Informatik IV
Compilerbau

Prof. Dr. R. Laue

Wintersemester 1991/92

Erstellt von Helmut Hahn und Walter Dörwald
Grafiken von Helmut Hahn und Walter Dörwald

Inhalt:

1. Übersicht

Es werden die folgenden Themen behandelt

- Was kann ein Rechner?
- Wie versteht ein Rechner die Programme? (Compilerbau)
- Fehlererkennung
- Welche Sprache kann ein Rechner verstehen?
- Was kann ein Rechner berechnen?
- Ausblicke (Expertensysteme)

Top-Down-Strategie

- Abstraktion des Problems; Grobe Sicht
- Schrittweise werden die in der Beschreibung enthaltenen Punkte genauer umfasst.

Vorteile

- Lokal ist stets nur ein überschaubar kleines Problem zu betrachten.
- Stets ist bekannt, welche Rolle der derzeit betrachtete Punkt global spielt.

Formale Hilfsmittel: Struktogramme

Name des Struktogramms Eingabe	
	Strukturblock
Ausgabe	

Ein Strukturblock ist entweder elementar — das heißt er enthält eine Anweisung oder den Namen eines Struktogramms — oder er ist zusammengesetzt

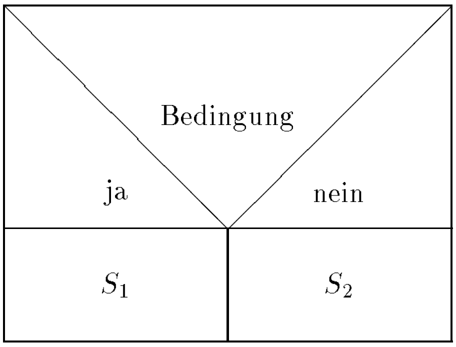
(a) Sequenz von Strukturblöcken

S_1
S_n

Jedes S_i ist ein Strukturblock.

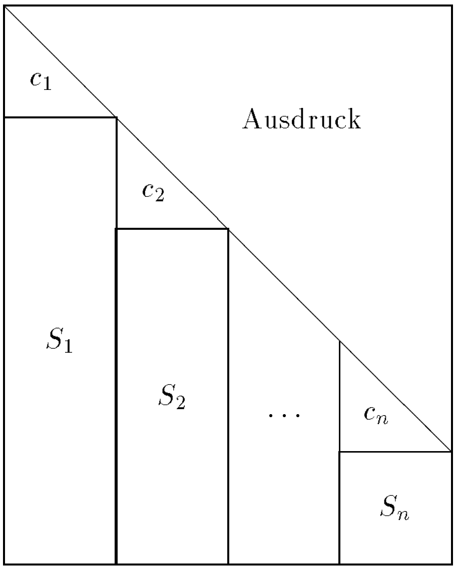
(b) Auswahl von Strukturblöcken

(1) Zweiseitige Auswahl



Dabei sind S_1 und S_2 Strukturblöcke.

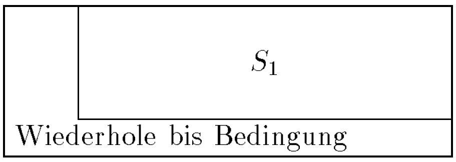
(2) Mehrseitige Auswahl



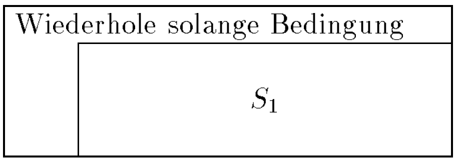
Dabei sind $c_1, \ldots, c_n$ Konstanten und $S_1, \ldots, S_n$ Strukturblöcke.

(c) Wiederholung

(1) Nicht abweisende Schleife



(2) Abweisende Schleife



Das Übersetzen eines Hochsprachenprogramms in Maschinensprache stellt sich auf der obersten Ebene somit folgendermaßen dar

Übersetzen Eingabeprogramm	
	Compilieren
Ausgabeprogramm	

Dabei sieht das Struktogramm `Compilieren` folgendermaßen aus

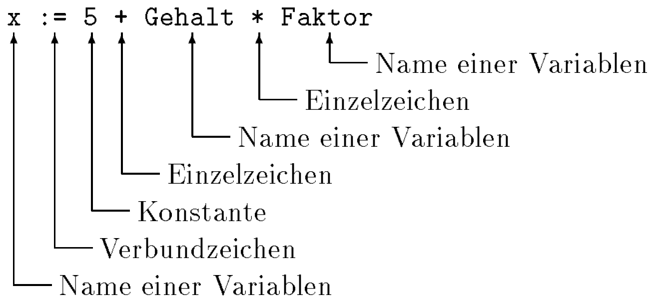
Compilieren Eingabeprogramm		
	Lexikale Analyse	maschinenunabhängig
	Syntax-Analyse	
	Semantische Analyse	
	Zwischencodeerzeugung	
	Codeoptimierung	maschinenabhängig
	Codeerzeugung	
Ausgabeprogramm		

(1) lexikale Analyse

Lexikale Analyse Eingabeprogramm	
	Beseitigung von überflüssigen Zeichen (Kommentare, White Space)
	Erkennung von Symbolen
	Codierung der Symbole
Symbole (Token)	

Symbole sind

- Schlüsselwörter (`if`, `else`, `for`, `while`, ...)
- Namen ($\neq$ Schlüsselwörtern) für Konstanten, Variablen, Prozeduren, Typen
- Zeichenketten, zum Beispiel auszugebende Wörter (`"Otto geht nach Hause\n"`)
- Einzelzeichen (`{`, `}`, `(`, `)`, ...)
- Verbundzeichen (`==`, `++`, `-=`, ...)

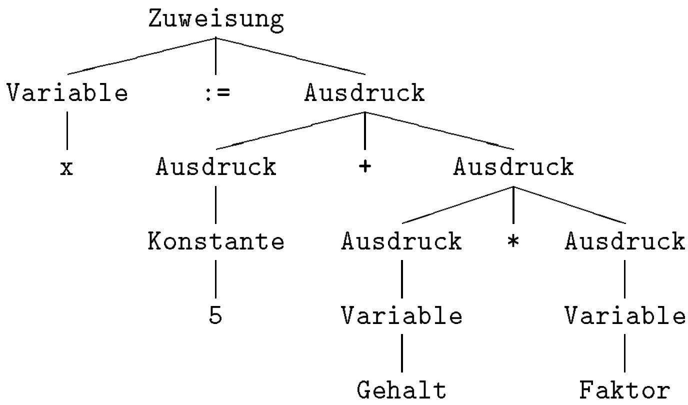


(2) Syntax-Analyse

Syntax-Analyse
Symbole
Zusammenfassen von Symbolen zu grammatikalischen Einheiten
Aufbau von Syntaxbäumen
Syntaxbaum

Eine Grammatik beschreibt ein Sprache durch Regeln, damit kann das Wort anhand von Regeln erkannt werden

`x := 5 + Gehalt * Faktor`



(3) Semantische Analyse

Semantische Analyse Syntax-Baum
Festlegung des Gütigkeitsbereichs von Namen
Prüfung, ob verwendete Namen vorher vereinbart sind
Prüfung, ob Operanden bei Verknüpfungen von ihrem Typ her zulässig sind
Fehlermeldungen Typinformationen Syntax-Bäume

(4) Zwischencode-Erzeugung

Zwischencode-Erstellung Syntax-Baum Symbole und Typinformationen
Erstellung eines Programms für eine abstrakte Maschine
Folge von Instruktionen für die abstrakte Maschine

(5) Codeoptimierung

Die Codeoptimierung nutzt die speziellen Befehlssätze der einzelnen Maschine, um die Laufzeit und den Speicherbedarf des Programms kleinzuhalten.

(6) Codeerzeugung

Die Codeerzeugung muß eine aktuelle Zuweisung von Speicherplätzen vornehmen und den entgültigen Code erzeugen.

2. Mustererkennung mit Automaten

Die zu besprechende Technik wird insbesondere bei der lexikalischen Analyse verwendet. Gegeben sei eine lange Zeichenkette w und eine kurze Zeichenkette u, dann ist die Frage, ob u in w als Teilwort vorkommt, und — falls ja — wo. Ist

$$u = u_1 u_2 \dots u_k,$$

so ist also zu testen, ob

$$z = z_1 z_2 z_3 \dots z_l \stackrel{?}{=} z_1 z_2 \dots u_1 u_2 \dots u_k \dots z_l$$

ist.

Die Aufgabe soll sein, das Wort "Auto" in einem Text zu finden

A u t a r k A u t o

z_1 $\quad$ z_2 keine Übereinstimmung $\Longrightarrow$ Zeiger z_1 weiterschieben und mit z_2 neu starten

A u t o

z_1 markiert den jeweiligen Startplatz für den laufenden Vergleich. z_2 markiert die zu vergleichende Position. Der Aufwand ist damit in der Größenordnung des Produkts der beiden Wortlängen. Um dieses Verhalten zu verbessern, kann man nach folgender Strategie vorgehen: man vergißt z_1, merkt sich aber, was bisher durchlaufen war. Dieses Merken erfolgt in Zuständen. Ist $x_{j+1} = a$, so gehe in den Zustand "$x_1 \dots x_{j+1}$ erkannt" über, andernfalls muß das größte passende Anfangsstück im bereits gelesenen Text beim zu suchenden Wort ebenfalls als Teilwort auftreten. Für das zu suchenden Wort kann dieses Wiederfinden unabhängig vom Vergleichswort zu einem früheren Zeitpunkt geschehen.

Man merkt sich alle Anfangsstücke des zu suchenden Wortes, und zu jedem Eingabezeichen und Anfangsstück bestimme man das längste Endstück des Anfangsstücks im Text, das mit dem jeweiligen Eingabezeichen weitergeht.

2.1 Definition

Ein *endlicher Automat* A ist ein 5-Tupel $A = (S, \Sigma, s_0, F, \delta)$, wobei die endliche Menge S die *Zustandsmenge* und die ebenfalls endliche Menge Σ das sogenannte *Alphabet* sind. $s_0 \in S$ ist der *Startzustand* und $F \subseteq S$ die Menge der *Endzustände*. Weiter ist $\delta\colon S \times \Sigma \to S$ die *Übergangsfunktion*.

Mit Σ^* bezeichnen wir die Wörter, die sich mit den Zeichen aus Σ bilden lassen, das heißt

$$\Sigma^* := \Big\{ z_1 \dots z_n \;\Big|\; n \in I\!N^* \text{ und } z_i \in \Sigma \text{ für } i = 1, \dots, n \Big\} \cup \Big\{ \varepsilon \Big\}.$$

Dabei ist $\varepsilon \notin \Sigma$ das sogenannte *leere Wort*.

Eine Menge $L \subseteq \Sigma^*$ heißt *Sprache*, zum Beispiel sind $L = \emptyset$ und $L = \{\varepsilon\}$ Sprachen.

Der Automat verarbeitet ein Wort $w \in \Sigma^*$ wie folgt: Sei $s \in S$ und $w = x_1 \dots x_r$ oder $w = \varepsilon$, dann ist

$$\delta(s, w) := \begin{cases} s & \text{falls } w = \varepsilon \\ \delta\Big(\delta(s, x_1 \dots x_{r-1}), x_r\Big) & \text{sonst} \end{cases}$$

eine Fortsetzung von δ auf $S \times \Sigma^*$ und eas Wort $w \in \Sigma^*$ wird genau dann von A *akzeptiert*, wenn $\delta(s_0, w) \in F$ ist.

2.2 Definition

Sei $A = (S, \Sigma, s_0, F, \delta)$ ein Automat, dann heißt

$$L(A) := \{\, w \in \Sigma^* \mid A \text{ akzeptiert } w \,\}$$

die *Sprache von A*.

Beispiel

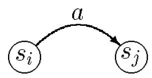
Wir wollen einen Automaten konstruieren, der erkennt, ob das Wort "erlernen" in einer langen Zeichenkette vorkommt, das heißt der Automat soll alle Wörter der Form

$$w\,\text{erlernen}\,u \quad \text{mit} \quad w, u \in \Sigma^*$$

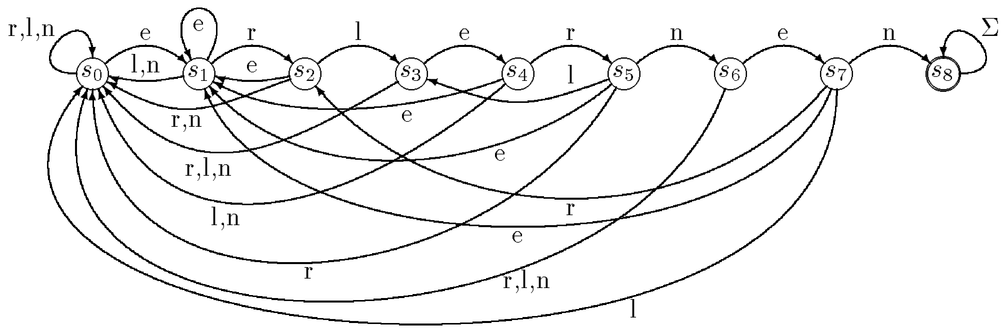
akzeptieren. Die Zustände des Automaten charakterisieren das bereits erkannte Anfangsstück des zu suchenden Wortes "erlernen". Als Alphabet haben wir

$$\Sigma = \{\mathrm{e}, \mathrm{r}, \mathrm{l}, \mathrm{n}\}.$$

Die Übergangsfunktion veranschaulichen wir durch einen Graphen. Dabei wird ein Pfeil vom Zustand s_i zum Zustand s_j gezeichnet und mit a beschriftet,



wenn $\delta(s_i, a) = s_j$ ist.



