Ergänzendes Skript zur Vorlesung Prog 1 &2 an der FH-Bingen

C-Progammierung

Bei Prof. Dr. M. Mengel

Fachbereich 2

Studiengänge :

Angewandte Informatik

Ingenieur-Informatik

Elektrotechnik

Inhaltsverzeichnis

Programmieren 1
-
Ziele

- Allgemeines Verständnis von Algorithmen bzw. deren Formulierung

- Entwicklung von Algorithmen
  Motto: vom Problem zur Problemlösung

- Erlernen der Programmiersprache C
  (Sprachelemente von C findet man auch in anderen Programmiersprachen – C++, Java, JavaScript, PHP, ...)

- Beschreibung von Algorithmen in C

- Strukturierter Entwurf und strukturiertes Programmieren

Vom Problem zur Problemlösung

Aufgabe:

Geben Sie die einzelnen Schritte an, die Sie durchführen müssen, wenn Sie an einem Münzautomaten telefonieren.

==> Vorschlag für Vorgehensweise:

Beschreibung schrittweise entwickeln und immer weiter verfeinern

Aufgabe zerlegen und strukturieren

Problem: wie genau? was ist die kleinste Einheit?

Vorbedingungen und Nachbedingungen formulieren

Für wen wurde die Beschreibung gemacht?

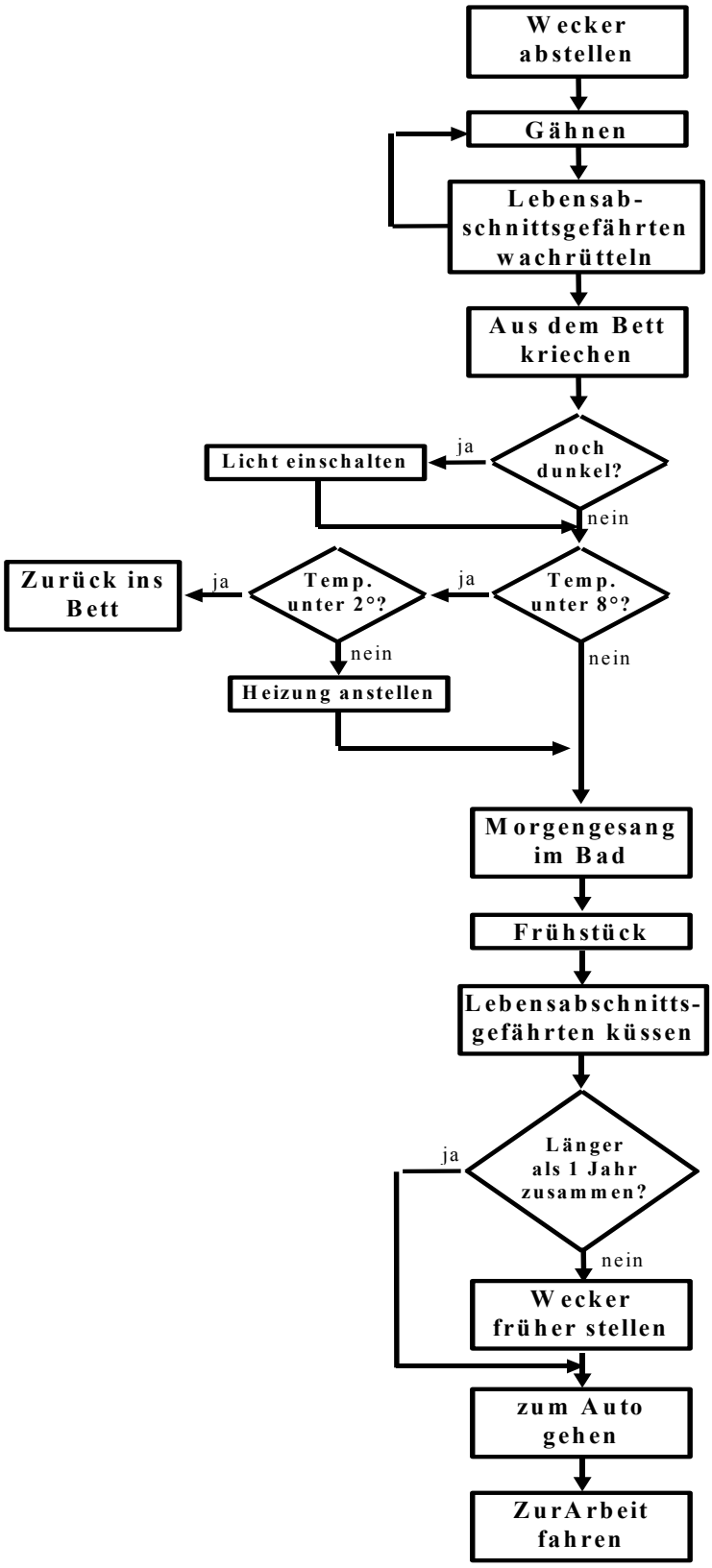
Die letzte Fragestellung führt uns zu unserem **Bezugssystem** oder einer **formalen Verfahrensbeschreibung**

- Ablaufplan
- Struktogramm
- Programmiersprache

Wir werden Verfahren/Beschreibungen betrachten, die von einer Maschine (=Computer) ausgeführt werden können.

