AMIGA_E_REFERENZ

COLLABORATORS

	TITLE : AMIGA_E_REFERENZ	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 16, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 AMIGA_E_REFERENZ **1**

1.1 Inhalt . . . 1
1.2 Bemerkung . . . 2
1.3 Index . . . 2
1.4 Formatierung . . . 6
1.5 Formatierung . . . 6
1.6 Formatierung . . . 6
1.7 Formatierung . . . 7
1.8 Direkte Werte . . . 7
1.9 Direkte Werte . . . 8
1.10 Direkte Werte . . . 8
1.11 Direkte Werte . . . 8
1.12 Direkte Werte . . . 8
1.13 Direkte Werte . . . 9
1.14 Direkte Werte . . . 9
1.15 Direkte Werte . . . 10
1.16 Ausdrücke . . . 10
1.17 Ausdrücke . . . 11
1.18 Ausdrücke . . . 11
1.19 Ausdrücke . . . 11
1.20 Ausdrücke . . . 12
1.21 Operatoren . . . 12
1.22 Operatoren . . . 13
1.23 Operatoren . . . 13
1.24 Operatoren . . . 13
1.25 Operatoren . . . 13
1.26 Operatoren . . . 14
1.27 Operatoren . . . 15
1.28 Operatoren . . . 15
1.29 Operatoren . . . 15

1.30 Operatoren . . . 16
1.31 Operatoreyn . . . 16
1.32 Statements . . . 16
1.33 Statements . . . 17
1.34 Statements . . . 18
1.35 Statements . . . 18
1.36 Statements . . . 18
1.37 Statements . . . 19
1.38 Statements . . . 19
1.39 Statements . . . 19
1.40 Statements . . . 20
1.41 Statements . . . 20
1.42 Statements . . . 20
1.43 Statements . . . 21
1.44 Statements . . . 21
1.45 Function Definitions and Declarations . . . 21
1.46 Funktions Deklaration und Definition . . . 22
1.47 Funktions Deklaration und Definition . . . 23
1.48 Funktions Deklaration und Definition . . . 24
1.49 Funktions Deklaration und Definition . . . 25
1.50 Funktions Deklaration und Definition . . . 25
1.51 Deklaration von Konstanten . . . 26
1.52 Deklaration von Konstanten . . . 27
1.53 Deklaration von Konstanten . . . 27
1.54 Deklaration von Konstanten . . . 27
1.55 Deklaration von Konstanten . . . 28
1.56 Typen . . . 28
1.57 Typen . . . 28
1.58 Typen . . . 29
1.59 Typen . . . 30
1.60 Typen . . . 30
1.61 Typen . . . 30
1.62 Typen . . . 31
1.63 Typen . . . 32
1.64 Eingebaute Funktionen . . . 32
1.65 Eingebaute Funktionen . . . 33
1.66 Eingebaute Funktionen . . . 34
1.67 Eingebaute Funktionen . . . 36
1.68 Eingebaute Funktionen . . . 38

1.69 Eingebaute Funktionen . . . 40
1.70 Eingebaute Funktionen . . . 41
1.71 Eingebaute Funktionen . . . 42
1.72 Eingebaute Funktionen . . . 44
1.73 Library Funktionen und Module . . . 45
1.74 Library Funktionen und Module . . . 45
1.75 Library Funktionen und Module . . . 45
1.76 Ausgewertete Ausdrücke . . . 46
1.77 Ausgewertete Ausdrücke . . . 47
1.78 Ausgewertete Ausdrücke . . . 47
1.79 Ausgewertete Ausdrücke . . . 48
1.80 Fließkommaunterstützung . . . 49
1.81 Fließkommaunterstützung . . . 49
1.82 Fließkommaunterstützung . . . 50
1.83 Exception Behandlung . . . 50
1.84 Exception Behandlung . . . 50
1.85 Exception Behandlung . . . 51
1.86 Exception Behandlung . . . 52
1.87 Exception Behandlung . . . 55
1.88 Objektorientierte Programmierung . . . 57
1.89 Der Inline-Assembler . . . 57
1.90 Der Inline-Assembler . . . 57
1.91 Der Inline-Assembler . . . 58
1.92 Der Inline-Assembler . . . 58
1.93 Der Inline-Assembler . . . 59
1.94 Dinge über den Compiler . . . 59
1.95 Dinge über den Compiler . . . 59
1.96 Dinge über den Compiler . . . 60
1.97 Dinge über den Compiler . . . 61
1.98 Dinge über den Compiler . . . 61
1.99 Dinge über den Compiler . . . 62
1.100 Dinge über den Compiler . . . 68
1.101 Dinge über den Compiler . . . 68

Chapter 1

AMIGA_E_REFERENZ

1.1 Inhalt

Amiga E v2.1b
Compiler für die E Sprache
Von Wouter van Oortmerssen

Supertolle Preise zu gewinnen
Sprachdokumentation

1. Formatierung

2. Direkte Werte

3. Ausdrücke

4. Operatoren

5. Statements

6. Funktions Definition und Deklaration

7. Deklaration von Konstanten

8. Typen

9. Eingebaute Funktionen

10. Library Funktionen und Module

11. Ausgewertete Ausdrücke

12. Fließkommaunterstützung

13. Exception Behandlung

14. Objekt-Orientiertes Programmieren

15. Der Inline Assembler

16. Eingebaute Ausgaben

1.2 Bemerkung

Hääää, reingelegt... ROFL

Für den Fall, daß irgendjemand Fragen zur Übersetzung einzelner Kapitel hat, wende er sich bitte an:

Kapitel 1,4-5,7-9,11-12,15	:	Rolf Breuer Marktstr. 13 45891 Gelsenkirchen
Kapitel 2,3,14, Amiga-Guide Fassung	:	Daniel van Gerpen Alter Postweg 3 26759 Hinte
Kapitel 10	:	Gregor Goldbach Grüner Weg 10 21423 Pattensen
Kapitel 16	:	Christoph Lange Altdorferstr. 19 63739 Aschaffenburg
Kapitel 6, 13	:	Jörg Wach Waitzstr. 75 24105 Kiel

Wenn jemand eine, seiner Meinung nach besser, Übersetzung eines oder mehrerer Kapitel hat, dann kann er diese an Daniel van Gerpen (s.o.) schicken, und wird dann beim nächsten Mal hier erwähnt.

1.3 Index

Amiga E v2.1b

Compiler für die E Sprache

Von Wouter van Oortmerssen

Index

1. Formatierung

A. Tabulatoren(tabs), Zeilenvorschübe(lf) usw.

B. Kommentare

C. Identifier und Typen

2. Direkte Werte

A. Dezimalzahlen (1)

B. Hexadezimalzahlen ($1)

C. Binärzahlen (%1)

D. Fließkommazahlen (1.0)

E. Zeichen ('a')

F. Zeichenketten ('bla')

G. Listen ([1,2,3]) und symbolische Listen

3. Ausdrücke

A. Format

B. Abarbeitung und Anordnung

C. Arten von Ausdrücken

D. Funktionsaufrufe

4. Operatoren

A. Mathematische (+ - * /)

B. Vergleiche (= <> > < >= <=)

C. Logische und Bitweise (AND OR)

D. Unary (SIZEOF ` ^ {} ++ -- -)

E. Dreifach (IF THEN ELSE)

F. Strukturen (.)

G. Felder ([])

H. Fließkommaoperator (|)

I. Zuweisungsoperator (:=)

J. Reihenfolge (BUT)

5. Statements

A. Format (;)

B. Sprungmarken und Sprunganweisungen (JUMP)

C. Zuweisungen (:=)

D. Assembler Mnemonics

E. Bedingte Anweisungen (IF)

F. For-Anweisung (FOR)

G. While-Anweisung (WHILE)

H. Repeat-Anweisung (REPEAT)

I. Loop-Anweisung (LOOP)

J. Auswahl-Anweisungen (SELECT)

K. Zuwachs-Anweisungen (INC/DEC)

L. Void-Ausdrücke (VOID)

6. Funktions Deklaration und Definition

A. Prozedur Definition und Argumente (PROC)

B. Lokale und globale Definitionen (DEF)

C. Endproc/return

D. Die 'main' Funktion

E. E-eigene Systemvariablen

7. Deklaration von Konstanten

A. Konstanten (CONST)

B. Aufzählungen (ENUM)

C. Sets (SET)

D. E-eigene Konstanten

8. Typen

A. Über das 'type' System

B. Der Grundtyp (LONG/PTR)

C. Die einfachen Typen (CHAR/INT/LONG)

D. Der Feldtyp (ARRAY)

E. Die komplexen Typen (STRING/LIST)

F. Der Verbundtyp (OBJECT)

G. Einrichtung

9. Eingebaute Funktionen

A. Ein-/Ausgabeoperationen

B. Zeichenketten und Zeichenkettenfunktionen

C. Listen und Listen Funktionen

D. Intuition unterstützende Funktionen

E. Grafikfunktionen

F. Systemfunktionen

G. Mathematische Funktionen

H. Funktionen zum Verbinden von Zeichenketten und Listen

10. Library Funktionen und Module

A. Eingebaute Library Aufrufe

B. Schnittstellen zum Amiga Sytem mit den 2.04 Modulen bilden

11. Ausgewertete Ausdrücke

A. Auswertung und Bereich

B. Eval()

C. Eingebaute Funktionen

12. Fließkommaunterstützung

A. Gebrauch/Überladen von Fließkommazahlen/-operatoren

B. Fließkommaausdrüke und Umwandlungen

13. Exception Behandlung

A. Definition von Exceptionhandlern (HANDLE/EXCEPT)

B. Benutzung der Raise() Funktion

C. Exceptions für eingebaute Funktionen (RAISE/IF)

D. Benutzung von Exception-ID's

14. Objektorientierte Programmierung

15. Der Inline-Assembler

A. Variablenteilung

B. Vergleich zwischen Inline-/Makroassembler

C. Wege, Binäredaten zu nutzen (INCBIN/CHAR..)

D. OPT ASM

16. Dinge über den Compiler

A. Das OPT Schlüsselwort

B. Small/Large Modell

C. Stack Organisation

D. Festgeschriebene Begrenzungen

E. Fehlermeldungen, Warnungen und nicht dokumentierte Tests

F. Compiler Puffer Organisation und Anforderung

G. Eine kurze Entstehungsgeschichte

1.4 Formatierung

1.FORMATIERUNG

A. Tabulatoren(tabs), Zeilenvorschübe(lf) usw.

B. Kommentare

C. Identifier und Typen

Vorheriges Kapitel

Nächstes Kapitel

1.5 Formatierung

1A. Tabulatoren(tabs), Zeilenvorschübe(lf) usw.

E-Sources sind reine ASCII-Dateien, mit Zeilenvorschüben (lf) und Semikolons ";" als Trennung zwischen zwei Ausdrücken. Ausdrücke die mehrere Argumente haben, die durch ein Komma "," getrennt sind, können über mehrere Zeilen verteilt werden, wenn die Zeile mit einem Komma endet, indem der folgende Zeilenvorschub ignoriert wird.
Jedes lexikalische Elemente in einer Source-Code-Datei kann durch eine beliebige Anzahl von Leerzeichen oder Tabulatoren vonm nächsten getrennt werden.

1.6 Formatierung

1B. Kommentare

Kommentare können überall im Sourcecode plaziert werden, wo Leerzeichen korrekt wären. Sie beginnen mit "/*" und enden mit "*/" und können unendlich verschachtelt werden.

1.7 Formatierung

1C. Identifier und Typen

Identifier sind Strings die Programmierer benutzen um bestimmte Objekte zu benennen, in den meisten Fällen Variabeln, oder nur Schlüsselwörter oder Funktionsnamen die vom Compiler vordefiniert wurden. Ein Identifier kann aus folgenden bestehen:

-große und kleine Buchstaben
-"0".."9" außer als ersten Buchstaben
-"_" dem Unterstrich

Alle Zeichen werden beachtet, aber der Compiler benutzt nur die ersten beiden um den Typ des Identifiers zu bestimmen, mit dem er ihn behandelt:

beide groß - Schüsselwort wie IF, PROC usw
- Konstanten wie MAX-LENGTH
- Assembler Mnemonic, wie MOVE

erster klein - Identifier von Variabel/Sprungmarken/Objekten usw.

erster groß und
zweiter klein - E-System-Funktionen
- Library Aufrufe: z.B. OpenWindow()

Zu bedenken ist, daß alle Identifier dieser Schreibweise folgen, zum Beispiel: WBenchToFront wird zu WbenchToFront

1.8 Direkte Werte

2.DIREKTE WERTE

Direkte Werte werden in E immer zu einem 32 Bit Ergebniss umgewandelt. Der einzige Unterschied zwischen diesen Werten (A-G) ist entweder ihre interne Darstellung, oder die Tatsache, daß sie einen Zeiger anstatt eines Wertes zurückgeben.

A. Dezimalzahlen (1)

B. Hexadezimalzahlen ($1)

C. Binärzahlen (%1)

D. Fließkommazahlen (1.0)

E. Zeichen ('a')

F. Zeichenketten ('bla')

G. Listen ([1,2,3]) und symbolische Listen

Vorheriges Kapitel

Nächstes Kapitel

1.9 Direkte Werte

```
2A. Dezimalzahlen (1)
---------------------

Eine Dezimalzahl ist eine Folge der Zeichen "0"..."9", möglicherweise durch
ein Minuszeichen "-" angeführt um negative Zahlen zu kennzeichnen.
Beispiel: 1, 100, -12, 1024
```

1.10 Direkte Werte

```
2B. Hexadezimalzahlen ($1)
--------------------------

Ein hexadezimaler Wert benutzt die zusätzlichen Zeichen "A"..."F" (oder
"a"..."f") und wird mit einem "$" Zeichen begonnen.
Beispiele: $FC, $DFF180, -$ABCD
```

1.11 Direkte Werte

```
2C. Binärzahlen (%1)
--------------------

Binärzahlen beginnen mit einem "%" Zeichen und benutzen nur die Zeichen
"1" und "0" um einen Wert zu bilden.
Beispiele: %111, %1010100001, -%10101
```

1.12 Direkte Werte

```
2D. Fließkommazahlen (1.0)
--------------------------
```

Fließkommazahlen unterscheiden sich nur durch einen "." von normalen Dezimalzahlen, der dazu dient die beiden Teile auseinander zu halten. Jeweils einer der beiden Teile darf weggelassen werden, nie beide. Zu Bedenken ist, daß Fließkommazahlen eine andere interne 32 Bit Darstellung (FFP) haben. Mehr Informationen über Fließkommazahlen gibt es im Kapitel 12.
Beispiele: 3.14159, .1 (entspricht 0.1), 1. (entspricht 1.0)

1.13 Direkte Werte

2E. Zeichen ("a")

Der Wert eines Zeichens (in Anführungszeichen "" eingeschlossen) ist ihr ASCII Wert, z.B. "A"=65. In E können direkte Zeichenwerte kurze Zeichenketten mit bis zu 4 Zeichen sein, zum Beispiel "FORM", wobei das erste Zeichen "F" das MSB und "M" das LSB (least significat Bit) der 32 Bit-Darstellung ist.

1.14 Direkte Werte

2F. Zeichenketten ('bla')

Zeichenketten sind irgendwelche ASCII-Darstellungen, die von Hochkommatas '' (Alt+'Ä') eingeschlossen sind. Der Wert einer solchen Zeichenkette ist ein Zeiger auf das erste Zeichen der Kette. Spezifischer: 'bla' erzeugt einen 32 Bit Zeiger zu einem Speicherbereich wo wir die Bytes "b", "l" und "a" finden können. Alle Zeichenketten in E werden mit einem 0 Byte abgeschlossen.
Zeichenketten können Formatzeichen, von einem Backslash "/" angeführt, enthalten. Entweder um Zeichen in eine Zeichenkette einzufügen, die aus irgendwelchen Gründen nicht darstellbar sind, oder um Zeichenkettenformatierungsfunktionen wie von WriteF(), TextF() und StringF(), oder der Kickstart 2.0 Funktion Vprintf zu nutzen.

```
\n          ein Zeilenvorschub
\a oder ''  ein Hochkomma (das zum Einschließen von Zeichenketten benutzt
            wird)
\e          Escape (ASCII 27)
\t          Tabulator (ASCII 9)
\          Backslash
\0          ein Null-Byte. Wird nur selten benutzt, da jede Zeichenkette
            mit einem Null-Byte abgeschlossen wird
\b          ein Carriage Return (Wagenrücklauf)
```

Zusätzlich, bei Benutzung mit Format-Funktionen:

```
\d          schreibt ein Dezimalzahl
\h          schreibt eine Hexadezimalzahl
\s          schreibt eine String
\c          schreibt ein Zeichen
```

```
\z          schreibt Füllzeichen bei Nullen
\l          formatiert linksbündig
\r          formatiert rechtsbündig

Feldvariabeln können /d,/h und /s Codes folgen:

[x]         bezeichnet die genaue Feldbreite x
(x,y)       bezeichnet das Minimum und das Maximum y (nur bei Zeichenketten)

Beispiel: Schreibt eine Hexadezimalzahl mit 8 Stllen und führenden Nullen
          WriteF ('\z\h[8]\n',num)

Eine Zeichenkette kann über meherer Zeilen verteilt werde, indem man sie mit
einem "+" Zeichen und einen <lf> (Zeilenvorschub) verbindet:

'dies ist eine sehr lange Zeichenkette'+
'sie wurde über mehrere Zeilen verteilt'
```