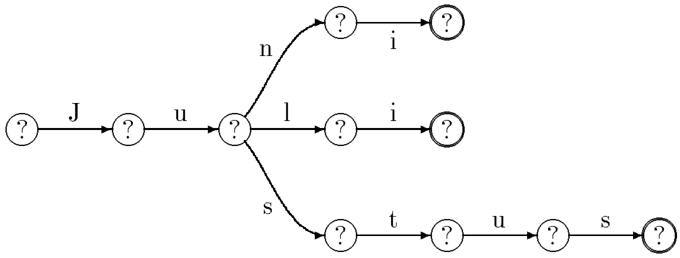
Schließlich haben wir $F = \{s_8\}$ als Menge der Endzustände.

In Tabellenform stellt sich das Übergangsfunktion δ folgendermaßen dar

$S \backslash \Sigma$	e	r	l	n
s_0	s_1	s_0	s_0	s_0
s_1	s_1	s_2	s_0	s_0
s_2	s_1	s_0	s_3	s_0
s_3	s_4	s_0	s_0	s_0
s_4	s_1	s_5	s_0	s_0
s_5	s_1	s_0	s_3	s_6
s_6	s_7	s_0	s_0	s_0
s_7	s_1	s_2	s_0	s_8
s_8	s_8	s_8	s_8	s_8

Ein endlicher Automat benötigt nur den Zeitaufwand zum einmaligen Durchlaufen des gegebenen Wortes, um ein Teilwort aufzudecken.

Man kann mit endlichen Automaten auch viele Wörter erkennen, man braucht nur entsprechend viele Endzustände einzuführen. Sind beispielsweise die Wörter `Juni`, `Juli` und `Justus` zu erkennen, so könnte ein Teil eines Übergangsfunktionsgraphen folgendermaßen aussehen:



Bei der lexikalen Analyse sind neben Schlüsselwörtern wie `begin`, `end` und `if` auch Klassen von Wörtern zu erkennen, so ist zum Beispiel `42` eine Zahl und `Jakob` oder `Meier1` sind Namen.

Bemerkung und Definition

Sind L, L_1 und L_2 Sprachen über dem Alphabet Σ, so sind auch

$$L_1 \cdot L_2 := \{ w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$$
$$L^2 := L \cdot L$$

und

$$L^* := \bigcup_{i \geq 0} L^i \quad \text{mit } L^0 := \{\varepsilon\}$$

Sprachen über dem Alphabet Σ. Für $L_1 \cdot L_2$ schreiben wir auch kürzer $L_1 L_2$.

2.3 Definition

Sei Σ ein (endliches) Alphabet, dann gilt

- $\emptyset$ ist ein regulärer Ausdruck mit der Sprache $\emptyset$.
- ε ist ein regulärer Ausdruck mit der Sprache $\{\varepsilon\}$.
- Für alle $a \in \Sigma$ ist a ein regulärer Ausdruck mit der Sprache $\{a\}$.

Sind α und β reguläre Ausdrücke, dann gilt

- $\alpha\beta$ ist ein regulärer Ausdruck mit der Sprache $L(\alpha)L(\beta)$.
- $(\alpha \mid \beta)$ ist ein regulärer Ausdruck mit der Sprache $L(\alpha) \cup L(\beta)$.
- (α) ist ein regulärer Ausdruck mit der Sprache $L(\alpha)$.
- α^* ist ein regulärer Ausdruck mit der Sprache $L(\alpha)^*$.

Die Bildung von regulären Ausdrucken ist ein Hilfsmittel um Klassen von Wörtern zu bechreiben. Beispielsweise erhalten wir für $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

- $L(\mathtt{(a|(b|c))}) = L(\mathtt{a}) \cup L(\mathtt{(b|c)}) = L(\mathtt{a}) \cup L(\mathtt{b}) \cup L(\mathtt{c}) = \{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
- $L(\mathtt{((a|b)|c)}) = \ldots = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, also können verschiedene reguläre Ausdrücke dieselbe Sprache haben.
- $L(\mathtt{((a|}\varepsilon\mathtt{)|a)}^*) = L(\mathtt{a}^*)$

Reguläre Ausdrücke können beispielsweise benutzt werden, um in Programmen vorkommende Tokens zu beschreiben. `Ziffer` ist

`0|1|2|3|4|5|6|7|8|9`,

Buchstabe ist

`A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|X|Y|Z`, dann ist `Zahl`

$$\texttt{Ziffer}\,\texttt{Ziffer}^*,$$

was wir im weiteren auch als $\texttt{Ziffer}^+$ abkürzen.

2.4 Definition

Seien $d_1, \ldots, d_n$ Namen und für alle i sei $d_i \notin \Sigma$ (wobei Σ wie üblich ein Alphabet ist). Weiter seien $r_1, \ldots, r_n$ reguläre Ausdrücke über $\Sigma \cup \{ d_1, \ldots, d_n \}$. Dann ist

$$d_1 \to r_1, d_2 \to r_2, \ldots, d_n \to r_n$$

eine reguläre Definition.

2.5 Satz

(a) Zu jedem endlichen Automaten A gibt es einen regulären Ausdruck α mit $L(A) = L(\alpha)$.

(b) Zu jedem regulären Ausdruck α gibt es einen endlichen Automaten A mit $L(\alpha) = L(A)$.

Beweis

(a) Sei $A = (S, \Sigma, s_0, F, \delta)$ gegeben. Ordne S an

$$S = (s_1, \ldots, s_n)$$

mit $s_1 < \ldots < s_n$. δ sei auf $S \times \Sigma^*$ fortgesetzt.

Wir beschreiben nun die Menge aller von A akzeptierten Wörter so, daß ein regulärer Ausdruck für diese Sprache angegeben werden kann.

Dazu definieren wir

$$E(i,j,k) := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \delta(s_i, w) = s_j \text{ und alle Zustände } s_l \text{, die bei der Berechnung} \\ \text{von } \delta(s_i, w) \text{ durchlaufen werden — wobei } s_j \text{ selbst nicht} \\ \text{berücksichtigt wird — erfüllen } s_l \leq s_k. \end{array} \right\}.$$

Damit können wir nun induktiv einen regulären Ausdruck aufbauen, dessen Sprache die Sprache von A ist.

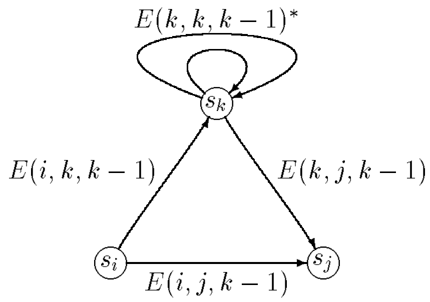
Es ist

$$E(i,j,0) = \left\{ a \in \Sigma \;\middle|\; \delta(s_i, a) = s_j \right\} =: \{a_1, \ldots, a_r\}$$

die Sprache des regulären Ausdrucks $\alpha(i,j,0) := (a_1 | \ldots | a_r)$.

Für $k > 0$ ist

$$E(i,j,k) = E(i,j,k-1) \cup E(i,k,k-1)E(k,k,k-1)^* E(k,j,k-1). \quad \text{(siehe Bild)}$$



Dann ist $E(i,j,k)$ die Sprache des Ausdrucks $\alpha(i,j,k)$ mit

$$\alpha(i,j,k) := \alpha(i,j,k-1) \,|\, (\alpha(i,k,k-1)\ \ \alpha(k,k,k-1)^*\ \ \alpha(k,j,k-1))).$$

Jedes $E(i,j,k)$ ist somit die Sprache eines regulären Ausdrucks. Falls $\varepsilon \in L(A)$ ist, so nehmen wir ε als regulären Ausdruck für dieses Wort.

Wir betrachten nun noch die Menge der von ε verschiedenen Wörter in $L(A)$. Es ist

$$L(A) \setminus \{\varepsilon\} = \bigcup_{s_j \in F} E(1,j,n)$$

die Sprache von

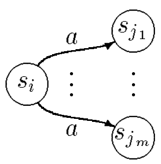
$$\Big|_{s_j \in F} \alpha(1,j,n).$$

(b) Um (b) zu beweisen ist noch einige Vorarbeit nötig.

2.6 Definition

Ein nichtdeterminierter endlicher ε-Automat ist ein 5-Tupel $A_\varepsilon = (S, \Sigma, s_0, s_f, R)$ mit

$$\begin{aligned} S &\text{ endliche Menge (Zustandsmenge)} \\ \Sigma &\text{ endliche Menge (Alphabet)} \\ s_0 \in S &\text{ Startzustand} \\ s_f \in S &\text{ Endzustand} \\ R \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S &\text{ Übergangsfunktion} \end{aligned}$$



$(s_i, a, s_{j_1}) \in R$

$\vdots$

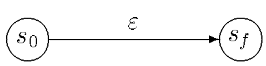
$(s_i, a, s_{j_m}) \in R$

$L(A_\varepsilon)$ ist die Menge der Wörter, bei deren Bearbeitung ein Weg möglich ist, der vom Start- zum Endzustand führt.

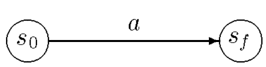
Wir können nun ε-Automaten für reguläre Sprachen definieren, das heißt Automaten, die in der Lage sind, reguläre Ausdrücke zu erkennen.

$\emptyset$ wird akzeptiert durch

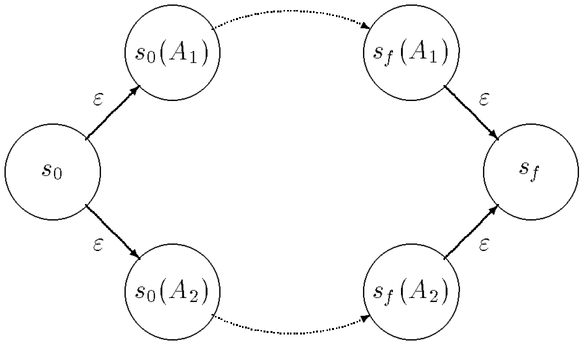
$\{\varepsilon\}$ wird akzeptiert durch



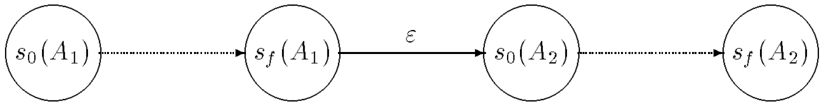
$\{a\}$ wird akzeptiert durch



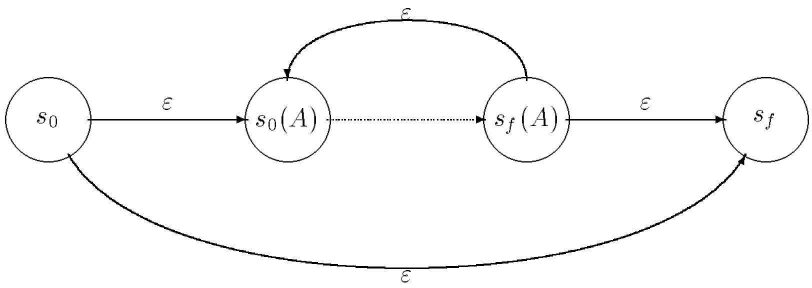
Seien zu den regulären Ausdrücken α_1 und α_2 bereits ε-Automaten $A_{1,\varepsilon}$ und $A_{2,\varepsilon}$ vorhanden mit $L(\alpha_i) = L(A_{i,\varepsilon})$ für $i = 1,2$, dann wird $L((\alpha_1 \,|\, \alpha_2))$ akzeptiert durch



$L(\alpha_1\alpha_2)$ wird akzeptiert durch



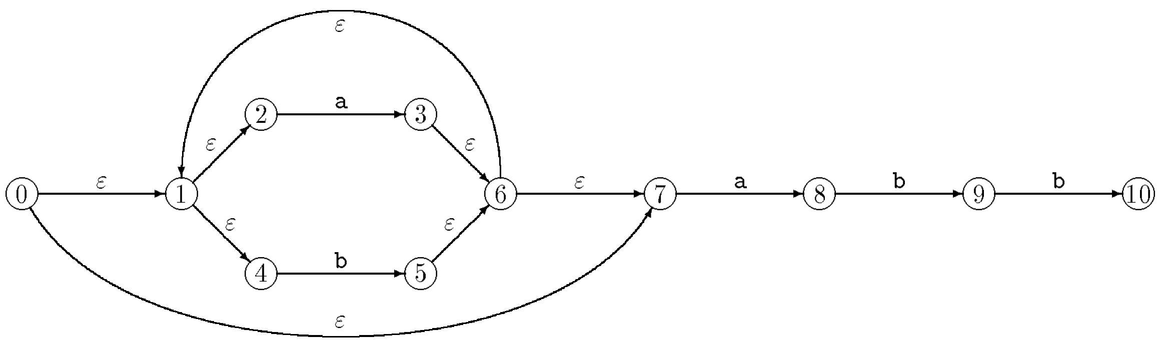
$L(\alpha^*)$ wird akzeptiert durch



Rekursiv läßt sich somit anhand des Aufbaus eines gegebenen regulären Ausdrucks α ein ε-Automat A_ε konstruieren, so daß $L(\alpha) = L(A_\varepsilon)$ ist.

