Algorithmen & Datenstrukturen I

WS 2001/02

2. Algorithmus, Berechenbarkeit und Programmiersprachen

Inhalt

1. **Algorithmen**
2. **Berechenbarkeit**
   - Berechenbarkeit und Turing-Maschinen
   - Church'sche These und Halteproblem
   - Semi-Thue-Systeme
3. **Programmiersprachen**
   - Algorithmus und Programm(iersprache)
   - Semiotik: Syntax, Semantik und Pragmatik
   - Grammatiken und Syntaktische Analyse

2.1 Algorithmen: Ausgangspunkt

1. Wie kann man eine Handlungsvorschrift (Rechenvorschrift) so formulieren, dass deren Befolgen
   - bei sinnvollen Ausgangsdaten (Eingabe)
   - zum angestrebten Ziel (Ausgabe) führt,

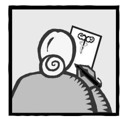
   ohne dass dazu ein Verständnis des Problems notwendig ist (mechanisches Lösen)?
2. Können alle Probleme so formuliert werden?

- Eine Vorschrift wie in 1 heißt Algorithmus (Präzisierung folgt)!
- Ein Problem (eine mathematische Funktion) für das ein Algorithmus existiert heißt berechenbar.

Algorithmen: Beispiele

1. Zerlegung handwerklicher Arbeiten in einzelne Arbeitsschritte (> Ausführbarkeit durch Maschinen)
2. Kochrezepte
3. Verfahren zum schriftlichen Multiplizieren
4. Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen (Euklid, 300 v. Chr)






Präzisierung Algorithmus

Ein Algorithmus ist ein eindeutig beschriebenes Verfahren, das bestimmt, mit welchen Operationen welche Objekte (Daten) bearbeitet werden sollen.

> In der Literatur viele (ähnliche) Definitionen

Wichtig

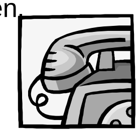
1. Sprache zur Abfassung des Algorithmus muss vereinbart sein (Umgangssprache, ..., Programmiersprache)
2. Objekte müssen klar umrissen sein
3. Operationen (Aktionen) müssen eindeutig und ausführbar sein (für Adressaten)
4. Reihenfolge der Operationen muss feststehen

Beispiel für einen Algorithmus (1/6)

„Suchen des Namen „Kundera“ im Telefonbuch“

1. Telefonbuch in der Mitte des Buches aufschlagen.
   Ein Name x wird gelesen.
2. Wenn x alphabetisch vor „Kundera“ liegt, dann wird der hinter x liegende Teil des Telefonbuches in der Mitte aufgeschlagen.

   Wenn x aber hinter diesem Namen liegt, dann wird der vordere Teil des Telefonbuchs in der Mitte aufgeschlagen.



3. Schritt 2 wird solange wiederholt, bis
   a. „Kundera“ gelesen wird
   b. zwei aufgeschlagene Namen hintereinander liegen
      > „Kundera“ steht nicht im Telefonbuch

Beispiel für einen Algorithmus (2/6)

Sprache: Umgangssprache

> Genauigkeit ist unter Umständen aufwendig

Objekte: Seiten, Namen

Operationen: aufschlagen, lesen, alphabetisches Vergleichen

Reihenfolge: Nummerierung, „Wenn..., dann“, „...solange wiederholt, bis... „

Beispiel für einen Algorithmus (3/6) Einschub Effizienz

„Suchen des Namen „Kundera“ im Telefonbuch“
Variante 2:

1. Schlage die erste Seite des Buches auf
2. Lese beginnend mit dem ersten Namen einen Namen nach dem anderen, bis
   a. „Kundera“ gelesen wird
   b. das Ende des Buches erreicht ist („Kundera“ steht nicht im Telefonbuch)

Führt auch zum Ziel, ist aber weniger *effizient*
effiziente Lösung = Lösung mit minimalem Ressourcenverbrauch (Rechenzeit, Speicherbedarf etc.) > Abwägung !

Beispiel für einen Algorithmus (4/6)

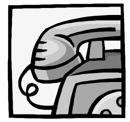
Verbesserung (bzgl. Genauigkeit) von Variante 1

Telefonbuch = alphabetisch geordnete Liste von Namen

1. X0 := erster Name, X1 := letzter Name
2. IF es gibt einen Namen zwischen X0, X1

   THEN lies beliebigen Namen X zwischen X0, X1

   ELSE ‚Nicht vorhanden', STOP



3. IF X < „Kundera" THEN GOTO 2 mit X0 := X
4. IF X > „Kundera" THEN GOTO 2 mit X1 := X
5. IF X = „Kundera" THEN ‚Gefunden', STOP

Beispiel für einen Algorithmus (5/6)

Sprache: Bestandteile einer Programmiersprache
Komplizierte Teile: Umgangsprachlich

Pseudocode: schrittweises Überführen in Programm
(Schritte: präzisieren der umgangssprachlichen Teile)

Objekte: Namen

Operationen: :=, <, > (zu präzisieren)

Reihenfolge: Nummerierung, IF ... THEN ... ELSE, GOTO, „,",
STOP

Beispiel für einen Algorithmus (5/6)

„Euklidscher Algorithmus (Variante) zur Bestimmung des GGT zweier natürlicher Zahlen"

Eingabe: Natürliche Zahlen n und m,

Ausgabe: GGT

1. IF n=m THEN GGT := n, STOP ← Abbruch der Schleife
2. IF n>m THEN n := n-m

   ELSE m := m-n ← Manipulation nach Fallunterscheidung*
3. GOTO 1 ← Rücksprung an den Beginn der Schleife

*die Differenz der größeren von der kleineren Zahl wird gebildet und ersetzt die größere Zahl

Beispiel für einen Algorithmus (6/6)

Sprache: Umgangssprache, mathematische Formelelemente

Objekte: natürliche Zahlen

Operationen: mathematische Operationen

Reihenfolge: Nummerierung, IF ... THEN ... ELSE, GOTO, „,",
END

Die Korrektheit des Algorithmus soll angenommen und hier nicht bewiesen werden!

Eigenschaften eines Algorithmus

- Terminierend
  Für alle korrekten Eingaben hält der A. nach endlich vielen Schritten an.
- Vollständigkeit
  Alle Fälle, die bei korrekten Eingabedaten im Verlaufe der Abarbeitung eines A. auftreten können, müssen berücksichtigt werden.
- Determiniert
  Der A. liefert bei jedem Ablauf mit den gleichen Eingaben das gleiche Ergebnis
- Deterministisch
  Der A. läuft bei jedem Ablauf mit den gleichen Eingaben durch dieselbe Berechnung

Beispiele zu Eigenschaften (1/2)

- Nichtdeterministischer Algorithmus
  Verbesserung von „Suchen des Namen „Kundera“ im Telefonbuch“ wegen der „*beliebigen*“ Auswahl des Namens X.
  Dieser Algorithmus ist dennoch determiniert!
- Nicht determinierter Algorithmus

  1. Wählen Sie eine *beliebige* Zahl zwischen 1 und 100
  2. Multiplizieren Sie die Zahl mit der Anzahl der seit dem 1.1.2000 vergangenen Minuten
  3. Schreiben Sie das Ergebnis auf

  > Nicht determiniert wegen
  – Beliebigkeit der Auswahl zum Ausführungszeitpunkt *und*
  – der Abhängigkeit vom Ausführungszeitpunkt

Beispiele zu Eigenschaften (2/2)

- Nichtterminierender Algorithmus 1
  Bestimmen Sie alle Nachkommastellen der Zahl *e* = 2,7182....
- Nichtterminierender Algorithmus 2
  1. Wählen Sie eine *beliebige* Zahl X
  2. Multiplizieren Sie X mit 2, bis X größer 100 ist.
     (Präzisere Schreibweise: X := X * 2)
  3. Schreiben Sie das Ergebnis auf

  > Nicht terminierend für x < 0 (Endlosschleife)
- Nicht vollständiger Algorithmus
  1. Wählen Sie eine *beliebige* Zahl X
  2. IF X < 0 THEN X := X*2 ELSE X := 2/X FI

  > Was passiert, wenn X = 0 gewählt wird?

