ANWENDER-DOKUMENTATION	Sprachbeschreibung Programmiersprache C	AC A7100/A7150
MUTOS 1700	11/87	

Programmtechnische Beschreibung Teil 2

Sprachbeschreibung Programmiersprache C

11/87

VEB Kombinat Robotron

VEB Robotron-Projekt Dresden

Die vorliegende Systemunterlagendokumentation, Sprachbeschreibung Programmiersprache C, 11/87, entspricht dem Stand von MUTOS 1700.

Die Ausarbeitung erfolgte durch ein Kollektiv des VEB Robotron-Elektronik Dresden.

Herausgeber:

VEB Robotron-Projekt Dresden, Leningrader Str. 9, Dresden 8010

Kurzreferat

C ist eine vielseitig verwendbare Programmiersprache. Sie ist gekennzeichnet durch

- Möglichkeiten zur Strukturierung von Daten und der Programmablaufsteuerung,
- ein flexibel nutzbares Datentypkonzept,
- eine umfangreiche Menge an Operatoren und
- eine kompakte Schreibweise.

Mit C können Programme für numerische wie auch nichtnumerische Probleme geschrieben werden.

Die Sprache C stellt einen Kompromiß zwischen einer typischen höheren Programmiersprache und der Assemblersprache der jeweiligen Maschine dar. Die Nutzung von C erfordert vom Programmierer einen diziplinierten Programmierstil. Für Neulinge auf dem Gebiet der Programmierung ist C nicht zu empfehlen.

Die Stärke von C liegt in der Systemprogrammierung. C ist eine maschinenorientierte Systemprogrammiersprache. Dank dieser Eigenschaften können in C geschriebene Programme relativ einfach portiert werden.

1. Methode der Sprachbeschreibung

1.1. Vorbemerkung

Das vorliegende Handbuch über die Programmiersprache C will ein Nachschlagewerk und kein Lehrbuch sein. Die Sprache C befindet sich seit dem Zeitpunkt ihres ersten Entwurfs in einer ständigen Weiterentwicklung. Derzeit gibt es Bemühungen, einen Sprachstandard festzulegen. Hier wird der Sprachumfang wie er in /1/, /2/ und /3/ beschrieben wird, zugrunde gelegt.

1.2. Darstellungsform

Die Sprache C wird halbformal beschrieben. Die Syntax wird in einer erweiterten Backus-Naur-Form und die Semantik verbal angegeben.

Eine Syntaxregel hat hier die Form:

```
        <metalinguistische_Variable> :
                  <syntaktische_Formel>
```

Die metalinguistische Variable links vom Definitionszeichen ist die syntaktische Einheit, die durch die syntaktische Formel erklärt wird. Eine so erhaltene metalinguistische Variable kann benutzt werden, um weitere metalinguistische Variable darzustellen. Die syntaktischen Formeln bestehen mithin aus metalinguistischen Variablen, aus C-Zeichen sowie aus den folgenden metalinguistischen Konnektoren mit der angegebenen Bedeutung:

`:` Definitionszeichen,

`<x>|<y>` Auswahl einer der syntaktischen Konstruktionen <x> oder <y> (Alternative),

`[<x>]` wahlweises Auftreten der syntaktischen Konstruktion <x> (Option),

`{<x>}` die Konstruktion <x> kann einmal, mehrmals oder überhaupt nicht auftreten (Wiederholung).

In der Beschreibung werden die metalinguistischen Variablen klein und in spitzen Klammern geschrieben. Sie werden der im Deutschen üblichen Flexion unterworfen. Den metalinguistischen Variablen können Ziffern angefügt sein, um damit auszudrücken, daß sie verschiedene Repräsentanten darstellen sollen. Auch für diese metalinguistischen Variablen gelten die Syntaxregeln, in denen den Metavariablen keine Ziffern angefügt sind.

~~~~~~
2. Elemente der Sprache C
~~~~~~~~~~~~~~~~~~~~~~~~~
2.1. Zeichenmenge von C
~~~~~~~~~~~~~~~~~~~~~~~

Die grundlegende Menge von <zeichen> kann man in <buchstaben>, <ziffern>, <sonderzeichen>, <apostroph> und <nichtgrafische_zeichen> klassifizieren.

```
        <zeichen> :
                  <buchstabe>
                | <ziffer>
                | <sonderzeichen>
                | <apostroph>
                | <nichtgrafisches_zeichen>

        <buchstabe> :
                | a | b | c | d | e | f | g | h | i
                | j | k | l | m | n | o | p | q | r
                | s | t | u | v | w | x | y | z
                | A | B | C | D | E | F | G | H | I
                | J | K | L | M | N | O | P | Q | R
                | S | T | U | V | W | X | Y | Z | _

        <ziffer> :
                  0 | <nicht_null_ziffer>

        <nicht_null_ziffer> :
                  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

        <sonderzeichen> :
                | ! | ? | ( | ) | [ | ] | { | } | +
                | - | = | < | > | / | | | . | , | :
                | ; |   | * | & | # | % | $ | ^ | ~

        <nichtgrafisches_zeichen> :
                | \n | \t | \b | \r | \f | \\ | \' | \0 | \"
                | \<oktalziffer >[<oktalziffer >]

        <apostroph> :
                    '


        <anführungszeichen> :
                  "


        <oktal_ziffer> :
                  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

        <hexadezimal_ziffer> :
                <ziffer> | a | b | c | d | e | f
                 | A | B | C | D | E | F
```

Aus der Menge der
<zeichen> werden als kleinste bedeutungtragende Sprachelemente die folgenden Morpheme gebildet:

<bezeichner>, <schlüsselwort>, <konstante>, <zeichenkette>, <operator> und <zwischenraum>.

Vom implementierten Zeichensatz ist es abhängig, ob Kleinbuchstaben vorhanden sind oder nicht. Die <sonder_zeichen> können
~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~
ebenfalls eingeschränkt sein und eventuell durch Ersatzzeichenfolgen dargestellt werden. Bei derartigen Implementierungen können die Programme schwerer lesbar werden.

Zeichen mit spezieller Funktion werden in den weiteren Abschnitten der Dokumentation durch Fettdruck hervorgehoben.

2.2. Bezeichner

~~~~~~~~~~~~~~~
Ein <bezeichner> ist eine Folge von <buchstaben> und <ziffern>. Das erste <zeichen> muß ein <buchstabe> sein. Das Unterstreichungszeichen _ zählt als ein <buchstabe>.

```
        <bezeichner> : <buchstabe> { <buchstabe> | <ziffer } 
```

Groß- und Kleinbuchstaben werden in Abhängigkeit von der Compilerimplementation unterschieden. Ebenso ist die signifikante Länge von <bezeichnern> vom Compiler abhängig (siehe Anlage 1 `Unterschiede in den Compilerimplementationen'). Ungeachtet dieser Tatsache kann man beliebig lange <bezeichner> schreiben.

Die <bezeichner> dienen der Identifizierung von Programmobjekten, zum Beispiel von Variablen, Schlüsselworten, Funktionen, Parametern, Datentypen, Deklaratoren, Marken und symbolischen Konstanten.

Werden Klein- und Großbuchstaben unterschieden, empfiehlt es sich, symbolische Konstanten mit Großbuchstaben zu notieren und die übrigen <bezeichner> mit Kleinbuchstaben zu schreiben (siehe auch Präprozessor).

2.3. Schlüsselworte

~~~~~~~~~~~~~~~~~~~
```
         <schlüsselwort> :
                   auto    | break   | case
                 | char    | continue| default
                 | do      | double  | else
                 | enum    | extern  | float
                 | for     | goto    | if
                 | int     | long    | register
                 | return  | short   | sizeof
                 | static  | struct  | switch
                 | typedef | while   | union
                 | unsigned
```

Die als <schlüsselworte> erklärten <bezeichner> sind reserviert und dürfen nicht anders verwendet werden. Sie müssen klein geschrieben werden.
In den folgenden Abschnitten der Dokumentation werden die <schlüsselworte> durch Fettdruck hervorgehoben. Beim Schreiben von Programmen werden die <schlüsselworte> wie alle anderen <bezeichner> geschrieben.
Infolge der Standardisierungsabsichten bezüglich der Sprache C können weitere Schlüsselworte zu den oben angegebenen hinzukommen.
~~~~~~~~~~~~~~~~~~~

2.4. Konstanten

Korrespondierend zu den in der Sprache C vorhandenen Daten-<typen> (siehe Abschnitt 3.) können die folgenden <konstanten> notiert werden.

```
        <konstante> :
                  <integer_konstante>
                | <long_integer_konstante>
                | <gleitkomma_konstante>
                | <zeichen_konstante>
                | <zeichenketten_konstante>
```

2.4.1. Integer-Konstante

Eine <integer-konstante> besteht aus einer Folge von <ziffern>. Die <integer_konstanten> treten als <dezimal_konstanten>, <oktal_konstanten> oder <hexadezimal_konstanten> auf.

```
        <integer_konstante> :
                  <dezimal_konstante>
                  <oktal_konstante>
                  <hexadezimal_konstante>

        <dezimal_konstante> :
                  <nicht_null_ziffer>[<dezimalzahl>]

        <oktal_konstante> :
                  0<oktalziffer>{<oktalziffer>}

        <hexadezimal_konstante> :
                  0x<hexadezimalziffer>{<hexadezimalziffer>}
                  0X<hexadezimalziffer>{<hexadezimalziffer>}
```

Eine <integer_konstante> wird als eine <oktal_konstante> betrachtet, wenn sie mit der <ziffer> 0 beginnt. Eine Folge von <ziffern>, die mit den <zeichen> 0x oder 0X beginnt, wird als <hexadezimal_konstante> interpretiert. Beginnt die <integer_konstante> mit einer von 0 verschiedenen <ziffer>, dann handelt es sich um eine <dezimal_konstante>.

<integer_konstanten> werden behandelt, als wären sie vom <typ> int. übersteigt jedoch der Wert einer <integer_konstanten> den, der mittels 16 Bits in Zweierkomplementdarstellung ausgedrückt werden kann, wird diese <konstante> automatisch als eine <long_integer_konstante> interpretiert. Explizit kann eine <long_integer_konstante> notiert werden, indem an die betreffende Zifferfolge der <buchstabe> L oder l angefügt wird. <long_integer_konstanten> werden behandelt, als wären sie vom <typ> long int.

```
        <long_integer_konstante> :
                  <integer_konstante>L>
                  <integer_konstante>l>
```

~~~~~~
2.4.2. Gleitkomma-Konstante
~~~~~~~~~~~~~~~~~~~~~~~~~~

Eine <gleitkomma_konstante> (Real-Konstante) besteht aus einem ganzzahligen Teil, einem Dezimalpunkt, einem echten Dezimalbruch und einem e oder E, dem wahlweise ein vorzeichenbehafteter <exponent> folgen kann. Ganzzahliger Teil , Dezimalbruch und <exponent> bestehen aus einer Folge von <ziffern>. Einige der genannten Bestandteile können gemäß den Syntaxregeln weggelassen werden.

Jede <gleitkomma_konstante> wird so behandelt, als wäre sie vom <typ> double.

```
        <gleitkomma_konstante> :
                  <dezimalzahl>.<dezimalzahl>[<exponent>]
                  <dezimalzahl>.[<exponent>]
                  <dezimalzahl><exponent>
                  .<dezimalzahl>[<exponent>]

        <exponent> :
                  e[+|-]<dezimalzahl>
                  E[+|-]<dezimalzahl>

        <dezimalzahl> :
                  <ziffer>{<ziffer>}
```

2.4.3. Zeichen-Konstante
~~~~~~~~~~~~~~~~~~~~~~~~

Eine <zeichen_konstante> ist ein in <apostrophe> eingeschlossenes <zeichen>. Ihr Wert ist der numerische Wert des <zeichens> im Zeichensatz der jeweiligen Maschine. <zeichen_konstanten> werden behandelt, als wären sie vom <typ> char.

Folgende Steuerzeichen sowie Bitkombinationen können durch die hier angegebenen Ersatzzeichenfolgen, die sogenannten <nicht_grafischen_zeichen>, notiert werden:

```
          \0        NUL, Nullzeichen
          \n        Newline NL (LF)
          \t        Tabulator HT
          \b        Backspace BS, Rücktaste
          \r        Carriage return CR, Wagenrücklauf
          \f        Formfeed FF, Seitenvorschub
          \\        Backslash, Fluchtsymbol
          \'        Apostroph
          \<n>    Bitmuster, dabei ist <n> eine Folge
                    von ein bis drei <oktalziffern>
```

In allen Fällen wird immer nur ein <zeichen> dargestellt. Folgt dem <zeichen> in einer <zeichen_konstante> ein anderes <zeichen>, als oben angegeben wurde, wird das <zeichen> Backslash ignoriert.

```
        <zeichen_konstante> :
                  '<buchstabe>'
                  '<ziffer>'
                  '<sonderzeichen>'
                  '<nichtgrafisches_zeichen>'
                  '<anführungszeichen>'
```
~~~~~~~~~~~~~~~~~~~~~~~~

2.4.4. Zeichenkettenkonstanten

Eine <zeichenketten_konstante> ist eine in <anführungszeichen> eingeschlossene Zeichenfolge. Der Compiler erzeugt ein eindimensionales Feld von <zeichen> und schließt dieses Feld mit \0 ab. Dieses abschließende <zeichen> mit dem Wert 0 kann für Endetests benutzt werden. Eine <zeichenketten_konstante> ist also immer um eins größer, als die Anzahl der angebenen <zeichen>.

Die <zeichenketten_konstante> hat die <speicherklasse> static. Sie wird mit den angegebenen <zeichen> initialisiert. Alle <zeichenketten_konstanten>, auch wenn sie identisch sind, erhalten jede für sich erneut Speicherplatz.
Ein <anführungszeichen> als Bestandteil einer <zeichenketten_konstante> muß mit \" angegeben werden. Steuerzeichen und Bitkombinationen können durch dieselben Ersatzzeichenfolgen dargestellt werden wie sie oben bei <zeichen_konstanten> beschrieben wurden. Ein \ und ein unmittelbar folgendes NL, die in der <zeichenketten_konstante> angegeben wurden, werden ignoriert.

```
        <zeichen_kettenkonstante> :
                  "{<zeichen>}"

        <zeichen> :
                  <buchstabe>
                  <ziffer>
                  <sonderzeichen>
                  <apostroph>
                  <nichtgrafisches_zeichen>
```

2.5. Zwischenraum

Als Zwischenraum werden Leerzeichen, Tabulatorzeichen, Zeilentrennzeichen und <kommentare> gewertet. C-Programme können in einem freien Format notiert werden. Die Zwischenräume werden vom Compiler ignoriert, außer wenn sie benachbarte < bezeichner>, <konstanten>, <schlüsselworte> und <operatoren> trennen. Bei der lexikalen Analyse wird durch den C-Compiler das längstmögliche Morphem unter Beachtung der Zwischenräume und der Art der Zeichen gebildet.

```
        <zwischenraum> :
                  \t |   | \n | <kommentar>
```

~~~~~~
2.6. Kommentar
~~~~~~~~~~~~~~~

Ein <kommentar> beginnt mit /* und wird mit */

abgeschlossen. <kommentare> können nicht ineinander verschachtelt werden.

```
        <kommentar> :
                  /* {zeichen} */
```

~~~~~~
3. Datentypen
~~~~~~~~~~~~~

Daten sind die Größen, die als Werte von Operanden auftreten und durch Operatoren verknüpft werden.
Die Daten, gleichgültig ob es sich um Konstanten handelt oder um Variable, besitzen stets einen bestimmten womit die Art der Repräsentation im Speicher und der darstellbare Wertebereich festgelegt ist.

Die Notierung der <konstanten> sowie der ihnen entsprechende <typ> sind in Abschnitt 2.4. erläutert.
Variable müssen vereinbart werden, wobei ihnen - implizit oder explizit - ein <typ> zugewiesen wird (siehe Abschnitt 8.2. und folgende).

üblicherweise repräsentiert eine Variable einen Speicherplatz, dem während der Programmbeabeitung unterschiedliche Werte zugewiesen werden können. Da in C <bezeichner> nicht immer Variable bezeichnen und da neben der Wertzuweisung noch andere Operationen zur Modifizierung von Speicherbereichen vorhanden sind, wird im folgenden für Variable der Begrif Objekt benutzt.

Ein Objekt ist ein Speicherbereich, der modifiziert werden kann und dessen Inhalt entsprechend dem zugeordneten <typ> zu interpretieren ist.

3.1. Elementare Datentypen
~~~~~~~~~~~~~~~~~~~~~~~~~~

Objekte des Zeichen-<typs> char können jedes beliebige Zeichen des implementierten Zeichensatzes beinhalten. Wird ein echtes Zeichen aus diesem Satz in einem char-Objekt abgespeichert, so entspricht sein Wert dem Integer-Code für dieses Zeichen. In char-Objekten können auch andere Größen abgelegt werden, die Implementation ist jedoch maschinenabhängig.

Ganzzahlige Objekte, die im weiteren Integer genannt werden, gibt es in folgenden <typen>: short int, int und long int. Entsprechend der speziellen Implementation und der vorhandenen Hardware können entweder short int oder long int oder beide äquivalent zu int sein.

Mit Angabe eines Integer-<typs> und vorangestelltem unsigned werden vorzeichenlose Integer-Objekte vereinbart. Sie entsprechen den Regeln der Arithmetik modulo 2 hoch n, wobei n die Bitanzahl der Repräsentation im Speicher ist.

Real-Objekte (Gleitkomma-Variablen) einfacher Genauigkeit werden durch das <schlüsselwort> float und die doppelter Genauigkeit durch das <schlüsselwort> double charakterisiert. Bei einigen Implementationen können Real-Objekte einfacher und doppelter Genauigkeit identisch repräsentiert werden.

Die <typen> char und int aller Größen werden zusammen <integer_typ> und die <typen> float und double werden <real_typen> genannt.

Die bisher angegebenen <typen> werden zusammenfassend als <arithmetische typen> bezeichnet.
~~~~~~~~~~~~~~~~~~~~~~~~~~

3.2. Abgeleitete Datentypen

Neben den elementaren Daten-<typen> gibt es eine unbegrenzte Anzahl von abgeleiteten <typen>, die aus elementaren und abgeleiteten <typen> auf folgende Art strukturiert werden können:

Felder (arrays, auch Vektoren genannt) bestehen aus Objekten gleichen <typs>.
Funktionen liefern Funktion nach ihrer Ausführung Objekte eines bestimmten <typs>.
Zeiger (pointer) zeigen auf Objekte eines festgelegten <typs> (Der Zeiger ist an den <typ> gebunden).
Strukturen (structures) enthalten eine Folge von Objekten, die möglicherweise verschiedene <typen> haben
Varianten (unions) enthalten jeweils ein Objekt aus einer Reihe von verschiedenen <typen>.
Aufzählungen (enumerations) definieren explizit eine Menge von int-Werten.

Diese Methoden zur Konstruktion von Objekten können rekursiv angewendet werden.