Beispiel

Es sei $\Sigma = \{\texttt{a}, \texttt{b}\}$ und $\alpha = \texttt{(a|b)}^*\texttt{abb}$. Dann erhalten wir nach obiger Konstruktion den folgenden α akzeptierenden ε-Automaten.



Im folgenden wird ein endlicher Automat A konstruiert, dessen Sprache gerade die eines gegebenen ε-Automaten A_ε ist.

Teilmengenkonstruktion

Die Zustände von A entsprechen genau gewissen Teilmengen der Zustände von A_ε.

Der Startzustand von A beinhalte $s_0(A_\varepsilon)$ und alle s_i, die mit ε-Übergängen von $s_0(A_\varepsilon)$ aus erreichbar sind.

Ist T eine Teilmenge der Zustandsmenge von A_ε und $a \in \Sigma$, so bilden wir den a-Abschluß von T

$$U := \Big\{ s \in S(A_\varepsilon) \;\Big|\; \text{Es existiert ein } t \in T \text{ mit } (t, a, s) \in R \Big\}$$

und davon den ε-Abschluß

$$\Big\{ s \in S(A_\varepsilon) \;\Big|\; \text{Es existiert ein } u \in U \text{ mit } (u, \varepsilon, s) \in R \Big\}.$$

Dann sei

$$\delta(T, a) := \varepsilon\text{-Abschluß}\Big(a\text{-Abschluß}(T)\Big)$$

und

$$F := \Big\{ T \;\Big|\; T \subseteq S_{A_\varepsilon} \text{ und } s_f(A_\varepsilon) \in T \Big\}.$$

A hat dann dieselbe Sprache wie A_ε.

Bemerkung

Der zu einem gegebenen ε-Automaten A_ε äquivalente determinierte Automat A hat maximal $2^{|S_{A_\varepsilon}|}$ Zustände, da S_{A_ε} maximal $2^{|S_{A_\varepsilon}|}$ Teilmengen hat. Es werden jedoch in den seltensten Fällen soviele Zustände wirklich benötigt.

Beispiel (Teilmengenkonstruktion für den determinierten Automaten)

Der ε-Automat aus dem vorherigen Beispiel hatte 11 Zustände, das heißt wir haben 2048 Teilmengen. Wir verwenden zur Konstruktion des äquivalenten determinierten Automaten jedoch nur die tatsächlich benötigten Teilmengen (hier 5).

Der Startzustand A ist der ε-Abschluß von $\{0\}$

$$A = \{0, 1, 2, 4, 7\}.$$

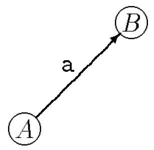
Bilde nun für jedes $a \in \Sigma$ die Menge der von A über a-Übergange erreichbaren Zustände und deren ε-Abschlüsse. Für $a := \mathtt{a}$ erhalten wir

$$\mathtt{a}\text{-Abschluß}(A) = \{3, 8\}$$

und schließlich

$$\varepsilon\text{-Abschluß}\Big(\mathtt{a}\text{-Abschluß}(A)\Big) = \{1, 2, 3, 4, 6, 7, 8\} =: B.$$

Dies ergibt den Teilgraphen

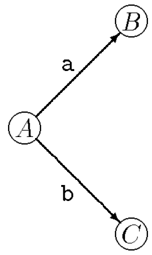


Für $a := \mathtt{b}$ erhalten wir

$$\mathtt{b}\text{-Abschluß}(A) = \{5\}$$

und

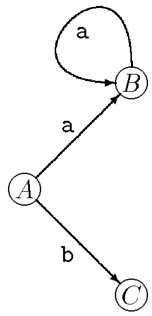
$$\varepsilon\text{-Abschluß}\Big(\mathtt{b}\text{-Abschluß}(A)\Big) = \{1, 2, 4, 5, 6, 7\} =: C.$$



Wende nun $\mathtt{a}$ auf B an

$$\mathtt{a}\text{-Abschluß}(B) = \{3, 8\}$$

Als ε-Abschluß erhalten wir also wieder B.

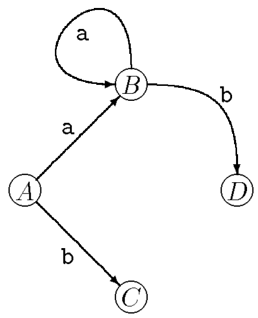


$\mathtt{b}$ auf B angewendet liefert

$$\mathtt{b}\text{-Abschluß}(B) = \{5, 9\}$$

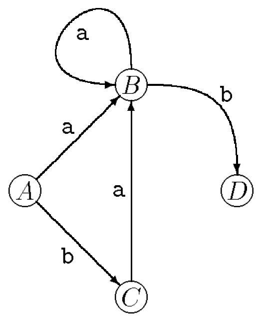
und daraus

$$\varepsilon\text{-Abschluß}\Big(\mathtt{b}\text{-Abschluß}(B)\Big) = \{1, 2, 4, 5, 6, 7, 9\} =: D.$$



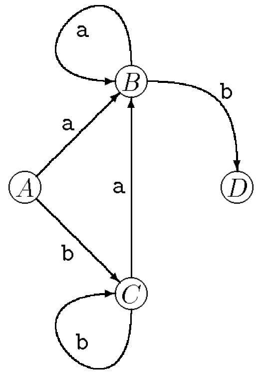
Aus C erhalten wir

$$\varepsilon\text{-Abschluß}\Big(\mathtt{a}\text{-Abschluß}(C)\Big) = \varepsilon\text{-Abschluß}(\{3, 8\}) = B$$



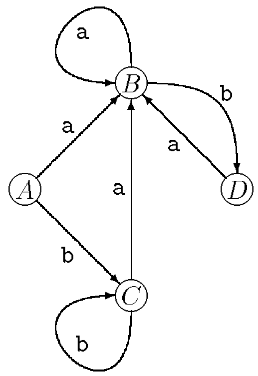
und

$$\varepsilon\text{-Abschluß}\Big(\mathtt{b}\text{-Abschluß}(C)\Big) = \varepsilon\text{-Abschluß}(\{5\}) = C.$$



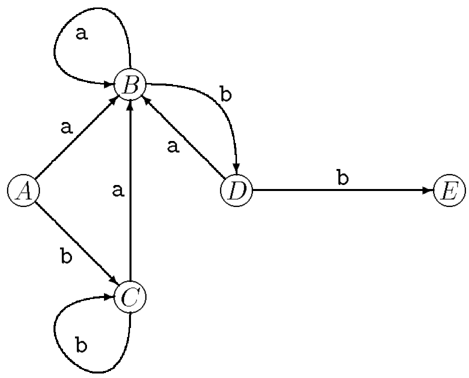
D liefert

$$\varepsilon\text{-Abschluß}\Big(\mathtt{a}\text{-Abschluß}(D)\Big) = \varepsilon\text{-Abschluß}(\{3,8\}) = B$$

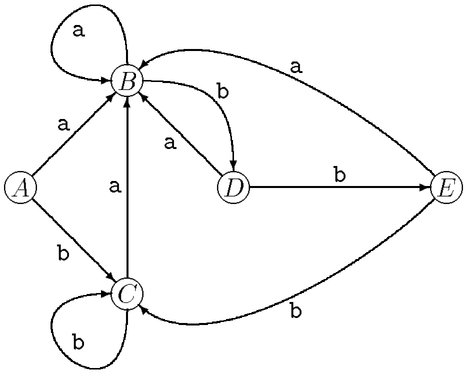


und

$$\varepsilon\text{-Abschluß}\Big(\mathtt{b}\text{-Abschluß}(D)\Big) = \varepsilon\text{-Abschluß}(\{5,10\} = (\{1,2,4,5,6,7,10\}) =: E.$$



Wenden wir $\mathtt{a}$ und $\mathtt{b}$ auf E an, so erhalten wir schließlich



mit E als akzeptierenden Zustand, da $10 \in E$ ist.

Bei größeren Alphabeten werden häufig zu viele Zustandsmengen erzeugt. Als Ausweg kann man für das vorgelegte Wort nur die für dessen Erkennung relevanten Zustände aus dem ε-Automaten jeweils erzeugen.

Implementation des a-Abschlusses und des ε-Abschlusses

- ε-Abschluss

  Es werden zwei Stacks benötigt für die abzuarbeitende Teilmenge von Zuständen des ε-Automaten und für die erzeugte Teilmenge (die Eingabe für den nächsten Aufruf).

  Um für einen Zustand s des ε-Automaten während des Aufbaus der neuen Teilmanege T schnell entscheiden zu können, ob $s \in T$ bereits gilt, verwaltet man T durch einen Bitvektor von der Länge der Anzahl der Zustände des ε-Automaten.

  Seien also S_1 und S_2 Stacks und T_2 ein Bitvektor

```
solange S_1 ≠ ∅ ist, tue
{
   pop t aus S_1
   ist t ∉ S_2 (über T_2 nachzuprüfen), so
      push t auf S_2 (notiere dies in T_2)
   für jeden Zustand u, der von t aus
      durch einen ε-Übergang erreichbar ist, tue
   {
      ist u ∉ S_2, so tue
      {
         push u auf S_2 (notiere dies in T_2)
         push u auf S_1
      }
   }
}
gebe S_2 aus
```

- a-Abschluss

  analog

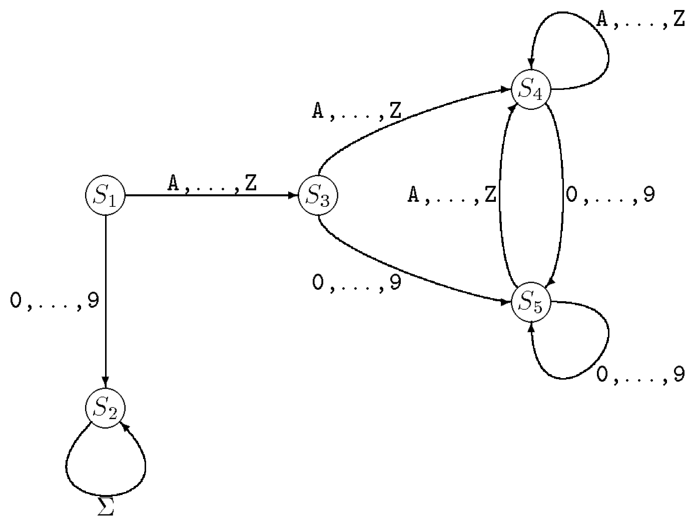
Für jeden betrachteten Zustand wird nur konstante Zeit benötigt.

Beispiel

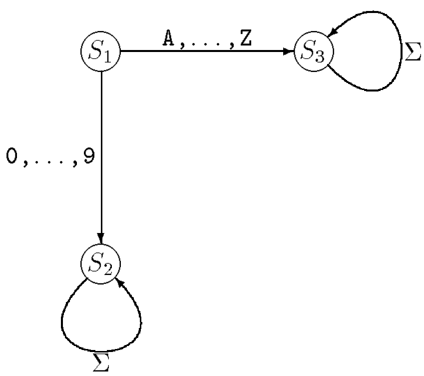
Sei die folgende reguläre Definition gegeben

$$\langle\texttt{identifier}\rangle ::= \langle\texttt{letter}\rangle\{\langle\texttt{letter\_or\_digit}\rangle\}$$
$$\langle\texttt{letter\_or\_digit}\rangle ::= \langle\texttt{letter}\rangle|\langle\texttt{digit}\rangle$$
$$\langle\texttt{letter}\rangle ::= \texttt{A}|\texttt{B}|\ldots|\texttt{Z}$$
$$\langle\texttt{digit}\rangle ::= \texttt{0}|\texttt{1}|\ldots|\texttt{9}$$

Dann erhalten wir den folgenden Automaten, der $\langle\texttt{identifier}\rangle$ akzeptiert



wobei $F = \{S_3, S_4, S_5\}$ die Menge der akzeptierenden Zustände ist. Da aber $\delta(S_i, a) \in F$ für alle $a \in \Sigma$ und $i = 3, 4, 5$ gilt, kann man diese drei Zustände zu einem zusammenfassen. Wir erhalten also den folgenden Übergangsgraph



mit $F = \{S_3\}$.

Definition

Sei $A = (S, \Sigma, s_0, F, \delta)$ ein determinierter Automat, dann heißen die Zustände s und s' *äquivalent* (in Zeichen $s \equiv s'$), wenn für alle $w \in \Sigma^*$ $\delta(s, w) \in F$ genau dann gilt, wenn $\delta(s', w) \in F$ gilt, das heißt s und s' sind bezüglich des Akzeptierens von Wörtern nicht unterscheidbar.

Äquivalente Zustände können identifiziert werden, ohne die Sprache des Automaten zu verändern.

Enthält ein Automat A keine Zustände s und s' mit $s \neq s'$ aber $s \equiv s'$, so heißt A *reduziert*.

Sei $A' = (S', \Sigma, s_0', F', \delta')$ ein weiterer Automat. Eine Abbildung $\varphi: S \to S'$ mit den folgenden Eigenschaften

$$\varphi(s_0) = s_0', \tag{1}$$

$$\varphi(\delta(s,a)) = \delta'(\varphi(s), a) \text{ für alle } s \in S \text{ und } a \in \Sigma, \tag{2}$$

$$s \in F \text{ genau dann, wenn } \varphi(s) \in F', \tag{3}$$

heißt Homomorphismus zwischen A und A'.