Aufbau von Algorithmen (1/5)

Algorithmen werden konstruiert, indem kleinere Bausteine zu größeren zusammengesetzt werden. Gängige Bausteine und Kompositionsprinzipien von Algorithmen:

- Elementare Operation: nicht weiter erläutert
  schneide Fleisch in kleine Würfel
- Sequentielle Ausführung: bei kausaler Abhängigkeit oder wenn nur ein ” Prozessor“ zur Verfügung steht
  bringe das Wasser zum Kochen, dann gib Paket Nudeln hinein, schneide das Fleisch, dann das Gemüse
- Parallele Ausführung: falls es keine Kausalitäts-beziehungen gibt und mehrere ” Prozessoren“ zur Verfügung stehen
  ich schneide das Fleisch, du das Gemüse

Aufbau von Algorithmen (2/5)

- Bedingte Ausführung: nur wenn Bedingung erfüllt ist
  wenn Soße zu dünn, dann füge Mehl hinzu
- Schleife: Wiederholung, bis Bedingung erfüllt ist
  Rühre so lange, bis Soße braun
- Unterprogramm: die Tätigkeit wird anderswo beschrieben und ist mehrfach benutzbar
  Bereite Soße (siehe S. 42)
- Rekursion: Anwendung desselben Algorithmus auf (kleineres) Teilproblem
  Hol Wasser, Henry! - Ein Loch ist im Eimer - So stopf es - Womit denn - Mit Stroh - Das Stroh ist zu lang - So kürz es - Die Axt ist zu stumpf - So schärf sie - Der Stein ist zu trocken - So hol Wasser, Henry! -...

Aufbau von Algorithmen (3/5) Beispiel

Bedingte Ausführung

Parallelität

Verbesserung von Variante 1 (modifiziert)

Telefonbuch = alphabetisch geordnete Liste von Namen

1. X0 := erster Name || X1 := letzter Name

Unter-programm

2. IF es gibt einen Namen zwischen X0, X1
   THEN lies beliebigen Namen X zwischen X0, X1
   ELSE ‚Nicht vorhanden‘, STOP

⋮

3. IF X < „Kundera“ THEN GOTO 2 mit X0 := X
4. IF X > „Kundera“ THEN GOTO 2 mit X1 := X
5. IF X = „Kundera“ THEN ‚Gefunden‘, STOP

Elementare Operation

Sequenz

Aufbau von Algorithmen (4/5) Beispiel

„Euklidscher Algorithmus (Variante) zur Bestimmung des GGT zweier natürlicher Zahlen“

Eingabe: Natürliche Zahlen n und m, Ausgabe: GGT

1. IF n=m THEN GGT := n, STOP
2. IF n>m THEN n := n-m
   ELSE m := m-n
3. GOTO 1

Schleife

Rekursion!

Anwendung von „die Differenz der größeren von der kleineren Zahl wird gebildet und ersetzt die größere Zahl“ auf „kleineres“ Problem

Aufbau von Algorithmen (5/5)

Mehr als die genannten Strukturen braucht man nicht!

Es reichen sogar

elementaren Operationen + Sequenzen + Wiederholungen
(weil in Schleifen jeweils versteckte Bedingungsabfragen stecken)

um alles algorithmisch beschreiben (programmieren) zu können, was überhaupt berechenbar (programmierbar) ist.

2.2 Berechenbarkeit

Ausgangsfrage:

Gibt es mathematisch beschreibbare Problemstellungen, die nicht berechenbar sind?

Problem:

Obige Definition von Algorithmen ist zu unscharf und daher nicht mathematisch nachprüfbar!

Lösung:

Präzisierung der Definition von Algorithmus!
> verschiedene Ansätze

Die Turing-Maschine (1/6)

A. M. Turing (1912–1954): britischer Mathematiker

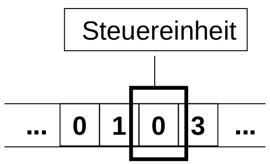
Ausgangspunkt:
‚berechenbar' = auf einer Maschine ausführbar

Turing-Maschine (1936):
mathematisches (theoretisches) Modell einer Rechenmaschine bzw. eines *Automaten*
> Unabhängigkeit von technischer Realisierung

Exkurs Automat (Informatik): mathematisches Modell eines Geräts, dass zu Eingaben eine Ausgabe produziert (ohne erstere zu zerstören)
> Automatentheorie (theoretische Informatik)
> formale Sprachen (Erkennen und Übersetzen)

Die Turing-Maschine (2/6)

Bestandteile:
- beliebig langes Band (z.B. Magnet-band) bestehend aus einzelnen Feldern
- Lese-/Schreibkopf für genau ein Feld
- Alphabet ***B*** von Zeichen, die in den Feldern gespeichert werden können
- Aktionen ***r,l,s*** $\notin$ ***B*** des Kopfes:
  1. ein Feld nach rechts bewegen: ***r***
  2. ein Feld nach links bewegen: ***l***
  3. stoppen: ***s***
  4. schreiben von **x**$\in$***B***: ***x***
- Steuereinheit: steuert Bewegung und Schreibaktion



Die Turing-Maschine (3/6)

Turing-Tafel:
- spezielles Programm
- Folge von Anweisungen $s \in \boldsymbol{S}$ der Gestalt

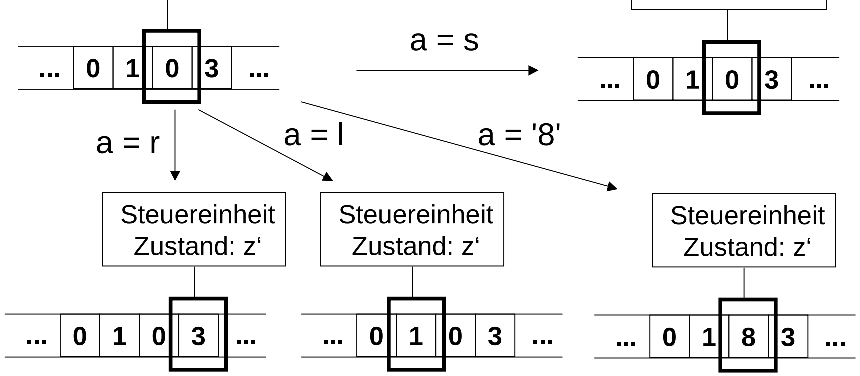
(z, x, a, z') alternativ z x a z'

z: Anfangszustand der Maschine *vor* Ausführung dieser Zeile, $z \in \{0,1, ...,m, \perp\} =: \boldsymbol{Z}$ mit '$\perp$' = Endzustand
x: Buchstabe (Zeichen), $x \in \boldsymbol{B}$
a: Aktion, $a \in \{r, l, s\} \cup \boldsymbol{B} =: \boldsymbol{A}$
z': Folgezustand *nach* Ausführung von s, $z' \in \boldsymbol{Z}$

$\forall\, z \in \boldsymbol{Z}, x \in \boldsymbol{B}\ \exists!\, s \in \boldsymbol{S}: s = (z,x,a,z')$ mit $a \in \boldsymbol{A}$, $z' \in \boldsymbol{Z}$
=> *Programm ist deterministisch*

Die Turing-Maschine (4/6)

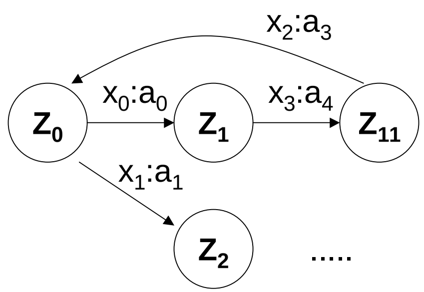
Ausführung
(***B*** := {1,..,9})



Die Turing-Maschine (5/6)

Alternative Darstellung der Turing-Tafel:

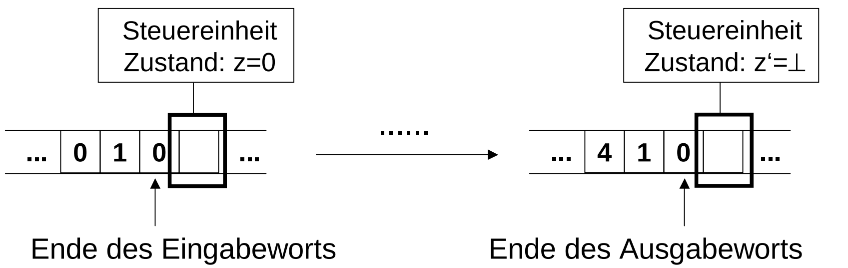
Zustandsüberführungsdiagramm



Die Turing-Maschine (6/6)

Zu Beginn der Ausführung:
z=0

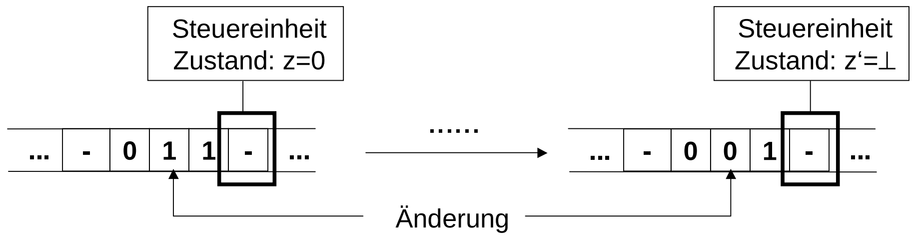
Zum Ende der Ausführung:
z'= ⊥



Die Turing-Maschine: Beispiel (1/3)

Ausgangssituation: Auf dem (sonst) leeren Band steht ein Eingabedatum (Folge aus '0' und '1' ohne '-' = leer)

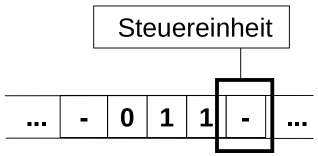
Aufgabe: Die am weitesten links stehende '1' soll in '0' gewandelt werden



Die Turing-Maschine: Beispiel (2/3)

Lösung:

- ***B***={0,1,-}
- ***Z***={0,1,2,3, ⊥}



Turing-Tafel:

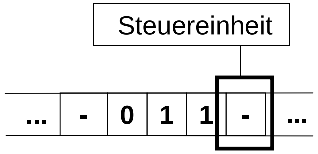
Anfangszustand, gelesenes Zeichen, Aktion, Endzustand

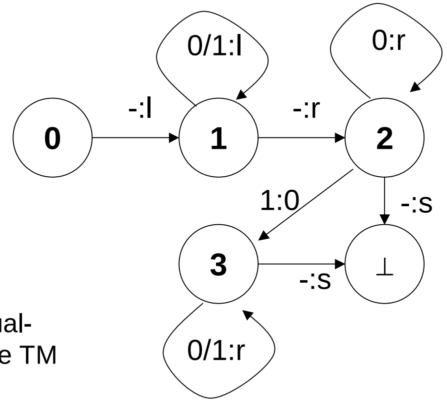
z	x	a	z'	
0	0	kann nicht auftreten		
0	1		"	
0	-	l	1	
1	0	l	1	
1	1	l	1	
1	-	r	2	linkes Feld erreicht
2	0	r	2	
2	1	0	3	Überscheiben der 1 mit 0
2	-	s		Eingabe nur 0'en oder '-'
3	-	s		
3	0	r	3	
3	1	r	3	

Die Turing-Maschine: Beispiel (3/3)

Lösung als Zustandsübergangsdiagramm:

- ***B***={0,1,-}
- ***Z***={0,1,2,3, ⊥}





Anmerkung:

Stellt die Inschrift eine Zahl in Dual-Schreibweise dar, so realisiert die TM folgende Funktion f:

f(y) := y - "größte in y enthaltene Zweierpotenz"

Turing-Berechenbarkeit

Definition:

Eine mathematische Funktion $f{:}\mathrm{IN}_0 \rightarrow \mathrm{IN}_0$ heißt Turing-berechenbar, wenn es eine TM (d.h. eine Turing-Tafel) gibt, die angesetzt auf ein Band mit beliebigem (zulässigem) Argument $y \in \mathrm{IN}_0$ nach endlich vielen Schritten anhält hinter dem Funktionswert f(y).

Anschaulich:

f ist Turing-berechenbar, falls sich die Berechnung in die elementaren Schritte einer Turing-Tafel zerlegen lässt, d.h. die Turing-Tafel ist ein Spezialfall von „Algorithmus“ (anschauliche Definition).

Church‘sche These (1/2)

Statt TM:

Festlegung einer Klasse von berechenbaren Funktionen mit mathematischen Methoden.

Idee:

Berechenbarkeit bedeutet „Konstruierbarkeit“ nach mathematischen Regeln (besonders „Induktion $n \rightarrow n+1$“)

Es existieren verschiedene derartige Regelätze. Der bekannteste beschreibt die Klasse der „µ-rekursiven Funktionen“

$F_{\text{Turing-berechenbar}} := \{f{:}\mathrm{IN}_0 \rightarrow \mathrm{IN}_0 \mid f \text{ Turing-berechenbar}\}$

$F_{\mu\text{-berechenbar}} := \{f{:}\mathrm{IN}_0 \rightarrow \mathrm{IN}_0 \mid f\ \mu\text{-berechenbar}\}$

....

Church‘sche These (2/2)

Satz: $F_{\text{Turing-berechenbar}} = F_{\mu\text{-berechenbar}} = \ldots = F_{\text{allgemein-rekursiv}}$

D.h.: Verschiede Ausgangspunkte zur Präzisierung von „berechenbar“ führen auf einen mathematischen Begriff.

berechenbar ⟶ **Turing-berechenbar**

Church‘sche These (1936):
„Der *intuitive* Begriff „berechenbar“ wird (für theoretische „Untersuchungen“) durch den mathematischen Begriff „Turing-berechenbar“ (oder einen äquivalenten Begriff) erfasst.“

These! Kein Satz! Dennoch „allgemein“ akzeptiert!

Unentscheidbarkeit des Halteproblems

Das Halteproblem anschaulich:

Existiert ein Programm (bzw. ein Algorithmus), dass

1. für ein beliebiges anderes Programm *P* und
2. eine Eingabe *x* des Programms *P*

entscheidet, ob das Programm P angesetzt auf die Eingabe *x* stoppt, oder nicht?

Falls Ja: automatische Programmverifikation!

Aber leider: Nein! Beweis?

Unentscheidbarkeit des Halteproblems: Grundlagen (1/7)

Relation:

- Beispiel: "<„ → $x_1 < x_2$ liefert <u>W</u>ahr oder <u>F</u>alsch
- Formal:
  1. Eine *zweistellige* Relation *R* ist nichtleere Teilmenge von $T \times U$, wobei *T* und *U* nichtleere Mengen sind
  2. $(t,u) \in R$ ist gleichbedeutend mit „*t* steht in Relation *R* zu *u*“ alternativ *tRu* oder auch *Rtu*
- *Analog n-stellige Relationen:* $R_n \subseteq T_1 \times T_2 \times_{..} \times T_n$ wenn $(t_1, t_2,\ldots,t_n) \in R_n$ dann alternativ $R(t_1, t_2,\ldots,t_n)$ oder $R(t_1\, t_2\ldots t_n)$ oder $R\, t_1\, t_2\ldots t_n$

Unentscheidbarkeit des Halteproblems: Grundlagen (2/7)

- Wenn T_i={A,....,z} für $1 \leq i \leq n$, dann können *n*-stellige Argumente auch als *n*-stellige Worte betrachtet werden
- Definition Alphabet:
  – Eine endliche nichtleere Menge ***B*** heißt *Alphabet*, $b \in$ ***B*** heißen *Zeichen*
  – Ein Wort der Länge *n* über ***B*** ist eine Aneinander-reihung von *n* Zeichen aus ***B***
  – Das „nicht-Vorhandensein“ eines Wortes wird als sogenanntes *Leeres Wort* beschrieben

Unentscheidbarkeit des Halteproblems: Grundlagen (3/7)