4. Speicherklassen

Der <bezeichner> von Objekten steht für den zugeordneten Speicherbereich. In C hängt die Bedeutung eines solchen <bezeichners> von zwei Attributen ab, vom <typ> und von der <speicherklasse>. Die beiden Attribute werden entweder explizit im Programm angegeben oder aus dem Kontext des Programms gefolgert.

Der <typ> bestimmt die Bedeutung des Wertes des im Speicherbereich des <bezeichners> vorgefundenen Inhalts (siehe Abschnitt 3.).

Die <speicherklasse> bestimmt den Ort und die "Lebensdauer" des Speicherbereiches, der durch den <bezeichner> symbolisch bezeichnet wird und so mit ihm verbunden ist.

Es gibt die vier <speicherklassen> auto, static, register und extern, die vereinbart werden können. Objekte der <speicherklasse> auto sind lokal bei Eintritt in einen <block> (siehe Abschnitt 9.2.) vorhanden und werden beim Austritt beseitigt. Die auto-Objekte werden bei jedem Eintritt in den <block> erneut eingerichtet. (Sie werden auf dem Stack geführt.) Die Objekte der <speicherklasse> static sind ebenfalls lokal für einen <block>, sie haben jedoch beim Wiedereintritt in den <block> noch die Werte, die sie beim Verlassen des <blocks> hatten.

Objekte der <speicherklasse> extern existieren während der gesamten Programmausführung und behalten auch während dieser Zeit ihre Werte. Sie können zur Kommunikation zwischen Funktionen, insbesondere zwischen getrennt compilierten Funktionen verwendet werden. Die Objekte der <speicherklasse> register werden, wenn es möglich ist, in den Registern der Maschine gespeichert. Sie sind ähnlich den auto-Objekten lokal zu jedem Block und verschwinden beim Austritt aus dem Block. Spezielle Register können jedoch explizit nicht zugewiesen werden.

5. Bezugnahme auf Objekte

Der Zugrif zu Objekten, also zu modifizierbaren Speicherbereichen, soll zunächst an der Wertzuweisung, wie sie auch in anderen Programmiersprachen zu finden ist, näher betrachtet werden.

Bei einer gültigen Wertzuweisung der Form E1 = E2 sind auf der rechten Seite <konstanten> ebenso möglich wie die Werte von Objekten, die aber nur "gelesen", nicht verändert werden.

Der Wert des rechten <ausdrucks> wird berechnet und mit dem Zuweisungsoperator = dem linken <ausdruck> E1 zugewiesen. Diese Zuweisung hat nur Sinn, wenn der linke <ausdruck> nach seiner Auswertung ein Objekt bezeichnet, dem der Wert zugewiesen werden kann bzw. das mit dem Wert der rechten Seite "beschrieben" werden kann.

Diese Eigenschaft, nämlich ein bestimmtes Objekt auszuwählen, ist beim linken <ausdruck> unbedingt gefordert, während sie beim rechten <ausdruck> vorhanden sein kann oder nicht. Ein <ausdruck> mit dieser Eigenschaft wird daher als <lvalue> bezeichnet (= left value, Wert der linken Seite).

In der Programmiersprache C wird verallgemeinert unter <lvalue> ein <bezeichner> oder ein <einfacher_ausdruck> verstanden, der auf ein Objekt Bezug nimmt.

Bei der Benutzung eines Operators ist stets zubeachten, ob als Operand ein <lvalue> gefordert ist. Beispielsweise verlangen die Zuweisungsoperatoren <ass_op> auf ihrer linken Seite einen <lvalue>. Ebenso erwarten Inkrement- und Dekrementoperatoren <inc_op> sowie der Adreßoperator & einen <lvalue> als Operanden.

Ferner ist zu beachten, ob der Wert, den ein Operator liefert, einen <lvalue> darstellt, der entsprechend weiterbenutzt werden kann. Es gibt Operatoren, die <lvalues> liefern; ist beispielsweise E ein <ausdruck> vom <typ> Zeiger, so ist *E ein <lvalue>, der Bezug auf das Objekt nimmt, auf welches E zeigt.

Bei der Diskussion der einzelnen Operatoren im Abschnitt 7. wird immer angegeben, ob der spezielle Operator einen <lvalue>-Operanden erwartet und ob er einen <lvalue> liefert.

6. Typkonvertierung

Bestimmte Operatoren können in Abhängigkeit von ihren Operanden implizit Konvertierungen der Werte der Operanden von einem <typ> in einen anderen veranlassen.

Im folgenden wird dargelegt, welches Resultat diese Konvertierungen haben. Besonderheiten der Konvertierung durch Operationen werden im Abschnitt 7. erklärt.

6.1. Zeichen und Integer

Ein char-, short- oder long-Objekt kann überall anstelle eines Integer-Objekts angegeben werden. In allen Fällen wird der Wert in einen Integer-Wert konvertiert.

Bei Konvertierungen von short int in long int bleibt das Vorzeichen erhalten.

Die Konvertierung von char-Werten in Integer-Werte ist maschinenabhängig. Es wird gesichert, daß ein Zeichen aus dem Standard-Zeichensatz "nicht negativ" wird. Objekte vom <typ> char können auf dem AC A7100/A7150 Werte im Bereich von -128 bis 127 darstellen. Wurde eine <zeichen_konstante> durch eine Ersatzzeichenfolge angegeben, kann sie negativ erscheinen. Zum Beispiel erhält '/377' den Wert -1.

Wird ein längerer Integer-Wert in einen kürzeren oder in char konvertiert, so werden die höherwertigen Bits abgeschnitten.

6.2. Real-Werte einfacher und doppelter Genauigkeit

Die gesamte Real-Arithmetik (Gleitkomma-Arithmetik) wird in C mit doppelter Genauigkeit ausgeführt. Erscheint ein float-Objekt in einem <ausdruck>, so wird sein Wert auf double erweitert, indem sein gebrochener Teil mit Nullen aufgefüllt wird. Muß ein double-Wert in float konvertiert werden, z.B. bei einer Zuweisung, so wird der double-Wert gerundet, bevor er auf float-Länge beschnitten wird.

6.3. Real- und Integer-Werte

Konvertierungen von Real- in Integer-Werte sind maschinenabhängig. Insbesondere variiert die Verfahrensweise bei der Konvertierung negativer Zahlen von Maschine zu Maschine. Wenn ein ganzzahliger Real-Wert nicht als Integer-Wert repräsentiert werden kann, ist das Ergebnis undefiniert.

Werden Werte vom <integer_typ> in <real_typen> konvertiert, kann eine Anzahl von signifikanten Bits verloren gehen, wenn der Ergebnis-<typ> nicht über genügend Bits verfügt. Beispielsweise kann die Mantisse eines float-Wertes zu klein sein, um einen long_int-Wert aufzunehmen.

6.4. Zeiger- und Integer-Werte

Zeiger- und Integer-Werte können addiert oder subtrahiert werden. Der Integer-Wert wird in Einheiten des <typs> umgerechnet, der an den Zeiger gebunden ist (siehe Abschnitt 7.4.). Das Ergebnis ist ein Zeiger-Wert.

Zwei Zeiger auf Objekte gleichen <typs> können subtrahiert werden. In diesem Fall wird das Ergebnis in eine Integer-Zahl konvertiert (siehe Abschnitt 7.4.).

6.5. Vorzeichenlose Typen (unsigned)

Wird ein Objekt vom <integer_typ> mit einem unsigned-Objekt verknüpft, wird der Integer-Wert in unsigned konvertiert. Das Ergebnis dieser Konvertierung ist die kleinste positive ganze Zahl, die kongruent zum ursprünglichen Integer-Wert (modulo 2 hoch Bitanzahl) ist. Diese Konvertierung entspricht dem Vorgehen bei der Zweierkomplementdarstellung.

Wird ein unsigned short-Wert in einen long int-Wert konvertiert, bleibt der numerische Wert gleich. Damit beläuft sich die Konvertierung nur auf ein Auffüllen mit Nullen im höherwertigen Wort.

6.6. Konvertierung bei arithmetischen Operationen

Bei den meisten Operatoren werden die gleichen Konvertierungsregeln angewendet.

Zunächst werden Operanden vom <typ> char oder short in den <typ> int und Operanden des <typs> float in double konvertiert.

Danach findet nur dann eine Konvertierung statt, wenn beide Operanden einen unterschiedlichen <typ> besitzen. Die Konvertierung erfolgt so, daß an den höheren <typ> angepaßt wird. Die Reihenfolge in diesem Sinne ist: double, long, unsigned. Besitzt einer der Operanden den <typ> long und der andere den <typ> double, so wird der long-Operand zu double konvertiert.

Wenn keiner der zuvor genannten Fälle vorliegt, sind beide Operanden vom <typ> int und das Ergebnis ist ebenfalls ein int-Wert.

Die gerade erläuterten Konvertierungen heißen arithmetische Konvertierungen.

7. Ausdrücke und ihre Operanden

Die <ausdrücke> bestehen aus Operanden und Operatoren. Die Operanden sind Variable, <konstanten> oder wiederum <ausdrücke>. Die Auswertung jedes <ausdrucks> liefert einen Wert, der sich aus der Verknüpfung von Operanden durch Operatoren ergibt.

Die folgenden Unterabschnitte sind entsprechend dem Vorrang der Operatoren in <ausdrücken> angeordnet, höchster Vorrang zuerst. Innerhalb jedes Unterabschnittes besitzen die Operatoren gleiches Vorrangniveau.

Links- oder Rechtsassoziativität des Operators wird im betreffenden Unterabschnitt angegeben.

Vorrang und Assoziativität aller <ausdrucks>-Operatoren sind in der Tabelle `Operatoren' zusammengefaßt. Die Reihenfolge der Berechnungen in den <ausdrücken> ist abgesehen vom Vorrang undefiniert. Insbesondere behält sich der Compiler vor, Teilausdrücke - auch wenn sie Nebeneffekte verursachen - in einer aus seiner Sicht effektiven Weise für die Berechnung zu übersetzen. Die Reihenfolge, mit der diese Nebeneffekte auftreten, ist nicht definiert! Die <ausdrücke>, die einen kommutativen oder assoziativen Operator enthalten (z.B. *, +, &, |, ^), können vom Compiler beliebig umgestellt werden, sogar bei vorhandenen Klammern. Soll eine bestimmte Reihenfolge der Berechnung eingehalten werden, muß der <ausdruck> explizit in Teilschritte unter Benutzung von Hilfsvariablen zerlegt werden.

Die Behandlung von überlauf und die Kontrolle der Division in <ausdrücken> sind maschinenabhängig.

Alle existierenden C-Implementationen ignorieren Integerüberläufe.

Die Behandlung von "Division durch Null" und die von Gleitkomma-Ausnahme-Bedingungen variert von Maschine zu Maschine und ist anwenderseitig durch Bibliotheksfunktionen anpaßbar.

7.1. Einfache Ausdrücke

Mit einem <einfachen_ausdruck> wird auf eine Variable, auf eine <konstante>, auf eine Komponente einer Struktur oder einer Variante (siehe auch Abschnitt 8.5.) oder auf ein Element eines Feldes (siehe auch Abschnitt 14.3.) zugegrifen, sowie eine Funktion aufgerufen und zu ihrem Ergebnis zugegrifen. <einfache_ausdrücke> enthalten die Operatoren . , -> , [ ] und ( ). Die genannten Operatoren sind linksassoziativ.

```
        <einfacher_ausdruck> :
                  <bezeichner>
                | <konstante>
                | (<ausdruck>)
                | <einfacher_ausdruck>([<argument_liste>])
                | <einfacher_ausdruck [<ausdruck>]
                | <lvalue>.<bezeichner>
                | <einfacher_ausdruck>-><bezeichner>

        <lvalue> :
                  <bezeichner>
```

~~~~~~
                | <einfacher_ausdruck>[<ausdruck>]
                | <lvalue>.<bezeichner>
                | <einfacher_ausdruck>-><bezeichner>
                | *<lvalue>
                | (<lvalue>)

Bezeichner
~~~~~~~~~~
Ein <bezeichner> ist ein <einfacher_ausdruck>, vorausgesetzt, der <bezeichner> wurde geeignet vereinbart. Der <typ> der durch den <bezeichner> benannten Variable, wird bei der Vereinbarung spezifiziert.
Ist der <bezeichner> der eines Feldes, so ist der Wert des nur aus dem <bezeichner> bestehenden <einfachen_ausdrucks> eine Adreßkonstante, nämlich ein Zeiger, der auf das erste Element im Feld zeigt (siehe Abschnitt 14.3.). Der <bezeichner> eines Feldes ist konstant und daher kein <lvalue>. ähnlich wird ein <bezeichner> einer Funktion, wenn er nicht zum Aufruf der Funktion verwendet wird, als Zeiger auf die Funktion verstanden, und stellt keinen <lvalue> dar. Der Operator & ist in beiden Fällen nicht notwendig.
Der <bezeichner> einer Struktur ist ein <lvalue>. Der <bezeichner>, der ein <enum_bezeichner> ist, ist konstant und damit kein <lvalue>.

Konstante
~~~~~~~~~
Eine <konstante> ist ein <einfacher_ausdruck>. Sie hat den <typ> int, long oder double. Die <zeichen_konstanten> haben den <typ> int, <gleitkomma_konstanten> den <typ> double.

Eine <zeichenketten_konstante> ist ein <einfacher_ausdruck>. Sie hat den <typ> eines Feldes, das aus <zeichen> besteht und ist somit ein `Zeiger auf char' (bestimmte Initialisierungen bilden eine Ausnahme).

Klammerausdruck (.)
~~~~~~~~~~~~~~~~~~~
Ein in runden Klammern eingeklammerter <ausdruck> ist ein <einfacher_ausdruck>. Sein <typ> und sein Wert sind die des in den Klammern stehenden <ausdrucks>. Die Klammern haben keinen Einfluß darauf, ob der <ausdruck> ein <lvalue> ist oder nicht.

Feldindex
~~~~~~~~~
Ein <einfacher_ausdruck>, dem ein <ausdruck> in eckigen Klammern folgt, ist insgesamt ein <einfacher_ausdruck>. Der <ausdruck> in eckigen Klammern besitzt die Bedeutung eines Index'. Mit diesem Index wird ein Element in einem Feld ausgewählt. Praktisch besitzt dieser <einfache_ausdruck> den Typ "Zeiger auf x", wobei "x" für einen <typ> steht. Der <typ> des <ausdrucks>, der als Index verwendet wird, muß int sein. Die Benutzung des beschriebenen <einfachen_ausdrucks> liefert einen Wert, der den <typ> "x" des Feldelementes hat. Der <ausdruck> E1[E2] ist identisch mit dem <ausdruck> *((E1)+(E2)) und ist ein <lvalue>.

Funktionsaufruf
~~~~~~~~~~~~~~~
Ein Funktionsaufruf besteht aus einem <einfachen_ausdruck>, dem in runden Klammern eine <argument_liste>, die möglicherweise leer

~~~~~~
ist, folgt. Die <argument_liste> enthält die durch Kommas getrennten aktuellen Parameter, also die Argumente, für die Funktion. Der <einfache_ausdruck> muß eine Funktion bezeichnen. Das Ergebnis des Funktionsaufrufs hat den <typ>, mit dem die Funktion vereinbart wurde. Ein Funktionsaufruf ist kein <lvalue>.
Tritt in einem Programmm ein nicht deklarierter <bezeichner> auf, dem unmittelbar eine linke Klammer folgt, wird dies als ein Funktionsaufruf betrachtet, der einen Wert vom <typ> int ergibt. Aus diesem Grund braucht der <typ> von Funktionen, die einen Funktionswert vom <typ> int ergeben, nicht explizit deklariert zu werden.
Die aktuellen Parameter werden vor dem Aufruf der Funktion konvertiert, und zwar die vom <typ> float in double, die vom <typ> char oder short in int und die Feld-<bezeichner> in Zeiger. Andere Konvertierungen werden nicht automatisch ausgeführt, sie müssen explizit mittels cast angewiesen werden. Der Compiler vergleicht die <typen> der aktuellen und formalen Parameter nicht miteinander! Vor einem Funktionsaufruf wird jeder aktuelle Parameter, der der Funktion übergeben wird, kopiert (auf den Stack), so daß alle Argumentübergaben in C generell "by value" sind. Eine Funktion kann den Wert ihrer Parameter nur innerhalb der Funktion ändern, d.h. diese Veränderung bleibt ohne Einfluß auf die Werte der aktuellen Parameter in der rufenden Funktion. Es besteht die Möglichkeit, als Parameter Zeiger an die Funktion zu übergeben. In diesem Fall kann der Wert des Objektes, auf welches der Zeiger zeigt, verändert werden.
Die Reihenfolge der Berechnung der Argumente ist durch die Sprache nicht definiert.
Der rekursive Aufruf von Funktionen ist gestattet.

Identifikator von Strukturen und Varianten
~~~~~~
Ein <lvalue>, dem ein Punkt . und ein <bezeichner> folgt, ist ein <einfacher_ausdruck>. Der <lvalue> muß eine Struktur (struct) oder Variante (union) bezeichnen und der <bezeichner> muß eine Komponente der Struktur bzw. Variante benennen. Der so gebildete <einfache_ausdruck> ist ein <lvalue>.

Einem <einfachen_ausdruck>, dem die Zeichenkombination -> und ein <bezeichner> folgt, ist ein <einfacher_ausdruck>. Der <einfache_ausdruck> muß ein Zeiger auf eine Struktur oder Variante sein, der <bezeichner> muß eine Komponente davon benennen. Der <einfache_ausdruck> ergibt einen <lvalue>, der die bezeichnete Komponente liefert.
Der <einfache_ausdruck> E1->bez ist äquivalent zu ((*E1).bez).

Die <bezeichner> der Komponenten von Strukturen und Varianten werden nicht auf Verträglichkeit überprüft. Es besteht daher die Möglichkeit, mit einem Komponenten<bezeichner>, der in einer bestimmten Strukturdefinition festgelegt wurde, auf Komponenten in anderen Strukturen oder Varianten zuzugreifen. Wenn die Position (der Offset) zweier Komponenten innerhalb verschiedener Strukturen oder Varianten gleich ist, können sie gleiche <bezeichner> haben.

7.2. Unäre Operatoren

Die <ausdrücke> bzw. <konst_ausdrücke, die mit unären Operatoren gebildet werden können, sind nachfolgend aufgeführt. <konst_ausdrücke> gehören zur Klasse der <ausdrücke>.

Treten in einem <ausdruck> mehrere unäre Operatoren auf gleichem Niveau auf, werden sie von rechts nach links bearbeitet (Rechtsassoziativität).

```
        <ausdruck> :
                  !<ausdruck>
                | ~<ausdruck>
                | <inc_op> <lvalue>
                | <lvalue> <inc_op>
                | -<ausdruck>
                | (<typ_spezifikation>)<ausdruck>
                | &<lvalue>
                | *<ausdruck>

        <konst_ausdruck> :
                  sizeof <ausdruck>
                | sizeof (<typ_spezifikation>)

        <inc_op> :
                  ++ | --
```

Operandenadresse

Die unären Operatoren * und & sind Zeigeroperatoren. Der Operator & liefert die Adresse des Operanden, der ein <lvalue> sein muß. Wenn der Operand den <typ> "xyz" hat, dann hat das Ergebnis der Operation dem <typ> "Zeiger auf xyz".