Ist $\varphi: A \to A'$ ein Homomorphismus, so ist $L(A) = L(A')$.

Bemerkung

Sei

$$K[s] := \{ s' \mid s \equiv s' \}$$

und

$$S' := \{ K[s] \mid s \in S \}.$$

sowie

$$\delta': S' \times \Sigma \to S'; \ \delta'(K[s], a) := K[\delta(s,a)].$$

Dann ist

$$\varphi: S \to S'; \ s \mapsto K[s]$$

ein wohldefinierter Homomorphismus zwischen A und $A' = (S', \Sigma, K[s_0], K[F], \delta')$.

"Zustandsverschmelzung" durch Aufspaltung

S wird ausgehend von einer groben Zerlegung solange verfeinert, bis sich die Zustandsmenge im Bezug auf die Akzeptanz von Wörtern nicht mehr aufspalten läßt.

```
Initialisiere K_1 := S \ F, K_2 := F, Π := {K_1, K_2}
Wiederhole (durchlaufe Π mit T)
{
   bilde die gröbsten Klassen von T, so
   daß für jedes a ∈ Σ die Elemente einer Klasse von a auf die
   Zustände in demselben Block T' ∈ Π abgebildet werden.
   Ersetze T durch die in T enthaltenen Klassen.
} bis sich Π nicht mehr ändert
```

Beispiel

In obigem Beipspiel zu ⟨`identifier`⟩ erhalten wir die Start-Zerlegung

$$K_1 = \{S_1, S_2\} \quad \text{und} \quad K_2 = \{S_3, S_4, S_5\}.$$

Wenden wir nun `0...9` beziehungsweise `A...Z` an, so erhalten wir

	Klasse	`0...9`	`A...Z`
S_1	1	1	2
S_2	1	1	1
S_3	2	2	2
S_4	2	2	2
S_5	2	2	2

S_1 und S_2 unterscheiden sich also in ihrem Verhalten bei Eingabe von A...Z, somit muß K_1 aufgespalten werden.

	Klasse	0...9	A...Z
S_1	11	12	2
S_2	12	12	12
S_3	2	2	2
S_4	2	2	2
S_5	2	2	2

Es ist hier keine weitere Aufspaltung nötig und wir erhalten die bereits bekannte Verschmelzung

$$\{S_1\},\ \{S_2\} \text{ und } \{S_3, S_4, S_5\}.$$

Gegeben seien die endlichen Automaten A_1 und A_2. Entscheide, ob $L(A_1) = L(A_2)$ ist. Zu testen ist, ob für alle $w \in \Sigma^*$

$$\delta_1(s_0(A_1), w) \in F_1$$

genau dann gilt, wenn

$$\delta_2(s_0(A_2), w) \in F_2$$

gilt.

Bilde alle Paare $(t_1, t_2) \in S_1 \times S_2$ (dabei ist S_i die Zustandsmenge von A_i für $i = 1, 2$). Verfolge vom Startzustand aus alle erreichbaren Wege, die bei einer der beiden Komponenten zu einem akzeptierten Zustand führen. Dazu verwenden wir als Datenstrukturen eine Warteschlange X für Paare $(t_1, t_2) \in S_1 \times S_2$ und einen Bitvektor B der Länge $|S_1 \times S_2|$.

```
Markiere alle Paare aus S1 × S2 als unbetrachtet (in B)
X := ∅
isomorph := TRUE
B[s0(A1), s0(A2)] := 1
Füge (s0(A1), s0(A2)) in X ein
Solange X ≠ ∅, tue
{
   nehme (t1, t2) aus X heraus.
   ist (t1 ∈ F1 und t2 ∉ F2) oder (t1 ∉ F1 und t2 ∈ F2), so
   {
      isomorph := FALSE
      break
   }
   sonst
   {
      durchlaufe Σ mit a
      {
         sei (t'1, t'2) := (δ1(t1, a), δ2(t2, a))
         ist (t'1, t'2) unbetrachtet, so
            markiere (t'1, t'2) als betrachtet (in B) und füge (t'1, t'2) in X ein.
      }
   }
}
return (isomorph);
```

Der Zeitaufwand für diesen Algorithmus beträgt $O(|S_1| \cdot |S_2| \cdot |\Sigma|)$.

Man kann auf dem Umweg über die endlichen Automaten entscheiden, ob zwei gegebene reguläre Sprachen gleich sind. Für reguläre Ausdrücke ist kein effizienter Entscheidungsalgorithmus bekannt. Die Teilmengenkonstruktion verhindert die Effizienz.

Es läßt sich zeigen, daß die Sprache

$$L = \{ \, (^n \mathtt{a})^n \mid n \in I\!N \, \}$$

durch keinen endlichen Automaten erkannt werden kann. Damit ist L keine reguläre Sprache. Dies bedeutet, daß es mit Hilfe der regulären Ausdrücke nicht möglich ist eine Sprache zu definieren, in der gefordert wird, daß gleich viele Klammern zu schließen sind, wie geöffnet werden.

3 Grammatiken

Die Lösung des Problems der Klammerregel erfolgt mit einer Grammatik

$$S \longrightarrow \mathtt{a}S\mathtt{b}$$
$$S \longrightarrow \varepsilon$$

Dabei ist $\Sigma_T = \{\mathtt{a}, \mathtt{b}\}$ und $\Sigma_N = \{S\}$. Man ersetzt vom Startsymbol S ausgehend Nichtterminalsymbole anhand von Regeln, bis ein Wort entsteht, das nur aus Terminalsymbolen besteht. Eine Ableitungsfolge obiger Grammatik sieht wie folgt aus

$$\begin{aligned} S &\longrightarrow \mathtt{a}S\mathtt{b} \\ &\Longrightarrow \mathtt{aa}S\mathtt{bb} \\ &\Longrightarrow \mathtt{aaa}S\mathtt{bbb} \\ &\vdots \\ &\Longrightarrow \mathtt{a}^i\mathtt{b}^i \end{aligned}$$

Dabei wurde zunächst i-mal ($i \in \{0, 1, 2, \ldots, n\}$) die Regel $S \longrightarrow \mathtt{a}S\mathtt{b}$ angewendet und im letzten Schritt einmal die Regel $S \longrightarrow \varepsilon$. Damit läßt sich also $\mathtt{a}^n\mathtt{b}^n$ für jedes $n \in I\!N$ produzieren.

3.1 Definition

Eine Grammatik ist ein Vier-Tupel

$$G = (V_N, V_T, R, w_0),$$

wobei

$$\begin{gathered} V_N \text{ eine endliche Menge, die Menge der Nichtterminalsymbole ist} \\ V_T \text{ eine endliche Menge, die Menge der Terminalsymbole ist} \\ V_N \cap V_T = \emptyset, \quad V = V_N \cup V_T \\ w_0 \in V^* \text{ das Startwort und} \\ R \subseteq V^* \times V^* \text{ die Menge der Regeln} \end{gathered}$$

ist. Die Sprache $L(G)$ von G ist

$$L(G) = \{\, v \in V_T^* \mid w_0 \overset{*}{\Longrightarrow} v \,\}.$$

Bezeichnung

Eine Regel $r \in R$ wobei $r = (w_1, w_2)$ wird auch als $w_1 \longrightarrow w_2$ geschrieben.

Regeln nennt man auch Produktionen.

Sei $w \in V^*$ und $r = w_1 \longrightarrow w_2$. Ist $w = uw_1v$, so können wir anhand von r in w den Wortteil w_1 durch w_2 ersetzen. Wir schreiben dann

$$w \Longrightarrow uw_2v.$$

Bei einer Folge von mehreren Ersetzungsprozessen

$$w \Longrightarrow w' \Longrightarrow w'' \Longrightarrow \ldots \Longrightarrow w^{(n)}$$

notieren wir für das Resultat auch

$$w \overset{*}{\Longrightarrow} w^{(n)}.$$

Dabei läßt $*$ auch zu, daß keine Regel angewandt wurde.

Die Grammatikregeln

$$S \longrightarrow \alpha_1,\ S \longrightarrow \alpha_2,\ \ldots,\ S \longrightarrow \alpha_n$$

schreiben wir auch als

$$S \longrightarrow \alpha_1|\alpha_2|\ldots|\alpha_n.$$

Satz

Der Ausdruck

$$(\mathtt{a}|\mathtt{b})^*\mathtt{abb}$$

hat eine Sprache, die auch durch eine Grammatik beschrieben werden kann.

Beweis

Wir geben die gesuchte Grammatik explizit an.

$$\begin{aligned}
A_0 &\longrightarrow \mathtt{a}A_0|\mathtt{b}A_0|\mathtt{a}A_1\\
A_1 &\longrightarrow \mathtt{b}A_2\\
A_2 &\longrightarrow \mathtt{b}A_3\\
A_3 &\longrightarrow \varepsilon
\end{aligned}$$

wobei $V_N = \{A_0, A_1, A_2, A_3\}$, $V_T = \{\mathtt{a}, \mathtt{b}\}$ und A_0 das Startwort ist. Diese Grammatik hat die gewünschte Sprache.

3.2 Definition

Sei $G = (V_N, V_T, R, w_0)$ eine Grammatik.

a) Ist das Startwort $w_0 \in V_N$ also ein einzelnes Symbol, und es gibt in R keine Regel der Form $\varepsilon \longrightarrow w$, so heißt G eine *Chomsky-Grammatik* oder *Typ-0-Grammatik*.

b) Eine Grammatik G heißt *kontextsensitiv* oder *Typ-1-Grammatik*, wenn G Typ-0-Grammatik ist und jede Regel aus R die Form

$$\alpha A\beta \longrightarrow \alpha\gamma\beta \quad \text{mit } \gamma \neq \varepsilon,\ A \in V_N \text{ und } \alpha, \beta, \gamma \in V^*$$

oder

$$A \longrightarrow \varepsilon$$

hat.

c) Eine Grammatik G heißt *kontextfrei* oder *Typ-2-Grammatik*, wenn G Typ-0-Grammatik ist und jede Regel aus R die Form

$$A \longrightarrow \gamma \quad \text{mit } A \in V_N \text{ und } \gamma \in V^*$$

hat. Damit ist c) ein Spezialfall von b).

d) Eine Grammatik G heißt *rechtslinear* oder *Typ-3-Grammatik*, wenn G Typ-0-Grammatik ist, und alle Regeln aus R die Form

$$\begin{aligned}
A &\longrightarrow \mathtt{a}B\\
A &\longrightarrow B\\
A &\longrightarrow \mathtt{a}\\
A &\longrightarrow \varepsilon
\end{aligned}$$

mit $A, B \in V_N$ und $\mathtt{a} \in V_T$ haben.

Ist $L = L(G)$ für ein G, und G eine Typ-i-Grammatik, so ist L eine *Typ-i-Sprache*. Mit $\mathcal{L}_i$ bezeichnen wir die Familie der Typ-i-Sprachen.

Satz

Es gilt

$$\mathcal{L}_0 \supsetneq \mathcal{L}_1 \supsetneq \mathcal{L}_2 \supsetneq \mathcal{L}_3.$$

Beweis

Wir zeigen später, daß $\mathcal{L}_3$ die Familie der regulären Sprachen ist und

$$L = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\} \in \mathcal{L}_2 \setminus \mathcal{L}_3.$$

Das Pumping Lemma wird beweisen, daß

$$L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\} \in \mathcal{L}_1 \setminus \mathcal{L}_2.$$

Beispiel einer Typ-1-Grammatik

Gegeben sei die Regelmenge R mit dem Startwert S

$$\begin{aligned}
S &\longrightarrow \mathtt{a}SBC|\varepsilon \\
\mathtt{a}B &\longrightarrow \mathtt{ab} \\
\mathtt{b}B &\longrightarrow \mathtt{bb} \\
\left.\begin{aligned} CB &\longrightarrow CX \\ CX &\longrightarrow BX \\ BX &\longrightarrow BC \end{aligned}\right\} & \begin{array}{l}\text{entspricht } CB \longrightarrow BC,\ CB \longrightarrow BC \text{ ist allerdings} \\ \text{keine Typ-1-Grammatikregel}\end{array} \\
C &\longrightarrow \mathtt{c}
\end{aligned}$$

i-maliges Anwenden der ersten Regel $S \longrightarrow \mathtt{a}SBC$ und anschlie endes Anwenden von $S \longrightarrow \varepsilon$ liefert

$$\underbrace{\mathtt{aa}\ldots\mathtt{a}}_{i}\underbrace{BCBC\ldots BC}_{i}.$$

Bevor nun $C \longrightarrow \mathtt{c}$ angewendet werden darf, muß jedes B links von allen C stehen, sonst blieben Nichtterminalsymbole B übrig, und man erhält

$$\underbrace{\mathtt{aa}\ldots\mathtt{a}}_{i}\underbrace{BB\ldots B}_{i}\underbrace{CC\ldots C}_{i}.$$

Nun wird i-mal die Regel $C \longrightarrow \mathtt{c}$ verwendet und es folgt

$$\underbrace{\mathtt{aa}\ldots\mathtt{a}}_{i}\underbrace{BB\ldots B}_{i}\underbrace{\mathtt{cc}\ldots\mathtt{c}}_{i}.$$

Schlie lich wird einmal $\mathtt{a}B \longrightarrow \mathtt{ab}$ verwendet

$$\underbrace{\mathtt{aa}\ldots\mathtt{a}}_{i}\mathtt{b}\underbrace{BB\ldots B}_{i-1}\underbrace{\mathtt{cc}\ldots\mathtt{c}}_{i}$$

und dann $(i-1)$-mal $\mathtt{b}B \longrightarrow \mathtt{bb}$, woraus

$$\mathtt{aa}\ldots\mathtt{abb}\ldots\mathtt{bcc}\ldots\mathtt{c} = \mathtt{a}^i\mathtt{b}^i\mathtt{c}^i$$

folgt. Also hat diese Grammatik

$$L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$$

als Sprache.