Formale Definition:

– $\boldsymbol{B}^n := \{ (b_1, b_2, ..., b_n) \mid b_i \in \boldsymbol{B} \text{ für } 1 \le i \le n\}$
$= \boldsymbol{B} \times \boldsymbol{B} \times ... \times \boldsymbol{B}$ (n Mal)
$= \{ b_1 b_2 ... b_n \mid b_i \in \boldsymbol{B} \text{ für } 1 \le i \le n\}$
(Je nach ***B*** und Kontext)

– $\boldsymbol{B}^0 := \{ \varepsilon \}$ ε ist das leere Wort

– $\boldsymbol{B}^* :=$ ist die Menge *aller* endlichen Worte über ***B***

– $\boldsymbol{B}^+ :=$ ist die Menge *aller* endlichen Worte über ***B*** *ohne das* leere Wort

Unentscheidbarkeit des Halteproblems: Grundlagen (4/7)

Vereinbarung:
Im Folgenden stets festes Alphabet ***B*** ={ 0,1 }
> Keine Einschränkung, da alle Probleme, die mit einer TM gelöst werden können, mit TM über {0,1} gelöst werden können (> theoretische Informatik)

Relationen über $\boldsymbol{B}^*$**:**

- für $b \in \boldsymbol{B}^*$ liefert ***R****x* <u>w</u>ahr oder <u>f</u>alsch
- Beispiel:

B = {A,...,z}, ***D***x für $x \in \boldsymbol{B}^*$ ist wahr, falls „x steht im Duden"

=> ***D***(steht) → f, ***D***(in) → w, ***D***(grmblfjx) → f

Unentscheidbarkeit des Halteproblems: Grundlagen (5/7)

Turing-Entscheidbarkeit:
Eine Relation ***R*** heißt *Turing-entscheidbar*, falls eine Turing Maschine T existiert, die

- angesetzt auf ein (zulässiges) Argument *x*
- nach endlich vielen Schritten stehen bleibt, und zwar

1. mit dem Ergebnis 1 falls R(x) gilt,
2. oder mit 0, falls R(x) nicht gilt.

Entscheidbarkeit und Berechenbarkeit sind eng verbunden!

Anschaulich:
R ist entscheidbar, wenn sich eine TM angeben lässt

Unentscheidbarkeit des Halteproblems: Grundlagen (6/7)

Turing-Tafeln als Bandinhalt (1/2):

- Durch Hintereinanderschreiben der vollständigen Anweisungen einer Turing-Tafel *TT* wird diese Wort über $\{r,l,s\} \cup \boldsymbol{B} \cup \boldsymbol{Z} =: \boldsymbol{B_T}$
Beispiel: 0-l1 00l1 0...
- Man kann eine *TT* einer *TM* T eindeutig (d.h. ohne Verlust von Informationen) durch ein Wort *P*(T) über $\boldsymbol{B_T}$ ausdrücken
- *P*(T) kann selbst wieder Eingabe einer TM sein
- Schreibweise:
$x,y \in \boldsymbol{B}$ => (x,y) = Aneinanderreihung von x und y

Unentscheidbarkeit des Halteproblems: Grundlagen (7/7)

Turing-Tafeln als Bandinhalt (2/2):

- Definition:
  Relation ***Q*** über B_T^* speziell über Worten der Gestalt (*P*(T), *x*), wobei
  1. *P*(T) eine *TT* und
  2. *x* ein Wort aus *B** (Eingabe für T) symbolisiert.

=> ***Q***(*P*(T), *x*) ist wahr, falls T angesetzt auf *x* stoppt nach endlich vielen Schritten
(Formalisierung des Halteproblems)

Unentscheidbarkeit des Halteproblems

Satz: **Q** ist nicht entscheidbar

⇒ Es gibt keine TM (keinen Algorithmus), die für eine beliebige *TT* und beliebigen Input *x* entscheidet, ob die TM T angesetzt auf *x* stoppt, oder nicht.

⇒ Es gibt kein (Java-)Programm, das für ein beliebiges Java-Programm *J* entscheidet, ob *J* Endlosschleifen enthält, oder nicht.

Unentscheidbarkeit des Halteproblems: Beweisskizze

Indirekter Beweis (durch Widerspruch)

1. Annahme: Es gibt eine TM T_0, die ***Q*** entscheidet
2. Wir ziehen logische Folgerungen aus der Annahme
3. Wir zeigen: Folgerungen führen auf logische Widersprüche
4. Sind die Folgerungen korrekt, sind Widersprüche (in mathematischer Logik) nicht möglich, also muss die Annahme falsch sein
5. Also muss das Gegenteil der Annahme gelten: d.h. der Satz gilt!

Unentscheidbarkeit des Halteproblems: Beweis (1/4)

1. Annahme

   Satz gilt nicht, d.h. es gibt eine TM T_0, die angesetzt auf (*P*(T), *x*) stoppt
   - auf 1, falls ***Q***(*P*(T), *x*) wahr ist
   - auf 0, falls ***Q***(*P*(T), *x*) falsch ist

2. Definiere weitere Relation ***Q'*** auf *P*(T) mit
   Q' (*P*(T)) := ¬ ***Q***(*P*(T), *P*(T))

   d.h. ***Q'*** (*P*(T)) ist wahr, falls T angesetzt auf *P*(T) (d.h. auf Ihre eigene Tafel) nicht stoppt.

Unentscheidbarkeit des Halteproblems: Beweis (2/4)

3. Da ***Q*** entscheidbar (Annahme), ist offenbar auf auch ***Q`*** entscheidbar,
   d.h. es gibt eine TM T‘, die angesetzt auf beliebiges *P*(T) stoppt
   - auf 1, falls ***Q‘***(*P*(T)) wahr ist
   - auf 0, falls ***Q‘***(*P*(T)) falsch ist
4. T‘ kann zu T* geändert werden, so dass T* angesetzt auf *P*(T)
   - auf 1 stoppt, falls ***Q‘***(*P*(T)) wahr ist
   - nicht stoppt, falls ***Q‘***(*P*(T)) falsch ist (durch Anhängen einer Endlosschleife)

Unentscheidbarkeit des Halteproblems: Beweis (3/4)

5. Gilt ***Q‘***(*P*(T*)) ? Zwei Fälle:

5.1 ***Q‘***(*P*(T*)) ist wahr

$\Rightarrow$ T‘ angesetzt auf *P*(T*) stoppt auf 1

$\Rightarrow$ T* angesetzt auf *P*(T*) stoppt auf 1

$\Rightarrow$ ***Q***(*P*(T*), *P*(T*)) = wahr = ¬ ***Q‘*** (*P*(T*)) wg. Def. ***Q***

$\Rightarrow$ ***Q‘*** (*P*(T*)) ist falsch, also muss 5.1 falsch sein! > 5.2

5.2 ***Q‘***(*P*(T*)) falsch

$\Rightarrow$ *T‘* angesetzt auf *P*(T*) stoppt auf 0

$\Rightarrow$ *T** angesetzt auf *P*(T*) stoppt nicht

$\Rightarrow$ ***Q***(*P*(T*), *P*(T*)) = falsch = ¬ ***Q‘*** (*P*(T *)) wg. Def. ***Q***

$\Rightarrow$ ***Q‘*** (*P*(T*)) ist wahr, also muss 5.2 falsch sein!

Unentscheidbarkeit des Halteproblems: Beweis (4/4)

6. ***Q***‘(*P*(T*)) ist weder falsch noch wahr!

$\Rightarrow$ Logischer Widerspruch, da für alle Argumente (Turing-Tafeln) *P*(T) eines gelten muss.

$\Rightarrow$ Unsere Folgerungen waren korrekt (!), also war unsere Annahme falsch!