Adreßbezug

Mit Hilfe des unären Operators * wird eine Adreßbezugnahme aufgelöst (Entreferenzierung) und auf das Datenobjekt, auf das der Operand verweist, "indirekt" zugegrifen. Die Operation ergibt einen <lvalue>. Wenn der Operand den <typ> "Zeiger auf xyz" besitzt, hat das Ergebnis den <typ> "xyz".

Arithmetische Negation

Der unäre Operator "Operator Arithmetische Negation" - liefert den negativen Wert seines Operanden. Hat der Operand den <typ> unsigned, dann wird sein Wert von 2 hoch n subtrahiert. Dabei ist n die Anzahl der Bits, die bei der Darstellung einer Zahl vom <typ> int im Speicher belegt werden. Ein unäres + gibt es nicht.

Logische Negation

Das Ergebnis des logischen Negations-Operators ! ist gleich 1, wenn der Wert seines Operanden gleich Null ist. Das Ergebnis ist gleich Null, wenn der Wert des Operanden ungleich Null ist. Der <typ> des Ergebnisses ist int. Der Operator ist auf jeden <arithmetischen_typ> und auf Zeiger anwendbar.

~~~~~~
Einerkomplement
~~~~~~~~~~~~~~~
Der Operator ~ liefert das Einer-Komplement seines Operanden. Es werden die üblichen arithmetischen Konvertierungen ausgeführt. Der Operand muß vom <integer_typ> sein.

Inkrementierung und Dekrementierung
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Die <inc_op>-Operatoren ++ und -- dienen der Inkrementierung bzw. Dekrementierung. Die <ausdrücke>, die mit diesen beiden Operatoren gebildet werden können, sind:

```
        <inc_op> <lvalue>        /* Präfixnotation */
        <lvalue> <inc_op>        /* Postfixnotation */
```

Beide Operatoren können nur auf <lvalues> angewendet werden, d.h. auf Operanden, die sich auf einen modifizierbaren Speicherbereich beziehen.

Der Wert des mit dem Operator in Präfixnotation gebildeten <ausdrucks> ist der Wert des Operanden nach der Inkrementierung bzw. Dekrementierung. Im Gegensatz dazu ist der Wert des <ausdrucks> bei der Postfixnotation gleich dem Wert des Operanden vor Ausführung der Operation.

Der <typ> des Ergebnisses ist gleich dem des Operanden. Das Ergebnis ist kein <lvalue>.

Konvertierung
~~~~~~~~~~~~~
Eine in runden Klammern stehende <typ_spezifikation> vor einem <ausdruck> bewirkt die Konvertierung des <ausdrucks> in den durch die <typ_spezifikation> explizit angegebenen <typ>. Diese Konstruktion heißt cast. Sie entspricht praktisch einer Wertzuweisung der Form

```
        <lvalue> =  <ausdruck>
```

wobei <lvalue> ein anonymes Datenobjekt von dem <typ> bezeichnet, den die <typ_spezifikation> angibt.

Operandenlaenge
~~~~~~~~~~~~~~~
Der Operator sizeof liefert die Größe seines Operanden, gemessen in Bytes.

Der mit sizeof gebildete <ausdruck> ist ein <konst_ausdruck> (Konstanten-Ausdruck), der zur Compilationszeit ausgewertet wird. Bytes werden durch die Sprache nur als Einheiten von sizeof definiert. Der Wert von sizeof (char) ist gleich 1. Wird sizeof auf ein Feld angewandt, so ist das Ergebnis die Länge des gesamten Feldes. Der Operator wird hauptsächlich zur Kommunikation mit Routinen für die Speicherverwaltung und für die Ein- und Ausgabe verwendet.
~~~~~~~~~~~~~~~

7.3. Multiplikation und Division

Die multiplikativen Operatoren <mul_op> sind:

```
<mul_op> :  * | / | %
```

Die Operatoren für Multiplikation und Division sind linksassoziativ. Auf die Operanden werden die üblichen arithmetischen Konvertierungen angewendet.

```
<ausdruck> :  <ausdruck> <mul_op> <ausdruck>
```

Der binäre Operator * ist der Multiplikations-Operator. Er ist assoziativ und kommutativ. Durch den Compiler können auf gleichem Vorrangniveau stehende Multiplikations-Operatoren umgeordnet werden.

Der binäre Operator / ist der Divisions-Operator. Bei der Division von Integer-Werten wird ein möglicher Rest "abgeschnitten". Ist einer der Operanden negativ, ist die Behandlung des Restes maschinenabhängig. Es wird in jedem Fall gewährleistet, daß (a/b)*b+a%b=a ist, unter der Voraussetzung, daß b nicht 0 ist sowie a und b Integer sind. Beispiel: (5/3)*3+5%3=5, da 5/3=1 und 5%3=2.

Der binäre Operator % liefert den Rest, der bei der Division des ersten <ausdrucks> durch den zweiten entsteht. Real-Operanden sind nicht zulässig.

7.4. Addition und Subtraktion

```
<add_op> :  + | -
```

Die additiven Operatoren <add_op> sind + und -. Die additiven Operatoren sind rechtsassoziativ. Die <ausdrücke> haben die Form:

```
<ausdruck> :  <ausdruck> <add_op> <ausdruck>
```

Das Ergebnis des Operators + ist die Summe der Operanden. Ein Zeiger auf ein Feldelement und Integer-Werte können addiert werden. Der Integer-Wert wird vor der Addition mit der Länge des Elementes multipliziert. Das Ergebnis der Addition ist ein Zeiger auf das Element im Feld (Zeigerarithmetik).
Beispiel: Ist z ein Zeiger auf ein Feldelement eines beliebigen <typs>, so zeigt das Ergebnis von z+1 auf das nächste Feldelement. Andere Operanden-<typ>-kombinationen, als die hier angebenen, sind unzulässig.

Der Operator + ist assoziativ und kommutativ. Der Compiler legt die Berechnungsreihenfolge fest.

Das Ergebnis des Operators - ist die Diferenz seiner Operanden. Die üblichen arithmetischen Konvertierungen werden ausgeführt. Von einem Zeiger kann ein Integer-Wert subtrahiert werden. Der Integer-Wert wird in Einheiten des <typs> gerechnet, der an den Zeiger gebunden ist bzw. durch einen cast erzwungen wird. Werden zwei Zeiger auf Objekte gleichen <typs> subtrahiert, so wird das Ergebnis in einen Integer-Wert konvertiert. (Durch Division durch die Länge des Objektes). Der Integer-Wert gibt die Anzahl der Ob-

jekte an, um die die beiden Objekte, auf welche die Zeiger zeigen, getrennt sind. Bei dieser Konvertierung können jedoch unerwartete Ergebnisse geliefert werden, wenn z.B. die Zeiger nicht auf Objekte im gleichen Feld zeigen.

7.5. Verschiebungs-Operatoren

Die Verschiebungs-Operatoren <shift_op> << und >> sind linksassoziativ. Die arithmetischen Konvertierungen werden ausgeführt.

```
<ausdruck> :  <ausdruck> <shift_op> <ausdruck>

<shift_op> :  << | >>
```

Beide Operanden müssen vom <integer_typ> sein. Der rechte Operand wird zu int konvertiert. Das Ergebnis der Verschiebungsoperation hat den <typ> des linken Operanden.

Das Ergebnis ist undefiniert, wenn der rechte Operand negativ ist oder sein Wert größer oder gleich der Länge des linken Operanden (in Bits) ist. Der Wert von E1<<E2 ergibt sich aus E1, welches als Bitmuster interpretiert wird, nach links um E2-Bits verschoben. Dabei wird von rechts mit Nullen aufgefüllt.

Der Wert der Rechtsverschiebung E1>>E2 ist das Bitmuster von E1 um E2 Bit-Positionen nach rechts verschoben. Ist E1 vom <typ> unsigned, so erfolgt eine logische Rechtsverschiebung, d.h. von links wird mit 0 aufgefüllt. Falls E1 nicht vom <typ> unsigned ist, kann in Abhängigkeit von der Maschine eine arithmetische Verschiebung stattfinden, wobei das Vorzeichenbit auf seiner Position erhalten bleibt. Beim AC A7100/A7150 findet eine arithmetische Verschiebung in solch einem Fall statt.

7.6. Vergleichs-Operatoren

Die Relationalen-Operatoren <rel_op> werden in Vergleichs-Operatoren <vgl_op> und äquivalenz-Vergleichs-Operatoren <äqu_op> untergliedert (siehe auch Abschnitt 7.7.). Die Relationalen-Operatoren sind linksassoziativ. Man überlege sich, was z.B. E1<E2<E3bedeutet, und erkennt, daß sich diese Linksassoziativität nicht besonders nutzen läßt.

```
<ausdruck> :  <ausdruck> <vgl_op> <ausdruck>

<rel_op> :    <vgl_op> | <äqu_op>

<vgl_op> :    < | <= | > | >=
```

Die Operatoren

```
<     kleiner als
>     größer als
<=    kleiner als oder gleich
>=    größer als oder gleich
```

liefern alle den Wert Null vom <typ> int, wenn die angegebene Relation falsch ist. Ist die Relation wahr, dann wird der int-Wert 1 geliefert. Die üblichen arithmetischen Konvertierungen werden ausgeführt. (In C gibt es keinen <typ> Boolean!)

Zwei Zeiger können miteinander verglichen werden. Das Ergebnis hängt von der relativen Anordnung der Objekte im Adreßraum ab, auf die sie zeigen. Der Zeiger-Vergleich ist nur portabel, wenn die Zeiger auf Objekte im gleichen Feld zeigen.

7.7. äquivalenz-Vergleichs-Operatoren

```
        <äqu_op> :  == | !!!=
```

Die beiden Operatoren

```
        ==  gleich
        !!!=  ungleich
```

besitzen einen niedrigeren Vorrang als die Vergleichs-Operatoren aus dem Abschnitt 7.6. Zum Beispiel liefert A1<A2 == A3<A4 genau dann den Wert 1, wenn die beiden Vergleiche A1<A2 und A3<A4 denselben Wert liefern.

Ein Zeiger kann zwar mit einem Integer-Wert auf äquivalenz verglichen werden, aber das Ergebnis ist maschinenabhängig, falls der Integer-Wert nicht 0 ist. Ein Zeiger, dem Null zugewiesen wurde, zeigt in keinem Fall auf irgendein Objekt.

```
        <ausdruck> :
                  <ausdruck> == <ausdruck>
                | <ausdruck> !!!=       <ausdruck>
```

7.8. Bitorientierter UND-Operator

```
        <ausdruck> :  <ausdruck> & <ausdruck>
```

Der &-Operator ist assoziativ und kommutativ. Die <ausdrücke>, die ihn enthalten, können durch den Compiler umgeordnet werden. Es werden die üblichen arithmetischen Konvertierungen ausgeführt. Die Bits der beiden Operanden werden in jeder Position einzeln mit UND verknüpft (Konjunktion). Der Operator ist nur auf Operanden vom <integer_typ> anwendbar.

7.9. Bitorientierter Exklusiv-ODER-Operator

```
        <ausdruck> :  <ausdruck> ^^^ <ausdruck>
```

Der ^-Operator ist assoziativ und kommutativ. Die <ausdrücke>, die ihn enthalten, können durch den Compiler umgeordnet werden. Es werden die üblichen arithmetischen Konvertierungen ausgeführt. Die Bits der beiden Operanden werden in jeder Position einzeln mit dem Exklusiv-ODER (Antivalenz, ausschließendes ODER) verknüpft. Der Operator ist nur auf Operanden vom <integer_typ> anwendbar.

7.10. Bitorientierter Inklusiv-ODER-Operator

```
        <ausdruck> :  <ausdruck> | <ausdruck>
```

Der |-Operator ist assoziativ und kommutativ. Die <ausdrücke>, die ihn enthalten, können durch den Compiler umgeordnet werden. Es werden die üblichen arithmetischen Konvertierungen ausgeführt. Die Bits der beiden Operanden werden in jeder Position einzeln mit dem Inklusiv-ODER (Disjunktion, einschließendes ODER) verknüpft. Der Operator ist nur auf Operanden vom <integer_typ> anwendbar.

7.11. Logischer UND-Operator

```
        <ausdruck> :  <ausdruck> && <ausdruck>
```

Der &&-Operator ist linksassoziativ. Er liefert als Ergebnis 1, wenn beide Operanden ungleich Null sind, anderenfalls liefert er das Ergebnis Null. Im Gegensatz zu & wird beim &&-Operator garantiert, daß eine Bearbeitung von links nach rechts erfolgt. Der rechte Operand wird nur dann ausgewertet, wenn der linke Operand nicht 0 ist. (Einsparung von Operationen!)

Die Operanden brauchen nicht vom selben <typ> zu sein, sie müssen aber mit 0 vergleichbar sein. Das Ergebnis ist immer vom <typ> int.

7.12. Logischer ODER-Operator

```
        <ausdruck> :  <ausdruck> | <ausdruck>
```

Der logische Oder-Operator | ergibt 1, wenn einer der Operanden ungleich Null ist. Anderenfalls ergibt dieser Operator Null. Im Gegensatz zum |-Operator garantiert der |-Operator eine Berechnungsfolge von links nach rechts. Der rechte Operand wird nicht berechnet, wenn der linke Operand 0 ergibt. (Einsparung von Operationen!)

Die Operanden können verschiedene <typen> haben, müssen aber beide mit 0 vergleichbar sein. Das Ergebnis des Logischen-ODER-Operators ist immer vom <typ> int.

7.13. Bedingter Ausdruck

Der durch folgende Syntaxegel angegebene <ausdruck> ist ein Bedingter-Ausdruck:

```
        <ausdruck> :  <ausdruck> ? <ausdruck> : <ausdruck>
```

In Abhängigkeit vom ersten <ausdruck>, der die Bedingung darstellt, wählt der Entscheidungsoperator ?: aus den beiden weiteren <ausdrücken> den zu berechnenden aus. Ist der erste <ausdruck>, d.h. der links vom Fragezeichen stehende, ungleich Null, so wird der zweite <ausdruck> berechnet und sein Wert stellt das Ergebnis des gesamten bedingten Ausdrucks dar. Anderenfalls wird der dritte <ausdruck> berechnet.

Die üblichen arithmetischen Konvertierungen werden ausgeführt, damit der zweite und der dritte <ausdruck> denselben <typ> haben. Sind beide <ausdrücke> Zeiger, müssen sie auf Objekte gleichen

<typs> verweisen. Das Ergebnis des Bedingten-Ausdrucks hat in diesem Fall diesen <typ> der Objekte. Wenn der zweite oder dritte <ausdruck> ein Zeiger ist und der andere davon die Konstante 0, dann hat das Ergebnis den <typ> des Zeigers.

7.14. Zuweisungs-Operatoren

Es gibt eine Vielzahl von Zuweisungs-Operatoren, die rechtsassoziativ sind.

```
        <ausdruck> :  <lvalue> <ass_op> <ausdruck>

        <ass_op> :
                       =     | <mul_op>= | <add_op>=
                | <bit_op>= | <shift_op>=
```

Als linker Operand ist ein <lvalue> erforderlich. Der <typ> eines Zuweisungs-Ausdrucks ist der seines linken Operanden. Der Wert des Zuweisungs-Ausdrucks ergibt sich aus dem, der im linken Operanden nach erfolgter Zuweisung abgespeichert ist.

Bei der einfachen Zuweisung mit dem Operator = ersetzt der Wert des <ausdrucks> den des Objektes, auf welches durch den <lvalue> Bezug genommen wird.

Besitzen beide Operanden <arithmetischen_typ>, so wird vor der Zuweisung der rechte Operand in den <typ> des linken konvertiert.

Das Verhalten eines Ausdrucks der Form E1 <op>= E2 ist dem <ausdruck> E1 = E1 <op> E2 äquivalent. E1 wird nur einmal berechnet.

Wenn der linke Operand von += oder -= ein Zeiger ist, kann der rechte Operand vom <integer_typ> sein. Der rechte Operand wird wie im Abschnitt 7.4 beschrieben konvertiert.

Alle rechten und linken Operanden, die keine Zeiger sind, müssen vom <arithmetischen_typ> sein.

Zur Zeit gestatten die Compiler, daß einer Integer-Variablen ein Zeiger, eine Integer-Variable einem Zeiger und ein Zeiger einem Zeiger anderen <typs>, zugewiesen wird. Die Zuweisungs-Operation stellt dabei ein reines Kopieren ohne Typkonvertierung dar. Diese Anwendung ist jedoch nicht portabel und kann Zeiger ergeben, die eine Adreß-Ausnahmebedingung verursachen, wenn sie verwendet werden. Es wird jedoch garantiert, daß die Zuweisung der Konstanten 0 zu einem Zeiger einen Null-Zeiger ergibt, der sich von allen anderen Zeigern unterscheidet.

Mehrfache Zuweisungen sind möglich, wenn der linke Operand jeder Zuweisung ein <lvalue> ist.

7.15. Komma-Operator

```
        <ausdruck> :  <ausdruck>, <ausdruck>
```

Werden <ausdrücke> durch Komma getrennt, werden sie von links nach rechts berechnet, wobei der Wert des linken <ausdrucks> nicht weiter verwendet wird. Der <typ> und der Wert des Resultates entsprechen denen des rechten Operanden. Die Operation ist linksassoziativ.

In Kontexten, in denen das Komma eine spezielle Bedeutung hat, beispielsweise in einer Liste von Parametern für Funktionen oder einer Liste von Initialisierungen, kann der Komma-Operator nur in Klammern erscheinen.

Beispiel: Der Funktionsaufruf

```
        f(a, (t=3, t+2), c)
```

besitzt drei Argumente: a, 5 und c.

8. Vereinbarungen

Um die Eigenschaften, die ein <bezeichner> in einem <programm> haben soll, festzulegen, werden in C <externe_vereinbarungen> (siehe auch Abschnitt 10.) und <interne_vereinbarungen> verwendet.

```
        <externe_vereinbarung> :
                  <funktions_definition>
                | <daten_definition>
                | <extern_deklaration>
                | <esu_typ_definition>
                | <typ_definition>

        <interne_vereinbarung> :
                  <funktions_deklaration>
                | <interne_daten_definition>
                | <extern_deklaration>
                | <esu_typ_definition>
                | <typ_definition>
```

Unter Vereinbarungen werden Definitionen und Deklarationen verstanden. Deklarationen dienen der Spezifikation der Eigenschaften, die ein <bezeichner> in C hat, und Definitionen sorgen darüberhinaus noch für die Bereitstellung von Speicherplatz, sofern es sich nicht um eine <esu_typ_definition> handelt.

Nachfolgend werden die Syntaxregeln für die interne und externe Definition von Variablen gegenübergestellt.

