Teil II: Vom Algorithmus zur Programmiersprache

1. Algorithmen, Programme, Sprachen
2. Programm- & Sprachelemente
3. Modula-2

Organisation der Vorlesungsunterlagen

Strukturierung

Zwecks weiterer Strukturierung werden verschiedene Symbole und Zeichen verwendet, die als „Highlighter“ dienen sollen

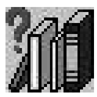
Merker:
Kommentierung und Hervorhebung bestimmter Inhalte

Warnung:
Hinweis auf häufige Probleme, Fehlerquellen

Achtung – Hinweis:
Hinweiszeichen für verschiedene wichtige Inhalte und Zusammenhänge

Zusammenfassung:
Resumee – Darstellung von Inhalten des besprochenen Abschnitts

Vertiefung:
Betrachtung von Details (Hintergrund, Theorie, etc.)



Literatur:
Hinweise auf besondere oder wichtige Literaturquellen

Algorithmen, Programme, Sprachen

<u>Der Algorithmusbegriff</u>

Definition:

Ein Algorithmus ist eine Vorschrift, eine Anleitung zur Lösung einer Aufgabenstellung, die so präzise formuliert ist, dass man sie im Prinzip „mechanisch" ausführen kann.

Das heisst,

es werden Verarbeitungsvorschriften als digitale Nachrichten formuliert, die interpretiert werden müssen.

Randbedingungen:

→ verschiedene Formalismen sind möglich
→ gemeinsames Sprachverständnis ist Voraussetzung

Der „ganze Alltag" besteht aus Algorithmen:

- Gebrauchsanweisung, Bedienungsanleitung
- Rezepte zur Zubereitung von ...
- Anleitungen zum „Basteln"
- Vorschriften nach denen man sich richtet (oder auch nicht)
- Prozessabläufe (Waschmaschine)

Aha:
Algorithmus ➨➨➨➨ Reproduzierbarkeit

Beispiel 1: Rezept für Marmorkuchen

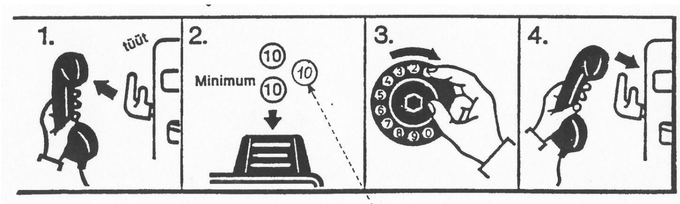
Zutaten: **mehr oder minder eindeutige Eingabeparameter**

250g Mehl, 150g Zucker, 120g Butter, 1/2 Päckchen Backpulver, 3 Eier, 1/4 l Milch, 1 Löffel Kakaopulver, eine Prise Salz, gestossene Vanille

Zubereitung:

1 Die Butter mit den Eigelben schaumig rühren, nach und nach abwechselnd Milch, Zucker, das mit Backpulver gesiebte Mehl, Vanille, Salz und zum Schluss den festen Eischnee untermischen. 2 Den Teig halbieren und unter die eine Hälfte den Kakao mischen. 3 Den hellen Teig in eine gefettete, mit Mehl ausgestreute Form füllen, den dunklen Teig darauf verteilen, eine Gabel mehrmals spiralig durch den Teig ziehen. 4 Den Kuchen im vorgeheizten Backofen bei mittlerer Hitze (ca. 180°) backen. Backzeit ca. 50 min.

Beispiel 2: Anleitung Münzfernsprecher



Elementare Eigenschaften eines Algorithmus:

- ♣ Ausführbarkeit und Reproduzierbarkeit
- ♣ Prozessor und Prozess
- ♣ schrittweise Ausführung elementarer Operationen
  - was heisst „elementar“;
    Abstraktionsniveau (minimaler Grad an Detaillierung)
- ♣ Endlichkeit der Beschreibung
- ♣ Termination (Ausführungszeit endlich)
- ♣ unabhängig von Programmiersprache und Computer

weitere Merkmale von Algorithmen:

- ♣ Ausführung von Algorithmus-Teilen sequentiell/parallell
- ♣ Operationen auf benannten Objekten
- ♣ $\exists$ Eingabe-Objekte zur Ausführung des Algorithmus
- ♣ $\exists$ Ausgabe-Objekte nach Ausführung des Algorithmus
- ♣ Algorithmen und Objekte können elementar aber auch zusammengesetzt sein.
- ♣ $\exists$ nicht-elementare Unteralgorithmen, die durch ihren „Namen“ aufgerufen werden. $\rightarrow$ Prozedurkonzept
- ♣ $\exists$ Wiederholungsanweisungen $\rightarrow$ Mehrfachausführungen
- ♣ $\exists$ bedingte Ausführung von Verarbeitungsschritten

Einige Begriffe:

- ♦ Sequentialität
- ♦ Parallelität
- ♦ Kollateralität (sequentiell & parallel)

- ♠ Termination
  - terminierender Algorithmus kommt in endlich vielen Schritten zum Ende
- ♠ Determinismus
  - derterministischer Algorithmus besitzt eindeutig vorgeschriebenen Verlauf
- ♠ Determiniertheit
  - derterminierter Algorithmus führt zu eindeutigem Ergebnis
  - derterministischer Algorithmus ist immer determiniert (verkörpert also eine Abbildung)
  - nicht-derterministischer Algorithmus ist bisweilen determiniert

Aussagen über Algorithmen:

- Voraussetzungen
  - unter welchen Bedingungen arbeitet der Algorithmus?
    - Menge aller erlaubten Eingaben
    - Menge aller möglichen Ausgaben (bei erlaubten Eingaben)
  - Was geschieht bei falscher Eingabe
- Termination
  - endet ein Algorithmus für **alle** möglichen Eingaben? Beweis!
  - Ist es möglich, den Algorithmus in einen nicht-endenden Zyklus zu bringen?
- Korrektheit
  - „Übereinstimmung“ Algorithmus - Prozess?
  - liefert der Algorithmus bei Ausführung die richtige Ausgabe als Funktion der Eingabe?