Aufgabe

- Geben Sie den Algorithmus, der Ihren morgentlichen Aufbruch zur FH beschreibt
- Tic-Tac-Toe (3 Gewinnt) - Beschreiben Sie das Spiel Tic-Tac-Toe, so dass es anschließend implementiert werden könnte.


Wecker abstellen
Gähnen
Lebensab-schnittsgefährten wachrütteln
Aus dem Bett kriechen
noch dunkel?
ja
Licht einschalten
nein
Zurück ins Bett
ja
Temp. unter 2°?
ja
Temp. unter 8°?
nein
nein
Heizung anstellen
Morgengesang im Bad
Frühstück
Lebensabschnitts-gefährten küssen
Länger als 1 Jahr zusammen?
ja
nein
Wecker früher stellen
zum Auto gehen
ZurArbeit fahren

Algorithmusbegriff

Ein Algorithmus ist - intuitiv gesehen - eine Anleitung zur Lösung eines Problems, wie auch ein Kochrezept, ein Strickmuster oder eine Reparatur- oder Montageanweisung.

Was einen Algorithmus jedoch von einem Kochrezept abhebt, ist die Tatsache, dass er von einer Maschine ausgeführt werden muß. Dies bedeutet, dass ein Programm so geschrieben sein muß, dass der Computer es ausführen kann, ohne es inhaltlich zu verstehen. Wie bei jeder anderen Sprache müssen hier die Regeln der **Syntax** (Einhaltung der formalen Regeln) und die der **Semantik** (Bedeutung der Konstrukte untereinander) eingehalten werden.

Er bildet eine Einheit aus den dafür festgelegten Datenstrukturen und Operationen.

Eigenschaften eines Algorithmus

Algorithmen die wir betrachten müssen folgende grundsätzlichen Eigenschaften erfüllen:

Allgemeinheit

Ein Algorithmus löst im allgemeinen eine Klasse von Problemen. Die Auswahl des Einzelfalles erfolgt meist über Parameter.

Ein Algorithmus entsteht oft dadurch, dass ein bestimmter Fall gelöst wird und andere Situationen darauf zurück geführt werden.

Determiniertheit

Algorithmen sind in der Regel determiniert, d.h. bei gleichen Eingabewerten und Startbedingungen erfolgt stets dasselbe Ergebnis. Beispiele für nicht determinierte Algorithmen sind z.B. stochastische Simulationen mit Hilfe von Zufallszahlen.

Determinismus

Ein Algorithmus heißt deterministisch, wenn zu jedem Zeitpunkt seiner Bearbeitung höchstens eine Möglichkeit der Fortsetzung besteht.

Terminierung

In der Regel sind nur solche Algorithmen von Interesse, die für jede Eingabe nach endlichen vielen Schritten terminieren, d.h. anhalten. Eine Ausnahme sind hier Betriebssystem-Funktionen und Prozeß-Steuerungen.

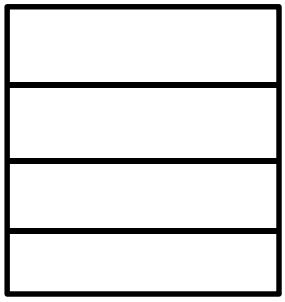
Beschreibung von Algorithmen

Es gibt mehrere Möglichkeiten, einen Algorithmus anzugeben:

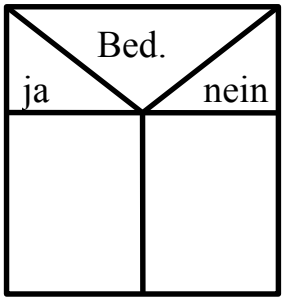
- umgangssprachlich
- Pseudocode
- Ablaufplan, **Struktogramm**
- Mathematische Beschreibung
- Programmiersprache

Ein Struktogramm ist ein Hilfsmittel mit dem Algorithmen in formalisierter Form dargestellt werden können. Die graphische Darstellungsmethode zeichnet sich insbesondere durch die überschaubare Anzahl von elementaren Strukturelementen aus, die bel. miteinander kombiniert werden können und durch ihre Übersichtlichkeit.