```
        <daten_definition> :
                  static [<typ>] <init_deklarator_liste>;
                | <typ> <init_deklarator_liste>;

        <interne_daten_definition> :
                  <def_speicherklasse> [<typ>]
                          <init_deklarator_liste>;
                | <typ> <init_deklarator_liste>;
```

Um die Beschreibung zu vereinfachen, kann man eine allgemeinere Form für diese Vereinbarungen angeben:

```
        <attribute> <init_deklarator_liste>
wobei
        <attribute> :
                  [<speicherklasse>] <typ>
                | [<typ>] <speicherklasse>
```

gilt. Die in der zweiten Alternative von <attribute> angegebene Möglichkeit ist in den Syntaxregeln der Einfachheit halber nicht angegeben, ist aber zulässig.
Betrachten wir als nächstes die Syntaxregel für <funktions_kopf>, die bei einer <funktions_definition> benötigt wird,

```
        <funktions_kopf> :
                  [static] [<typ>] <funktions_de-
                          klarator>[<par_deklarations_liste>]
```

so kann man allgemeiner schreiben:

~~~~~~
```
        <funktions_kopf> :
                  <attribute> <funktions_de-
                          klarator> [<par_deklarations_liste>]
```

Stellen wir jetzt die Deklarationen gegenüber.

```
        <extern_deklaration> :
                  extern [<typ>] <deklarator_liste>;
                | extern <funktions_deklaration>;

        <par_deklaration> :
                  register [<typ>] <deklarator_liste>;
                | <typ> <deklarator_liste>;
                | <typ_definition>

        <typ_definition> :
                  typedef [<typ>] <deklarator_liste>;

        <funktions_deklaration> :
                  [<typ>] <funktionsbezeichner>;
```

Man kann für die rechten Regelseiten, die syntaktischen Formeln, allgemeiner schreiben:

```
        <attribute> <deklarator_liste>
        <attribute> <funktions_bezeichner>
```

Mit diesen Vereinfachungen, die nun zwar jeweils bestimmter Einschränkungen bedürfen, kann in den weiteren Abschnitten die Sprachbeschreibung komprimiert werden.

8.1. Speicherklassen-Spezifizierung

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
        <speicherklasse> :
                  extern
                | <def_speicherklasse>
                | typedef

        <def_speicherklasse> :
                  auto
                | static
                | register
```

Zur Vereinfachung wird der Spezifizierer typedef als <speicherklasse> geführt. Er reserviert keinen Speicherplatz (näheres dazu im Abschnitt 8.8.). Die Bedeutung der einzelnen <speicherklassen> wird im Abschnitt 4. beschrieben.

In Definitionen können nur die <def_speicherklassen> auto, static und register verwendet werden. Sie legen die verschiedenen Varianten der Speicherplatzreservierung und deren Verwaltung fest. Das <schlüsselwort> extern kennzeichnet <extern_deklarationen> (keine Definitionen, siehe Abschnitt 10.).

Eine register-Vereinbarung betrachtet man am besten als auto-Vereinbarung jedoch mit dem zusätzlichen Hinweis für den Compiler, daß die so deklarierten Variablen sehr häufig benutzt werden. Die Absicht ist, derartige Variable in den effizienter zugreifbaren Maschinenregistern unterzubringen. (Dieser Behandlung liegen die derzeit verfügbaren Rechnerarchitekturen zugrunde.)

~~~~~~
Der Compiler kann entsprechend der Anzahl der zur Verfügung stehenden Register nur die ersten solcher Vereinbarungen , wie sie im Programmtext stehen, in Registern unterbringen. Die übrigen register-Vereinbarungen bleiben unberücksichtigt. Werden register-Vereinbarungen jedoch geschickt eingesetzt, können kleinere und schnellere Programme erzielt werden.
Es können nur Variablen bestimmter <typen> in Registern gespeichert werden. Für den AC A7100/A7150 sind das die Typen int, char und Zeiger. Eine weitere Einschränkung besteht darin, daß für register-Variablen nicht der Operator & anwendbar ist. In einer Vereinbarung kann nur eine <speicherklasse> angegeben werden. Fehlt die <speicherklasse>, so wird bei Vereinbarungen innerhalb einer Funktion, den <internen_vereinbarungen>, auto, bei Vereinbarungen außerhalb von Funktionen, also den <externen_vereinbarungen>, extern implizit angenommen. Ausnahme: Funktionen selbst haben niemals die <speicherklasse> auto für ihren Funktionswert.

8.2. Typen

~~~~~~~~~~
Es gibt folgende <typen>:

```
        <typ> :
                  <arithmetischer_typ>
                | unsigned
                | unsigned <integer_typ>
                | struct <strukt_deklarator>
                | union <strukt_deklarator>
                | enum <enum_deklarator>
                | <typ_bezeichner>

        <arithmetischer_typ> :
                  <integer_typ>
                | <real_typ>

        <integer_typ> :
                  char
                | int
                | short
                | long
                | short int
                | long int

        <real_typ> :
                  float
                | long float
                | double
```

Die Worte long, short und unsigned können als Adjektive aufgefaßt werden. Dabei hat long float dieselbe Bedeutung wie double.

Es sind nur die hier angegebenen Kombinationen dieser "Adjektive" zugelassen! In jeder Vereinbarung darf nur eine Angabe des <typs> stehen. Fehlt in einer Vereinbarung die Angabe des <typs>, so wird durch den Compiler der <typ> int angenommen. Die Datentypen werden im Abschnitt 3. erklärt. Im Abschnitt 8.5. werden <strukt_deklaratoren>, im Abschnitt 8.8. <typ_definitionen> mit <typ_bezeichnern> und im Abschnitt 8.9. <enum_deklaratoren> diskutiert.

~~~~~~
Ist der <typ> ein <typ_bezeichner>, so wurde er mit einer <typ_definition> vereinbart.

8.3. Deklaratoren

In Deklarationen werden <deklarator_listen> und in <daten_definitionen> bzw. in <interne_daten_definitionen> werden <init_deklaratorlisten> verwendet. In diesen Listen werden die <deklaratoren> bzw. <init_deklaratoren> durch Kommas getrennt.

```
        <deklarator_liste> :
                  <deklarator>
                | <deklarator>, <deklarator_liste>

        <init_deklarator_liste> :
                  <init_deklarator>
                | <init_deklarator>, <init_deklarator_liste>

        <deklarator> :
                  <bezeichner>
                | (<deklarator>)
                | *<deklarator>
                | (*<deklarator>) ()
                | (*<deklarator>) []
                | <feld_deklarator>

        <feld_deklarator> :
                  <bezeichner>[<konst_ausdruck>]

        <init_deklarator> :
                  <deklarator>
                | <deklarator> = <init_wert>
                | <deklarator> = {<init_wertliste>[,]}
```

Eine <extern_deklaration> kann eine <funktions_deklaration> enthalten, die ihrerseits aus einer <typ>-Angabe und einem <funktions_bezeichner> besteht:

```
        <extern_deklaration> :
                  extern [<typ>] <deklarator_liste>;
                | extern <funktions_deklaration>;

        <funktions_deklaration> :
                  [<typ>] <funktionsbezeichner>;

        <funktions_bezeichner> :
                  <bezeichner> ()
                | (<funktions_bezeichner>)
                | *<funktions_bezeichner>
                | (*<funktions_bezeichner>) ()
                | (*<funktions_bezeichner>) []
```

Bei einer <funktions_definition> wird im <funktions_kopf> ein <funktions_deklarator> verwendet:

```
        <funktions_deklarator>:
                  <funktions_name>
                | (<funktions_deklarator>)
                | *<funktions_deklarator>
                | (*<funktions_deklarator>) ()
```
~~~~~~

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                | (*<funktions_deklarator>) [[<konst_ausdruck>]]

        <funktions_name>:
                  <bezeichner>([<parameter_liste>])
```

Bei einem Vergleich von <deklarator>, <funktions_deklarator>, <funktions_bezeichner> und der Syntaxregel für <abstrakter_deklarator>,

```
        <abstrakter_deklarator> :
                  [<abstrakter_deklarator>]
                | (<abstrakter_deklarator>)
                | *<abstrakter_deklarator>
                | <konst_ausdruck>()
                | <abstrakter_deklarator>[[<konst_ausdruck>]]
```

erkennt man den ähnlichen syntaktischen Aufbau. Aus diesem Grunde werden solche Deklaratoren im folgenden gemeinsam beschrieben.

Die Operatoren in den Syntaxregeln für Deklaratoren haben denselben Vorrang und dieselbe Assoziativität wie in <ausdrücken>, d.h. Klammern binden stärker als der Operator *.

Die Initialisierung mittels der <init_wert_liste> wird im Abschnitt 8.6. diskutiert.

8.4. Bedeutung der Deklaratoren

Jeder <deklarator>, <funktions_bezeichner> und <funktions_deklarator>, im Folgenden kurz Deklarator genannt, wird als eine Informationsquelle angesehen. Erscheint in einem <ausdruck> eine Konstruktion, die denselben Aufbau wie der Deklarator hat, so liefert der <ausdruck> ein Objekt. Dieses Objekt wird in einer Vereinbarung, in der der Deklarator Bestandteil ist, vereinbart. In der Vereinbarung werden für das betreffende Objekt als <attribute> der <typ> und die <speicherklasse> explizit oder implizit angegeben.

Nachfolgend werden die verschiedenen Alternativen, in denen ein Deklarator in Vereinbarungen auftritt, diskutiert. Die <speicherklasse> der entsprechenden Vereinbarung wird hier nicht behandelt (siehe Abschnitt 8.2). In einer Vereinbarung gibt es unter anderen folgende Bestandteile:

```
        <typ> <deklarator>
```

Ist der <deklarator> ein einfacher <bezeichner>, so besitzt dieser <bezeichner> den <typ>, der in der Vereinbarung angegeben wurde. Steht ein <funktions_bezeichner> oder <funktions_deklarator> anstatt des <deklarators>, so gibt der <typ> den <typ> des von der Funktion gelieferten Ergebnisses, des Funktionswertes, an.

Ein Deklarator in Klammern beeinflußt die Vorränge bei der Bildung komplexerer Deklaratoren.

Mit der Konstruktion

```
        <typ> *<deklarator>
```

~~~~~~
wird angegeben, daß der durch den <deklarator> angegebene <bezeichner> die Bedeutung Zeiger auf <typ> hat (Zeigervereinbarung). Wenn anstelle des <deklarators> ein <funktions_bezeichner> oder <funktions_deklarator> steht, liefert die betreffende Funktion einen Zeiger auf den angegebenen <typ>.

Die Konstruktion

```
<typ> <deklarator> ()
```

vereinbart, daß der durch den <deklarator> angegebene <bezeichner> ein <bezeichner> eines <funktions_namens> ist. Die Funktion liefert als Ergebnis einen Wert des angegebenen <typs>.

Die Konstruktionen

```
<typ> <bezeichner>[<konst_ausdruck>]
<typ> <bezeichner>[]
```

vereinbaren den <bezeichner> als Feld von <typ>. Der <konst_ausdruck> ergibt während der Compilierung eine Konstante. Sie gibt die Dimension des eindimensionalen Feldes an. Soll das Feld mehrdimensional sein, sind mehrere <konst_ausdrücke> in eckigen Klammern anzugeben (vgl. Syntaxregel für <feld_deklarator>). Zwischen den eckigen Klammern wird kein <konst_ausdruck> angegeben, wenn die Konstruktion in einer Deklaration auftritt oder durch eine Initialisierung alle Feldelemente angegeben werden. Ist ein mehrdimensionales Feld in einer Deklaration zu spezifizieren, darf nur der die erste Dimension angebende <konst_ausdruck> weggelassen werden. Die eckigen Klammern müssen angegeben werden.
Ein Feld kann aus Elementen gebildet werden, die einen Grunddaten-<typ> haben, die den <typ> Zeiger oder Struktur oder Variante oder Feld haben. Lezteres wird verwendet, um mehrdimensionale Felder zu bilden.

Nicht alle Möglichkeiten, die die Syntax formal erlauben, sind zugelassen. Es gibt folgende Einschränkungen:
Funktionen können keine Felder, Varianten oder Funktionen zurückliefern. Es ist also falsch, wenn der <bezeichner> eines <funktions_bezeichners> oder eines <funktions_deklarators> ohne die möglichen umgebenden Zeichen Klammer oder Stern auftritt. Funktionen können Strukturen liefern. Zeiger auf Variablen dieser <typen> einschließlich Felder können jedoch zurückgeliefert werden. Es gibt keine `Felder von Funktionen', aber `Felder von Zeigern auf Funktionen' sind zugelassen.
Analog dazu können Strukturen oder Varianten keine Funktionen als Komponenten enthalten, jedoch Zeiger auf Funktionen sind gestattet.

Beispiele:

```
        int i,      /* Integer-Variable iii */
        *iz,        /* Zeiger iz auf eine Integer-Variable */
        f(),        /* Funktion fff, die einen Integer-Wert
                       zurückliefert */
        *fiz(),     /* Funktion fiz, die einen Zeiger auf eine
                       Integer-Variable zurückliefert */
        (*zfi)();   /* Zeiger zfi auf eine Funktion,
                       die einen Integerwert zurückliefert */
```
~~~~~~

~~~~~~
Die letzten beiden Deklarationen sollen etwas genauer betrachtet werden: Die Konstruktion *fiz() ist der Konstruktion *(fiz()) äquivalent. Die Form der Deklaration soll darauf hinweisen, daß zunächst die Funktion fiz aufgerufen wird, und dann indirekt auf das Funktionsergebnis, das vom <typ> sein soll, zugegrifen wird. Bei dem <deklarator> (*zfi)() sind alle Klammern notwendig. Er bedeutet, daß zfi ein Zeiger ist, der auf eine Funktion zeigt, die nach ihrem Aufruf einen Integerwert liefert.

Das nächste Beispiel deklariert ein Feld feld, das aus 21 Real-Variablen besteht und ein Feld zeigerfeld, das aus 17 `Zeigern auf Real-Variable' besteht.

```
        float feld[21], *zeigerfeld[17];
```

In dem abschließenden Beispiel wird ein dreidimensionales Feld, das als Elemente Integervariable haben soll mit dem Format 3 X 5 X 4 vereinbart:

```
        static int xyz3dim[3][5][4];
```

Das Feld xyz3dim hat die <speicherklasse> static. Jeder der folgenden <ausdrücke> kann sinnvoll sein: xyz3dim, xyz3dim[i], xyz3dim[i][j] und xyz3dim[i][j][k]. Der lezte <ausdruck> hat den <typ> int während die vorangegangenen den <typ> `Feld' haben.

8.5. Strukturen und Varianten

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Strukturen und Varianten sind strukturierte <typen>.
Ein Objekt, das den <typ> einer noch näher zu spezifizierenden Struktur hat, faßt Objekte unterschiedlichen <typs> zusammen. In der Mathematik spricht man vom n-tupel, in der Programmiersprache PASCAL vom record.
In der Praxis hat sich herausgestellt, das es manchmal zweckmäßig ist, mehrere <typen> als Varianten eines <typs> zu betrachten. Ein einfaches Beispiel ist, einen <typ> `koordinate' zu definieren, der die Varianten `kartesische_koordinate' und `polare_koordinate' vereinigt. Bei einer Variante werden mehrere <typen> auf demselben Speicherplatz überlagert. Man nennt eine Variante auch union oder Vereinigung. In der Programmiersprache PASCAL nennt man diesen <typ> varianten record.

Die Vereinbarung von Struktur- und Varianten-<typen> ist durch die <schlüsselworte> struct bzw. union gekennzeichnet,

```
        strukt <strukt_deklarator>
        union  <strukt_deklarator>
```

Man erkennt, daß abgesehen vom charakterisierenden <schlüsselwort> Strukturen und Varianten syntaktisch gleich aufgebaut werden.

```
        <strukt_deklarator> :
                  [<strukt_typ_bezeichner>]
                            {<element_deklarations_liste>[;]}
                 | <strukt_typ_bezeichner>
```

~~~~~~

```
        <strukt_typ_bezeichner> :
                  <bezeichner>

        <element_deklarations_liste> :
                  <element_deklaration>
                | <element_deklaration> <ele-
                           ment_deklarations_liste>

        <element_deklaration> :
                  <typ> <element_deklarator_liste>;

        <element_deklarator_liste> :
                  <element_deklarator>
                | <element_deklarator>, <ele-
                           ment_deklarator_liste>

        <element_deklarator> :
                  <deklarator>
                | [<deklarator>] : <konst_ausdruck>
```

Der Name eines bestimmten Struktur-<typs> ist der <strukt_typ_bezeichner>. Die Komponenten, die auch Elemente genannt werden, stehen in der <element_deklarations_liste>, durch Semikolons getrennt und durch geschweifte Klammern umschlossen. Eine <element_deklaration> gibt den <typ> der nachfolgend angegebenen <element_deklaratoren> an, also letztlich die <bezeichner> der Komponenten. Die Komponenten dürfen beliebige <typen> haben.

Bevor eine Variable mit dem <typ> einer Struktur oder Variante deklariert werden kann, muß dieser <typ> definiert werden. Dies kann entweder bei der Definition einer Variablen in einer <daten_definition> oder einer <internen_daten_definiton> geschehen oder völlig selbständig. Das Folgende ist ein Beispiel für eine Vereinbarung der Struktur kartesisch. Damit wird lediglich ein <typ> deklariert; es wird keine Variable definiert und also auch kein Speicherplatz zugewiesen! Wir bezeichnen solch eine Konstruktion als <esu_typ_definition>. Sie wird mit einem Semikolon abgeschlossen!

```
        struct kartesisch {
                float x;
                float y;
        };
```

Der <typ> `Struktur kartesisch' hat den <strukt_typ_bezeichner> kartesisch. Er besteht aus zwei Komponenten, die in der <element_deklarations_liste> angegeben sind. Beide <element_deklarationen> geben den <typ> float an. Die <bezeichner> der <deklaratoren> sind x und y. Da beide Komponenten denselben <typ> haben, kann man obige Strukturvereinbarung vereinfachen, indem nur eine <element_deklarations_liste> verwendet wird,

```
        struct kartesisch {
                float x, y;
        };
```

Die <element_deklaratoren> können auch sogenannte Bitfelder sein. Sie werden in diesem Zusammenhang durch einen Doppelpunkt gekenn-
~~~~~~

~~~~~~
zeichnet, zweite Alternative in der Syntaxregel für <element_deklarator>. Ein Bitfeld besteht aus der explizit durch den <konst_ausdruck> angegebenen Anzahl von Bits.

Innerhalb einer Struktur werden die deklarierten Komponenten sequentiell angeordnet. Dabei erfolgt eine Ausrichtung der Komponenten, die nicht Bitfelder oder vom <typ> char sind, entsprechend ihrem <typ> an Wortgrenzen. Komponenten vom <typ> char werden paarig in den Maschinenworten plaziert. Bei diesem Ausrichten können "Löcher" in der Struktur bzw. Variante entstehen, die nicht bezeichnet sind (keinen <bezeichner> haben). Die Bitfelder werden in int-Variablen (die dem Maschinenwortumfang entsprechen) plaziert. Ein Bitfeld kann nicht größer als eine int-Variable sein. Werden in einer Struktur bzw. Variante mehrere Bitfelder angegeben, so werden sie hintereinander in einer int-Variable untergebracht. Reicht eine bereits teilweise mit Bitfeldern gefüllt int-Variable nicht aus, um ein weiteres Bitfeld, weil es zu lang ist, aufzunehmen, wird eine neue int-Variable eröffnet. Ob die Bitfelder in einem Wort von links nach rechts oder umgekehrt angeordnet werden, ist maschinenabhängig. Beim AC A7100/A7150 werden die Bitfelder in einer int-Variablen von rechts nach links angeordnet.
Ein nichtbezeichnetes Bitfeld, d.h. im <element_deklarator> wird nur der Doppelpunkt und der <konst_ausdruck> angegeben, kann verwendet werden, um die Struktur für irgendwelche Zwecke aufzufüllen und auszurichten. Ist der <konst_ausdruck> gleich 0, so kann damit auf die nächste Wortgrenze ausgerichtet werden. Wenn eine Komponente, die kein Bitfeld ist, angegeben wird, geschieht das, wie bereits ausgeführt wurde, automatisch.

