Formale Sprachen

Formale Sprachbeschreibung

Lexikalische Regeln definieren die Wörter, aus denen Programme aufgebaut werden dürfen → Vokabeln

Syntaktische Regeln legen fest, wie aus den Wörtern korrekte Programme gebildet werden → Satzbau

- Die Überprüfung erfolgt mit einem **Parser**.
- Die Syntax wird durch eine **Grammatik** definiert.

Nicht alle Spracheigenschaften können syntaktisch festgelegt werden → **semantische Regeln:**

- Wurde Bezeichner vor seiner Benutzung deklariert?
- Sind keine Typfehler vorhanden?

Grammatik

Sei Σ eine endliche Menge von **Symbolen** (Buchstaben). Dann ist die **Menge alle Wörter** über Σ definiert als

$$\Sigma^* = \{a_1 a_2 \dots a_n \mid n \geq 0,\ a_i \in \Sigma\}.$$

Beispiel: Sei $\Sigma = \{a, b, c\}$, dann ist

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}.$$

- Die Elemente in Σ^* heißen **Wörter**.
- Wir schreiben die Buchstaben der Wörter direkt nebeneinander, nicht durch Kommata getrennt.
- Das leere Wort bezeichnen wir mit ϵ.
- Iterationsoperator $*$ wird auch **Kleene-Star**-Operator genannt (nach S.C. Kleene).

Grammatik (2)

Eine **kontextfreie Grammatik** ist ein 4-Tupel (N, Σ, P, S), wobei gilt:

1. N: endliche Menge **nicht-terminaler Symbole**
2. Σ: endliche Menge **terminaler Symbole**
3. $P \subseteq N \times (\Sigma \cup N)^*$: endliche Menge von **Produktionen**
4. $S \in N$: das **Startsymbol**

Schreibweise:

- schreibe nicht-terminale Symbole groß (A, B, C)
- schreibe terminale Symbole klein (a, b, c)
- Produktionen: $A \to w$ anstelle von (A, w)
- $A \to u \mid v \mid \dots$ ist Kurzschreibweise zu $(A, u), (A, v), \dots$

Grammatik (3)

Sei $G = (N, \Sigma, P, E)$ eine kontextfreie Grammatik mit $N = \{E, I, D\}$, $\Sigma = \{(,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$ und

$$P = \left\{ \begin{array}{rcl} E & \rightarrow & I \mid (E) \mid E + E \mid E - E \mid E * E \mid E/E \\ I & \rightarrow & 0 \mid 1D \mid 2D \mid 3D \mid \ldots \mid 8D \mid 9D \\ D & \rightarrow & 0D \mid 1D \mid 2D \mid 3D \mid \ldots \mid 8D \mid 9D \mid \epsilon \end{array} \right\}$$

Die Sprache $L(G)$ der Grammatik G ist die Menge aller Wörter $w \in \Sigma^*$, die aus dem Startsymbol ableitbar sind. Hier: arithmetische Ausdrücke.

Beispiel: Ableitung von $2 * (8 + 7)$

$$\begin{array}{rcl} E & \rightarrow & E * \underline{E} \rightarrow E * (\underline{E}) \rightarrow \underline{E} * (E + E) \rightarrow \underline{I} * (E + E) \\ & \rightarrow & 2\underline{D} * (E + E) \rightarrow 2 * (\underline{E} + E) \rightarrow 2 * (\underline{I} + E) \\ & \rightarrow & 2 * (8\underline{D} + E) \rightarrow 2 * (8 + \underline{E}) \rightarrow \ldots \rightarrow 2 * (8 + 7) \end{array}$$

Grammatik (4)

Anmerkungen:

- Für eine Produktion $A \rightarrow u$ ist A die **linke Seite** und u die **rechte Seite der Produktion**.
- kontextfreie Grammatik: linke Seite der Produktion besteht aus genau einem nicht-terminalen Symbol $\rightarrow$ die Ersetzung der linken Seite erfolgt unabhängig von der Umgebung.
- Es gibt auch **kontext-sensitive Grammatiken**.