1.15 Direkte Werte

```
2G. Listen ([1,2,3]) und symbolische Listen
-------------------------------------------

Eine direkte Liste ist das konstante Gegenstück des List Datentyps, genauso
wie eine "Zeichenkette" das konstante Gegenstück für den STRING oder den
ARRAY OF CHAR Datentyp ist. Beispiel:

[3,2,1,4]

ist ein Ausdruck, das als Wert einen Zeiger auf eine bereits fertigestellte
Liste hat. Eine Liste ist eine Darstellung im Speicher, die gleichwertig
mit ARRAY OF LONG ist, mit einigen zusätzlichen Längeninformationen an
einem negativen Offset. Du kannst diese direkten Listen überall benutzen,
wo eine Funktion einen Zeiger auf ein Feld von 32 Bit Werten oder eine Liste
erwartet. Beispiele:

['Zeichenkette',1.0,2.1]
[WA_FLAGS,1,WA_IDCMP,$200,WA_WIDTH,120,WA_HEIGHT,150,TAGDONE]

Schaue Dir das Kapitel über List-Funktionen für eine Besprechung von symbo-
lischen Listen und Detailinformationen an.
```

1.16 Ausdrücke

```
                        3.AUSDRÜCKE


                A. format

                B. precedence and grouping

                C. types of expressions
```

D. function calls

Vorheriges Kapitel

Nächstes Kapitel

1.17 Ausdrücke

```
3A. Format
----------

Ein Ausdruck ist ein Stück Quelltext, das aus Operatoren, Funktionen und
Klammern besteht, um einen Wert zu erstellen.
Sie bestehen meistens aus:

- direkten Werten wie im Kapitel 2 besprochen
- Operatoren wie im Kapitel 4 besprochen
- Funktionsaufrufen wie im Kapitel 3D besprochen
- Klammern wie im Kapitel 3B besprochen
- Variabeln oder Variabelausdrücken (siehe Kapitel 3C)

Beispiele für Ausdrücke

1
'Hallo'
$ABCD+(2*5)+Abs(a)
(a<1) OR (b>=100)
```

1.18 Ausdrücke

```
3B. Abarbeitung und Anordnung
-----------------------------

E hat keinen Abarbeitungsvorrang. D. h., daß Ausdrücke von links nach
rechts ausgewertet werden. Du kannst die Abarbeitung durch Einklammern
von (Unter-) Funktionen ändern:

1+2*3 /*=9*/    1+(2*3) /*=7*/   2*3+1 /*=7*/
```

1.19 Ausdrücke

```
3C. Arten von Ausdrücken
------------------------

Es gibt drei Arten von Ausdrücken, die für unterschiedliche Verwendungs-
zwecke genutzt werden:

- <var> besteht nur aus einer Variabel
- <varexp> bestehend aus einer Variabel, möglicherweise mit einem unary(??)
```

```
  Operator dazu, wie ++ (increment) oder [] (Array Operator). Hierfür siehe
  Kapitel 4D und 4G. Es zeigt einen veränderbaren Ausdruck an, wie Lvalues
  in C.
  Bedenke das diese unary (??) Operatoren nie Teil der Abarbeitung sind
- <exp>. Dies beinhaltet <var> und <varexp, und jede andere Art von Aus-
  druck
```

1.20 Ausdrücke

```
3D. Funktionsaufrufe
--------------------

Ein Funktionsaufruf ist eine zeitweilig Unterbrechung des aktuellen Codes,
um in eine Funktion zu springen. Dies kann entweder eine selbstgeschriebene
Funktion (PROC), oder eine Funktion die vom System zu verfügung gestellt
wird.
Das Format eines solchen Funktionsaufruf ist der Name der Funktion, gefolgt
von zwei Klammern(), die von Null bis zu unendlich vielen Argumenten, die
durch Kommas getrennt werden, enthält. Argumente für Funktionen sind
wiederum Ausdrücke. In Kapitel 6 steht wie man seine eigene Funktion er-
stellt und Kapitel 9 und 10 beschreiben die eingebauten Funktionen. Bei-
spiele:

foo (1,2)
Gadget(buffer,glist,2,0,40,80+offset,100,'Cancel')
Close(handle)
```

1.21 Operatoren

```
                    4.OPERATOREN


          A. mathematische (+ - * /)

          B. Vergleiche (= <> > < >= <=)

          C. Logische und Bitweise (AND OR)

          D. Unary (SIZEOF ` ^ {} ++ -- -)

          E. Dreifach (IF THEN ELSE)

          F. Strukturen (.)

          G. Felder ([])

          H. Fließkommaoperator (|)

          I. Zuweisungsoperator (:=)

          J. Reihenfolge (BUT)
```

Vorheriges Kapitel

Nächstes Kapitel

1.22 Operatoren

```
4A. Mathematische (+ - * /)
---------------------------
```

Diese festen Operatoren verbindett einen Ausdruck mit einem anderen Wert, um einen neuen Wert zu bilden. Beispiele:

```
1+2, MAX-1*5
```

in Kapitel 12 wird beschrieben, wie man diese Operatoren überlädt um sie Fließkommazahlen zu benutzen. "-" kann als erster Teil eines Ausdrucks benutzt werden, natürlich inklusive 0.

```
z.B. sind -a oder -b+1 zulässig
```

Bedenke, daß * und / standardmäßig 16 Bit Operatoren sind: siehe Mul()

1.23 Operatoren

```
4B. Vergleiche (= <> > < >= <=)
-------------------------------
```

Gleich wie mathematische Operatoren, mit dem Unterschied, das ihr Ergebniss entweder TRUE (Wahr) (32 Bit Wert -1) oder FALSE (Falsch). Diese können auch für Fließkommazahlen überladen werden.

1.24 Operatoren

```
4C. Logische und Bitweise (AND OR)
----------------------------------
```

Diese Operatoren kombinieren entweder Wahrheitswerte zu neuen oder operieren bitweise mit AND und OR. Beispiele:

```
(a>1) AND ((b=2) OR (c>=3))   /*logisch */
a:=b AND $FF                  /*bitweise*/
```

1.25 Operatoren

```
4D. Unary (SIZEOF ^ {} ++ -- `)
-------------------------------
```

```
- SIZEOF <Objektidentifikator>
  gib einfach nur die Größe eines bestimmten Objekts zurück
  Beispiel: SIZEOF newscreen

- [<var>]
  Gibt die Addresse einer Variabel oder eines Lables zurück. Die ist der
  Operator Du benutzen würdest, wenn Du eine Variabel als Argument an eine
  Funktion über Referenz anstatt des Wertes (wie es Standard in E ist)
  übergibst. Siehe auch "^"
  Beispiel: Val(input,[x])
- ^<var>
  Das Gegenstück von [], schreibt und ließt Variabel die über Referenz
  übergeben wurden.
  Beispiel: ^a:=1    b:=^a
  Es kann aber auch einfach genutzt werden um Longwerte aus dem Speicher
  zu "peeken" und zu "pooken", wenn <var> ein Zeiger auf einen solchen Wert
  ist.
  Beispiel für [] und ^: Wir schreiben unsere eigene Zuweisungsfunktion.

  PROC set(var,exp)
    ^var:=exp
  ENDPROC

  und rufen sie auf mit: set ([a],1) (gleich a:=1)

- <varexp>++ und <varexp>--
  Erhöht (++) oder erniedrigt (--) den Zeiger der von <varexp> bezeichnet
  wird um die Größe des Datums auf das er zeigt. Dies hat den Effekt, das
  der Zeiger auf das nächste bzw. das vorherige Element zeigt. Wenn diese
  Operatoren mit einer Variabel benutzt werden, die kein Zeiger ist, so
  wird diese einfach um eins verändert. Wichtig ist das ++ immer nach der
  Berechnung von <varexp> abgearbeitet wird, während -- immer vorher be-
  rechnet wird.

  a++       /* gibt den Wert von A zurück, und erhöht A um eins */
  sp[]--    /* erniedrigt den Zeiger um 4 (wenn es sich um ARRAY OF LONG
               handelt) und ließt den Wert auf den sp zeigt */

- '<exp>
  Dies ist ein quotierter Ausdruck von Lisp genannt. <exp> wird nicht aus-
  gewertet, stattdessen gibt es die Addresse des Ausdrucks zurück, welcher
  später ausgewertet werden kann. Mehr über diese speziellen Fähigkeit in
  Kapitel 11.
```

1.26 Operatoren

```
4E. Dreifach (IF THEN ELSE)
---------------------------

Der IF Operator hat genau dieselbe Funktion wie das IF Statement, nur das
zwischen zwei Ausdrücken anstatt von zwei Statements oder Blöcken von
Statements aussucht. Es gleicht dem x?y:z Operator in C.

IF <wahrheitsausdruck> THEN <ausdruck1> ELSE <ausdruck2>
```

```
sich auf einen wahrheitsausdruck beziehend gibt dies ausdruck1 oder aus-
druck2 zurück. Zum Beispiel anstatt:

IF a<1 THEN b:=2 ELSE b:=3
IF x=3 then WriteF('x ist 3\n) ELSE WriteF('x ist etwas anderes\)

schreibe

b:=IF a>1 THEN 2 ELSE 3
WriteF(IF x=3 THEN 'x ist 3\n' ELSE 'x ist etwas anders\n')
```

1.27 Operatoren

```
4F. Strukturen (.)
------------------

<prt2objekt>.<memberofobjekt> [übersetzt heißt das <zeigeraufeinobjekt>.
<teileinesobjekts> Anmerkung des Übersetzers] bilden eine <varexp>.
Der Zeiger muß als PTR TO <objekt> oder ARRAY OF <objekt> deklariert sein,
(näheres in Kapitel 8) und der Teil muß ein realer Objektidentifier sein.
Das lesen eines Unterobjekts in einem Objekt auf diesem Weg, ergibt sich
ein Zeiger auf dieses Objekt. Beispiele:

thistask.userdata:=1
rast:=myscreen.rastport
```

1.28 Operatoren

```
4G. Felder ([])
---------------

<var>[<indexexp>] (ist eine <varexp>)
Dieser Operator ließt den Wert vom Feld auf die <var> zeigt, mit dem Index
<indexexp>. Der Index kann fast jeder Ausdruck sein, mit kleinen Ausdruck
sein, mit kleinen Ausnahmen, wie das keine Funktionsaufrufe darin vorkommen
dürfen, und wie es auf der linken Seite einer Zuweisung benutzt wird.
Bemerkung1: "[]" ist eine Abkürzung für "[0]"
Bemerkung2: bei einem Feld von n Elementen ist der Index 0..n-1

a[1]:=10            /*setzt das zweite Element auf 10*/
x:=table[y*4+1]     /*ließt aus dem Feld*/
```

1.29 Operatoren

```
4H. Fließkommaoperator (|)
--------------------------

<exp>\<exp>
Wandelt Ausdrücke von Integer nach Float und zurück, und überlädt die Opera-
toren + - * / = <> < > <= >= mit Fließkommaoperatoren. In Kapitel 12 steht
```

alles über Fließkommazahlen und diesen Operator.

1.30 Operatoren

```
4I. Zuweisungsoperator (:=)
---------------------------
```

Zuweisungen (setzen eines Variabelwerts) existieren als Statement und als Ausdruck. Der einzige Unterschied ist, daß die Statementversion die Form <varexp>:=<exp> hat am Ende hat ist das Ergebniss der Wert von <exp>. Bedenke das, wenn <var>.= als Ausdruck benutzt wird, man oft Klammern braucht um einen korrekten Interpretation zu erzwingen. So wie:

```
IF mem:=New(100)=NIL THEN error()
```

so interpretiert wird:

```
IF mem:=(New(100)=NIL) THEN error()
```

was nicht das ist, was Du wolltest. mem sollte ein Zeiger und kein Wahrheitswert sein. Also solltest Du schreiben

```
IF (mem:=New(100))=NIL THEN error()
```

1.31 Operatoreyn

```
4J. Reihenfolge (BUT)
---------------------
```

Der Reihenfolgenoperatoren "BUT" erlaubt es zwei Ausdrücke dorthin schreiben, wo nur einer erlaubt ist. Oft, beim schreiben komplexer Funktionsaufrufe/Ausdrücke, möchte man noch direkt eine zweite Sache machen, wie eine Zuweisung. Syntax:

```
<exp1> BUT <exp2>
```

D. h. Berechne Wert eins aber gib den Wert von exp2 zurück. Beispiel:

```
myfunc((x:=2) BUT x*x)
```

weißt 2 x zu und ruft dann myfunc mit x*x auf. Die () um die Zuweisung werden wieder benötigt um den := Operator davon abzuhalten (2 BUT x*x) als einen Ausdruck anzusehen.

1.32 Statements

```
                        5.STATEMENTS


                A. Format (;)
```

B. Sprungmarken und Sprunganweisungen (JUMP)

C. Zuweisungen (:=)

D. Assembler Mnemonics

E. Bedingte Anweisungen (IF)

F. For-Anweisung (FOR)

G. While-Anweisung (WHILE)

H. Repeat-Anweisung (REPEAT)

I. Loop-Anweisung (LOOP)

J. Auswahl-Anweisungen (SELECT)

K. Zuwachs-Anweisungen (INC/DEC)

L. Void-Ausdrücke (VOID)

Vorheriges Kapitel

Nächstes Kapitel

1.33 Statements

```
5A. Format (;)
--------------
```

Wie im Kapitel 1A, steht ein Statement in seiner eigenen Reihe, aber mehrere von ihnen können in einer Reihe, durch ein Semikolon getrennt, geschrieben werden. Auch kann ein Statement über mehr als eine Zeile verteilt werde, wobei jede Zeile mit einem Komma enden muß. Beispiel:

```
a:=1, WriteF('Hallo!\n')
DEF a,b,c,d,        /*zuviele Argumente für eine Zeile (vorgemacht)*/
    e,f,g
```

Statement können sein

- Zuweisungen
- Bedingt Zuweisungen für Statement und sowas, siehe auch Kapitel 5E-5K
- Rückgabefreie Ausdrücke
- Sprungmarken
- Assembleranweisungen

Das Komma ist das erste Zeichen um zu zeigen, das Du das Statement nach dem nächsten Zeilenvorschub noch nicht beenden willst, aber auch die folgenden Zeichen zeigen an, daß das Statement in der folgenden Zeile fortgeführt wird:

```
+ - * /
```

```
= > < <> >= <=
AND OR BUT THEN
```

1.34 Statements

```
5B. Sprungmarken und Sprunganweisungen (JUMP)
---------------------------------------------

Lables sind globale Identifier mit einem zusätzlichen ":", wie bei

mylable:

sie können von Anweisungen wie JUMP benutzt werden, aber auch um statische
Daten zu erzeugen. Sie können benutzt werden, um aus jeder Art von Schleife
zu springen (obwohl diese Technik nicht gut ist), aber es kann nicht aus
einer Prozedure gesprungen werden. In normalen E-Programmen werden sie
meistens für den Inline-Assembler benutzt. Lables sind immer global sicht-
bar.

Benutzung von JUMP: JUMP <lable>

führt die Abarbeitung beim <lable> fort. Du solltest dies aber nicht be-
nutzen, es ist eigentlich nur für Situationen da, wo man die Komplexität
eines Programms vermindert. Beispiel:

IF Mouse()=1 THEN JUMP stopnow

/*andere Programmteile*/

stopnow:
```

1.35 Statements

```
5C. Zuweisungen (:=)
--------------------

Das grundsätzliche Format einer Zuweisung ist <var>:=<exp>
Beispiele: a:=1, a:=myfunc(), a:=b*3
```

1.36 Statements

```
5D. Assembler Mnemonics
-----------------------

In E ist der Inline-Assembler ein wirklicher Teil der Sprache, er muß nicht
in spezielle "ASM" Blöcke oder sowas, wie es in anderen Sprachen notwendig
ist, noch ist ein extra Assembler nötig, um den Code zu Asseblieren. Das
heißt auch, daß er den normalen E Syntax-Regeln gehorcht, usw. In Kapitel
15 kannst Du alles über den Inline-Assembler lesen. Beispiele:
```

```
DEF a,b
b:=2
MOVEQ   #1,D0       /*nur einige Assembler-Statements*/
MOVEL   D0,a        /*a:=1+D*/
ADD.C   b,a
WriteF('a=\d\n',a)  /*a wird 3 sein*/
```

1.37 Statements

```
5E. Bedingte Anweisungen (IF)
-----------------------------

IF, THEN, ELSE, ELSEIF, ENDIF

Syntax: IF <exp> THEN <Statment> [ELSE <Statement>]
oder  : IF <exp>
                <Statements>
        [ELSEIF <exp>           /*mehrere elseifs können vorkommen*/
                <Statements>]
        [ELSE
                <Statements>]
        ENDIF
```

bilden einen Bedingten-Block. Bedenke das es zwei Hauptformen von diesem Statement gibt, eine einzeilen und eine mehrzeilen Version.

1.38 Statements

```
5F. For-Anweisung (FOR)
-----------------------

FOR, TO, STEP, DO, ENDFOR

Syntax: FOR <var>:=<exp> STEP <Schrittweite> DO <Statement>
oder  : FOR <var>:=<exp> STEP <Schrittweite>
                        <Statements>
        ENDFOR
```

bilden einen FOR-Block, beachte die beiden Hauptformen. <Schrittweite> kann jede positive oder negative Konstanten, außer 0 sein. Beispiele:

```
FOR a:=1 TO 10 DO WriteF('\d\n'a)
```

1.39 Statements

```
5G. While-Anweisung (WHILE)
---------------------------

WHILE, DO, ENDWHILE
```

```
Syntax: WHILE <exp> DO <Statement>
oder  : WHILE <exp>
                <Statements>
        ENDWHILE
```

bilden eine While-Schleife, die solange durchlaufen wird, wie <exp> TRUE ergibt. Beachte die Ein-Zeile/Ein-Statement-Version und die Mehrzeilen-Version.

1.40 Statements

```
5H. Repeat-Anweisung (REPEAT)
-----------------------------

REPEAT, UNTIL

Syntax: REPEAT
            <Statements>
        UNTIL <exp>
```

bilden einen Repeat-Until-Block. Die Schleife wird fortgesetzt bis <exp>= TRUE ist. Beispiel:

```
REPEAT
    WriteF('Möchtest du wirklich dieses Programm beenden?\n'
    ReadStr(stdout,s)
UNTIL StrCmp(s,'ja bitte!')
```

1.41 Statements

```
5I. Loop-Anweisung (LOOP)
-------------------------

LOOP, ENDLOOP

Syntax: LOOP
            <Statements>
        ENDLOOP
```

bilden eine unendliche Schleife.