In einer <element_deklaration> für Bitfelder können die <typen> int und unsigned verwendet werden. Felder, die aus Bitfeldern bestehen, sind nicht zulässig. Der Operator & darf nicht auf Bitfelder angewendet werden. Zeiger auf Bitfelder gibt es nicht.

Eine Variante sorgt dafür, das Speicherplatz verschieden interpretiert werden kann. Die größte Komponente einer Variante bestimmt den Umfang des reservierten Speicherplatzes. Alle Komponenten der Variante werden auf diesem gleich großen Speicherplatzbereich untergebracht und beginnen bei derselben gedachten relativen Adresse Null. Die Komponenten der Variante deklarieren wie die Variante selbst nur einen damit bestimmten <typ>, keinen Speicherplatz! Die Reservierung von Speicherplatz erfolgt erst bei der Definition einer Variablen dieses <typs>!

Ein Beispiel für eine <typ>-Vereinbarung ist die Vereinbarung der `Variante koordinate', deren Komponenten die Strukturen kartesisch und polar sind. Die `Struktur polar' bestimmt die Größe des vorzusehenden Speicherplatzes.

```
        struct kartesisch {
                float x, y;
        };

        struct polar {
                double r;
                float phi;
        };
```
~~~~~~

~~~~~~

```
        union koordinate {
                struct kartesisch k;
                struct polar p;
        };
```

Unter Verwendung der zweiten Alternative der Syntaxregel <strukt-
_deklarator> kann man für <typ> folgende Konstruktionen bilden:

```
        struct <strukt_typ_bezeichner>
        union <strukt_typ_bezeichner>
```

Mit dieser Konstruktion wird in einer Vereinbarung auf den durch den <strukt_typ_bezeichner> benannten strukturierten <typ> und dessen Definition Bezug genommen. Zum Beispiel definiert man mit

```
        struct polar poko;
```

eine Variable mit dem <bezeichner> poko, die den <typ> `Struktur polar' hat. Man bezeichnet solch eine Variable auch als eine Instanz des angegebenen <typs>. Der <typ> `Struktur polar' muß zuvor definiert worden sein! Man kann die Definition der Variablen mit der des strukturierten <typs> auf folgende Weisen verbinden:

```
        struct polar {
                double r;
                float phi;
        } poko;
```

oder

```
        struct {
                double r;
                float phi;
        } poko;
```

Bei der zweiten Variablenvereinbarung gibt es keinen <strukt_typ_bezeichner>, so daß man sich später nicht wieder auf diesen Struktur-<typ> beziehen kann.

Eine Struktur oder Variante darf sich nicht selbst als Komponente enthalten, aber Zeiger auf gleiche Strukturen oder Varianten sind zugelassen.

Beispiel:

```
        struct baumknoten {
                char knoteninf[20];
                int knoten_nr;
                struct baumknoten *linker;
                struct baumknoten *rechter;
        };
```

Der Struktur-<typ> baumknoten besteht aus vier Komponenten, einem Zeichenfeld knoteninf, einer Integervariablen knotenn_nr und zwei Zeigern, linker und rechter, die auf eine Struktur gleichen <typs> zeigen. Mit

```
        struct baumknoten k, *zk;
```

werden zwei Variable definiert, nämlich eine Strukturvariable des <typs> `Struktur baumknoten' k und eine Zeigervariable zk, die
~~~~~~

~~~~~~
auf eine derartige Variable zeigen kann. Mit

```
        zk->knoten_nr
```

wird die Komponente knoten_nr der Variablen bezeichnet, auf die zk zeigt. Mit

```
        k.rechter
```

wird ein Zeiger auf eine Struktur des hier behandelten <typs> bezeichnet und mit

```
        k.rechter->knoteninf[5]
```

wird das fünfte Zeichen im Feld knoteninf des rechten Unterknotens von kkk erreicht.

Der <strukt_typ_bezeichner> und der <bezeichner> einer Variablen des durch den <strukt_typ_bezeichner> angegebenen Struktur- bzw. Varianten-<typs> können gleich sein! Desweiteren ist eine Gleichheit zwischen den <bezeichnern> von Komponenten und sonstigen Variablen möglich. Der <bezeichner> einer Komponente darf keinem <strukt_typ_bezeichner> gleich sein. Die <bezeichner> von Komponenten in verschiedenen Strukturen dürfen nur dann gleich sein, wenn sie in den betreffenden Strukturen dieselbe relative Position (Offset, Verschiebung) angeben. Dies ist gegeben, wenn z.B. bei zwei Strukturen die <element_deklarations_listen> in ihrem Anfangsteil gleich sind. (Die <bezeichner>-Gleichheit ist möglich, da die <bezeichner> für Variable in einer anderen Symboltabelle als die für <strukt_typ_bezeichner> und Komponenten-<bezeichnern> vom Compiler geführt werden.)

8.6. Daten-Initialisierung

Unter Initialisierung eines Datenobjektes versteht man die implizite oder explizite Zuweisung eines Anfangswertes bei der Definition. Diese Definitionen sind die <daten_definitionen> und die <internen_daten_definitionen>. Die explizit zugewiesenen Anfangswerte sind die <init_werte>.

```
        <init_deklarator> :
                  <deklarator
                | <deklarator> = <init_wert>
                | <deklarator> = {<init_wertliste>[,]}

        <init_wert_liste> :
                  <init_wert>
                | <init_wert>, <init_wert_liste>

        <init_wert> :
                  <ausdruck>
                | {<init_wert_liste>[,]}
```

Dem <init_wert> bzw. der in geschweiften Klammern stehenden <init_wert_liste> steht das Zeichen = voran.

Extern definierte Daten und Daten der <speicherklasse> static können nur mit konstanten Ausdrücken, den <konst_ausdrücken>, explizit initialisiert werden, oder die <ausdrücke> enthalten eine Adresse einer vorher vereinbarten Variable (Operator &), die mög-
~~~~~~

licherweise noch mit einer Integer- oder Zeichenkonstante additiv verknüpft sein kann. Diese Initialisierungen finden zur Compile-Zeit statt.
Variable der <speicherklassen> static und extern werden, wenn keine <init_werte> angegeben sind, implizit mit 0 initialisiert.

Variable der <speicherklasse> auto oder register können mit beliebigen <ausdrücken> initialisiert werden.
Werden für diese Variablen keine <init_werte> angegeben, sind die Anfangswerte dieser Variablen undefiniert.

Soll ein Objekt, das vom <arithmetischen_typ> ist oder ein Zeiger ist, initialisiert werden, so wird der <init_wert> aus einem einzelnen <ausdruck> gebildet. Dieser <ausdruck> kann mit oder ohne geschweifte Klammern angegeben werden. Es werden dieselben Konvertierungen wie bei Zuweisungen ausgeführt.

Ist das zu initialisierende Objekt eine Struktur oder ein Feld, so werden die die <init_werte> ergebenden <ausdrücke> bzw. <init_wert_listen> durch Kommas getrennt angegeben. Die <init_wert_liste> kann in geschweiften Klammern stehen. Die Reihenfolge der <init_werte> entspricht der Reihenfolge der Strukturkomponenten bzw. der Feldelemente. Die Verwendung einer <init_wert_liste> als <init_wert> ergibt sich, wenn in der zu initialisierenden Struktur eine Komponente ebenfalls eine Struktur oder ein Feld ist. Dieses Vorgehen ist rekursiv. Werden weniger <init_werte> angegeben als die Struktur Komponenten bzw. das Feld Elemente hat, werden die restlichen Komponenten bzw. Elemente mit 0 initialisiert.

Objekte eines strukturierten <typs>, also Strukturen, Felder und Varianten, der <speicherklasse> auto und Varianten überhaupt, können nicht initialisiert werden. Bitfelder können nicht initialisiert werden.

Zur Verwendung der geschweiften Klammern: Wenn die <init_wert_liste> mit { beginnt und wenn mehr <init_werte> angegeben werden als eine Struktur Komponenten oder ein Feld Elemente hat, ist dies falsch.
Hat das zu initialisierende Objekt einen strukturierten Datentyp und haben die Komponenten zum Teil ebenfalls einen strukturierten Datentyp, und erhalten bei der Initialisierung alle Komponenten einen <init_wert>, so können die geschweiften Klammern innerhalb der <init_wert_liste> weggelassen werden. Felder werden zeilenweise abgespeichert.

Ein Zeichenfeld (char-Feld) kann anstatt aufwendig durch <zeichen_konstanten> durch eine <zeichenketten_konstante> initialisiert werden.

Ein Beispiel für die Definition eines Feldes a mit vier Elementen vom <typ> int, die initialisiert werden, ist:

```
int a [] = {1, 5, 7, 9};
```

Auf Grund der vollständigen Angabe aller <init_werte>, braucht in den eckigen Klammern kein <konst_ausdruck> als Dimension des Feldes angegeben werden. Die geschweiften Klammern können hier ebenfalls weggelassen werden. Infolge der Initialisierung haben die Feldelemente folgende Werte: a[0] = 1, a[1] = 5, a[2] = 7,

~~~~~~
a[3] = 9.

Der Zweck der geschweiften Klammern in der <init_wert_liste> soll an folgendem Beispiel gezeigt werden.

```
        int b[3][2] = {
                {7, 8},
                {4},
                {2}
        };
```

Das Ergebnis der Initialisierung ist: b[0][0] = 7, b[0][1] = 8, b[1][0] = 4, b[2][0] = 2 und implizit b[1][1] = 0 und b[2][1] = 0. Die obige <init_wert_liste> spiegelt den zeilenweisen Aufbau des Feldes wieder. In der zweiten und dritten Zeile fehlt jeweils der zweite <init_wert> und damit ergibt sich die Zuordnung von 0.

Ein Beispiel für eine vollständig geklammerte <init_wert_liste> ist:

```
        int c[4][3]={
                {1,3,5}, {2,4,6}, {3,5,7},
        };
```

Hierbei initialisieren 1, 3, 5 die erste Zeile des Feldes y[0], nämlich y[0][0], y[0][1], y[0][2]. Analog werden die nächsten zwei Zeilen y[1] und y[2] initialisiert. Da die <init_wert_liste> nicht alle Elemente explizit initialisiert, wird y[3] implizit mit 0 initialisiert.
Der gleiche Effekt wird durch folgendes erreicht:

```
        int y[4][3] = {
                1,3,5,2,4,6,3,5,7
                };
```

Die <init_wert_liste> für y beginnt zwar mit einer geschweiften Klammer, nicht aber die für y[0]. Deshalb werden 3 <init_werte> aus der <init_wert_liste> genommen. Analoges gilt für die <init-_werte> von y[1] und y[2].

Abschließend ein Beispiel für die Initialisierung eines Zeichenfeldes:

```
        char z[] = {'E', 'N', 'D', 'E', '\0'};
        z[] = "ENDE";
```

Beide Zeichenfelder bestehen aus 5 Zeichen. Sie sind wertmäßig gleich. Die zweite Form ist schreibökonomischer als die erste.
~~~~~~

8.7. Typ-Spezifikation

Eine <typ_spezifikation> wird zum Angeben einer expliziten Konvertierung, dem cast, und als Argument von sizeof benötigt (siehe Abschnitt 7.2.).

Die <typ_spezifikation> hat im Prinzip die Form einer Deklaration, bei der der <bezeichner> des Objektes weggelassen wird.

```
        <typ_spezifikation> :
                  <typ> <abstrakter_deklarator>

        <abstrakter_deklarator> :
                  [<abstrakter_deklarator>]
                | (<abstrakter_deklarator>)
                | *<abstrakter_deklarator>
                | <konst_ausdruck>()
                | <abstrakter_deklarator>[[<konst_ausdruck>]]
```

Um Mehrdeutigkeiten zu vermeiden, darf in der zweiten oben angegebenen Alternative

```
        (<abstrakter_deklarator>)
```

der <abstrakte_deklarator> nicht leer sein (was nach der ersten Alternative in der Syntaxregel für <typ_spezifikation> formal möglich ist). Diese Einschränkung ist notwendig, um <abstrakte_deklaratoren> von anderen Deklarationen unterscheiden zu können.

Die Ähnlichkeit von Deklarationen und <typ_spezifikationen> wird an folgenden Beispielen gezeigt:

Deklarationen:

```
        int a;            /* int */
        int *b;           /* Zeiger auf int */
        struct datum *c;  /* Zeiger auf Strukturtyp datum */
        float *d[4];      /* Feld von 4 Zeigern auf
                             float-Werte */
        float (*e)[4];    /* Zeiger auf Feld von 4
                             float-Elementen */
        int *f();         /* Funktion, die einen Zeiger auf int
                             liefert */
        int (*g)();       /* Zeiger auf eine Funktion,
                             die einen int-Wert liefert */
```

und die entsprechenden <typ_spezifikationen>:

```
        int
        int *
        struct datum *
        float *[4]
        float (*)[4]
        int *()
        int (*)()
```

8.8. Typ-Definition und <esu_typ_definition>

Eine <typ_definition> wie auch eine <esu_typ_definition> wird als eine <externe_vereinbarung> oder <interne_vereinbarung> verwendet.

```
        <typ_definition> :
                  typedef [<typ>] <deklarator_liste>;

        <esu_typ_definition> :
                  enum <enum_typ_bezeichner>
                          {<enumerator_liste>[,]};
                | struct <strukt_typ_bezeichner>
                          {<element_deklarations_liste>[;]};
                | union <strukt_typ_bezeichner>
                          {<element_deklarations_liste>[;]};
```

Bei einer <typ_definition> ist der <bezeichner>, der leztlich als <deklarator> in der <deklarator_liste> steht, als Synonym für den <typ> eingeführt. Dieser <bezeichner> wird <typ_bezeichner> genannt. Die <typ_definition> ist für beliebige Datentypen einschließlich Zeiger erlaubt. Mit einer <typ_definition> wird kein Speicherplatz reserviert und auch kein neuer <typ> definiert!

Innerhalb des Gültigkeitsbereiches der <typ_definition> kann der definierte <bezeichner> anstatt des angegebenen <typs> und der sich aus der Form des <deklarators> ergebenden <typs> verwendet werden. Es empfiehlt sich, die in <typ_definitionen> verwendeten <bezeichner> mit großen <buchstaben> zu schreiben, um den Charakter des Synonyms zu unterstreichen. Die durch eine <typ_definition> eingeführten <bezeichner> sollen die Lesbarkeit von C-Programmen verbessern. Die Portabilität von Programmen kann durch <typ_definitionen> unterstüzt werden, indem maschinenabhängige <typen> mittels typedef notiert werden. Bei der Portierung sind in so geschriebenen Programmen nur die <typ_definitionen> an einer Stelle zu ändern.

Beispiel: Mit den angegebenen <typ_definitionen>

```
        typedef int KILOMETER, *ZAI;
        typedef struct  { double re, im;} complex;
```

stellen folgende Konstruktionen legale Deklarationen dar:

```
 KILOMETER entfernung;    /* Der <typ> ist int */
 extern ZAI zeigint,      /* Der <typ> ist "zeiger auf int" */
 complex z,*zp;           /* Der <typ> von z ist "Struktur
                           complex" und der <typ> von zp ist
                           "Zeiger auf die Struktur complex" */
```

Den mittels typedef eingeführten <bezeichnern> dürfen nicht die "Adjektive" short, long oder unsigned vorangestellt werden.

Unter Verwendung obiger <typ_definition> für KILOMETER ist es also falsch, wenn man schreibt:

```
        long KILOMETER von_erde_zum_mond;
```

Richtig muß es heißen:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        typedef long int GROSSE_ENTFERNUNG;
        GROSSE_ENTFERNUNG von_erde_zum_mond;
```

Mit einer <esu_typ_definition> wird der <typ> einer Aufzählung (siehe 8.9.), einer Struktur oder einer Variante (siehe 8.6.) definiert, ohne dabei eine Variable dieses <typs> zu definieren und somit auch keinen Speicherplatz zu reservieren.

8.9. Aufzählungen

In C besteht die Möglichkeit einen einfachen, unstrukturierten Daten-<typ> durch Aufzählung von <bezeichnern>, den <enum_bezeichnern>, zu definieren:

```
        <typ> :
                  enum <enum_deklarator>

        <enum_deklarator> :
                  [<enum_typ_bezeichner>] {<enumerator_liste>[,]}
                | <enum_typ_bezeichner>

        <enum_typ_bezeichner> :
                  <bezeichner>

        <enumerator_liste> :
                  <enumerator>
                | <enumerator>, <enumerator_liste>

        <enumerator> :
                  <enum_bezeichner> [= <konst_ausdruck>]

        <enum_bezeichner> :
                  <bezeichner>
```

Eine durch Aufzählung definierte Menge hat als Namen den durch den <enum_typ_bezeichner> festgelegten <bezeichner>. Die Elemente dieser Menge werden in der <enumerator_liste>, die in geschweiften Klammern eingeschlossen ist, notiert. Die Elemente sind die <enum_bezeichner>, die mit einem <konst_ausdruck> initialisiert werden können. Diese Elemente werden wie konstante Integerwerte behandelt. Sie werden auch <enum_konstanten> genannt. Erfolgt keine Initialisierung durch einen <konst_ausdruck>, so haben die <enumeratoren> von links nach rechts die Werte, die bei Null beginnen und jeweils um 1 vergrößert sind. Eine Initialisierung wird als Anfangswertzuordnung solcher Werte genutzt. Es besteht die Möglichkeit, mehreren <enum_bezeichner> denselben Wert des <konst_ausdrucks> zuzuordnen, auch kann durch die Initialisierung die Reihenfolge der Werte beliebig sein. Eine Ordnungsrelation besteht zwischen den <enumeratoren> nicht. (Daher gibt es auch keine Funktion zur Bestimmung des Vorgängers oder Nachfolgers.) Es ist in C nicht möglich, eine Schleife über die Werte einer Aufzählungsvariable zu schreiben, da sie nicht eine lineare Folge zu bilden brauchen. Im Gegensatz zu Strukturen ist keine <bezeichner>-Gleichheit zwischen <enum_typ_bezeichnern> und <enum_bezeichnern> sowie irgendwelchen Variablen-<bezeichnern> erlaubt.

