden STACK-Pointer angegeben.

Für Arrays ist die Operation Selektion definiert (vgl. 5.1.2). Für Stacks sind implizit die Operationen PUSH, POP, TOP definiert. Diese bedeuten wie üblich, ein Objekt auf den Stack zu laden oder vom Stack zu entfernen. TOP bezeichnet ein Objekt des Stacks, ohne diesen zu verändern.

3.3 Datentyp-Anpassung

Die Objekte der Art

- Allgemeine Register (R0, ..., R5, SP)
- Indirekte Register-Adressierung (vgl. 5.1.4)
- Literale
- numerische Konstante

können in Abhängigkeit vom Kontext innerhalb eines Ausdrucks einen der beiden Datentypen BYTE oder WORD annehmen. Allgemein gelten folgende Regeln:

i) Hat in Operationen mit zwei Operanden ein Operand einen veränderlichen Datentyp, so gilt der Typ des zweiten Operanden.

ii) Sind die Typen beider Operanden veränderlich, so gilt implizit der Typ WORD.

iii) In Operationen mit einen Operanden gilt implizit der Typ WORD.

iv) Zusatzregel:
Indirekte Register Operanden können zusätzlich auch den Typ REAL annehmen (vgl. 5.1.4).

4. Deklarationen

Alle Variablen müssen deklariert werden. Die Deklaration weist der Variablen einen Datentyp zu und reserviert evtl. benötigten Speicherplatz. Der Datentyp ist innerhalb des gesamten Programms gültig und kann nicht geändert werden. In der Deklaration können den Variablen gleichzeitig Anfangswerte zugewiesen werden.

4.1 Einfache Objekte (BYTE, WORD, REAL, LIT)

Die Deklaration erfolgt nach der Syntax:

```
<TYPE><OBJECT> [,<OBJECT 1>, <OBJECT 2>, ...];

<TYPE>          := BYTE / WORD / REAL / LIT
<OBJECT>        := <IDENTIFIER> [=<INITIAL>] (*)
<INITIAL>       := <KONSTANTE> / <MACRO ASSEBMLER EXPR> /

  (*) FUER <TYPE> = LIT IST EINE INITIALISIERUNG OBLIGATORISCH!
```

BEISPIELE:

```
LIT     LSR=1000., LPT=LSR-^O100, LLA=62, LG=10., LST=LG-1,
        COLON=':, ASSIGN="=>, LZ="AB, TRU=0, FALS=1;

BYTE    A,B,C,D='Y, E=5;

WORD    IX, X1=0, X2=128., X3=120+LLA*2, AB=.-2,
        XY="EF, NM=$ABC, PT, ADR;

REAL    ARL, XR1=0, XR2=1234.56, XR3=123.4E21,
        DRL=0.0012, NAM=$RADIX, ALF;
```

Bei Deklarationen sind die folgenden Regeln zu beachten:

- Eine Deklaration darf über mehrere Zeilen verteilt sein. Sie wird grundsätzlich durch ein Semikolon ';' abgeschlossen.

- Der Typ der Konstanten muß mit dem Typ der Deklaration übereinstimmen.
- Literale müssen initialisiert werden. Konstante in einer Literal-Initialisierung müssen vom Typ BYTE oder WORD sein, RAD50-Konstante sind verboten.
- Als Initialisierung sind für die Typen BYTE, WORD, LIT MACRO-11 Assembler Ausdrücke erlaubt, für REAL MACRO-11 Real-Konstanten. Die Auswertung erfolgt im Assembler.

4.1.1 MACRO-11 Ausdrücke

In der Initialisierung dürfen für die Datentypen BYTE, WORD, LIT Ausdrücke vorkommen, die erst im nachgeschalteten MACRO-11 Assembler analysiert werden. Diese Ausdrücke werden zur Übersetzungszeit des Assemblers ausgewertet, die Initialisierung der Variablen erfolgt mit dem resultierenden Wert des Ausdrucks. Die Syntax legaler MACRO-11 Assembler Ausdrücke ist in Anhang F.2) angegeben.

Innerhalb von SIMPL11 gilt die Regel:

- Erfolgt die Initialisierung nicht mit einer definierten Konstanten (vgl. Tab.1), wird implizit ein MACRO-11-Assembler Ausdruck angenommen, d.h. ein Teil der Übersetzung wird explizit dem Assembler überlassen. Das Ende des Ausdrucks ist definiert durch ein Komma oder Zeilen-Ende oder Deklarationsende.

4.1.2 MACRO-11 Real Konstante

Real-Objekte dürfen mit reellen Konstanten initialisiert werden. Die Syntaxprüfung und Auswertung erfolgt im Assembler. Die Syntax legaler reeller Konstanten ist in Anhang F.3) angegeben. Es wird eine reelle Zahl einfacher Genauigkeit (2 Worte) erzeugt.

4.2 Zusammengesetzte Objekte (ARRAY, STACK)

4.2.1 ARRAY

Syntax:

```
ARRAY <ARRAY TYPE><IDENTIFIER><ARRAY INITIAL>;

<ARRAY TYPE>     := BYTE / WORD / REAL
<ARRAY INITIAL>  := [ <ARRAY LENGTH> ] / =<LIST OF INITIALS>
<ARRAY LENGTH>   := <NUM CONST> / <LITERAL>
```

Die Länge eines Arrays kann explizit durch eine legale Zahl oder ein Literal oder implizit durch eine Liste von Anfangswerten definiert werden. Die Deklaration darf sich über mehrere Zeilen erstrecken und wird durch ein Semikolon ';' abgeschlossen.

Beispiele für explizite Array Deklaration:

```
ARRAY BYTE ABC12[100];
ARRAY BYTE SOURCE[LSR];

ARRAY WORD XYZ1[^O500];
ARRAY WORD POINTR[LPT];

ARRAY REAL ARL12[^D30];
ARRAY REAL SYMBOL[LLA];
```

Für implizite Array-Deklarationen gelten dieselben Regeln wie in 4.1. D.h. die Typen der Konstanten müssen mit dem Array-Typ übereinstimmen; außer SIMPL11-Konstanten sind MACRO-11 Assembler-Ausdrücke und reelle Konstanten erlaubt.

Repetitionen:

Häufig sind Arrays zu initialisieren, in denen Wiederholungen einzelner oder mehrerer Anfangswerte vorkommen. Dies kann durch Repetitionen verkürzt ausgedrückt werden, nach der Syntax

<Repetitions Faktor> × [<Repetitions-Liste>]

Die Klammern dürfen weggelassen werden, wenn nur ein einzelner Wert zu wiederholen ist. Der Repetitionsfaktor ist entweder eine numerische Konstante oder eine Literal-Variable. Die Repetitionsliste besteht aus einer Liste von Anfangswerten oder weiteren Repetitionen.

Beispiele: LG, LST sind Literale, mit den Werten 10., bzw. 9. (vgl. Beispiel in 4.1)

```
ARRAY BYTE ATRIB=20,LG*[0,'E, 7*[3*[0,1], 4*5]];

ARRAY WORD LISTE=LST*[.+4,0], 0,0;

ARRAY WORD PARMS=1024.,, "AB, ".R, 4*[0,1,2],
      LG+LST+LSR*2, LSR, LISTE,LISTE+100,$RAD;

ARRAY REAL TERM=6, $AC12, $R5, $RADIX, $FUNCTN,
      $MACRO, $SUBRTN;

ARRAY REAL NUMBER=10.*[0,0.12], -13.5E-03, 3.145,
      4.14, 5*0;
```

Im ersten Beispiel wird ein Byte-Array der Länge 721 (dezimal) deklariert, mit den angegebenen Anfangswerten. Im zweiten Beispiel besteht das Array LISTE aus 20 Einträgen zu jeweils 2 Worten. Die Einträge sind durch einen MACRO-11 Ausdruck in der Weise verkettet, daß alle ungeraden Worte die Adresse des übernächsten Wortes enthalten. Das Ende der Verkettung ist durch eine Null gekennzeichnet. In der MACRO-11 Syntax bedeutet '.' den augenblicklichen Stand des Adresszählers, also ist '.+4' die Adresse des übernächsten Wortes.

4.2.2 STRINGS

Für String-Deklarationen kann eine verkürzte Schreibweise verwendet werden. Ein String ist eine beliebige Zeichenfolge, eingeschlossen von den Zeichen '.

Beispiel:

Die folgende Deklaration

```
ARRAY STRING TEXT = 'TEXTSTRING';
```

ist eine Abkürzung von

```
ARRAY BYTE TEXT = 'T, 'E, 'X, 'T, 'S, 'T, 'R, 'I, 'N, 'G,Ø;
```

die abschließende Null markiert das Ende des Strings.

4.2.3 LIMIT

Die LIMIT-Deklaration entspricht der im MACRO-11 Assembler vorhandenen .LIMIT Direktive.

Syntax:

```
LIMIT <Identifier>;
```

Es wird ein WORD-ARRAY, bestehend aus 2 Worten reserviert und dem angegebenen Identifikator zugeordnet. Zur Ladezeit setzt der Lader in das erste bzw. zweite Wort die unterste bzw. oberste Adresse des Lademoduls ein.

4.2.4 STACK

Syntax:

```
STACK <Identifier> [<Stack Length>];
```

Die Stack-Deklaration reserviert lediglich einen Speicherbereich, ohne Typ-Angabe. Die Stack-Länge wird durch eine numerische Konstante oder ein Literal angegeben und bestimmt die Anzahl der reservierten Worte. Der Stack Identifier bezeichnet das 1. Wort nach dem Stack-Bereich (=Höchste Stack-Adresse +2)!

Beispiel:

```
STACK  XSTACK [500.];
STACK  LCSTCK [LSR] ;
```

4.3 SIZE-Literal

Die Anzahl von Komponenten in einem Array ist eine zur Übersetzungszeit definierte Konstante und kann durch den Operator SIZE bestimmt werden, nach der Syntax

```
SIZE (<Array-Identifier>)
```

Das Resultat des Operators ist vom Typ LIT. Es ist zu beachten, daß SIZE echt die deklarierte Komponentenzahl liefert und nicht die tatsächlich im Speicher reservierte Bereichslänge.

Beispiel:

```
SIZE (LISTE) hat den dezimalen Wert 20.
```

4.4 Zusammenfassung der Beispiele

In Tab.3 sind die bisherigen Beispiele für Deklarationen nochmals zusammengefaßt.
Für die in den nächsten Kapiteln angeführten Beispiele sind die unten aufgeführten Deklarationen gültig.

Tab.3 Deklarations-Beispiele

```
LIT     LSR=1000.,LPT=LSR-^O100,LLA=62,LG=10.,LST=LG-1,
        COLON=':,ASSIGN="=>,LZ="AB,TRU=0,FALS=1;

BYTE    A,B,C,D='Y,E=5;

WORD    IX,X1=0,X2=128.,X3=120+LLA*2,AB=.-2,
        XY="EF,NM=$ABC,PT,ADR;

REAL    ARL,XR1=0,XR2=1234.56,XR3=123.4E21,
        DRL=0.0012,NAM=$RADIX,ALF;

ARRAY BYTE ABC12[100];
ARRAY BYTE SOURCE[LSR];
ARRAY BYTE ATRIB=20,LG*[0,'E,7*[3*[0,1],4*5]];

ARRAY STRING TEXT = 'TEXT STRING';


ARRAY WORD XYZ12[^O500];
ARRAY WORD POINTR[LPT];
ARRAY WORD LISTE=LST*[.+4,0],0,0;
ARRAY WORD PARMS=1024.,"AB,".R,4*[0,1,2],
      LG+LST+LSR*2,LSR,LISTE,LISTE+100,$RAD;


ARRAY REAL ARL12[^D30];
ARRAY REAL SYMBOL[LLA];
ARRAY REAL TERM=6,$AC12,$R5,$RADIX,$FUNCTN,
      $MACRO,$SUBRTN;
ARRAY REAL NUMBER=10.*[0,0.12],-13.5E-03,3.145,
      4.14,5*0;


STACK XSTACK[500.];
STACK LCSTCK[LSR];


LIMIT HICORE;
```

5. Operanden, Operatoren, Ausdrücke

Ausdrücke beschreiben die Transformation von Objekten. Die Objekte in Ausdrücken sind Operanden, die Transformationsvorschriften Operatoren. In SIMPL11 sind Operanden und Operatoren im wesentlichen durch die Maschinenarchitektur der PDP11 vorgegeben.

5.1 Operanden

Operanden sind innerhalb von Ausdrücken grundsätzlich Datentypen zugeordnet. Die Datentypen sind durch Deklaration, Definition oder Kontext festgelegt.

5.1.1 Einfache Operanden

Einfache Operanden sind zusammen mit den erlaubten Datentypen in Tab.4 dargestellt.

Tab.4: Einfache Operanden mit zugehörigen Datentypen

	Byte	Word	Real	Beispiele
Variable	+	+	+	XY, ARL
Literale	+	+		LSR, TRU
Register	+	+		RØ, ..., R5
String-Konstante	+	+		'A, "AB
RAD50-Konstante		+	+	$RAD, $ RADIX
Num-Konstante	+	+		12, 12., ↑O12

5.1.2 Indizierte Variable

Der Zugriff zu den Elementen eines Arrays erfolgt über Indizes nach der Syntax:

<Array Identifier> [<Exp>]

Der Index besteht aus einem beliebigen Ausdruck (vgl. Abschn. 5.3), sofern das Ergebnis des Ausdrucks ein Objekt vom Typ BYTE oder WORD liefert. Insbesondere dürfen in Index-Ausdrücken weitere Index-Ausdrücke beliebig tief verschachtelt sein. Indizes sind ab 1 numeriert.
Bei der Auswertung indizierter Variabler wird direkt das jeweilige Array-Element adressiert, d.h. der Benutzer braucht sich nicht um die intern unterschiedliche Repräsentation von BYTE-, WORD-, REAL-Objekten zu kümmern. Der berechnete Wert des Index-Ausdrucks bleibt unverändert erhalten.

Beispiele:

Legale Indizierungen sind:
(Die hier als Indizes verwendeten Ausdrücke werden erst später in Kapitel 5.3 beschrieben)

```
SOURCE[5];  SOURCE[RØ]; POINTR[IX]

SOURCE[RØ→ R1 + 5]

SOURCE[TOP-2→ IX+POP]

SOURCE[R1+POINTR[IX+POINTR[1]-RØ]→ PUSH]
```

5.1.3 ARRAY-Pointer

Es können verschiedentlich Anwendungen auftreten, in denen der Zugriff zu Objekten nur durch indirekte Adressierung erfolgen kann, z.B. wenn Objekte in einem "Arbeitsbereich" während der Programmlaufzeit dynamisch verschiebbar sind. Nicht indizierte Variable lassen sich einfach über einen Pointer zugreifen, während die Indizierung von Array-Elementen nicht ohne weiteres möglich ist. Die Adresse eines Array-Elementes muß aus Array-Anfangsadresse, Index und Elementlänge jeweils berechnet werden.

In SIMPL11 lassen sich die Vorteile der direkten Indizierung auch für den genannten Fall anwenden, wenn nur die Adresse eines Arrays bekannt ist. Der Mechanismus hierfür sind "ARRAY-Pointer".

Syntax:

```
<><Type> : <Pointer-Identifier><Array-Index>
```

Das Symbol <> Kennzeichnet den Array-Pointer. Der Datentyp der Array Elemente (BYTE, WORD, REAL) ist zur Übersetzungszeit nicht bekannt und muß daher explizit angegeben werden.
Der Identifikator muß vom Typ WORD sein. Er enthält die Adresse eines Arrays. Auf diesen Pointer folgt ein legaler Array-Index.

Beispiele:
Der folgende Ausdruck lädt die Adresse des WORD-Arrays POINTR in die Variable ADR vom Typ WORD:

```
REF(POINTR)→ADR
```

Die beiden folgenden Array-Zugriffe adressieren dasselbe Array-Element:

```
POINTR[RO→R1-5]
<>WORD:ADR[RO→R1-5]
```

Anmerkung:

Da bei Verwendung von Array-Pointern für den Zugriff zu Array-Komponenten ihr Datentyp explizit mit angegeben werden muß, kann der in der Deklaration festgelegte Array-Typ übergangen werden. Es ist daher z.B. möglich, über Array-Pointer einen WORD-Array byteweise zuzugreifen und umgekehrt. Für möglicherweise auftretende Fehler ist der Benutzer jedoch selbst verantwortlich. Z.B. kann kein Fehler festgestellt werden, wenn bei WORD-Zugriff in einem BYTE-Array die berechnete Adresse ungerade ist.

5.1.4 Indirekte Adressierung

Indirekte Adressierung adressiert ein Speicherobjekt über einen Operanden, dessen Inhalt die Adresse des Objektes ist. In der PDP11 ist indirekte Adressierung sowohl mit Registern als auch mit speicherbezogenen Objekten möglich. Die Syntax ist ähnlich wie im MACRO-11 Assembler, mit dem Unterschied, daß in indirekter Register-Adressierung alle Attribute hinter das Register geschrieben werden.

Indirekte Register-Adressierung

(vgl. auch DEC PDP11 Processor Handbuch)
Tab.5 enthält eine Zusammenstellung der möglichen Adressierungen. Rn bezeichnet die Register: RO, R1, ... R5, SP . Der Datentyp von indirekten Register-Operanden ist veränderlich. Je nach Kontext im Ausdruck sind BYTE, WORD, REAL erlaubt.

Tab.5: Indirekte Register-Adressierungen

Bezeichnung	Syntax	Erläuterung
Register Indirekt	@Rn	Rn = Adresse vom Speicherplatz
Auto Increment	(Rn)+	Rn:=Adresse vom Speicherplatz. Rn erhöhen(+)
	@(Rn)+	Rn:=Adresse der Adresse vom Speicherplatz. Rn erhöhen
Auto Decrement	-(Rn)	Rn:=Adresse vom Speicherplatz. Rn erniedrigen (+)
	@-(Rn)	Rn:=Adresse der Adresse vom Speicherplatz. Rn erniedrigen
Register Index	Rn(<Macro Ass Expr>)	Rn+<Macro Ass Expr>:=Adresse vom Speicherplatz (++)
	@Rn(<Macro Ass Expr>)	Rn+<Macro Ass Expr>:=Adresse der Adresse vom Speicherplatz

Anmerkungen:

+) Die Erhöhung bzw. Erniedrigung des Registers ist abhängig vom Datentyp, der dem indirekten Register-Operanden in der jeweiligen Operation zugeordnet ist:
BYTE-Operationen: Erhöhung bzw. Erniedrigung um 1
WORD-Operationen: " 2
REAL-Operationen: " 4

Dies entspricht dem jeweils für die interne Darstellung benötigten Speicherplatz. In SIMPL11 werden, im Gegensatz zum Assembler, auch REAL-Operanden bei Auto-Increment oder Auto-Decrement in dieser Weise berücksichtigt.

++) <Macro Ass-Expr> beschreibt einen legalen Ausdruck der Macro-11 Assembler Syntax (vgl. Abschnitt 4.1.1). Die Übersetzung und Auswertung erfolgt während der Assembler-Übersetzung.