1.42 Statements

```
5J. Auswahl-Anweisungen (SELECT)
--------------------------------

Syntax: SELECT <var>
            [CASE <exp>
                <Statements>]
            [CASE <exp>
```

```
                <Statements>]   /*eine beliebige Anzahl weiterer Blöcke*/
            [DEFAULT <exp>
                <Statement>]
        ENDSELECT
```

bilden einen Select-Case-Block. Beliebige Ausdrücke werden mit der Variabel <var> verglichen, und der erste passende Block wird ausgeführt. Wenn kein gleicher Ausdruck vorhanden ist, kann ein Default-Block ausgeführt werden.

```
SELECT zeichen
    CASE 10
        WriteF('Hey, wir haben einen Zeilenvorschub gefunden\n')
    CASE 9
        WriteF('Wow, das muß ein Tabulator sein!\n')
    DEFAULT('Kennst Du dieses Zeichen: \c?\n',zeichen)

ENDSELECT
```

1.43 Statements

```
5K. Zuwachs-Anweisungen (INC/DEC)
---------------------------------

INC, DEC

Syntax: INC <var>
        DEC <var>
```

kurz für <var>:=<var+1> und <var>:=var-1. Der einzige Unterschied zu var++ und var-- ist, daß dies Statements sind, und keinen Wert zurückgeben und folglich optimaler sind.

1.44 Statements

```
5L. Void-Ausrücke (VOID)
------------------------

Syntax: VOID <exp>
```

Berchnet den Ausdruck ohnen das der Wert irgendwohin geht. Dies ist nur für eine lesbarere Syntax nützlich, da Ausdrücke in E auch als Statement ohne VOID benutzt werden können. Dies kann heikle Fehler verursachen, da "a:=1" a den Wert 1 zuweist, aber "a=1" als Statement nichts tut. E warnt Dich, wenn das passiert.

1.45 Function Definitions and Declarations

6.FUNKTIONS DEKLARATION UND DEFINITION

A. Prozedur Definition und Argumente (PROC)

B. Lokale und globale Definitionen (DEF)

C. Endproc/return

D. Die 'main' Funktion

E. E-eigene Systemvariablen

Vorheriges Kapitel

Nächstes Kapitel

1.46 Funktions Deklaration und Definition

6A. Prozedur Definition und Argumente (PROC)

Du darfst PROC und ENDPROC zur Zusammenfassung von Ausdrücken in deinen eigenen Funktionen zu verwenden.

Solche ein Funktion darf irgendwelche Anzahl von Argumente haben und liefert einen Wert zurück.

PROC, ENDPROC

Syntax: PROC <Marke> (<Argumente> , ...)

ENDPROC <Rückgabewert>

Definiert eine Prozedur mit irgendeiner Anzahl von Argumenten.
Argumente sind von Typ LONG oder Optional als PTR TO <TYP> (siehe
Kapitel 8
)
und brauchen kein weitere Deklarationen.

Das Ende einer Prozedur wird gekennzeichnet durch ENDPROC. Wenn kein Rückgabewert mitgegeben wird, so wird 0 zurückgeliefert.

Beispiel: Eine Funktion, die zwei Argumente zusammengefasst zurückliefert:

```
PROC add(x,y)          /* x und y sind lokale Variablen */
ENDPROC x+y            /* liefert das Resultat zurück  */
```

1.47 Funktions Deklaration und Definition

```
                6B. Lokale und globale Definitionen (DEF)
----------------------------------------

Du darfst lokale Variablen definieren, indem Du die Argumente mit der
DEF-Angabe bestimmst. Der leichteste Weg ist


DEF a,b,c


erklärt die Bezeichner a, b und c als Variablen deiner Funktion.

Beachten daß solche Deklarationen am Anfang Deiner Funktionen stehen müssen.



DEF


Syntax:         DEF <Deklarationen>,...

Beschreibung    definiert Variablen. Eine Deklarationen hat eine der Formen:

                <Variable>

                <Variable>:<Typ>          wobei <typ>=LONG,<objectidentifizierer>

                <Variable>[<Größe>]:<Typ> wobei <typ>=ARRAY,STRING,LIST



Siehe
                Kapitel 8
                 für mehr Beispiele, wo steht, wie die Typen benutzt
werden.

Fürs erste ist es gut die <Variable>-Form zu benutzen.
Argumente für Funktionen sind beschränkt auf die Basis-Typen; siehe
                Kapitel 8B
                .

Ein Deklarationen eines Basis-Types kann eine Initialisierung haben, in der
aktuellen Version muß dieses ein Integer sein (kein Ausdruck):


DEF a=1,b=2



Ein Programm beinhaltet mehrere Funktionen, genannt PROCs. Jede

procedure may have Local variables, and the program as a whole may have
Prozedur darf lokale Variablen haben, und das Programm als ganzes
```

```
darf globale Variablen haben.
Ein Programm muß mindestens aus der Prozedur PROC main() bestehen, da diese
Prozedur den Anfang der Ausführung bestimmt.

Ein einfaches Programm könnte wie folgt ausschauen:


DEF a, b                            /* Definition der globalen Variablen */

PROC main()                         /* alle Funktionen in zufälliger Ordnung */

  bla(1)

ENDPROC

PROC bla(x)

  DEF y,z                           /* Eigene, lokalen Variablen möglich. */

ENDPROC


Zusammengefasst: lokale Definitionen sind diejenigen, die Du am Start der
Prozeduren nennst und welche erst in der Prozedur sichtbar werden. Globale
Definitionen werden vor der ersten PROC gemacht, also am Start deines
Quell-Codes, und sie sind global verfügbar. Globale und lokale Variablen
(und natürlich lokale Variablen von zwei unterschiedlichen Funktionen)
dürfen den gleichen Name haben.

Lokale Variablen haben immer höhere Priorität!
```

1.48 Funktions Deklaration und Definition

```
                6C. Endproc/return
------------------

Wie zuvor gesagt, markiert ENDPROC das Ende einer Funktionen-Definition,
und darf einen Wert zurückliefern.
Optional kann RETURN an irgendwelchen Punkt in der Funktion zum beenden der
Funktion benutzt werden. Wenn REURN in der Prozedur main() benutzt wird,
dann endet das Programm.

Siehe auch CleanUp() in
                Kapitel 9F
                .

RETURN [<Rückgabewert>]          /* optional */


Beispiel:



PROC getresources()
```

```
  /* ... */

  IF error THEN RETURN FALSE  /* etwas ist schiefgegangen, also raus
                                 aus der Funktion mit dem Rückgabewert
                                 "FALSCH". */

  /* ... */

ENDPROC TRUE  /* wir sind soweit, damit liefern wir TRUE zurück. */


Eine sehr kurz Version bei einer Funktion-Definition ist



PROC <marke> ( <Argument> , ... ) RETURN <Ausdruck>


These are function definitions that only make small computations, like
Diese sind Funktions-Definitionen die kleine Berechnungen machen, wie
fakultative Funktionen und sie sind (ein-Zeiler :-)



PROC fac(n) RETURN IF n=1 THEN 1 ELSE fac(n-1)*n
```

1.49 Funktions Deklaration und Definition

```
6D. Die "main" Funktion
-----------------------
Die PROC mit dem Namen main() ist unheimlich Wichtig, weil es
die erste aufgerufene Funktion ist; sie ist genauso aufgebaut wie andere
Funktionen und darf auch lokale Variablen haben.

main() hat keine Argumente; die CLI-Kommando-Zeilen Argumente werden in
der System-Variable "arg" geliefert, oder können überprüft werden mit

ReadArgs()
```

1.50 Funktions Deklaration und Definition

```
6E. E-eigene Systemvariablen
----------------------------
Folgende globale Variablen sind immer verfügbar in Deinem Programm,
sie werden System Variablen genannt.



arg             Wie oben gesagt, beinhaltet "arg" einen Zeiger zu einem mit
                mit Null abgeschlossen String. Der String wiederum
```

```
                beinhaltet die CLI-Kommando-Zeilen-Argumente.
                Benutze nicht diese Variable wenn du ReadArgs() in
                Deinem Programm benutzt.

stdout          Beinhaltet ein File-Handle zu der Standard Ausgabe
                (und Eingabe).
                Wenn dein Programm von der Workbench gestartet wurde, so daß
                kein Shell-Fenster verfügbar ist, WriteF() öffnet ein CON:
                Fenster für dich und setzt den entsprechenden File-Handler
                hier hinein.

conout          Hier wird das File-Handle abgelegt, und das CON:
                Fenster wird automatisch geschlossen nach einem exit
                von Deinem Programm.
                Siehe WriteF() in Abschnitt 9E, um zu sehen, wie die zwei
                Variablen richtig benutzt werden.

execbase,       Diese fünf Variablen enthalten IMMER die
dosbase,        richtigen Werte.
gfxbase,
intuitionbase,
mathbase

stdrast         Zeiger zum Standard-Rastport, um diesen in deinem Programm
                benutzten zu können, oder NIL.
                Die eingebauten Grafik-Funktionen wie Line() benutzen
                diese Variable.

wbmessage       Beinhaltet einen Zeiger zu ein Nachricht die du bekommst,
                wenn du von der Worbench startest, sonst NIL.
                Darf benutzt werden wie ein Boolean-Wert um festzustellen,
                ob du von der Workbench gestartet bist oder sogar zum
                überprüfen irgendwelcher Argumente, die mit Deinem ICON
                "Shift-ausgewählt" wurden.
                Siehe WbArgs.e in dem sources/Examples Verzeichnis, um zu
                sehen, wie gut man wbmessage benutzen kann.
```

1.51 Deklaration von Konstanten

```
                        7.DEKLARATION VON KONSTANTEN


                A. Konstanten (CONST)

                B. Aufzählungen (ENUM)

                C. Sets (SET)

                D. E-eigene Konstanten

                Vorheriges Kapitel

                Nächstes Kapitel
```

1.52 Deklaration von Konstanten

```
7A. Konstanten (CONST)
----------------------
Syntax: CONST <Deklaration>

Ermöglicht es Dir eine Konstante zu deklarieren. Eine Deklaration sieht so
aus:

<ident>=<Wert>

Konstanten müssen großgeschrieben sein, und werden für den Rest des Pro-
gramms als <Wert> behandelt. Beispiel:

CONST MAX_LINES=100, ER_NOMEM=1, ER_NOFILE=2

Man kann keine Konstanten mit Termen deklarieren in denen Konstanten sind
die im selben Statement deklarier werden: Schreibe diese ins nächste.
```

1.53 Deklaration von Konstanten

```
7B. Aufzählungen (ENUM)
-----------------------

Aufzählungen sind ein spezieller Typ von Konstanten, bei denen kein Wert an-
gegeben werden braucht, da sie im Bereich von 0..n definiert sind, das erste
Element ist 0. An einem beliebigen Punkt in einer Aufzählung kannst Du
"=<Wert>" einsetzen um den Zähler auf einen Wert zu setzen oder rückzuset-
zen. Beispiele:

ENUM ZERO, ONE, TWO, THREE, MONDAY=1, TUESDAY, WENDSDAY
ENUM ER_NOFILE=100, ER_NOMEM, ER_NOWINDOW
```

1.54 Deklaration von Konstanten

```
7C. Sets (SET)
--------------
Sets sind den Aufzählungen ähnlich, mit dem Unterschied, daß anstatt einem
Wert wie (0,1,2..) eine Bitnummer hat (0,1,2...) die erhöht wird. So haben
Sets die Werte (1,2,4,8...). Das hat den zusätzlichen Vorteil, daß sie als
Sets von Flags benutzt werden können, wie das Schlüsselwort sagt.
Stellen wir uns ein Set wie oben vor, das die Einrichtung eines Fensters
beschreibt:

SET SIZEGAD, CLOSEGAD, SCROLLBAR, DEPTH

um eine Variabel mit den Werten DEPTH und SIZEGAD vorzubereiten:

winflags:=DEPTH OR SIZEGAD

um einen zusätzlichen SCROLLBAR einzurichten
```

```
winflags:=winflags OR SCROLLBAR

und testen ob zwei Einrichtung gehalten wurden

IF winflags AND (SCROLLBAR OR DEPTH) THEN /*   */
```

1.55 Deklaration von Konstanten

```
7D. E-eigene Konstanten
-----------------------
Folgende eingebaute Konstanten können benutzt werden:

TRUE, FALSE         Repräsentiert die boolschen Wert Wahr und Falsch (-1,0)
NIL                 (=0) nicht initialisierter Pointer
ALL                 wird bei String-Funktionen wie StrCopy benutzt, damit
                    alle Zeichen kopiert werden.
GADGETSIZE          kleinste Größe in Bytes um ein Gadget zu erhalten, siehe
                    auch Gadget() 9D
OLDFILE, NEWFILE    Modusprameter beim benutzen von Open()
STRLEN              Hat immer den Wert der zuletzt benutzten Zeichenkette.
                    Beispiel: Write(handle,'Hallo Leute!',STRLEN) /* =9 /*
```

1.56 Typen

```
                    8.TYPEN


               A. Über das 'type' System

               B. Der Grundtyp (LONG/PTR)

               C. Die einfachen Typen (CHAR/INT/LONG)

               D. Der Feldtyp (ARRAY)

               E. Die komplexen Typen (STRING/LIST)

               F. Der Verbundtyp (OBJECT)

               G. Einrichtung

               Vorheriges Kapitel

               Nächstes Kapitel
```

1.57 Typen

8A. Über das 'type' System

E hat kein strenges Typen-System wie in Pascal oder Modula2, es ist sogar flexibler als in C. Du kannst es auch ein Datentypen-System. Dies geht Hand in Hand mit der Philosophie, daß alle Datentypen in E gleich sind: alle kleinen Grundtypen wie Zeichen, Dezimalzahlen etc. Alle haben die gleichen 32 Bitgröße und alle anderen Daten und alle anderen Datentypen wie Felder und Zeichenketten werden von auf einen 32 Bitzeiger auf sie. So kann der Compiler einen vielseitigen Code generieren.
Die Vor- und Nachteile sind offensichtlich:

Nachteile des E-Typen-Systems:

- weniger Compiler-Überprüfungen für dumme Fehler von DIR!

Vorteile:

- Low-Level Vielseitigkeit
- flexible Programmgestaltung: kein Problem das einige Typen Werte zurück geben die nicht zu anderen Aufrufe usw.
- es ist nicht schwer Fehler beim mischen von verschieden großer Daten in Ausdrücken zu finden
- er werden immer noch selbstdokumentierende Typen unterstützt, wie:

  PTR to newscreen

1.58 Typen

8B. Der Grundtyp (LONG/PTR)

Es gibt nur einen nicht komplexen Grundtyp in E, es ist der 32 Bit Typ LONG. Da er der Defaulttyp ist, kann er so definiert werden:

```
DEF a:LONG          oder nur    DEF a
```

Diese Variabel kann das aufnehmen, was in anderen Programmiersprachen die Typen CHAR/INT/PTR/LONG enthalten. Eine spezielle Art des LONG Typen ist der PTR-Typ. Dieser Typ ist kompatibel mi dem LONG-Typen, mit dem Unterschied, daß angegeben wird, auf was er zeigt. Als Voreinstellung gilt, das LONG als PRT TO CHAR angegeben wird. Syntax:

```
DEF <var>:PTR TO <Typ>
```

wobei Typ entweder ein einfacher oder ein zusammengesetzter Typ ist. Beispiel:

```
DEF x:PTR TO INT, myscreen: PTR TO screen
```

Beachte, daß 'screen' der Name eines Objekts ist, das im Module intuition/screens.m definiert ist. Zu Beispiel wenn Du Deinen eigenen Screen öffnest mit:

```
myscreen:=OpenS(... usw.
```

kannst Du den Zeiger myscreen z.B. als 'myscreen.rastport' benutzen. Wenn
Du aber nichts mit den Variabeln machen willst, bis Du CloseS(myscree)
aufrufst, kannst Du sie so deklarieren:

```
DEF myscreen
```

1.59 Typen

```
8C. Die einfachen Typen (CHAR/INT/LONG)
---------------------------------------
```
Die einfachen Typen CHAR(8 Bit) und INT (16 Bit) sollten nicht als Grund-
typen für (einzelne) Variabeln benutzt werden; der Grund dafür sollten
jetzt wohl klar sein. Trotzdem können sie als Datentypen zum Bilden von
Arrays benutzt werden. Sie können auch in Objektdefinitionen benutzt wer-
den, und man kann einen Zeiger auf sie setzen.

1.60 Typen

```
8D. Der Feldtyp (ARRAY)
-----------------------
```
ARRAYs werden über ihre Länge in Bytes definiert:

```
DEF b[100]:ARRAY
```

dies definiert ein Feld von 100 Bytes. Intern ist b eine Variabel des Typs
LONG und ein Zeiger auf diesen Speicherbereich. Standardmäßig ist der Typ
eines Feldelements CHAR, es kann aber jeder andere sein:

```
DEF x[100]: ARRAY OF LONG
DEF mymenus[10]:ARRAY OF newmenu
```

wobei "newmenu" ein Beispiel für eine Struktur ist, die in E OBJECT genannt
werden.
Der Feldzugriff ist sehr einfach mit <var>[<sexp>]:

```
b[1]:="a"
z:=mymenu[a+1].multiexclude
```

Bedenke das der Index eines Felds der Größe n von 0 bis n-1 und nicht von
1 bis n geht.
ARRAY OF <Typ> ist kompatibel mit PRT TO <Typ>, mit dem Unterschied das die
Variabeln die in einem Feld sind schon eingerichtet.