$\Rightarrow$ Also gilt ihr Gegenteil:

d.h. ***Q*** ist nicht entscheidbar

Unentscheidbarkeit des Halteproblems: Schlussfolgerungen (1/2)

Turing-Tafeln bilden eine primitive, aber universelle Programmiersprache

$F_{\text{a-berechenbar}} = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \text{ anschaulich berechenbar}\}$

$= F_{\text{Turing-berechenbar}} = \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \text{ Turing-berechenbar}\}$

Church‘sche These

$= \{f : \mathbb{N}_0 \to \mathbb{N}_0 \mid f \text{ berechenbar durch Java-Programm}\}$

Z.B. kann man Java-Programme schreiben, die Turing-Maschinen simulieren.

Unentscheidbarkeit des Halteproblems: Schlussfolgerungen (2/2)

Daher:
Es gibt kein Java-Programm P_0, das für ein beliebiges Java-Programm P_1 und beliebige Eingabe x entscheidet, ob P_1 bei Eingabe x in eine Endlosschleife läuft, oder nicht.

=> diese Arbeit ist nicht automatisierbar!

Allerdings:
Für spezielle Programme P1 oder spezielle x ist schon entscheidbar, ob P1 terminiert, oder nicht.

Aber:
Auffinden des Verfahrens erfordert mehr als mechanische Arbeit bzw. programmierte Abläufe.

=> Notwendigkeit für menschlichen Geist, wissenschaftliche Intuition, Kreativität.

Semi-Thue-Systeme (1/10)

A. Thue (1863–1922): norwegischer Mathematiker und Logiker

Semi-Thue-Systeme: Alternative theoretische Darstellungsform für Algorithmen (einfach und allgemein)

Grundidee:
Überführe ein *Eingabewort* durch sukzessive Anwendung sogenannter *(Produktions-/Ersetzungs-)Regeln* in ein *Ausgabewort*

> Semi-Thue-Systeme sind sogenannte *Ersetzungssysteme*

> *Formale Sprachen (Theoretische Informatik), Grammatiken*

Semi-Thue-Systeme (2/10)

Elemente (Erinnerung):

Σ	Alphabet
$x = x_1 x_2 \ldots x_n$	Wort über Σ
$\Sigma^* = \{x_1 x_2 \ldots x_n \mid n \in \mathbb{N}_0, x_n \in \Sigma\}$	Menge aller Worte über Σ
$\lvert x \rvert = \lvert x_1 x_2 \ldots x_n \rvert = n$	Länge eines Wortes
$x_i \ldots x_{i+k-1}$	Teilwort der Länge k

Semi-Thue-Systeme (3/10)

Ersetzungsregeln:

$$\boxed{l \rightarrow r} \qquad l, r \in \Sigma^*$$

Beispiel:

$abb \rightarrow c \qquad l = abb,\ r = c \in \Sigma^* = \{a,b,c\}^*$

Eine Ersetzungsregel ist *anwendbar* auf ein Wort $x \in \Sigma^*$, wenn l als *Teilwort* in x vorkommt.

Ist eine Ersetzungsegel anwendbar, so wird das *Teilwort* in x, dass l entspricht durch r *ersetzt*!

Semi-Thue-Systeme (4/10)

Regelanwendung:

$l = x_i \ldots x_{i+k-1} \rightarrow r = y_1 \ldots y_m \qquad l, r \in \Sigma^*$

Ableitung

$x = x_1 x_2 \ldots x_{i-1} \underbrace{x_i \ldots x_{i+k-1}}_{l} x_{i+k} \ldots x_n \underbrace{\Rightarrow}_{l \rightarrow r} x_1 x_2 \ldots x_{i-1} \underbrace{y_1 \ldots y_m}_{r} x_{i+k} \ldots x_n = x'$

Beispiel:

$abb \rightarrow c \qquad l=abb,\ r=c \in \Sigma^* = \{a,b,c\}^*$

$x = \underbrace{baabba}_{l} \underbrace{\Rightarrow}_{l \rightarrow r} \underbrace{baca}_{r} = x'$

Semi-Thue-Systeme (5/10)

Ableitung/Erzeugung (synonym):

$x \Rightarrow x' \Rightarrow x'' \Rightarrow x''' \ldots$	<u>Ableitung</u>/Erzeugung von *x'* aus *x* in *n* Schritten
$x \Rightarrow^+ x'$	Ableitung in *einem* oder *mehreren* Schritten
$x \Rightarrow^* x'$	Ableitung in *keinem* oder *mehreren* Schritten $x \Rightarrow^* x'$ bedeutet $x \Rightarrow^+ x'$ oder $x=x'$

Alternative Sprechweise: *x'* kann auf *x* reduziert werden

Semi-Thue-Systeme (6/10)

Definition Semi-Thue System

$\boldsymbol{T} = (\Sigma, \boldsymbol{P})$ mit Σ = Alphabet und $\boldsymbol{P}$ = endliche oder ab-zählbar unendliche Menge von Ersetzungsregeln über Σ^* ist ein Semi-Thue-System, wenn folgende Metaregeln gelten:

Metaregeln bei Semi-Thue-Systemen:

- Wenn *mindestens eine* Ersetzungsregel auf ein Ergebnis einer Ableitung *anwendbar* ist, so *muss* ein weiterer Ableitungsschritt erfolgen.
- Wenn mehrere Regeln anwendbar sind (oder eine Regel an mehreren Stellen), wähle Regel und Stelle *beliebig*.
- Wiederhole die Anwendung von Regeln *beliebig* oft.

Semi-Thue-Systeme (7/10)

Definition „von *x* erzeugte (formale) Sprache":

$L_x = L(T, x) = \{ y \mid x \Rightarrow^* y \wedge \text{Ableitung genügt Metaregeln} \}$ ist die von $x \in \Sigma^*$ erzeugte Sprache (=Menge von Worten/Sätzen)

L_x enthält *alle* ausgehend von *x* unter Einhaltung der Metaregeln ableitbaren Worte.

Beispiel: *„Sprache" der ungeraden Binärzahlen*

$\Sigma = \{0, 1, -\}$, $\boldsymbol{P} = \{1- \overset{1}{\rightarrow} 01-,\ 1- \overset{2}{\rightarrow} 11-,\ - \overset{3}{\rightarrow} \varepsilon \}$

$L(T, 1-) = \{1, 01, 11, 001, 011, 101, 111, \ldots\}$

(Eine) Ableitung: $1- \underset{2}{\Rightarrow} 11- \underset{1}{\Rightarrow} 101- \underset{2}{\Rightarrow} 1011- \underset{3}{\Rightarrow} 1011$

Semi-Thue-Systeme (8/10)

Beispiel: *Addition im Strichcode*

$\Sigma = \{ \mid, +, = \}$, $\boldsymbol{P} = \{ \underset{1}{+| \to |+}, \underset{2}{+= \to \varepsilon} \}$

Ableitung:

$||+|||= \underset{1}{\Rightarrow} |||+||= \underset{1}{\Rightarrow} ||||+|= \underset{1}{\Rightarrow} |||||+= \underset{2}{\Rightarrow} |||||$

> Semi-Thue-Systeme zeigen Grundform von Algorithmen

Operatoren = Ersetzungsregeln

Eingabe: Ausgangswort *x*

Ausgabe: Abgeleitetes Wort *y* mit $x \Rightarrow^* y$

Endebedingung: Keine Regel mehr anwendbar *(Metaregel)*

Semi-Thue-Systeme (9/10)

Beispiel: *Ableitungsvarianten (> ST-System als Algorithmus)*

$\Sigma = \{0, 1\}$, $\boldsymbol{P} = \{ \underset{1}{0 \to 00}, \underset{2}{0 \to 1} \}$

Ableitungen:

$0 \underset{1}{\Rightarrow} 00 \underset{2}{\Rightarrow} 01 \underset{2}{\Rightarrow} 11$ terminiert

$0 \underset{1}{\Rightarrow} 00 \underset{2}{\Rightarrow} 10 \underset{2}{\Rightarrow} 11$ andere Berechnung, gleiches Ergebnis (> nicht deterministisch)

$0 \underset{1}{\Rightarrow} 00 \underset{1}{\Rightarrow} 000 \underset{2}{\Rightarrow} 100 \underset{2}{\Rightarrow} 110 \underset{2}{\Rightarrow} 111$ andere Berechnung, anderes Ergebnis (> nicht determiniert)

$0 \underset{1}{\Rightarrow} 00 \underset{1}{\Rightarrow} 000 \underset{1}{\Rightarrow} 0000 \ldots$ terminiert nicht

Semi-Thue-Systeme (10/10)

Probleme bei der Verwendung von Semi-Thue-Systemen zur Beschreibung von Algorithmen:

1. für welche Worte **terminiert** die Folge der Ableitungen?
2. ist der Algorithmus **deterministisch**, d.h. ist in jedem Schritt höchstens eine Regel anwendbar?
3. ist der Algorithmus **determiniert**, d.h. kommt bei jedem Ableitungsweg dasselbe heraus: $|L(T, x)| = 1$?