Aussagen über Algorithmen: (cont‘d)

- Aufwand/Effizienz
  - Speicherplatzbedarf?
  - Ausführungszeit?
  - Abhängigkeit der Ausführungszeit von der Eingabe?

Der Korrektheitsbeweis für einen Algorithmus umfasst den Beweis der Termination und den Beweis der Korrektheit!

Urfrage zur Algorithmisierbarkeit/Berechenbarkeit:
Was lässt sich algorithmisch lösen (berechnen)?
Gibt es einen Algorithmus, der in endlich vielen Schritten mit der Ausgabe = f(Eingabe) stoppt, d.h. die (bekannte) Abbildung f: (Eingabe) $\longrightarrow$ (Ausgabe) realisiert?

<u>Form und Interpretation von Nachrichten</u>

Die Form einer Nachricht wird durch eine Sprache festgelegt.

Zeichen $\longrightarrow$ Folgen von Zeichen

Wörter $\longrightarrow$ Folgen von Wörtern

Sätze $\longrightarrow$ Folgen von Sätzen

Sprache

Aufbau von Strukturen - Sprache:

- Grundelement „Zeichen“
  - bildet die kleinste Einheit (bzgl. best. Abstraktionsebene)
  - Pragmatik
- Zeichen $\in$ Alphabet
  - Zeichenfolgen werden durch Verkettung gebildet
  - Regeln für erlaubte/gültige Verkettungen $\rightarrow$ Syntax
- Interpretation der Inhalte
  - Feststellung der Bedeutung der Inhalte $\rightarrow$ Semantik

Definition algorithmischer Sprachen:

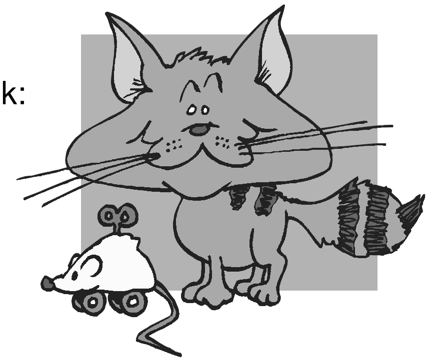
- Syntax
  - regelhafte Beschreibung (Festlegung) der äusseren Form einer Zeichen- oder Wortfolge, die ein legales Konstrukt der algorithmischen Sprache ergibt.
  - Regeln abhängig oder unabhängig von der Interpretation formulierbar.

- Semantik
  - regelhafte Bedeutungszuweisung zu bestimmten Kombinationen von Wörtern.
  - Voraussetzung: „höhere“ Sprache (Metasprache) zur Erläuterung der Bedeutung der Programmiersprachenkonstrukte.

- Pragmatik
  - Zurückführung der in der Sprache als primitiv anzusehenden Operationen auf einzelne Textersetzungsschritte.

Beispiel 1: Katz und Maus

Pragmatik:



Semantik: Maus ist Beutetier von Katzen
Katze jagt Vögel, Mäuse

Syntax: Katze frisst Maus
Maus frisst Katze

Beispiel 2: Multiplikation

a:= 5
b:= 6
c:= 5 • 6 $\longrightarrow$ c = 30

Syntax: lt. Konstruktionsregeln ein gültiger Ausdruck

Semantik: Bedeutung: Multiplikation zweier Zahlen a, b mit bestimmtem Typ (und Wertebereich)

Pragmatik: Abbildung:
Multiplikation $\longrightarrow$ Anzahl von Additions- und Shiftbefehlen

Prinzipielle Fragestellungen zu:
Formale Sprachen und Compiler $\longleftrightarrow$ Spezifikation und Analyse von Computersprachen

? Wie kann/muss eine Grammatik für eine (Programmier-)sprache aussehen?

? Wie kann die Zugehörigkeit eines „Textes“ (Satz, Wort, ...) zu der durch die Grammatik definierten Sprache überprüft werden?

? Wie kann ein Programm analysiert werden ($\rightarrow$ Parsing), um die strukturelle Analyse durch den Compiler zu ermöglichen?

? Wie können Mehrdeutigkeiten bestimmt werden mit dem Ziel einer eindeutigen Analyse (eindeutige Semantik), deren Ergebnis (ausführbarer Code) dann auf einem Rechner ausgeführt werden kann?

- Die Konstruktion von Kunstsprachen und die Analyse ihrer Sprachelemente ist wegen der Sprachform von Algorithmen für die Informatik von zentraler Bedeutung!
- Es geht vornehmlich darum, eine sehr umfangreiche (meist nicht endliche) Menge von Worten in möglichst einfacher Form zu beschreiben und die (in der Wortmenge unterstellte) syntaktische Struktur auszudrücken

Rechner als programmierbare Maschine

- Formulierung maschineller Verfahren zur operativen Nachrichtenverarbeitung (Informationsverarbeitung)
  $\Rightarrow$ Algorithmen
- Eindeutigkeit von

$$N \xrightarrow{\alpha} I$$

Formulierung von „Handlungsanweisungen“ in eindeutiger Art und Weise
$\Rightarrow$ Aufbau von Strukturen, Sprachen

Formale Sprachen

Syntax und Semantik von Programmiersprachen

Natürliche Sprachen und Programmiersprachen haben feste Regeln für den Aufbau von Sätzen in der Sprache (Syntax = Lehre vom Satzbau)

Programmiersprache: Zusammenfassung aller laut der Syntax korrekten Satzkonstruktionen (Programme)

Syntax: Erzeugung gültiger Satzkonstruktionen mittels der in der Grammatik festgelegten Regeln (→ Formale Sprachen)

Semantik (Bedeutungslehre): Beschreibung der Wirkung eines Programms auf den Rechner, d.h. welche Funktion durch das Programm berechnet wird.