Schreibweise Backus-Naur-Form

In dieser Notationsweise schreibt man "::=" für "$\longrightarrow$" und "$\{\alpha\}$" für "α^*". Nichtterminalsymbole werden gekennzeichnet, indem sie von $\langle\ \rangle$ umschlossen werden. Für das einfache Oder-Zeichen "|" schreibt man weiterhin "|". Reservierte Wörter werden in Fettdruck oder alternativ in Kursivschrift dargestellt.

Beispiel

Wir betrachten die Sprache der Ausdrücke wobei $\langle\text{Ausdruck}\rangle$ ein Nichtterminalsymbol ist.

$$\begin{aligned}
\langle\text{Ausdruck}\rangle &::= \langle\text{Variable}\rangle|\langle\text{Konstante}\rangle|(\langle\text{Ausdruck}\rangle\ \langle\text{Operator}\rangle\ \langle\text{Ausdruck}\rangle)\\
\langle\text{Variable}\rangle &::= a|b\\
\langle\text{Konstante}\rangle &::= 0|1|2|\ldots|9\\
\langle\text{Operator}\rangle &::= +|-|*
\end{aligned}$$

Es handelt sich offenbar um eine kontextfreie Grammatik.

Wir wollen nun eine Linksableitung mithilfe dieser Regeln bilden, das heißt jeweils von links aus wird das erste Nichtterminalsymbol ersetzt. Dann ist eine mögliche Ableitungsfolge

$$\begin{aligned}
\langle\text{Ausdruck}\rangle &\Longrightarrow (\langle\text{Ausdruck}\rangle\langle\text{Operator}\rangle\langle\text{Ausdruck}\rangle)\\
&\Longrightarrow (\langle\text{Variable}\rangle\langle\text{Operator}\rangle\langle\text{Ausdruck}\rangle)\\
&\Longrightarrow (a\langle\text{Operator}\rangle\langle\text{Ausdruck}\rangle)\\
&\overset{*}{\Longrightarrow} (a+(\langle\text{Variable}\rangle\langle\text{Operator}\rangle\langle\text{Ausdruck}\rangle))\\
&\Longrightarrow (a+(b\langle\text{Operator}\rangle\langle\text{Ausdruck}\rangle))\\
&\Longrightarrow (a+(b*\langle\text{Ausdruck}\rangle))\\
&\Longrightarrow (a+(b*\langle\text{Variable}\rangle))\\
&\Longrightarrow (a+(b*a))
\end{aligned}$$

Damit haben wir eine Ableitungsfolge für $(a+(b*a))$ gefunden, das heißt $(a+(b*a))$ ist ein Wort der Sprache obiger Grammatik.

3.3 Satz

Die regulären Sprachen sind genau die Sprachen, die mit rechtslinearen Grammatiken erkannt werden können.

Beweis

Der Beweis erfolgt in zwei Teilen.

a) Sei $A = (S, \Sigma, S_0, F, \delta)$ ein endlicher Automat. Dann definieren wir zu A eine Grammatik G mit L(G)=L(A) und G ist rechtslinear.

Setze $V_N := S$, $V_T := \Sigma$ und Startwort $w_0 := S_0$.

$S_i \xrightarrow{a} S_j = \delta(S_i, a)$ liefert die Regel $S_i \longrightarrow aS_j$. Ist $S \in F$, so ist $S \longrightarrow \varepsilon$ eine Regel.

Die Grammatik G enthält nur die so aus A erzeugten Regeln, damit ist G rechtslinear.

Es ist
$$\varepsilon \in L(A) \Longleftrightarrow S_0 \in F \Longleftrightarrow S_0 \longrightarrow \varepsilon \in R \Longleftrightarrow \varepsilon \in L(G).$$

Wir können uns also darauf beschränken nur noch Wörter ungleich ε zu betrachten.

1. Behauptung

Jedes $w \neq \varepsilon$ mit $w \in L(A)$ erhält man durch Regeln aus G.

Sei dazu $w = a_1 a_2 \ldots a_r$ wobei jedes $a_i \in \Sigma$ ist. Damit gilt

$$S_r = \delta(\ldots\delta(\underbrace{\delta(S_0, a_1)}_{S_1}, a_2), \ldots, a_r) \in F,$$

da $w \in L(A)$.

Nach Konstruktion existieren die Regeln

$$\begin{aligned} S_0 &\longrightarrow a_1 S_1 \\ S_1 &\longrightarrow a_2 S_2 \\ &\vdots \end{aligned}$$

Das heißt, es läßt sich die Ableitungsfolge

$$\begin{aligned} S_0 &\Longrightarrow a_1 S_1 \\ &\Longrightarrow a_1 a_2 S_2 \\ &\vdots \\ &\Longrightarrow a_1 a_2 \ldots a_r S_r \qquad S_r \in F \Longrightarrow S_r \longrightarrow \varepsilon \in R \\ &\Longrightarrow a_1 a_2 \ldots a_r = w \end{aligned}$$

bilden, das heißt $w \in L(G)$.

2. Behauptung

Jedes $w \neq \varepsilon$ mit $w \in L(G)$ erfüllt $w \in L(A)$.

Sei dazu die Ableitungsfolge

$$\begin{aligned} S_0 &\Longrightarrow w_1 \\ &\Longrightarrow w_2 \\ &\vdots \\ &\Longrightarrow w_{r-1} \\ &\Longrightarrow w_r = w \end{aligned}$$

eine Ableitung anhand der Regeln aus G. Welche Wörter lassen sich dann mit G produzieren?

Jedes Nichtterminalsymbol tritt am Ende auf. Das heißt im letzten Schritt $w_{r-1} \Longrightarrow w$ muß ein Nichtterminalsymbol S_r an der letzten Position weggefallen sein, und zwar durch Anwenden einer Regel $S_r \longrightarrow \varepsilon$. Damit ist $S_r \in F$.

Somit sind Ableitungen der Form

$$\begin{aligned} S_0 &\Longrightarrow a_1 S_1 \\ &\Longrightarrow a_1 a_2 S_2 \\ &\vdots \\ &\Longrightarrow a_1 a_2 \ldots a_r S_r \\ &\Longrightarrow a_1 a_2 \ldots a_r = w \end{aligned}$$

mit G produzierbar, und $S_r = \delta(S_0, a_1a_2 \dots a_r)$. Das heißt aber gerade $a_1 \dots a_r \in L(A)$.

b) Zu jeder rechtslinearen Grammatik $G = (V_N, V_T, R, w_0)$ kann man einen ε-Automaten A_ε konstruieren, so daß $L(G) = L(A_\varepsilon)$ ist.

Setze $S := V_N \cup \{S_f\}$, wobei S_f der Finalzustand ist, $\Sigma := V_T$ und $S_0 := w_0$. Die Übergangsrelation δ_{A_ε} sei wie folgt definiert

$$
\begin{array}{llll}
\text{Für} & A \longrightarrow aB & \text{sei} & A \xrightarrow{a} B \in \delta_{A_\varepsilon} \\
\text{Für} & A \longrightarrow B & \text{sei} & A \xrightarrow{\varepsilon} B \in \delta_{A_\varepsilon} \\
\text{Für} & A \longrightarrow a & \text{sei} & A \xrightarrow{a} S_f \in \delta_{A_\varepsilon} \\
\text{Für} & A \longrightarrow \varepsilon & \text{sei} & A \xrightarrow{\varepsilon} S_f \in \delta_{A_\varepsilon}
\end{array}
$$

Nur solche Übergänge seien erlaubt. Dann ist

$$X \in L(G)$$

äquivalent zur

> Existenz eines Weges von $w_0 \in S$ zu S_f, so daß die Verkettung der Bezeichnungen der durchlaufenden Pfeile gerade X ergibt

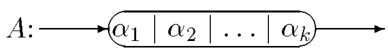
Das heißt $L(G) = L(A)$.

3.4 Definition

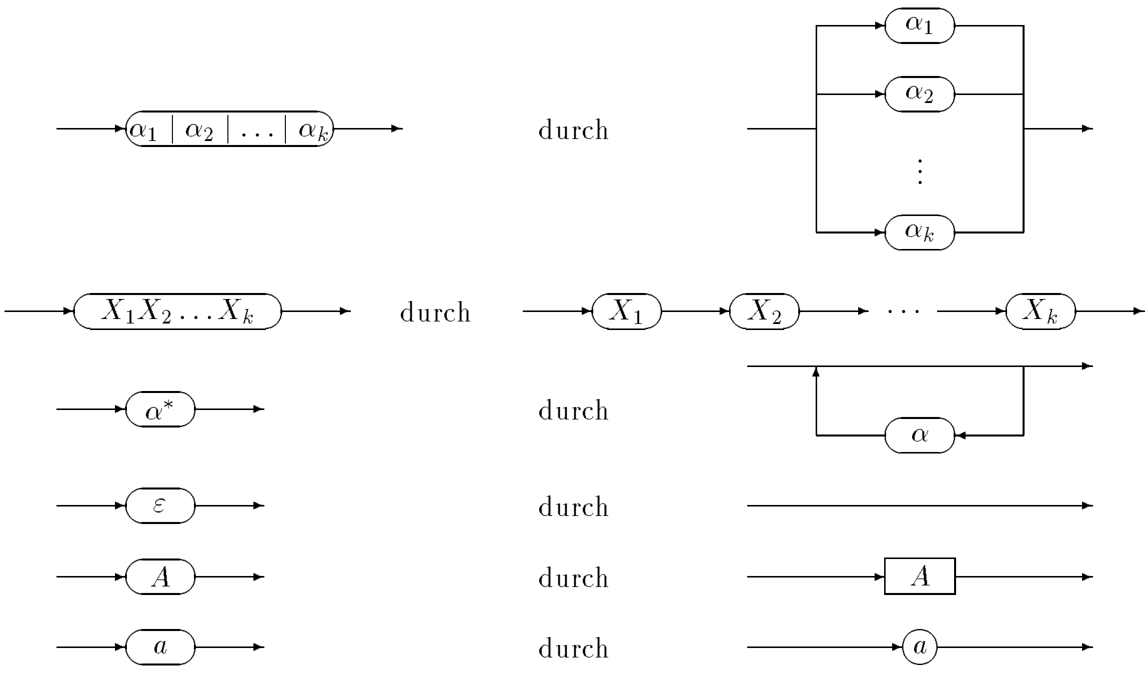
Sei G eine kontextfreie Grammatik. Dann ist zu G ein Syntaxgraph als gerichteter Graph rekursiv beschreibbar. Ist $r \in R$ eine Regel,

$$r = A \longrightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$$

so zeichnen wir



Desweiteren ersetzen wir



Dabei ist $A \in V_N$ und $a \in V_T$.

Beispiel

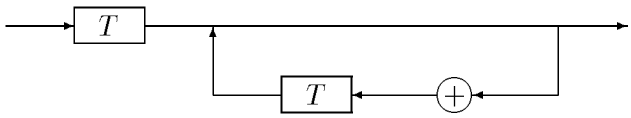
Gegeben sei $V_N = \{E, T, F\}$, $V_T = \{(, +, *, a,)\}$ und die Regelmenge R

$$\begin{aligned} E &\longrightarrow E + T | T \\ T &\longrightarrow T * F | F \\ F &\longrightarrow (E) | a \end{aligned}$$

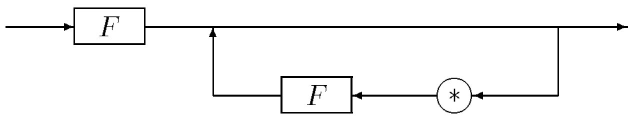
Die Backus-Naur-Form für diese Regelmenge lautet

$$\begin{aligned} E &::= T\{+T\} \\ T &::= F\{*F\} \\ F &::= (E) | a \end{aligned}$$

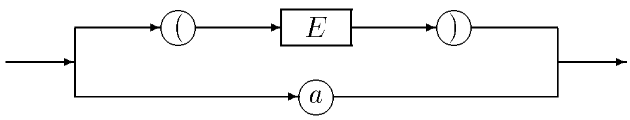
woraus sich die zugehörigen Ersetzungen für die Symbole E, T und F leichter ableiten lassen. Das Symbol E läßt sich demnach ersetzen durch



Das Symbol T durch



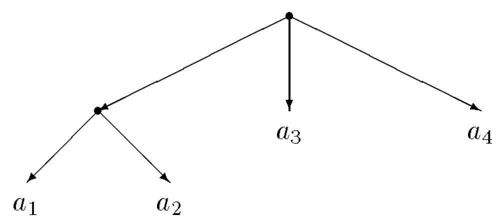
Und das Symbol F kann ersetzt werden durch



3.5 Syntaxbäume

Sei G eine kontextfreie Grammatik und $w \in L(G)$. Dann ist ein Syntaxbaum für w bezüglich G ein geordneter Wurzelbaum. Die Knoten sind mit Symbolen aus $V = V_N \cup V_T$ markiert. Die Wurzel (das Startwort) und innere Knoten haben Marken aus V_N, die Blätter Marken aus V_T. Hat ein Knoten v die Folge $(v_1, v_2, \ldots, v_n)$ als Folge der Nachfolgeknoten, so existiert in G eine Regel der Form $v \longrightarrow v_1 v_2 \ldots v_n$.