1.61 Typen

```
8E. Die komplexen Typen (STRING/LIST)
-------------------------------------
```
- STRINGs. Ähnlich den Feldern, sind aber anders, weil sie nur von den E
  Zeichkettenfunktionen geändert werden können, und das sie Größen- und

Maximalgrößenangaben enthalten. So können die E-Funktionen den String auf sichere Art verändern. Z. B.: Der String kann nie größer werden als der Speicherbereich, indem er steht. Definition:

```
DEF s[80]:STRING
```

Der String-Typ ist abwärtskompatibel mit PTR OF CHAR und natürlich mit ARRAY OF CHAR, aber nicht andersherum. Mehr Details im Abschnitt über Zeichenkettenfunktionen.

- LISTs. Diesen Datentyp gibt es in anderen prozedualen Programmiersprachen nicht, es gibt ihn in Programmiersprachen wie Lisp oder Prolog. Die E-Variante kann als Mischung zwischen STRING und ARRAY OF LONG interpretiert werden. Zum Beispiel kann diese Datenstruktur eine Liste von LONG-Variabeln aufnehmen, welche als STRINGs erweitert oder verkürzt werden können. Definition:

```
DEF x[100]:LIST
```

Eine mächtige Erweiterung dieses Typs ist, daß er ein konstantes Gegenstück [], wie Zeichenketten mit '', hat. LIST ist abwärtskompatibel mit PTR TO LONG und natürlich ARRAY OF LONG, aber nichts andersherum. In Kapitel 2G und 9C steht mehr darüber.

1.62 Typen

8F. Der Verbundtyp (OBJECT)

OBJECTs sind fast wie eine struct in C oder ein RECORD in Pascal. Beispiel:

```
OBJECT meinobject
    a: LONG
    b: CHAR
    c: INT
ENDOBJECT
```

Dies definiert eine Struktur die aus drei Elementen. Syntax:

```
OBJECT
    <Elementname>[:<Typ>]   /*hiervon eine unbeschränkte Zahl*/
ENDOBJECT
```

wobei Typ wieder ein einfacher Typ, ein zusammengesetzter Typ oder ein einfacher Feldtyp ist, z. B. [<Elementzahl>]:ARRAY mit der Größe von CHAR für ein Element. Bedenke das <Elementname> nicht ein einzigartiger Identifier sein muß, er kann auch in anderen Objekten enthalten sein. Es gibt viele Arten um Objekte zu benutzen:

```
DEF x:meinobj               /* x ist eine Strucktur*/
DEF y:PRT TO meinobj        /* y ist ein Zeiger auf ein Struktur*/
DEF z[10]:ARRAY OF meinobj

y:=[-1,"a",100]:meinobj     /*geschriebene Liste*/
IF y.b="a" THEN /*...*/
z[4].c:=z[d+1].b++
```

```
ARRAY in Objekten werden immer auf gerade Zahlen gerundet und werden auf
gerade Offsets gesetzt.

OBJECT meinstring
    len:CHAR, daten[9]:ARRAY
ENDOBJECT

SIZEOF meinstring ist 12, und "daten" beginnt auf dem Offset 2.
Bedenke: OBJECTs in E sind nicht so wie sie in anderen Sprachen benutzt
werden können. Z. B. kann nicht jeder Typ ein Element eines Objekts bilden,
und deswegen machen rekursive Zugriffe wie x.y.z wenig Sinn (bis jetzt).
```

1.63 Typen

```
8G. Einrichtung
---------------
1. Immer mit NIL eingerichtet (oder anders wenn extra angegeben)
   -Globale Variabel
    Bedenke: Für Dokumentationszwecke ist es immer besser wnn Du =NIL in
             Definitionen von Variabeln schreibst, die du als NIL erwartest
2. Als '' oder [] eingerichtet:
   -Globale und lokale STRINGs
   -Globale und lokale LISTs
3. Nicht eingerichtet
   -Lokale Variabel (wenn nicht extra angegeben)
   -Globale und lokale ARRAYs
   -Globale und lokale OBJECTs
```

1.64 Eingebaute Funktionen

```
                    9.EINGEBAUTE FUNKTIONEN


               A. Ein-/Ausgabeoperationen

               B. Zeichenketten und Zeichenkettenfunktionen

               C. Listen und Listen Funktionen

               D. Intuition unterstützende Funktionen

               E. Grafikfunktionen

               F. Systemfunktionen

               G. Mathematische Funktionen

               H. Funktionen zum Verbinden von Zeichenketten und Listen

               Vorheriges Kapitel
```

Nächstes Kapitel

1.65 Eingebaute Funktionen

```
9A. Ein-/Ausgabeoperationen
---------------------------
WriteF(Formatzeichenkett, Argumente,...)

schreibt eine Zeichenkette (die Formatcodes enthalten kann) in den stdout.
Es können von keinem bis zu unendlich vielen Argumenten angefügt werden.
Bedenke, daß Formatzeichenketten dynamisch erzeugt werden können. Die An-
zahl der Argumente wird nicht überprüft. Beispiele:

WriteF('Hallo Welt!\n') /*schreibt nur eine Zeichenkette mit einem Zeilen-
                          vorschub am Ende*/
WriteF('a=\d\n',a)      /*schreibt "a=123" wenn a=123 ist*/

Alles andere mußt Du unter dem Thema Strings nachschauen.
Wenn stdout=NIList, zum Beispiel wenn Dein Programm von der Workbench ge-
startet wir, erzeugt WriteF() ein Ausgabefenster, und schreibt den Handle
in conout und stdout. Dieses Fenster wird am Ende des Programms geschlossen,
nachdem der Anwender ein <Return> eingegeben hat. WriteF() ist die einzige
Funktion die dieses Fenster öffnet. als, wenn Du eine Ein-/Ausgabefunktion
über stdout machen willst, un nicht sicher weißt ob stdout<>NIL benutze ein
WriteF('') als ersten Befehl deines Programms um die Ausgabe sicher zu
stellen. Wenn Du selbst ein Consolen-Fenster öffnen willst, solltest Du den
resultierenden Filehandle in die 'stdout' und 'conout' Variabel schreiben,
da Dein Fenster so nach dem Programmende automatisch geschlossen wird. Wenn
Du das Fenster manuelle schließen willst, vergewisser Dich, daß Du 'conout'
wieder auf NIL gesetzt hast, um E anzuzeigen das es kein Console-Fenster
gibt, das geschlossen werden muß.

Out(filehandle, char) und char:=Inp(filehandle)

Schreibt oder ließt ein einzelnes Byte in/aus einem File oder stdout. Wenn
char=-1 ist, ist das Ende des Files erreicht oder ein Fehler ist aufgetre-
ten.

    len:=FileLength(Namenstring)

Bestimmt die Länge eines Files, den Du vielleicht laden willst, und gibt
ebenso an, ob er existiert (gibt -1 zurück, wenn ein Fehler auftritt oder
der File nicht vorhanden ist.

    ok:=ReadStr(filehandle, estring)

siehe Stringunterstützung

    oldout:=StdOut(newstdout)

setzt die Standard-Output-Variabel 'stdout' auf den neuen Wert. Gleich mit:
oldout:=stdout; stdout:=newstdout
```

1.66 Eingebaute Funktionen

```
9B. Zeichenketten und Zeichenkettenfunktionen
---------------------------------------------
```

E hat einen Datentyp STRING. Dies ist ein String, von nun an 'EString' genannt, der in der Größe modifizier und verändert werden kann, als Gegenstück zu einem normalen 'String', welcher als durch ein Null-Byte beendete Zeichenfolge benutzt wird. EStrings sind abwärtskompatibel, aber nicht andersherum. Also wen ein Argument ein String erfordert, können beide benutzt werden. Wenn ein EString erforderlich ist, kann nur ein solcher benutz werden. Beispiele fü die Anwendung:

```
DEF s[80]:STRING    /*s ist ein EString mit einer maximalen Länge von 80
                      Zeichen*/
ReadStr(stdout,s)   /*ließt eine Eingabe von der Console*/
m:=Val(s,N)         /*holt eine Nummer von der Console*/
```

Bei allen Zeichenkettenfunktionen bei denen Strings größer werden können als ihr Maximum, werden Vorsichtsmaßnahmen getroffen.

```
DEF s[6]:STRING
StrAdd(s,'dieser String ist länger als 6 Zeichen',ALL)
```

s wird nur 'dieser' enthalten.
Ein String kann dynamisch aus dem Systemspeicher mit der Funktion String() angefordert werden (der Zeiger hierfür muß auf NIL geprüft werden)

```
    s:=String(maxlen)
```

DEF s[80] ist gleich mit DEF s und s:=String(10)

```
    bool:=StrCmp(string,string,len)
```

vergleicht zwei Zeichenketten, len ist die Anzahl der Bytes, die verglichen werden sollen, oder All, wenn die ganze Länge überprüft werden soll. Gibt True oder False zurück.

```
    StrCopy(estring,string,len)
```

kopiert den String in EString. Wenn len=ALL ist, wird alles kopiert.

```
    StrAdd(estring,string,len)
```

genauso wie StrCopy(), nur das der String am Ende angehängt wird.

```
    len:=StrLen(string)
```

berechnet die Länge eines Strings der durch ein Null-Byte abgeschlossen wurde

```
    len:=EstrLen(estring)
```

gibt die maximale Länge eines EStrings zurück.

```
    RightStr(estring,estring,n)
```

Füllt den ersten EString mit den letzten n Bytes des zweiten EStrings.

```
    MidStr(estring,string,pos,len)
```

kopiert jede beliebige Anzahl von Zeichen (alle eingeschloseen, wenn len=
All) von der Position pos im String in den EString.
WICHTIG: Bei allen stringverarbeitenden Funktionen ist zu Bedenken, daß das
erste Zeichen die Position 0 hat, und nicht 1, wie in normalen
Sprachen, wie Basic.

```
    wert:=Val(string,read)
```

Findet eine Dezimalzahl im ASCII-Code codiert aus einr Zeichenkette. Führende Leerzeichen/Tabulatoren werden übersprungen. Auch Hexadezimalzahlen (1234567890ABCDEFabcdef) und Binärzahlen (01) können so gelesen werden, wenn sie von einem "$"- oder "%"-Zeichen angeführt werden. Ein Minuszeichen kann eine negative Zahl anzeigen. Val() gibt die Anzahl der gelesenen Zeichen im zweiten Argument zurück, welches über Referenz (<-!!!) übergeben werden muß. Wenn read den Wert 0 (wert wird auch 0 sein) hat, dann enthält der String keine Dezimalzahl, oder der Wert ist zu groß um in 32 Bit aufgenommen zu werden. "read" kann NIL sein.

Beispiel für Zeichenketten die korrekt übersetzt werden:

'-12345', '%10101010', '-$ABcd12'

diese würden in "wert" und der Variabel [read] eine 0 zurückgeben:

'', 'Hallo'

```
    findepos:=IntStr(string1, string2, startpos)
```

sucht in string1 nach dem Vorkommen von string2. Es kann auch von einer anderen Position als 0 gestartet werden. Zurückgegeben wird die *Addresse*, an der der Unterstring gefunden wurde, ansonsten -1.

```
    neuestringadr:=TrimStr(string)
```

gibt die *Addresse* des ersten Zeichen in einem String, z.B. nach führenden Leerzeiche, Tabulatoren usw.

```
    UperStr(string)     und     LowerStr(string)
```

ändert die Groß- und Kleinschreibung einer Zeichenkette.
BEACHTE: diese Funktion verändert die Elemente einer Zeichenkette. Deshalb sollte man sie nur bei EStrings und Zeichenketten die ein Teil des Codes sind verwenden. Effektiv heißt das, daß wenn Du eine Stringaddresse vom Betriebssystem bekommen hast, mußt Du diesen erst mit StrCopy() in einen String Deines Programms kopieren, und dann erst mit diesem Programm benutzen.

```
    ok:=ReadStr(filehandle, EString)
```

ließt eine String (mit ASCII 10 endent) aus einem File oder stdout. ok enthält -1 wenn ein Fehler festgestellt oder das EOF erreicht wurde.
Bedenke: Die Elemente des Strings, die bis dahin gelesen wurden sind gültig.

```
    SetStr(EString,neuelänge)
```

Setzt manuelle die Länge eines Strings. Dies ist nur nützlich, wenn Du Daten in einen EString über nicht E-String-Funktionen ließ, und ihn als EString weiter benutzen willst. Z. B. nach der Benutzung einer Funktion die nur einen Null-Byte-Beendeten String auf die Addresse eines EStrings schreibt, benutze SetStr(meinstr,StrLen(meinstr)) um ihn wieder manipulierbar zu machen.
Für String-Zusammenführungsfunktionen siehe Kapitel 9H

1.67 Eingebaute Funktionen

```
9C. Listen und Listen Funktionen
--------------------------------
```

Listen sind wie Strings nur das sie sich aus LONGs und nicht aus CHARs zusammensetzen. Sie können auch entweder global, lokal oder dynamisch angefordert werden:

```
DEF meineliste[100]:LIST    /*lokal oder global*/
DEF a
a:=LIST(10)                 /*dynamisch*/
```

(beachte das im letzteren Fall der Zeiger a NIL enthalten kann). Genauso wie Strings als Konstanten in Ausdrücken dargestellt werden können, haben Listen ihr konstantes Equivalent:

```
[1,2,3,4]
```

Der Wert eines solchen Ausdrucks ist ein Zeiger auf eine fertig initalisierte Liste. Eine spezielle Eigenschaft ist, das sie dynamische Teile, z. B., welche während der Laufzeit gefüllt werden:

```
a:=3
[1,2,a,4]
```

außerdem können Listen auch einige andere Typen als die Voreinstellung LONG enthalten, wie:

```
[1,2,3]:INT
[65,66,67,0]:CHAR               /*gleich mit 'ABC' *
['topaz.font',8,0,0]:textattr
OpenScreenTagList(NIL,[SA_TITEL,'MeinScreen',TAG_DONE])
```

Wie im letzten Beispiel gezeigt, sind Listen extrem nützlich bei der Benutzung von Systemfunktionen: Sie sind abwärtskompatibel mit einem ARRAY OF LONG, und auf ein Objektzeigende können überall dort benutzt werden, wo eine Systemfunktion einen benutzt werden, wo eine Systemfunktion einen Zeiger auf einige Strukturen oder ein Feld von Zeigern benötigt. Taglist und vararg-Funktionen können auch auf diese Weise benutzt werden.
BEACHTE: Alle Funktionen arbeiten nur mit LONG Listen, typenzugeordnete Listen sind nur geeignet um komplexe Datenstrukturen und Ausdrücke aufzubauen.

Wie bei Strings gibt es eine klare Hierachie.

Listen Variabeln -> Konstante Liste -> Feld von Longs/Zeiger auf Longs

Wenn eine Funktion ein Feld von Longs braucht, kannst Du auch eine Liste als Argument angegeben. Wenn aber eine Funktion eine Listenvariabel oder eine konstante Liste braucht, reicht ein Feld von Longs nicht aus.

Es ist wichtig, daß Du die Mächtigkeit von Listen besonders typenorientierter verstehst: dis kann Dir viel Ärger beim Aufbau irgendeiner Datenstruktur ersparen. Versuche diese Listen in Deinen eigenen Programmen zu benutzen und schauen welche Funktion sie in den Beispielprogrammen haben. Wenn Du Listen einmal im Griff hast, möchtest Du niemals ein Programm ohne sie schreiben.

Zusammenfassung:

[<Element>, <Element>,...]	direkte Liste (von LONGs, Benutzung mit Listenfunktion
[<Element>, <Element>,...]:<Typ>	Typenorientierte Liste (nur um Datenstrukturen zu bauen)

Wenn <Typ> ein einfacher Typ wie INT oder CHAR ist, hast Du nur das eingerichtete Gegenstück zu ARRAY OF <Typ> ein Objektname ist erzeugst Du ein eingerichtetes Objekt oder ARRAY OF <Objekt>, sich auf die Länge der Liste stützend.

Wenn Du schreibst [1,2,3]:INT erzeugst Du eine Datenstruktur von 6 Bytes, von 3 16 Bit Werten um genau zu sein. Der Wert eines solchen Ausdrucks ist dann ein Zeiger zu einem Speicherbereich um genau zu sein. Es funktoniert z. B. mit einem Objekt.

```
OBJECT meinobjekt
    a:LONG, b:CHAR, c:INT
ENDOBJECT
```

wenn Du jetzt schreibst: [1,2,3]:meinobjekt würde dann eine Datenstruktur von 8 Bytes im Speicher erstellen. Diese ersten vier Byte sind eine LONG-Zahl mit dem Wert 1, das folgende Byte ist ein CHAR mit dem Wert 2, dann ein Füllbyte, um die Wort-Anordnung zu erhalten (16 Bit). Es ist trotzdem sehr wahrscheinlich, daß ein E-Compiler für eine 80x86 Architektur das Füllbyte nicht nutzen wird und eine 7 Byte Struktur erzeugen wird. Und ein E-Compiler für eine Sun-Sparc Architektur (wenn ich mich nicht vertue) wird versuchen alles auf 32 Bit Grenzen zu stzen, so macht er eine 10 oder 12 Byte Struktur. Einige Mikroprozessoren (sie sind selten aber es gibt sie) benutzen sogar (36:18:9) als Anzahl von Bits für ihre Typen (LONG:INT:CHAR) anstatt von (32:16:8) wie wir es gewöhnt sind. Mache Dir kein zu großen Vorstellungen der Struktur eines OBJECTS oder einer LISTe, wenn Du Code schreiben willst, der eine Chance haben soll portabel zu sein oder sich auf Seiteneffekte zu verlassen.