Im Folgenden beschäftigen wir uns mit syntaktischen Strukturen, die Semantik bleibt unberücksichtigt.

Grammatiken und (Formale) Sprachen:

➘ Zeichen, Zeichenfolgen, Sprachen

- Zeichenmengen
  S: endliche Menge von Symbolen (Alphabet)
  $S \neq \{\ \}$, $card(S) = |S| = n < \infty$

- Worte (Symbol-, Zeichenketten, strings)
  mithilfe der Elemente $s_i \in S$ werden Zeichenketten „über dem Alphabet S“ gebildet:

  $S^1 = S$
  $S^2 = S \times S$
  $\vdots$
  $S^n = S \times S \times \ldots \times S$ (n-fach)

  S^n: Menge aller Zeichenketten der Länge n über S
  S^*: Menge aller Zeichenketten endlicher Länge über S incl. leerem Wort λ (S^* ist freie Sprache über S)

- Sprache über S
  Wort: $w = s_1 s_2 ... s_n$
  $L \in S^*$
  $(\forall w \in L)$: $|w| = n < \infty \neq \{\ \}$, $|\lambda| = 0$, $w \in S^{|w|}$

  S^* enthält alle, durch beliebige Anordnung von $s_i \in S$ konstruierbaren Worte.

$$\downarrow \textbf{ Beschränkung auf „legale" Konstrukte}$$

L = R(S) Regelsprache aus S^*

Grammatik

- Generierung („Produktion") einer Sprache mittels einer Grammatik

- Semi - Thue System

  $G_{ST} = (S_N, S_T, P, w_S)$ mit:
  S_N: Menge der Nichtterminalzeichen
  S_T: Menge der Terminalzeichen
  $S_N \cap S_T = \varnothing$

  $P = \{p_i\}$, $|P| = n < \infty$: Produktionen (Ersetzungsregeln)
  $P \subseteq (S_N \cup S_T)^* \times (S_N \cup S_T)^*$
  $p_i : \alpha \rightarrow \beta$, wenn $(\alpha, \beta) \in P$

  w_S : Startwort, $w_S \in (S_N \cup S_T)^*$, $|w_S| \geq 0$

- Aus einem Wort u kann ein Wort v dadurch abgeleitet werden, indem ein α in u durch β ersetzt wird und damit aus dem Wort u das Wort v entsteht.

➘ Ableitung einer Symbolfolge (Wort)

- Problem: Es liegt eine Symbolfolge w vor; gehört w zur Sprache, ist also $w \in R(S)$

- Strategien:
  Top-Down: ausgehend vom Startsymbol wird mittels der Produktionen (Ersetzungsregeln) abgeleitet, bis die gegebene Folge produziert ist.
  Bottom-Up: umgekehrt

- Beispiel:
  $G = (S_N, S_T, P, w_S)$
  $G = (\{S,A\}, \{0,1\}, \{S \underset{(1)}{\to} 0A11,\ A \underset{(2)}{\to}\ ,\ A \underset{(3)}{\to} 0A,\ A \underset{(4)}{\to} 1A\}, S)$
  Problem: Gehört 0000111 zur Sprache?

  Top-Down: $S \underset{(1)}{\to} 0A11 \underset{(3)}{\to} 00A11 \underset{(3)}{\to} 000A11 \underset{(3)}{\to} 0000A11 \underset{(4)}{\to}$
  $\underset{(4)}{\to} 00001A11 \underset{(2)}{\to} 0000111$

➘ Beispiel: Syntax für einfache deutsche Sätze

- Nicht-Terminalzeichen:
  S_N= {Satz, Subjekt, Prädikat, Objekt, Nomen, Artikel, Substantiv, Verb} ← Variable der Sprache
- Terminalzeichen:
  S_T= {die, Katze, Maus, frisst} ← Symbole, Darstellungen! (schriftlich, bildlich, ...)

- Startsymbol: w_S = Satz

- Ableitungsregeln (Produktionen $P = \{p_i\}$):

Satz	$\to$	Subjekt Prädikat Objekt
Subjekt	$\to$	Nomen
Nomen	$\to$	Artikel Substantiv
Artikel	$\to$	die
Substantiv	$\to$	Katze
Substantiv	$\to$	Maus
Prädikat	$\to$	Verb
Verb	$\to$	frisst
Objekt	$\to$	Nomen

- Beispiel: Syntax für einfache deutsche Sätze (cont'd)
  - Startsymbol: w_S = Satz
  - Ableitungsregeln (Produktionen P = $\{p_i\}$):

Satz	→	Subjekt Prädikat Objekt
Subjekt	→	Nomen
Nomen	→	Artikel Substantiv
Artikel	→	die
Substantiv	→	Katze
Substantiv	→	Maus
Prädikat	→	Verb
Verb	→	frisst
Objekt	→	Nomen

  - die Katze frisst die Maus (✓)
    die Katze jagt die Maus (**x**, es gibt keine Regel: Verb → jagt)
    Katze die Maus frisst (**x**, ...)
    die Maus frisst die Katze (✓, obwohl semantisch unsinnig!)