Beispiele:

```
R1(5)
@RO(LLA*2-20.)
```

Hinweis:

Die PDP11 Hardware behandelt Auto-Increment/Decrement Adressierungen der Register R0, R1,...,R5 und des Registers SP etwas unterschiedlich. Byte-Operationen verändern R0,...,R5 um 1 nach oben oder unten, Register SP jedoch grundsätzliche um 2! (vgl. DEC-Processor Handbuch)

Indirekte Speicher-Adressierung

Alle bisher erwähnten Operanden der Art

- Identifikator, Literale, Numerische Konstante
- Indizierte Variable
- Array-Pointer

dürfen auch in indirekter Adressierung verwendet werden. Der Datentyp des indirekten adressierten Operanden muß jedoch in jedem Falle WORD sein.

Syntax: `@ <INDIRECT OPERAND>`

Beispiele:
```
@PT
@300
@POINTR[R0+XY-R1]
@<>WORD: PT[0=>R1+BT]
```

Indirekten Speicher-Adressierungen ist implizit der Datentyp WORD zugeordnet.

5.1.5 Stack-Operanden

An Stelle indirekter Register Operanden können auch die Stack-Operanden PUSH, POP, TOP verwendet werden, mit der Zuordnung:

PUSH[Rn] Auto-Decrement: Objekt auf dem Stack laden
POP [Rn] Auto-Increment: Objekt vom Stack entfernen

TOP [Rn, Zahl] {Register Indirekt / Register Index}: Zugriff zum Stack ohne Stack-Änderung

Es gelten die Regeln:

- PUSH, POP, TOP beziehen sich immer auf ein Register, als Stack Pointer.
- Fehlt eine explizite Register-Angabe, ist implizit Register SP - der System-Stack Pointer - gemeint.
- Im Operand TOP kann mit einer Zahlenangabe die Nummer des Stackelementes angegeben werden. Fehlt eine Zahlenangabe, ist das oberste Stackelement gemeint.
- Beim Operanden TOP erfolgt der Zugriff zum Stack implizit Wort-weise. Durch explizite Datentyp-Änderung (vgl. 5.3.1) kann ein Zugriff zu Byte- oder REAL Objekten auf dem Stack erreicht werden.

Beispiele:

Stack-Operanden	Indirekte Register Operanden	Erklärungen
R0=>PUSH	R0=>(SP)-	Auto-Decrement
@POP[R3]=>R0	@(R3)+ =>R0	Auto-Increment-Indirekt
TOP	@SP	Register Indirekt
TOP[R4]	@R4	Register Indirekt
@TOP[1]	@SP(0)	Register Index-Indirekt: Wortzugriff
TOP[2]	SP(2)	Register Index: Wortzugriff
JEDOCH:		
TOP[2']	SP(1)	Byte-Zugriff!
@TOP[R1,3']	@R1(2)	Byte-Zugriff!

Tab.6) Beispiele für Stack-Operanden

Es sei darauf hingewiesen, daß PUSH, POP, TOP zwar identisch sind mit den entsprechenden indirekten Register Operanden. Jedoch ist es zweckmäßig, in Verbindung mit der Datenstruktur STACK- und nur dort - auch die allgemein üblichen mnemonischen Bezeichnungen für die hierauf definierten Operationen zu verwenden!

Achtung:

Die indirekte Adressierung TOP[Rn], ohne Angabe der Nummer des Stack-Elementes ist illegal! Dies würde der in der Hardware verbotenen Adressierung @@Rn entsprechen. Das oberste Stack-Element kann indirekt nur durch @TOP[Rn, 1] adressiert werden, d.h. mit Angabe der Nummer auf dem Stack.

5.1.6 REF-Operanden

Die Adresse von Variablen wird durch den Operator REF bestimmt.

Syntax:

```
REF(<Macro-11 ASS EXPR>)     oder
REF(<Array Reference>)
```

Argumente von REF dürfen Macro Assembler Ausdrücke sein (vgl. Abschnitt 4.1.1) oder indizierte Variable.

Beispiele:

```
REF(SOURCE)
REF(SOURCE + 20)
REF(POINTER[POP RØ+IX])
REF(<>WORD:ADR[R1])
```

5.1.7 ABS-Operanden

Normalerweise sind die absoluten Speicheradressen der Daten eines Programms für den Programmierer bedeutungslos. Sie werden erst durch den Lader jeweils neu festgelegt. In manchen Fällen ist es jedoch notwendig fest definierte Speicherplätze zu adressieren, z.B. um einen Interrupt-Vector zu laden oder das Processor Status Wort zu testen. Dies geschieht mit dem Operator ABS:

```
ABS(<ABS-Operand>)
```


Der absolute Operand ist entweder ein Identifikator oder eine numerische Konstante.

Beispiele:

```
ABS(FIBO)
ABS(1ØØØ)
ABS(↑O3Ø)
```

5.1.8 SIZE-Operanden

Der Operator SIZE, gefolgt von einem Array-Identifikator, ist ein legaler Operand von Typ LIT (vgl. 4.3). SIZE bestimmt die Anzahl der Komponenten eines Arrays

Beispiel:

```
ARRAY REAL SYMBOL [100.];
```

Der Operand

```
SIZE (SYMBOL)
```

ergibt ein Resultat von dezimal 100.

5.1.9 Zusammenfassung der Operanden

Tab.7) gibt eine Übersicht über mögliche Operanden. Die Eintrage + in den Spalten geben an, welche Datentypen in Ausdrücken zulässig sind. Typen können durch Deklaration, Kontext im Ausdruck oder Fest vorgegeben sein. Dies ist jeweils durch D, K oder F in der Spalte "Definition" angezeigt. Treffen in Ausdrücken kontextabhängige Operanden zusammen, so gilt implizit der Datentyp WORD.

Operand	BYTE	WORD	REAL	ARRAY	Definition	Beispiele
Register	+	+			K	RØ, R1, ..., R5, SP
Float Register			+		F	ACØ, ..., AC5
Variable	+	+	+	+	D	ABC, XY, SOURCE
Literal-Variable	+	+			K	LSR, LPT
Indizierte Variable(+)	+	+	+		D	SYMBOL[INDEX]
Array Pointer	+	+	+		D	<>REAL:ADR[INDEX]
Num. Konstante	+	+			K	1234, 128., ↑O123, ↑D123
String-Konstante	+	+			F	'A, "AB
RAD50-Konstante		+	+		D	$RAD, $RADNAM
Indirekte Register	+	+	+		K	(RØ)-, @(SP)+, @SP
Stack Operanden	+	+	+		K	PUSH[RØ], POP , TOP
Indirekte Variable		+			F	@XY
REF-Operanden		+			F	REF(XY)
ABS-Operanden		+			F	ABS(100)
SIZE-Operanden	+	+			K	SIZE (POINTR)

Tab.7) Operanden in Ausdrücken

(+) Der Index von indizierten Variablen muß von Typ BYTE oder WORD sein!

5.2 Operatoren

Anzahl und Funktion der in SIMPL11 vorgesehenen Operatoren entspricht genau den im Rechenwerk der PDP11 definierten Operationen. Diese werden unterteilt in dyadische und monadische Operationen. Dyadischen Operationen sind zwei Operanden, monadischen Operatoren ist nur ein Operand zugeordnet.
Für die Bezeichnung der Operatoren wurden möglichst weitgehend die aus dem MACRO-11 Assembler bekannten mnemonischen Namen für Operationscodes beibehalten. Ausnahmen bilden die arithmetischen Operationen, für die die üblichen Zeichen +, -, *, / benutzt werden und die Zuweisungsoperation (MOV), die durch das Symbol → dargestellt wird.

Die vorhandenen Operatoren sind in Tab.8) und Tab.9) zusammengestellt. Die Tabellen enthalten Angaben darüber, welche Datentypen in Verbindung mit den einzelnen Operatoren erlaubt sind.

Hardware Einschränkungen

Für eine Reihe von Operationen gelten Besonderheiten bzw. Einschränkungen, die in der PDP11 Hardware-Konzeption begründet sind. Der Vorteil, alle Hardware Eigenschaften der Maschine voll ausnutzen zu können, bringt es auf der anderen Seite mit sich, daß Eigenheiten der Hardware beachtet werden müssen.

Die beiden wichtigsten Einschränkungen sind:
(vgl. auch DEC-Processor Handbuch).

a) Verschiedene Operationen sind nur für bestimmte Datentypen zulässig. Dies ist in Tab.8) und Tab.9) durch einen Eintrag in der entsprechenden Datentyp-Spalte angezeigt.

b) Verschiedene Operationen sind nur in Verbindung mit einem Register oder Float-Register erlaubt. Dies ist durch Einträge in die respektive Register-Spalte angegeben.

Beispiele:

- Dyadisch + (Addition) ist nur für WORD und Register erlaubt.
  Monadisch + (Increment) ist zusätzlich auch für BYTE erlaubt.

- Multiplikation and Division erfordern immer ein Register bzw. Float-Register, während Addition und Division nur für REAL-Objekte ein Float-Register benötigen, für WORD-Objekte nicht!

Achtung bei Multiplikation und Division mit den Registern RØ, ... R5:

Wenn ein gerades Register als Operand angegeben ist, sind zwei Register an der Operation beteiligt (vgl. DEC-Processor Handbuch).

Multiplikation: - R1, R3, R5 : Resultat ist im angegebenen Register
- RØ, R2, R4 : Resultat ist jeweils in den Registern (RØ, R1), (R2, R3), (R4, R5)

Division: - R1, R3, R5 : verboten
- RØ, R2, R4 : Tatsächliche Operanden sind (RØ, R1), (R2, R3), (R4, R5), wobei Quotient und Rest jeweils in (RØ, R1), (R2, R3), (R4, R5) sind.

Operator	BYTE	WORD	REG	REAL	FLOAT REG	Erklärungen
→	+	+	+	+	+	Zuweisung (MOV)
+		+	+		+	Addition
-		+	+		+	Subtraktion
*			+		+	Multiplikation
/			+		+	Division
ASH			+			Shift arithmetisch
ASHC			+			Shift arithmetisch Doppel-Register
BIC	+	+	+			Bit Clear
BIS	+	+	+			Bit Set (OR)
BIT	+	+	+			Bit Test (AND)
MODF					+	Multiplikation & Integer Konversion
XOR			+			Exclusive OR

Tab.8) Dyadische Operatoren

Operator	BYTE	WORD	REG	REAL	FLOAT REG	Erklärungen
+	+	+	+			Increment
-	+	+	+			Decrement
ABSF				+	+	Absolutwert
ADC	+	+	+			Addition CARRY-Bit
ASL	+	+	+			Shift arithmetisch Links
ASR	+	+	+			Shift arithmetisch Rechts
COM	+	+	+			Complement
NEG	+	+	+	+	+	Negation
ROL	+	+	+			Rotation Links
ROR	+	+	+			Rotation Rechts
SBC	+	+	+			Subtraktion CARRY-Bit
SWAB		+	+			Bytes vertauschen
SXT		+	+			Vorzeichen Extension

Tab.9) Monadische Operatoren

5.3 Ausdrücke

Ausdrücke werden als eine Folge von Operanden und Operatoren in eine Zeile geschrieben, nach den folgenden Regeln:

i) Die Abarbeitung erfolgt strikt von links nach rechts.
Es gibt keine Vorrang-Regelung von Operatoren und keine Klammerung von Teilausdrücken. Jeder Operation entspricht i.a. genau eine Assembler-Anweisung.

ii) Dyadische Operatoren sind zwischen zwei Operanden eingebettet (Infix-Schreibweise). Monadische Operatoren stehen hinter dem zugehörigen Operanden (Postfix-Schreibweise).

iii) Datentypen müssen übereinstimmen. Ausdrücke mit gemischten Typen sind verboten (Ausnahme s. 5.3.3). Operanden mit veränderlichem Datentyp werden angepaßt (vgl.Tab.7).

iv) Das Ergebnis von jeder Operation wird im Arbeits-Operanden abgelegt.
Der Arbeits-Operand ist entweder der
- Erste Operand eines Ausdrucks oder der
- Erste Operand, der auf die letzte Zuweisungs-Operation (→) folgt.
Das Resultat einer Folge von Operationen wird im jeweiligen Arbeits-operanden akkumuliert.
Ausnahme: Literale und Konstanten sind als Arbeits-Operand verboten!

v) Eine Zeile darf mehrere voneinander unabhängige Ausdrücke enthalten. Die einzelnen Ausdrücke werden durch Semikolon getrennt. Der letzte Ausdruck einer Zeile ist durch Zeilenende abgeschlossen.
z.B. <Ausdruck 1> ; <Ausdruck 2> ; ...

Beispiel:

```
R0-XY; 0=>R1; 2=>R2
```
Von RØ XY subtrahieren. Resultat in RØ, Ø nach R1, 2 nach R2 laden.

```
(R0)- =>R0(2)
```
Ein Wort über Pointer RØ umspeichern. RØ auf das nächste Wort setzen.

```
@TOP[R5,1]=>AC0 * @TOP[R5,2] => @TOP[R5,1]
```

Die Adressen zweier REAL-Zahlen sind auf Stack R5. Die erste Zahl (Adresse oben auf dem Stack) durch Produkt beider Zahlen ersetzen.

Weitere Ausdrücke:

```
@R4=>R4 + REF(PROG) + 2

REF(<>WORD: PT[IX+12.]) => R1+; @R1=>ATRB[IX]

X1=>PARMS[COUNT=>R1 ASL-] + POP NEG

NAMES[R1+XYZ[1]=>X0-2] + AB - R1 => STEP[POP]

POINTR[R0+] => R1+BZ * FAK + TOP[3] / X2=>PUSH
```

5.3.1 Explizite Datentyp-Änderung

Einem Operanden kann innerhalb eines Ausdrucks explizit ein anderer Datentyp als der vereinbarte zugewiesen werden. Die Änderung des Datentyps ist nur lokal für die Operationen gültig, an denen der Operand beteiligt ist. Die Typänderung wird durch folgende Operatoren (hinter den Operanden geschrieben!) bewirkt:

```
'   : BYTE
"   : WORD
""" : REAL
```

Beispiele:

```
WORD  XY;
ARRAY BYTE [100];

SOURCE[R1] + XY'
SOURCE[3=>R0 ASL]"=>PUSH
@(R0)+'=>@(R1)+'
```


Der Benutzer hat die Freiheit, durch explizite Datentyp-Änderung jede Typ-Vereinbarung, falls die Anwendung es erfordert, zu übergeben. Für evtl. entstehende Fehler (etwa bei Konversion von BYTE in WORD möglich) ist der Benutzer dann jedoch selbst verantwortlich!

5.3.2 Zuweisung zu Auto-Increment/Decrement Register:

Erfolgt eine Zuweisung zu einem Auto-Increment/Decrement Register, z.B.

```
(RØ)-, (RØ)+, bzw.
PUSH , POP
```

als neuem Arbeitsoperanden, gefolgt von weiteren Operationen, so wird das Register nur einmal, während der Zuweisung incrementiert oder decrementiert. Der verbleibende Ausdruck wird mit indirekter Register-Adressierung als Arbeitsoperand ausgewertet.

Beispiele:

IX→(RØ)+ - 10	identisch mit	IX→(RØ)+; @RØ - 10
XY→PUSH + ADR + 2	identisch mit	XY→PUSH; TOP + ADR + 2

Bei der Stack Operation PUSH wird also mit dem TOP-Element weitergearbeitet.

5.3.3 Ausdrücke mit gemischten Typen

Von der Regel 5.3 iii), nach der Ausdrücke mit gemischten Datentypen verboten sind, gibt es eine Ausnahme. Ist nämlich in einer Zuweisung

```
OP1→OP2
```

einer der beiden Operanden ein Float-Register (Typ REAL) und ist der andere Operand vom Typ WORD, so wird eine Zahlen-Konversion Integer → Float bzw. umgekehrt durchgeführt, nach der Regel:

a) <WORD> → <Float AC>
   Konversion von Integer- in Float-Darstellung

b) <Float AC>→<WORD>
Konversion von Float- in Integer-Darstellung.

Diese Konversionen werden durch die Hardware ausgeführt, falls ein entsprechendes Floating Point Processor vorhanden ist. Erlaubt sind (bedingt durch die Hardware) nur die Register ACØ, AC1, AC2, AC3!

Beispiele:

XY→ACØ	Konversion eines Integer in XY in eine Real-Zahl.
ACØ→(R1)+"	Konversion einer Real-Zahl in ein Integer. (R1) + muß explizit auf WORD geändert werden, sonst nimmt (R1)+ in Verbindung mit ACØ den Typ REAL an!
RØ→ACØ	Integer in RØ zu einer Float Zahl in ACØ konvertieren.

5.3.4 Repetition Ausdrücke

Die Anwendung gleicher Operationen auf eine Reihe verschiedener Operanden oder Ausdrücke kann in Repetitions-Ausdrücken verkürzt geschrieben werden. Die Ausdrücke in Repetitions-Ausdrücken dürfen selbst wieder Repetitions-Ausdrücke enthalten.

Beispiel:

a) [RØ, R1+POP→AB[IX] , IX] + REF(ADR) - 2

Dies ist identisch mit der Folge von Ausdrücken:

```
RØ + REF(ADR)-2
R1+POP→AB[IX] + REF(ADR) - 2
IX + REF(ADR) - 2
```

b) [AB, [RØ, AB[3] + 4] + IND, R1 + 3]→PUSH

Dies ist identisch mit

```
AB→PUSH
RØ + IND→PUSH
AB[3] + 4 + IND→PUSH
R1 + 3→PUSH
```

Repetitions Ausdrücke nach Zuweisungen

Repetitions Ausdrücke dürfen auch auf Zuweisungen folgen.

Beispiel:

(RØ)↔[XY, INDEX, R1, AB[2]] + POP[R4]→PUSH

Dies wird zerlegt in die Folge:

```
(RØ)+→ XY + POP[R4]→ PUSH
(RØ)+→ INDEX + POP[R4]→ PUSH
(RØ)+→ R1 + POP[R4]→ PUSH
(RØ)+→ AB[2] + POP[R4]→ PUSH
```

Repetitions Ausdrücke lassen sich besonders vorteilhaft verwenden, um Parameter zu initialisieren bzw. um Objekte auf dem Stack zwischenzuspeichern und wieder zu laden.

Beispiel:

```
[RØ, R1, PARM, ACØ, AC1]→PUSH
     Beliebiges
     Programm
        .
        .
        .
POP→[AC1, ACØ, PARM, R1, RØ]
```

5.3.5 Ausdrücke ohne Operatoren

Im Sonderfall, daß ein Ausdruck nur aus einem Operanden besteht, wird ein Test durchgeführt, dessen Ergebnis in einem bedingten Ausdruck abgefragt werden kann. Bei Auto Increment/Decrement Operanden wird auch das Register verändert!

Beispiele:

ADR	wird übersetzt in	TST	ADR
(RØ)+	wird übersetzt in	TST	(RØ)+
(RØ)+'	wird übersetzt in	TSTB	(RØ)+

Register RØ wird um 1 bzw. 2 erhöht, je nachdem ob es ein Byte- oder Word-Ausdruck ist.