Nachweis der Terminierung:

- Finde Funktion f: $\Sigma^* \to \mathbb{N}_0$ derart, dass für alle direkten Ableitungen $x \Rightarrow y$ gilt: $f(y) < f(x)$
- Im obigen ersten Beispiel:
  $f(x)$ = Zahl der "|" rechts von "+" plus Zahl der "+"-Zeichen

Markov-Algorithmen (1/3)

A. A. Markov (1903–1997): russischer Mathematiker

Definition Markov-Algorithmus

Ein Markov-Algorithmus ist ein deterministisches Semi-Thue-System $\boldsymbol{T} = (\Sigma, \boldsymbol{P})$ mit *endlich vielen* Regeln (d.h. $|\boldsymbol{P}| = n \in \mathbb{N}$), speziellen *haltenden* Regeln ($x \to\!. \, y$), und folgenden *Metaregeln*:

Metaregeln bei Markov-Algorithmen:

- Wähle in jedem Schritt *die erste anwendbare* Regel. Wende sie auf das *am weitesten links stehende* Teilwort an.
- Wende Regeln an, *bis haltende Regel* angewendet wurde, oder *bis keine Regel* mehr anwendbar ist.

Markov-Algorithmen (2/3)

Definition „*Gesteuerter Markov-Algorithmus*"

Ein *gesteuerter Markov-Algorithmus* $\boldsymbol{M} = (\Sigma, \Sigma_N, \boldsymbol{P})$ ist ein Markov-Algorithmus mit zusätzlichem Vorrat an *Steuerzeichen* (Schiffchen) Σ_N, wobei $\Sigma \cap \Sigma_N = \emptyset$ und $\boldsymbol{P}$ Regeln über $(\Sigma \cup \Sigma_N)^*$

- Steuerzeichen kommen *weder* in der Eingabe- *noch* in der Ausgabe vor
- Hiermit ist *jeder beliebige Algorithmus* zu formulieren bzw. jede algorithmisch beschreibbare Berechnung zu beschreiben (gilt natürlich ebenso für die allgemeineren Semi-Thue-Systemen)

Markov-Algorithmen (3/3)

Beispiel zum *gesteuerten Markov-Algorithmus* ($x := x + 1$)

$\Sigma = \{0, 1\}$, $\Sigma_N = \{\alpha, \beta\}$

$$\boldsymbol{P} = (\ \alpha 1 \rightarrow 1\alpha,\ \ \alpha 0 \rightarrow 0\alpha,\ \ \alpha \rightarrow \beta,\ \ 1\beta \rightarrow \beta 0,\ \ 0\beta \rightarrow_{\bullet} 1,\ \ \beta \rightarrow_{\bullet} 1,\ \ \varepsilon \rightarrow \alpha\ )$$

Eine Ableitung

$1011 \Rightarrow \alpha 1011 \Rightarrow 1\alpha 011$
$\Rightarrow 10\alpha 11 \Rightarrow 101\alpha 1$
$\Rightarrow 1011\alpha \Rightarrow 1011\beta$
$\Rightarrow 101\beta 0 \Rightarrow 10\beta 00$
$\Rightarrow_{\bullet} 1100$

Reihenfolge der Regeln ist entscheidend! Z.B. werden Regeln *nach* $\varepsilon \rightarrow \alpha$ nie ausgeführt.

Formale Systeme (1/2)

Definition:

($\boldsymbol{U}$, $\Rightarrow$) heißt *formales System*, wenn gilt:

1. $\boldsymbol{U}$ ist eine endliche oder abzählbar unendliche Menge
2. $\Rightarrow \subseteq \boldsymbol{U} \times \boldsymbol{U}$ ist eine Relation über $\boldsymbol{U}$
3. es gibt einen Algorithmus, der zu jedem $l \in \boldsymbol{U}$ jedes $r \in \boldsymbol{U}$ mit $l \Rightarrow^* r$ in endlich vielen Schritten berechnen kann

Beispiele:

1. Semi-Thue-Systeme ($\boldsymbol{U} = \Sigma^*$, $\Rightarrow \cong$ Semi-Thue-Ableitbarkeit)
2. Markov-Algorithmen ($\boldsymbol{U} = \Sigma^*$, $\Rightarrow \cong$ Markov-Ableitbarkeit)
3. $\boldsymbol{U}$ = ¤ × ¤, (m,n) $\Rightarrow$ (m+n,0), ¤ = Menge ganzer Zahlen. Dagegen bilden die reellen Zahlen mit der Addition kein formales System!

Formale Systeme (2/2)

Definition:

Ein *formales System* heißt *entscheidbar*, wenn es einen Algorithmus gibt, der für beliebige $l,r \in \boldsymbol{U}$ feststellt, ob $l \Rightarrow^* r$

Erinnerung: (Turing-)Entscheidbarkeit einer Relation

Kalkül:

Formales System, in dem $\Rightarrow$ durch eine endliche Menge von *Grundregeln* $x \rightarrow y$ zusammen mit einer endlichen Menge von *Metaregeln* definiert ist.

2.3 Programmiersprachen: Algorithmus & Programm(iersprache)

In den Anfängen:
Formulierung von Algorithmen durch direkte Anweisungen an den Prozessor in sogenannter Maschinensprache (0,1). Eingabe per Schalter und Kabel.

Assembler:
Eintippen von Buchstaben und einfachen Anweisungen
(z.B. add A,9 → Addiere zum Akkumulator 9)
Nachteil: Programmieren kompliziert und maschinenabhängig

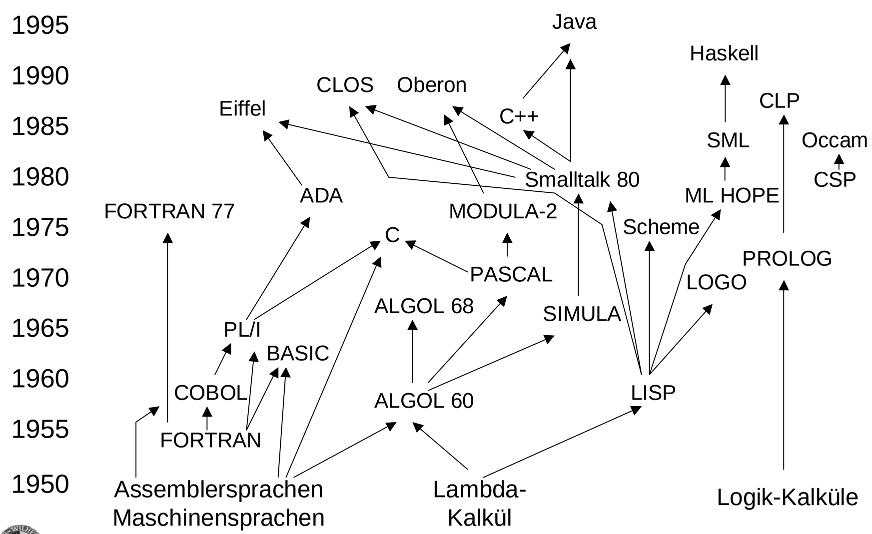
Entwicklung „höherer Programmiersprachen“ mit Übersetzern:
ein Übersetzer (Compiler) ist ein Programm, welches Programme einer (höheren) Programmiersprache in Programme einer anderen Programmiersprache übersetzt.