Programm- & Sprachelemente

<u>Programmieren und Sprachen</u>

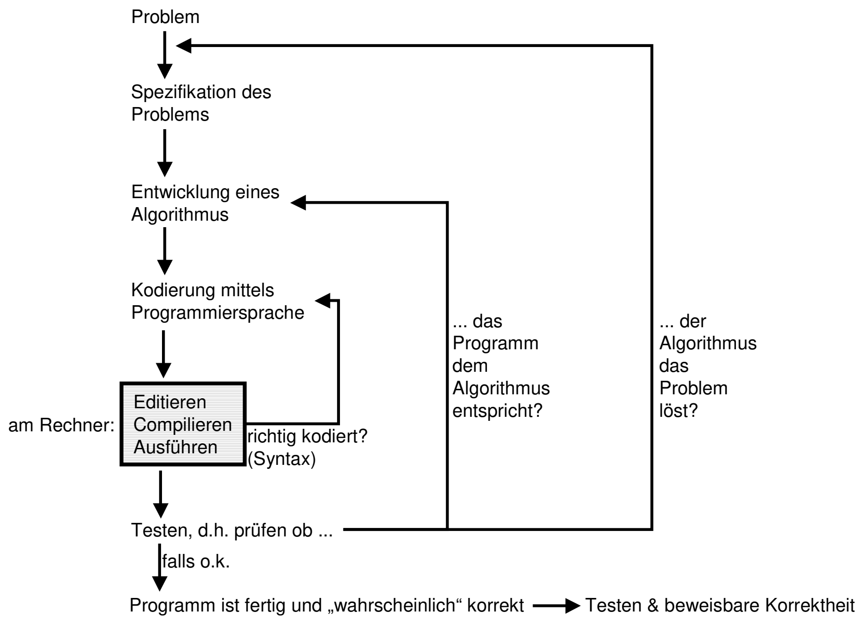
Entwurf und Notation von Programmen

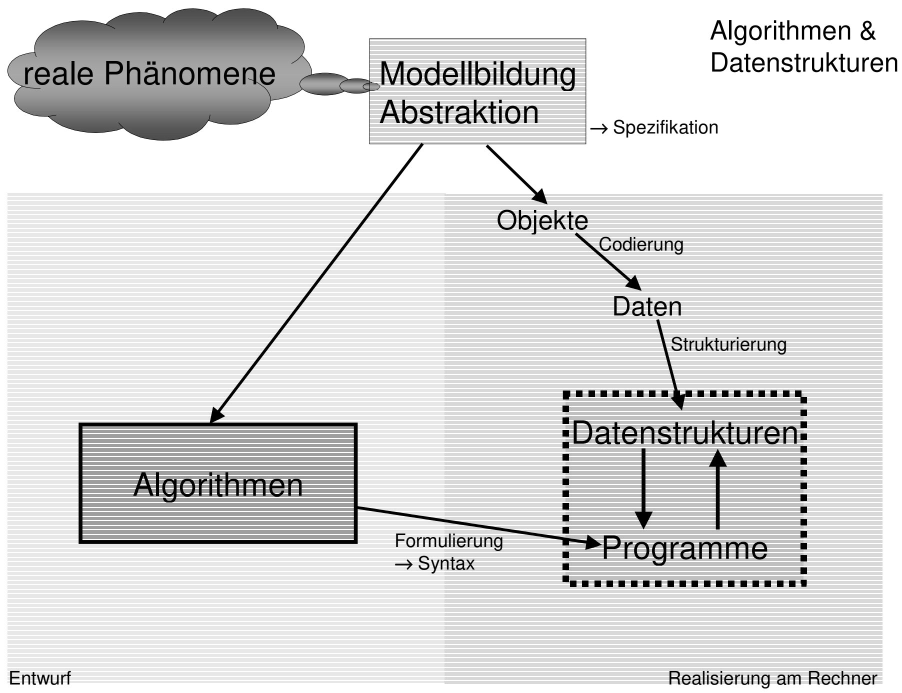
Aufgabe ist es,
einen (abstrakten) Algorithmus in einer maschinell erfassbaren Form zu formulieren.

Hauptaufgabe ist es allerdings,
einen (abstrakten) Algorithmus zu konzipieren und verifizieren;
Programmieren ist somit eine konstruktive Tätigkeit, mehr als nur Codierung.
→ Spezifikation, Entwurf, Modularisierung, ...

Programmiersprache mit ...
Aufbauregeln (Syntax)
Standardinterpretation (Semantik)

Programmierzyklus (aus Puchan et al., 1994)





Programmiersprachen - Programmierstile

- imperativ (algorithmisch, prozedural)
  - PASCAL, MODULA-2, BASIC, ALGOL, COBOL, C, FORTRAN
- funktional
  - LISP
- objektorientiert
  - Smalltalk, C++
- prädikativ (logisch)
  - PROLOG

<u>Erste Schritte in MODULA-2</u>

Grundlegender Aufbau von MODULA-2 Programmen

- Komponenten eines Programms
  - Beschreibung von Aktionen
    - Anweisungen („statements")
    - Kontrollkonstrukte
    - Abstraktion durch Prozeduren
  - Bechreibung von Daten
    - Deklarationen und Definitionen
    - Manipulationen auf einfachen und komplexen Daten
- Programmschema

MODULE Name [Priorität]
IMPORT
Deklarationen von Konstanten, Typen, Variablen, ...

BEGIN
Initialisierungsteil
Verarbeitungsteil   } Rumpf
Ausgabeteil
END Name.