5.3.6 Zusammenfassung der Operanden-Kombinationen in Ausdrücken

In Tab.10) und Tab.11) sind die erlaubten Kombinationen von Datentypen in Ausdrücken nochmals zusammengefaßt.
Tab.10) gilt für Operationen der Art

OP1 ⊕ OP2,

wobei OP1 und OP2 legale Operanden und ⊕ ein legaler Operator mit Ausnahme der Zuweisungsoperation ist.
Tab.11) gilt für Zuweisungen der Art

OP1 → OP2.

Die Zeilen und Spalten der Tabellen 10) und 11) enthalten die Datentypen von OP1 und OP2.

OP1 \ OP2	LIT	BYTE	WORD	REG	REAL	FLOAT REG
LIT						
BYTE	+	+		+		
WORD	+		+	+		
REG	+	+	+	+		
REAL						
FLOAT REG					+	+

Tab.10) Kombinationen von Operanden in Ausdrücken:
OP1 ⊕ OP2 OP1, OP2: legale Operanden
⊕ : legale Operation außer Zuweisung

OP1 \ OP2	LIT	BYTE	WORD	REG	REAL	FLOAT REG
LIT		+	+	+		+
BYTE		+		+		
WORD			+	+		+
REG		+	+	+		+
REAL					+	+
FLOAT REG			+	+	+	+

Tab.11) Kombinationen von Operanden in Zuweisungs-Ausdrücken:
OP1 → OP2

6. Bedingte Ausdrücke

Bedingte Ausdrücke werden in Kontroll Anweisungen verwendet, um

- Operanden miteinander zu vergleichen,
- Das Resultat der letzten Operation zu testen, oder
- Den Status des Rechenwerks zu testen.

Auf Grund des Resultates eines bedingten Ausdrucks, das wahr oder falsch sein kann, können Programm-Verzweigungen vorgenommen werden.

Intern werden die im PDP11-Rechenwerk vorhandenen "Condition Codes" gesetzt (vgl. DEC-Processor-Handbuch),bzw. nur abgefragt. Bedingte Ausdrücke müssen einer der folgenden Syntax-Regeln genügen:

```
Dyadischer Ausdruck:     <Expr> <Cond. Operator><Operand>
Monadischer Ausdruck:    <Expr> <Cond. Operator>
Niladischer Ausdruck:           <Cond. Operator>

oder:                    <Expr>
```

<Expr> bezeichnet einen beliebigen Ausdruck. Die bedingten Operatoren <Cond. Operator> sind in Tab.12) zusammengefaßt. Sämtliche Operatoren dürfen ohne Operanden (niladisch) verwendet werden. Für die Operatoren CARRY, OFLOW ist monadische und dyadische, für die Operatoren TRUE, FALSE nur die dyadische Verwendung verboten.

Bei Anwendung der Operatoren GT, GE, LT, LE wird das Vorzeichen der Operanden mitberücksichtigt. Bei Anwendung der Operatoren LOWER, LOSAME, HIGHER, HISAME erfolgt ein reiner BIT-Vergleich der Operanden, ohne Beachtung eines Vorzeichens!

	Nil	Mon	Dya	Condition Codes	Erläuterung
EQ	+	+	+	Z	Gleich
NE	+	+	+	Z	Ungleich
GT	+	+	+		Größer
GE	+	+	+	N, V,	Größer oder Gleich
LT	+	+	+	Z	Kleiner
LE	+	+	+		Kleiner oder Gleich
LOWER	+	+	+		Niedriger
LOSAME	+	+	+		Niedriger oder Gleich
HIGHER	+	+	+	C,Z	Höher
HISAME	+	+	+		Höher oder Gleich
TRUE	+	+		Z	Test Z = 1 (+)
FALSE	+	+		Z	Test Z = Ø
CARRY	+			C	Test C = 1
OFLOW	+			V	Test V = 1

Tab.12) Bedingte Operatoren

+) Z = 1 bedeutet: die letzte Operation resultiert in einer Null
Z = 0 bedeutet: Das Resultat der letzten Operation war ungleich Null.

6.1 Vergleichs-Operatoren

6.1.1 Dyadische Operatoren

Beispiele:

```
RØ GT AB
 R1 EQ "ST
TOP EQ 'A
ATRIB [R1]+ POP→RØ  LOWER REF(ADR)
SYMBOL [3] HISAME NAME
ACØ EQ ALFA
```

Hardware Einschränkungen bei REAL-Operanden

Die Struktur der PDP11 Hardware erzwingt die Beachtung der folgenden Restriktionen:

- Beim Vergleich eines Float Registers mit einer REAL-Variablen muß das Float-Register links vom bedingten Operator stehen. Es dürfen nur die Float-Register ACØ, ..., AC3 und die Operatoren EQ, NE, GT, GE, LT, LE benutzt werden.
- Sind beide Operanden REAL-Variable, so sind nur die Operatoren LOWER, LOSAME, HIGHER, HISAME sinnvoll. Die Operatoren EQ, ..., LE sind zwar syntaktisch zulässig, jedoch wird kein numerischer Vergleich, sondern nur ein Vergleich der Bit-Muster durchgeführt. Numerische Vergleiche von Real-Operanden müssen in einem Float Register durchgeführt werden.

Beispiele:

Legal			Illegal		
ACØ	EQ	AC1	ACØ	LOWER	AC1
ACØ	EQ	POP""	POP""	EQ	ACØ
ACØ	GT	NUMB	AC4	GT	NUMB
NAME	LOWER	ALFA	NAME	EQ	ALFA

6.1.2 Monadische Operatoren

Bei monadischer Benutzung der Operatoren EQ, ..., HISAME wird der 2. Operand implizit als Null angenommen. TRUE und FALSE testen entweder den angegebenen Operanden oder das Ergebnis der letzten Operation.

Die Bedingung ist in den folgenden Beispielen erfüllt:

```
0 EQ

-1 LT

2 HIGHER

1=:R0 ASL-2 TRUE

0=:R0+2 FALSE
```

In den folgenden Beispielen ist die Bedingung nicht erfüllt:

```
1 EQ

10=:R1 LT

0=:R1*4 LOWER

1=:R0 ASL-2 FALSE

0=:R0+2 TRUE
```

6.1.3 Niladische Operatoren

Es werden nur die "Condition Codes", wie sie durch vorhergehende Operationen gesetzt wurden, abgefragt.

6.2 Beliebige Ausdrücke als Bedingungen

Ein beliebiger Ausdruck (vgl. Kap. 5.3) kann legal als bedingter Ausdruck verwendet werden. Dasselbe gilt für den Aufruf von Subroutinen. In diesen Fällen wird nach Auswertung des Ausdrucks bzw. nach Rückkehr von der Subroutine das Resultat der letzten Operation auf TRUE geprüft. D.h. die Bedingung ist erfüllt, wenn der Condition Code Z = 1 ist.

6.3 Zusammengesetzte logische Ausdrücke

Logische Ausdrücke dürfen durch UND bzw. ODER Operationen beliebig verknüpft werden.

Es gelten die Abkürzungen

& : Logisches UND

: Logisches ODER

Beispiel:

```
A # AB[4] GE -1 # CARRY & R0 EQ    .....
```

6.4 Bedingte Operanden und Datentypen

In Tab. 13) ist die erlaubte Anwendung bedingter Operatoren auf die Datentypen von Operanden zusammengefaßt.

Logischer Operator	BYTE	WORD / Register	REAL	FLOAT Register
EQ	+	+		+
NE	+	+		+
GT	+	+		+
GE	+	+		+
LT	+	+		+
LE	+	+		+
LOWER	+	+	+	
LOSAME	+	+	+	
HIGHER	+	+	+	
HISAME	+	+	+	
TRUE	+	+	+	+
FALSE	+	+	+	+
CARRY				
OFLOW				

Tabelle 13) Anwendung bedingter Operatoren auf Typen von Operanden

7. Kontroll-Anweisungen

Ein Programm wird gewöhnlich zeilenweise abgearbeitet. Durch Kontroll-Anweisungen kann diese sequentielle Abarbeitung abgeändert werden. Folgende Arten von Kontroll-Anweisungen sind möglich:

- Einfache Schleifen (FOR ... DO ...)
- Bedingte Anweisungen (IF ... THEN ... ELSE ...)
- Bedingte Schleifen (WHILE... DO ...; REPEAT ... UNTIL ...)
- Sprung-Anweisungen (GO ...)

7.1 Blöcke

Kontroll-Anweisungen dürfen nicht unmittelbar ineinander verschachtelt sein. Innerhalb einer Zeile dürfen nur Ausdrücke bzw. eine Folge von Ausdrücken vorkommen.
Sind in einer Kontrollanweisung jedoch

- eine Folge von Zeilen oder
- weitere Kontrollanweisungen

auszuführen, so muß das betreffende Programmstück als Block geschrieben werden, d.h. es muß zwischen den Beggrenzungssymbolen BEGIN ... END enthalten sein. Blöcke dienen in SIMPL11 nur der Strukturierung von Kontrollausdrücken, eine Block-Struktur im üblichen Sinne für den Geltungsbereich von Namen ist nicht definiert!

Beispiele:
Formen legaler Kontrollanweisungen:

a) FOR ... DO BEGIN IF ... THEN ... ELSE ... END

b)
```
IF ... THEN
BEGIN
  FOR ... DO ...
  IF  ... THEN ...
END ELSE
BEGIN
  .
  .
END
```

c) REPEAT
   BEGIN
      IF ... THEN ... ELSE ...
   END UNTIL . . .

Illegal sind Anweisungen der Art

a) IF ... THEN IF ... THEN ... ELSE ... ELSE ...
b) IF ... THEN ... ELSE FOR ... DO ...
c) WHILE ... DO IF ... THEN ...

7.2 FOR-Anweisung

Die Anweisung FOR ... DO ... wird für die Formulierung von Schleifen angewandt, bei denen die Anzahl der Wiederholungen bekannt ist.

Syntax:

```
FOR [<EXPR>=>] <REGISTER> DO <COMMAND>
<COMMAND> := <EXPR> / <EXPR> [; <EXPR> ...] /
             <BLOCK>
```

Nach dem FOR darf ein beliebiger Ausdruck folgen. Das Resultat muß vom Typ BYTE oder WORD sein, und muß einem Register zugewiesen werden. Das Register kontrolliert den Ablauf der Schleife. Es wird bei jedem Durchlauf um 1 erniedrigt bis zum Wert Ø.

Beispiel:
Alle Werte des REAL-Arrays NUMBER addieren und den Mittelwert bilden. Das Resultat soll auf den Stack:

```
0=>AC0
FOR SIZE(NUMBER)=>R0  DO  AC0+NUMBER[R0]
SIZE(NUMBER)=>AC1
AC0/AC1=>PUSH
```


7.3 IF-Anweisung

Mit IF ... THEN ... ELSE werden Programmverzweigungen auf Grund einer Bedingung formuliert.

Syntax:

```
IF <CONDITION> THEN <COMMAND> [ELSE <COMMAND>]
```

Der ELSE-Ausdruck darf auch fehlen. <Condition>ist ein bedingter Ausdruck (vgl. Kap. 6).

Beispiele:

a)
```
IF XY EQ 'DD  THEN  0=>R0  ELSE  1=>R0
```

b)
```
IF SOURCE[R0]=>R1 GE 'A & R1 LE 'Z  THEN  PRINT R1'
```

c)
```
IF CARRY  THEN  GO ERROR  ELSE  RETURN
```

d)
```
IF @R0 NE THEN
BEGIN
   IF @R0 EQ R1(2)'  &  R0(2) EQ R2(2)'  THEN
   BEGIN
      R0(4)=>X1
      TRU=>R0
      RETURN
   END
   IF @R0 EQ R2(2)'  &  R0(2)'  THEN
   BEGIN
      R0(4)=>X1
      FALS=>R0; RETURN
   END  ELSE R0+6
END
```

Abkürzung

Zur Verkürzung der Schreibarbeit dürfen in IF-Anweisungen THEN und ELSE durch das Zeichen ':' ersetzt werden.

Beispiele:

```
IF XY EQ "DD : 0→RØ : 1→RØ
IF CARRY : GO ERROR : RETURN
```

7.4 Bedingte Schleifen

Bedingte Schleifen werden benutzt, wenn die Anzahl der Wiederholungen von einer Bedingung abhängt.

7.4.1 WHILE-Anweisung

Die Bedingung wird vor Ausführung jeder Wiederholung geprüft.

Syntax:

```
WHILE <CONDITION> DO <COMMAND>
```

7.4.2 REPEAT-Anweisung

Die Bedingung wird nach Ausführung jeder Wiederholung geprüft.

Syntax:

```
REPEAT <COMMAND> UNTIL <CONDITION>
```

Beispiele:

a) Wieviel aufeinanderfolgende natürliche Zahlen müssen addiert werden, bis die Summe größer oder gleich 15.000. ist? Das Resultat soll auf den Stack.

Mit WHILE:

```
0=>[R0,PUSH]
WHILE R0 LT 15000. DO TOP+; R0+TOP
```


Mit REPEAT:

```
0=>[R0,PUSH]
REPEAT TOP+; R0+TOP UNTIL R0 GE 15000.
```

b) In einem Text String alle evtl. vorhandenen Ausrufezeichen durch ein Leerzeichen ersetzen:

```
ARRAY STRING TEXT = '!TEXT ! STRING!';
LIT BLANK = 40;

0=>R0

REPEAT
BEGIN
   IF TEXT[R0+] EQ '! THEN BLANK=>TEXT[R0]
END UNTIL R0 EQ SIZE(TEXT)
```

7.5 Blocklänge in Kontrollanweisungen

Programmverzweigungen und Schleifen werden vom SIMPL11-Compiler aus Gründen der Effizienz direkt in BRANCH-Befehle übersetzt. Die Reichweite von Branch-Befehlen ist jedoch durch die PDP11-Hardware in der Weise eingeschränkt, daß nur Adressen im Abstand von 128 Worten (vorwärts und rückwärts) vom augenblicklichen Stand des Adreß-Zählers erreicht werden können. Blöcke sollten daher eine Länge von 128 Worten nicht überschreiten. Ein diesbezüglicher Fehler kann erst im Macro-11 Assembler festgestellt werden. Ist die Blocklänge zu groß, so empfiehlt sich die Einführung geeigneter Subroutinen!

7.6 Sprünge

Sprünge in einer beliebigen geraden Speicheradresse werden gebildet nach der Syntax:

```
GO <WORD Address>
```

8. Macros und Subroutinen

8.1 Macros

Alle Macro-Eigenschaften des Assemblers (Macro-Erzeugung und -Aufruf, bedingte Übersetzung von Macros) können auf der Ebene von SIMPL11 mitbenutzt werden. Die Syntax der Macro-Definition wurde insofern geändert, als die Datentypen der Macro-Parameter mit angegeben werden müssen. Die Parameter selbst sind nur innerhalb der Macro-Definition definiert (Lokale Gültigkeit). Bei Macro-Aufruf werden sie durch aktuelle Parameter ersetzt.

Syntax:

a) Definition:

```
MACRO <MACRO-IDENTIFIER> ( <TYPE> : P1,P2,...; <TYPE> : P3,...)
<MACRO-BODY>
.ENDM
```

b) Aufruf:

```
<MACRO-IDENTIFIER> ( <EXPR 1> , <EXPR 2> , <EXPR 3>  , ...)
```

Beispiel:

a) Ein Macro "EVEN" soll WORD-Operanden um 1 erhöhen (gerade machen), falls deren Inhalt ungerade ist.

```
Definition      MACRO EVEN (WORD : ARG)
                ARG + BIC 1
                .ENDM
Aufrufe:        EVEN (R1)
                EVEN (1→IX + POINTR [RØ+ 1])
```

8.2 Subroutinen

SIMPL11 enthält kein besonderes Procedure- bzw. Subroutinen-Konzept, vergleichbar etwa mit ALGOL oder FORTRAN. Bei diesen zeichnen sich Prozeduren bzw. Subroutinen im wesentlichen durch den Begriff der "Formalen Parameter" aus, die formal definiert, aktuell beim Aufruf spezifiziert und der Prozedur auf geeignete Weise übergeben werden. Der Begriff der formalen Parameter-Definition und ein Parameter-Übergabe-Mechanismus fehlen in SIMPL11.
Ein Subroutinen-Aufruf benutzt nur die in der Hardware dafür vorgesehenen

Möglichkeiten (vgl. DEC-Processor-Handbuch). Die Parameter müssen vom Benutzer selbst übergeben werden.

8.2.1 Aufruf

Syntax:

```
JSR <ADDRESS> [( <REGISTER> )]
```

Die PDP11 benutzt für den Subroutinen-Aufruf immer ein Register. Wird in der obigen Syntax die Register-Angabe weggelassen, so erfolgt der Aufruf implizit über Register R7, den Adreß-Zähler (PC). Jeder speicherbezogene Operand, eine gerade Adresse vorausgesetzt, darf als Subroutine angesprungen werden.

Beispiele:

```
JSR     LEXA(R5)
JSR      @(R0)+
JSR     @SWITCH[IX]
```

Abkürzung:

Zur Verkürzung der Schreibarbeit bei vielen Subroutine-Aufrufen darf 'JSR' durch das Zeichen '!' abgekürzt werden.

Beispiele:

```
! LEXA(R5)
! @(R0)+
! @SWITCH[IX]
```

8.2.2 Rücksprung

Syntax:

```
RETURN [<TRUE/FALSE>] [( <REGISTER> )]
```

Beim Rücksprung von einer Subroutine muß dasselbe Register wie beim Aufruf mit angegeben werden. Erfolgte der Aufruf ohne Register-Angabe, kann auch beim Rücksprung das Register weggelassen werden.
Zusätzlich darf jedem Rücksprung das Attribut TRUE oder FALSE beigegeben werden. Der Condition Code Z wird vor Rücksprung entsprechend auf 1 oder 0

gesetzt und kann in einem bedingten Ausdruck abgefragt werden.

Beispiele:

a) RETURN

   RETURN TRUE

   RETURN (R5)

   RETURN FALSE (R4)

b) Alle Kommas in einem Text-Buffer SOURCE zählen:

```
COUNT:  FOR SIZE(SOURCE)=>R1  DO
        BEGIN

           SOURCE [SIZE(SOURCE)=>R0-R1+1] => B
           IF !ISCOMA THEN CT+

        END
        RETURN

ISCOMA: IF B EQ ',  THEN RETURN TRUE  ELSE  RETURN FALSE
```

8.3 Aufruf von FORTRAN-Subroutinen

In FORTRAN geschriebene Subroutinen lassen sich mit einem CALL-Aufruf vereinfacht aufrufen.

Syntax:

```
CALL <IDENTIFIER> ( <EXPR 1> , <EXPR 2> , ...)
```

Die Parameter-Übergabe in FORTRAN Subroutinen wird in den verschiedenen DEC-Fortran Compilern unterschiedlich gehandhabt. Die gegenwärtige Implementierung von SIMPL11 berücksichtigt die Konvention des RT11-Fortran Compilers. Der SIMPL11-Übersetzer kann jedoch vom Benutzer - ohne Eingriff in den Compiler selbst - durch Änderung eines Macros jederzeit anderen Konventionen angepaßt werden.