Beispiele:

~~~~~~

```
        enum regenbogen {rot, gelb, gruen, blau, violett};
         .
        enum regenbogen spektrum, *fazei;
```

Die in der Aufzählung regenbogen genannten <bezeichner> rot, gelb, gruen, blau und violett sind als Konstanten (<enum_konstanten>) vereinbart und haben in der angegebenen Reihenfolge die Werte rot = 0, gelb = 1, gruen = 2 und violett = 4. Das erste Beispiel stellt eine <esu_typ_definition> dar, während das zweite eine Definition der Variablen spektrum und fazei ist. Wie bei Strukturen oder Varianten können <typ> und Variable zusammen definiert werden. Die Variable spektrum hat den <typ> `Aufzählung regenbogen' und fazei ist ein Zeiger auf ein Objekt dieses <typs>. Ebenfalls zulässige Aufzählungen sind zum Beispiel:

```
        enum einheit { eins = 1, odin = 1, one = 1, uno = 1 };

        enum permission { read = 4, write = 2, execut = 1 };
```
~~~~~~

9. Anweisungen

Die <anweisungen> werden in der Reihenfolge, wie sie im Programm als <anweisungs_liste> stehen, ausgeführt. Die Reihenfolge der Ausführung kann durch bestimmte <anweisungen> verändert werden. In den folgenden Abschnitten werden die einzelnen Alternativen, durch die eine <anweisung> darstellbar ist, behandelt.

```
        <anweisung> :
                  <ausdruck>;
                | <block>
                | if(<ausdruck>) <anweisung> [else <anweisung>]
                | while(<ausdruck>) <anweisung>


                | do <anweisung> while(<ausdruck>);
                | for([<ausdruck>]; [<aus-
                          druck>]; [<ausdruck>]) <anweisung>
                | <switch_anweisung>
                | break;
                | continue;
                | return [<ausdruck>];
                | <marke> : <anweisung>
                | goto <marke>;
                | ;
```

9.1. Ausdrucks-Anweisung

Ein <ausdruck>, dem ein Semikolon folgt, ist eine <anweisung>. Diese Art der <anweisung> tritt oft als Zuweisung oder Funktionsaufruf auf.

9.2. Blöcke

Mit einem <block> können mehrere <anweisungen> zu einer <anweisung> zusammengefaßt werden.

```
        <block> :
                  {[<vereinbarungs_liste>] [<anweisungs_liste>]}

        <vereinbarungs_liste> :
                  <interne_vereinbarung>
                | <interne_vereinbarung> <vereinbarungs_liste>

        <anweisungs_liste> :
                  <anweisung>
                | <anweisung> <anweisungs_liste>
```

Der schließenden geschweiften Klammer eines <blocks> folgt kein Semikolon. Variablen, die in einem <block> definiert werden, sind außerhalb des <blockes> nicht sichtbar. Die Gültigkeit einer derartigen Variablen bleibt für alle tiefer geschachtelten <blöcke> erhalten. Dies ist nicht der Fall, wenn durch eine erneute Vereinbarung des gleichen Variablen-<bezeichners> in dem geschachtelten <block> die Gültigkeit des vorhergehenden <bezeichners> außer Kraft gesetzt wird.

Die Initialisierung von auto- oder register-Variablen, die in einem <block> definiert wurden, findet jedesmal beim Eintritt in den <block> statt. Es ist jedoch auch möglich, in den <block>

hinein zu springen (schlechte Programmierpraxis). In diesem Fall werden die Initialisierungen nicht ausgeführt.

Die Initialisierung von static-Variablen wird nur einmal zu Beginn der Programmbearbeitung ausgeführt.

Innerhalb eines <blockes> wird durch eine <extern-deklaration> kein Speicherplatz reserviert. Eine Initialisierung derartig deklarierter Variablen ist nicht zulässig.

9.3. If_Anweisung

Die if-Anweisung ist eine Zweiwegeauswahl. Sie tritt in zwei Formen auf:

```
        if (<ausdruck>) <anweisung_1>
oder
        if (<ausdruck>) <anweisung_1> else <anweisung_2>
```

In beiden Fällen wird der Wert des <ausdrucks> berechnet. Ist dieser ungleich Null, so wird die <anweisung_1> ausgeführt. Ist der Wert des <ausdrucks> gleich Null, so wird bei der ersten Variante die <anweisung_1> übergangen und bei der zweiten Variante die <anweisung_2> ausgeführt.

Das else gehört zum letzten "else-losen" if-Zweig. Die if-Anweisungen dürfen geschachtelt werden.

9.4. While -Anweisung

Mit der while-Anweisung wird eine Schleife mit Test am Anfang realisiert. Sie hat die Form

```
        while (<ausdruck>) <anweisung>
```

Die <anweisung> wird solange wiederholt wie der Wert des <ausdrucks> ungleich Null ist.

9.5. Do-Anweisung

Mit der do-Anweisung wird eine Schleife mit Test am Ende realisiert.

```
        do <anweisung> while (<ausdruck>)
```

Die <anweisung> wird solange wiederholt bis der Wert des <ausdrucks> ungleich Null ist. Der Unterschied zwischen der while-Anweisung und der do-Anweisung besteht darin, daß bei der do-Anweisung die <anweisung> mindestens einmal ausgeführt wird.

9.6. For-Anweisung

Mit der for-Anweisung wird eine universelle Schleife realisiert.

```
        for (<ausdruck_1>) <aus-
                druck_2>; <ausdruck_3>) <anweisung>
```

Diese <anweisung> ist äquivalent zu folgendem Programmfragment:

```
        <ausdruck_1>;
        while (<ausdruck_2>) {
        <anweisung>;
        <ausdruck_3>;
        }
```

Der <ausdruck_1> dient der Schleifeninitialisierung. Mit dem <ausdruck_2> wird der Wiederholungstest ausgeführt und mit dem <ausdruck_3> erfolgt die Wiederinitialisierung der Schleife.

Jeder der drei <ausdrücke> kann weggelassen werden, wobei aber die Semikolons geschrieben werden müssen. Ein weggelassener <ausdruck_2> entspricht einem while (1) und die Schleife muß mit einer break-, goto-, [ber return-Anweisung verlassen werden. Fehlen alle drei <ausdrücke> gleichzeitig, liegt eine endlose Schleife vor.

9.7. Switch-Anweisung

Die switch -Anweisung dient der Mehrwegeauswahl. Sie hat die Form:

```
        switch (<ausdruck>) {
        case <konst_ausdruck_1>   : <anweisungs_liste_1>
        case <konst_ausdruck_i>   : <anweisungs_liste_i>
        default : <anweisungsliste_d>
        case <konst_ausdruck_i+1> : <anweisungs_liste_i+1>
        case <konst_ausdruck_n>   : <anweisungs_liste_n>
        }
```

Eine <anweisungs_liste> ist eine Folge von <anweisungen>, die auch leer sein kann.

```
        <anweisungs_liste> :
                  <anweisungs_liste>
                | <anweisung> <anweisungs_liste>
```

Der default-Zweig, als auch die case-Zweige vor oder nach dem default-Zweig können fehlen. Von den <konst_ausdrücken> wird gefordert, daß sie voneinander verschiedene Werte, die sogenannten case-Werte, ergeben. Der <ausdruck> muß bei seiner Auswertung einen Integer-Wert, der switch-Wert genannt wird, liefern. Dieser switch-Wert wird mit den case-Werten verglichen, bis Gleichheit festgestellt wird. In diesem Fall beginnt die Abarbeitung mit der zu diesem case-Zweig gehörenden <anweisungs_liste>. Alle nachfolgenden <anweisungs_listen> werden ebenfalls ausgeführt, bis eine break-Anweisung auftritt. Die Abarbeitung wird hinter der switch-Anweisung fortgesetzt. Wenn dem switch-Wert kein case-Wert gleicht, beginnt die Abarbeitung mit der <anweisungs_liste_d> und wird wie in der zuvor beschriebenen Situation bis zum Auftreten einer break-Anweisung in der betreffenden oder den nachfolgenden

~~~~~~
<anweisungs_listen> fortgesetzt.

Die case-Werte selbst haben bei der Ausführung keinen weiteren Einfluß.

9.8. Break-Anweisung

Die <anweisung>

```
break;
```

wird verwendet, um die Abarbeitung einer unmittelbar übergeordneten while-, do-, for- oder switch-Anweisung abzubrechen. Die Steuerung wird an die der abgebrochenen folgenden <anweisung> übergeben.

9.9. Continue-Anweisung

Die <anweisung>

```
continue;
```

bewirkt, daß die Steuerung an die nächste Schleifeniteration übergeben wird. Bei einer while- oder do-Anweisung wird als nächstes der Schleifentest ausgeführt und bei einer for-Anweisung wird vor dem Wiederholungstest noch die Wiederinitialisierung ausgeführt.

In jeder der folgenden <anweisungen> ist ein continue einem goto contin; äquivalent. Der <marke> cccooonnnttttiiinnn folgt eine leere <anweisung>.

```
while (...){      do{               for (...){
contin:;          contin:;          contin:;
}                }while (...);     }
```

9.10. Return-Anweisung

Mit Hilfe der return-Anweisung erfolgt der Rücksprung aus einer Funktion zur rufenden <anweisung>. Es gibt zwei Formen:

```
return;
```

Die Funktion wird verlassen und es wird kein Funktionswert übergeben. Der Funktionswert ist undefiniert. Bei

```
return <ausdruck>;
```

wird die Funktion verlassen und der Wert des <ausdrucks> als Ergebnis der Funktion übergeben. Falls erforderlich, findet eine Konvertierung in den für die Funktion definierten Datentyp statt.

Wird das Ende des den <funktions_körper> bildenden <blockes> erreicht, entspricht dies einem einfachen return; und der Funktionswert ist undefiniert.
~~~~~~

9.11. Goto-Anweisung und Marke

Eine unbedingte übergabe der Steuerung kann mit Hilfe der <anweisung> erfolgen:

```
goto <marke>;
```

Jeder <anweisung> können beliebig viele <marken> vorangestellt werden. Eine <marke> ist ein <bezeichner>, dem ein Doppelpunkt folgt:

```
<bezeichner> :
```

Die <marken> dienen ausschließlich als Ziele für goto-Anweisungen. Der Geltungsbereich einer <marke> ist die gesamte Funktion, mit Ausnahme irgendwelcher untergeordneten <blöcke> innerhalb der Funktion, in denen die gleichen <bezeichner> erneut deklariert werden (siehe Abschnitt 11.).

9.12. Leere Anweisung

Die leere <anweisung> hat die Form:

```
;
```

Die leere <anweisung> kann sinnvoll benutzt werden, um eine <marke> am Ende eines <blockes> zu definieren (wie in den Beispielen des Abschnittes 9.9.) oder als <anweisung> in einer for-Anweisung.

10. Externe Vereinbarungen

In der Sprache C besteht ein <programm> aus einer Folge <externer_vereinbarungen>.

```
<programm>:
          {<modul>}

<modul>:
          <externe_vereinbarung>
        | <externe_vereinbarung> <modul>

<externe_vereinbarung>:
          <funktions_definition>
        | <daten_definition>
        | <extern_deklaration>
        | <esu_typ_definition>
        | <typ_definition>
```

Bei einer <externen_vereinbarung> wird automatisch die <speicherklasse> eines <bezeichners> als extern implizit vereinbart. Wenn diese <speicherklasse> nicht gewünscht wird, sondern static, muß dies explizit angegeben werden.

Wenn in den Vereinbarungen kein <typ> angegeben wird, wird für den betreffenden <bezeichner> der <typ> int implizit vereinbart.

Der Gültigkeitsbereich <externer_vereinbarungen> erstreckt sich bis zum Ende des Files, in dem sie auftreten, so wie sich der Gültigkeitsbereich von Vereinbarungen bis zum Ende eines Blockes erstreckt.

Die Syntax <externer_vereinbarungen> gleicht denen der <internen_vereinbarungen>, mit der Ausnahme, daß nur bei <externen_vereinbarungen> für Funktionen der <funktions_körper> angegeben werden darf.
Die <typ_definition> und die <esu_typ_definition> werden im Abschnitt 8.8. behandelt.

10.1. Funktionen

Eine <funktions_definition> besteht aus einem <funktions_kopf> und einem <funktions_körper>. Funktionen dürfen nicht verschachtelt werden.

```
<funktions_definition>:
          <funktions_kopf> <funktions_körper

<funktions_kopf>:
          [static] [<typ>] <funktions_de-
                  klarator> [<par_deklarations_liste>]
```

Wenn im <funktions_kopf> die <speicherklasse> static nicht angegeben wird, hat die Funktion die <speicherklasse> extern (siehe Abschnitt 11.2.). Der <bezeichner> der Funktion erscheint in Form eines Deklarators, dem <funktions_deklarator>. Der <typ> vor dem <funktions_deklarator> im <funktions_kopf> gibt den <typ> des Funktionswertes an. In der <parameter_liste> des <funktions_namens> stehen die formalen Parameter der Funktion bei einer <funktions_definition>. In der Parameter-Deklarations-

~~~~~~
Liste, hier mit <par_deklarations_liste> abgekürzt, stehen die Deklarationen des <typs> und der <speicherklasse> für die formalen Parameter. Es können nur <bezeichner>, die auch in der <parameter_liste> stehen, angegeben werden. Wird für einen Parameter kein <typ> deklariert, wird für ihn implizit der <typ> int deklariert. Die einzige für Parameter angebbare <speicherklasse> ist register. Dieser Parameter wird, sofern ein Register frei ist, in dieses Register kopiert.

```
        <funktions_deklarator>:
                  <funktions_name
                | (<funktions_deklarator>)
                | *<funktions_deklarator>
                | (*<funktions_deklarator>) ()
                | (*<funktions_deklarator>) [[<konst_ausdruck>]]

        <funktions_name>:
                  <bezeichner> ([<parameter_liste>])

        <parameter_liste>:
                  <bezeichner>
                | <bezeichner>, <parameter_liste>

        <par_deklarations_liste> :
                  <par_deklaration>
                | <par_deklarations_liste> <par_deklaration>

        <par_deklaration> :
                  register [<typ>] <deklarator_liste>;
                | <typ> <deklarator_liste>;
                | <typ_definition>
```

Die <anweisungen>, die beim Aufruf einer Funktion ausgeführt werden sollen, stehen im <funktions_körper>.

```
        <funktions_körper> : <block>
```

Als Beispiel wird eine Funktion definiert, die das Maximum von drei int-Parametern bestimmt und dieses Maximum als int-Funktionswert liefert.

```
        int max (a,b,c)
        int a,b,c;
        {
        int m;
        m = (a > b) ? a : b;
        return((m > c) ? m : c);
        }
```

In dieser <funktions_definition> ist das int in der ersten Zeile die Deklaration des <typs> des Funktionsergebnisses. max(a,b,c) ist der <funktions_name> einschließlich der formalen Parameter a,b,c (<parameter_liste>). In der zweiten Zeile ist int a,b,c; die <par_deklarations_liste> für die formalen Parameter. In den geschweiften Klammern {...} stehen im <block> die bei Aufruf der Funktion max auszuführenden <anweisungen>.

In C werden alle aktuellen Parameter vom <typ> float in double konvertiert. Formale Parameter, die mit float deklariert werden, sind demnach ebenfalls als double zu betrachten.
~~~~~~

~~~~~~
Der <bezeichner> eines Feldes wird, wenn er als Parameter einer Funktion auftritt, wie auch in jedem anderen Kontext, in einen Zeiger konvertiert (siehe Abschnitt 14.3.). In der <par_deklaration> wird die erste Dimension eines Feldes weggelassen und nur die öffnende und schließende eckige Klammer angegeben. Die übrigen Dimensionen müssen angegeben werden.

Als <typen> der Funktionsergebnisse sind alle Grunddatentypen, also char, int, unsigned, float, double, sowie deren Modifikationen mittels short und long zugelassen, wobei bestimmte Typkonvertierungen stattfinden können (vergleiche Abschnitt 6.). Desweiteren kann der <typ> eines Funktionsergebnisses eine Struktur sein oder ein Zeiger auf beliebige Objekte. Nicht zugelassen ist die Rückgabe von Feldern, Varianten und Funktionen.

10.2. Daten-Definitionen und Extern-Deklarationen

~~~~~~

```
        <daten_definition> :
                  static [<typ>] <init_deklarator_liste>;
                | <typ> <init_deklarator_liste>;
```

Als <speicherklasse> kann für solche Daten static angegeben werden. Fehlt diese Angabe, haben die Daten implizit die Speicherklasse extern. Die <speicherklassen> auto oder register sind unzulässig. Wird explizit extern angegeben, so liegt keine <daten_definition> vor, sondern eine <extern_deklaration>.

```
        <extern_deklaration> :
                  extern [<typ>] <deklarator_liste>;
                | extern <funktions_deklaration>;
```

Eine <extern_deklaration> besagt, daß die angegebenen <deklaratoren> oder <funktions_bezeichner> die <speicherklasse> extern haben, und daß die Objekte, d.h. Variablen oder Funktionen entweder später im aktuellen File oder in einem anderen File, das zum <programm> gehört, definiert werden.

Die <esu_typ_definition> wurde nachträglich in die C-Syntax aufgenommen.

```
        <esu_typ_definition>:
                  enum <enum_typ_bezeichner>{<enu-
                            merator_liste>[,]};
                | struct <strukt_typ_bezeichner>{<ele-
                            ment_deklarations_liste>[;]};
                | union <strukt_typ_bezeichner>{<ele-
                            ment_deklarations_liste>[;]};
```

Der <typ> einer Aufzählung, einer Struktur oder einer Variante kann unabhängig von einer <daten_vereinbarung> oder

<internen_daten_vereinbarung> definiert werden.

11. Gültigkeitsbereiche

Ein C-Programm braucht nicht zur gleichen Zeit als Ganzes compiliert zu werden. Der Quellprogramm-Text kann in verschiedenen Files abgelegt sein und bereits übersetzte Funktionen können aus Bibliotheken geladen werden. Die Kommunikation zwischen den Funktionen eines Programms kann sowohl über explizite Aufrufe als auch über die gemeinsame Benutzung externer Daten erfolgen.

Es werden zwei Arten von Gultigkeitsbereichen unterschieden:
1. Der Gültigkeitsbereich von <bezeichnern> im Text ist der Text des Programms, in der der <bezeichner> verwendet wird, ohne daß es eine Fehlermitteilung "undefined identifier" bei der Compilierung auftritt (siehe 11.1).
2. Der Gültigkeitsbereich von <bezeichnern> der <speicherklasse> extern ist dadurch charakterisiert, daß die Bezugnahme auf gleiche extern <bezeichner> sich auch auf das gleiche Objekt beziehen.
Zwischen Gultigkeitsbereich, Lebensdauer und <speicherklasse> besteht ein enger Zusammenhang.

11.1. Gültigkeitsbereich im Text

Der Gültigkeitsbereich im Text erstreckt sich für <bezeichner>, die

- in <externen_vereinbarungen> vereinbart werden, von der Vereinbarung selbst bis zum Ende des Quellprogramm-Files,
- formale Parameter sind, über die Funktion, für die sie deklariert wurden,
- in der <vereinbarungs_liste> eines <blockes> vereinbart werden, bis zum Ende des <blockes>,
- als <marke> in einer Funktion auftreten, über die gesamte Funktion, in der sie erscheinen.