Syntaxdiagramme

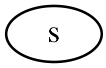
Konstruktionsmittel

→ Graphische Notation zur Festlegung einer kontextfreien Grammatik (Typ 2)

Grundbausteine der Diagramme :

1. Nicht-terminale Zeichen (Variablen) A
   (→ Platzhalter für ein weiteres Diagramm)

2. Terminalzeichen s
   (→ Zeichen der formalen Sprache)

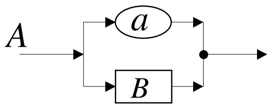


3. Konkatenationen (Wortkonstruktionen)
   (→ Übergang von einem Knoten
   ( ) zum folgenden)

Konstruktionsmittel (EBNF – Syntaxdiagramm) :

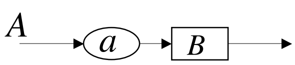
1. **Alternative**

$$A \rightarrow a \mid B.$$ ⟺



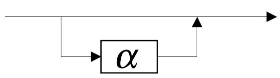
2. **Verkettung**

$$A \rightarrow aB.$$ ⟺

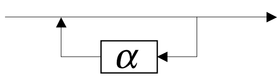


3. **Wiederholungen**

0- oder 1-mal α $A \rightarrow [\alpha]$ ⟺



0-, 1- oder n-mal α $A \rightarrow \{\alpha\}$ ⟺



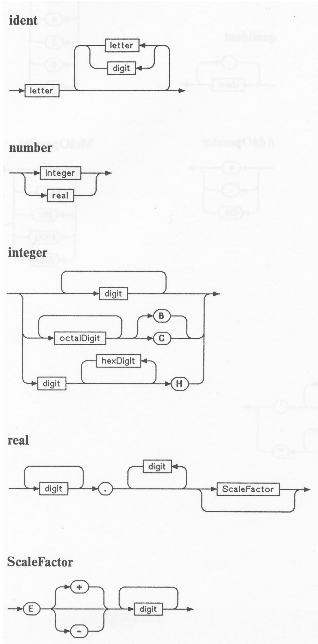
Operatoren und Begrenzer für Modula-2

+	Addition, Vereinigung von Mengen
−	Subtraktion, Differenz von Mengen
*	Multiplikation, Durchschnitt von Mengen
/	Division, symmetrische Mengendifferenz
:=	Zuweisung
&	logisches Und (AND)
~	logische Verneinung (NOT)
=	gleich
# <>	ungleich
<	kleiner als
>	größer als
<=	kleiner als oder gleich
>=	größer als oder gleich
( )	Klammern
[ ]	Index-Klammern
{ }	Mengen-Klammern
(* *)	Kommentar-Klammern
↑	Dereferenzier-Operator
, . ; : .. |	Satzzeichen

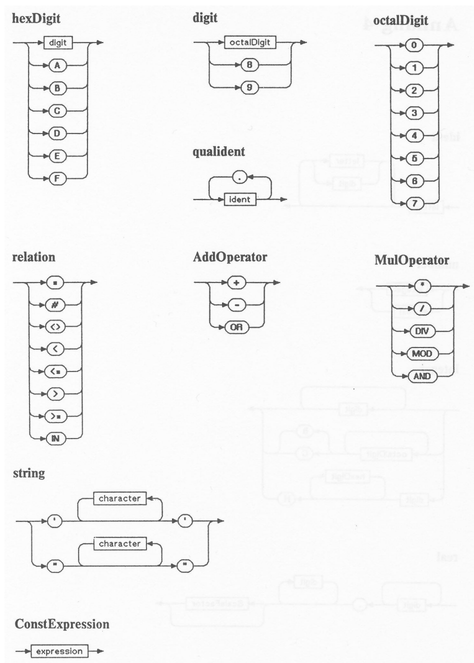
Reservierte Begriffe für Modula-2

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

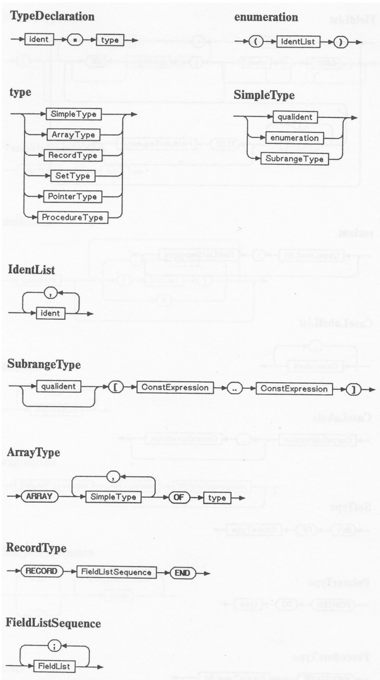
Syntaxdiagramme
für Modula-2 (1)



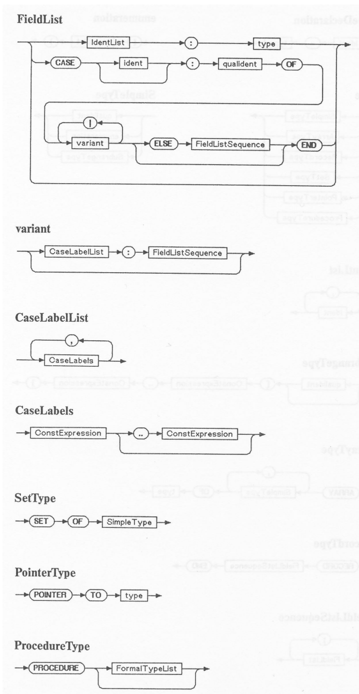
Syntaxdiagramme
für Modula-2 (2)



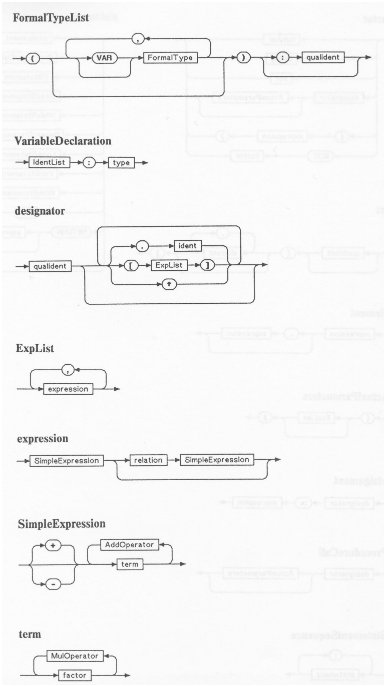
Syntaxdiagramme
für Modula-2 (3)