Die Konventionen der RT11-Fortran Parameterübergabe und Änderungsmöglichkeiten sind in Anhang A.4 beschrieben.

9. SIMPL11/MACRO11-Kommunikation

9.1 Assembler Direktiven und System Macros

Der Macro11-Assembler enthält eine Reihe nützlicher Direktiven zur Steuerung der Übersetzung. Ebenso sind im Betriebssystem bereits eine Reihe von System-Macros vordefiniert. Beide können in SIMPL11 ohne Einschränkung mitbenutzt werden.

Assembler-Direktiven und System-Macros sind gekennzeichnet durch einen Punkt '.' vor dem Instruktionsnamen. In SIMPL gilt die Vereinbarung:

> Alle Ausdrücke, die als erstes Zeichen einen Punkt '.' enthalten, bleiben unverändert und werden bei der Übersetzung umkopiert. Innerhalb dieser Zeilen muß die Assembler-Syntax beachtet werden!

Das Sonderzeichen >

Oft ist es wünschenswert, System-Macros mit dem Macro-Aufrufmechanismus von SIMPL11 zu verwenden (vgl. 8.1). Durch Einfügen des Zeichens >
vor dem Macro-Aufruf wird die obige Regel außer Kraft gesetzt. Der Übersetzer setzt dann implizit voraus, daß ein Ausdruck der Art

```
> .<MACRO-NAME>
```

ein Macro-Aufruf ist!

Beispiel:

Benutzung des System Macros READW aus RT11

a) Assembler-Syntax:

```
        INC     CHANN

        .READW  #LIST, CHANN, #INBUF, MAX, BLK
```

b) SIMPL11-Syntax

```
> .READW (REF(LIST), CHANN+, REF(INBUF), MAX, BLK)
```

9.2 Einfügen von Assembler Code

Außer der in 9.1 erwähnten impliziten Übernahme von Assemblerzeilen kann beliebiger Assembler-Code in SIMPL11 Programme eingebettet werden. Mit dem Zeichen '%' kann zu jeder Zeit zwischen SIMPL11- und Assembler-Programmierung hin- und hergeschaltet werden.

Beispiel:

```
        R0 + X2 => R1 * 3

%       MOVB    R1,-(SP)        ;BEGINN ASSEMBLER PROGRAMM
        CMPB    A,#10.
        BEQ     1$
        INC     X2
1$:     ADD     (SP)+,X2        ;ENDE ASSEMBLER PROGRAMM
%

        X2 + XYZ12[IX+] => PUSH // HIER GEHT'S WEITER IM SIMPL11-
                               // PROGRAMM
```

Der zwischen den Zeichen '%' stehende Programmtext wird unverändert umkopiert. % muß jeweils das erste gültige Zeichen in der Zeile sein.

10. Eingabe / Ausgabe von Daten

Für die Eingabe/Ausgabe von Werten gibt es die Kommandos

READ

PRINT

Zusätzlich sind in SIMPL11 zur Vereinfachung des Programm-Testens DUMP-Kommandos enthalten:

DUMPS

DUMP

DREG

DSTACK

10.1 READ, PRINT

Mit READ, PRINT werden Werte bzw. Listen von Werten ein- bzw. ausgegeben. Kontroll-Zeichen steuern ASCII-, Oktal-, Dezimal-, Float- oder RAD50 Modus der Ein-/Ausgabe.

Syntax:

```
<READ/PRINT> := [<FORMAT-CONTROL>] <PR-EXPR 1> , <PR-EXPR 2> , ...
<PR-EXPR>    := [<CONTROL-CHARACTER>] <EXPR> / <STRING> / <EMPTY>
```

10.1.1 Ausdrücke in E/A-Anweisungen

Wenn in Eingabe-/Ausgabe-Anweisungen Ausdrücke enthalten sind, werden diese erst ausgewertet. Die E/A-Operation erfolgt dann mit dem resultierenden Arbeitsoperanden. In READ-Anweisungen sind zwar beliebige Ausdrücke syntaktisch zulässig, i.a. jedoch wenig sinnvoll, da das Resultat des Ausdruckes durch die READ-Operation sofort wieder ersetzt wird.
Ausnahme: Auto-Increment/Decrement Register Operanden.

Sowohl für PRINT als auch READ-Anweisungen gilt:

- Für Strings erfolgt Ausgabe auf dem Terminal
- Leere Ausdrücke (aufeinanderfolgende Kommas) bewirken Ausgabe einer neuen Zeile.

10.1.2 Kontrollzeichen

Eingabe bzw. Ausgabe des Arbeitsoperanden erfolgt nach Maßgabe eines Kontrollzeichens. Folgende Kontrollzeichen gelten sowohl für READ als auch für PRINT:

- % E/A-Operation dezimal ausführen, entweder als Integer bzw. Float-Zahl, je nach dem Datentyp BYTE, WORD bzw. REAL
- %% E/A-Operation oktal ausführen. Erlaubt sind die Datentypen BYTE, WORD.
- $ E/A-Operation in RAD50-Darstellung ausführen. Erlaubt sind die Datentypen WORD, REAL.
- ! Operanden, der die E/A-Operation verursacht, vor Ausführung der Operation selbst als String ausgeben.
- # Wert des Ausdrucks enthält im Arbeitsoperanden die Adresse eines Strings.
  Dieser wird wie ein String innerhalb der E/A-Anweisung selbst ausgegeben.
- leer: ist kein Kontrollzeichen angegeben, erfolgt die E/A-Operation im ASCII-Mode (Default Option!).

10.1.3 Format-Kontrolle

Für Ausgaben gelten zwei Formatierungs-Regeln:

i) Jede PRINT-Anweisung beginnt mit einer neuen Zeile
ii) Die Argumente einer PRINT-Liste werden fortlaufend in einer Zeile ausgegeben, getrennt jeweils durch 2 Leerzeichen.

Diese Regeln können durch Formatierungs-Kontrollzeichen aufgehoben werden:

- / Unterdrückung der Ausgabe einer neuen Zeile
- - Unterdrückung der Leerzeichen innerhalb der Zeile.

Formatierungs-Kontrollzeichen dürfen einzeln oder gemeinsam auftreten. Die Reihenfolge ist unwichtig, die Zeichen müssen jedoch direkt auf die PRINT-Anweisung folgen.

10.1.4 Leerzeile

Die Verwendung von PRINT, ohne Angabe von Argumenten, bewirkt die Ausgabe einer Leerzeile.

10.1.5 Beispiele

a)
```
PRINT                          // EINE LEERZEILE AUSGEBEN
PRINT , , ,                    // 3 LEERZEILEN AUSGEBEN
```

b)
```
READ !%X1, !X4, %TERM[R1+IX], %(R1)+""
READ 'BITTE EINEN NAMEN EINGEBEN: ' ,, $NAM
```

c)
```
// EINEN TEXT EINLESEN BIS ZU EINEM ENDZEICHEN ODER
// BUFFERENDE:
ARRAY BYTE SOURCE[LSR];
0=:R0
SIZE(SOURCE)=:R1
REPEAT READ SOURCE[R0+] UNTIL R0 EQ R1 # SOURCE[R0] EQ ESIGN
```

d) das Programm

```
ARRAY   STRING STR= 'NEUE ZEILE  ';
REAL    NAM=$RAD50;
START:  PRINT 'AUSGABE',,#REF(STR),"XY=:R0,!%100.=:R1,!%%R1,$NAM
        .END START
```

ergibt die Ausgabe:

```
AUSGABE
NEUE ZEILE  XY R1=100 R1=000144 RAD50
.
```

10.2 DUMP-Anweisungen

Beim Programm-Testen ist es oft zweckmäßig, Register, einzelne Speicherwerte oder Speicherbereiche gleichzeitig in Oktal-, ASCII- oder RAD50-Darstellung auszugeben. Hierfür sind DUMP-Anweisungen vorgesehen. Zwar sind die Möglichkeiten der DUMP-Anweisungen teilweise bereits in der PRINT-Anweisung enthalten, doch ist es beim Testen meist einfacher, die kürzere DUMP-Schreibweise zu benutzen.

10.2.1 DUMPS

DUMPS dient zum Ausdrucken einzelner Operanden bzw. Variabler. Die Form ist:

```
DUMPS [$] <Expr 1> , [$]< Expr 2>, ...
```

Der Ausdruck erfolgt in Oktal-Wort, Oktat-Byte und ASCII-Darstellung. RAD50-Ausgabe kann durch das Zeichen $ vor jedem einzelnen Ausdruck verlangt werden.

Beispiel:

Programm:

```
WORD    XY="AB, X1=0, X2=128., NM=$RAD;

START:  DUMPS XY,100=:R1*4,R1=:X1+X2,X2,$NM
        .END START
```

Ausdruck:

```
XY=     041101  102 101   /AB/
R1=     000400  001 000   //
X1=     000600  001 200   //
X2=     000200  000 200   //
NM=     070254  160 254   /,P/RAD
```


10.2.2 DUMP

DUMP druckt einen zusammenhängenden Kernspeicherbereich aus.

Syntax:

```
DUMP [$] <Expr 1>, <Expr 2>
```

Die beiden Ausdrücke geben die untere bzw. obere Adresse des zu druckenden Bereichs an. Die Reihenfolge ist unwichtig. Anfangs- und Endadresse werden auf oktal 10 ab- bzw. aufgerundet.

Beispiel:

```
        LIT     LST=20;
ARRAY   WORD FARMS= 5*[-4,0], 10*12345, "TE,"XT,1024.,LST,$RAD;
START:  DUMP REF(FARMS)=:R0, SIZE(FARMS)=:R1 ASL4R0
        .END START
```

```
FROM 001000    TO 001056
001000 001004 000000 001010 000000  002 004  000 000  002 010  000 000   /
001010 001014 000000 001020 000000  002 014  000 000  002 020  000 000   /
001020 001024 000000 012345 012345  002 024  000 000  024 345  024 345   /
001030 012345 012345 012345 012345  024 345  024 345  024 345  024 345   /
001040 012345 012345 042524 052130  024 345  024 345  105 124  124 130   /e e TEXT/
001050 002000 000020 070254 012700  004 000  000 020  160 254  025 300   /,P@ /
```

10.2.3 DREG, DSTACK

Der Inhalt der allgemeinen Register und des Stacks wird durch die Anweisungen

DREG
DSTACK

ohne Argumente ausgegeben.

Beispiel:

a)
```
START:  0=:[R0,R1,R2]
        123=:[R3,R4]
        'XY =: R5
        DREG
        .END START
```

Ausgabe:

```
R0=     000000  000 000   //
R1=     000000  000 000   //
R2=     000000  000 000   //
R3=     000123  000 123   /S/
R4=     000123  000 123   /S/
R5=     054530  131 130   /XY/
SP=     000776  001 376   /~/
```

b) Der oberste Stack-Eintrag resultiert vom Aufruf der STACK-Routine und ist nicht zu beachten.

b)
```
START:  [1,2,3,4,5,'X,"AB]=:PUSH
        DSTACK
        .END START
```

Ausdruck:

```
SP =    000760
000760  001040  002 040   / /
000762  041101  102 101   /AB/
000764  000130  000 130   /X/
000766  000005  000 005   //
000770  000004  000 004   //
000772  000003  000 003   //
000774  000002  000 002   //
000776  000001  000 001   //
```


10.3 Zusammenfassung der Kontrollzeichen

In Tab. 14 sind alle Kontrollzeichen und ihre Anwendung auf bestimmte Datentypen und E/A-Kommandos noch einmal zusammengestellt.

Tab. 14: Kontroll-Zeichen in Eingabe/Ausgabe-Anweisungen

Kontroll-zeichen	BYTE	WORD	REAL	READ	PRINT	DUMPS DUMP	DREG DSTACK	Bedeutung
/					+			Zeilenvorschub-Unterdrückung
-					+			Leerzeilen-Unterdrückung
%	+	+	+	+	+			Dezimal/Float-E/A
%%	+	+		+	+			Oktal E/A
$		+	+	+	+	+		RAD50 E/A
!	+	+	+	+	+			Operand als String ausgeben
#		+		+	+			String über Adresse ausgeben
leer	+	+	+	+	+	+	+	ASCII- E/A

11. Spezielle Anweisungen

Die folgenden in der PDP11-Hardware definierten Kommandos können direkt in SIMPL11 verwendet werden (ohne Umschaltung zum Assembler):

TRAP <Zahl>
SETF
SETI
SETD
SETL
LDFPS
STFPS
STST

Die beiden folgenden Kommandos sind in funktionaler Schreibweise zu benutzen. Die Operanden müssen vom Typ WORD sein.

LDEXP (<Operand> ,<Float Reg>)
STEXP (<Float Reg>, <Operand>)

Zur Bedeutung dieser Kommandos siehe DEC-Processor-Handbuch.

12. Beispiele

12.1 Einlesen einer Integer-Zahl vom Terminal

```
        .MCALL  .TTYIN

array   byte INP[100];
lit     LF=12,BLANK=40;


        macro ERROR
        pop=>top
        return false
        .ENDM


        // INP-array ANALYSIEREN.
        // RESULTAT AUF DEN stack

INTEGR: top[SP,2]=>R0; 0=>[R1,R2]
        while @R0' eq BLANK do R0+
        while @R0' se '0 & (R0)+'=>R2 le '9 do
        begin
          R2 bic '0
          R1*10.; if carry then ERROR
          R1+R2 ; if carry then ERROR
        end
        R1=>top[2]
        return true


        // EINE ZEILE IN DEN array INP EINLESEN

START:  print 'ZAHL EINGEBEN  '
        ref(INP)=>R1=>push
        repeat .TTYIN ((R1)+) until R0 eq LF

        if !INTEGR : print /'ZAHL=', %pop,  : print /'FEHLER',
        so START

        .END START
```

```
 R INTEG

ZAHL EINGEBEN  1234
ZAHL=1234

ZAHL EINGEBEN  123456789
FEHLER

ZAHL EINGEBEN  332
ZAHL=332

ZAHL EINGEBEN  ^C

.
```

12.2 Polynomwert Berechnung

```
        // POLYNOM:
        // 5.4*X*X*X-0.72*X*X+2.4*X-12.5=0

array   real FAKT = 5.4, -0.72, 2.4, -12.5;
real    X;

POLYNM: (R5)+=>R0-2                 // PARAMETER ANZAHL
        @(R5)+**=>[AC3,AC0]         // X=>AC3; FAKT[1]=>AC0

        for R0 do AC0*AC3+@(R5)+**
        return

START:  read 'BITTE ZAHL EINGEBEN ', %X
        call POLYNM(X,FAKT[1],FAKT[2],FAKT[3],FAKT[4])
        print /'RESULT = ', %AC0,
        so START

ENDE:   .END START
```

```
 R POLYNM

BITTE ZAHL EINGEBEN 0
RESULT = -12.500

BITTE ZAHL EINGEBEN 2
RESULT = 32.620

BITTE ZAHL EINGEBEN 123.45
RESULT = 1.0148E 7

BITTE ZAHL EINGEBEN -11.3
RESULT = -07923.

BITTE ZAHL EINGEBEN ^C

.
```


12.3 Quadratische Gleichung

Die Lösung der quadratischen Gleichung

$$Ax^2 + Bx + C = \emptyset$$

ist gegeben durch:

$$X_{1/2} = \frac{1}{2A}\left(-B \pm \sqrt{B^2 - 4AC}\right)$$

(Imaginäre Lösungen sollen unberücksichtigt bleiben).
Die Berechnung der Wurzel erfolgt in einer FORTRAN-Subroutine.

```
        // QUADRATISCHE GLEICHUNG
        // LOESUNG FUER A=2, B=32, C=96

        .GLOBL  WURZEL

real    A=2, B=32., C=96.;
real    VIER=4, DISKR, WRZ, X1,X2;

START:  if A=>AC0 eq then print !%A; so ENDE

        VIER=>AC1*A*C
        B=>AC0*AC0-AC1
        if AC0 lt then print  'DISKR = '%AC0

        call WURZEL(AC0=>DISKR,WRZ)

        2=>AC1*A
        B=>AC0 neg+WRZ/AC1=>X1
        B=>AC0 neg-WRZ/AC1=>X2

        print !%X1,,!%X2

ENDE:   .END START
```

```
        SUBROUTINE WURZEL(ARGIN,ARGOUT)
        ARGOUT=SQRT(ARGIN)
        RETURN
```

```
 R QUADR

X1=-04
X2=-12
.
```

12.4 Größter Gemeinsamer Teiler

(Das Beispiel ist entnommen aus:
BARRON: "Rekursive Techniken in der Programmierung", Carl Hanser, München).
Der Algorithmus für den größten gemeinsamen Teiler zweier positiver, ganzer Zahlen kann wie folgt geschrieben werden:

```
GGT(n,m) = [m>n → GGT(m,n),
            m=∅ → n,
            GGT(m, REST(n,m))]
```

REST(n,m) ist eine Funktion, deren Wert der REST der ganzzahligen Division von n durch m ist).