    ListCopy(Listenvar,Liste,Anzahl)

Kopiert Anzahl Elemente aus Liste in die Listenvar. Beispiel:

```
DEF meineliste[10]:LIST
ListCopy(meineliste,[1,2,3,4,5],ALL)
```

    ListAdd(Listenvariabel, Liste, Anzahl)

```
Kopiert Anzahl Elemente von Liste an die erste nicht belegte Stelle von
Listenvariabel

    ListCmp(Liste, Liste, Anzahl)

Vergleicht zwei Listen, oder Anzahl Teile von ihnen.

    länge:=ListLen(Liste)

Gibt die Länge der Liste zurück, wie ListLen([a,b,c]) 3 zurückgeben würde.

    maximum:=ListMax(Listenvariabel)

Gibt die maximal mögliche Listenlänge von Listenvariabel zurück.

    wert:=ListItem(Liste,Index)

funktioniert wie wert:=Liste[Index] mit dem Unterschied, daß Liste ein kon-
stanter Wert anstatt eines Zeiger sein kann. Dies kann in Situationen sehr
nützlich sein, in denen wir direkt ein Liste von Werten benutzen wollen:

WriteF(ListItem(['ok','kein Speicher','keine Datei'],fehler))

Es verhält sich wie:

dummy:=['ok','kein Speicher','keine Datei']
WriteF(dummy[fehler])

    SetList(Listenvariabel, neuelänge)

setzt manuell die Länge einer Liste. Dies is nur nützlich, wenn Du mit nicht
listenspezifischen Funktionen Daten in eine Liste schreibst, sie aber als
richtige Liste weiterverwenden willst.

Für Listenfunktion, die ausgewertete Ausdrücke benutzen siehe Kapitel 11C.
Für Listenverbindungsfunktionen siehe Kapitel 9H
```

1.68 Eingebaute Funktionen

```
9D. Intuition unterstützende Funktionen
---------------------------------------
    wptr:=OpenW(x,y,Breite,Höhe,IDCMP,WFlags,Titel,Screen,SFlags,Gadgets)

erzeugt ein Fenster, wobei WFlags die Flags für das Windowlayout sind (wie
BACKDROP, SIMPLEREFRESH usw, normalerweise $F) und SFlags ist da um die
Screenart zu bestimmen auf der das Fenster geöffnet werden soll (1=WB,
15=Custom). Screen muß nur gültig sein, wenn SFlags=15, ansonsten reicht
NIL. Gadgets kann auf eine GList-Struktur zeigen, welche einfach mit Gad-
get() erzeugt werden kann, ansonsten NIL.

    CloseW(wptr)

schließt das Fenster wieder. Der einzige Unterschied zu CloseWindow ist
```

das es NIL-Zeiger akzeptiert und den stdrast wieder auf NIL setzt.

```
    sptr:=OpenS(Weite,Höhe,Tiefe,SFlags,Titel)
```

Öffnet einen Customscreen. Tiefe ist die Anzahl der Bitplanes (1-6, 1-8 AGA), SFlags ist etwas wie 0, oder $8000 für Hires (addiere 4 für Interlace)

```
    CloseS(sptr)
```

wie CloseW(), nun für Screens.

```
    nächsterbuffer:=Gadget(buffer,glist,id,flags,x,y,Breite,string)
```

Diese Funktion erstellt eine Liste von Gadgets, welche in Deinem Fenster angezeigt werden können, in dem Du sie als Argument bei OpenW() als Argument übergibst. Später kannst Du das auch mit der Intuitionfunktion AddGlist() erledigen.
Bufer ist meistens ein ARRAY on mindestens der GrößeGADGETSIZE Bytes um alle Strukturen die zu einem Gadget gehören aufzunehmen. ID ist eine beliebige Nummer die Dir helfen soll später zu erkennen welches Gadget gedrückt wurde, wenn ine IntuiMessage ankommt. Flags sind 0=normal, 1=Boolean Gadget, 3= Boolean Gadget das ausgewählt ist. Width ist die Weite in Pixeln, die groß genug sein sollte um den String aufzunehmen, welcher automatisch zentriert wird. Glist sollte beim ersten Gadget NIL sein, und glistvar für alle anderen, so kann E alle Gadget zusammenlinken. Die Funktion gibt einen Zeiger auf den nächsten Buffer zurück (=buffer+GADGETSIZE). Beispiel für 3 Gadgets:

```
CONST   MAXGADGETS=GADGETSIZE*3

DEF buf[MAXGADGETS]:ARRAY, next, wptr

next:=Gadget(buf,NIL,1,0,10,20,80,'bla' /*das erste Gadget*/
next:=Gadget(next,buf...)
next:=Gadget(next,buf...)    /*jede Anzahl kann zum ersten gelinkt werden*/

wptr:=OpenW(...,buf)
```

Schaue Dir die richtigen Beispiele wie SuperVisor.e für richtige Anwendugen an.

```
    code:=Mouse()
```

gibt Dir den momentanen Status von allen 2 oder 3 Mausknöpfen zurück, links =1, rechts=2 und mitte=4. Wenn der Code z.B. gleich 3 ist, sind die linke und die rechte Maustaste gedrückt.
WICHTIG: die ist keine richtige Inituition-Funktion, wenn Du auf regulären Weg etwas über die Maus-Events erfahren willst, muß Du die IntuiMessages kontrollieren, die an Deinem Window ankommen. Das ist die einezige E-Funktion die direkt auf die Hardware zugreift, und ist deshalb nur für demoartige Programme nützlich:

```
    x:=MouseX(win)       und     y:=MouseY(win)
```

ermöglicht es Dir die Mauskoordinaten zu lesen. Win ist das Window zu dem sie relativ sind.

```
    class:=WaitIMessage(window)
```

Diese Funktion macht es einfacher auf ein Window-Event zu warten. Es speichert andere Variabeln wie Cod und Qualifiers als private, globale Variabeln, für den Zugriff mit Funktionen, die unten beschrieben werden. WaitIMessage() representiert folgenden Code:

```
PROC waitimessage(win:PTR TO window)
  DEF port,mes:PTR TO intuimessage,class,code,qual,iaddr
  port:=win.userport
  IF (mes:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (mes:=GetMsg(port))<>NIL
  ENDIF
  class:=mes.class
  code:=mes.code             /* intern abgespeichert */
  qual:=mes.qualifier
  iaddr:=mes.iaddress
  ReplyMsg(mes)
ENDPROC class
```

wie Du siehst, holt es exakt eine Nachricht, und vergißt keinen mehrfach Message, die bei einem Ereigniss ankommt. Wenn mehr als ein Aufruf erfolgt. Zum Beispiel, sagen wir Du hast ein Window geöffnet, das etwas anzeigt, und nur auf das Cosegadget wartet (Du hast nur IDCMP_CLOSEWINDOW angegeben):

```
    WaitMessage(meinwindow)
```

oder Du hast ein Programm das auf mehrere Arten von Events wartet, sie in einer Schleife bearbeitet, un mit einem Closewindow-Event endet:

```
WHILE (class:=WaitIMessag(win))<>IDCMP_CLOSWINDOW
    /*Bearbeitung der anderen Klassen */
ENDWHILE

    code:=MsgCode()     qual:=MsgQualifier()    iaddr:=MsgIaddr()
```

Dies alles versorgt Dich mit den privaten globalen Variabel, die vorher erwähnt wurden. Die Werte die zurückgegeben werden, wurden alle vom letzten Aufruf von WaitIMessage() definiert. Beispiel:

```
IF class:=IDCMP_GADGETUP
    mygadget:=MsgIaddr()
    If mygadget.userdata=1 THEN     /* der Anwender hat Gadget #1 gedrückt*/
ENDIF
```

1.69 Eingebaute Funktionen

```
9E. Grafikfunktionen
--------------------
```

Alle grafikunterstützenden Funktionen, die nicht extra einen Rastport verlangen, benutzen die System-Variabel 'stdrast'. Sie wird automatisch vom letzten Aufruf von OpenW() oder OpenS() definiert, und wird von CloseW() und CloseS() auf NIL gesetzt. Der Aufruf dieser Routinen, während 'stdrast' NIL ist, ist erlaubt. stdrast kann manuelle über SetStdRast() oder stdrast:=

meinrast geändert werden.

```
    Plot(x,y,Farbe)
```

Malt einen einzelnen Punkt auf Deinem Screen/Window in einer der verfügbaren Farben. Die Farbe geht von 0-255, oder 0-31 auf vor-AGA Maschinen.

```
    Line(x1,y1,x2,y2,Farbe)
```

Malt eine Linie.

```
    Box(x1,y1,x2,y2,Farbe)
```

Malt ein Rechteck.

```
    Colour(Vordergrund, Hintergrund)
```

setzt die Farbe für alle Grafikfunktionen (aus dem Library), die kein Farbargumente annehmen. Dies ist das *Farbregister* (z.B. 0-31) und nicht der *Farbwert*.
BEACHTE: Funktionen, die ein "Farbe" als Argument haben, verändern den Apen des stdrast.

```
    TextF(x,y,Formatstring,args,....)
```

hat genau dieselbe Funktion wie WriteF(), nur an irgendeiner (x,y) Position in deinem stdrast, anstatt von stdout. Siehe auch WriteF() und Strings in der Sprachbeschreibung.

```
    alterrast:=SetStdRast(neuerrast)
```

verändert den Ausgaberastprot der E-Grafik-Funktionen.

```
    SetTopaz(größe)
```

setzt den Font des Rastport "stdras" auf Topaz, nur um sicher zu sein, daß einige Custom-Fonts des Anwenders nicht unser Bildschirmlayout durcheinander werfen. Größe ist natürlich 8 oder 9.

1.70 Eingebaute Funktionen

```
9F. Systemfunktionen
--------------------
```

```
    bool:=KickVersion(vers)
```

Gibt TRUE zurück, wenn die Kickstart in der Maschine, auf der Dein Programm läuft, gleich oder größer ist, ansonsten FALSE.

```
    mem:=New(n)
```

Dies erzeugt dynamisch ein Feld (oder Speicherbereich, wenn Du möchtest) von n Bytes. Unterschiede zu AllocMem() sind, daß es automatisch die Flaggs $10000 (also gelöschter Speicher, jeder Typ) angibt und keine Dispose() Aufrufe notwendig sind, da es an eine Speicherliste angefügt wird, die auto-

matisch am Ende des Programms freigegeben wird.

```
    Dispose(mem)
```

Gibt jedes mem frei, das durch New() angefordert wurde. Du muß diese Funktion nur benutzen, wenn Speicher während des Programmablaufs freigegeben werden soll, da es am Ende sowieso freigegeben wird.

```
    CleanUp(Rückgabewert)
```

Beendet das Programm von jeden Punkt. Es ist der Ersatzt für den DOS-Aufruf Exit(): benutze diesen niemals!!!!!! anstatt von CleanUp(), welcher es ermöglicht Speicher freizugeben, Libraries richtig zu schließen usw. Der Rückgabewert wird als Returncode an DOS zurückgegeben.

```
    menge:=FreeStack
```

gibt die Menge des freien Stackspeichers zurück. Diese sollte immer 1000 oder mehr betragen. Siehe in Kapitel 16 wie E seinen Stack organisiert. Wenn Du nicht im Rekursionswahn bist, brauchst Du Dir über den freien Stack-Speicher keine Sorgen zu machen.

```
    bool:=CtrlC()
```

Gibt TRUE zurück, wenn Ctrl-C nach der letzten Überprüfung gedrückt wurde, ansonsten FALSE. Dies arbeitet nur mit Programmen, die über die Console laufen, d.h. CLI-Programme.

```
/*berechnet die Fakultät des CLI-Arguments*/

OPT STACK=100000

PROC main()
  DEF num,r
  num:=Val(arg,{r})
  IF r=0 THEN WriteF('Argumentfehler.\n') ELSE WriteF('Wert: \d\n',fac(num))
ENDPROC

PROC fac(n)
  DEF r
  IF FreeStack()<1000 OR CtrlC() THEN CleanUp(5)    /* Extra Controlle */
  IF n=1 THEN r:=1 ELSE r:=fac(n-1)*n
ENDPROC r
```

Natürlich wird diese Rekursion kaum den Stack zum überlaufen bringen, und wenn dies passiert, wird es so schnell von FreeStack() angehalten, das Du keine Zeit hast Ctrl-C zu drücken, aber es ist die Idee die hier zählt. Eine Definition von fac(n) wie:

```
Proc fac(n) RETURN IF n=1 THEN 1 ELSE fac(n-1)*n
```

wäre nicht so sicher.

1.71 Eingebaute Funktionen

9G. Mathematische Funktionen

```
    a:=And(b,c)     a:=Or(b,c)      a:=Not(b)
    a:=Eor(b,c)
```

Diese arbeiten mit den normalen Operatoren, boolschen genauso wie aritmethische. Beachte, daß für And() und Or() Operatoren existieren:

```
    a:=Mul(b,c)     a:=Div(a,b)
```

Macht dasselbe wie die '*' und '/' Operatoren, aber jetzt mit vollen 32 Bit. Aus Geschwindigkeitsgründen sind normale Operationen 16 Bit * 16 Bit = 32 bit und 32 Bit/ 16 Bit = 16 Bit. Dies reicht für fast alle Rechnungen aus und wo nicht, kannst Du Mul() und Div() benutzen. BEACHTE: im Fall von Div() wird a durch b geteilt und nicht umgekehrt.

```
    bool:=Odd(x)        bool:=Even(x)
```

Gibt TRUE oder FALSE zurück wenn ein Ausdruck gerade oder ungerade ist.

```
    randnum:=Rnd(max)   seed:=RndQ(seed)
```

Rnd() berechnet eine Zufallszahl aus einer Internen seed im Raum von 0.. max-1. Zum Beispiel, Rnd(1000) gibt eine Dezimalzahl von 0..999 zurück. Um die Interne Quelle zu Initalisieren, rufe Rnd() mit einem negativen Wert auf. Der Abs() dieses Werts wird dann als Ursprungs'seed' genommen. RndQ() berechnet eine Zufallszahl schneller als Rnd(), aber gibt nur ganze 32 Bit Zufallszahlen zurück. Benutze das Ergebniss als seed für den nächsten Aufruf, und als Anfangs'seed' benutze einen großen Wert wie $AGF87EC1

```
    abswert:=Abs(wert)
```

berechnet den absoluten Wert.

```
    a:=Mod(b,c)
```

Dividiert 32 Bit b durch 16 Bit c und gibt den 16 Bit Modulo a zurück.

```
    x:=Shl(y,num)       x:=Shr(y,num)
```

Schiebt y um num Bits nach links oder nach rechts.

```
    a:=Long(adr)        a:=Int(adr)     a:=Char(adr)
```

Ließt aus einer Speicheraddresse einen Wert und gibt ihn zurück. Dies geht mit 32, 16 und 8 Bit Werten in dieser Reihenfolge. Der Compiler überprüft nicht ob die Addresse gültig ist. Diese Funktionen sind in E verfügbar für den Fall, daß das Lesen und Schreiben im Speicher mit PTR TO <Typ> das Programm nur noch komplexer und uneffizienter machen würde. Du solltest aber nicht dazu ermutigt werden, diese Funktionen zu benutzen.

```
    PutLong(adr,a)      PutInt(adr,a)       und     PutChar(adr,a)
```

Poket (schreibt) den Wert 'a' in den Speicher, Siehe auch Long()

1.72 Eingebaute Funktionen

```
9H. Funktionen zum Verbinden von Zeichenketten und Listen
---------------------------------------------------------
```

E ist mit einer Reihe von Funktionen ausgestattet, die die Erstellung von verketteten Listen mit dem STRING und LIST Datentyp, oder Strings und Listen, die mit String() und List() erstellt wurden, erlaubt. Wie Du vielleicht weißt, sind Listen, Strings, komplexe Datentypen Zeiger auf ihre verschiedenen Daten, und haben ein extra Feld an einem negativen Offset dieses Zeigers die ihre aktuelle und ihre maximale Länge enthält. Die Offsets dieses Feld sind PRIVATE. Als Zusatzt zu diesen beiden, hat jeder komplexe Datentyp ein 'next' Feld, welches Defaultmäßig auf NIL gesetzt ist. Dieses kann benutzt werden, um verkettete Listen zu erzeugen, z.B. von Strings. Ab jetzt verstehe ich unter komplex einen ptr auf einen STRING oder eine LISTe, und unter 'tail' noch einen solchen Zeiger, oder einen der schon einen solchen String anghängt hat. 'tail' kann auch ein NIL Zeiger sein, der das Ende einer verketteten Liste anzeigt. Die folgenden Funktionen können benutzt werden.

```
    komplex:=Link(komplex,tail)
```

schreibt den Wert von tail in das 'next'-Feld von komplex. Beispiel:

```
DEF s[10]:STRING, t[10]:STRING
Link(s,t)
```

erzeugt eine verkettete Liste wie: s-->t-->NIL

```
    tail:=Next(komplex)
```

ließt das 'next' Feld einer komplexen Variabel. Dies kann natürlich NIL sein, oder eine komplett verkettete Liste. Next(NIL) aufrufen ergibt NIL, so ist es sicher Next aufzurufen, wnn man sich nicht sicher ist ob man am Ende einer verketteten Liste ist.

```
    tail:=Forward(c,1)
```

genau dasselbe, geht nur um num Links vorwärts, anstatt von einem, also:

```
Next(c)=Forward(c,1)
```

Du kannst Forward sicher mit Nummern aufrufen, die zu weit gehen; Forward hält an sobald es ein NIL beim suchen eines Links entdeckt und gibt NIL zurück.

```
    DisposeLink(complex)
```

dasselbe wie Dispose(), mit dem Unterschied: es ist nur für Strings und Listen die mit String() und List() angefordert wurden, und entfernt automatisch den 'tail' eines komplexen Datentyps. Beachte, daß in große verketteten Listen, die Strings enthalten, die sowohl mit String() als auch einige lokal und global mit STRING allociert wurden können auch auf diese Art freigegeben werden.

Für ein gutes Beispiel, wie man Listen von Strings gut im wirklichen Leben

gebrauchen kann, siehe 'D.e'.