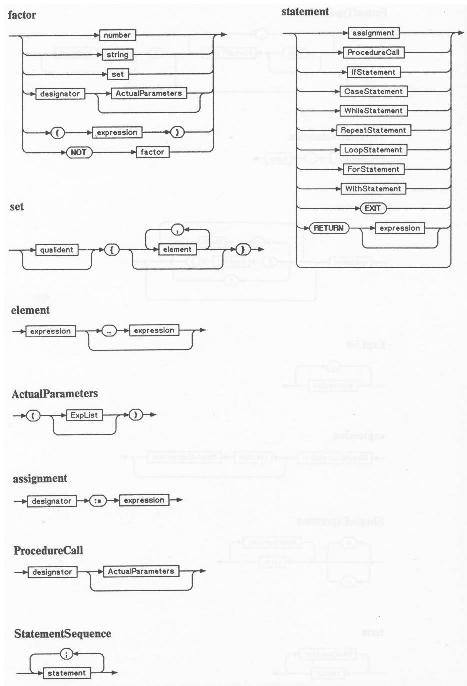
Syntaxdiagramme
für Modula-2 (4)



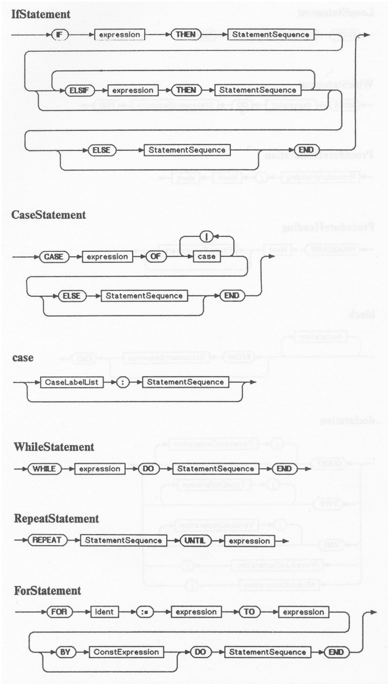
Syntaxdiagramme
für Modula-2 (5)



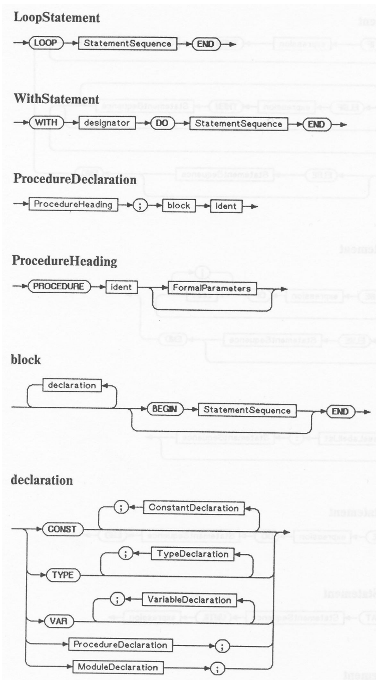
Syntaxdiagramme
für Modula-2 (6)



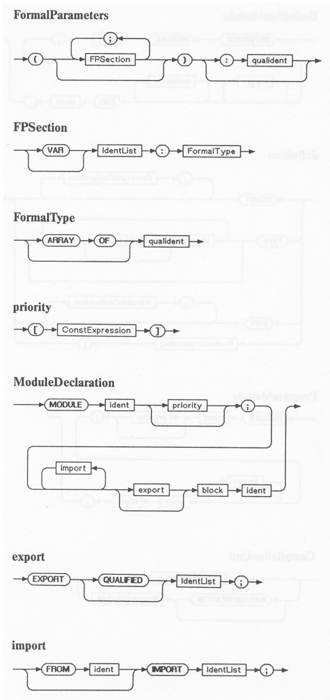
Syntaxdiagramme
für Modula-2 (7)



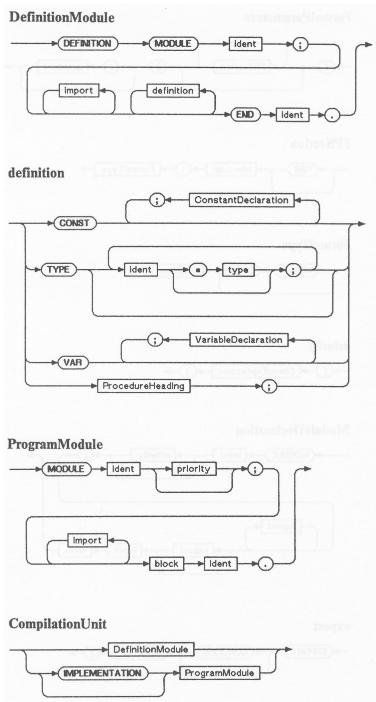
Syntaxdiagramme
für Modula-2 (8)



Syntaxdiagramme
für Modula-2 (9)



Syntaxdiagramme
für Modula-2 (10)