Grammatik (5)

Beispiel: Kontext-sensitive Grammatik

Sei $G = (N, \Sigma, P, S)$ mit $N = \{A, B, S\}$, $\Sigma = \{a, b\}$ und

$$P = \left\{ \begin{array}{rcl} S & \rightarrow & ABS \mid \epsilon \\ A & \rightarrow & a \\ B & \rightarrow & b \\ AB & \rightarrow & BA \\ BA & \rightarrow & AB \end{array} \right\}$$

Dann ist $L(G) = \{w \in \{a, b\}^* \mid \#a = \#b\}$.

Backus Naur Form

Die BNF-Notation wurde von John Backus und Peter Naur für die Definition von Algol 60 entwickelt.

BNF-Notation wurde später zur **Extended BNF** (EBNF) Notation erweitert.

Programmiersprachen: oft in EBNF-Notation beschrieben

In der EBNF-Notation werden Sprachen über kontextfreie Grammatiken (N, Σ, P, S) definiert, wobei die Produktionsmenge durch Regeln der Art $A ::= R$ beschrieben ist.

Extended Backus Naur Form

Form einer Regel:

- linke Seite: nicht-terminales Symbol A
- rechte Seite: ein **EBNF-Ausdruck** R über $\Sigma \cup N$ mit:
  * spitze Klammern < und >: nicht-terminale Symbole
  * der senkrechte Strich | kennzeichnet Alternativen
  * geschweifte Klammern { und }: Kleene-Star-Operator
  * eckige Klammern [und] bedeuten entweder null oder einmal
  * runde Klammern (und) dienen zur Klammerung

Extended Backus Naur Form (2)

Beispiel: ganzzahlige arithmetische Ausdrücke

```
<exp> ::= <int> | <exp> (+|-|*|/) <exp>
<int> ::= <dig> | <dig><int>
<dig> ::= 0|1|2|3|4|5|6|7|8|9
```

Anmerkungen:

- Um die in EBNF-Ausdrücken verwendeten Metasymbole von terminalen Symbolen zu unterscheiden, werden sie im Konfliktfall fett geschrieben.
- Gelegentlich erfolgt die Kennzeichnung der terminalen Symbole durch Anführungszeichen.

Extended Backus Naur Form (3)

Eine EBNF-Beschreibung für Bezeichner in C:

```
<name> ::= <char> {<dig> | <char>}
<char> ::= A|B|C|...|Z|a|b|c|...|z|_
<dig>  ::= 0|1|2|3|4|5|6|7|8|9
```

und dasselbe rekursiv:

```
<name> ::= <char> | <name><char> | <name><dig>
<char> ::= A|B|C|...|Z|a|b|c|...|z|_
<dig>  ::= 0|1|2|3|4|5|6|7|8|9
```

Alle mittels EBNF definierbare Sprachen sind kontextfreie Sprachen.

Syntaxdiagramme

Die Syntax einer Programmiersprache lässt sich grafisch mit Syntaxdiagrammen darstellen:

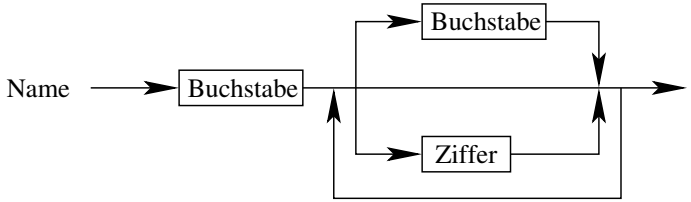
- nicht-terminale Symbole in rechteckigen Kästen
- terminale Wörter in runden Kästen
- Pfeile kennzeichnen möglichen weiteren Verlauf

Syntaktisch korrekter Text: durchlaufe das Syntaxdiagramm vom Eingangspfeil zum Ausgangspfeil und notiere dabei alle Wörter in runden Kästen, auf die man trifft.

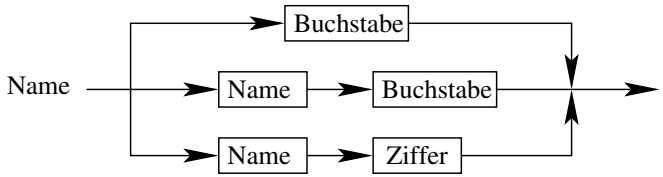
Syntaxdiagramme können auch **rekursiv** sein, d.h. im Diagramm mit Bezeichner X (bzw. in einem von X indirekt angegebenen Diagramm) kommt X selbst wieder vor.