1.73 Library Funktionen und Module

10.LIBRARY FUNKTIONEN UND MODULE

A. Eingebaute Library Aufrufe

B. Schnittstellen zum Amiga Sytem mit den 2.04 Modulen bilden

Vorheriges Kapitel

Nächstes Kapitel

1.74 Library Funktionen und Module

10A. Eingebaute Library Aufrufe

Wie Du vielleicht schon aus den vorherigen Abschnitten weißt, wird vor Dein Programm automatisch um ein Programmteil (der "initialisation code") ergänzt, der beim Programmstart immer folgende vier Bibliotheken öffnet: Intuition, Dos, Graphics und Mathffp. Daher sind die Funktionsaufrufe zu diesen fünf Bibliotheken (Exec eingeschlossen) im Compiler integriert (es sind einige Hundert). Jedenfalls bis zu AmigaDos v2.04, v3.00 sollte bis zur nächsten Version von Amiga E implementiert sein. Um Open() von der dos.library aufzurufen, genügt ein schlichtes:

```
handle:=Open('meinedatei',OLDFILE)
```

oder AddDisplayInfo() von graphics.library:

```
AddDisplayInfo(meindispinfo)
```

So einfach ist das.

1.75 Library Funktionen und Module

10B. Schnittstellen zum Amiga Sytem mit den 2.04 Modulen bilden
---

Um eine beliebige andere Bibliothek als die fünf im vorherigen Abschnitt genannten zu benutzen, mußt Du Module wählen. Du brauchst auch Module, wenn Du - wie in C oder Assembler üblich - OBJECT oder CONST Definition aus den Amiga-Includes benutzt. Module sind Binärdateien, die Definitionen von Konstanten, Objekten, Bibliotheken und Funktionen (code) beinhalten können. Die Tatsache, daß sie binär vorliegen, hat den ASCII-Dateien (benutzt in C und Assembler) gegenüber den Vorteil, daß sie nicht jedesmal neu compiliert werden müßen, wenn Dein Programm neu compiliert wird. Der Nachteil ist, daß man sie sich nicht einfach anschauen kann; um ihren Inhalt sichtbar zu machen, ist ein Utility wie ShowModule (siehe utility.doc) nötig. Die

Module, die die Bibliotheksdefinitionen (d.h. deren Aufrufe) enthalten, stehen im Wurzelverzeichnis von emodules: (dem Modul-Verzeichnis in der Distribution), die Definitionen der Konstanten/Objekte befinden sich in den Unterverzeichnissen, sie sind wie die Originale von Commodore aufgebaut.

MODULE

Sytax: MODULE <Modulname>,...
Lädt ein Modul. Ein Modul ist eine Binärdatei, die Informationen über Bibliotheken, Konstanten und manchmal auch Funktionen enthält. Durch Modulbenutzung ist es Dir möglich, Bibliotheken und Funktionen zu benutzen, die dem Compiler vorher nicht bekannt waren.

Nun zu einem Beispiel, unten steht ein kleine Version des Sources sources/examples/asldemo.e, das Module verwendet, um einen Filerequester der 2.0 Asl.library darzustellen.

```
MODULE 'Asl', 'libraries/Asl'

PROC main()
  DEF req:PTR TO filerequestr
  IF aslbase:=OpenLibrary('asl.library',37)
    IF req:=AllocFileRequest()
      IF RequestFile(req) THEN WriteF('File: "\s" in "\s"\n',req.file,req.dir)
      FreeFileRequest(req)
    ENDIF
    CloseLibrary(aslbase)
  ENDIF
ENDPROC
```

Aus dem Modul 'asl' erfährt der Compiler die Definitionen der asl-Funktionen, wie zum Beispiel RequestFile(), und die globale Variable 'aslbase', die vom Programmierer lediglich initialisiert werden muß. Aus 'libraries/asl' erfährt er die Definition des Objektes filerequestr, das wir benutzen, um zu erfahren, welche Datei der Anwender ausgewählt hat. Das war doch nun wirklich nicht schwer: hast Du gedacht, daß es so einfach wäre, einen Filerequester in E zu programmieren?

1.76 Ausgewertete Ausdrücke

11.AUSGEWERTETE AUSDRüCKE

A. Auswertung und Bereich

B. Eval()

C. Eingebaute Funktionen

Vorheriges Kapitel

Nächstes Kapitel

1.77 Ausgewertete Ausdrücke

```
11A. Auswertung und Bereich
---------------------------
```
Quotierte Ausdrücke beginnen mit dem Hochkomma. Der Wert eines ausgewerteten Ausdrucks ist nicht das Ergebniss von einer Berechnung eines Ausdrucks, sondern die Adddresse des Codes. Dieses Ergebniss kann als eine normale Variabel weiterverwendet werden, oder als ein Argument für bestimmte Funktionen.

```
meinefunk:=‘x*x*x
```

meinefunk ist nun ein Zeiger auf eine Funktion die x^3 berechnet, wenn sie berechnet wird. Diese Zeiger auf Funktionen sind sehr unterschiedlich zu normalen PROCs, und Du solltest die beiden nie durcheinander bringen. Die größten Unterschiede sinde, daß quotierte Ausdrücke nur einfache Ausdrücke sind, und deshalb keine eigenen lokalen Variabeln haben kann. In unserem Beispiel ist "x" nur eine lokale oder globale Variabel. Das ist warum vorsicht sein muß: Wenn meinefunk irgendwo später gleichen PROC auswertest, kann x lokal sein, aber wenn meinefunk als Parameter an einen anderen PROC übergibst, und dann auswertest, muß x natürlich global sein. Es gibt keine Bereichsprüfung hierbei.

1.78 Ausgewertete Ausdrücke

```
11B. Eval()
-----------
    Eval(funk)
```

wertet einfach einen quotierten Ausdruck (exp=Eval(’exp)) aus.
BEDENKE: Weil E eine etwas typenlose Sprache ist, wird vom Compiler dummerweise nicht bemerkt, wenn wir "Eval(x*x)" anstatt von "Eval(‘x*x)" schreiben, und das macht Dir große Laufzeitprobleme: der Wert von x*x wird als Zeiger auf Code benutzt.
Um zu verstehen warum "quotierte Ausdrücke" so mächtig sind, danke an die folgenden Fälle: wenn Du eine Reihe von Aktionen mit einer Reihe von Variabeln abarbeiten mußt, schreibst Due eine Funktion und rufst die Funktion mit verschiedenen Argumenten auf. Aber was ist, wenn das Argument ein Teil des Codes ist. In traditionellen Programmiersprachen wäre dies nicht möglich, so müßtest Du die Blöcke die Deine Funktion repräsentieren "kopieren", und dann den Ausdruck hineinschreiben. Nicht in E. Sagen wir, Du möchtest ein Programm schreiben, das die Arbeitungszeit von verschiedenen Ausdrücken vergleicht. In E würdest Du einfach nur schreiben:

```
PROC timing(funk,titel)

    /*macht alle Dinge um die Zeit zu stellen+/

    Eval(funk)

    /*und den Rest*/
```

```
    Write('Die gemessene Zeit von \s war \d\n',titel,t)

ENDPROC
```

und rufen es auf mit:

```
timing ('x*x*x,'Multiplication')
timing ('großeberechnung(),'Große Rechnung')
```

in jeder anderen Befehlssprache, müßtest Du für jeden timing-Aufruf eine Kopie schreiben, oder Du müßtest jeden Ausdruck in eine seperate Funktion schreiben. Dies ist nur ein einfaches Beispiel: denke daran, was Du mit Datenstrukturen (LISTs) mit unausgewerteten Code machen kannst.

```
malfunks:   ['Plot(x,y,c), 'Line(x,y,x+10,y+10,c), 'Box(x,y,x+20,y+20,c)]
```

Die Idee von Funktionen als normale Variabeln/Werten ist keine Neuheit von E. Quotierte Ausdrücke wurden von LISP beschrieben, welche auch noch etwas mächtigeres, Lambda Funktionen genannt, hat, was auch als Argument an Funktionen übergeben wird. E's quotierte Ausdrücke können auch als parameterlose (oder nur globale Parameter) Lambdas angesehen werden.

1.79 Ausgewertete Ausdrücke

```
11C. Eingebaute Funktionen
--------------------------

    MapList(variabeladr,liste,listenvar,funk)
```

unterstützt einige Funktionen auf alle Elemente von liste und gibt alle Ergebnisse in listenvar zurück. funk muß ein quotierter Ausdruck sein (siehe oben) und variabel (im welchen Bereich der liste) muß als Referenz übergeben werden.

```
MapList([x],[1,2,3,4,5],r,'x*x) ergibt in r:[1,4,9,16,25]

    ForAll(variabeladr,liste,funk)
```

Gibt TRUE zurück, wenn alle Werte in liste die Funktion (quotierter Ausdruck) zu TRUE verarbeiten, ansonsten FALSE. Kann auch benutzt werden, um eine bestimmte Funktion auf alle Elemente einer Liste abzuarbeiten.

```
ForAll([x],['eins','zwei','drei'],'WriteF('Beispiel: \s\n',x)

    Exists(variabeladr,liste,funk)
```

Wie ForAll(), nur diese gibt TRUE zurück, wenn irgend ein Element TRUE(<>) ergibt. Beachte, daß ForAll() immer alle Elemente berechnet, aber Exists() möglicherweise nicht.

Beispiele, wie man diese Funktionen auf eine spezielle Art und Weise benutzt:
wir allozieren verschiedene Größen von Speicher in einem Statement, überprüfen Sie alle auf einmal und geben nur die frei, die erfolgreich angefor-

```
dert wurden. (Beispiel für v37+)

PROC main()
  LOCAL mem[4]:LIST,x
  MapList({x},[200,80,10,2500],mem,'AllocVec(x,0)) /* einige allozieren */
  IF ForAll({x},mem,'x)                            /* Erfolg ? */
    WriteF('Yes!\n')
  ELSE
    WriteF('No!\n')
  ENDIF
  ForAll({x},mem,'IF x THEN FreeVec(x) ELSE NOP)   /* nur die <>NIL frei-
                                                      geben*/
ENDPROC

Beachte das fehlen von Wiederholungen in diesem Code. Versuche doch einfach
dieses Beispiel in irgendeiner anderen Programmiersprache zu schreiben, und
siehe warum dies besonders ist.
```

1.80 Fließkommaunterstützung

```
                        12.FLIEßKOMMAUNTERSTÜTZUNG


                A. Gebrauch/Überladen von Fließkommazahlen/-operatoren

                B. Fließkommaausdrüke und Umwandlungen

                Vorheriges Kapitel

                Nächstes Kapitel
```

1.81 Fließkommaunterstützung

```
12A. Gebrauch/Überladen von Fließkommazahlen/-operatoren
-------------------------------------------------------
Das überladen der standard Operatoren + * usw mit den fließkomma Gegen-
stücken ist seit der E Version 2.0 möglich, aber ich habe die Hauptdokumen-
tation davon entfernt, da wahrscheinlich das Fließkommakonzept ab der Ver-
sion v2.2 oder später sich ändern wird: diese Version wird dann 68881 In-
line-Code neben den normalen FFP-Routinen in einer transparenten Art er-
lauben.
Wenn Du wirklich Fließkommazahlen benutzen willst, mußt Du die eingebauten
SpXxx()-Routinen des mathffp.library benutzen.

Beispiel

x:=SpMul(y,0.013483)

Sei Dir bewußt, daß wenn v2.5 rauskommt, Dein Code vielleicht geändert wer-
den muß. (Für die besseren)
```

1.82 Fließkommaunterstützung

```
12B. Fließkommaausdrüke und Umwandlungen
----------------------------------------
wie 12A.
```

1.83 Exception Behandlung

```
                    13.EXCEPTION BEHANDLUNG


            A. Definition von Exceptionhandlern (HANDLE/EXCEPT)

            B. Benutzung der Raise() Funktion

            C. Exceptions für eingebaute Funktionen (RAISE/IF)

            D. Benutzung von Exception-ID's

            Vorheriges Kapitel

            Nächstes Kapitel
```

1.84 Exception Behandlung

```
13A. Definition von Exceptionhandlern (HANDLE/EXCEPT)
-----------------------------------------------------
Der Ausnahme Mechanismus in E ist hauptsächlich der gleiche wie in ADA;
es steht für flexible Reaktionen auf Fehler in deinem Programm und
komplexe Ressourcen Leitung. Beachte: der Ausdruck 'exeption' in E hat
sehr wenig zu tun mit Ausnahmen ("GURUS"), die vom 680x0 Prozessor verursacht ↩
   werden!

Ein Exeption-Handler ist ein Stück des Programm-Codes, daß aufgerufen wird,
wenn Laufzeitfehler geschehen, solche wie erfolgloses Öffnen von Fenstern
oder Speicher, der nicht mehr verfügbar ist. Du, oder das Laufzeit-System
selber, dürfen Signalisieren, daß etwas falsch ist (diese wird "Reaktion
auf einen Ausnahmezustand genannt), und dann wird das Laufzeit-System
versuchen, den zutreffenden Ausnahme Handler zu finden.

Ich sage "zutreffend", weil ein Programm mehr als einen Ausnahme Handler
enthalten kann, auf allen Stufen eines Programmes.

Eine normale Funktionen-Definition darf (wie wir alle wissen) folgendermaßen
ausschauen:


PROC bla()

  /* ... */

ENDPROC
```

Eine Funktion mit einem Ausnahme Handler sieht wie diese aus:

```
PROC bla() HANDLE

  /* ... */

EXCEPT

  /* ... */

ENDPROC
```

Der Block zwischen PROC und EXCEPT wird normal ausgeführt, und wenn keine Ausnahme passiert ist, wird der Block zwischen EXCEPT und ENDPROC übersprungen, und die Prozedur wird bei ENDPROC verlassen.

Wenn eine Ausnahme passiert, entweder im PROC-Abschnitt oder in irgendeiner Funktion, die in dem Block aufgerufen wurde, dann wird der Ausnahme-Handler ausgelöst.

1.85 Exception Behandlung

```
13B. Benutzung der Raise() Funktion
-----------------------------------
```

Es gibt viele Wege um ein Ausnahme-Situation auszulösen, der einfachste ist der über die Funktion Raise():

```
        Raise(exceptionID)
```

Die exeptionID ist einfach eine Konstante die den Typ der Ausnahme definiert und wird benutzt von Ausnahme-Handlern, um zu untersuchen, was schief gegangen ist.

Beispiel:

```
ENUM NOMEM,NOFILE  /* und andere */


PROC bla() HANDLE

  DEF mem

  IF (mem:=New(10))=NIL THEN Raise(NOMEM)
  myfunc()

EXCEPT
```

```
  SELECT exception

    CASE NOMEM

      WriteF('Kein Speicher!\n')

    /* ... und anderes */

  ENDSELECT

ENDPROC


PROC myfunc()

  DEF mem

  IF (mem:=New(10))=NIL THEN Raise(NOMEM)

ENDPROC
```

Die "Ausnahme"-Variable im Handler beinhaltet immer den Wert des Arguments, das durch den Aufruf der Raise()-Funktion übergeben worden ist. In beiden New()-Fällen übergab die Raise()-Funktion dem Handler der Funktion bla(), und dann ging es richtig zurück zum Aufrufer von bla().

Wenn myfunc() einen eigenen Ausnahme-Handler hätte würde dieser aufegrufen werden für den New()-Funktionsaufruf in myfunc(). Der Umfang eines Ausnahme-Handler ist vom Start der PROC, in welcher er definiert wurde, bis zum EXCEPT-Schlüsselwort, einschliesslich alle Aufrufe, die von hier gemacht werden.

Dieses hat drei Konsequenzen:

A. Handler sind rekursiv organisiert, und welcher Handler eigentlich aufgerufen wird ist abhängig von dem Fuknktionsaufruf bei Programmausführung;

B. wenn eine Ausnahme in einem Handler ausgelöst wird, dann wird der Handler einer niedrigeren Stufe ausgeführt. Dieses Verhalten der Handler darf benutzt werden, um komplex zusammengesetzte rekursive Zuteilungsarien mit großartig Bequemlichkeit zu benutzen, wie wir in Kürze sehen werden.

C. Wenn eine Ausnahme ausgelöst wird auf einer Stufe, in der kein niedriger "Stufen"-Handler verfügbar ist (oder in einem Programm, daß keinen Handler bekommen hat), dann wird das Programm abgebrochen

   Z.B.: Raise(x) hat den gleichen Effekt wie CleanUp(0)

1.86 Exception Behandlung

13C. Benutzung der Raise() Funktion (RAISE/IF)

Mit Ausnahmen, wie zuvor beschrieben, haben wir etwas Bedeutendes erreichet
Über den alten Weg der Definition unserer eigenen "Fehler()"-Funktionen,
aber dennoch gibt es eine Menge einzugeben, um NIL bei jedem Aufruf von
New() zu prüfen.

Das E-Ausnahme-Laufzeitsystem erlaubt Definition von Ausnahmen
für alle E Funktionen (wie New(), OpenW() usw..), und für alle Library
Funktionen (OpenLibrary(), AllocMem() usw..), sogar für eingebundene Module.

Syntax:

```
RAISE <exceptionId> IF <Funktion> <Vergleich> <Wert> , ...
```

der Teil nach RAISE darf wiederholt werden mit einem ",".

Beispiel:

```
RAISE NOMEM IF New()=NIL,

      NOLIBRARY IF OpenLibrary()=NIL
```

die ersten Zeilen sagen etwas wie "immer wenn ein Aufruf von New()
in NIL resultiert, dann rufe automatisch die NOMEM Ausnahme auf".

<Vergleich> kann irgendetwas wie = <> > < >= <= sein.

Nach dieser Definition dürfen wir alles über unser Programm schreiben:

```
mem:=New(size)
```

ohne zu schreiben:

```
IF mem=NIL THEN Raise(NOMEM)
```

Beachte, daß der einzige Unterschied ist, daß "mem" nie einen Wert bekommt,
wenn das Laufzeit-System den Handler aufruft: Code wird erzeugt für
jeden aufruf zu New() um nach der Rückkehr von New() zu prüfen
und evtl. Raise() aufzurufen, wenn dieses notwendig ist.

Wir haben jetzt ein kleines Beispiel, daß komplex genug ist, um ohne
Ausnahme-Handling auszukommen: wir rufen eine Funktion rekursiv auf und
in jedem Aufruf teilen wir eine Ressource zu (in diese Fall Speicher),
welchen wir vorher allokiert haben, und führen danach den rekursiv Aufruf
aus.