Ein erstes Programm

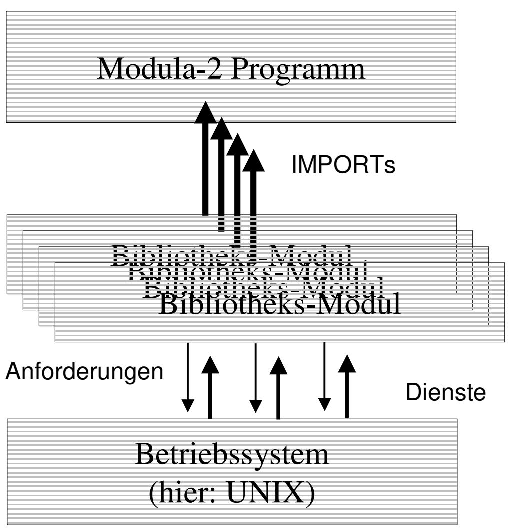
2 Bibliotheken (explizite Nennung)
alternativ: globaler Import: `IMPORT RealInOut`

```
MODULE Mehrwertsteuer;
FROM RealInOut IMPORT ReadReal, WriteReal, WriteFloat;
FROM InOut     IMPORT WriteString, WriteLn;

CONST  MWStSatz = 0.16;
VAR    Preis, MWSt :  REAL;

BEGIN
       WriteString("Geben Sie den Preis ein: ");
       ReadReal(Preis);
       MWSt := Preis • MWStSatz;

       WriteLn;
       WriteString("Die Mehrwertsteuer beträgt (Float): ");
       WriteFloat(MWSt, 8, 2);
       WriteLn;
       WriteString("Die Mehrwertsteuer beträgt (Real): ");
       WriteReal(MWSt, 10);
       WriteLn;
END Mehrwertsteuer.
```

8 Stellen vor, 2 hinter Dez.pkt.

Länge der Ausgabe

Kommunikation: Programm - Betriebssystem



➘ in Modula-2

- Sprachkern ist standardisiert
- Zugriff auf Betriebssystem-Dienste via Bibliotheks-Module
  - ∗ Bibliotheken sind nicht standardisiert
  - ∗ Bibliotheken gehören nicht zum Sprachumfang
- Beurteilung
  - ∗ Nachteil: Unklarheit bzgl. verfügbarer Funktionalität und Struktur der Bibliotheken
  - ∗ Vorteil: Implementierung der Sprache rechner-/systemunabhängig

➘ Ulmer Modula-2 Implementierung

- Liste und Doku unter:
  http://www.mathematik.uni-ulm.de/modula/man/

Gestaltung von Programmen (Grundsätzliches)

➢ Grundsätzliches

- Quelltext eines Programms (= Nachricht)
  ⇒ Arbeitsergebnis, Doku, Unterlage
  (→ Wartung, Programmerweiterung ...)

➢ Grundregeln zu Programmaufbau und -gestaltung

- jedes Programm besteht aus (mehreren) Modulen
- jeder Modul ist ein eigenes Dokument
- Quelltext ist zeilenweise organisiert, mit je 1 Anweisung oder Vereinbarung pro Zeile
- Verwendung von Leerzeilen (Absetzen von Blöcken)
- Einrücken von Anweisungen
- sinnvolle Verwendung von Kommentaren (* ... *)

Gestaltung von Programmen (Grundsätzliches)

➢ Kommentare

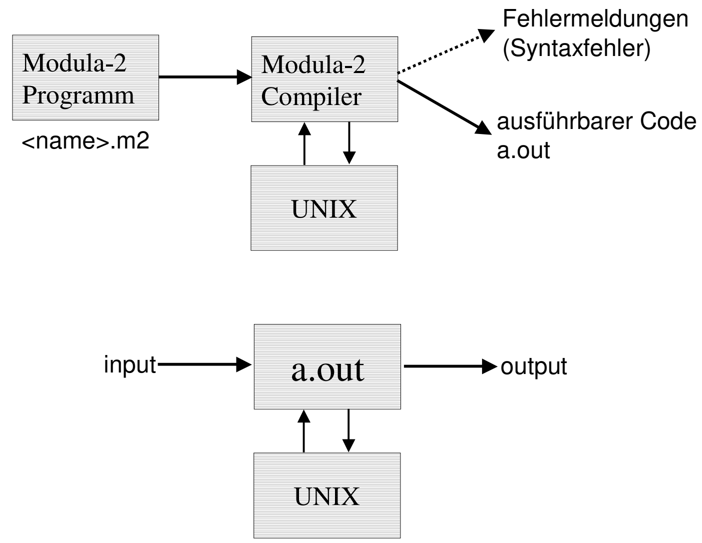
- Syntax: beliebiger, in (* ... *) eingeschlossener Text
- sinnvoll einsetzen:
  - ∗ Modulanfang: Kurzbeschreibung der Aufgabe
  - ∗ Beschreibung der Ein- und Ausgabegrößen
  - ∗ Erläuterung des logischen Programmablaufs (grob strukturierter Algorithmus, Invarianten)
  - ∗ Erklärung von Variablen, Typen, ... (falls nötig)
  - ∗ Kennzeichnung von END
- nicht sinnvoll: Unter-, aber auch Überkommentierung

Ein inakzeptables Programm

```
MODULE Mehrwertsteuer;
FROM RealInOut IMPORT ReadReal, WriteReal, WriteFloat; FROM InOut
IMPORT WriteString, WriteLn;
CONST   M = 0.16;VAR   x,y :    REAL;

BEGIN (*BEGIN Anweisung   *)
WriteString("Geben Sie den Preis ein: ");
ReadReal(x); (* Einlesen von x   *)
y := x • M; WriteLn;WriteString("Die MWSt beträgt (Float): ");
WriteFloat(MWSt, 8, 2); WriteLn;
WriteString("Die Mehrwertsteuer beträgt (Real): ");WriteReal(y, 10);
WriteLn; END Mehrwertsteuer.
```

pfui

Übersetzung von Programmen („Compilation“)