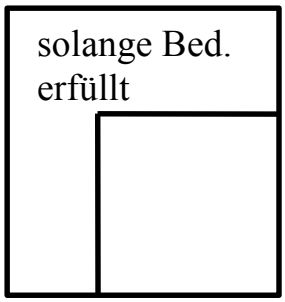
Ein Strukturblock ist entweder vollständig in einem anderen enthalten oder er steht außerhalb von diesem:



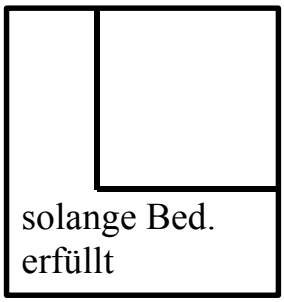
Sequenz
(Folge von
Anweisungen):
-Zuweisung
-Eingabe
-Ausgabe



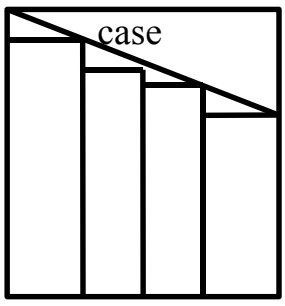
Bedingung
(Verzweigung)



Abweisende
Schleife



nicht abweisende
Schleife



1-aus-n
Auswahl

Vom Algorithmus zum Programm

Wenn man einen Algorithmus auf einem Computer ausführen möchte, muss dieser in ein Programm überführt werden. Neben der Beschreibung des Algorithmus in einer Programmiersprache bedarf es verschiedener Werkzeuge um ein lauffähiges Programm zu erhalten.

Je nachdem ob eine Compiler- oder eine Interpreter-Sprache zur Formulierung eines Programms verwendet wird, gibt es bei der Programmentwicklung einen Entwicklungszyklus:

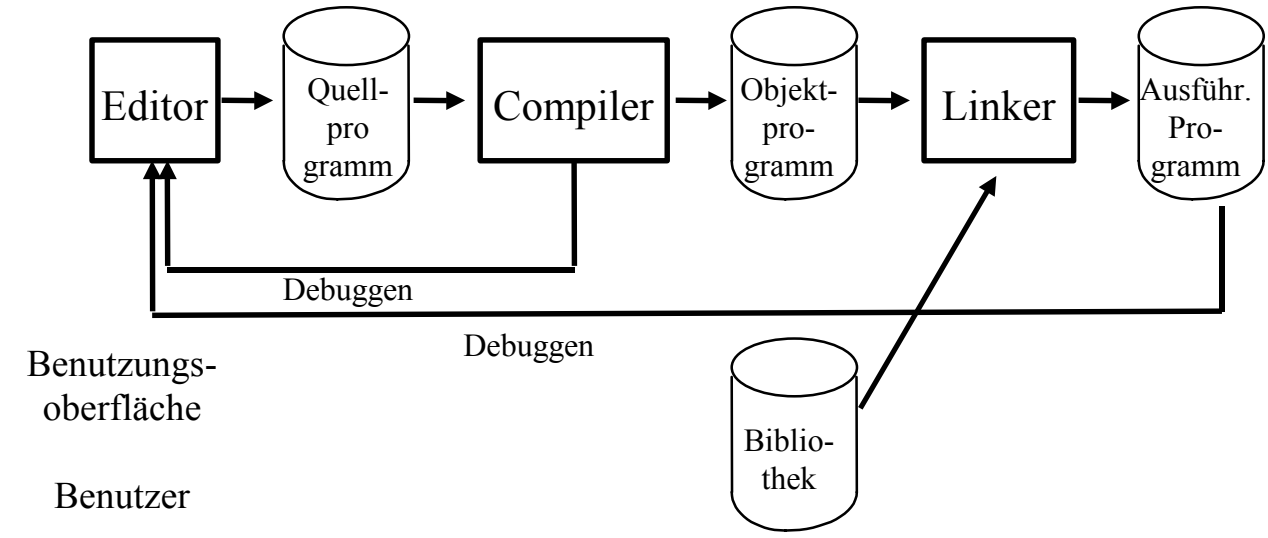


Abbildung: Programmentwicklung mit einer Compiler-Sprache

Bei Programmen in einer Compilersprache muß der Code von einem Compiler zuerst vollständig in Maschinensprache übersetzt werden, bevor er ausgeführt werden kann, während bei Interpretersprachen jede Anweisung bei der Ausführung interpretiert wird.

In jedem Fall muß zuerst der Programmtext (Quelltext - *.pas-, *.c- oder *.cpp-Dateien) mit einem Texteditor in eine Datei geschrieben werden. Im Falle einer Compilersprache wird dieser Quelltext vom Übersetzer (Compiler) gelesen, der daraus eine Objektdatei (*.obj-Dateien) macht, die aus Maschinenbefehlen besteht, aber noch „verschiebbar“ ist, d.h. noch nicht auf endgültige Adressen im Hauptspeicher festgelegt ist.

Der Binder (Linker) bindet mehrere Objektdateien zu einem ausführbaren Programm zusammen. Dazu können auch Objektdateien aus Bibliotheken. Zuletzt wird das Lauffähige Programm (executable) gestartet, das heißt, es wird in den Hauptspeicher gebracht und bekommt den Prozessor zugeteilt.

Kodieren bedeutet aber immer auch, dass der implementierte Code verbessert werden muß. Es müssen nach und nach syntaktische und unter Umständen nach Tests semantische Fehler verbessert werden. Im schlimmsten Fall muß der Algorithmus komplett neu kodiert werden. Dann hat man allerdings vermutlich zu Beginn der Programmentwicklung einen grundsätzlichen Fehler begangen.