Werden die Blätter des Baumes von links nach rechts durchlaufen, so ergibt die Verkettung der Marken der Blätter das Wort w. Vergleiche dazu die folgende Abbildung, in der $w = a_1 a_2 a_3 a_4$ ist.



I Top-down-Strategie zum Aufbau eines Syntax-Baumes

Beginne beim Startwort und ersetze stets Nichtterminalsymbole durch die rechte Seite einer passenden Regel, bis der Syntaxbaum zum vorgelegten Wort erzeugt ist.

Beispiel

Gegeben sei die Regelmenge

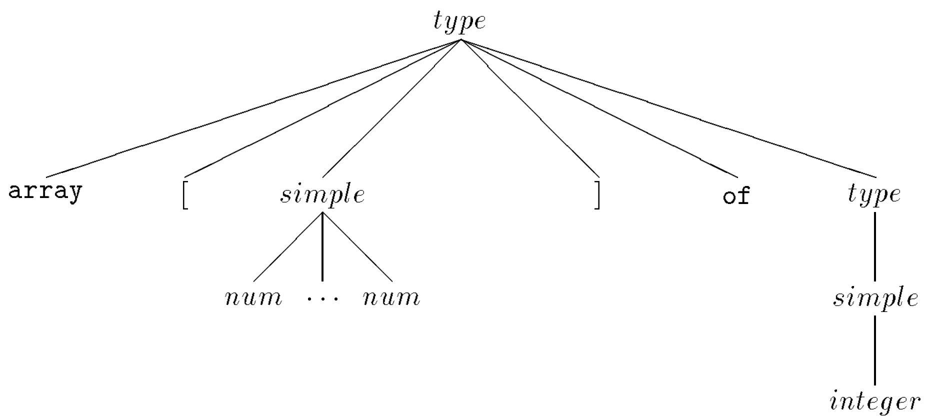
$$type \longrightarrow simple \,|\, *\, id \,|\, \texttt{array}[simple] \texttt{ of } type$$
$$simple \longrightarrow integer \,|\, char \,|\, num \,|\, num \ldots num$$

Startwort $w_0 = type$, $V_N = \{type, simple\}$ und $V_T = \{*, id, \texttt{array}, [, \texttt{of}, type,], char, num\}$.

Erkannt werden soll

$$\texttt{array}[num \ldots num] \texttt{ of } integer$$

Damit ergibt sich der zugehörige Syntaxbaum



II Bottom-up-Strategie

Betrachte nochmals die Grammatikregeln aus obigem Beispiel. Erkannt werden soll ebenfalls

$$\texttt{array}[num \ldots num] \texttt{ of } integer.$$

Bei der Bottom-up-Strategie ersetzt man Teilwörter, die als rechte Seite einer Regel auftreten, durch die linke Seite der Regel.

	$\texttt{array}[num \ldots num] \texttt{ of } integer$	verwende $simple \longrightarrow num \ldots num$
$\Longrightarrow$	$\texttt{array}[simple] \texttt{ of } integer$	verwende $simple \longrightarrow integer$
$\Longrightarrow$	$\texttt{array}[simple] \texttt{ of } simple$	verwende $type \longrightarrow simple$
$\Longrightarrow$	$\texttt{array}[type] \texttt{ of } simple$	
$\Longrightarrow$	$\texttt{array}[type] \texttt{ of } type$	Hier ist keine Fortsetzung möglich! Deshalb ist Backtracking nötig
$\Longrightarrow$	$\texttt{array}[simple] \texttt{ of } type$	verwende $type \longrightarrow \texttt{array}[simple] \texttt{ of } type$
$\Longrightarrow$	$type$	$type$ ist das Startsymbol, also ist man fertig

Bei dieser Strategie wird offensichtlich Backtracking erforderlich! Das heißt der Aufwand wächst exponentiell in der Schachtelungstiefe.

Ziel ist es deshalb die Grammatik so zu definieren, daß möglichst kein Backtracking erforderlich wird, beziehungsweise eine gegebene Grammatik entsprechend abzuändern.

Eindeutigkeit von Syntaxbäumen

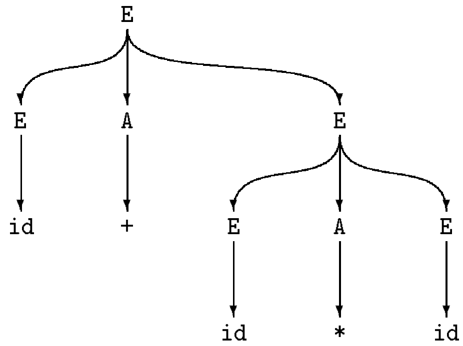
Beispiel 1

Wir betrachten folgende Grammatik

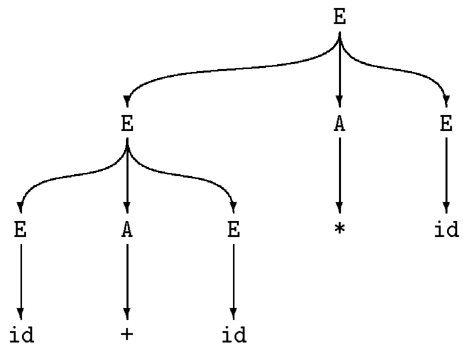
$$E \longrightarrow EAE|id$$
$$A \longrightarrow +|*$$

wobei $w_0 = E$, $V_N = \{E, A\}$ und $V_T = \{+, *, id\}$.

Zu $id + id * id$ soll in der Top-down-Strategie ein Syntaxbaum erstellt werden. Man erhält die beiden Syntaxbäume



und



Somit gibt es zwei verschiedene Syntaxbäume für dasselbe Wort! Mit dem ersten Syntaxbaum wird

$$id + (id * id)$$

berechnet. Mit dem zweiten Syntaxbaum wird

$$(id + id) * id$$

berechnet. Das heißt die Grammatik ist nicht eindeutig, es sind deshalb verschiedene Resultate für dasselbe Programm erlaubt, das mit einem Compiler zu dieser Grammatik erkannt wird.

Beispiel 2 ("dangling else")

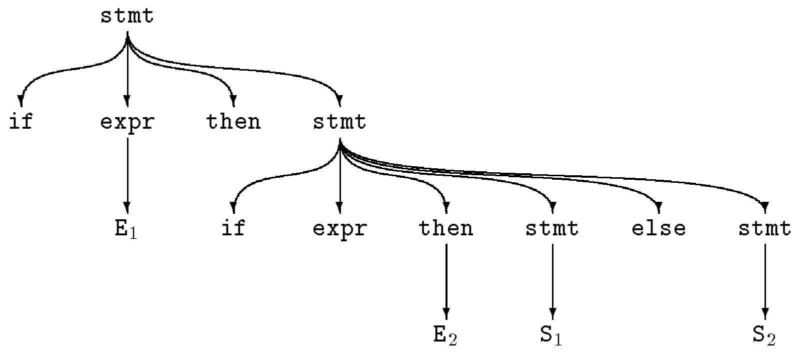
Gegeben sei die Grammatik

$$stmt \longrightarrow \texttt{if}\ expr\ \texttt{then}\ stmt|\texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt|S$$
$$expr \longrightarrow E$$

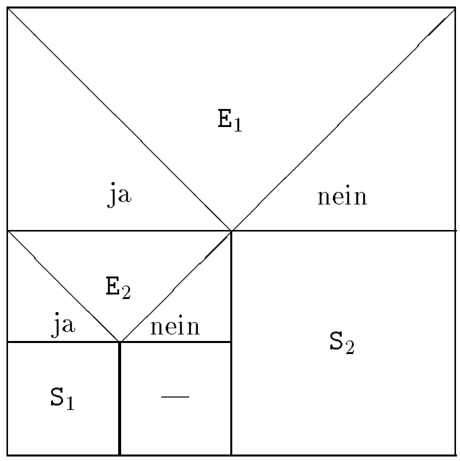
Erkannt werden soll

$$\texttt{if}\ E_1\ \texttt{then}\ \texttt{if}\ E_2\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$

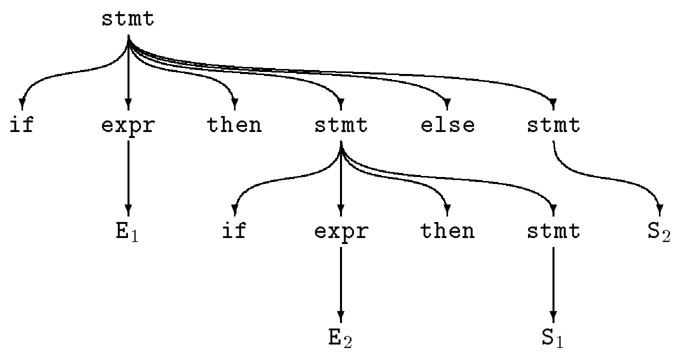
Es ergeben sich wieder zwei Syntaxbäume, zum einen



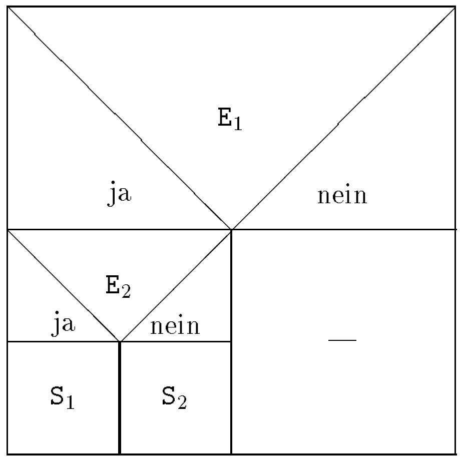
mit den entsprechenden Struktogramm



Und zum anderen der Syntaxbaum



mit den entsprechenden Struktogramm

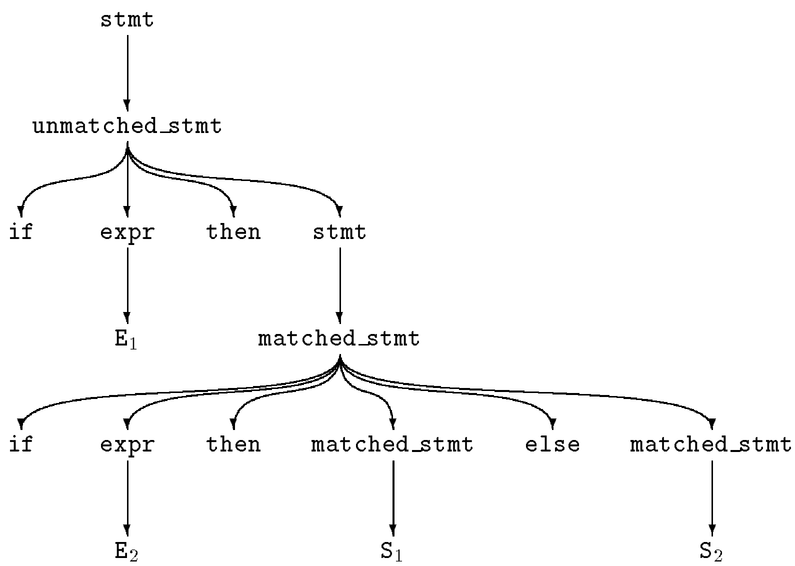


Das bedeutet natürlich, daß auch diese Grammatik nicht eindeutig ist.

Man löst dieses als "dangling else" bekannte Problem durch die Grammatik

$$\begin{aligned}
stmt &\longrightarrow matched_stmt \,|\, unmatched_stmt\\
matched_stmt &\longrightarrow S \,|\, \texttt{if}\ expr\ \texttt{then}\ matched_stmt\ \texttt{else}\ matched_stmt\\
unmatched_stmt &\longrightarrow \texttt{if}\ expr\ \texttt{then}\ stmt\\
unmatched_stmt &\longrightarrow \texttt{if}\ expr\ \texttt{then}\ matched_stmt\ \texttt{else}\ unmatched_stmt
\end{aligned}$$

Vor jedem `else` ist nur ein *matched_stmt* zugelassen ("`else` zu letztem offenen `if`"). Daraus resultiert als einziger Syntaxbaum:



Man kann Sprachen finden, zu denen es keine eindeutige Grammatik gibt. Beispielsweise

$$L = \{a^i b^j c^k \mid i = j \text{ oder } j = k\}.$$

Man kann keinen Algorithmus finden, der eindeutig machbare Grammatiken eindeutig macht.

Linksrekursion

Um die Top-down Konstruktion von Syntaxbäumen zu ermöglichen, müssen die Grammatiken vermeiden, daß beim Einsetzen von Regeln Endlosschleifen entstehen. Diese Problematik taucht in

$$A \stackrel{+}{\Longrightarrow} A\alpha$$

auf. Beim Ersetzen des ersten Nichtterminalsymbols in einem Wort kann bei Auftreten von A eine Endlosschleife entstehen.

Betrachte dazu die Grammatik

$$A \longrightarrow A\alpha|\beta$$

mit der Sprache

$$L = \{\beta\alpha, \beta\alpha\alpha, \ldots\} = \{\beta\alpha^*\}.$$

Da auf der rechten Seite der Regel A ganz links steht, bleibt der Wortanfang des erzeugten Wortes bis zum letzten Schritt unklar.