Übersetzung und Ausführung von Programmen

- Übersetzung
  - m2c <Name>.m2
    - Aufruf des Modula-2 Compilers (Programmstart)
    - input = <Name>.m2; output = a.out (ausführbares Programm)
  - m2c -o <Name> <Name>.m2
    - -o ist Compileroption
    - das ausführbare Programm heisst statt a.out jetzt <Name>

- Ausführung
  - Start mittels ...$ a.out
  - bzw. Start mittels ...$ <Name>

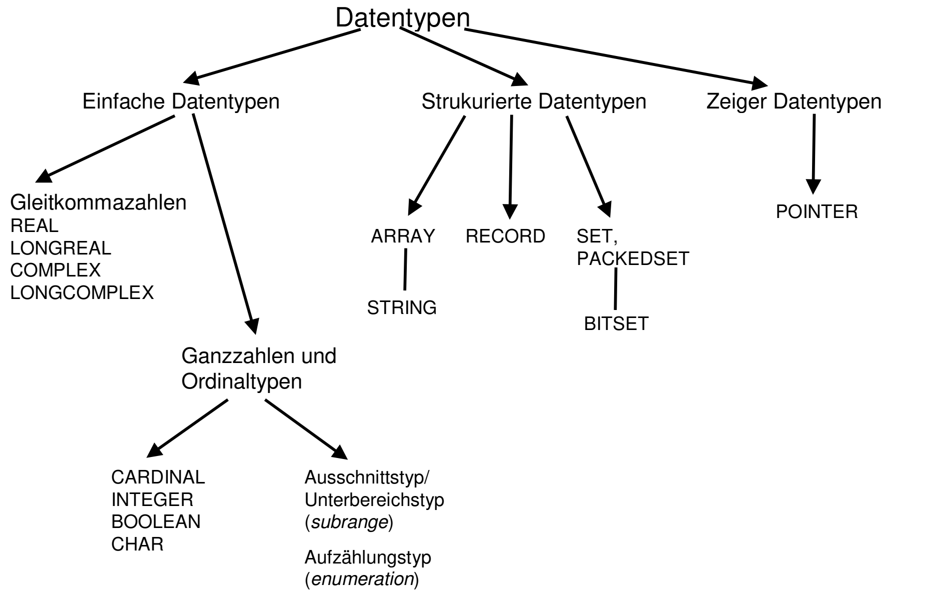
Einfache Datentypen

Eigenschaften eines „Typs“

- Grundsätzliches
  - Der (Daten-)Typ kennzeichnet die aktuelle Wertemenge eines Objekts (Konstante, Variable, Ausdruck, Funktion, ...)
  - Typ-Zuweisung durch Vereinbarung (= Deklaration)
- Eigenschaften
  - Der Datentyp bestimmt die Wertemenge,
    - zu der eine Konstante gehört.
    - deren Werte durch eine Variable oder einen Ausdruck angenommen werden können.
    - deren Werte durch einen Operator oder eine Funktion berechnet werden können.
  - Der Datentyp ist anhand seiner Notation oder Vereinbarung erkennbar.
  - Jeder Operator bzw. jede Funktion
    - erwartet Argumente eines bestimmten Typs
    - liefert Resultate eines bestimmten Typs

Datentyp

Taxonomie der Datentypen



Einfache Standardtypen

➢ Cardinal

- Wertemenge: nicht-negative ganze Zahlen
  {0,1, ... ,MaxCard}; MaxCard = 2^{32} - 1
- Deklaration:
  ```
  VAR    c,anzahl,monat,hausnummer: CARDINAL;
  ```
- Ein-/Ausgabe:
  ```
  FROM InOut IMPORT ReadCard(var),Write(var,breite);
  ```
- Operationen:
  - ∗ Addition +
  - ∗ Subtraktion –
  - ∗ Multiplikation *

  nur beschränkter Wertebereich verfügbar !!
  - ∗ ganzzahlige Division mit Rest DIV, MOD
  - ∗ Vergleiche =,<>,>=,>,<=,<

➢ Integer

- Wertemenge: vorzeichenbehaftete ganze Zahlen
  $\{-2^{N-1}, \ldots, 2^{N-1}-1\}$; N = 32
- Deklaration:

```
VAR  k, wert,index,faktor: INTEGER;
```

- Ein-/Ausgabe:

```
FROM InOut IMPORT ReadInt(var),WriteInt(var,breite);
```

- Operationen:
  - Addition +
  - Subtraktion –
  - Multiplikation *

  (nur beschränkter Wertebereich verfügbar !!)
  - ganzzahlige Division mit Rest DIV, MOD
  - Vergleiche =,<>,>=,>,<=,<
  - Vorzeichenumkehrung –
- Konvertierung ↔ Cardinal

```
VAR   i,int    : INTEGER;
      c,card   : CARDINAL;
i:= INTEGER(card);
c:= CARDINAL(int)
```

➢ Definition von DIV und MOD

- $(x,y) \rightarrow (q,r)$ mit $x,y,q,r \in N$
  wobei $x \text{ DIV } y \rightarrow q$ („Quotient“)
  $x \text{ MOD } y \rightarrow r$ („Rest“)
- $x \text{ DIV } y = \text{trunc}(x/y) =: q$
  wobei $q > 0$, if $\text{sgn}(x)=\text{sgn}(y)$
  $q = 0$, if $\text{abs}(x)<\text{abs}(y)$
  $q < 0$, if $\text{sgn}(x)\neq\text{sgn}(y)$
- $x \text{ MOD } y =: r$
  wobei $r = x - (x\text{DIV}y)\bullet y$, if $r \geq 0$
  $r = x - (x\text{DIV}y)\bullet y + y$, if $r < 0$
- „DIV-MOD“ - Identität
  $(x\text{DIV}y)\bullet y + (x\text{MOD}y) = x$

Logisch:
abgeschnittener Quotient • Nenner + Rest = Zähler