Da sich alle Bezugnahmen eines <bezeichners>, der die <speicherklasse> extern hat, auf das gleiche Objekt beziehen (siehe Abschnitt 11.2.), kontrolliert der Compiler alle Vereinbarungen dieses <bezeichners> in dem File auf Verträglichkeit.

In den Fällen, wo ein <bezeichner> explizit in der <vereinbarungs_liste> eines <blockes> vereinbart wurde, wird jede für denselben <bezeichner> außerhalb des <blockes> durchgeführte Vereinbarung, im betreffenden <block> verdrängt. Es gilt im <block> die in seiner <vereinbarungs_liste> angegebene Vereinbarung für den betreffenden <bezeichner>. Nach dem Verlassen des <blockes> ist die frühere Vereinbarung wieder gültig.

An dieser Stelle wird nochmal daran erinnert, daß der C-Compiler zwei Klassen für <bezeichner> führt. Eine Klasse existiert für <bezeichner> von Strukturen, Varianten und deren Komponenten und eine andere für alle übrigen <bezeichnern>. Zwischen <bezeichnern>, die zu beiden Klassen gehören, gibt es keine Konflikte (siehe Abschnitt 8.5.). Für die <bezeichner> der Klasse für Strukturen, Varianten und deren Komponenten gelten dieselben Gültigkeitsbereichsregeln wie für die anderen <bezeichner>.

Der <bezeichner>, der in einer <typ_definition> definiert wird, gehort zur selben Klasse wie die üblichen <bezeichner>. Damit kann es zu folgendem Problem kommen:

```
        typedef float wert;
         . . .
        {
                register int wert;  . . .
        }
```

Würde in der Zeile register int wert; nur register wert; stehen, wäre dies aufgrund der <typ_definition> eine unvollständige Deklaration, weil sie keinen <bezeichner> angibt. (Im <block> wird in dem obigen Beispiel wert umdeklariert.)

11.2. Gültigkeitsbereich externer Bezeichner

Wird in einer Funktion auf einen als extern deklarierten <bezeichner> Bezug genommen, so muss es in irgendeinem der Files oder Bibliotheken, die das gesamte Programm bilden, eine externe Definition für diesen <bezeichner> geben. Ein <bezeichner>, der die <speicherklasse> extern haben soll, muß durch eine <daten_definition> oder <funktions_definition> definiert werden. In den <extern_deklarationen> des selben <bezeichners> dürfen keine zur Definition widersprüchlichen Angaben stehen. Die <extern_deklaration> wird sinnvoller Weise in den Files angegeben, in denen nicht die externe Definition steht. Damit wird die "Verbindung" zwischen diesen Files und den Definitionen hergestellt.

Tritt in einer Vereinbarung das <schlüsselwort> extern auf, so liegt eine <extern_deklaration> vor. Speicherplatz wird an dieser Stelle nicht bereitgestellt. Vom Compiler wird nur bei einer Definition (<daten_definition>, <funktions_definition>) Speicherplatz vorgesehen.

Die in <externen_vereinbarungen> mit der <speicherklasse> static deklarierten <bezeichner> sind in anderen Files nicht sichtbar, d.h. Funktionen, die in den anderen Files stehen, können auf diese Variablen nicht zugreifen.
Variable, die in einem <block> mit der <speicherklasse> static vereinbart werden, sind nur in diesem <block> sichtbar. In einem <block> vereinbarte static-Variablen verhalten sich bezüglich ihrer Sichtbarkeit wie auto-Variablen, behalten aber beim Verlassen ihre Werte, die bei einem erneuten Eintritt in den <block> weiter verwendet werden können. (Dagegen verlieren auto-Variablen beim Verlassen des <blocks> ihre Werte.)
Funktionen können mit der <speicherklasse static vereinbart werden.

12. Der C-Präprozessor

Der C-Compiler enthält einen Präprozessor. Mit diesem Präprozessor ist es möglich, Makros und symbolische Konstanten zu definieren, Files in den Quelltext einzufügen und eine bedingte übersetzung zu steuern.

Präprozessorzeilen beginnen mit # und enden nicht mit Semikolon! Diese Zeilen sind von der Syntax der Sprache C vollkommen unabhängig. Sie können überall im Quelltext auftreten und behalten ihre Wirkung bis zum Ende des Quellprogramm-Files und sind von den sonst in C geltenden Gültigkeitsbereichsregeln unabhängig.

```
#define <bezeichner> <zeichenfolge>
#define <bezeichner>(<bezeichner>,.,<be-
                                    zeichner>) <zeichenfolge>
#undef <bezeichner>
#include "<filename>"
#include <<filename>>
#if <konst_ausdruck>
#ifdef <bezeichner>
#ifndef <bezeichner>
#else
#endif
#line <integer_konstante> <bezeichner>
```

12.1. Ersetzen von Text, Makros

1.Form:

```
#define <bezeichner> <zeichenfolge>
```

Der Präprozessor ersetzt den <bezeichner> im folgenden Programmtext durch die angegebene <zeichenfolge> (daher der Begrif `Ersetzungstext').

2.Form:

```
#define <bezeichner>(<bezeichner>,.,<be-
                                    zeichner>) <zeichenfolge>
```

Bemerkung: Kein Leerzeichen zwischen erstem <bezeichner> und der öffnenden Klammer!

Diese Form entspricht einer Makrodefinition mit Parametern. Ein Makroaufruf im Programmtext kann durch den Makronamen (erster <bezeichner>), die öffnende Klammer mit folgenden Argumenten, sowie der schließenden Klammer, angegeben werden. Der Präprozessor ersetzt den Makroaufruf durch die <zeichenfolge>, wobei die Namen der Makroparameter immer durch die entsprechenden Argumente ersetzt werden, die im Makroaufruf stehen. Die aktuellen Argumente im Ruf sind durch Kommas getrennte <zeichenfolgen>. Kommas innerhalb von in <anführungszeichen> eingeschlossenen <zeichenketten>, bewirken keine Trennung der Argumente.
Die Anzahl der formalen und der aktuellen Parameter muß gleich sein.
Text innerhalb einer <zeichenketten_konstante> oder einer <zeichen_konstanten> wird nicht ersetzt.

Die Ersetzungs-<zeichenfolge> wird nach weiteren bereits defi-

~~~~~~
nierten <bezeichnern> durchmustert.

Ein `\' als letztes Zeichen auf der Zeile bedeutet, daß eine Fortsetzungszeile folgt.

Beispiel für eine symbolische Konstante und ihre Verwendung:

```
        #define TABLESIZE 100
        int table [TABLESIZE];
```

Die Definition von TABLESIZE kann durch die folgende Zeile aufgehoben werden.

```
        #undef TABLESIZE
```

12.2. Einfügen von Files

~~~~~~~~~~~~~~~~~~~~~~~~

1.Form:

```
        #include "<filename>"
```

Die laufende Zeile wird durch den Gesamtinhalt des durch den <filenamen> bezeichneten Files ersetzt. Das File wird zuerst in der Directory (Verzeichnis, Katalog), in dem das Quellfile steht, gesucht. Ist es dort nicht vorhanden, wird es in Standard-Directories gesucht.

2.Form:

```
        #include <<filename>>
```

Die Suche erfolgt nur in Standard-Directories, nicht in der Directory des Quellfiles.

Die #includes können ineinander verschachtelt werden.

12.3. Bedingte übersetzung

~~~~~~~~~~~~~~~~~~~~~~~~~~

Tritt die folgende Zeile in einem Programmtext auf, so testet der Präprozessor, ob der <konst_ausdruck> einen Wert ungleich Null liefert. (Siehe auch Abschnitt 15.). Der <konst_ausdruck> stellt eine Bedingung dar.

```
        #if <konst_ausdruck>
```

Die beiden Zeilen

```
        #ifdef <bezeichner>
```

und

```
        #ifndef <bezeichner>
```

veranlassen den Präprozessor, zu testen, ob der <bezeichner> definiert wurde, oder nicht. (siehe Abschnitt 12.1.)

Allen drei Formen kann eine beliebige Anzahl von Zeilen folgen, in denen auch die Präprozessorzeile

```
        #else
```

enhalten sein kann. Das Ende der bedingt zu übersetzenden Zeilen
~~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~
wird durch die Zeile

```
        #endif
```

bezeichnet.

Ist die Bedingung "wahr", d.h. ist der vom <konst_ausdruck> gelieferte Wert verschieden von Null, so werden die Zeilen zwischen #else und #endif ignoriert, anderfalls alle Zeilen zwischen dem Test und einem #else oder, falls kein solches vorhanden ist, dem #endif.

Diese Konstruktionen können ineinander verschachtelt sein.

12.4. Zeilen-Numerierung

~~~~~~~~~~~~~~~~~~~~~~~~
Es existieren im MUTOS Programme, die C-Programme generieren, z.B. der Compiler-Compiler <yacc>. Um dem C-Compiler die Möglichkeit zu geben, Fehlerausschriften bezüglich der Originalquelle zu erzeugen, also z.B. einer <yacc>-Grammatik, gibt es folgende Präprozessorzeile:

```
        #line <integer_konstante> <bezeichner>
```

Die <integer_konstante> gibt die nächste Zeilennummer in dem durch den <bezeichner> benannten File an. Mit dieser Präprozessorzeilenart hat der C-Compiler die Möglichkeit, in Fehlermeldungen auf das durch den <bezeichner> angegeben File und dessen Zeilennumerierung Bezug zu nehmen. Ist der <bezeichner> nicht angegeben, so wird das bis dahin verwendete File weiter verwendet.

13. Implizite Vereinbarungen

Für einen <bezeichner> müssen nicht in jedem Fall die <speicherklasse> und der <typ> durch eine Vereinbarung angegeben werden. In C gibt es implizite Vereinbarungen, die aus dem Kontext des betreffenden <bezeichners> hergeleitet werden.

Die <speicherklasse> extern ergibt sich implizit für eine Variable, wenn diese Variable außerhalb einer Funktion definiert wurde. Eine Funktion hat immer die <speicherklasse> extern, falls nicht explizit static angegeben wird.
Parameter haben implizit die <speicherklasse> auto, falls nicht explizit register angegeben wurde.
Bei Komponenten von Strukturen und Varianten ist die <speicherklasse> von deren Benutzung abhängig.

In einem <block> gilt für Vereinbarungen:
Ist nur die <speicherklasse> angegeben, so hat der <bezeichner> implizit den <typ> int.
Ist nur der <typ> angegeben und nicht die <speicherklasse>, so hat der <bezeichner> die <speicherklasse> auto. Eine Funktion kann nicht die <speicherklasse> auto annehmen und stellt somit eine Ausnahme für die eben genannte Regel dar.

Hat ein <bezeichner> den <typ> `Funktion', wird er implizit als extern vereinbart.

Tritt in einem <ausdruck> ein <bezeichner> auf, dem eine runde Klammer folgt und der noch nicht vereinbart wurde, so wird dieser <bezeichner> als Funktion mit dem Funktionswert-<typ> int deklariert. Man sollte daher eine Funktion vor ihrer Benutzung vereinbaren.

14. Operationen über Objekten höheren Datentyps

Im folgenden wird eine Zusammenfassung der Operationen über Objekten höherer <typen> angegeben.

14.1. Strukturen und Varianten

Es können folgende Operationen auf Strukturen oder Varianten angewendet werden:

Eine Komponente kann mit den Operatoren . oder -> ausgewählt werden und die Adresse eines Objektes kann mit dem Operator & bestimmt werden.

Eine Struktur kann an eine Struktur gleichen <typs> zugewiesen werden, auch kann eine Struktur als Parameter an eine Funktion übergeben werden. Eine Funktion kann als Ergebnis eine Struktur liefern. Neben diesen Möglichkeiten können Zeiger auf Strukturen oder Zeiger auf Varianten als Parameter und als Ergebnisse von Funktionen verwendet werden.

Bei der Behandlung des <einfachen_ausdrucks> im Abschnitt 7.1 wurde angegeben, daß bei der Auswahl einer Komponente einer Struktur oder Variante mit den Operatoren . oder -> der <bezeichner> der Komponente in der durch den <lvalue> benannten bzw. gezeigten Struktur oder Variante vereinbart sein soll.
Von einem <lvalue> wird bei Verwendung des Operators . angenommen, daß er eine Struktur oder Variante angibt, die die Komponente enthält. Bei der Verwendung des Operators -> wird angenommen, daß der <lvalue> auf eine Struktur oder Variante zeigt, in der die angegebene Komponente zu finden ist. Wie bereits bemerkt wurde, kontrolliert der Compiler diese Festlegung nicht und gestattet auf diesem Wege, die Typisierung "aufzuweichen". Werden diese Festlegungen nicht eingehalten, sind die Programme nicht in jedem Fall portabel.

14.2. Funktionen

Auf Funktionen können zwei Operationen angewandt werden: Eine Funktion kann aufgerufen werden und von einer Funktion kann die Adresse bestimmt werden. Erscheint der <funktions_name> in einem <ausdruck> nicht an einer Stelle, wo der <funktions_name> als Funktionsaufruf betrachtet werden kann, wird ein Zeiger, der auf diese Funktion zeigt, generiert.

Beispiel:
Um eine Funktion einer anderen zu übergeben, könnte man schreiben:

```
        int f();
        g(f);
```

Die Definition von g könnte sein:

```
        g(funcp)
        int (*funcp)();
        {
         .
        (*funcp)();
         .
```

~~~~~~
```
        }
```

Hier ist wichtig zu bemerken, daß f im Programm explizit vereinbart sein muß. Bei der Verwendung von f in der Zeile g(f) folgt dem f keine öffnende runde Klammer. Damit kann der Compiler nicht erkennen, daß f eine Funktion sein soll und benötigt aus diesem Grund die explizite Vereinbarung von f.

14.3. Felder, Zeiger und Indizes

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Wenn der <bezeichner> eines Feldes in einem <ausdruck> auftritt, wird er in einen Zeiger konvertiert, der auf das erste Elemnt des Feldes zeigt. Aus diesem Grunde sind Felder keine <lvalues>.

Per Definition wird der Operator [ ] so interpretiert, daß E1[E2] identisch ist zu *((E1)+(E2)). Wegen der Konvertierungsregeln, die mit dem Operator + wirksam werden, gilt, daß E1[E2] das Element an der E2-ten Position im Feld E1 bezeichnet, wenn E1 ein Feld und E2 letztlich ein Integerwert ist. Damit zeigt sich, daß der Operator [ ] trotz seines asymmetrischen Erscheinens kommutativ ist (da + auch kommutativ ist).

Für mehrdimensionale Felder gilt folgende einheitliche Regel: Ist E ein n-dimensionales Feld des Formats i X j X . X k, dann wird E beim Auftreten in einem <ausdruck> in einen Zeiger auf ein (n-1)-dimensionales Feld des Formats j X . X k konvertiert. Wird der Operator * entweder explizit oder implizit über eine Indizierung auf diesen oben genannten Zeiger angewendet, so ist das Ergebnis eben dieses (n-1)-dimensionale Feld, was nun seinerseits in einen Zeiger konvertiert wird.

Beispiel: aaa sei ein Feld von 3 X 5 Integer-Variablen,

```
        int a[3][5];
```

Tritt a in einem <ausdruck> auf, wird a in einen Zeiger konvertiert, der auf das erste von drei fünf-elementigen Feldern, die aus Integer-Variablen bestehen, zeigt. In dem <ausdruck> a[i], der äquivalent zu *(a+i) ist, wird a zuerst in einen Zeiger konvertiert. Danach wird der Integerwert i mit der Länge des Objektes, auf das der Zeiger zeigt, multipliziert, hier also mit der Länge eines Feldes, das aus fünf int-Elementen besteht. Dann werden der Zeigerwert und das Multiplikationsergebnis addiert. Mittels der Referenzierung erhält man ein Feld, das aus 5 int-Objekten besteht. Dieses wird wiederum in einen Zeiger auf das erste der Integer-Objekte konvertiert.
Wird hinter a noch ein weiterer Index angegeben, so wird das Verfahren wiederholt. Das Ergebnis wäre hier in unserem Beispiel ein int-Wert.

Kurz zusammengefaßt heißt dies: Felder werden in C zeilenweise gespeichert, d.h. der letzte Index variiert am schnellsten. Den erste Index in der Vereinbarung braucht der Compiler für die Bestimmung des für das Feld benötigten Speicherplatzes, ansonsten spielt dieser erste Index keine Rolle bei der Indexberechnung.

14.4. Explizite Konvertierung von Zeigern

Bestimmte Konvertierungen von Zeigern sind erlaubt. Diese Konvertierungen sind maschinenabhängig. Sie werden alle mit Hilfe eines expliziten Typ-Konvertierungs-Operators (siehe Abschnitte 7.2. und 8.7.) angegeben.

Ein Zeiger kann in irgendeinen der Integerwerte konvertiert werden. Die Bedingung ist, daß der <typ> des Integerwertes genügend groß ist, um den Zeiger vollständig aufzunehmen. Ob ein int oder long erforderlich ist, ist maschinenabhängig, ebenso die Konvertierungsfunktion selbst.

Ein Objekt vom <integer_typ> kann explizit in einen Zeiger konvertiert werden. Diese Konvertierungsoperation ist umkehrbar und liefert dieselben Werte. Sie ist maschinenabhängig.

Ein Zeiger auf einen bestimmten <typ> kann in einen Zeiger auf einen anderen <typ> konvertiert werden. Der resultierende Zeiger kann jedoch Adreß-Ausnahmebedingungen verursachen, wenn er auf ein nicht geeignet im Speicher ausgerichtetes Objekt Bezug nimmt (z.B. ungerade Wortadresse beim AC A7100/A7150).

Es wird gewährleistet, dass ein Zeiger auf ein Objekt bestimmter Größe in einen Zeiger auf ein kleineres Objekt - und auch wieder zurück - ohne änderungen konvertiert werden kann und dabei dieselben Zeigerwerte geliefert werden.
Beispiel: Die Speicherplatz-Zuteilungsfunktion calloc stellt für eine anzugebende Anzahl von Objekten, desssen Größe in Bytes (char) angegeben wird, Speicherplatz und einen `Zeiger auf char' auf diesen Speicherplatz bereit. Diese Funktion kann wie folgt benutzt werden:

```
        extern char *calloc();
        double *dp;

         .
        dp = ( double*) alloc( sizeof( double));
        *dp = 19.0/6.0;
```

In diesem Beispiel stellt die Funktion calloc einen Speicherbereich im Umfang einer doppeltgenauen Real-Variablen bereit. Die Funktion liefert zunächst einen Zeiger, der auf char zeigt, als Ergebnis. Dieser `Zeiger auf char' wird mit dem Cast (double *) explizit in einen `Zeiger auf double' konvertiert. Die Funktion calloc, die Bestandteil der MUTOS-Bibliothek ist, ist selbst maschinenabhängig. Programme, die calloc verwenden, sind aber dadurch selbst nicht maschinenabhängig. (Die Maschinenabhängigkeit muß durch den Programmierer, der die Funktion calloc für die betreffende Maschine schreibt, bewältigt werden.)

Für den AC A7100/A7150 gilt:
Ein Zeiger wird durch einen 16 Bit großen Integerwert repräsentiert und wird in Bytes gemessen.
Zeichen (chars) erfordern nicht das Ausrichten auf bestimmte Adreßgrenzen. Objekte, die nicht den <typ> char haben, müssen ab einer geraden Adresse beginnen.