Die Lösung dieses Problems bietet die Grammatik

$$\begin{aligned} A &\longrightarrow \beta A' \\ A' &\longrightarrow \alpha A'|\varepsilon \end{aligned}$$

Diese Grammatik hat offensichtlich die gleiche Sprache L und den Vorteil, daß der Wortanfang β bereits nach Anwenden der ersten Regel bekannt ist.

Wir betrachten dazu drei konkrete Beispiele

Beispiel 1

Gegeben sei die Regelmenge

$$\begin{aligned} E &\longrightarrow E + T|T \\ T &\longrightarrow T * F|F \\ F &\longrightarrow (E)|id \end{aligned}$$

mit besagter Problematik. Dann löst die Regelmenge

$$\begin{aligned} E &\longrightarrow TE' \\ E' &\longrightarrow +TE'|\varepsilon \\ T &\longrightarrow FT' \\ T' &\longrightarrow *FT'|\varepsilon \\ F &\longrightarrow (E)|id \end{aligned}$$

das Problem und hat die gleiche Sprache.

Beispiel 2

Gegeben sei die Regel

$$A \longrightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\ldots|\beta_n$$

wobei für alle i $\alpha_i \neq \varepsilon$ ist und kein β_i mit A beginnt.

Man entfernt die Linksrekursion, indem man zu der Regelmenge

$$\begin{aligned} A &\longrightarrow \beta_1 A'|\beta_2 A'|\ldots|\beta_m A' \\ A' &\longrightarrow \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_m A'|\varepsilon \end{aligned}$$

mit der selben Sprache übergeht.

Beispiel 3 (indirekte Linksrekursion)

Gegeben sei die Regelmenge

$$\begin{aligned} S &\longrightarrow Aa|b \\ A &\longrightarrow Sd|Ac|\varepsilon \end{aligned}$$

mit indirekter Linksrekursion. Zunächst wird die indirekte Linksrekursion durch

$$\begin{aligned} S &\longrightarrow Aa|b \\ A &\longrightarrow Aad|bd|Ac|\varepsilon \end{aligned}$$

entfernt, und anschlie end die direkte Linksrekursion wie oben entfernt. Man erhält schlie lich

$$\begin{aligned} S &\longrightarrow Aa|b \\ A &\longrightarrow bdA'|\varepsilon A' \\ A' &\longrightarrow cA'|adA'|\varepsilon. \end{aligned}$$

3.6 Algorithmus zum Entfernen von Linksrekursion

Eingabe

Grammatik G ohne Zyklen ($A \overset{+}{\Longrightarrow} A$) und ohne ε-Produktionen ($A \longrightarrow \varepsilon$).

Ausgabe

Äquivalente Grammatik ohne Linksrekursion.

Methode

Man führt die folgenden beiden Schritte aus

1.Ordne V_N an, das heißt $V_N = (A_1, A_2, \ldots, A_n)$.
2.for $i = 1, \ldots, n$
  {
    for $j = 1, \ldots, i-1$
    {
      Ersetze jedes $A_i \longrightarrow A_j\gamma$ durch die Produktionen $A_i \longrightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$
      wobei $A_j \longrightarrow \delta_1|\delta_2|\ldots|\delta_k$ alle derzeitigen A_j-Produktionen sind.
    } // Die indirekten Linksrekursionen werden zu direkten
      // Linksrekursionen aufgelöst.
    Entferne alle (direkten) Linksrekursionen der Form $A_i \longrightarrow A_i\alpha$
  }

3.7 Linksfaktorisierung

Um Backtracking zu vermeiden, darf stets nur eine Regel anwendbar sein. Bei

$$A \longrightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$$

kann anhand des Anfangs von α_i erkannt werden, welche Regel anzuwenden ist. Die Regel

$$A \longrightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n$$

erlaubt dies nicht. Deshalb wird sie ersetzt durch

$$\begin{aligned} A &\longrightarrow \alpha A' \\ A' &\longrightarrow \beta_1|\beta_2|\ldots|\beta_n \end{aligned}$$

Allgemeine Methode

Suche zu jedem $A \in V_N$ und allen Regeln $A \longrightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$ das längste Anfangsstück der $\alpha_i =: \alpha$, welches mindestens zwei Regeln gemeinsam haben. Ersetze

$$A \longrightarrow \alpha\beta_1|\alpha\beta_2|\ldots\alpha\beta_r|\gamma_{r+1}|\ldots|\gamma_n$$

wobei α kein Anfangsstück von γ_i ist durch

$$\begin{aligned} A &\longrightarrow \alpha A'|\gamma_{r+1}|\ldots|\gamma_n \\ A' &\longrightarrow \beta_1|\beta_2|\ldots|\beta_r \end{aligned}$$

Wiederhole, bis kein Anfangsstück $\neq \varepsilon$ mehrfach vorkommt.

Top-down-Erstellung von Syntaxbäumen

Vorarbeiten

Die zu betrachtende Grammatik ist so abzuändern, daß keine Linksrekursion mehr auftritt. Weiterhin ist die Linksfaktorisierung durchzuführen, um in jeder Situation *eine* Regel eindeutig auswählen zu können.

Bei dem Syntaxbaum ist ein Knoten mit dem Symbol A zu betrachten und ein Zeichen a wird derzeit aus dem Eingabewort angezeigt. Dann ist zu entscheiden, ob a erkannt ist, oder eine Ersetzung von A durch die rechte Seite einer eindeutig festgelegten Regel vorzunehmen ist. Diese Entscheidung wird wie folgt realisiert.

Man hat einen Stack für die noch nicht endgültig abgeschlossene Historie. Die einzelnen "Entscheidungen" liegen tabelliert vor!

3.8 Top-down-Erstellung eines Syntaxbaumes

Erstelle einen Stack zur Aufnahme von Symbolen aus $V = V_N \cup V_T$, wobei $\$ \in V$. Anfangs lege \$ in den Stack, und darauf das Startsymbol. Die zu erkennende Zeichenfolge werde durch \$ abgeschlossen. Ein Zeiger p zeige auf eine Position der Zeichenfolge, anfangs zeige p auf das erste Zeichen.

```
Wiederhole
{
   Sei X das oberste Symbol im Stack, a das Zeichen, auf das p zeigt.
   Ist X ∈ V_T oder X = $
   {
      Ist X = a
         Entferne X aus dem Stack und rücke p um einen Platz vor.
      Sonst
         Gebe eine Fehlermeldung aus und breche ab.
   }
   Ist X ∈ V_N
   {
      Wähle die Regel X ⟶ Y_1...Y_k, die bei Vorliegen von X und
      Betrachtung von a anzuwenden ist, gebe X ⟶ Y_1...Y_k aus.
      Entferne X und lege Y_k...Y_1 in den Stack.
   }
} bis X = $.
```

Beispiel

Gegeben sei die linksrekursive Grammatik mit der Regelmenge R

$$\begin{aligned} E &\longrightarrow E+T|T \\ T &\longrightarrow T*F|F \\ F &\longrightarrow (E)|id \end{aligned}$$

Nach dem Entfernen der Linksrekursion lautet R

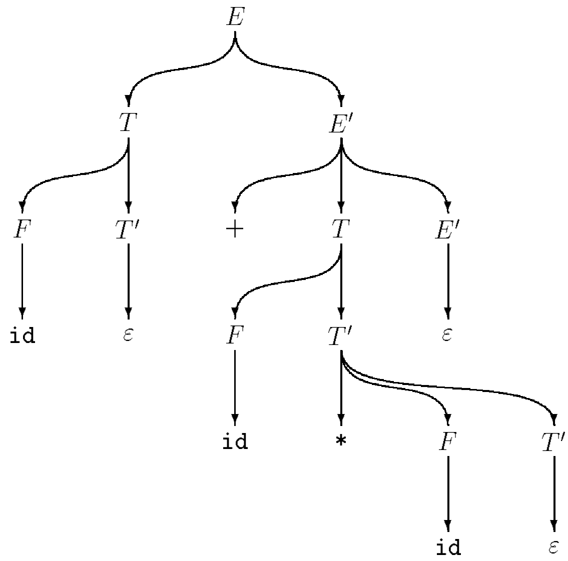
$$\begin{aligned} E &\longrightarrow TE' \\ E' &\longrightarrow +TE'|\varepsilon \\ T &\longrightarrow FT' \\ T' &\longrightarrow *FT'|\varepsilon \\ F &\longrightarrow (E)|id \end{aligned}$$

Dabei ist E das Startsymbol.

Der Ausdruck $w = id + id * id$ soll erkannt werden. Dann verdeutlicht die folgende Tabelle den obigen Algorithmus.

Stack	id	$+$	id	$*$	id	$\$$	Ausgabe
$\$E$	$\uparrow$						$E \longrightarrow TE'$
$\$E'T$	$\uparrow$						$T \longrightarrow FT'$
$\$E'T'F$	$\uparrow$						$F \longrightarrow id$
$\$E'T'id$	$\uparrow$						
$\$E'T'$		$\uparrow$					$T' \longrightarrow \varepsilon$
$\$E'$		$\uparrow$					$E' \longrightarrow +TE'$
$\$E'T+$		$\uparrow$					
$\$E'T$			$\uparrow$				$T \longrightarrow FT'$
$\$E'T'F$			$\uparrow$				$F \longrightarrow id$
$\$E'T'id$			$\uparrow$				
$\$E'T'$				$\uparrow$			$T' \longrightarrow *FT'$
$\$E'T'F*$				$\uparrow$			
$\$E'T'F$					$\uparrow$		$F \longrightarrow id$
$\$E'T'id$					$\uparrow$		
$\$E'T'$						$\uparrow$	$T' \longrightarrow \varepsilon$
$\$E'$						$\uparrow$	$E' \longrightarrow \varepsilon$
$\$$						$\uparrow$	

In der folgenden Abbildung ist der zugehörige Syntaxbaum dargestellt.



Aufbau einer Parsertafel

Die Entscheidungen werden in einer Top-down-Parsertafel notiert. Die Einträge entsprechen der partiellen Funktion M

$$M\colon V_N \times V_T' \dashrightarrow R.$$

Dabei ist $V_T' := V_T \cup \{\$\}$.

Für die obige Grammatik ergibt sich eine Parsertafel, die im folgenden erstellt werden soll. Falls in $V_N \times V_T$ kein Eintrag steht, dann wird eine Fehlermeldung ausgegeben, das heißt das zu erkennende Wort gehört nicht zu der Sprache der Grammatik.

Für den allgemeinen Aufbau einer Parsertafel werden zwei Hilfsfunktionen verwendet.

- $\texttt{First}(\alpha)$

  Ist $\alpha \in V^*$, so sei

  $$\texttt{First}(\alpha) := \{a \in V_T \cup \{\varepsilon\} \mid a \text{ ist Anfangssymbol eines } \beta \in V^* \text{ mit } \alpha \stackrel{*}{\Longrightarrow} \beta\},$$

  das heißt alle Terminalsymbole, die als erstes Zeichen von α aus erreichbar sind.

- $\texttt{Follow}(A)$

  Ist $A \in V_N$ und w_0 das Startsymbol der Grammatik, so sei

  $$\texttt{Follow}(A) := \{a \in V_T \cup \{\varepsilon\} \mid w_0 \stackrel{*}{\Longrightarrow} \alpha A a \beta\}$$

  das heißt das erste Terminalsymbol, das vom Startsymbol aus hinter A steht. $\texttt{Follow}(A) = \emptyset$ falls A nur rechts in einer Regel auftaucht oder A Startsymbol ist.

Algorithmus (zum Aufbau einer Top-down-Parsertafel)

```
Durchlaufe die Regelmenge R mit A ⟶ α
{
   Für alle a ∈ First(α) mit a ≠ ε setze
      M[A, a] = M[A, a] ∪ {A ⟶ α}
   Ist ε ∈ First(α)    // das heißt α =*=> ε
   {
      Durchlaufe Follow(A) mit b ∈ V_T und setze
         M[A, b] = M[A, b] ∪ {A ⟶ α}
      Ist $ ∈ Follow(A), so setze
         M[A, $] = M[A, $] ∪ {A ⟶ α}
   }
}
Alle unbesetzten Stellen werden mit Fehleraufrufen besetzt.
```

3.9 Definition

Hat die Parsertafel zur Grammatik G keine mehrfachen Einträge, so heißt G eine *LL(1)-Grammatik*.

Dabei heißt

L	von links nach rechts lesen
L	das am weitesten links liegende $A \in V_N$ wird zuerst ersetzt
1	ein Zeichen wird vorausgeschaut

Um den Aufbau der Parsertafel vornehmen zu können, brauchen wir `First`(α) für jedes $\alpha \in V^*$ und `Follow`(A) für jedes $A \in V_N$.