Schritte zur algorithmischen Lösung von Problemen

Die algorithmische Lösung eines Problems besteht nicht in einem einzigen Schritt, nämlich dem Schreiben eines C-Programms, sondern es lassen sich mehrere Teilschritte identifizieren, die beim Entwurf von Programmen auftreten:

1. Formulierung des Problems
2. Die formale, abstrakte Beschreibung des Problems (z.B. mit Hilfe der Prädikatenlogik als Vor- und Nachbedingung)
3. Entwurf eines Lösungsalgorithmus
4. Kontrolle und Nachweis der Korrektheit des Lösungsalgorithmus
5. Die Übertragung des Lösungsalgorithmus in eine Programmiersprache (sogenannte Kodierung)
6. Fehlerbeseitigung und Tests
7. Die Effizienzuntersuchung
8. Die Dokumentation

Allgemein und insbesondere bei C ist es wichtig, dass man beim Einstieg in die Programmiersprache besonders sorgfältig vorgeht und dadurch langwierige und mühsame Phasen der Fehlersuche und -verbesserung vermeidet.

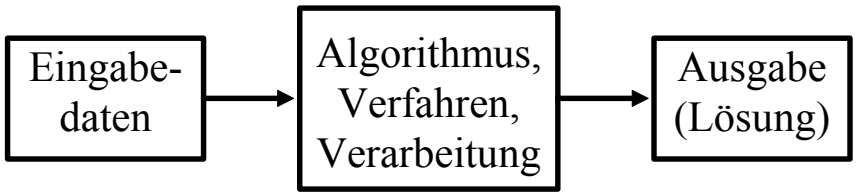


Abbildung: Schematische Darstellung eines Algorithmus (sehr grob):

Ziel höherer Programmiersprachen wie FORTRAN, PASCAL, C, C++, Java:

Der Programmiervorgang ist ein iterativer Vorgang bei dem man eine Vorgehensweise nach formalen Aspekten aufschreiben muß. Der Entwickler kann also umso besser Programme entwickeln je komfortabler die Programmiersprache und die Programmierwerkzeug sind. Die höheren Programmiersprachen zielen insbesondere auf die folgenden Aspekte:

- Hoher Bedienkomfort und Sicherheit bei der Entwicklung von Programmen
- Verwendung von symbolischen Namen (Variablen) - der Mensch kann sich Namen leichter merken als Speicheradressen
- Entlastung des Programmierers von systemspezifischen, prozessorabhängigen (spez. Befehle, spez. Register, ...) Dingen durch abstrakte, mathematische Beschreibung
- Verwendung von Konstrukten zur Entwicklung komplexer, wiederverwendbarer Programme (Unterprogramme) - wiederverwendbar hat in diesem Zusammenhang mindestens zwei Bedeutungen:
  - das entwickelte Programm soll auf unterschiedlichen Maschinen/Rechner ablauffähig sein und
  - das entwickelte Programmteil sollte von der Konzeption und Kodierung so angelegt sein, dass es dazu geeignet ist, ähnliche Problemstellungen wie die ursprüngliche zu lösen oder aber einfach auf eine ähnliche Problemstellung anpassbar sein

- Konzepte zur sicheren und schnellen Programmentwicklung sollen bereits Bestandteil der Programmiersprache sein (zu mindest teilweise) - Beispiele: vordefinierte Datentypen, Mechanismen zur Beschreibung eigener anwendungsspezifischer Datenstrukturen, Kontrollkonstrukte (möglichst KEINE Sprunganweisungen), Variablennamen müssen vor der Verwendung festgelegt werden und müssen eindeutig sein.
- Es existieren bereits vorgefertigte, komplexe Funktionen, die häufig vorkommende problemorientierte Aufgaben lösen (Programmbibliotheken, Modulbibliotheken).

Programmiersprachen sind streng nach vorgegebenen Regeln (Grammatiken) aufgebaute formale Sprachen, in denen der Programmierer seine Rechenvorschriften ablegt.

Das Programm wird somit in einer maschinenunabhängigen Programmiersprache geschrieben. Der entsprechende Compiler übersetzt das Programm aus einer höheren Programmiersprache in die Assembler- oder Maschinensprache.

Bei einer Interpretersprache werden die Anweisungen bei der Ausführung in die maschinenspezifischen Befehle umgestzt und ausgeführt. Wird eine Anweisung wiederholt ausgeführt, so muß sie auch wiederholt umgesetzt werden.

Warum C?