Für die maschinenabhängigen Aufgaben der Parameterübergabe und der ganzzahligen Division werden zwei Macros (PARMS und REST) eingeführt. Der Algorithmus kann dann direkt nach der rekursiven Definition von GGT hingeschrieben werden.

```
        // GROESSTER GEMEINSAMER TEILER

        .RADIX 10

array word X=   493,    40,     207,    99,     11682;
array word Y=   1508,   15,     437,    63,     1475;
array string GG='       GGT =   ';
word    N,M;
lit     TAB=^O11;

        .RADIX

        macro   PARMS(word:N,M)
        (R5)+
        @(R5)+=>[push,push]
        pop=>[M,N]
        .ENDM


        macro REST(word:N,M)
        N=>R1
        0=>R0/M; R1=>N
        .ENDM
```



```
GGT:    PARMS(N,M)
        if M gt N then call GGT(M,N)    else
        begin
          if M eq then print /#ref(GG),%N; return  else
          begin
            REST(N,M)
            call GGT(M,N)
          end
        end
        return

START:  for size(X)=>R2 do
        begin
          X[R2]=>N;  Y[R2]=>M
          print %N,TAB,%M
          call GGT(N,M)
        end

        .END START
```

```
 R GGT

11682     1475    GGT =   59
99        63      GGT =   9
207       437     GGT =   23
40        15      GGT =   5
493       1508    GGT =   29
.
```

12.5 Binäres Suchen

Alle END-Symbole von SIMPL11 alphabetisch sortieren und sieben Symbole pro Zeile ausdrucken. Die Symbole liegen im Array VOKAB unsortiert vor. Das Programm trägt jedes Symbol aus VOKAB in den Array SYMBOL alphabetisch geordnet ein.

```
        // BINAERES SUCHEN

array   real VOKAB = $end,$begin,$else,$until,$for,$abs,$if,$return,
        $lit,$byte,$word,$real,$array,$string,$stack,$then,$while,
        $do,$repeat,$go,$jsr,$R0,$R1,$R2,$R3,$R4,$R5,$SP,$AC0,$AC1,
        $call,$ref,$size,$macro,$push,$pop,$top,$dump,$dumps,$dstack,
        $endm,$list,$nlist,$trap,$ldfps,$stfps,$stst,$ldexp,$stexp,
        $asr,$asl,$swab,$adc,$sbc,$sxt,$bit,$bic,$bis,$ash,$ashc,
        $st,$se,$eq,$ne,$lt,$le,$higher,$hisame,$lower,$losame,$true,
        $print,$absf,$read,$AC2,$AC3,$AC4,$AC5,$dreg,$radix,$limit,
        $com,$neg,$ror,$rol,$xor,$modf,$false,$carry,$oflow;
array   real SYMBOL[200.];
word    HTRAG=0, UNTEN, MITTE, OBEN;
real    NAME;
lit     TAB=11;

                // ALLE EINTRAEGE AUS VOKAB
                // SORTIERT NACH SYMBOL EINTRAGEN

MAIN:   for size(VOKAB)=>R2 do
        begin
          VOKAB[R2]=>push
          !BNSUCH
        end

                // 7 NAMEN PRO ZEILE AUSDRUCKEN

        0=>R2
        7=>AC1; 1=>AC0/AC1
        for size(VOKAB)=>R0 do
        begin
          print /- TAB,$SYMBOL[R2+]
          if R2=>AC1 modf AC0 lt AC0 : print
        end
        .EXIT
```



```
                // BINAERER SUCHALGORITHMUS

BNSUCH: pop=>[R0,NAME]
        R0=>push

                // 1. NAMEN GESONDERT EINTRAGEN

        if HTRAG : NAME=>SYMBOL[HTRAG+]; return

        1=>UNTEN; HTRAG=>OBEN
        repeat
        begin
          UNTEN=>MITTE+OBEN asr
          if NAME eq SYMBOL[MITTE] then return
          if lower : MITTE=>OBEN-1 : MITTE=>UNTEN+1
        end until UNTEN st OBEN

                // NAME NOCH NICHT VORHANDEN.
                // DURCH VERSCHIEBEN PLATZ SCHAFFEN

        HTRAG+=>[R0,R1]
        R0-
        while R0 se UNTEN do
        begin
          SYMBOL[R0]=>SYMBOL[R1]
          [R0,R1]-
        end

        NAME=>SYMBOL[UNTEN]
        return

ENDE:   .END MAIN
```

```
R BNSUCH

ABS     ABSF    AC0     AC1     AC2     AC3     AC4
AC5     ADC     ARRAY   ASH     ASHC    ASL     ASR
BEGIN   BIC     BIS     BIT     BYTE    CALL    CARRY
COM     DO      DREG    DSTACK  DUMP    DUMPS   ELSE
END     ENDM    EQ      FALSE   FOR     GE      GO
GT      HIGHER  HISAME  IF      JSR     LDEXP   LDFPS
LE      LIMIT   LIST    LIT     LOSAME  LOWER   LT
MACRO   MODF    NE      NEG     NLIST   OFLOW   POP
PRINT   PUSH    RADIX   READ    REAL    REF     REPEAT
RETURN  ROL     ROR     R0      R1      R2      R3
R4      R5      SBC     SIZE    SP      STACK   STEXP
STFPS   STRING  STST    SWAB    SXT     THEN    TOP
TRAP    TRUE    UNTIL   WHILE   WORD    XOR
```

ANHANG

A. Implementierung

A.1 ÜBERSETZUNGS-Technik

SIMPL11 ist als Preprocessor zum Macro Assembler der DEC-Betriebssysteme für die PDP11 implementiert. Ein SIMPL11-Quellprogramm wird in ein Assembler-Quellprogramm übersetzt und kann anschließend mit den üblichen Betriebssystem-Hilfsmitteln weiterverarbeitet werden.

Die gegenwärtige Implementierung erfolgte im Betriebssystem RT11. Der Compiler kann jedoch nach Änderung spezifischer vom Betriebssystem abhängiger Ein/Ausgabe-Macros in jedem anderen Betriebssystem, das den Macro-Assembler unterstützt, verwendet werden.

Die Implementierung des SIMPL11-Compilers erfolgte im Macro-Assembler. Der benötigte Kernspeicherbedarf beträgt - ohne Verwendung von "Overlays" - ca. 15k Worte. Mit Overlays kann der Platzbedarf auf etwa die Hälfte reduziert werden.

Code Erzeugung

SIMPL11 Quellprogramme werden in einem Durchlauf durch rekursiven Abstieg syntaktisch analysiert. Die Code Erzeugung, d.h. die Generierung des Assembler Quelltextes, erfolgt parallel zur Syntax Analyse, sobald eine Produktionsregel der Syntax-Beschreibung zutrifft.

Die maschinennahe Struktur von SIMPL11 impliziert, daß die meisten SIMPL11-Anweisungen genau einer Maschinenanweisung entsprechen. In diesen Fällen kann direkt die jeweilige Assembler-Instruktion generiert werden. In den anderen Fällen, in denen einer SIMPL11-Anweisung mehrere Maschinenanweisungen entsprechen, wird ein Macro-Aufruf erzeugt, mit Macro Parametern, die die Art der Operation beschreiben. Die eigentliche Code Erzeugung erfolgt dann später in der Assemblierungsphase während der Macro Expansion. Die Macro Definitionen sind zusammen mit einigen vom Compiler benötigten Hilfsgrößen in einem besonderen File, mit dem Namen MAC.FIX enthalten. Dieses File wird vor jedes zu übersetzende Programm kopiert. Das File MAC.FIX kann jederzeit neu editiert werden, um bei Bedarf die Generierungs-Macros zu ändern, ohne in den Compiler selbst einzugreifen.

Die wichtigsten Macros des Files MAC.FIX sind im Folgenden aufgelistet und kurz beschrieben:

ARRY$$: Index-Berechnung von indizierten Variablen.
CMPR$$: Vergleich von REAL-Variablen (keine Float-Register)
CMPA$$: Vergleich von Float-Registern
CLRR$$: REAL-Variable gleich Null setzen
MOVR$$: Zuweisung von REAL-Variablen
CALL$$: Fortran Subroutinen Aufruf
E/A-Macros: Diverse Macros für Ein/Ausgabe

A.2 Compiler Optimierungen

Die Optimierung der Übersetzung spielt bei SIMPL11-Programmen nur eine untergeordnete Rolle, da die meisten Sprachkonstruktionen eindeutig in Assembleranweisungen zu übersetzen sind. Ausnahmen sind die beiden folgenden Fälle:

1.) Ausdrücke der Art: Ø → <Operand>.
Abhängig vom Typ des Operanden wird entweder eine CLEAR-Anweisung erzeugt,
CLRB <Byte-Operand>
CLR <Word-Operand>
CLRF <Float-Register> ,

oder ein Macro-Aufruf

CLRR$$ <Real Operand ≠ Float Register> .

2.) Array-Zugriffe mit Indizes.
Im allgemeinen Fall wird die Adresse der indizierten Array-Komponente in einer internen Hilfsvariablen berechnet, gesteuert durch das Macro ARRY$$.

Beispiel:
Für einen Operanden XY[IX] wird in Abhängigkeit vom Datentyp folgender Code erzeugt:

XY: BYTE	XY: WORD	XY: REAL

MOV IX,$00
ADD #XY-1,$00
	MOV IX,$00
	ASL $00
	ADD #XY-2,$00
		MOV IX,$00
		ASL $00
		ASL $00
		ADD #XY-4,$00

Die Hilfsvariable $ØØ enthält die Adresse der indizierten Komponente.
In zwei Fällen kann der Array-Zugriff optimiert werden:

i) Der Index ist eine numerische Konstante. Der erzeugte Code ist:
XY + <IX - 1 * FAKT>
Der geklammerte Ausdruck wird während der Assembler-Übersetzung von Links nach Rechts ausgewertet. Je nach dem, ob der Typ von XY BYTE, WORD, REAL ist, nimmt FAKT die Werte 1, 2, 4 an.

ii) XY ist ein BYTE-Array und der Index ist ein Register. In diesem Falle wird ein Index-Register-Zugriff erzeugt:

XY-1(Rn)

A.3 Array-Zugriffs Optimierung durch den Benutzer

Die Berechnung der Adresse einer Array-Komponente erfolgt im allgemeinen Falle in einer von zwei Hilfsgrößen $ØØ bzw. $Ø1, die im File MAC.FIX definiert sind. Für jeden Zugriff zu $ØØ bzw. $Ø1 werden zwei Worte benötigt (Relative Adressierung, Adressierungs-Modus = 67). Der Platzbedarf läßt sich für jeden Zugriff auf ein Wort reduzieren, wenn $ØØ und $Ø1 in Register umdefiniert werden, wie das Beispiel demonstriert:

```
ARRAY WORD XY[100];
WORD IX;

%
$C0=R0                  ; UMDEFINITION VON $00 UND $01
$01=R1
%
        XY[IX]=>PUSH    // INDEX BERECHNUNG IN REGISTERN
%
$00=BASE$$              ;$00 UND $01 ADRESSIEREN WIEDER DIE
$01=BASE$$+2            ;ALTEN SPEICHER PLAETZE
%
        XY[IX]=>PUSH    // INDEX BERECHNUNG IN SPEICHER-
                        // PLAETZEN
        .END
*
```

Zur Umdefinition in Register muß zum Assembler umgeschaltet werden. BASE$$ ist eine in MAC.FIX definierte Variable, die die Adresse von $ØØ enthält. Der erzeugte Code ist:


```
0000            XY:     .BLKW   100             ;* ARRAY WORD XY[100];
                        .EVEN
0200 000000     IX:     .WORD   0               ;* WORD IX;
                        .EVEN

     000000 $00=R0                      ; UMDEFINITION VON $00 UND $01
     000001 $01=R1

0202                    ARRY$$  ,IX,,XY,'W,$00, ;* XY[IX]=>PUSH // INDEX
0202 016700             MOV     IX,$00
     177772
0206 006300             ASL     $00
0210 062700             ADD     #XY-2,$00
     177776'
0214 011046             MOV     @$00,-(SP)

     000000'$00=BASE$$              ;$00 UND $01 ADRESSIEREN WIEDER DIE
     000002'$01=BASE$$+2            ;ALTEN SPEICHER PLAETZE

0216                    ARRY$$  ,IX,,XY,'W,$00, ;* XY[IX]=>PUSH // INDEX
0216 016767             MOV     IX,$00
     177756
     000000'
0224 006367             ASL     $00
     000000'
0230 062767             ADD     #XY-2,$00
     177776'
     000000'
0236 017746             MOV     @$00,-(SP)
     000000'

                                            ;* // PLAETZEN
0242                    .EXIT
0242 104350                     EMT     ^0350
     000001'            .END
```

Aus dem Beispiel ist zu ersehen, daß für denselben Zugriff zu einem WORD-Array ohne Register 10 Worte benötigt werden, mit Register jedoch nur 6 Worte! Auf diese Weise läßt sich bei vielen Array-Zugriffen Speicherplatz sparen, allerdings auf Kosten zweier Register.

A.4 FORTRAN-Subroutinen Aufruf

FORTRAN Subroutinen werden mit einer CALL-Anweisung vereinfacht aufgerufen, nach der Syntax:

```
CALL <SUBROUTINE-IDENTIFIER> ( <EXPR 1> , <EXPR 2> , ... )
```

Als Parameter sind beliebige Ausdrücke erlaubt. Die Ausdrücke werden erst ausgewertet und die resultierenden Operanden anschließend im Macro 'CALL$$' als Parameter übergeben. Das Macro CALL$$ ist im File MAC.FIX definiert.

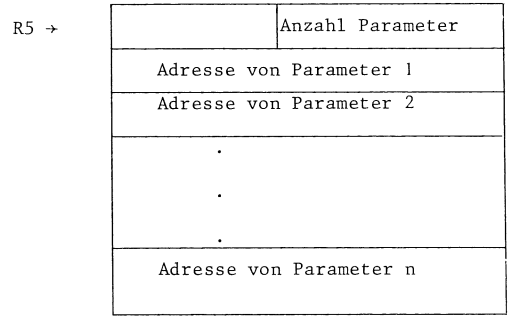
Parameter-Übergabe im RT11-Fortran

Für die Parameter-Übergabe in Fortran Subroutinen gelten in den Fortran Compilern der DEC-Betriebssysteme unterschiedliche Konventionen.

Im RT11-Fortran ist folgende Konvention zu beachten:

a) Aufruf: JSR PC,IDENTIFIER
   mit Inhalt von Register 5 = Adresse einer Adreßliste

b) Format der Adreßliste:



Das Macro CALL$$ erzeugt einen dieser Konvention entsprechenden Subroutinen-Aufruf. Für die Parameter gelten in Abhängigkeit vom Adressierungsmodus folgende Regeln:

i) Bei Relativer Adressierung (Modus = 67) ist die Parameter-Adresse zur Übersetzungszeit bekannt und kann direkt in die Adreßliste eingesetzt werden.

ii) Bei allen anderen Adressierungen (MODUS ≠ 67) enthält der Operand zur Laufzeit die Adresse des Parameters. Diese Adresse wird vor dem Subroutine-Aufruf explizit in die Adressliste eingeschrieben.

Beispiel:

```
CALL SUBR (XY, @A1, (R0)+)
```

Erzeugter Macro-Aufruf:

```
CALL$$  SUBR,<XY, @A1, (R0)+>
```

In diesem Beispiel ist nur Parameter XY relativ adressiert. Folgende Sequenz von Assembleranweisungen wird erzeugt:

```
        CALL$$  SUBR,<XY,@A1,(R0)+>      ;* CALL SUBR (XY,@A1,(R0)+)
017767   MOV    @A1,64$+LX$$$$
177772
000024
012067   MOV    (R0)+,64$+LX$$$$
000022
010546   MOV    R5,-(SP)
012705   MOV    #64$,R5
000032'
004767   JSR    PC,SUBR
000006
000404          BR      65$
000003 64$:     .WORD   SYM$$$
000000'         .WORD   XY
000000          .WORD   0
000000          .WORD   0
012605 65$:     MOV     (SP)+,R5
```

Durch Änderung der Macro-Definition von CALL$$ in MAC-FIX kann der Aufruf jeder Compiler- oder auch Benutzer-Konvention angepaßt werden.

B. Vordefinierte Macros

B.1 Konversions-Macros

Zur Konversion zwischen verschiedenen Zahlensystemen sind sechs Macros vordefiniert. Sie entsprechen im Aufruf weitgehend den gleichnamigen Macros aus dem DOS-Betriebssystem.

BIN2D (ADR,WORD)

Ein Binäres Wort in 5 Dezimale ASCII-Zeichen konvertieren. ADR = Adresse des ersten Bytes des Buffer für die Zeichen. WORD = Zu konvertierende Zahl

D2BIN (ADR)

5 Dezimale ASCII-Zeichen in ein Binäres Wort konvertieren. ADR = Adresse des ersten Zeichens

Auf dem Stack werden zwei Worte zurückgegeben. Oben auf dem Stack befindet sich das konvertierte Wort. Darunter ist die Adresse des ersten Bytes nach der Konversion. Der Benutzer muß den Stack wieder aufräumen.

BIN2O (ADR,WORD)

Ein Binäres Wort in 6 Octale ASCII-Zeichen konvertieren. ADR = Adresse des ersten Bytes des Buffers für die Zeichen. WORD = Zu konvertierende Zahl

O2BIN (ADR)

6 Oktale ASCII-Zeichen in ein Binäres Wort konvertieren.
ADR = Adresse des ersten Zeichens.

Auf dem Stack werden 2 Worte zurückgegeben. Oben auf dem Stack befindet sich das konvertierte Wort. Darunter ist die Adresse des ersten Bytes nach der Konversion. Der Benutzer muß den Stack aufräumen.

RADPK (ADR1,ADR2)

Die Zeichen zwischen ADR1 (einschließlich) und ADR2 (ausschließlich), maximal jedoch 6 Zeichen in die RAD50-Darstellung konvertieren. Der Parameter ADR2 darf fehlen; dann werden implizit 6 Zeichen umgewandelt.

Es werden zwei RAD50-Worte auf den Stack zurückgegeben. Der Benutzer muß den Stack aufräumen.

RADUP (ADR,WORD)

Ein RAD50-Wort in 3 ASCII-Zeichen konvertieren.
ADR = Adresse des ersten Bytes des Buffers für die Zeichen
WORD = Zu konvertierendes RAD50-Wort.

B.2 Index-Macro für zweidimensionale Arrays

In SIMPL11 sind nur eindimensionale Arrays definiert. Mehr-dimensionale Arrays müssen immer linear dargestellt werden. Für den Zugriff zu zweidimensionalen Arrays ist das Macro INDEX2 vordefiniert:

INDEX2 (ARRAY,BWR,SPALTEN,I,J)

ARRAY = Array Adresse
BWR = 'B, 'W, 'R für BYTE, WORD bzw. REAL-Array
SPALTEN = Anzahl der Arrayspalten
I, J = Zeilen-, Spalten-Index

Der Array muß zeilenweise abgespeichert sein. Auf dem Stack ist die Adresse der indizierten Array-Komponente. Der Benutzer muß den Stack aufräumen.

C. Benutzung im Betriebssystem RT11

C.1 SIMPL11-Übersetzung

SIMPL11 wird in dem für DEC üblichen Kommando-Format benutzt. Es können ein oder mehrere SIMPL11-Quellprogramme in ein Assembler-Quellprogramm übersetzt werden.

Kommando-Format:

*DEV:AS-SOURCE,DEV:LIST<DEV:SOURCE1,..., DEV:SOURCE n

DEV: Beliebiges RT11 Device
AS-Source: Assembler Quellprogramm
LIST: Formatiertes SIMPL11-Quellprogramm (Kleinschreibung der SIMPL11 End-Symbole, Einrücken von Blöcken)
SOURCE: SIMPL11-Quellprogramm

Es gelten folgende implizite File-Extensionen:

AS-Source: .SPL
LIST: .LST
SOURCE: -

Beispiel:

```
.R SIMPL
*PARSER,TT:<DT0:SECT,SY:GLOB,PARSER
```

Die drei SIMPL11 Quell-Programme: SECT,GLOB,PARSER werden in ein Assembler Quell-Programm PARSER.SPL übersetzt. Die formatierte Programmliste wird auf dem Terminal ausgegeben.

C.2 Assembler-Übersetzung und Laden

Das von SIMPL11 erzeugte File mit der Extension .SPL wird vom Macro-Assembler weiter übersetzt.

Beispiele:

```
.R MACRO
*PARSER<PARSER.SPL
^C
```

Laden:

```
.R LINK
PARSER<PARSER,SLIB
^C
```

Bei der Ladeprozedur muß immer das File SLIB.OBJ mitgeladen werden. Es enthält alle zur Laufzeit benötigten Subroutinen.

Anmerkung:

Die Schreibarbeit für das zweimalige Übersetzen kann durch Verwendung eines Batch-Files reduziert werden. Die Batch-Benutzung empfiehlt sich, wenn dasselbe Programm häufig, z.B. beim Testen, übersetzt werden muß.