Beschreibung von Algorithmen

Algorithmen können auf versch. Arten beschrieben werden:

- funktional
  > *Grundlage: Lambda-Kalkül*
  > AuD II
- imperativ
  > *Grundlage: Maschinenbezug (Programm = Folge von Befehlen, von-Neumann-Architektur)*
  > Abschnitt 3
  - objektorientiert
    > Abschnitt 4 u.a.
- *deduktiv*
  > *Grundlage: Logik-Kalküle*
  > *AuD II*

Programmiersprachen: Stammbaum



Semiotik

Lehre von der Entstehung, dem Aufbau und der Wirkweise von Zeichen und Zeichenkomplexen (Symbolen)

$$\boxed{\text{Semiotik = Syntax + Semantik + Pragmatik}}$$

Syntax: Regeln zur Anordnung von Zeichen

Semantik: Objektive Interpretation der Zeichen und Zeichenfolgen, d.h. ihre *kontexttunabhängige Bedeutung*

Pragmatik: Subjektive Interpretation der Zeichen und Zeichenfolgen, d.h. ihre *kontextabhängige Bedeutung*

Semiotik: Beispiele (Programmierung)

Syntax: Beschreibt korrekte Programmtexte

```
S:=S+i, for ....
```

hier muß ein ',' stehen

Semantik: Beschreibt das Ein-/Ausgabeverhalten syntaktisch korrekter Programme

```
S:=S+i
```

der Wert der Variablen S wird um den Wert der Variablen i erhöht

Pragmatik: Bedeutung eines Programms im Bezugssystem

```
S:=S+i
```

Kontostand wird um Betrag i erhöht

Fehler auf Ebene der Pragmatik ;-)

Chomsky-Grammatiken (1/6)

Grammatiken dienen der Festlegung der Syntax einer Sprache.

Eine Grammatik ist eine Menge von Regeln, die festlegt, welche Sätze (Folge von Wörtern/Zeichen) zu einer Sprache gehören.

Noam Chomsky u.a. Linguisten untersuchten ab ca. 1950 Semi-Thue-Systeme mit dem Ziel, die Grammatiken *natürlicher Sprachen* zu formalisieren.

Grundidee: Beschreibung der Struktur von Sätzen in natürlicher Sprache als *Ableitungsbaum*

Bedeutung: Die Methoden werden heute für die *syntaktische Beschreibung* von Programmiersprachen verwendet.

Chomsky-Grammatiken (2/6)

Beispiel: grammatikalische Struktur des Satzes „Ein Fisch schwimmt"

Ableitungsregeln:

Satz → *Subjekt Prädikat*

Subjekt → *Artikel Substantiv*

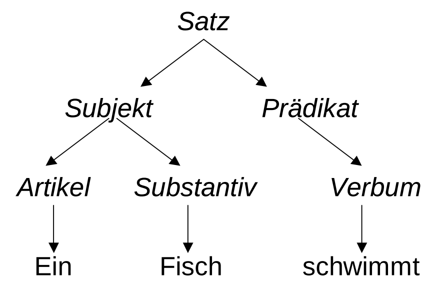
Artikel → ein

Substantiv → Fisch

Prädikat → *Verbum*

Verbum → schwimmt

Darstellung als **Ableitungsbaum**:



Chomsky nannte solche Semi-Thue-Systeme (formale) **Grammatiken** und ihre Regeln **Produktionen**.

Chomsky-Grammatiken (3/6)

Grammatiken als Semi-Thue-Systeme:

- Einführung eines Hilfsalphabets ***N*** von **nichtterminalen** Zeichen (Beispiel: *Satz*, *Prädikat* etc.)
- Die Elemente des Alphabets $\boldsymbol{\Sigma}$ heißen **terminale** Zeichen (Beispiel: ein, Fisch, schwimmt)
- Das Gesamtalphabet ist $\boldsymbol{V} = \boldsymbol{\Sigma} \cup \boldsymbol{N}$
- $\boldsymbol{\Sigma}$ und N sind disjunkt: $\boldsymbol{\Sigma} \cap \boldsymbol{N} = \varnothing$.
- es gibt ein ausgezeichnetes **Startsymbol** $Z \in \boldsymbol{N}$ (Beispiel: *Satz*)
- in Worten der definierten Sprache kommen nur *Terminale* vor, *Nichtterminale* steuern die Struktur und den Ableitungsprozess

Chomsky-Grammatiken (4/6)

Definition: Eine *Grammatik* $G = (\Sigma, \boldsymbol{N}, \boldsymbol{P}, Z)$ ist gegeben durch**:**

Σ	ein Alphabet von *terminalen* Zeichen
$\boldsymbol{N}$	ein Alphabet von *nichtterminalen* Zeichen
$\boldsymbol{P} \subseteq \boldsymbol{V^*NV^*} \times \boldsymbol{V^*}$	eine Menge von Produktionen (auf der linken Seite einer Produktion *muss* ein nichtterminales Zeichen vorkommen)
$Z \in \boldsymbol{N}$	ein Startsymbol

Definition: Die von einer *Grammatik* $G = (\Sigma, N, P, Z)$ erzeugte *formale Sprache* ist gegeben durch**:**

$$L(G) = \{w \in \Sigma^* \mid Z \Rightarrow^* w\}$$

Chomsky-Grammatiken (5/6)

Beispiel: Eine *Grammatik arithmetischer Ausdrücke*

$G = (\boldsymbol{\Sigma}, \boldsymbol{N}, \boldsymbol{P}, Z)$ mit $\boldsymbol{\Sigma} = \{+, *, (,), a, b, \ldots\}$,

$\boldsymbol{N} = \{A, B\}$, $Z = A$

B steht für beliebige Bezeichner

Eine Ableitung (von „(a+b)*c+d“)

$A \Rightarrow A+A \Rightarrow A * A+A$
$\Rightarrow (A)*A+A \Rightarrow (A+A)*A+A$
$\Rightarrow (B+A)*A+A \Rightarrow (B+B)*A+A$
$\Rightarrow (B+B)*B+A \Rightarrow (B+B)*B+B$
$\Rightarrow (a+B)*B+B \Rightarrow (a+b)*B+B$
$\Rightarrow (a+b)*c+B \Rightarrow (a+b)*c+d$

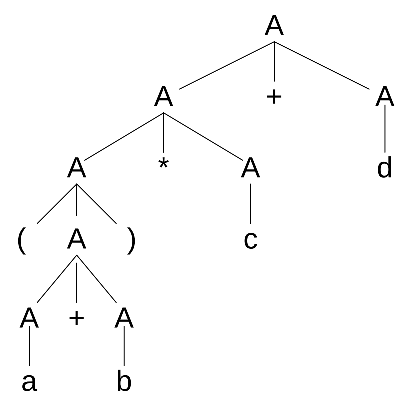
$$\boldsymbol{P} = \{\ A \to A+A,\ A \to A*A,\ A \to (A),\ A \to B,\ B \to a,\ B \to b \ldots\ \}$$

Chomsky-Grammatiken (6/6)

Fortsetzung Beispiel:

Ableitungsbaum zur Ableitung (von „(a+b)*c+d“)

$A \Rightarrow A+A \Rightarrow A*A+A$
$\Rightarrow (A)*A+A \Rightarrow (A+A)*A+A$
$\Rightarrow (B+A)*A+A \Rightarrow (B+B)*A+A$
$\Rightarrow (B+B)*B+A \Rightarrow (B+B)*B+B$
$\Rightarrow (a+B)*B+B \Rightarrow (a+b)*B+B$
$\Rightarrow (a+b)*c+B \Rightarrow (a+b)*c+d$



Erweiterte Backus-Naur-Form (1/3)

Bei Programmiersprachen haben sich kompakte Notationen für *kontextfreie* (links nur ein Nichtterminal) Grammatiken durchgesetzt.

EBNF (J. Backus 1959, P. Naur 1960)

- “ ::= ” statt “→”, Nichtterminale in ⟨...⟩, Terminale in ‘. . . ’
- Alternativen durch “ | “ getrennt
  (A ::= B|C statt A → B, A → C)
- (...) als Metaklammern
- [...] für optionale Teile ([A] ' A‘→ A | ε)
- Stern bedeutet *n*-malige Wiederholung ($n \geq 0$)
- Pluszeichen bedeutet *n*-malige Wiederholung ($n \geq 1$)

Erweiterte Backus-Naur-Form (2/3)

Beispiel (s.o.)