Syntaxdiagramme (2)

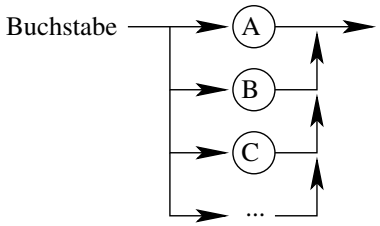
Beispiel: Syntaxdiagramm für Bezeichner in C

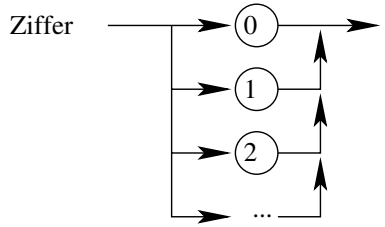


und dasselbe rekursiv



Syntaxdiagramme (3)





Durch Syntaxdiagramme definierte Sprachen sind kontextfreie Sprachen. Syntaxdiagramme haben dieselbe Beschreibungskraft wie EBNF-Ausdrücke.

ergänzende Literatur:
J.E. Hopcroft, J.D. Ullman: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley.

Programmiersprachen

Programmiersprachen

Maschinenbefehle: elementare Operationen, die der Prozessor des Rechners unmittelbar ausführen kann.

- Daten aus dem Speicher lesen
- elementare arithmetische Operationen
- Daten in den Speicher schreiben
- Berechnung an anderer Stelle fortsetzen (Sprünge)

moderne Programmiersprachen: orientieren sich am zu lösenden Problem.

- abstrakte Formulierung des Lösungsweges
- Eigenheiten der Hardware werden nicht berücksichtigt

Programmiersprachen (2)

Konzepte:

- Werte und Typen
- Variablen und Befehle
- Bindungen
- Abstraktion
- Kapselung
- Typsysteme
- Ablaufsteuerung mit Ausnahmen
- Nebenläufigkeit

Programmiersprachen (3)

Werte und Typen:

- Daten sind genau so wichtig wie Programme: Telefonbuch, Wörterbuch, Satellitendaten, usw.
- **Wert:** beliebige Größe, die während einer Berechnung existiert
- **Typen:** Mengen von Werten, die in der Programmiersprache als Daten manipuliert werden können
  * einfache oder zusammengesetzte Typen
  * rekursive Typen (enthalten Werte desselben Typs)
- **Typsysteme:** schränken die Operationen ein, statische oder dynamische Typisierung
- **Ausdrücke:** berechne aus alten Werten neue Werte

Programmiersprachen (4)

Variablen und Befehle:

- **Variable:** Objekt, das einen Wert enthält
  * modelliert Objekte der realen Welt
  * wird durch Zuweisung überschrieben
- **Speicher:** Zusammenfassung von Zellen, besitzt einen gegenwärtigen Inhalt (zeitabhängig)
- **Lebensdauer:** lokale, globale, Heap- und persistente Variablen
- **Befehle:** Zuweisungen, Prozeduraufrufe, bedingte Befehle, sequentielle oder nebenläufige Blöcke, Iteration (Wiederholung, Schleife)
- **Seiteneffekte:** Auswerten eines Ausdrucks soll einen Wert liefern, aber sonst keinen weiteren Effekt haben

Programmiersprachen (5)

Bindungen:

- Bezeichner an Konstanten, Variablen, Prozeduren und Typen binden
- unterscheide Programmiersprachen: welche Arten von Größen können an Bezeichner gebunden werden?
- **Reichweite:** Teil des Programmtextes, für den die Vereinbarung gilt → Blockstruktur
  * **statische Reichweite:** zur Übersetzungszeit bekannt
  * **dynamische Reichweite:** erst zur Ausführungszeit bestimmt
- **Sichtbarkeit:** Bezeichner in verschiedenen Blöcken vereinbaren → in der Regel wird in jedem Block eine andere Größe bezeichnet

Programmiersprachen (6)

Abstraktion:

- Konstrukte der Programmiersprache sind Abstraktion von Maschinenbefehlen
- unterscheide zwei Fragestellungen:
  * **Was** tut ein Programmstück? → Prozedur aufrufen
  * **Wie** ist es implementiert? → Prozedur schreiben
- Hierarchiestufen:
  * baue einfache Prozeduren aus Befehlen auf
  * baue komplexe Prozeduren aus einfacheren auf
  * baue sehr komplexe Prozeduren aus komplexen auf ...
- Abstraktionen: Prozeduren und Funktionen