- Hohe Praxisrelevanz (Unix, ...)
- Es existieren sehr viele moderne Entwicklungen in C
- gute Ausgangsbasis für alle anderen aktuellen Entwicklungen im Bereich der Programmiersprachentwicklungen (C++, Java, C#, ...)
  - sie besitzen ähnliche Konstrukte
- Bei dem Erlernen der Programmiersprache soll der Blick geschärft werden für die Konzepte von Programmiersprachen (Generalisierung, Abstraktion, ...)
- C gilt als einfache (= mit wenigen Programmkonstrukten) und flexible Programmiersprache –
  - VORSICHT:
    Die syntaktische und semantische Korrektheit eines Programmes muß der Programmierer noch mehr als bei Pascal sicherstellen!
- Denken Sie daran, dass ein Programm einfach zu lesen und pflegen sein soll (erweiterbar, wiederverwendbar, wartbar, robust) und daher nicht jede Möglichkeit der Programmiersprache im Sinne eines guten Programmierstils zur Übersichtlichkeit oder Verständlichkeit des Programmcodes beiträgt!

Formalismen zur Beschreibung von Programmiersprachen

Um eine Programmiersprache wie C vollständig und präzise beschreiben zu können, bedient man sich in der Literatur häufig sogenannter Syntaxdiagramme („Eisenbahndiagramme“) oder der Backus-Naur-Form (BNF) bzw. der Erweiterten BNF (EBNF).

Syntaxdiagramme

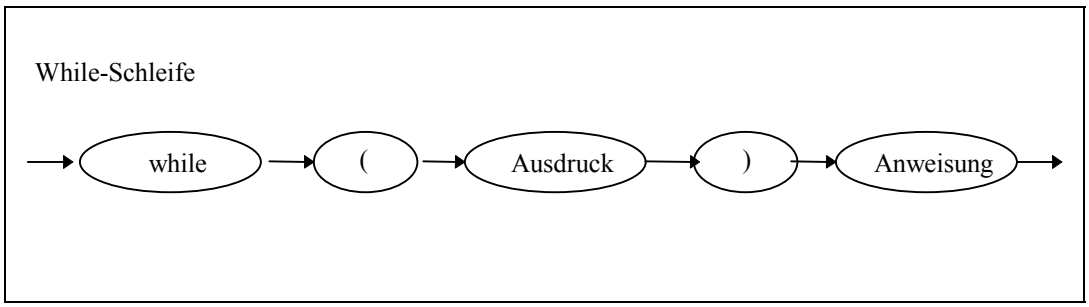
Syntaxdiagramme dienen der Beschreibung einer (formalen) Programmiersprache. Es ist ein Ausdrucksmittel, mit dem man gleichzeitig eindeutig die erlaubte Reihenfolge als auch alle möglichen Reihenfolgen der verschiedenen Elemente einer Programmiersprache beschreibt.

In den sog. Ableitungsregeln werden Sequenzen (Folgen) und Alternativen (Auswahl) beschrieben. Außerdem ist es begrenzt möglich, anzugeben, wie oft Elemente auftreten; man kann beschreiben, ob ein Element an einer bestimmten Stelle (zwingend) vorkommen muß, es optional ist (vorkommen kann aber nicht vorkommen muß) oder beliebig oft vorkommt. Diese Möglichkeiten können beliebig miteinander kombiniert werden.

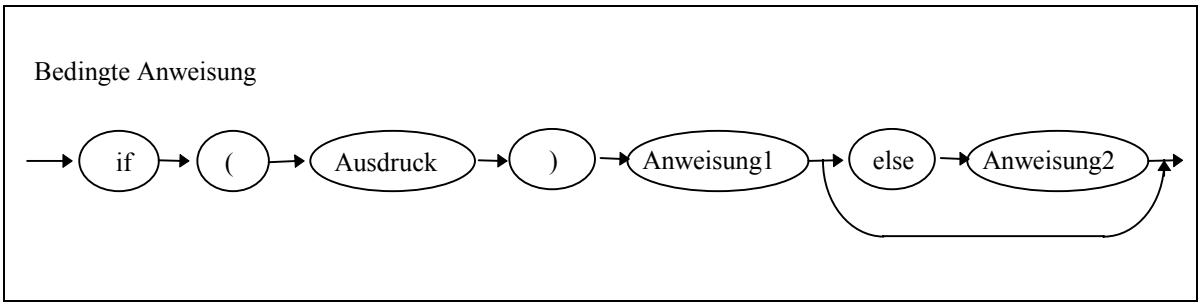
Dabei bezeichnet man die kleinsten (auf eine bestimmte Art nicht weiter teilbare) Teile einer Programmiersprache als Token oder Terminale. Beispiele für die Token sind alle vordefinierten Schlüsselworte in einer Programmiersprache (z.B. while, for, int, if, =, &&, {, }, [,], ...) oder die Variablen. Sie können nicht weiter zerlegt oder abgeleitet werden.

Anders ist es bei den Nonterminalen. Damit wird alles andere bezeichnet, was als komplexes Gebilde benannt werden kann aber nach Regeln (der Syntax der Programmierspache) weiter zerlegt werden kann (z.B. eine Anweisung, alle Teile einer while-Schleife, Unterprogramme, ...). Nonterminale stehen damit immer am Anfang einer Syntaxregel.