~~~~~~
15. Konstanten-Ausdrücke
~~~~~~~~~~~~~~~~~~~~~~~~

Im Zusammenhang mit switch-<anweisungen>, Initialisierungen und Feldgrenzen werden Konstantenausdrücke (<konst_ausdrücke>) verwendet. Die Operanden sind vom <integer_typ> oder sind <enum_konstanten>.

<konst_ausdruck> :

```
          <integer_konstante>
        | <long_integer_konstante>
        | <zeichen_konstante>
        | (<konst_ausdruck>)
        | ~<konst_ausdruck>
        | -<konst_ausdruck>
        | sizeof <ausdruck>
        | sizeof (<typ_spezifikation>)
        | <enum_konstante>
        | <konst_ausdruck>  <mul_op>  <konst_ausdruck>
        | <konst_ausdruck>  <add_op>  <konst_ausdruck>
        | <konst_ausdruck>  <shift_op>  <konst_ausdruck>
        | <konst_ausdruck>  <rel_op>  <konst_ausdruck>
        | <konst_ausdruck>  <bit_op>  <konst_ausdruck>
        | <konst_ausdruck> ? <konst_ausdruck> : <konst_ausdruck>
```

Der unäre Operator & kann auf Adressen von statischen oder externen Objekten und auf Adressen von statischen oder externen Feldern mit konstanten Dimensionsangaben angewendet werden. Der &-Operator kann auch implizit gebraucht werden, indem Feld<bezeichner> ohne Index und indem <bezeichner> von Funktionen angegeben werden. Grundsätzlich müssen die Ausdrücke, mit denen initialisiert wird, entweder <konst_ausdrücke> oder Adressen eines zuvor vereinbarten extern- oder static-Objektes sein, die aber noch mit einer <konstanten> additiv verknüpft sein können.

16. Syntaxübersicht

Die in der Syntaxübersicht verwendeten metalinguistischen Konnektoren : [] | {} sind im Abschnitt 1.2. erläutert. Sie werden normal gedruckt.
C-Zeichen als Bestandteil metalinguistischer Variablen sowie Schlüsselworte werden durch Fettdruck hervorgehoben. Zur bequemeren Handhabung der Syntaxübersicht sind die syntaktischen Regeln mit Nummern versehen. Am rechten Rand sind Verweise auf die Regeln angeführt, in denen die in der Zeile stehenden metalinguistischen Variablen erklärt sind.

```
1   <programm> :
          {<modul>}                                          2

2   <modul> :
          <externe_vereinbarung>                             3
        | <externe_vereinbarung> <modul>                     3,2

3   <externe_vereinbarung> :
          <funktions_definition>                             4
        | <daten_definition>                                 14
        | <extern_deklaration>                               15
        | <esu_typ_definition>                               16
        | <typ_definition>                                   17

4   <funktions_definition> :
          <funktions_kopf> <funktions_körper>                5,20

5   <funktions_kopf> :
          [static] [<typ>] <funktions_de-                    6,31,32
                klarator> [<par_deklarations_liste>]         12

6   <funktions_deklarator> :
          <funktions_name>                                   10
        | (<funktions_deklarator>)
        | *<funktions_deklarator>
        | (*<funktions_deklarator>) ()
        | (*<funktions_dekla-
                rator>) [[<konst_ausdruck>]]                 6,74

10  <funktions_name> :
          <bezeichner> ([<parameter_liste>])                 90,11

11  <parameter_liste> :
          <bezeichner>                                       90
        | <bezeichner>, <parameter_liste>                    90,11

12  <par_deklarations_liste> :
          <par_deklaration>                                  13
        | <par_deklarations_liste> <par_deklaration>         12,13

13  <par_deklaration> :
          register [<typ>] <deklarator_liste>;               31,32,56
        | <typ> <deklarator_liste>;                          32,56
        | <typ_definition>                                   17

14  <daten_definition> :
          static [<typ>] <init_deklarator_liste>;            31,32,52
        | <typ> <init_deklarator_liste>;                     32,52
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
15  <extern_deklaration> :
          extern [<typ>] <deklarator_liste>;              30,32,56
        | extern <funktions_deklaration>;                 30,24

16  <esu_typ_definition> :
          enum <enum_typ_bezeichner>                      32,45
                {<enumerator_liste>[,]};                  46
        | struct <strukt_typ_bezeichner>                  32,37
                {<element_deklarations_liste>[;]};        40
        | union <strukt_typ_bezeichner>                   32,37
                {<element_deklarations_liste>[;]};        40

17  <typ_definition> :
          typedef [<typ>] <deklarator_liste>;             30,32,56

20  <funktions_körper> :
          <block>                                         21

21  <block> :
          {[<vereinbarungs_liste>] [<anweisungs_liste>]} 22,60

22  <vereinbarungs_liste> :
          <interne_vereinbarung>                          23
        | <interne_vereinbarung> <vereinbarungs_liste>    23,22

23  <interne_vereinbarung> :
          <funktions_deklaration>                         24
        | <interne_daten_definition>                      26
        | <extern_deklaration>                            15
        | <esu_typ_definition>                            16
        | <typ_definition>                                17

24  <funktions_deklaration> :
          [<typ>] <funktionsbezeichner>;                  32,25

25  <funktions_bezeichner> :
          <bezeichner> ()                                 90
        | (<funkti_bezeichner>)
        | *<funktions_bezeichner>
        | (*<funktions_bezeichner>) ()
        | (*<funktions_bezeichner>) []

26  <interne_daten_definition> :

          <def_speicherklasse> [<typ>]                    31,32
                <init_deklarator_liste>;                  52
        | <typ> <init_deklarator_liste>;                  32,52

30  <speicherklasse> :
          extern
        | <def_speicherklasse>                            31
        | typedef

31  <def_speicherklasse> :
          auto
        | register
        | static

32  <typ> :
          <arithmetischer_typ>                            33
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        | unsigned
        | unsigned <integer_typ>                          34
        | struct <strukt_deklarator>                      36
        | union <strukt_deklarator>                       36
        | enum <enum_deklarator>                          44
        | <typ_bezeichner>                                51

33  <arithmetischer_typ> :
          <integer_typ>                                   34
        | <real_typ>                                      35

34  <integer_typ> :
          char
        | int
        | short
        | long
        | short int
        | long int

35  <real_typ> :
          float
        | long float
        | double

36  <strukt_deklarator> :
          [<strukt_typ_bezeichner>]                       37
                {<element_deklarations_liste>[;]}         40
        | <strukt_typ_bezeichner>                         37

37  <strukt_typ_bezeichner> :
        <bezeichner>                                      90

40  <element_deklarations_liste> :
          <element_deklaration>                           41
        | <element_deklaration> <ele-
                ment_deklarations_liste>                  41,40

41  <element_deklaration> :
          <typ> <element_deklarator_liste>;               32,42

42  <element_deklarator_liste> :
          <element_deklarator>                            43
        | <element_deklarator>, <ele-                     43,42
                ment_deklarator_liste>

43  <element_deklarator> :
          <deklarator>                                    57
        | [<deklarator>] : <konst_ausdruck>               57,74

44  <enum_deklarator> :
          [<enum_typ_bezeichner>]                         45,46
                {<enumerator_liste>[,]}
        | <enum_typ_bezeichner>                           45

45  <enum_typ_bezeichner> :
          <bezeichner>                                    90

46  <enumerator_liste> :
          <enumerator>                                    47
        | <enumerator>, <enumerator_liste>                47,46
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
47  <enumerator> :
          <enum_bezeichner> [= <konst_ausdruck>]         50,74

50  <enum_bezeichner> :
          <bezeichner>                                   90

51  <typ_bezeichner> :
          <bezeichner>                                   90

52  <init_deklarator_liste> :
          <init_deklarator>                              53
        | <init_deklarator>, <init_deklarator_liste>     53,52

53  <init_deklarator> :
          <deklarator>                                   57
        | <deklarator> = <init_wert>                     57,55
        | <deklarator> = {<init_wertliste>[,]}           57,54

54  <init_wert_liste> :
          <init_wert>                                    55
        | <init_wert>, <init_wert_liste>                 55,54

55  <init_wert> :
          <ausdruck>                                     70
        | {<init_wert_liste>[,]}                         54

56  <deklarator_liste> :
          <deklarator>                                   57
        | <deklarator>, <deklarator_liste>               57,56

57  <deklarator> :
          <bezeichner>                                   90
        | (<deklarator>)                                 57
        | *<deklarator>                                  57
        | (*<deklarator>) ()                             57
        | (*<deklarator>) []                             57
        | <feld_deklarator>                              58

58  <feld_deklarator> :
          <bezeichner>[<konst_ausdruck>]                 90,74
        | <feld_deklarator>[<konst_ausdruck>]            58,74

60  <anweisungs_liste> :
          <anweisung>                                    61
        | <anweisung> <anweisungs_liste>                 61,60

61  <anweisung> :
          <ausdruck>;                                    70
        | <block>                                        21
        | if(<ausdruck>) <anweisung> [else <anweisung>]  70,61,61
        | while(<ausdruck>) <anweisung>                  70,61
        | do <anweisung> while(<ausdruck>);              61,70
        | for([<ausdruck>]; [<aus-                       70,70
              druck>]; [<ausdruck>])<anweisung>          70,61
        | <switch_anweisung>                             62
        | break;
        | continue;
        | return[<ausdruck>];                            70
        | <marke> : <anweisung>                          66,61
        | goto <marke>;                                  66
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        | ;

62  <switch_anweisung> :
          switch <ausdruck> {<caseliste>                  70,63
                [<default_zweig>] [<case_liste>]}         65,63
        | switch <ausdruck>                               70
                {<default_zweig> <case_liste>}            65,64

63  <case_liste> :
          <case_zweig>                                    64
        | <case_zweig> <case_liste>                       64,63

64  <case_zweig> :
          case <konst_ausdruck> : [<anweisungs_liste>]    64,70

65  <default_zweig> :
          default : [<anweisungs_liste>]                  60

66  <marke> :
          <bezeichner>                                    90

70  <ausdruck> :
          <einfacher_ausdruck>                            71
        | <konst_ausdruck>                                74
        | !<ausdruck>
        | ~<ausdruck>
        | <inc_op> <lvalue>                               81,72
        | <lvalue> <inc_op>                               72,81
        | -<ausdruck>
        | (<typ_spezifikation>)<ausdruck>                 76,70
        | &<lvalue>                                       72
        | *<ausdruck>
        | <ausdruck> <mul_op> <ausdruck>                  70,82,70
        | <ausdruck> <add_op> <ausdruck>                  70,83,70
        | <ausdruck> <shift_op> <ausdruck>                70,84,70
        | <ausdruck> <rel_op> <ausdruck>                  70,85,70
        | <ausdruck> <bit_op> <ausdruck>                  70,86,70
        | <ausdruck> <log_op> <ausdruck>                  70,87,70
        | <ausdruck> ? <ausdruck> : <ausdruck>
        | <lvalue> <ass_op> <ausdruck>                    72,80,70
        | <ausdruck>, <ausdruck>
        | (<ausdruck>)

71  <einfacher_ausdruck> :
          <bezeichner>                                    90
        | <konstante>                                     91
        | (<ausdruck>)                                    70
        | <einfacher_ausdruck> ([<argument_liste>])       71,73
        | <einfacher_ausdruck>[<ausdruck>]                71,70
        | <lvalue>.<bezeichner>                           72,90
        | <einfacher_ausdruck>-><bezeichner>              71,90

72  <lvalue> :
          <bezeichner>                                    90
        | <einfacher_ausdruck>[<ausdruck>]                71,70
        | <lvalue>.<bezeichner>                           72,90
        | <einfacher_ausdruck>-><bezeichner>              71,90
        | *<lvalue>
        | (<lvalue>)
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
73  <argument_liste> :
          <ausdruck>                                          70
        | <ausdruck>, <argument_liste>                       70,73

74  <konst_ausdruck> :
          <integer_konstante>                                 92
        | <long_integer_konstante>                            96
        | <zeichen_konstante>                                 102
        | (<konst_ausdruck>)
        | ~~~<konst_ausdruck>
        | -<konst_ausdruck>
        | sizeof <ausdruck>
        | sizeof (<typ_spezifikation>)                        76
        | <enum_konstante>                                    75
        | <konst_ausdruck> <mul_op> <konst_ausdruck>        74,82,74
        | <konst_ausdruck> <add_op> <konst_ausdruck>        74,83,74
        | <konst_ausdruck> <shift_op> <konst_ausdruck>      74,84,74
        | <konst_ausdruck> <rel_op> <konst_ausdruck>        74,85,74
        | <konst_ausdruck> <bit_op> <konst_ausdruck>        74,86,74
        | <konst_ausdruck>  ? <konst-
              _ausdruck> : <konst_ausdruck>

75  <enum_konstante> :
          <enum_bezeichner>                                   50

76  <typ_spezifikation> :
        <typ> <abstrakter_deklarator>                         32,77

77  <abstrakter_deklarator> :
          [<abstrakter_deklarator>]
        | (<abstrakter_deklarator>)
        | *<abstrakter_deklarator>
        | <konst_ausdruck>()                                  74
        | <abstrakter_deklarator>[[<konst_ausdruck>]]        77,74

80  <ass_op> :
              =       | <mul_op>= | <add_op>=                82,83
        | <shift_op>= | <bit_op>=                            84,86

81  <inc_op> :
          ++ | --

82  <mul_op> :
          * | / | %

83  <add_op> :
          + | -

84  <shift_op> :
          << | >>

85  <rel_op> :
          < | <= | > | >= | == | !=

86  <bit_op> :
          & | ^       | |

87  <log_op> :
          && | |
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
90  <bezeichner> :
          <buchstabe>{<buchstabe>|<ziffer>}>             115,112

91  <konstante> :
          <integer_konstante>                            92
        | <long_integer_konstante>                       96
        | <gleitkomma_konstante>                         100
        | <zeichen_konstante>                            102
        | <zeichenketten_konstante>                      103
        | <konst_ausdruck> <rel_op> <konst_ausdruck>     74,85,74

92  <integer_konstante> :
          <dezimal_konstante>                            93
        | <oktal_konstante>                              94
        | <hexadezimal_konstante>                        95

93  <dezimal_konstante> :
          <nicht_null_ziffer>[<dezimalzahl>]>            113,110

94  <oktal_konstante> :
          0<oktalziffer>{<oktalziffer>}>                 114,114

95  <hexadezimal_konstante> :
          0x<hexadezimalziffer>{<hexadezimalziffer>}>    1,1
        | 0X<hexadezimalziffer>{<hexadezimalziffer>}>    1,1

96  <long_integer_konstante> :
          <integer_konstante>L                           92
        | <integer_konstante>l                           92

100 <gleitkomma_konstante> :
          <dezimalzahl>.<dezimalzahl>[<exponent>]        110,101
        | <dezimalzahl>.[<exponent>]                     110,101
        | <dezimalzahl> <exponent>                       110,101
        | .<dezimalzahl>[<exponent>]                     110,101

101 <exponent> :
          eee[+|-]<dezimalzahl>                          110
        | E[+|-]<dezimalzahl>                            110

102 <zeichen_konstante> :
          '<buchstabe>'                                  115
        | '<ziffer>'                                     112
        | '<sonderzeichen>'                              120
        | '<nichtgrafisches_zeichen>'                    121
        | '<anführungszeichen>'                          123

103 <zeichenketten_konstante> :
          "{<zeichen>}"                                  104

104 <zeichen> :
          <buchstabe>                                    115
        | <ziffer>                                       112
        | <sonderzeichen>                                120
        | <apostroph>                                    122
        | <nichtgrafisches_zeichen>                      121

110 <dezimalzahl> :
          <ziffer>{<ziffer>}                             112
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
111 <hexadezimalziffer>   :
          <ziffer> |a|A|b|B|c|C|d|D|e|E|f|F             112

112 <ziffer> :
          0 | <nicht_null_ziffer>                      113

113 <nicht_null_ziffer> :
         1|2|3|4|5|6|7|8|9

114 <oktalziffer> :
         0|1|2|3|4|5|6|7

115 <buchstabe> :
         a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s
        |t|u|v|w|x|y|z
        |A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S
        |T|U|V|W|X|Y|Z|_

120 <sonderzeichen> :
         !|?|(|)|[|]|{|}|+|-|=|<|>|h'-1'|.|,|:|;
        | |*|&|#|%|$|^|~

121 <nichtgrafisches_zeichen> :
          \n|\t|\b|\r|\f|\\|\'|\0|\"|
        | \<oktalziffer>[<oktalziffer>[<oktalziffer>]]   114

122 <apostroph> :
          '

123 <anführungszeichen> :
         """

124 <schlüsselwort> :
          auto      | break     | case
        | char      | continue  | default
        | do        | double    | else
        | enum      | extern    | float
        | for       | goto      | if
        | int       | long      | register
        | return    | short     | sizeof
        | static    | struct    | switch
        | typedef   | while     | union
        | unsigned
```

17. Literatur

/1/ Brian W. Kernighan; Dennis M. Ritchie:
"The C Programming Language"
Prentice-Hall Inc. Englewood Clifs, New Jersey 07632,
1978

/2/ Matthias Clauss; Günter Fischer:
"Sprachbeschreibung C-Sprache unter den Betriebssystemen
MUTOS 80 und MUTOS 1630"
Systemunterlagendokumentation
VEB Robotron Buchungsmaschinenwerk Karl-Marx-Stadt, 1985

/3/ Brian W. Kernighan; Dennis M. Ritchie:
"Programmieren in C - mit dem Reference Manual in deut-
scher Sprache"
Carl Hanser Verlag München Wien, 1983

Anlage 1 Unterschiede in den Compilerimplementationen

In der Tabelle sind die Implementationsabhängigkeiten aufgeführt, deren Kenntnis unter dem Blickwinkel der Portabiblität von Interesse sind.

	MUTOS 1700	MUTOS 80	ESER
Zeichensatz:			
KOI-7-Code (ASCII-Code)	x	x	
DKOI-Code			x
Implementationsabhängig erlaubte Datentypangaben:			
- unsigned short	ja	ja	ja
- unsigned long	nein	nein	ja
- unsigned char	nein	nein	ja
- float, double	ja	nein	ja
Vorzeichenausdehnung bei der Konvertierung von char	ja	nein	nein
Anzahl der Registervariablen	3	6	6
Externe Bezeichner:			
- signifikante Länge	7	7	7
- Groß- und Kleinbuchstaben	ja	ja	nein*

* Kleinbuchstaben werden in Grossbuchstaben konvertiert.

Anlage 2 Operatoren in C

Vorrang und Assoziativität aller Operatoren sind in der Tabelle zusammengefaßt.

L steht für Links- und R für Rechtsassoziativität.

Operatoren in einer Zeile besitzen den gleichen Vorrang, der ins%gesamt von oben nach unten abnimmt.

Die Operatoren ++ und -- sind als Präfix- oder Postfixoperatoren zu verwenden und haben verschiedene Nebeneffekte.