Beispiel:

Erzeugung eines Batch-Files mit Namen PARSER.BAT:

```
$JOB/RT11
.R SIMPL
*PARSER,TT:<DTO:SECT,SY:GLOB,PARSER
.R MACRO
*PARSER<PARSER.SPL
$EOJ
```

Für die gemeinsame SIMPL11- und Assembler-Übersetzung muß dann nur noch geschrieben werden:

```
.R BATCH
*PARSER
```


D. Beispiele für generierten Code

D.1 Fibonacci Zahlen

```
                        ;** // SIMPL11 PROGRAMM BEISPIEL:
                        ;** // BERECHNUNG UND AUSGABE DER  FIBONACCI ZAHLEN
                        ;** ARRAY               WORD FIBONI25.];
0000            FIBON:  .BLKW   25.
                .EVEN
0062 000002 IX:         .WORD   2
                .EVEN
000011 TAB=11
                .EVEN
0064            FIBO:   PRINT$  #0,-(SP)
0064 012746             MOV     #0,-(SP)
0066 000000
0070 004767
0074 000000G
0074 012746             PRINT$  #$00001
0100 000404             MOV     #$00001,-(SP)
0104 004767
0110 000000G
0104 004767             JSR     PC,PRINT
0110 000000G
0114 012667             RDDEC$  FIBON+<1-1*2>,W
177664                  JSR     PC,RDWDC$
0114 012746             MOV     (SP)+,FIBON+<1-1*2>
0120 000000
0122 004767
0126 000000G
                        ;** READ '2. ZAHL = ', %FIBONI2];
0132            PRINT$  MOV     #0,-(SP)
0132 012746             MOV     #0,-(SP)
0134 000000
0136 004767             JSR     PC,PRINT
0142 000000G
0144            FIBO:   PRINT$  #$00002
0144 012746             MOV     #$00002,-(SP)
0146 000002
0150 000417'
0152 004767             JSR     PC,PRINT
0156 000000G
0160            RDDEC$  FIBON+<2-1*2>,W
0160 012667             JSR     PC,RDWDC$
0164 000000G
0166 004767             MOV     (SP)+,FIBON+<2-1*2>
0172 000000G
0174 012667
177636
```

```
0246 062767          ADD     #FIBON-2,$01
     177776'
     000002'
0254 012677          MOV     (SP)+,@$01
     000002'
0260 026727          CMP     IX,#25.          ;* END UNTIL IX EQ SIZE(FIBON)
     177576
     000031
0266 001332          BNE     $00003
0270 005067 $00004:  CLR     IX               ;* 0=>IX
     177566
0274 012700 LOOP:    MOV     #5,R0            ;* LOOP:        FOR 5=>R0 DO
     000005
0300        $00005:
0300 026727          CMP     IX,#25.          ;*   IF IX LT SIZE(FIBON) THEN PRINT /%FIBON[IX+1],
     177556
     000031
0306 002024          BGE     $00007
0310 005267 $00006:  INC     IX
     177546
0314                 ARRY$$  ,IX,,FIBON,'W,$00,
0314 016767           MOV    IX,$00
     177542
     000000'
0322 006367           ASL    $00
     000000'
0326 062767           ADD    #FIBON-2,$00
     177776'
     000000'
0334                 PRDEC$  @$00,'W
0334 017746           MOV    @$00,-(SP)
     000000'
0340 004767           JSR    PC,BWDEC$
     000000G
0344                 PRASC$  #TAB,'W
0344 012746           MOV    #TAB,-(SP)
     000011
0350 004767           JSR    PC,BWASC$
     000000G
0354 000167          JMP     $00008
     000004
```



```
0144                 PRINT$
0144 012746           MOV    #0,-(SP)
     000000
0150 004767           JSR    PC,PRINT
     000000G
0154        $00003:                          ;* REPEAT
0154                 ARRY$$  ,IX,,FIBON,'W,$00,      ;*   FIBON[IX]=>PUSH+FIBON[IX=>R0-]
0154
0154 016767           MOV    IX,$00
     177702
     000000'
0162 006367           ASL    $00
     000000'
0166 062767           ADD    #FIBON-2,$00
     177776'
     000000'
0174 017746          MOV     @$00,-(SP)
     000000'
0200 016700          MOV     IX,R0
     177656
0204 005300          DEC     R0
0206                 ARRY$$  ,R0,,FIBON,'W,$00,
0206 010067           MOV    R0,$00
     000000'
0212 006367           ASL    $00
     000000'
0216 062767           ADD    #FIBON-2,$00
     177776'
     000000'
0224 067716          ADD     @$00, (SP)
     000000'
0230 005267          INC     IX               ;*   POP=>FIBON[IX+1]
     177626
0234                 ARRY$$  ,IX,,FIBON,'W,$01,
0234 016767           MOV    IX,$01
     177622
     000002'
0242 006367           ASL    $01
     000002'
```

```
                        .MCALL  .TTYIN
0000            INP:    .BLKB   100             ;* ARRAY         BYTE INP[100];
                        .EVEN
000012 LF=12
000040 BLANK=40                                 ;* LIT  LF=12,BLANK=40;
                        .EVEN
                        .MACRO  ERROR,          ;* MACRO ERROR
                        MOV     (SP)+,(SP)      ;* POP=>TOP
                        CLZ
                        RTS     PC
                        .ENDM
                                                ;* // INP-ARRAY ANALYSIEREN.
                                                ;* // RESULTAT AUF DEN STACK
0100 016600 INTEGR: MOV     00002.(SP),R0       ;* INTEGR:       TOP[SP,2]=>R0; 0=>[R1,R2]
     000002
0104 005001         CLR     R1
0106 005002         CLR     R2

0110            $00001:                         ;* WHILE @R0' EQ BLANK DO R0+
0110 121027         CMPB    @R0,#BLANK
     000040
0114 001003         BNE     $00003
0116 005200 $00002: INC     R0
0120 000167         JMP     $00001
     177764
0124            $00003:
0124            $00004:                         ;* WHILE @R0' GE '0 & (R0)+'=>R2 LE '9 DO
0124 121027         CMPB    @R0,#'0
     000060
0130 002423         BLT     $00006
0132 112002         MOVB    (R0)+,R2
0134 120227         CMPB    R2,#'9
     000071
0140 003017         BGT     $00006
0142 142702 $00005: BICB    #'0,R2              ;*   R2 BIC '0
     000060
0146 070127         MUL     #10.,R1             ;*   R1*10.; IF CARRY THEN ERROR
     000012
0152 103003         BCC     $00008
0154            $00007: ERROR
```

```
0360            $00007:
0360 000167         JMP     ENDE
     000016
0364 077033 $00008: SOB     R0,$00005           ;* END
0366                PRINT$                      ;* PRINT; GO LOOP
0366 012746          MOV    #0,-(SP)
     000000
0372 004767          JSR    PC,PRINT
     000006
0376 000167         JMP     LOOP
     177672
0402            ENDE:   .EXIT
0402 104350                     EMT     ^0350
0404    061 $00001: .ASCIZ  '1. ZAHL = '
0405    056
0406    040
0407    132
0410    101
0411    110
0412    114
0413    040
0414    075
0415    040
0416    000
0417    062 $00002: .ASCIZ  '2. ZAHL = '
0420    056
0421    040
0422    132
0423    101
0424    110
0425    114
0426    040
0427    075
0430    040
0431    000
     000064'        .END FIBO
```

```
0260          $00014: PRINT$   #$00016
0260 012746           MOV      #$00016,-(SP)
     000360'
0264 004767           JSR      PC,PRINT
     000000G
0270                  PRDEC$   (SP)+,'W
0270 012646           MOV      (SP)+,-(SP)
0272 004767           JSR      PC,BWDEC$
     000000G
0276                  PRINT$
0276 012746           MOV      #0,-(SP)
     000000
0302 004767           JSR      PC,PRINT
     000000G
0306 000167           JMP      $00017
     000020
0312          $00015:
0312                  PRINT$   #$00018
0312 012746           MOV      #$00018,-(SP)
     000366'
0316 004767           JSR      PC,PRINT
     000000G
0322                  PRINT$
0322 012746           MOV      #0,-(SP)
     000000
0326 004767           JSR      PC,PRINT
     000000G
0332 000167 $00017:   JMP      START
     177652
0336                  .EXIT
0336 104350                    EMT      ^0350
0340    132 $00011:   .ASCIZ   'ZAHL EINGEBEN  '
0341    101
0342    110
0343    114
0344    040
0345    105
0346    111
0347    116
0350    107
0351    105
0352    102
0353    105
0354    116
0355    040
0356    040
0357    000
0360    132 $00016:   .ASCIZ   'ZAHL='
0361    101
0362    110
0363    114
0364    075
0365    000
0366    106 $00018:   .ASCIZ   'FEHLER'
0367    105
0370    110
0371    114
0372    105
0373    122
0374    000
     000210'          .END START
```

89

```
0154 012616           MOV      (SP)+,(SP)       ;* POP=>TOP
0156 000244           CLZ
0160 000207           RTS      PC
0162 060201 $00008:   ADD      R2,R1            ;*   R1+R2 ; IF CARRY THEN ERROR
0164 103003           BCC      $00010
0166          $00009: ERROR
0166 012616           MOV      (SP)+,(SP)       ;* POP=>TOP
0170 000244           CLZ
0172 000207           RTS      PC
0174 000167 $00010:   JMP      $00004           ;* END
     177724
0200          $00006:
0200 010166           MOV      R1,00002.(SP)    ;* R1=>TOP[2]
     000002
0204 000264           SEZ
0206 000207           RTS      PC
                                                ;* // EINE ZEILE IN DEN ARRAY INP EINLESEN
0210          START:  PRINT$                    ;* START:       PRINT 'ZAHL EINGEBEN  '
0210 012746           MOV      #0,-(SP)
     000000
0214 004767           JSR      PC,PRINT
     000000G
0220                  PRINT$   #$00011
0220 012746           MOV      #$00011,-(SP)
     000340'
0224 004767           JSR      PC,PRINT
     000000G
0230 012701           MOV      #INP,R1 ;* REF(INP)=>R1=>PUSH
     000000'
0234 010146           MOV      R1,-(SP)
0236          $00012: .TTYIN   (R1)+            ;* REPEAT .TTYIN ((R1)+) UNTIL R0 EQ LF
0236 104340                            EMT      ^0340
0240 103776                            BCS      .-2
0242 110021 .IIF NB <(R1)+>,           MOVB     %0,(R1)+
0244 020027           CMP      R0,#LF
     000012
0250 001372           BNE      $00012
0252 004767 $00013:   JSR      PC,INTEGR        ;* IF !INTEGR : PRINT /'ZAHL=', %POP,  : PRINT /'FEHLER',
     177622
0256 001015           BNE      $00015
```

88

```
0072                    RDFLT$  ,X,'R
0072 004767              JSR    PC,RDRFL$
     000000G
0076 012667              MOV    (SP)+,X
     177716
0102 012667              MOV    (SP)+,X+2
     177714
0106                    CALL$$  POLYNM,<X,FAKT+<1-1*4>,FAKT+<2-1*4>,FAKT+<3-1*4>,FAKT+<4-1*4>>
0106 010546              MOV    R5,-(SP)
0110 012705              MOV    #64$,R5
     000122'
0114 004767              JSR    PC,POLYNM
     177704
0120 000406              BR     65$
0122 000005 64$:        .WORD   SYM$$$
0124 000020'            .WORD   X
0126 000000'            .WORD   FAKT+<1-1*4>
0130 000004'            .WORD   FAKT+<2-1*4>
0132 000010'            .WORD   FAKT+<3-1*4>
0134 000014'            .WORD   FAKT+<4-1*4>
0136 012605 65$:        MOV     (SP)+,R5
0140                    PRINT$  #$00003          ;* PRINT /'RESULT = ', %AC0,
0140 012746              MOV    #$00003,-(SP)
     000221'
0144 004767              JSR    PC,PRINT
     000000G
0150                    PRFLT$  ,AC0,'A
0150 174046              STF    AC0,-(SP)
0152 004767              JSR    PC,PRFLT$
     000000G
0156                    PRINT$
0156 012746              MOV    #0,-(SP)
     000000
0162 004767              JSR    PC,PRINT
     000000G
```



```
                                                ;* // POLYNOM:
                                                ;* // 5.4*X*X*X-0.72*X*X+2.4*X-12.5=0
0000 040654 FAKT:       .FLT2   5.4             ;* ARRAY        REAL FAKT = 5.4, -0.72, 2.4, -12.5;
0002 146315
0004 140070             .FLT2   -0.72
0006 050754
0010 040431             .FLT2   2.4
0012 114632
0014 141110             .FLT2   -12.5
0016 000000
                        .EVEN
0020 000000 X:          .FLT2   0               ;* REAL X;
0022 000000
                        .EVEN
0024 012500 POLYNM:     MOV     (R5)+,R0        ;* POLYNM:       (R5)+=>R0-2              // PARAMETER ANZAHL
0026 162700             SUB     #2,R0
     000002

0032 172735             LDF     @(R5)+,AC3      ;* @(R5)+''=>[AC3,AC0]  // X=>AC3; FAKT[1]=>AC0
0034 172435             LDF     @(R5)+,AC0
0036 172467             LDF     FAKT+<1-1*4>,AC0
     177736
0042 171003 $00001:     MULF    AC3,AC0         ;* FOR R0 DO AC0*AC3+@(R5)+''
0044 172035             ADDF    @(R5)+,AC0
0046 077003             SOB.    R0,$00001
0050 000207             RTS     PC
0052            START:  PRINT$                  ;* START:       READ 'BITTE ZAHL EINGEBEN ', %X
0052 012746              MOV    #0,-(SP)
     000000
0056 004767              JSR    PC,PRINT
     000000G
0062                    PRINT$  #$00002
0062 012746              MOV    #$00002,-(SP)
     000174'
0066 004767              JSR    PC,PRINT
     000000G
```

```
                                                    ;* // QUADRATISCHE GLEICHUNG
                                                    ;* // LOESUNG FUER A=2, B=32, C=96
                        .GLOBL  WURZEL
0000 040400 A:          .FLT2   2                   ;* REAL A=2, B=32., C=96.;
0002 000000
0004 041400 B:          .FLT2   32.
0006 000000
0010 041700 C:          .FLT2   96.
0012 000000
                        .EVEN
0014 040600 VIER:       .FLT2   4                   ;* REAL VIER=4, DISKR, WRZ, X1,X2;
0016 000000
0020 000000 DISKR:      .FLT2   0
0022 000000
0024 000000 WRZ:        .FLT2   0
0026 000000
0030 000000 X1:         .FLT2   0
0032 000000
0034 000000 X2:         .FLT2   0
0036 000000
                        .EVEN
0040 172467 START:      LDF     A,AC0               ;* START:        IF A=>AC0 EQ THEN PRINT !%A; GO ENDE
     177734
0044 170000             CFCC
0046 001020             BNE     $00002
0050            $00001: PRINT$
0050 012746              MOV    #0,-(SP)
     000000
0054 004767              JSR    PC,PRINT
     0000006
0060                    PRINT$  #$00003
0060 012746              MOV    #$00003,-(SP)
     000354'
0064 004767              JSR    PC,PRINT
     0000006
0070                    PRFLT$  ,A,'R
0070 016746              MOV    A+2,-(SP)
     177706
0074 016746              MOV    A,-(SP)
     177700
```



```
0166 000167             JMP     START               ;* GO START
     177660
0172            ENDE:   .EXIT
0172 104350                             EMT     ^0350
0174    102 $00002: .ASCIZ  'BITTE ZAHL EINGEBEN '
0175    111
0176    124
0177    124
0200    105
0201    040
0202    132
0203    101
0204    110
0205    114
0206    040
0207    105
0210    111
0211    116
0212    107
0213    105
0214    102
0215    105
0216    116
0217    040
0220    000
0221    122 $00003: .ASCIZ  'RESULT = '
0222    105
0223    123
0224    125
0225    114
0226    124
0227    040
0230    075
0231    040
0232    000
     000052'            .END START
```

```
0212 177127          LDCIF   #2,AC1          ;* 2=>AC1*A
     000002
0216 171167          MULF    A,AC1
     177556
0222 172467          LDF     B,AC0           ;* B=>AC0 NEG+WRZ/AC1=>X1
     177556
0226 170700          NEGF    AC0
0230 172067          ADDF    WRZ,AC0
     177570
0234 174401          DIVF    AC1,AC0
0236 174067          STF     AC0,X1
     177566
0242 172467          LDF     B,AC0           ;* B=>AC0 NEG-WRZ/AC1=>X2
     177536
0246 170700          NEGF    AC0
0250 173067          SUBF    WRZ,AC0
     177550
0254 174401          DIVF    AC1,AC0
0256 174067          STF     AC0,X2
     177552
0262                 PRINT$                  ;* PRINT !%X1,,!%X2
0262 012746           MOV    #0,-(SP)
     000000
0266 004767           JSR    PC,PRINT
     000000G
0272                 PRINT$  #$00007
0272 012746           MOV    #$00007,-(SP)
     000370'
0276 004767           JSR    PC,PRINT
     000000G
0302                 PRFLT$  ,X1,'R
0302 016746           MOV    X1+2,-(SP)
     177524
0306 016746           MOV    X1,-(SP)
     177516
0312 004767           JSR    PC,PRFLT$
     000000G
```



```
0100 004767           JSR    PC,PRFLT$
     000000G
0104 000167          JMP     ENDE
     000242
0110 172567 $00002:  LDF     VIER,AC1        ;* VIER=>AC1*A*C
     177700
0114 171167          MULF    A,AC1
     177660
0120 171167          MULF    C,AC1
     177664
0124 172467          LDF     B,AC0           ;* B=>AC0*AC0-AC1
     177654
0130 171000          MULF    AC0,AC0
0132 173001          SUBF    AC1,AC0
0134 170500          TSTF    AC0             ;* IF AC0 LT THEN PRINT  'DISKR = '%AC0
0136 170000          CFCC
0140 002010          BGE     $00005
0142        $00004:  PRINT$
0142 012746           MOV    #0,-(SP)
     000000
0146 004767           JSR    PC,PRINT
     000000G
0152                 PRINT$  #$00006
0152 012746           MOV    #$00006,-(SP)
     000357'
0156 004767           JSR    PC,PRINT
     000000G
0162 174067 $00005:  STF     AC0,DISKR       ;* CALL WURZEL(AC0=>DISKR,WRZ)
     177632
0166                 CALL$$  WURZEL,<DISKR,WRZ>
0166 010546           MOV    R5,-(SP)
0170 012705           MOV    #64$,R5
     000202'
0174 004767           JSR    PC,WURZEL
     000000G
0200 000403           BR     65$
0202 000002 64$:     .WORD   SYM$$$
0204 000020'         .WORD   DISKR
0206 000024'         .WORD   WRZ
0210 012605 65$:     MOV     (SP)+,R5
```

```
                                        ;* // GROESSTER GEMEINSAMER TEILER
     000012             .RADIX 10
0000 000755 X:          .WORD   493             ;* ARRAY WORD X=        493,    40,     207,    99,     11682;
0002 000050             .WORD   40
0004 000317             .WORD   207
0006 000143             .WORD   99
0010 026642             .WORD   11682
                        .EVEN
0012 002744 Y:          .WORD   1508            ;* ARRAY WORD Y=        1508,   15,     437,    63,     1475;
0014 000017             .WORD   15
0016 000665             .WORD   437
0020 000077             .WORD   63
0022 002703             .WORD   1475
                        .EVEN
0024    011 GG:         .ASCIZ  '       GGT =   '       ;* ARRAY STRING GG='    GGT =   ';
0025    107
0026    107
0027    124
0030    040
0031    075
0032    011
0033    000
                        .EVEN
0034 000000 N:          .WORD   0               ;* WORD N,M;
0036 000000 M:          .WORD   0
                        .EVEN
     000011 TAB=^O11                            ;* LIT  TAB=^O11;
                        .EVEN
     000000             .RADIX
                        .MACRO  PARMS,N,M       ;* MACRO        PARMS(WORD:N,M)
                        TST     (R5)+           ;* (R5)+
                        MOV     @(R5)+,-(SP)    ;* @(R5)+=>[PUSH,PUSH]
                        MOV     @(R5)+,-(SP)

                        MOV     (SP)+,M         ;* POP=>[M,N]
                        MOV     (SP)+,N

                        .ENDM
                        .MACRO  REST,N,M        ;* MACRO REST(WORD:N,M)
```



```
0316                    PRINT$
0316 012746              MOV    #0,-(SP)
     000000
0322 004767              JSR    PC,PRINT
     000000G
0326                    PRINT$  #$00008
0326 012746              MOV    #$00008,-(SP)
     000374'
0332 004767              JSR    PC,PRINT
     000000G
0336                    PRFLT$  ,X2,'R
0336 016746              MOV    X2+2,-(SP)
     177474
0342 016746              MOV    X2,-(SP)
     177466
0346 004767              JSR    PC,PRFLT$
     000000G
0352            ENDE:   .EXIT
0352 104350                             EMT     ^0350
0354    101 $00003: .ASCIZ  'A='
0355    075
0356    000
0357    104 $00006: .ASCIZ  'DISKR = '
0360    111
0361    123
0362    113
0363    122
0364    040
0365    075
0366    040
0367    000
0370    130 $00007: .ASCIZ  'X1='
0371    061
0372    075
0373    000
0374    130 $00008: .ASCIZ  'X2='
0375    062
0376    075
0377    000
     000040'            .END START
```

```
0140 004767          JSR     PC,BWDEC$
     000000G
0144 000207          RTS     PC
0146 000167          JMP     $00006
     000042'
0152         $00005:
0152                 REST    N,M                 ;*      REST(N,M)
0152 016701          MOV     N,R1                ;* N=>R1
     177656
0156 005000          CLR     R0                  ;* 0=>R0/M; R1=>N
0160 071067          DIV     M,R0
     177652
0164 010167          MOV     R1,N
     177644
0170                 CALL$$  GGT,<M,N>           ;*      CALL GGT(M,N)
0170 010546           MOV    R5,-(SP)
0172 012705           MOV    #64$,R5
     000204'
0176 004767           JSR    PC,GGT
     177636
0202 000403           BR     65$
0204 000002 64$:     .WORD   SYM$$$
0206 000036'         .WORD   M
0210 000034'         .WORD   N
0212 012605 65$:     MOV     (SP)+,R5
0214 000207 $00006:$00003:   RTS     PC
0216 012702 START:   MOV     #00005.,R2          ;* START:       FOR SIZE(X)=>R2 DO
     000005
0222         $00007: ARRY$$   ,R2,,X,'W,$00,      ;*   X[R2]=>N;  Y[R2]=>M
0222 010267           MOV    R2,$00
     000000'
0226 006367           ASL    $00
     000000'
0232 062767           ADD    #X-2,$00
     177776'
     000000'
0240 017767          MOV     @$00,N
     000000'
     177566
```

```
                     MOV     N,R1                ;* N=>R1
                     CLR     R0                  ;* 0=>R0/M; R1=>N
                     DIV     M,R0
                     MOV     R1,N
                     .ENDM
0040         GGT:    PARMS   N,M                 ;* GGT: PARMS(N,M)
0040 005725          TST     (R5)+               ;* (R5)+
0042 013546          MOV     @(R5)+,-(SP)        ;* @(R5)+=>[PUSH,PUSH]
0044 013546          MOV     @(R5)+,-(SP)
0046 012667          MOV     (SP)+,M             ;* POP=>[M,N]
     177764
0052 012667          MOV     (SP)+,N
     177756
0056 026767          CMP     M,N                 ;* IF M GT N THEN CALL GGT(M,N)          ELSE
     177754
     177750
0064 003414          BLE     $00002
0066         $00001: CALL$$   GGT,<M,N>
0066 010546           MOV    R5,-(SP)
0070 012705           MOV    #64$,R5
     000102'
0074 004767           JSR    PC,GGT
     177740
0100 000403           BR     65$
0102 000002 64$:     .WORD   SYM$$$
0104 000036'         .WORD   M
0106 000034'         .WORD   N
0110 012605 65$:     MOV     (SP)+,R5
0112 000167          JMP     $00003
     000076
0116         $00002:
0116 005767          TST     M                   ;*   IF M EQ THEN PRINT /#REF(GG),%N; RETURN  ELSE
     177714
0122 001013          BNE     $00005
0124         $00004: PRINT$   #GG
0124 012746           MOV    #GG,-(SP)
     000024'
0130 004767           JSR    PC,PRINT
     000000G
0134                 PRDEC$  N,'W
0134 016746           MOV    N,-(SP)
     177674
```