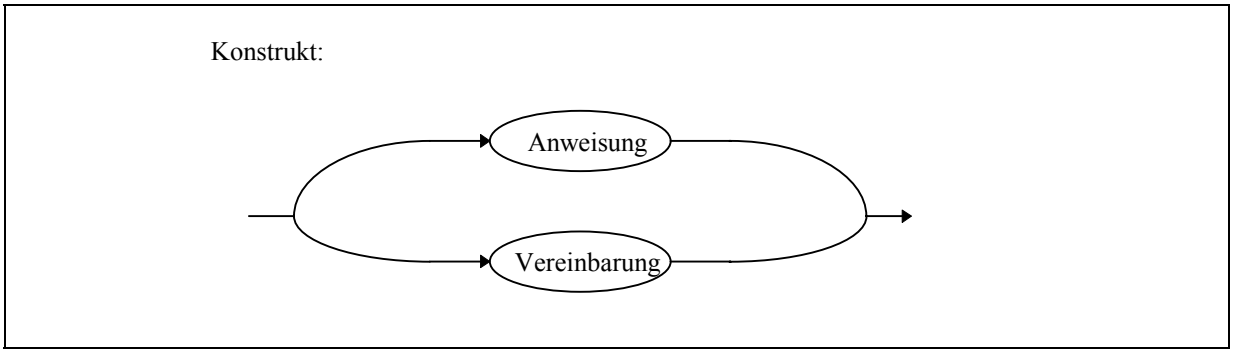
Sequenz:



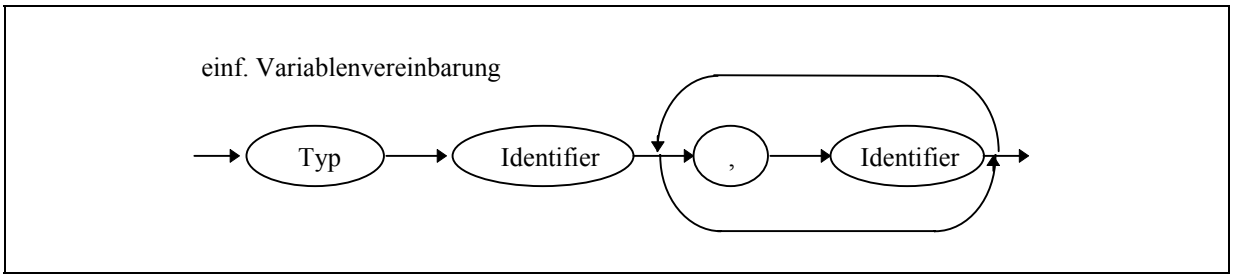
Optionale Elemente:



Alternativen:



Beliebige Wiederholung



Backus-Naur-Form (BNF)

Mit Hilfe dieser Metasprache kann man die Syntax einer Programmiersprache als Ableitungsregeln beschreiben. Im folgenden werden die Elemente vorgestellt:

< >

Innerhalb dieser Klammersymbole steht der Name einer Gruppe von Sprachelementen. Dieser Name taucht dann in einer anderen Ableitungsregel auf der linken Seite auf und wird weiter aufgelöst (Bsp.: <Anweisung>). Falls in einer Syntaxbeschreibung mehrmals das gleiche Sprachelement vorkommt und unterschieden werden muß, wird der Name zusätzlich mit einem Namen versehen (Bsp.: <Anweisung1>, <Anweisung2>).

::=

Das links von diesem Definitionszeichen stehende Sprachelement wird durch den rechts davon stehenden syntaktischen Ausdruck definiert.

|

Die vor und nach diesem Zeichen stehenden Sprachelemente schließen sich gegenseitig aus. Es werden entweder die vor dem Zeichen stehenden Elemente oder nach diesem Zeichen stehenden Elemente verwendet.

Ø

Das Leere Zeichen

Alle übrigen Zeichen und Worte, die außerhalb der spitzen Klammern stehen sind Schlüsselworte (Endsymbole) und müssen stets angegeben werden.

In der erweiterten BNF können zusätzlich weitere Elemente vorkommen:

[]

In eckige Klammern eingeschlossene Sprachelemente sind optional.

{ }

In geschweifte Klammern eingeschlossene Elemente können wiederholt vorkommen.

*****

Mit * versehene geschweifte Klammern können beliebig oft vorkommen (auch keinmal).

+

Mit + versehene geschweifte Klammern müssen mindestens einmal vorkommen.

 Aufgabe:

Geben Sie die Regel an, wie man syntaktisch korrekt eine beliebige Dezimalzahl mit diesem Formalismus beschreibt.

Daten und Variablen

Konzept der Variablen

Eine der grundlegendsten Konzepte höherer Programmiersprachen sind Variablen. Variablen haben:

- eine eindeutige Bezeichnung (Identifier),
- einen eindeutigen Typ (Wertebereich) und
- zur Laufzeit einen Ort im Speicher.

Im Speicher steht der Wert der Variable, der hoffentlich definiert ist.