Programmiersprachen (7)

Kapselung: → Programmieren im Großen

- Setze große Programme aus Modulen zusammen!
  analog: Fernseher/Computer besteht aus Baugruppen
- **Modul:**
  * benannte Programmeinheit, die (mehr oder weniger) unabhängig vom Rest implementiert werden kann.
    Beispiele: Liste, Wörterbuch, ...
  * hat klar umrissenen Zweck und klare Schnittstelle zu anderen Modulen

  ⇒ **Wiederverwendbarkeit**
- nur wenige der Modul-Komponenten sind nach außen sichtbar → Abstraktion: Was tut das Modul, nicht wie!

Programmiersprachen (8)

in der Vorlesung *Programmentwicklung:*

- **Typsysteme:**
  * monomorph: jede Konstante, Variable, Funktion usw. muss von einem bestimmten Typ vereinbart werden → nicht ausreichend
  * deshalb: Überladen, Polymorphie, Vererbung
- **Ablaufsteuerung mit Ausnahmen**
- **Nebenläufigkeit:** bspw. bei GUI-Programmierung

Programmiersprachen (9)

Paradigmen:

- imperatives Programmieren
- objekt-orientiertes Programmieren
- funktionales Programmieren
- logisches Programmieren

Programmiersprachen (10)

imperatives Programmieren:

- beruht auf Befehlen, die Variablen im Speicher überschreiben (lat. *imperare*: befehlen)
- seit den 50er Jahren: Variablen und Zuweisungen sind nützliche Abstraktion von Lade- und Speicherbefehlen in Maschinensprachen → Basic, Cobol, Fortran
- heute extrem weit verbeitet (OOP ist Spezialfall der imperativen Programmierung) → C/C++, Delphi, Java
- natürliche Art der Modellierung von Prozessen der realen Welt: Zustand von Objekten der realen Welt ändert sich mit der Zeit → beschreiben durch Variablen

Programmiersprachen (11)

objekt-orientiertes Programmieren:

- Ein Modul, das globale Variablen benutzt, kann nicht unabhängig von anderen Modulen, die die Variable auch benutzen, entwickelt und verstanden werden!
- Schnittstellen: jede globale Variable wird in einem Modul gekapselt und mit einem Satz von Prozeduren versehen, die als einzige direkten Zugriff auf die Variable haben
- heutzutage nennt man solche Schnittstellen **Klassen**
- Klassen geben Programmen eine modulare Struktur
- man kann auch in C objekt-orientiert programmieren, aber es wird nicht erzwungen oder unterstützt

Programmiersprachen (12)

funktionales Programmieren:

- Programm als Implementierung einer Abbildung: Eingabewerte auf Ausgabewerte abbilden
- Konzepte:
  * **Mustervergleich:** ein Funktionsname für verschiedene Parametertypen
  * **Funktionen höherer Ordnung:** Parameter oder Ergebnis sind Funktionen
  * **verzögerte Auswertung:** das Argument einer Funktion wird erst bei der ersten Benutzung ausgewertet, nicht beim Aufruf der Funktion

Programmiersprachen (13)

logisches Programmieren:

- Programm berechnet Relation → allgemeiner als Abbildung, höhere Stufe als funktionales Programmieren
- Sei $R : S \times T$ eine zweistellige Relation:
  * gegeben a, b: bestimme, ob $R(a, b)$ gilt
  * gegeben a: finde alle $t \in T$, so dass $R(a, t)$ gilt
  * gegeben b: finde alle $s \in S$, so dass $R(s, b)$ gilt
  * finde alle $s \in S$ und $t \in T$, so dass $R(s, t)$ gilt
- volle Mächtigkeit der Prädikatenlogik kann nicht ausgeschöpft werden → beschränken auf Hornklauseln

Programmiersprachen (14)

ergänzende Literatur:

- Watt: Programmiersprachen. Carl Hanser Verlag
- Ghezzi, Jazayeri: Konzepte der Programmiersprachen. Oldenbourg Verlag
- Louden: Programming Languages. PWS Publishing

Programmiersprachen (15)

Programme: Text nach genau festgelegten Regeln, durch **Grammatik der Programmiersprache** definiert.