E. Fehler-Codes

Bei Syntaxfehlern wird im Ausgabefile eine Fehlermeldung erzeugt und diese gleichzeitig auf dem Terminal ausgegeben:

```
.ERROR <Error-Code>
;<Illegal-Statement>
```

Die Position des Fehlers wird durch ein Zeichen ↑ angezeigt.

Beispiel:

```
.ERROR  00264
;       R1=>PUSH+_^_SYMB-R0
```

Fehlercode 264 bedeutet ein undefiniertes Symbol. In der Assembler-Übersetzung wird die Direktive .ERROR als Fehlermeldung angezeigt. Die fehlerhafte Zeile selbst bleibt als Kommentar unberücksichtigt.

Beispiel für die Assembler-Fehlermeldung:

```
.R MACRO
*<TEST.SFL
******F
702 0710 000264 .ERROR  00264
ERRORS DETECTED: 1
FREE CORE: 14228. WORDS
```



```
0246                    ARRY$$  ,R2,,Y,'W,$00,
0246 010267             MOV     R2,$00
     000000'
0252 006367             ASL     $00
     000000'
0256 062767             ADD     #Y-2,$00
     000010'
     000000'
0264 017767             MOV     @$00,M
     000000'
     177544
0272                    PRINT$                  ;*   PRINT %N,TAB,%M
0272 012746             MOV     #0,-(SP)
     000000
0276 004767             JSR     PC,PRINT
     000000G
0302                    PRDEC$  N,'W
0302 016746             MOV     N,-(SP)
     177526
0306 004767             JSR     PC,BWDEC$
     000000G
0312                    PRASC$  #TAB,'W
0312 012746             MOV     #TAB,-(SP)
     000011
0316 004767             JSR     PC,BWASC$
     000000G
0322                    PRDEC$  M,'W
0322 016746             MOV     M,-(SP)
     177510
0326 004767             JSR     PC,BWDEC$
     000000G
0332                    CALL$$  GGT,<N,M>
0332 010546             MOV     R5,-(SP)
0334 012705             MOV     #64$,R5
     000346'
0340 004767             JSR     PC,GGT
     177474
0344 000403             BR      65$
0346 000002 64$:        .WORD   SYM$$$
0350 000034'            .WORD   N
0352 000036'            .WORD   M
0354 012605 65$:        MOV     (SP)+,R5
0356 077257             SOB     R2,$00007       ;* END
0360                    .EXIT
0360 104350                             EMT     ^0350
     000216'            .END START
```

SIMPL11 FEHLER CODES
====================

1 <INPUT> ILLEGAL.

2 <COMMAND> IN ZUSAMMENGESETZTEN ANWEISUNGEN
ILLEGAL: <EXPRESSION>; <COMMAND>
-^-

3 MEHRFACHE VERWENDUNG EINES LABEL-NAMENS.

4 MACRO DEFINITION. ILLEGALER MACRO IDENTIFIKATOR:
MACRO 100
-^-

6 <INPUT> IN EINEM BLOCK ILLEGAL:
BEGIN <INPUT> END
-^-

7 FEHLENDES 'END' IN EINEM BLOCK.

15 SUBROUTINE AUFRUF. ILLEGALER OPERAND:
JSR <OPERAND>
-^-

16 SUBROUTINE AUFRUF. ILLEGALES REGISTER:
JSR <OPERAND> (<REGISTER>)
-^-

17 MACRO AUFRUF. ILLEGALER IDENTIFIKATOR:
<MACRO IDENTIFIER> (....)
-^-

18 MACRO AUFRUF. ILLEGALER PARAMETER AUSDRUCK:
<MACRO IDENTIFER> (<EXPR>, ...)
-^-

19 MACRO AUFRUF. SCHLIESSENDE KLAMMER FEHLT:
<MACRO IDENTIFIER> (....)
-^-

20 IF - ANWEISUNG. LOGISCHER AUSDRUCK ILLEGAL:
IF <LOGIC EXPR> THEN
-^-

21 IF - ANWEISUNG. 'THEN' FEHLT.

22 IF - ANWEISUNG. ILLEGALER BLOCK NACH THEN ODER ELSE:
IF <LOGIC EXPR> THEN <BLOCK> ELSE <BLOCK>
-^- -^-

24 LOGISCHER AUSDRUCK. ILLEGALER LOGISCHER OPERATOR.

26 LOGISCHER AUSDRUCK. ILLEGALE VERBINDUNG VON
FLOAT-REGISTER UND IDENTIFIKATOR:
IF AC0 EQ FLT THEN ...
-^-

27 LOGISCHER AUSDRUCK. ILLEGALER OPERATOR ZWISCHEN
FLOAT REGISTERN:
IF AC0 LOWER AC1 THEN ...
-^-

30 FOR - SCHLEIFE. ILLEGALER AUSDRUCK NACH 'FOR':
FOR <EXPR>=><REGISTER> DO ...
-^-

31 FOR - SCHLEIFE. ILLEGALES REGISTER:
FOR ... <REGISTER> DO ...
-^-

32 FOR - SCHLEIFE. FEHLENDES 'DO'

40 WHILE - SCHLEIFE. ILLEGALER LOGISCHER AUSDRUCK:
WHILE <LOGIC EXPR> DO
-^-

41 WHILE - SCHLEIFE. FEHLENDES 'DO'

46 REPEAT - SCHLEIFE. FEHLENDES 'UNTIL'

47 REPEAT - SCHLEIFE. ILLEGALER LOGISCHER AUSDRUCK:
REPEAT ... UNTIL <LOGIC EXPR>
-^-

50 GO - ANWEISUNG. ILLEGALER OPERAND:
GO <OPERAND>
-^-

52 RETURN - ANWEISUNG. ILLEGALES ATTRIBUT:
RETURN <TRUE/FALSE>
-^-

53 RETURN - ANWEISUNG. ILLEGALES REGISTER:
RETURN (<REGISTER>)
-^-

54 ILLEGALE RADIX KONTROLLE IN EINER ZAHL:
^<RADIX> <NUMBER>
-^-

55 ILLEGALE ZAHL.

56 ILLEGALE DEZIMALZAHL (RADIX STEHT AUF OKTAL).

57 SYMBOL TABLE OVERFLOW.

58 <MACRO EXPRESSION> ILLEGAL ABGESCHLOSSEN.

100 LIT - DEKLARATION. ILLEGALE UMDEKLARTATION
EINER SCHON DEFINIERTEN VARIABLE

101 LIT - DEKLARATION. FEHLENDE INITIALISIERUNG.

102 ILLEGALES SYMBOL.

103 MEHRFACHE DEKLARATION EINES SYMBOLS.

104 ILLEGALER TYP FUER RAD50 INITIALISIERUNG.

105 ILLEGALE BYTE- ODER WORD KONSTANTE.

106 ARRAY DEKLARATION. ILLEGALER TYP.

107 ARRAY DEKLARATION. ILLEGALE INITIALISIERUNG.

108 ARRAY DEKLARATION. ILLEGALE LAENGEN-ANGABE.

109 ARRAY DEKLARATION. FEHLENDE KLAMMER:
```
ARRAY <TYPE><ARRAY NAME> [ <NUMBER> ]
                                    _^_
```

110 FEHLENDES SEMIKOLON ALS ABSCHLUSS EINER DEKLARATION.

111 ILLEGALE STRING DEKLARATION. FEHLENDE APOSTROPHE.
```
ARRAY STRING STR = ' ... '
                   _^_
```

112 ARRAY INITIALISIERUNG. FEHLENDE KLAMMER IN EINER
REPETITION.
```
3*[....]
 _^_
```

113 STACK DEKLARATION. NAME SCHON VORHANDEN.

114 STACK DEKLARATION. FEHLENDE LAENGENANGABE:
```
STACK ST[...];
        _^_
```


200 ILLEGALES ZUWEISUNGSZEICHEN:
```
A => B
 _^_
```

202 MULTIPLIKATION. ZWEITER OPERND FEHLT.

203 <OPERAND> ILLEGAL.

204 MULTIPLE ZUWEISUNG. FEHLENDE KLAMMER:
```
A=>[X,Y,...]
    _^_
```

205 ZUWEISUNG. ILLEGALER OPERAND:
```
... => <OPERAND>
       _^_
```

207 FEHLENDER OPERAND BEI EINEM DYADISCHEN OPERATOR.

208 ILLEGALER TYP FUER MONADISCHEN OPERATOR.

209 ILLEGALER TYP. ES IST NUR 'WORD' ERLAUBT.

210 ILLEGALER TYP. ES IST NUR 'WORD' ODER 'BYTE' ERLAUBT.

211 OPERAND MUSS REGISTER SEIN:
```
AB * 10
_^_
```

212 ILLEGALER TYP 'REAL'.

213 GEMISCHTE TYPEN IN EINEM AUSDRUCK.

214 REPETITONS AUSDRUCK. ILLEGALER TEILAUSDRUCK:
```
[<EXPR1> , <EXPR2> , ...]
  _^_
```

215 REPETITONS-AUSDRUCK. FEHLENDE KLAMMER:
```
[<EXPR1> , <EXPR2> , ...]
                     _^_
```

216 REPETITONS-AUSDRUCK. ILLEGALER FOLGEAUSDRUCK:
```
[<EXPR1> , <EXPR2> , ...] <REST OF EXPRESSION>
                          _^_
```

```
250     INDIREKTE ADRESSIERUNG. ILLEGALER OPERAND.

251     INDIREKTE ADRESSIERUNG. 'REAL' ILLEGAL.

252     INDIREKTE ADRESSIERUNG. 'BYTE' ILLEGAL.

253     INDIREKTE ADRESSIERUNG. FLOAT REGISTER ILLEGAL.

254     ABSOLUTE ADRESSIERUNG. UNDEFINIERTES SYMBOL.

255     ABSOLUTE ADRESSIERUNG. ILLEGALER NAME.

256     OEFFNENDE KLAMMER '(' FEHLT.

257     SIZE - OPERAND. UNDEFIERTES SYMBOL.

258     SIZE - OPERAND. TYP UNGLEICH ARRAY.

259     SCHLIESSENDE KLAMMER ')' FEHLT.

260     ILLEGALES REGISTER IN
        AUTO INCREMENT/DECREMENT ADRESSIERUNG:
                ( <REGISTER> ) +
                  -^-

262     REF - OPERAND. ILLEGALER MACRO AUSDRUCK:
                REF ( <MACRO EXPR> )
                      -^-

263     REF - OPERAND. ILLEGALES SYMBOL.
                REF ( <SYMBOL> )
                      -^-

264     UNDEFINIERTES SYMBOL.

266     ILLEGALER OPERAND. MUSS EINE ZAHL SEIN.

267     ARRAY REFERENZ. FEHLENDER INDEX.

268     ARRAY REFERENZ. ILLEGALER INDEX-AUSDRUCK.

269     ILLEGALER INDEX-TYP (REAL).

271     SCHLIESSENDE KLAMMER ']' FEHLT.

272     ILLEGALE REAL-KONSTANTE.

280     ILLEGALE STACK REFERENZ:
                PUSH [ <REGISTER> ]
                       -^-

281     SCHLIESSENDE KLAMMER IN STACK REFERENZ FEHLT:
                PUSH [ ... ]
                       -^-
```



```
290     DUMP. ILLEGALER AUSDRUCK:
                DUMP <EXPR> , <EXPR>
                     -^-      -^-

291     DUMP:   FEHLENDES KOMMA.

292     DUMPS. ILLEGALER AUSDRUCK.

294     PRINT. ILLEGALER RAD50 AUSDRUCK.

295     PRINT. FEHLENDER APOSTROPH IN STRING KONSTANTE.

296     TOP OPERAND. ILLEGALE ZAHL:
                TOP [ <REGISTER>, <NUMBER> ]
                                  -^-

297     TOP OPERAND. ZAHL MUSS GROESSER ODER
        GLEICH EINS SEIN:
                TOP [ <REGISTER>, <NUMBER> ]
                                  -^-

300     POINTER OPERAND. ILLEGALES POINTER ZEICHEN '<>'

301     POINTER OPERAND. ILLEGALER TYP:
                <> <TYPE> :
                   -^-

302     POINTER OPERAND. FEHLENDER DOPPELPUNKT:
                <> <TYPE> :
                          -^-

303     POINTER OPERAND. ILLEGALES SYMBOL ODER TYP
        UNGLEICH 'WORD'.
                <> <TYPE> : <SYMBOL> [...]
                            -^-

310     ILLEGALE OPERATION MIT FLOAT REGISTER
                ACO ROL
                  -^-

311     OPERAND MUSS EIN FLOAT REGISTER SEIN.
```