Dies wollen wir uns noch einmal verdeutlichen:

Der Ort ist im Rechner eine adressierbare Speicherzelle, der Wert wird durch die darin enthaltene Bitfolge repräsentiert (Die Bedeutung der Bitfolge ergibt sich durch die Festlegung des Typs der Variable!). Die Kombination aus symbolischem Namen und dem Paar Adresse und Inhalt - also Ort und Wert - wird als Variable bezeichnet.

Beispiel:

Die Zuweisung:

```
meineZahl = 5;
```

bedeutet, dass im Speicher an dem der Variablen „meineZahl“ zugeordneten Speicherplatz, der beispielsweise die Adresse 45110, besitzen könnte, der Wert 5 als Bitmuster 00...0101 eingetragen wird.

Dieser Zugriff gilt sowohl für die einfachen Grundtypen **int, float** und **char** als auch für die strukturierten Typen (etwa **Felder - eine Zusammenfassung von Datenobjekten des gleichen Typs -**).

Einzelne Elemente einer entsprechenden Feld-Variablen können über einen Index oder mehrere Indices angesprochen werden.

Jede Variable muß vor ihrem Gebrauch deklariert (vereinbart) werden, und sie sollte initialisiert (mit einem Startwert versehen) werden. Die Variablendeklaration prüft der Compiler und gibt, falls eine Variable nicht deklariert ist, dies als Fehler an.

Möglicherweise folgenschwerer für die Programmtests sind versäumte Initialisierungen von Variablen, die der Compiler nicht erkennt. Die Variablen sind (je nach Compiler) in einem undefinierten Zustand!.

Grunddatentypen in C

In C kennt man drei Grunddatentypen:

- int (ganze Zahlen),
- float (Gleitpunktzahlen)
- char (Zeichen).

Beispiel:

```
int i, j, k;
float f;
char s;
i = 1;
f = 1.0;
char = '1';
```

Bei den Zahlen gibt es zusätzlich folgende Varianten bezüglich der Rechengenauigkeit:

short [int]	ganze Zahlen, im Allg. 16 Bit lang
long [int]	ganze Zahlen, im Allg. 32 Bit lang
unsigned [int]	das Vorzeichen Bit gehört zum Wert,
unsigned short	d.h. es gibt nur positive Zahlen
unsigned long	
double	rationale Zahlen
	(bei double: doppelte Genauigkeit gegenüber float)

Weitere Datentypen

Neben den elementaren Typen in C gibt es zusätzlich die Möglichkeit, abstrakte (= anwendungsspezifische) Typen zu deklarieren (z.B. Temperatur, Luftdruck, Entfernung, ... Position, ...).

Sie stehen im Deklarationsteil nach den Konstanten- und vor den Variablendeklarationen und beginnt mit dem Schlüsselwort **typedef**.

Variablen mit konstantem Wert

Im Deklarationsteil in C können konstante Zeichenfolgen oder Zahlen durch eine Variable mit konstantem Wert (constant identifier) bezeichnet werden. Wie bei einer „normalen“ Variable muss der Name eindeutig sein. Im Gegensatz zu Variablen darf bzw. kann der Wert einer Konstanten nicht verändert werden.

Konstanten

Konstanten und logische Datentypen gibt es zunächst nicht. Sie können allerdings sehr einfach festgelegt werden.

Mit Hilfe der Präcompileranweisung **#define** können beliebige Literale (Werte) mit einem Symbol bezeichnet werden.

C könnte man somit folgendermaßen um einen logischen Datentyp erweitern:

```
#define FALSE      0
#define TRUE       !FALSE

int logical, a, b;
...
logical = TRUE;
a = 5;
b = 3;
...
if (logical && a == b)
   printf (“Aussage ist wahr“);
```

Bemerkung: In C gilt der Zahlenwert **0 als logisch falsch**, jeder Wert **ungleich 0 als logisch wahr**.

Die Negation (!) entspricht dem Test des Ausdruck auf 0
(== 0)

Feld-/Array-Definitionen

Neben einzelnen Variablen können auch Felder (Arrays) von einzelnen Elementen des gleichen Typs definiert werden.

Die Länge und Anzahl der Dimensionen wird bei der Deklaration der Variablen in eckigen Klammern angegeben und an den Bezeichner (Identifier) gehängt.

Für jede Dimension gibt es ein eigenes Klammerpaar. Die Dimensionslänge darf nur als Konstante oder konstanter Ausdruck angegeben werden, der zum Zeitpunkt des Übersetzens ausgewertet werden kann. Bei einem Zugriff auf eine Feldvariable muß deren Dimension bekannt sein!

```
char farbe[10], brett[8][8];
```

Typfestlegung und Reservierung von Speicherbereich, dessen Inhalt **nicht** definiert ist!

Beispiel farbe[10]:

Index	0	1	2	3	4	5	6	7	8	9
Wert	?	?	?	?	?	?	?	?	?	?

Wurde ein Feld der Größe n angelegt, so werden die einzelnen Elemente unter Verwendung eines Index in einer eckigen Klammer angesprochen.