Grammatik-Regeln sind exakt einzuhalten, sonst wird das Programm als Ganzes nicht verstanden!

Beispiel:

```
#include <stdio.h>
main() {
    printf("Hello, world!\n")
}
```

übersetzen mit GNU C-Compiler liefert:

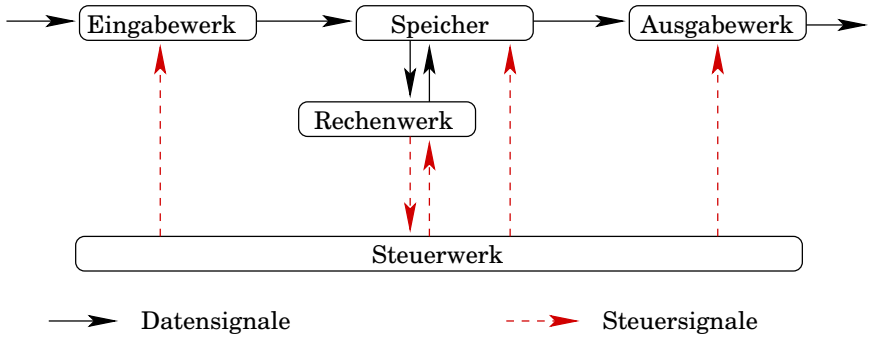
```
hw.c: In Funktion >>main<<:
hw.c:4: error: Fehler beim Parsen before '}' token
```


Rechnerarchitektur

Von-Neumann-Rechner

John von Neumann: 1903 - 1957, Mathematiker, schuf die wesentlichen theoretischen Grundlagen für programmgesteuerte Automaten → Basis heutiger Computer

Der Rechner besteht aus fünf **Funktionseinheiten**:



Von-Neumann-Rechner (2)

Ohne Programm ist die Maschine nicht arbeitsfähig: Zur Lösung eines Problems muss von außen ein Programm eingegeben und im Speicher abgelegt werden.

Programme, Daten, Zwischen- und Endergebnisse werden in demselben Speicher abgelegt.

der Speicher:

- unterteilt in gleichgroße Zellen
- Zellen sind fortlaufend nummeriert
- über die Nummer (Adresse) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden

Von-Neumann-Rechner (3)

Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt.

- nächster Befehl: Steuerwerk → Befehlsadresse + 1
- **Sprungbefehle:** Abweichen von der Bearbeitung der Befehle in der gespeicherten Reihenfolge.

unterschiedliche Befehlsarten:

- Arithmetik: Addieren, Multiplizieren, Konstanten laden
- Logik: Vergleiche, logisches NICHT, UND, ODER
- Transport: Speicher zum Rechenwerk, Ein-/Ausgabe
- bedingte Sprünge
- sonstiges: Schieben, Unterbrechen, Warten

Von-Neumann-Rechner (4)

Das Rechenwerk besteht aus zwei Komponenten:

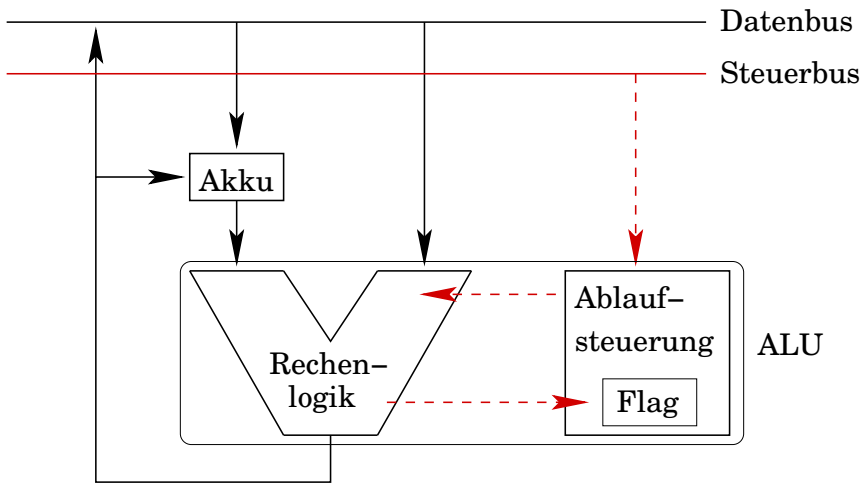
- **Akkumulator:** einfaches Register, ist als Operand an jeder Berechnung beteiligt und nimmt das Ergebnis auf.
- **ALU:** Arithmetic Logical Unit