⟨Ausdruck⟩	::=	⟨Ausdruck⟩ ('+'|'∗') ⟨Ausdruck⟩
⟨Ausdruck⟩	::=	'(' ⟨Ausdruck⟩ ')'
⟨Ausdruck⟩	::=	⟨Bezeichner⟩
⟨Bezeichner⟩	::=	'a' | 'b' | 'c' | 'd'

P = {A → A+A,
A → A∗A,
A → (A),
A → B,
B → a,
B → b ... }

Alternative:

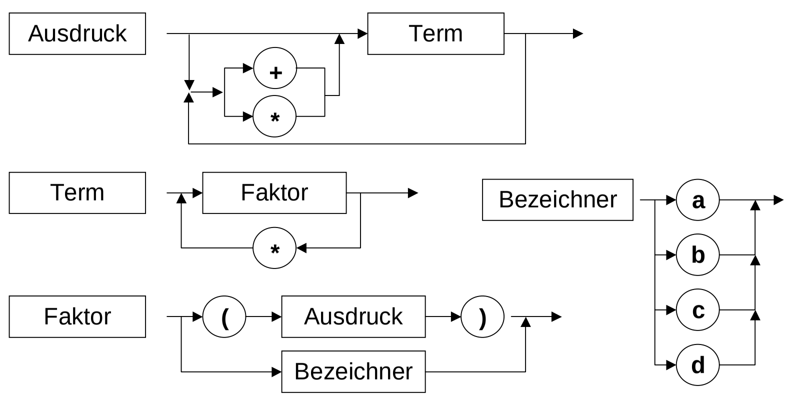
⟨Ausdruck⟩	::=	['+'|'∗'] ⟨Term⟩ (('+'|'∗') ⟨Term⟩)*
⟨Term⟩	::=	⟨Faktor⟩ ('∗' ⟨Faktor⟩)*
⟨Faktor⟩	::=	'(' ⟨Ausdruck⟩ ')' | ⟨Bezeichner⟩
⟨Bezeichner⟩	::=	'a' | 'b' | 'c' | 'd'

Frage: Beschreiben obige Grammatiken die gleiche Sprache?

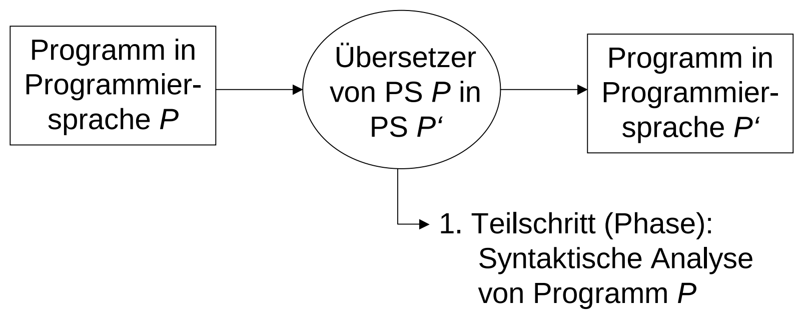
Erweiterte Backus-Naur-Form (3/3)

Syntaxdiagramme: Grafische Beschreibung für EBNF

Beispiel (s.o.: Alternative)



Syntaktische Analyse (1/4)



Anmerkungen:
- Syntaktische Analyse überprüft syntaktische Korrektheit eines Programms bzw. Programmtextes vor dessen Weiterverarbeitung
- *P‘* kann höhere Programmiersprache oder ausführbares Maschinenprogramm sein

Syntaktische Analyse (2/4) Methode

1. $G = (\Sigma, \boldsymbol{N}, \boldsymbol{P}, Z)$ wird als Semi-Thue-System $S(G)=(\boldsymbol{V}, \boldsymbol{P})$ mit $\boldsymbol{V}=\Sigma\cup\boldsymbol{N}$ aufgefasst.

2. Zu Semi-Thue-Systemen können *inverse* Semi-Thue-Systeme (sog. *Reduktionssysteme*) angegeben werden: $S^{-1}(G)=(\boldsymbol{V}, \boldsymbol{P}^{-1})$, mit = $\{ w \rightarrow v \mid v \rightarrow w \in \boldsymbol{P}\}$

3. Ein Wort- bzw Zeichenfolge $x \in \boldsymbol{\Sigma^*}$ gilt als syntaktisch korrekt genau dann, wenn $x \Rightarrow^* Z$

Syntaktische Analyse (3/4) Beispiel (1/2)

Betrachte *Grammatik arithmetischer Ausdrücke (s.o.)*

$G = (\Sigma, \boldsymbol{N}, \boldsymbol{P}, Z)$ mit $\Sigma = \{+, *, (,), a, b, ...\}$,

$\boldsymbol{N} = \{A,B\}$, $Z = A$

B ist ein beliebiger Bezeichner

$$\boldsymbol{P} = \{ \begin{array}{lll} A & \rightarrow A+A & , \\ A & \rightarrow A*A & , \\ A & \rightarrow (A) & , \\ A & \rightarrow B & , \\ B & \rightarrow a & , \\ B & \rightarrow b\ ... & \} \end{array}$$

Inverses Semi-Thue-System

$S(G)=(\boldsymbol{V}, \boldsymbol{P}^{-1})$

$\boldsymbol{V}=\Sigma\cup\boldsymbol{N}$

$$\boldsymbol{P}^{-1} = \{ \begin{array}{ll} A+A & \rightarrow A, \\ A*A & \rightarrow A, \\ (A) & \rightarrow A, \\ B & \rightarrow A, \\ a & \rightarrow B, \\ b & \rightarrow B...\ \} \end{array}$$

Syntaktische Analyse (4/4) Beispiel (2/2)

Analyse des Satzes „(a+b)∗c+d)“

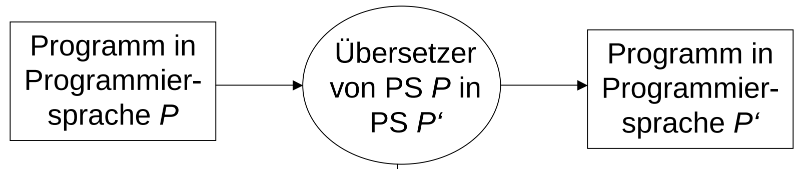
$(a+b)*c+d \Rightarrow (a+b)*c+B$
$\Rightarrow (a+b)*B+B \Rightarrow (a+B)*B+B$
$\Rightarrow (B+B)*B+B \Rightarrow (B+B)*B+A$
$\Rightarrow (B+B)*A+A \Rightarrow (B+A)*A+A$
$\Rightarrow (A+A)*A+A \Rightarrow (A)*A+A$
$\Rightarrow A*A+A$
$\Rightarrow A+A$
$\Rightarrow A = Z$, d.h. „(a+b)∗c+d)“ ist ein syntaktisch korrekter arithmetischer Ausdruck

$$\boldsymbol{P}^{-1} = \{ \begin{array}{ll} A+A & \rightarrow A, \\ A*A & \rightarrow A, \\ (A) & \rightarrow A, \\ B & \rightarrow A, \\ a & \rightarrow B, \\ b & \rightarrow B...\ \} \end{array}$$

Analyse des Satzes „a+∗b“

$a+*b \Rightarrow a+*B \Rightarrow B+*B \Rightarrow B+*A$
$\Rightarrow A+*A$???
d.h. „(a+∗b) c+d)“ ist ein kein syntaktisch korrekter Ausdruck

Semantische Analyse



Anmerkungen:
- Grob: Semantische Analyse übersetzt Programmelemente der Programmiersprache *P* in *semantisch gleichbedeutende* Programmelemente der Programmiersprache *P‘*
- *Nicht Gegenstand dieser Vorlesung (>Compilerbau)*

1. Teilschritt (Phase): Syntaktische Analyse von Programm *P*
2. Teilschritt (Phase): *Semantische Analyse*