F. Syntax

F.1 SIMPL11-SYNTAX

```
<SIMPL11-PROGRAM> := <SIMPL11-INPUT> [<SIMPL11-PROGRAM>] / <S-END>

<S-END>           := .END [<EXECUTION START>]

<EXECUTION START> := <LABEL-NAME>

<SIMPL11-INPUT> := <INPUT><END-OF-LINE> / % <MACRO-ASS-PROGRAM> % /
                   . <MACRO-11-LINE> / > . <SYSTEM-MACRO-CALL>

<END-OF-LINE>   := [<COMMENT>] CR LF

<COMMENT>       := // <ANY-CHARACTER \ END-OF-LINE-CHARACTER>

<INPUT>         := <LABEL-TRAIN> / [<LABEL-TRAIN>] <COMMAND> /
                   <DECLARATION>

<LABEL-TRAIN>   := <LABEL> / <LABEL><LABEL-TRAIN>

<LABEL>         := <NAME> :

<COMMAND>       := <ACTION> / <CONTROL> / <SPECIAL-FUNCTIONS>

<ACTION>        := <STATEMENT> / <BLOCK>

<BLOCK>         := <BEGIN><SIMPL11-INPUT><END>

<STATEMENT>     := <EXPRESSION> / <EXPRESSION> ; <COMMAND>

<BEGIN>         := BEGIN [<END-OF-LINE>]

<END>           := [<LABEL-TRAIN>] END


<MACRO-ASS-PROGRAM> := <MAC-11-INPUT> [<MACRO-ASS-PROGRAM>]

<MAC-11-INPUT>  := <ASSEMBLER-LINE><END-OF-LINE>

<MACRO-11-LINE> := <MAC-11-INPUT> / <SPECIAL-DIRECTIVES><END-OF-LINE>

<SPECIAL-DIRECTIVES> := RADIX [<NUMBER-REF>] /
                        LIST / NLIST

<SPECIAL-FUNCTIONS> := TRAP <NUMBER-REF> /
                       SETF / SETI / SETD / SETL /
                       LDFPS / STFPS / STST /
                       LDEXP ( <WORD-IDENTIFIER> , <FLOAT-AC> )
                       STEXP ( <FLOAT-AC> , <WORD-IDENTIFIER> )
```



```
<DECLARATION>   := <MACRO-DECLARATION> /
                   <SIMPLE-TYPE-DECLARATION> /
                   <ARRAY-DECLARATION> / <STACK-DECLARATION> /
                   <LIMIT-DECLARATION>

<MACRO-DECLARATION> := <MACRO-HEADER> [<SIMPL11-INPUT>] <MACRO-END>

<MACRO-HEADER>  := <MACRO-DEFINITION><END-OF-LINE>

<MACRO-DEFINITON> := MACRO<MAC-IDENTIFIER> [(<MACRO-PARM-LIST>)]

<MACRO-PARM-LIST> := <PARM-ITEM> / <PARM-ITEM> ; <MACRO-PARM-LIST>

<PARM-ITEM>     := <MACRO-TYPE> : <NAME-TRAIN>

<NAME-TRAIN>    := <NAME> / <NAME> , <NAME-TRAIN>

<MAC-IDENTIFIER>:= <IDENTIFIER>

<MACRO-TYPE>    := <SIMPLE-TYPE> / LIT

<MACRO-END>     := .ENDM

<SIMPLE-TYPE-DECLARATION> := <LITERAL-DECLARE> /
                             <BYTE-WORD-REAL-DECLARE>

<LITERAL-DECLARE> := LIT <LITERAL-TRAIN> ;

<LITERAL-TRAIN> := <LITERAL-ITEM> /
                   <LITERAL-ITEM> , [<END-OF-LINE>] <LITERAL-TRAIN>

<LITERAL-ITEM>  := <IDENTIFIER> = <LITERAL-INITIAL>

<LITERAL-INITIAL> := <BYTE-CONST> / <WORD-CONST> /
                     <MACRO-ASS-EXPR>

<BYTE-WORD-REAL DECLARE> := <SIMPLE-TYPE><BWR-TRAIN> ;

<BWR-TRAIN>     := <BWR-ITEM> /
                   <BWR-ITEM> , [<END-OF-LINE>] <BWR-TRAIN>

<BWR-ITEM>      := <IDENTIFIER> [= <BWR-INITIAL>]

<BWR-INITIAL>   := <BYTE-WORD-INITIAL> / <REAL-INITIAL>

<BYTE-WORD-INITIAL> := <BYTE-CONST> / <WORD-CONST> /
                       <MACRO-ASS-EXPR>

<REAL-INITIAL>  :=  <REAL-CONST>

<BYTE-CONST>    := ' <CHAR>
<WORD-CONST>    := " <CHAR><CHAR> / $ [<CHAR>]_1^2

<REAL-CONST>    := <FLOAT-NUMBER> / $ [<CHAR>]_1^6

<SIMPLE-TYPE>   := BYTE / WORD / REAL
```

```
<ARRAY-DECLARATION> := ARRAY <ARRAY-TYPE><IDENTIFIER><ARRAY-INITIAL> ;

<ARRAY-TYPE>     := <SIMPLE-TYPE> / STRING

<ARRAY-INITIAL> := <EXPLICIT-INITIAL> / = <IMPLICIT-INITIAL>

<EXPLICIT-INITIAL> := [ <LENGTH> ]

<LENGTH>         := <NUMBER-REF> / <LITERAL-IDENTIFIER>

<LITERAL-IDENTIFIER> := <IDENTIFIER>

<IMPLICIT-INITIAL> := <ARRAY-ITEM> /
                      <ARRAY-ITEM> , [<END-OF-LINE>]  <IMPLICIT-INITIAL>

<ARRAY-ITEM>     := [<REPETITION>] <ARRAY-INITIAL>

<REPETITION>     := <REP-FACTOR> * <REP-ITEM>

<REP-FACTOR>     := <LENGTH>

<REP-ITEM>       := [ <IMPLICIT-INITIAL> ] / <ARRAY-INITIAL>

<ARRAY-INITIAL> := <BWR-INITIAL> / <BYTE-STRING>

<BYTE-STRING>    := ' <ANY-CHARACTER \ END-OF-LINE-CHARACTER> '


<STACK-DECLARATION> := STACK <IDENTIFER><EXPLICIT-INITIAL> ;

<LIMIT-DECLARATION> := LIMIT <IDENTIFIER> ;
```



```
<CONTROL>        := <IF> / <FOR> / <WHILE> / <REPEAT>

<IF>             := IF <CONDITION><THEN><ACTION> [<ELSE><ACTION>]

<THEN>           := THEN / :
<ELSE>           := ELSE / :

<FOR>            := FOR [<EXPR>=>] <REGISTER> DO <ACTION>

<WHILE>          := WHILE <CONDITION> DO <ACTION>

<REPEAT>         := REPEAT <ACTION> UNTIL <CONDITION>

<CONDITION>      := <LOGIC-EXPR>


<LOGIC-EXPR>     := <LOGIC-STATEMENT> /
                    <LOGIC-STATEMENT><COMPOUND-OPERATOR><LOGIC-EXPR>

<COMPOUND-OPERATOR> := <AND-COMPOUND> / <OR-COMPOUND>

<AND-COMPOUND>   := &

<OR-COMPOUND>    := #

<LOGIC-STATEMENT> := <LOGIC-ITEM> / <BOOL-OP> / <LOGIC-ITEM><BOOL-OP> /
                     <BYTE-WORD-EXPR><BOOL-DYA-OP><OPERAND> /
                     <REAL-LOGIC-EXPR>

<LOGIC-ITEM>     := <BYTE-WORD-EXPR> / <PROC-CALL>

<BYTE-WORD-EXPR>:= <EXPR>

<REAL-LOGIC-EXPR>:= <REAL-IDENTIFIER><BOOL-DYA-OP><REAL-IDENTIFIER> /
                    <FLOAT-AC><BOOL-REAL-OP><RI-RAC>

<RI-RAC>         := <REAL-IDENTIFIER> / <FLOAT-AC>

<BOOL-REAL-OP>   := EQ / NE / GT / GE / LT / LE

<BOOL-DYA-OP>    := <BOOL-REAL-OP> /
                    LOWER / LOSAME / HIGHER / HISAME

<BOOL-OP>        := <BOOL-DYA-OP> /
                    TRUE / FALSE / CARRY / OFLOW
```

```
<EXPRESSION>      := <EXPR> / <PROC-CALL> / <MACRO-CALL> /
                     <GO-TO> / <RETURN> / <IO-EXPR>

<EXPR>            := <REPEAT-EXPR> / <SIMPLE-EXPR>

<REPEAT-EXPR>     := [ <EXPR-TRAIN> ] <REST-OF-EXPR> /
                     <EXPR>  => <REPEAT-EXPR>

<EXPR-TRAIN>      := <EXPR> / <EXPR> , <EXPR-TRAIN>

<SIMPLE-EXPR>     := <OPERAND> [<REST-OF-EXPR>]

<REST-OF-EXPR>    := <MONADIC-OPERATOR> [<REST-OF-EXPR>] /
                     <DYADIC-OPERATOR><OPERAND> [<REST-OF-EXPR>]

<MONADIC-OPERATOR> := <MON-BYTE-OP> / <MON-WORD-OP> /
                      <MON-REAL-OP>

<MON-BYTE-OP>     := + / - / COM / NEG / ROR / ROL / ASR / ASL /
                     ADC / SBC

<MON-WORD-OP>     := <MON-BYTE-OP> / SWAB / SXT

<MON-REAL-OP>     := NEG / ABSF

<DYADIC-OPERATOR> := <DYA-BYTE-OP> / <DYA-WORD-OP> /
                     <DYA-REAL-OP>

<DYA-BYTE-OP>     := BIT / BIC / BIS

<DYA-WORD-OP>     := <DYA-BYTE-OP> / + / - / * / <DIV> / ASH / ASHC / XOR

<DYA-REAL-OP>     := + / - / * /  <DIV> / MODF

<DIV>             := /
```



```
<OPERAND>         := <BYTE-WORD-OPERAND> / <REAL-OPERAND> /
                     <STACK-OPERAND> / <ARRAY-REFERENCE> /
                     <ARRAY-POINTER> / <ARRAY-SIZE-LITERAL> /
                     <INDIRECT-REFERENCE>

<BYTE-WORD-OPERAND> := <BYTE-OPERAND> / <WORD OPERAND> /
                       <NUMBER-REF>

<STACK-OPERAND>   := <CHANGE-STACK> / <TOP>

<CHANGE-STACK>    := <PUSH-POP>  [ <REGISTER> ]

<PUSH-POP>        := PUSH / POP

<TOP>             := TOP [<TOP-REF>]

<TOP-REF>         := [ <TOP-ARG> [<TYPE-CHANGE>] ]

<TOP-ARG>         := <REGISTER> [,<NUMBER-REF>] /
                     [<REGISTER> ,] <NUMBER-REF>

<ARRAY-REFERENCE> := <ARRAY-IDENTIFIER><ARRAY-INDEX>

<ARRAY-INDEX>     := [ <BYTE-WORD-EXPR> ]

<ARRAY-IDENTIFIER> := <IDENTIFIER>

<ARRAY-POINTER>   := <> <SIMPLE-TYPE> : <WORD-IDENTIFIER><ARRAY-INDEX>

<WORD-IDENTIFIER> := <IDENTIFIER>

<ARRAY-SIZE-LITERAL> := SIZE ( <ARRAY-IDENTIFIER> )

<INDIRECT-REFERENCE> := @ <INDIRECT OPERAND>

<INDIRECT-OPERAND> := <WORD-LITERAL-IDENTIFIER> / <NUMBER-REF> /
                      <WORD-ARRAY-REFERENCE> / <WORD-ARRAY-POINTER> /
                      <STACK-OPERAND> / <REGISTER-OPERAND>

<WORD-ARRAY-REFERENCE> := <ARRAY-REFERENCE>

<WORD-ARRAY-POINTER> := <ARRAY-POINTER>
```

```
<BYTE-OPERAND>   := <BYTE-LITERAL-IDENTIFIER> /
                    <BYTE-CONST> /
                    <OPERAND><CHANGE-TO-BYTE>

<BYTE-LITERAL-IDENTIFIER> := <IDENTIFIER>

<WORD-OPERAND>   := <WORD-LITERAL-IDENTIFIER> /
                    <WORD-CONST> / <REGISTER-OPERAND> /
                    <ABSOLUT-REFERENCE> /
                    <REF-REFERENCE>

<WORD-LITERAL-IDENTIFIER> := <IDENTIFIER>

<ABSOLUT-REFERENCE> := ABS ( <IDENTIFIER> )

<REF-REFERENCE>  := REF ( <REF-OPERAND> )

<REF-OPERAND>    := <IDENTIFIER> / <MACRO-ASS-EXPR> /
                    <ARRAY-REFERENCE> / <ARRAY-POINTER> /

<REGISTER-OPERAND> := <REGISTER> / <AIX-REGISTER>

<REGISTER>       := R0 / R1 / R2 / R3 / R4 / R5 / SP

<AIX-REGISTER>   := <AI-REGISTER> / <INDEX-REGISTER>

<AI-REGISTER>    := ( <REGISTER> ) <AI>

<AI>             := + / -

<INDEX-REGISTER>:= <REGISTER> ( <MACRO-ASS-EXPR> )

<REAL-OPERAND>   := <REAL-IDENTIFIER> / <FLOAT-AC> /
                    <OPERAND><CHANGE-TO-REAL>

<REAL-IDENTIFIER> := <IDENTIFIER>

<FLOAT-AC>       := AC0 / AC1 / AC2 / AC3 / AC4 / AC5


<TYPE-CHANGE>    := <CHANGE-TO-BYTE> / CHANGE-TO-WORD> /
                    <CHANGE-TO-REAL>

<CHANGE-TO-BYTE> := '

<CHANGE-TO-WORD> := "

<CHANGE-TO-REAL> := ""
```



```
<NUMBER-REF>     := <NUMBER> / <DEC-NUMBER> /
                    <OCT-NUMBER>

<DEC-NUMBER>     := <NUMBER> . / ^D <NUMBER>

<OCT-NUMBER>     := ^O <NUMBER>

<NUMBER>         := <DIGIT> / <DIGIT><NUMBER>

<DIGIT>          := 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

<IDENTIFIER>     := <NAME>

<NAME>           := [<CHAR>]_1^6

<CHAR>           := A / B / ... / Z


<PROC-CALL>      := <JSR><WORD-IDENTIFIER> [<CALL-REGISTER>] /
                    CALL <WORD-IDENTIFIER> [<PARM-LIST>]

<JSR>            := JSR / !

<CALL-REGISTER>  := ( <REGISTER> )

<PARM-LIST>      := ( <PARM-TRAIN> )

<PARM-TRAIN>     := <PARM> / <PARM> , <PARM-TRAIN>

<PARM>           := <EXPR>


<MACRO-CALL>     := <SYSTEM-MACRO-CALL> /
                    <USER-MACRO-CALL>

<SYSTEM-MACRO-CALL> := . <USER-MACRO-CALL>

<USER-MACRO-CALL> := <MAC-IDENTIFIER> [<PARM-LIST>]

<GO-TO>          := GO <WORD-IDENTIFIER>

<RETURN>         := RETURN [<RETURN-ARG>]

<RETURN-ARG>     := <TRUE-FALSE> / <RETURN-REGISTER> /
                    <TRUE-FALSE><RETURN-REGISTER>

<TRUE-FALSE>     := TRUE / FALSE

<RETURN-REGISTER> := <CALL-REGISTER>
```

```
<IO-EXPR>          := <READ-PRINT> / <DUMPS> / <DUMP> /
                      <DSTACK> / <DREG>

<READ-PRINT>       := <RP-COMMAND> [<FORMAT-CONTROL>] <RP-TRAIN>

<RP-COMMAND>       := READ / PRINT

<RP-TRAIN>         := <RP-ITEM> / <RP-ITEM> , <RP-TRAIN>

<RP-ITEM>          := <BYTE-STRING> /
                      [<RP-CONTROL>] <EXPR> / <EMPTY>

<FORMAT-CONTROL>   := <SLASH>  / -

<SLASH>            := /

<RP-CONTROL>       := ! / % / %% / # / <RAD>

<RAD>              := $

<DUMPS>            := DUMPS <DP-TRAIN>

<DP-TRAIN>         := <DP-ITEM> / <DP-ITEM> , <DP-TRAIN>

<DP-ITEM>          := [<RAD>] <EXPR>

<DUMP>             := DUMP [<RAD>] <EXPR> , <EXPR>

<DSTACK>           := DSTACK

<DREG>             := DREG
```


F.2 Macro-11 Assembler Ausdrücke

```
<MACRO-ASS-EXPR> := <U-TERM> [<BINARY-OPERATOR><U-TERM>]

<U-TERM>         := [<UNARY-OPERATOR>] <TERM>

<TERM>           := <SYMBOL> / <NUMBER> /
                    <  <MACRO-ASS-EXPR>  >

<SYMBOL>         := <NAME>

<UNARY-OPERATOR>:= + / - / ^<SPEC-OPERATOR>

<SPEC-OPERATOR> := B / C / D / F / O

<BINARY-OPERATOR> := + / - / * / <DIV> / & / !
```

F.3 Macro-11 Reelle Zahlen

```
<FLOAT-NUMBER>   := [<SIGN>] <UNSIGNED-FLOAT-NUMBER>

<UNSIGNED-FLOAT-NUMBER> := <INTEGER> [<FLOAT>] /
                           [<INTEGER>] <FLOAT> /

<INTEGER>        := <DIGIT> / <DIGIT><INTEGER>

<FLOAT>          := <FRACTION> [<EXPONENT>] /
                    [<FRACTION>] <EXPONENT>

<FRACTION>       := . <INTEGER>

<EXPONENT>       := E [<SIGN>] <INTEGER>

<SIGN>           := + / -
```

DANKSAGUNG

Diese Arbeit entstand im Rahmen einer Zusammenarbeit des Deutschen Elektronen-Synchrotron DESY mit dem Universitätskrankenhaus Hamburg-Eppendorf (UKE).

Mein Dank gilt Herrn Professor Dr. H. Schopper und Herrn Professor Dr. G. Weber für die tatkräftige Unterstützung des DESY-UKE-Projektes.

Herrn Dr. K. H. Höhne danke ich für die vielen fruchtbaren Diskussionen und die aus diesen Diskussionen erwachsenen Anregungen.

Meinen Kollegen M. Böhm, W. Ebenritter und Dr. B. Sonne danke ich für die kritische Durchsicht des Manuskripts.

Mein besonderer Dank gilt Herrn K. Dahlmann, auf dessen Vorschlag die Namensgebung von SIMPL11 beruht.

Literaturverzeichnis

1. Wirth, N.:
PL360, A Programming Language for the 360 Computers.
Journal of ACM, 15, 1968, 37-74

2. Russell, R.D., Streater, T.C. (ed.):
PL-11: A Programming Language for the DEC PDP-11 Computer.
CERN, Data Handling Division, 74-24 (Dec. 1974)

3. Plauger, P.J.:
A Little Implementation Language.
SIGPLAN Notices, 11, 4, 135-137 (April 1976)

4. Russell, R.D.:
Experience in the Design, Implementation and Use of PL-11, a Programming Language for the PDP-11.
SIGPLAN Notices, 11, 4, 27-34 (April 1976)

5. Santo, H.:
Vergleich niederer Programmiersprachen.
GI-Bericht 4, 2. Fachtagung über Programmiersprachen, Saarbrücken 1972, 362-373

6. Höhne, K. H., Nicolae, G., Pfeiffer, G., Dix, W. R., Ebenritter, W., Novak, D., Böhm, M., Sonne, B., Bücheler, E.:
An Interactive System for Clinical Application of Angiodensitometry.
GI-Fachtagung, Digitale Bildverarbeitung, München 1977.

7. Pfeiffer, G., Höhne, K. H.:
A Dialog Language for Interactive Processing of Scintigraphic Data.
Proc. of the 4th International Conference on Information Processing in Scintigraphy, Orsay, 1975, 221-232.