Auch die **ALU** besteht aus zwei Komponenten:

- **Rechenlogik:** realisiert mathematische Funktionen
- **Ablaufsteuerung:**
  * erzeugt die sogenannten Flags
  * wählt die gewünschte Funktion aus
  * stellt komplexe Funktionalität bereit (Multiplikation durch Addition mittels Barrel-Shifter-Verfahren)

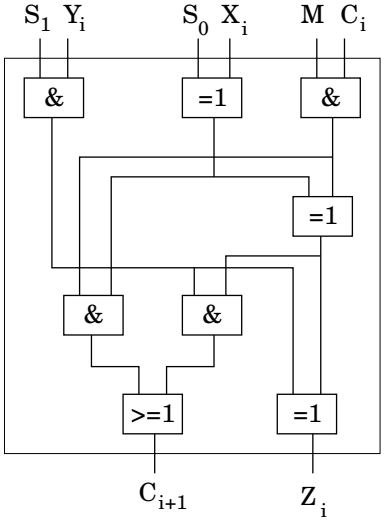
Von-Neumann-Rechner (5)

das Rechenwerk:



Von-Neumann-Rechner (6)

1-Bit ALU:



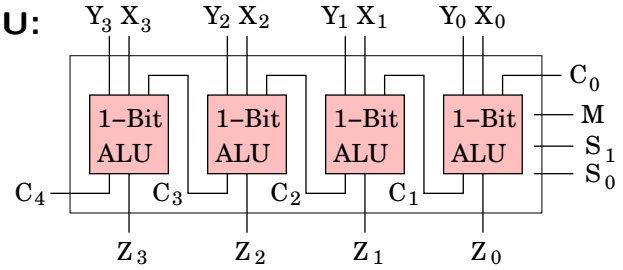
S_0, S_1, M: Steuerleitungen
$M = 0$: logischer Modus
$M = 1$: arithmetischer Modus

X_i, Y_i: Eingabewerte
C_i: Carry-Eingang

Z_i: Ergebnis
C_{i+1}: Carry-Bit

Von-Neumann-Rechner (7)

4-Bit ALU:



Eingabe-/Ausgabewerk: Logik, die den eigentlichen von-Neumann-Computer mit der *Außenwelt* verbindet:

- Grafikkarte
- PCI-Bus für Einsteckkarten
- SCSI-Interface zum Anschluß von Peripheriegeräten
- parallele (Drucker) und serielle (Maus) Schnittstellen

Von-Neumann-Rechner (8)

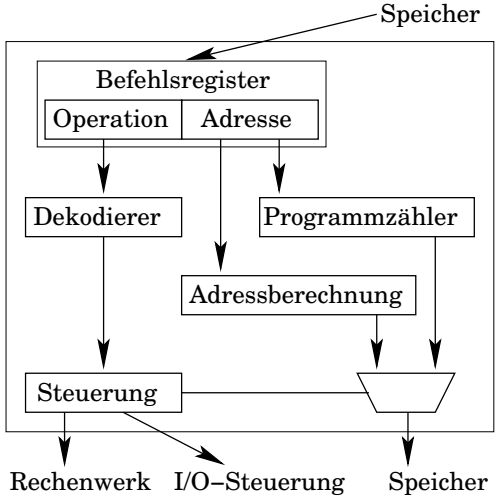
Alle Daten (Befehle, Adressen usw.) werden binär codiert. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (Decodierung).

das Steuerwerk: Abarbeiten eines Befehls in drei Phasen

- **Laden:** holen des Befehls aus dem Speicher, dekodierern des Befehlscodes, ggf. berechnen einer Adresse
- **Verarbeiten:** ausführen der Anweisung, z.B. Addition
- **Speichern:** Ergebnis vom Akkumulator/Register in eine Speicherzelle schreiben

Von-Neumann-Rechner (9)

das Steuerwerk:



Steuerung: als endlicher Automat oder als Mikroprogramm realisiert

Von-Neumann-Rechner (10)

ergänzende Literatur:

- Herrmann: Rechnerarchitektur. Vieweg Verlag
- Märtin: Einführung in die Rechnerarchitektur. Carl Hanser Verlag

Modellierung/Spezifikation

Modellbildung

- Abstraktion (weglassen) von unnötigen Details
- Wahl der geeigneten Darstellung

Realität und Repräsentation:

Realität	Repräsentation
Gegenstände	Zahlen, Wörter
Eigenschaften	Mengen
Zusammenhänge	Relationen

Beispiel: Getränkeautomat

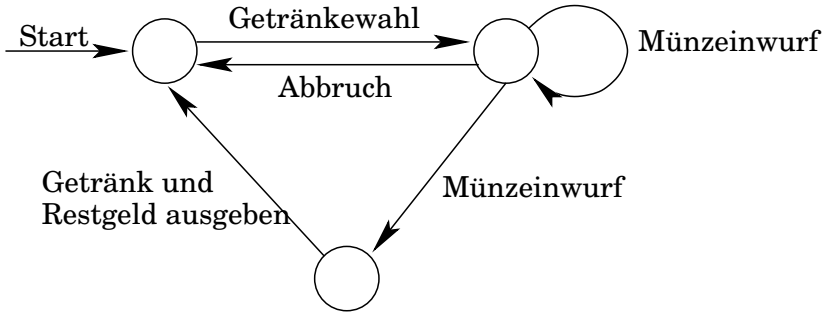
Gegenstände	Cola, Wasser, Bier, 20
Eigenschaften	wählbare Getränke: $\{g_1, \ldots, g_n\}$
Zusammenhänge	Preise: $\{(g_1, p_1), \ldots, (g_n, p_n)\}$

Modellbildung (2)

Es wird abstrahiert von: Farbe und Größe des Automaten, Verfügbarkeit der Getränke usw.

reale Welt $\longrightarrow$ Modellbildung $\longrightarrow$ Repräsentation

Repräsentiert werden müssen Zustände und Operationen.



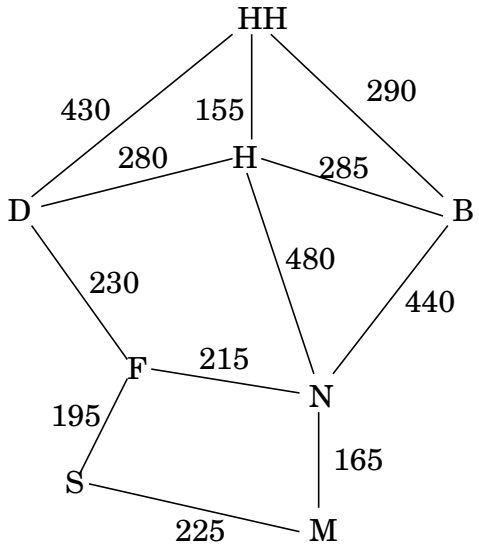
Modellbildung (3)

Beispiel: Wegesuche im Autobahnnetz

Realität	Repräsentation
Entfernung x zwischen A und B	Tripel (A, B, x)
x km Stau zwischen A und B	Tripel (A, B, x)
Baustellen	?
Höchstgeschwindigkeit	?
Wetter	?
Fahrbahnzustand	?
Kürzester Weg	?

Modellbildung (4)

Darstellung eines Verbindungsnetzes:



Gesucht:
die kürzeste Verbindung von Stuttgart nach Berlin.

Modellbildung (5)

Repräsentation im Rechner:

Ort 1	Ort 2	Entfernung
HH	D	430
HH	H	155
HH	B	290
D	HH	430
D	H	280
D	F	230
...	...	...

Ort 1	Ort 2	Entfernung
H	D	280
H	HH	155
H	B	285
H	N	480
F	D	230
F	N	215
...	...	...

Gesucht: Kürzeste Verbindung von Berlin nach Stuttgart.

Spezifikation

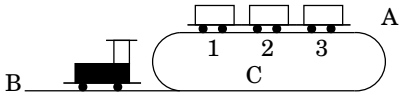
Um einen Algorithmus zu entwickeln, muss das zu lösende Problem genau beschrieben sein.