Was geschieht, wenn irgendwo oben in der Rekursion ein Fehler entsteht und wir das Programm zu verlassen haben?

Richtig: wir würden (in einer konventionellen Sprache) nicht die niedrigeren Resourcen freibekommen, während wir das Programm verlassen, weil alle Zeiger zu diesem Speicher in unerreichbaren lokalen Variablen hinterlegt sind!

In E erhöhen wir einfach eine Ausnahme und von dem Ende des Handlers erhöhen wir wieder eine Ausnahme, so daß wir alle Handler Rekursiv aufrufen und alle Resourcen freigeben.

Beispiel:

```
CONST SIZE=100000
ENUM NOMEM  /* ,... */

RAISE NOMEM IF AllocMem()=NIL

PROC main()

  alloc()

ENDPROC


PROC alloc() HANDLE

  DEF mem

  mem:=AllocMem(SIZE,0)          /* sehen, wie viele Blöcke wir bekommen
                                    können */

  alloc()                        /* und jetzt die Rekursion .... */

  FreeMem(mem,SIZE)              /* wir werden nie hierher kommen ... */

EXCEPT

  IF mem THEN FreeMem(mem,SIZE)
  Raise(exception)               /* Rekursiver Aufruf aller Handler */

ENDPROC
```

Dieses ist, natürlich, eine Simulation eines natürlichen Programm-Problem, daß gewönlich komplexer ist, und soll auch nur den Gebrauch der Ausnahme-Benutzung darstellen. Für ein echtes Beispiel Programm würde das Fehlerhandling ohne Ausnahmezustände wesentlich schwieriger werden, siehe auch das 'D.e'-Utility Programm.

1.87 Exception Behandlung

```
13D. Benutzung von Exception-ID's
---------------------------------

Im echten Leben ist die Ausnahme-ID ein normaler 32-Bit-Wert
und du darfst alles mögliche an einen Ausnahme-Handler geben, z.B.
um es als Ausgabe für fehlerhafte Strings zu nutzen:


Raise('Could not open "gadtools.library"!')


Wie auch immer, wenn du die Ausnahmen in einer ausführbaren Weise
nutzen möchtest und Du möchtest auch zukünftige Module nutzen,
deren Ausnahmen nicht in Deinem Programm definiert sind, dann folge
den folgenden Vorschlägen:

- Benutze und definiere die ID 0 als "kein Fehler" (z.B. normaler Abbruch)

- Um Ausnahmezustände in Deinem Programm zu bestimmen, nutze die
  ID's 1-10000.

  Definiere diese mit der gewöhnlichen Methode von ENUM:


  ENUM OK,NOMEM,NOFILE,...


  (OK wird 0, und andere werden 1+)


- ID's 12336 bis 2054847098 (dieses sind alles Bezeichner als
  Bestandteil von groß-/kleingeschriebenen Buchstaben und Ziffern
  der Länge 2,3 oder 4 eingeschlossen in "") sind reserviert als
  gemeinsam benutzte Ausnahmen. Eine gemeinsame Ausnahme ist ein Ausnahme,
  die nicht in Deinem Programm definiert werden braucht, und die
  benutzt wird von Vorgaben von Modulen (mit Funktionen in ihnen) um
  Ausnahmen zu erhöhen: z.B., wenn du eine Anzahl von Prozeduren
  erstellst die in einem eigenem Task laufen, dann kannst Du die Ausnahmen
  erhöhen.

  Wenn du diese Funktionen in verschiedenen Programme nutzen möchtest,
  dann würde es nicht praktisch sein, die ID's mit dem Haupt Programm
  zu koordinieren, und ferner, wenn du mehr als eine Funktionen benutzt
  (in einem Modul, in der Zukunft) und jedes Modul würde eine
  unterschiedliche Id haben für 'kein Speicher!', dann können Dir die Dinge
  aus der Hand gleiten.

  Und hier kommen die gemeinsamen Ausnahmen zum tragen: die gemeinsame
  'kein Speicher'-ID ist "MEM" (einschliesslich den Anführungsstrichen):
  jeder kann jetzt einfach von jedem Punkt


  Raise("MEM")
```

von allen unterschiedlichen Prozeduren aufrufen, und der Programmierer,
der diese Module benutzt, braucht nur einen Ausnahme-Handler,
der "MEM" versteht.

Zukünftige Module, die verschiede Funktionen beinhalten, werden angeben,
was für Ausnahmen ein gesichertes Verfahren auslösen darf, und wenn diese
sich überlappen mit den ID's von anderen Prozeduren, dann wird die
Umgebung des Programmierers, die mit der Ausnahme zu arbeiten hat,
außerordentlich schwierig sein.

Beispiele

(system)

"MEM"	kein Speicher
"FLOW"	(beinahe) Stack überfließend
"^C"	Kontrollieren-C-Tasten-Abbruch
"ARGS"	schlecht Argumente

(exec/libraries)

"SIG"	konnte kein Signal zuteilen
"PORT"	konnte keinen Nachrichtenport erstellen
"LIB"	Library nicht verfügbar
"ASL"	keine asl.library
"UTIL"	keine utility.library
"LOC"	keine locale.library
"REQ"	keine req.library
"RT"	keine reqtools.library
"GT"	keinen gadtools.library (ähnlich für anderen)

(intuition/gadtools/asl)

"WIN"	kein Fenster zu öffnen
"SCR"	kein Schirm zu öffnen
"REQ"	kein Requester zu öffnen
"FREQ"	Kein Filerequester zu öffnen
"GAD"	konnte kein Gadget erstellen
"MENU"	konnte kein Menu erstellen

(dos)

"OPEN"	konnte kein File aufmachen/File existiert nicht
"OUT"	Proble beim lesen
"IN"	Probleme beim schreiben
"EOF"	Ende des Files
"FORM"	Eingabe Format Fehler

Die allgemeine Tendenz ist Großschreibung für allgemeine System

Ausnahmen und Kleinschreibung (oder gemischt) für spezifizierte Module.

- alles anderen (einschliesslich aller negativen ID's) sind reserviert.

1.88 Objektorientierte Programmierung

14.OBJEKTORIENTIERTE PROGRAMMIERUNG

Da in Version 2.1b noch nichts eingebaut ist, ist auch nichts dokumentiert. (Im Gegensatz zu Version 3.0 !!!)

Vorheriges Kapitel

Nächstes Kapitel

1.89 Der Inline-Assembler

15.DER INLINE-ASSEMBLER

A. Variablenteilung

B. Vergleich zwischen Inline-/Makroassembler

C. Wege, Binäredaten zu nutzen (INCBIN/CHAR..)

D. OPT ASM

Vorheriges Kapitel

Nächstes Kapitel

1.90 Der Inline-Assembler

15A. Variablenteilung

Wie Du wahrscheinlich beim Beispiel im Kapitel 5D erraten hast, können Assembleranweisungen frei mit E-Anweisungen vermischt werden. Das große Geheimniss ist, das ein kompletter Assembler in den Compiler eingebaut wurde.
Getrennt von den normalen Assembler Addressierungsmodies kannst Du auch folgende Identifiers benutzen:

```
meinlabel:
LEA mylabel(PC),A1      /* Labels */
```

```
DEF a                   /* Variablen */
MOVE.L (A0)+,a          /* Beachte das <var> ein <offset>(A4) (or A5) ist */

MOVE.L dosbase,A6       /* Identifiers für Library-Aufrufe */
JSR    Output(A6)

MOVEQ  #TRUE,D0         /* Konstanten */
```

1.91 Der Inline-Assembler

```
15B. Vergleich zwischen Inline-/Makroassembler
----------------------------------------------
Der Inline-Assembler unterscheidet sich etwas von einem normalen Macro-
Assembler. Dies ist dadurch bedingt, daß er eine Erweiterung von E ist, und
deshalb der E-Syntax folgt. Hauptunterschiede sind:

- Kommentare werden nicht mit einem ';' Semikolon eingeleitet, sondern in /*
  */ eingeschlossen, sie haben unterschiedlich Bedeutung.
- Schlüsselworte und Register werden großgeschrieben, alles ist von der Groß-
  und Kleinschreibung abhängig.
- keine Macros und ander luxuriöse Assemblereigenschaften (es gibt schließ-
  lich den kompletten E-Sprachumfang dafür)
- Du solltest aufpassen, daß Du den Inhalt der Register A4/A5 nicht mit dem
  Inline-Assembler überschreibst, da sie vom E-Code benutzt werden.
- keine Unterstützung des Large Modells/Relloc-Hunks im Assembler -JETZT-
  Dies bedeutet hauptsächlich, das Du bis jetzt die PC-Relative Addressie-
  rung benutzen mußt.
```

1.92 Der Inline-Assembler

```
15C. Wege, Binäredaten zu nutzen (INCBIN/CHAR..)
------------------------------------------------
INCBIN

Syntax: INCBIN <filename>

Fügt einen Binär-File genau am Punkt des Statements ein, und sollte deshalb
vom Code getrennt werden. Beispiel:

meinetab:   INCBIN <filename>

LONG, INT, CHAR

Syntax: LONG <werte>
        INT  <werte>
        CHAR <werte>

Erlaubt Dir binäre Daten direkt in ein Programm einzufügen. Funktioniert
fast wie DC.x in Assembler. Beachte, daß das CHAR-Statement auch Strings
annimmt und immer auf gerade Wortaddressen gelegt wird. Beispiel:
```

```
meinedaten: LONG 1,2; CHAR 3,4,'Hey Leute',0,1
```

1.93 Der Inline-Assembler

```
15D. OPT ASM
------------
OPT ASM  wird im Kaptel 16A besprochen. Es erlaubt Dir EC wie einen Assemb-
ler zu programmieren. Es gibt keinen guten Grund, EC anstatt eines Assemb-
lers zu  nehmen, außer der Geschwindigkeit. Er ist wesentlich schneller als
zum Beispiel A68k, gleich mit dem DevPac und langsamer als AsmOne. Du wirst
auch eine schwere Zeit haben, wenn Du die alten Seka-Sources von deiner
Disk schröpfen willst, aufgrund der aufgeführten Unterschiede (siehe 15B).
Wenn Du Assembler-Programme mit EC schreiben und sie zu anderen Assemblern
kompatibel halten willst, schreibe vor jede E-Speziefische Funktion ein ";",
EC wird sie benutzen und jeder andere Assembler ihn wird als Kommentar an-
sehen.
Beispiel:

;OPT ASM

start:  MOVEQ   #1,D0   ;/*macht etwas dummes*/
        RTS             ;/*und steigt aus*/

dies wird von jedem Assembler assembliert, EC eingeschlossen.
```

1.94 Dinge über den Compiler

```
                        16.DINGE üBER DEN COMPILER


                A. Das OPT Schlüsselwort

                B. Small/Large Modell

                C. Stack Organisation

                D. Festgeschriebene Begrenzungen

                E. Fehlermeldungen, Warnungen und nicht dokumentierte Tests

                F. Compiler Puffer Organisation und Anforderung

                G. Eine kurze Entstehungsgeschichte

                Vorheriges Kapitel

                Nächstes Kapitel
```

1.95 Dinge über den Compiler

```
16A. Das OPT Schlüsselwort
--------------------------

OPT, LARGE, STACK, ASM, NOWARN, DIR, OSVERSION

syntax: OPT <Optionen>

Bietet die Möglichkeit, die Einstellungen des Compilers zu verändern.

LARGE  Das Code- und Datenmodell wird auf "LARGE" gestellt. Grundein-
       stellung ist "SMALL"; der Compiler generiert einen 100% pc-
       ähnlichen Code, mit einer Maximalgröße von 32K. Mit "LARGE"
       gibt es keine solchen Begrenzungen, zudem werden "reloc-hunks"
       generiert. Siehe -L
STACK=x  Setzt die Stackgröße auf x Bytes. Man sollte diese Option
         nur verwenden, wenn man weiß, was man tut. Normalerweise
         schätzt der Compiler die benötigte Stackgröße selbstständig
         recht gut ein.
ASM  Der Compiler wird in den Assembler-Modus geschaltet. Von da an
     sind nur noch Assembler-Befehle erlaubt und es wird kein Initi-
     alisierungscode generiert. Siehe: das Kapitel über integriertes
     Assembler.
NOWARN  Schaltet die Warnungen aus. Der Compiler gibt eine Warnung
        aus, wenn er *glaubt*, daß das Programm falsch ist, syntakt-
        isch aber in Ordnung ist. Siehe -n
DIR=moduledir  Legt das Verzeichnis fest, in dem der Compiler nach
               Modulen sucht. Grundeinstellung ist 'emodules:'
OSVERSION=vers  Grundeinstellung ist 33.(V1.2). Setzt die Minimum-
                Version des Kickstarts (wie z.B. 37. für V2.04) fest,
                ab denen das Programm laufen soll. Auf diesem Weg
                bricht das Programm einfach ab, wenn die dos.library
                einer älteren Version in der Initialisierungsroutine
                auf einer älteren Maschine geöffnet wird. Allerdings
                ist es für den User hilfreicher, wenn man die Kick-
                startversion selber testet und eine geeignete Fehler-
                meldung ausgibt.
Beispiel:
OPT STACK=20000,NOWARN,DIR='DF1:Modules',OSVERSON=39
```

1.96 Dinge über den Compiler

```
16B. Small/Large Modell
-----------------------

Amiga-E läßt einem die Wahl zwischen einem SMALL und einem LARGE Code/
Daten-Modell. Allerdings dürften die meisten Programme, die man schreibt
(besonders, wenn man gerade erst mit Amiga-E angefangen hat), in die
32K passen, wenn compiliert wird: man braucht sich also keine Gedanken
über das Einstellen eines Code-Genrierungs-Modelles machen. Man kann die
Notwendigkeit eines LARGE-Modelles daran erkennen, daß sich EC darüber
beschwert, daß es den Code nicht mehr in die 32K hineinbekommt. Folgender
Befehl compiliert einen Source-Code mit dem LARGE-Modell:
1> ec -l sizy.e
oder, besser, setzen sie die Anweisung
```

```
OPT LARGE
in ihren Code.
```

1.97 Dinge über den Compiler

```
16C. Stack Organisation
-----------------------

Um alle lokalen und globalen Variablen zu speichern, weißt das run-time Sys-
tem eines von E generierten ausführbaren Programmes einen Speicherbereich zu,
von dem es einen festen Teil verwendet, um alle globalen Variablen zu speichern.
Der Rest wird dynamisch verwendet, wenn Funktionen aufgerufen werden. Wenn in E
eine Funktion aufgerufen wird, wird ein Bereich im Stack reserviert, um alle
lokalen Daten zu speichern, sobald die Funktion beendet ist, wird der Bereich
wieder freigegeben. Deshalb ist es gefährlich, große Arrays für lokale Daten
zu haben, wenn diese rekursiv aufgerufen werden: alle Daten der vorherigen Auf-
rufe der selben Funktion befinden sich noch immer im Stack und besetzen somit
einen großen Bereich in diesem. Werden Prozeduren jedoch linear aufgerufen, dann
kann der Stack nicht überlaufen.
Beispiel:

global data:        10K (arrays e.d)
local data PROC #1:  1K
local data PROC #1:  3K

Das run-time System reserviert immer zusätzliche 10K für normale Rekursion (z.B.  ↩
    mit
kleinen lokalen Arrays) und weitere Buffers/Systemspeicher, so daß insgesamt 24K
Stackspeicher zugewiesen werden.
```

1.98 Dinge über den Compiler

```
16D. Festgeschriebene Begrenzungen
----------------------------------

Beachte diese Zeichen:  (+-)   ungefähr, hängt von der Situation ab
                        (n.l.) kein klares Limit, aber dieser Wert scheint  ↩
                           sinnvoll

OBJEKT/GEGENSTAND                                          WERT/MENGE/MAX
-------------------------------------------------------------------------------
Wert des Datentyps CHAR                                    0 ... 255
Wert des Datentyps INT                                     -32K ... +32K
Wert des Datentyps LONG/PTR                                -2Gig ... +2 Gig

Identifierlänge                                            100 bytes (n.l.)
Länge einer Quellcodezeile                                 2000 lexikalische Zeichen  ↩
    (+-)
Quellcode Länge                                            2 Gig (theoretisch)
konkrete listen                                            einige hundert Elemente  ↩
    (+-)
konstante Strings                                          1000 Zeichen (n.l.)
```

```
max. Tiefe der Schleifen                                500 tief
max. Tiefe der Kommentare                               unbegrenzt

# der lokalen Variablen pro Prozedur                    8000
# der globalen Variablen                                7500
# der Argumente für eigene Funktionen                   8000 (zusammen mit locals ↩
    (?))
# der für E-varargs Funktionen [WriteF()]               64

ein Objekt (lokal/global oder dyn. zugewiesen)          8K
ein Array, List oder String (lokal oder global)         32K
ein String (dynamisch)                                  32K
ein List dynamisch)                                     128K
ein Array (dynamisch)                                   250MB

lokale Daten pro Prozedur                               250MB
globale Daten                                           250MB

Code-Größe einer Prozedur                               32K
Code-Größe eines ausführbaren Progs.                    32K SMALL, 2Gig LARGE ↩
    Modell

Buffer-Größe eines generierten Codes und Identifiers    abhängig vom Quellcode
Buffer-Größe von Sprungmarken/Zweigen                   unabhängig (wieder) ↩
    zugewiesen
```