Beim Zugriff sind als Indexangabe beliebige arithmetische Ausdrücke erlaubt.

In C kann man den Index auch als Distanzangabe eines Elementes zum Beginn des Arrays auffassen.

Besonders häufig werden eindimensionale Zeichenfelder (Strings) benötigt.

Operatoren in C

Operator	Bedeutung	Verarbeitungsreihenfolge

`*`
`/`
`%`
	Multiplikation
	Division
	Rest bei Ganzzahldivision
		Links -> Rechts

`+` `-`	Addition Subtraktion	L -> R
`<<` `>>`	Linksshift Rechtsshift	L -> R

`<`
`<=`
`>`
`>=`
	kleiner
	kleiner oder gleich
	größer
	größer oder gleich
		L -> R

`==` `!=`	gleich ungleich	L -> R
`&&`	logische UND	L -> R
`||`	logische ODER	L -> R
`!`	Logische Negation	L -> R
`=, +=, ...`	Zuweisungen	R -> L
`? :`	Bedingter Ausdruck	R -> L

Typumwandlung (allgemein):

- short, char, Aufzähltypen können immer an Stelle von int benutzt werden
- es wird auf den nächst größeren Datentyp ausgeweitet:
  - float -> double -> long double
  - signed -> unsigned, int -> long
- abschließend erfolgt eine Typanpassung an den Typ auf der linken Seite

Welchen Wert hat b nach der Zuweisung?

```
int b;
b = 2 / 3 +  0.333;
```

Typumwandlung (explizit)

Die Datentypumwandlung oder -konvertierung (casting) bzw. der cast-Operator enthält in runden Klammern eine skalare Typbezeichnung ohne Speicherklasse und geht dem Operanden unmittelbar voraus.

Beispiel:

```
int n = 5;
double y;

y = sqrt ( (double)n );
```

n ist eine integer-Zahl. Die Funktion sqrt( ) erwartet jedoch als Argument eine Gleitpunktzahl doppelter Genauigkeit. Die Typumwandlung von n wird durch den cast-Operator **(double)** bewirkt.

Die Typumwandlung im Zusammenhang mit dem cast-Operator hat in der Folge für die Verwendung der Variable keine Bedeutung.

Es liegt in der Sorgfaltspflicht des Programmierers sicherzustellen, dass die Argumente bei Funktionen und Prozeduren vom richtigen Typ sind (Welche Möglichkeiten hat er oder sie?).Nicht alle Compiler geben Hinweise auf falsche Typen in Form von Warnung an.

Anweisungen

Wie bereits bei den Struktogrammen gesehen, gibt es prinzipiell nur eine Handvoll Anweisungen, die allerdings in einer Programmiersprache beliebig kombiniert werden können:

- Zuweisungen/Inkremente/Ein-Ausgabe-Anweisungen
- Bedingte Zuweisungen
- Schleife (for-, abweisende, nicht abweisende Schleife)
- Fallunterscheidungen
- Unterprogramme zu Strukturierung

Wir werden uns nun die Anweisungen im einzelnen genauer ansehen und dabei immer darauf achten, daß wir beabsichtigen, strukturierte Programme zu erstellen und somit zumindest nicht am Anfang jeden Trick und Kniff, die in C möglich sind und die es in C zahlreich gibt, ausprobieren und vorstellen wollen.

Zuweisung (Assignment):

```
<variable> = <expr>;
```

Die Variable erhält den Wert des Ausdruckes expr

Folge von Anweisungen (Sequenz):

```
{
[<definitions>]
<statement1>
<statement2>
<statement3>
...
<statement n>
}
```

Bedingung:

```
if ( <condition> )
  <statement1>
[ else <statement2> ]
```

Bedingte Ausführung von Anweisungen bei Unterscheidung mehrerer Fälle:

```
if ( <condition1> )
  <statement1>
else if ( <condition2> )
  <statement2>
...
else if ( <condition n> )
  <statement n>
[ else <statement n+1> ]
```

Fallunterscheidung

```
switch (<expr>)
{
  case <const term1> : <statement1> [break;]
  case <const term2> : <statement2> [break;]
...
  case <const term n> : <statement n> [break;]
  [default          <statement def> [break;] ]
}
```

Leere Anweisung

```
;
```

In C gehört das Semikolon zur Anweisung!

Schleifen

1. while Schleife

Test der Bedingung vor Schleifendurchlauf

```
while ( <expr> )
  <statement>
```

2. do-while Schleife

Test der Bedingung nach Schleifendurchlauf

```
do
  <statement>
while ( <expr> );
```

3. for Schleife

Test der Bedingung vor Schleifendurchlauf

```
for ( [<expr1>]; [<expr2>];[<expr3>])
<statement>
```

expr1 wird einmal vor dem 1. Schleifendurchlauf ausgewertet (= Initialisierung),

expr2 wird zu Beginn jedes neuen Schleifendurchlaufes ausgewertet,

expr3 wird an Ende eines jeden Schleifendurchlaufes ausgewertet.