Anforderungen an eine Spezifikation:

- **vollständig**: Es müssen alle Anforderungen und alle relevanten Rahmenbedingungen angegeben werden.
- **detailliert**: Welche Hilfsmittel bzw. Operationen sind zur Lösung zugelassen?
- **unzweideutig**: Klare Kriterien geben an, wann eine vorgeschlagene Lösung akzeptabel ist.

Spezifikation (2)

Beispiel: Eine Lok soll die Wagen 1, 2, 3 vom Abschnitt A in der Reihenfolge 3, 1, 2 auf Gleisstück C abstellen.



Vollständigkeit:

- Wieviele Wagen kann die Lokomotive auf einmal ziehen?
- Wieviele Wagen passen auf Gleisstück B?

Detailliertheit:

- Was kann die Lokomotive? (fahren, koppeln, ...)

Unzweideutigkeit:

- Darf die Lok am Ende zwischen den Wagen stehen?

Spezifikation (3)

Warum Spezifikation?

- Problemstellung präzisieren
- Entwicklung unterstützen
- Überprüfbarkeit verbessern
- Wiederverwendbarkeit erhöhen

Ziel: Erst denken, dann den Algorithmus entwerfen!

Spezifikation (4)

Bestandteile der funktionalen Spezifikation:

1. Ein Algorithmus berechnet eine Funktion:
   (a) festlegen der gültigen Eingaben (Definitionsbereich)
   (b) festlegen der gültigen Ausgaben (Wertebereich)

2. Funktionaler Zusammenhang zwischen Ein-/Ausgabe:
   (a) Welche Eigenschaften hat die Ausgabe?
   (b) Wie sehen die Beziehungen der Ausgabe zur Eingabe aus?

3. Festlegen der erlaubten Operationen.

Spezifikation: Beispiele

Euklidischer Algorithmus

gegeben: $n, m \in \mathbb{N}$
gesucht: $g \in \mathbb{N}$
funktionaler Zusammenhang: $g = ggT(n, m)$

Jüngste Person

gegeben: $(a_1, \ldots, a_n) \in \mathbb{N}^*, n > 0$
gesucht: $p \in \{1, \ldots, n\}$
funktionaler Zusammenhang:

- $\forall i \in \{1, \ldots, n\}$ gilt: $a_p \leq a_i$

oder alternativ:

- $\forall j \in \{1, \ldots, n\}$ gilt: $a_j \neq a_p \Rightarrow a_j > a_p$.

Verifikation

Ziel: Beweise, dass der Algorithmus korrekt ist!

Wechselspiel zwischen:

- statische Aussagen über den Algorithmus ohne ihn auszuführen → nicht vollständig möglich: zu komplex und umfangreich
- dynamisches Testen des Algorithmus → zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit

Programmverifikation: zeige, dass der Algorithmus die funktionale Spezifikation erfüllt

- der Algorithmus liefert zu jeder Eingabe eine Ausgabe
- die Ausgabe ist die gewünschte Ausgabe

Verifikation (2)

Notation:

- $\{P\}$ Schritt $\{Q\}$
  - P: Vorbedingung
  - Q: Nachbedingung
  - falls vor Ausführung P gilt, dann gilt nachher Q
- $\{P_0\}$ Schritt 1 $\{P_1\}$ Schritt 2 $\{P_2\}$ ... Schritt n $\{P_n\}$
  - falls vor Ausführung des Algorithmus P_0 gilt, dann gilt nachher P_n

Zuweisung: $\{P(t)\} \quad v := t \quad \{P(t) \wedge P(v)\}$
Was vorher für t gilt, gilt nachher für v.

$$\begin{array}{lll} \{t > 0\} & v := t & \{t > 0 \ \wedge \ v > 0\} \\ \{x \in \mathbb{N}\} & y := x + 5 & \{x, y \in \mathbb{N} \ \wedge \ y \geq 5\} \end{array}$$

Verifikation (3)

Wiederholung: $\{P\}$ solange B wiederhole Schritt $\{P \wedge \neg B\}$
P gilt vor und nach jeder Ausführung von Schritt.

$x := y$
$k := 0$
$\{y = k \cdot a + x\}$
solange $x \geq 0$ wiederhole
    $x := x - a$
    $k := k + 1$
$\{y = k \cdot a + x \ \wedge \ x < 0\}$