1.99 Dinge über den Compiler

```
16E.  Fehlermeldungen, Warnungen und nicht dokumentierte Tests
-------------------------------------------------------------

Manchmal. wenn sie ihren Quellcode mit EC compilieren, erhalten sie eine Meldung, ↩
    die
ungefähr folgendermaßen aussieht: UNREFERENCED <ident.>, <ident.>, ...
Dies passiert, wenn sie Variable, Funktionen oder Sprungmarken definieren, diese ↩
    aber
nicht verwenden. Dies ist ein Extra-Service des Compilers, der helfen soll, solche
schwer zu findenden Fehler zu entdecken. Es gibt mehrere Warnungen, die der ↩
    Compiler
ausgiebt, um sie darauf aufmerksam zu machen, daß etwas nicht in Ordnung ist, dies ↩
     aber
kein echter Fehler ist.

- "A4/A5 used in line assembly"
  Diese Warnung wird ausgegeben, wenn sie Register A4 und A5 in ihrem Assembler ↩
     Code
  verwenden. Der Grund dafür ist, daß diese Register von E intern verwendet werden ↩
     ,
  um lokale und globale Variable richtig zu addressieren. Natürlich kann es gute ↩
     Gründe
  geben, diese zu gebrauchen, wie MOVEM.L A4/A5,-(A7) vor einen großen Stück ↩
     Inline-
  Assembler-Code.

- "keep eye on stacksize"
```

```
- "stack is definitely too small"
  Beides kann ausgegeben werden, wenn sie OPT STACK=<size> verwenden. Der Compiler ↩
      ver-
  gleicht einfach ihre Angabe mit seiner eigene Schätzung (siehe Kapitel 16C.),  ↩
     und gibt
  erstere Meldung aus, wenn er meint, die Größe sei etwas knapp kalkuliert oder  ↩
     letztere,
  wenn sie definitiv zu klein ist.

- 'suspicious use of "=" in void expression'
  Diese Warnung erscheint, wenn sie den Ausdruck 'a=1' als Anweisung gebrauchen.  ↩
     Ein Grund
  ist, daß ein Vergleich als Anweisung wenig Sinn macht, aber der Hauptgrund ist,  ↩
     daß dies
  oft vorkommende Rechtschreibfehler bei 'a:=1' ist. Den vergessenen ":" zu finden ↩
      ist
  schwer, aber er kann ernsthafte Konsequenzen haben.

FEHLER:

- 'syntax error'
  Häufigster Fehler. Diese Fehlermeldung erscheint, wenn entweder keine andere  ↩
     Meldung passt,
  oder ihre Anordnung des Codes dem Compiler etwas seltsam erscheint.

- 'unknown keyword/const'
  Sie haben einen Identifier in Großbuchstaben (wie "IF" oder "TRUE") verwendet,  ↩
     und der Compiler
  konnte keine Definition dafür finden. Gründe:
  * falschgeschriebenes Schlüsselwort
  * sie haben eine Konstante verwendet, diese aber nicht zuvor mit CONST definiert
  * sie haben vergessen, das Modul anzugeben, in dem ihre Konstante definiert ist

- '":=" expected'
  Sie haben eine FOR Anweisung oder eine Zuweisung geschrieben, und haben dabei  ↩
     etwas anderes als
  ":=" verwendet.

- 'unexpected characters in line'
  Sie haben Zeichen verwendet die in E außerhalb von Strings keine syntaktische  ↩
     Bedeutung haben.
  Beispiel: "§!&Öß"

- 'label expected'
  In bestimmten Fällen, zum Beispiel nach den Schlüsselwörtern PROC oder JUMP, ist ↩
      ein Iden-
  tifier notwendig. Sie haben irgendetwas anderes geschrieben.

- '"," expected'
  Innerhalb einer Gegenstandsliste (z.B. eine Parameter Liste), haben sie etwas  ↩
     anderes als
  ein Komma verwendet.

- 'variable expected'
  Diese Konstruktion braucht eine Variable, Beispiel:
  FOR <var>:= ... etc.
```

```
- 'value does not fit into 32 bit'
  Beim spezifizieren einer Konstanten haben sie einen zu großen Wert einegegeben, ↩
     Beispiel:
  $FFFFFFFFF, "abcdef"  /siehe Kapitel 2A-2E)
  Diese Meldung erscheint auch, wenn ein SET-Befehl mit mehr als 32 Elementen ↩
     verwendet wird.

- 'missing apostrophe/quote'
  Sie haben ein ' am Ende einer String vergessen.

- 'incoherrent program structure'
  * sie haben eine neue PROCedure gestartet, ohne die vorherige zu beenden.
  * die Verzweigung ihrer Programme ist falsch, z.B.:
    FOR
      IF
      ENDFOR
    ENDIF

- 'illegal command-line option'
  Innerhalb der Befehlszeile 'EC -opt source' haben sie für -opt einen Ausdruck ↩
     verwendet, der
  EC unbekannt ist.

- ' division and multiplication 16 bit only'
  Der Compiler hat festgestellt, daß sie einen 32 bit Wert für * oder / verwendet ↩
     haben. Dies
  würde nicht den erwünschten Wert im Runtime ergeben.
  Siehe Mul() und Div().

- 'superfluous items in expression/statement'
  Nachdem der Compiler ihren Anweisung bearbeitet hat, hat er immer noch Zeichen ↩
     anstelle
  eines Linefeeds gefunden. Sie haben wahrscheinlich den <lf> oder ";" vergessen, ↩
     um zwei
  Anweisungen zu trennen.

- 'procedure "main" not available'
  Ihre Programm hat keine 'main procedure'.

- 'double declaration of label'
  Sie haben eine Sprungmarke zweimal vergeben, z.B.:
  label:
  PROC label()

- 'unsafe use of "*" or "/"'
  Dies hat wieder etwas mit 16 Bit anstelle von 32 Bit * und / zu tun. Siehe ' ↩
     division and
  multiplication 16 bit only'.

- 'reading sourcefile didn't succed'
  Überprüfen sie ihre Angaben für die Quelle, die sie mit 'ec mysource' angegeben ↩
     haben.
  Achten sie darauf, daß die Quelle und nicht die Kommandozeile auf '.e' endet.

- 'writing executable didn't succed'
  Der Versuch, das eben generierte ausführbare Programm zu schreiben verursachte ↩
     einen DOS-
```

```
  Fehler. Unter umständen existierte das Programm bereits und konnte nicht ↩
     überschrieben
  werden.

- 'no args'
  "GEBRAUCH: ec [-opts] <Quellcodedateiname> ('.e' wird hinzugefügt)"
  Sie erhalten diese Meldung, wenn sie ec ohne Argumente verwenden.

- 'unknown/illegal addressing mode'
  Dieser Fehler erscheint nur, wenn sie den inline Assembler verwenden. Mögliche ↩
     Gründe:
  * sie haben eine Addressierungsweise verwendet, die es für den 68000er nicht ↩
     gibt.
  * die Addressierungsmethode existiert, aber nicht für diesen Befehl. Nicht alle ↩
     Assembler-
    Befehle unterstützen alle Kombinationen der effektiven Addressen für Quelle ↩
       und Ziel.

- 'unmatched parentheses'
  Ihre Anweisung hat mehr "(" als ")" oder umgekehrt.

- 'double declaration'
  Ein Identifier wird in zwei oder mehr Deklarationen verwendet.

- 'unknown'
  Ein Identifier wird in keiner Deklaration verwendet; er ist unbekannt. ↩
     Wahrscheinlich haben
  sie vergessen, ihn in eine DEF-Anweisung zu setzen.

- 'incorrect # of args or use of ()'
  * sie haben vergessen "(" oder ")" an die richtige Stelle zu setzen
  * sie haben eine falsche Anzahl von Argumenten für eine Funktion verwendet

- 'unknown e/library function'
  Sie haben einen Identifier mit einem Großbuchstaben begonnen und dann mit ↩
     Kleinbuchstaben
  fortgesetzt, aber der Compiler konnte keine Definition finden.
  Mögliche Gründe:
  * eine Funktion wurde falschgeschrieben
  * sie haben das Modul miteinzuschließen, das diesen Bibliotheksaufruf enthält

- 'illegal function call'
  Erscheint selten. Sie erhalten diesen Fehler, wenn sie seltsame Funktionsaufrufe ↩
      starten,
  wie z.B. verzweigte WriteF()'s :
  WriteF(WriteF('hi'))

- 'unknown format code following ""'
  Sie haben in einer String einen Formatcode verwendet, der unzulässig ist.
  Siehe Kapitel 2F für eine Liste der Formatcodes.

- '/* not properly nested comment structure */'
  Die Anzahl der '/*' stimmt nicht mit der Anzahl der '*/' überein, oder haben ↩
     eine komische
  Reihenfolge.

- 'could not load binary'
```

```
  <filespec> innerhalb von INCBIN <filespec> konnte nicht gelesen werden.

- '"}" expected'
  Sie haben einen Ausdruck mit "{<var>" begonnen, aber das "}" vergessen.

- 'immediate value expected'
  Manche Konstruktione erfordern einen direkten Wert anstelle eines Ausdrucks.
  Beispiel:
  DEF s[x*y]:STRING   /* falsch, nur etwas wie s[100]:STRING ist zulässig */

- 'incorrect size of value'
  Sie haben einen unzulässig großen/kleinen Wert in einer Konstruktion verwendet.
  Beispiel:
  DEF s[-1]:STRING, +[1000000]:STRING    /* muß 0 ... 32000 sein */
  MOVEQ #1000,D2                         /* muß -128 ... 127 sein */

- 'no e code allowed in assembly modus'
  Sie haben den Compiler als Assembler arbeiten lassen, aber, aus Versehen, E-Code
  geschrieben.

- 'illegal/inappropriate type'
  An einer Stelle wo eine <type> Spezifikation notwendig gewesen wäre, haben sie ↩
     etwas
  unpassendes eingegeben. Beispiele:
  DEF a:PTR TO ARRAY        /* es gibt keinen solchen Typ */
  [1,2,3]:STRING

- '"]" expected
  Sie haben mit einem "[" begonnen, aber nie mit einem "]" aufgehört.

- ' statement out of local/global scope'
  Ein wesentlicher Punkt bei der Kontrolle ist die erste PROC-Anweisung. Davor ↩
     sind nur
  globale Definitionen (DEF, CONST,MODULE etc.) erlaubt, und keinerlei Code. Im ↩
     zweiten
  Teil sind nur Code, aber keine globalen Definitionen erlaubt.

- 'could not read module correctly'
  Es gab einen DOS-Fehler beim Versuch, ein Modul von einer MODULE-Anweisung ↩
     einzulesen.
  Gründe:
  * emodules:wurden nciht korrekt zugewiesen (assign emodules: ...)
  * der Modulname wurde falsch geschrieben, oder es existiert nicht
  * sie haben MODULE 'bla.m' anstelle von MODULE 'bla'

- 'workspace full'
  Erscheint selten. Wenn dieser Fehler erscheint, müssen sie EC mit der '-m' ↩
     Option dazu
  zwingen, die Schätzung über die benötigte Speichermenge höher anzusetzen. ↩
     Versuchen sie
  es zuerst mit '-m2', dann '-m3', bis der Fehler verschwindet. Sie müssen aber ↩
     schon rie-
  sige Anwendungsprogramme, mit einer Unmenge Daten schreiben, damit dieser Fehler ↩
      er-
  scheint.

- 'not enough memory while (re-)allocating'
```

```
  Mögliche Lösungen für dieses Problem:
  1. Sie haben andere Programme im Multitasking laufen. Stoppen sie diese und ↩
     versuchen
     sie es nocheinmal.
  2. Sie haben akuten Speichermangel und der Speicher war fragmentiert. Rebooten ↩
     sie.
  3. Weder 1. noch 2., kaufen sie sich eine Speichererweiterung (hüstel).

- 'incorrect object definition'
  Sie haben bei einer Definition zwischen OBJECT und ENDOBJECT Blödsinn ↩
     geschrieben.
  Siehe Kapitel 8F, um herauszufinden, wie's richtig geht.

- 'incomplete if-then-else expression'
  Wenn sie IF als einen Operator verwenden, dann muß ELSE ein Teil dieses ↩
     Ausdrucks sein:
  ein Ausdruck mit einer IF-Anweisung muß immer einen Wert zurückgeben, aber wenn ↩
     keine
  ELSE-Anweisung da ist, kann IF im Prinzip nichts tun.

- 'unknown object identifier'
  Sie haben einen Identifier verwendet, den der Compiler als einen Teil eines ↩
     Objekts er-
  kannt hat, aber sie haben vergessen, ihn zu deklarieren. Gründe:
  * falsch geschriebener Name
  * fehlendes Modul
  * der Identifier innerhalb des Modules wird nicht so geschrieben, wie sie es aus ↩
      den
    Rom-Kernel-Manuals erwartet haben. Überprüfen sie es mit ShowModule.
    Beachten sie, daß Amiga-System-Objekte auf Assembler Identifiern basieren und ↩
       nicht auf
    C. Zweitens: Identifier folgen dem E-Syntax.

- 'double declaration of object identifier'
  Ein Identifier wurde in zwei Objekt Definitionen verwendet.

- 'reference(s) out of 32K range: switch to LARGE model'
  Ihr Programm wird größer als 32K. Fügen sie einfach 'OPT LARGE' in ihren ↩
     Quellcode mit
  ein. Siehe Kapitel 16B.

- 'reference(s) out of 256 byte range'
  Sie haben wahrscheinlich BRA.S oder Bcc.S über eine zu große Distanz geschrieben ↩
     .

- 'too sizy expression'
  Sie haben wahrscheinlich eine Liste von [], möglicherweise [[]], geschrieben, ↩
     die zu groß
  ist.

- 'incomplete exception handler definition'
  Sie haben unter Umstaänden EXCEPT ohne HANDLE verwendet, oder aber auch anders ↩
     herum.
  Siehe Kapitel 13 für "exception handling".
```

1.100 Dinge über den Compiler

```
16F. Compiler Puffer Organisation und Anforderung
-------------------------------------------------

Wenn sie den Fehler 'workspace full' (sehr unwahrscheinlich) erhalten, oder wissen ↩
     wollen was
wirklich passiert, wenn ihr Programm compiliert wird, ist es wichtig, zu wissen,  ↩
    wie EC seine
Buffer organisiert.
Ein Compiler, und in diesem Fall EC, braucht Buffer, um alle möglichen Dinge, wie  ↩
    z.B. Identi-
fier nachverfolgen zu können. Es braucht diese auch, um darin den generierten Code ↩
     zu speichern.
EC weiß nicht, wie groß diese Buffer sein müssen. Bei manchen Buffern, wie dem für ↩
     Konstanten,
ist das kein Problem: wenn der Buffer voll ist, weißt EC einfach ein weiteres  ↩
    Stück Speicher zu
und arbeitet dann weiter. Andere Buffer, wie der für den generierten Code, müssen  ↩
    ein einziger
Speicherblock sein, der sich während des Compilierens nicht verändert: EC muß also ↩
     den notwendigen
Speicher gut abschätzen, um große und kleine Quellcodes compilieren zu können. EC  ↩
    berechnet also
anhand des Quellcodes den benötigten Speicherplatz und fügt dazu nocheinmal eine  ↩
    ansehnliche Menge
hinzu. Somit reicht in 99% aller Fälle der Speicher aus, andern Falls erhalten sie ↩
     eine Fehler-
meldung un müssen weiteren Speicher mit der '-m' Option hinzufügen.
Experimentieren sie mit unterschiedlichen Typen und Größen von Quellcoden in  ↩
    Verbindung mit der
'-b' Option um herauszufinden, wie es funktioniert.
```

1.101 Dinge über den Compiler

```
16G. Eine kurze Entstehungsgeschichte
-------------------------------------

E ist nicht einfach einen weitere Programmiersprache: sie wurde schrittweise und  ↩
    vorsichtig
vom Autor entwickelt, da er mit den existierenden Programmiersprachen nicht  ↩
    besonders glück-
lich war, speziell den langsamen und "schlabrige-Codes-erzeugenden" Compilern, die ↩
     es für
diese gab. E hatte als primäres Ziel dem Autor für seine Entwickling von Amiga  ↩
    Programmen zu
dienen, und war dabei bisher sehr erfolgreich. E wurde in einen Zeitraum von 1½  ↩
    Jahren in Ver-
bindung mit intensiver Arbeit entwickelt und war nicht der erste Compiler, den der ↩
     Autor ge-
schrieben hat: manch einer erinnert sich vielleicht an den DEX-Compiler.

Dieser war langsam und nicht besonders mächtig und kann nur schwerlich mit dem  ↩
    Amiga E Com-
```

piler verglichen werden, aber er gab dem Autor wichtige Erfahrungen, die halfen, ↩
Amiga E zu
dem zu machen, was es heute ist. DEX Programierer werden feststellen, daß es sehr ↩
einfach ist
ihre Quellcodes in E umzuwandeln, und die Entwicklung mit der 10fachen Power und ↩
der 20fachen
Geschwindigkeit fortzusetzen. Eine witzige Sache an DEX ist, daß sich die ↩
Entwicklung von DEX
und E überschnitten hat: als DEX fertig war, war E V1.6 zur Hälfte geschrieben. ↩
Weil E schon
damals wesentlich besser war, wurden E Bibliotheken/Beispiele und Codes auf ↩
allgemeinen Wunsch
hin auf DEX übertragen, so daß der Vorgänger Teile seines Nachfolgers beinhaltete.
Der AUtor hat auch noch weitere Compiler und Interpreter geschrieben, die aber ↩
teilweise nie
veröffentlicht wurden.

Amiga E ist ein Produkt, das weiter entwickelt wird, bis es die ultimative Sprache ↩
und Amiga
Entwicklungssystem ist:

- indem diese, bisher fehlenden Teile, in die Sprache miteinbezogen werden
  * Objektorientierung
  * besseres Fließkomma Konzept

- indem am Compiler einige Veränderungen vorgenommen werden
  * mögliche Genrierung von 020/030/881 Code
  * Optimierung des Compilierungsprozesses, so daß unter Umstäden sich Zeile/ ↩
    Minute Figuren
    verdoppeln
  * es soll dem User ermöglicht werden, eigenen Code in Module zu wandeln, um ↩
    somit große
    Anwendungen modular aufbauen zu können

- indem wetvolle Elemente dem Packet hinzugefügt werden
  * ein integrierte Editor ?
  * source-level debugger ?
  * CASE Werkzeuge, zum Beispiel

- indem Bugs entfernt werden (welche Bugs ??!!)

Das ENDE! Schnauf! Viel Spaß mit E!