Algorithmus (zur Berechnung von `First`(α) für jedes $\alpha \in V^*$)

I Berechnung von $\texttt{First}(a)$ für alle $a \in V$

```
Iteriere bis kein First(a) mehr wächst
{
   Ist a ∈ V_T, so setze
      First(a) = {a}.
   Ist a ⟶ ε ∈ R, so setze
      First(a) = First(a) ∪ {ε}.
   Ist a ∈ V_N, so tue
   {
      Für alle a ⟶ y_1 ... y_k ∈ R und alle r mit ε ∈ ⋂_{j=1}^{r} First(y_j)
      // Beachte: für r = 0 liegt die leere Bedingung vor
      // das heißt die ersten r y_i können zu ε gemacht werden
      {
         Ist b ∈ First(y_{r+1}), so setze
            First(a) = First(a) ∪ {b}.
         Ist dabei r = k, so setze
            First(a) = First(a) ∪ {ε}.
      }
   }
}
```

II Berechnung von $\texttt{First}(\alpha)$ wobei $\alpha = x_1 \ldots x_n$

Sei $j := \max\limits_i \{ \text{ für alle } l \le i \text{ ist } \varepsilon \in \texttt{First}(x_l) \}$.

$$\texttt{First}(\alpha) = \bigcup_{i=1}^{j+1} \{ a \mid a \in \texttt{First}(x_i) \text{ und } a \neq \varepsilon \}.$$

Ist $j = n$, so setze
   $\texttt{First}(\alpha) = \texttt{First}(\alpha) \cup \{\varepsilon\}$.

Beispiel zu II

Sei $a = x_1 x_2$. Ist dann $\varepsilon \in \texttt{First}(x_1)$, dann betrachte $\texttt{First}(x_2)$ zur Berechnung von $\texttt{First}(a)$, sonst ist $\texttt{First}(x_1 x_2) = \texttt{First}(x_1)$.

Algorithmus (zur Berechnung von $\texttt{Follow}(A)$ für alle $A \in V_N$)

```
Setze Follow(A) = ∅ für alle A, und Follow(w_0) = {$}.
Iteriere bis kein Follow(A) mehr wächst
{
   Ist A ⟶ αBβ ∈ R, so setze
      Follow(B) = Follow(B) ∪ {a ∈ First(β)|a ≠ ε}.
   Ist A ⟶ αB ∈ R, so setze
      Follow(B) = Follow(B) ∪ Follow(A).
   Ist (A ⟶ αBβ ∈ R und ε ∈ First(β)), so setze ebenfalls
      Follow(B) = Follow(B) ∪ Follow(A).
}
```

Beispiel

Wir betrachten nochmals die Beispielgrammatik

$$
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE'|\varepsilon \\
T &\longrightarrow FT' \\
T' &\longrightarrow *FT'|\varepsilon \\
F &\longrightarrow (E)|id
\end{aligned}
$$

und wollen $\texttt{First}(x)$ für alle Symbole $x \in V^*$ und $\texttt{Follow}(x)$ für alle Symbole $x \in V_N$ berechnen.

Die Berechnung von "First"

Der erste Durchlauf der Regeln liefert

$E \longrightarrow TE'$	$\texttt{First}(T) = \emptyset$, das heißt $\texttt{First}(E)$ bleibt zunächst leer
$E' \longrightarrow \varepsilon$	$\texttt{First}(E') = \{\varepsilon\}$
$E' \longrightarrow +TE'$	$\texttt{First}(E') = \texttt{First}(E') \cup \texttt{First}(+) = \{\varepsilon, +\}$
$T \longrightarrow FT'$	$\texttt{First}(F) = \emptyset$, das heißt $\texttt{First}(T)$ bleibt zunächst leer
$T' \longrightarrow \varepsilon$	$\texttt{First}(T') = \{\varepsilon\}$
$T' \longrightarrow *FT'$	$\texttt{First}(T') = \texttt{First}(T') \cup \texttt{First}(*) = \{\varepsilon, *\}$
$F \longrightarrow (E)$	$\texttt{First}(F) = \{(\}$
$F \longrightarrow id$	$\texttt{First}(F) = \texttt{First}(F) \cup \texttt{First}(id) = \{(, id\}$

Da sich einige First's geändert haben, erfolgt ein zweiter Durchlauf aller Regeln

$E \longrightarrow TE'$	$\texttt{First}(T) = \emptyset$, das heißt $\texttt{First}(E)$ bleibt zunächst leer
$E' \longrightarrow \varepsilon$	keine Änderung von $\texttt{First}(E')$
$E' \longrightarrow +TE'$	keine Änderung von $\texttt{First}(E')$
$T \longrightarrow FT'$	$\texttt{First}(T) = \texttt{First}(T) \cup \texttt{First}(F) = \{(, id\}$
$T' \longrightarrow \varepsilon$	keine Änderung von $\texttt{First}(T')$
$T' \longrightarrow *FT'$	keine Änderung von $\texttt{First}(T')$
$F \longrightarrow (E)$	keine Änderung von $\texttt{First}(F)$
$F \longrightarrow id$	keine Änderung von $\texttt{First}(F)$

Da sich $\texttt{First}(T)$ geändert hat müssen alle Regeln nochmals durchlaufen werden

$E \longrightarrow TE'$	$\texttt{First}(E) = \texttt{First}(E) \cup \texttt{First}(T) = \{(, id\}$
$E' \longrightarrow \varepsilon$	keine Änderung von $\texttt{First}(E')$
$E' \longrightarrow +TE'$	keine Änderung von $\texttt{First}(E')$
$T \longrightarrow FT'$	keine Änderung von $\texttt{First}(T)$
$T' \longrightarrow \varepsilon$	keine Änderung von $\texttt{First}(T')$
$T' \longrightarrow *FT'$	keine Änderung von $\texttt{First}(T')$
$F \longrightarrow (E)$	keine Änderung von $\texttt{First}(F)$
$F \longrightarrow id$	keine Änderung von $\texttt{First}(F)$

Nun hat sich $\texttt{First}(E)$ geändert, daß heißt es müssen nochmals alle Regeln durchlaufen werden

$$\begin{array}{rcll}
E & \longrightarrow & TE' & \text{keine Änderung von } \texttt{First}(E) \\
E' & \longrightarrow & \varepsilon & \text{keine Änderung von } \texttt{First}(E') \\
E' & \longrightarrow & +TE' & \text{keine Änderung von } \texttt{First}(E') \\
T & \longrightarrow & FT' & \text{keine Änderung von } \texttt{First}(T) \\
T' & \longrightarrow & \varepsilon & \text{keine Änderung von } \texttt{First}(T') \\
T' & \longrightarrow & *FT' & \text{keine Änderung von } \texttt{First}(T') \\
F & \longrightarrow & (E) & \text{keine Änderung von } \texttt{First}(F) \\
F & \longrightarrow & id & \text{keine Änderung von } \texttt{First}(F)
\end{array}$$

Somit hat sich kein `First` mehr verändert, das heißt die Berechnung von `First` ist abgeschlossen und wir erhalten zusammenfassend

x	E	E'	T	T'	F	$+$	$*$	$($	$)$	id
`First`(x)	$(, id$	$\varepsilon, +$	$(, id$	$\varepsilon, *$	$(, id$	$+$	$*$	$($	$)$	id

Die Berechnung von "`Follow`"

Da E das Startsymbol ist wird zunächst $\texttt{Follow}(E) = \{\$\}$ gesetzt, alle anderen `Follow` sind leer. Der erste Durchlauf der Regeln liefert

$$\begin{array}{ll}
E \longrightarrow TE' & \left\{\begin{array}{l} \texttt{Follow}(T) = \texttt{Follow}(T) \cup \{a \in \texttt{First}(E') \mid a \neq \varepsilon\} = \{+\} \\ \texttt{Follow}(T) = \texttt{Follow}(T) \cup \texttt{Follow}(E) = \{\$, +\} \\ \texttt{Follow}(E') = \texttt{Follow}(E') \cup \texttt{Follow}(E) = \{\$\} \end{array}\right\} \\
E' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
E' \longrightarrow +TE' & \left\{\begin{array}{l} \text{keine Änderung von } \texttt{Follow}(T) \\ \text{keine Änderung von } \texttt{Follow}(E') \end{array}\right\} \\
T \longrightarrow FT' & \left\{\begin{array}{l} \texttt{Follow}(F) = \texttt{Follow}(F) \cup \{a \in \texttt{First}(T') \mid a \neq \varepsilon\} = \{*\} \\ \texttt{Follow}(F) = \texttt{Follow}(F) \cup \texttt{Follow}(T) = \{\$, +, *\} \\ \texttt{Follow}(T') = \texttt{Follow}(T') \cup \texttt{Follow}(T) = \{\$, +\} \end{array}\right\} \\
T' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
T' \longrightarrow *FT' & \left\{\begin{array}{l} \text{keine Änderung von } \texttt{Follow}(F) \\ \text{keine Änderung von } \texttt{Follow}(T') \end{array}\right\} \\
F \longrightarrow (E) & \texttt{Follow}(E) = \texttt{Follow}(E) \cup \{a \in \texttt{First}()) \mid a \neq \varepsilon\} = \{\$,)\} \\
F \longrightarrow id & \text{keine } \texttt{Follow}\text{-Berechnung möglich}
\end{array}$$

Da sich einige $\texttt{Follow}$'s geändert haben, erfolgt ein zweiter Durchlauf aller Regeln

$$
\begin{array}{ll}
E \longrightarrow TE' & \left\{\begin{array}{l}\text{keine Änderung von } \texttt{Follow}(T) \\ \text{keine Änderung von } \texttt{Follow}(E')\end{array}\right\} \\
E' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
E' \longrightarrow +TE' & \left\{\begin{array}{l}\texttt{Follow}(T) = \texttt{Follow}(T) \cup \{a \in \texttt{First}(E') \mid a \neq \varepsilon\} = \{\$,+,)\} \\ \texttt{Follow}(T) = \texttt{Follow}(T) \cup \texttt{Follow}(E') = \{\$,+,)\} \\ \text{keine Änderung von } \texttt{Follow}(E')\end{array}\right\} \\
T \longrightarrow FT' & \left\{\begin{array}{l}\text{keine Änderung von}\texttt{Follow}(F) \\ \text{keine Änderung von}\texttt{Follow}(T')\end{array}\right\} \\
T' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
T' \longrightarrow *FT' & \left\{\begin{array}{l}\texttt{Follow}(F) = \texttt{Follow}(F) \cup \{a \in \texttt{First}(T') \mid a \neq \varepsilon\} = \{\$,+,*,)\} \\ \texttt{Follow}(F) = \texttt{Follow}(F) \cup \texttt{Follow}(T') = \{\$,+,*,)\} \\ \texttt{Follow}(T') = \texttt{Follow}(T') \cup \texttt{Follow}(T') = \{\$,+,)\}\end{array}\right\} \\
F \longrightarrow (E) & \text{keine Änderung von } \texttt{Follow}(E) \\
F \longrightarrow id & \text{keine } \texttt{Follow}\text{-Berechnung möglich}
\end{array}
$$

Wieder haben sich einige $\texttt{Follow}$'s geändert, das heißt es ist ein weiterer Durchlauf durch alle Regeln notwendig.

$$
\begin{array}{ll}
E \longrightarrow TE' & \left\{\begin{array}{l}\text{keine Änderung von } \texttt{Follow}(T) \\ \text{keine Änderung von } \texttt{Follow}(E')\end{array}\right\} \\
E' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
E' \longrightarrow +TE' & \left\{\begin{array}{l}\text{keine Änderung von } \texttt{Follow}(T) \\ \text{keine Änderung von } \texttt{Follow}(E')\end{array}\right\} \\
T \longrightarrow FT' & \left\{\begin{array}{l}\text{keine Änderung von } \texttt{Follow}(F) \\ \text{keine Änderung von } \texttt{Follow}(T')\end{array}\right\} \\
T' \longrightarrow \varepsilon & \text{keine } \texttt{Follow}\text{-Berechnung möglich} \\
T' \longrightarrow *FT' & \left\{\begin{array}{l}\text{keine Änderung von } \texttt{Follow}(F) \\ \text{keine Änderung von } \texttt{Follow}(T')\end{array}\right\} \\
F \longrightarrow (E) & \text{keine Änderung von } \texttt{Follow}(E) \\
F \longrightarrow id & \text{keine } \texttt{Follow}\text{-Berechnung möglich}
\end{array}
$$

Somit hat sich kein $\texttt{Follow}$ mehr verändert, das heißt die Berechnung von $\texttt{Follow}$ ist abgeschlossen und wir erhalten zusammenfassend

x	E	E'	T	T'	F
$\texttt{Follow}(x)$	$\$,)$	$\$,)$	$\$,+,)$	$\$,+,)$	$\$,+,*,)$

Damit sind alle Vorarbeiten zum Aufbau einer Parsertafel erledigt, und wir können mithilfe des Algorithmus die Top-down-Parsertafel zu der betrachteten Beispielgrammatik erstellen. Wir erhalten

V_N \ V_T	id	$+$	$*$	$($	$)$	$\$$
E	$E \longrightarrow TE'$			$E \longrightarrow TE'$		
E'		$E' \longrightarrow +TE'$			$E' \longrightarrow \varepsilon$	$E' \longrightarrow \varepsilon$
T	$T \longrightarrow FT'$			$T \longrightarrow FT'$		
T'		$T' \longrightarrow \varepsilon$	$T' \longrightarrow *FT'$		$T' \longrightarrow \varepsilon$	$T' \longrightarrow \varepsilon$
F	$F \longrightarrow id$			$F \longrightarrow (E)$		

Fehlerbehandlung

Nach einem aufgetretenen Fehler soll der Ablauf an einer "sinnvollen" Stelle fortgesetzt werden.

a) Panic-Mode

"Überspringe" die Eingabetokens solange, bis ein Token aus einer vorgegebenen Tabelle gefunden wird. Mit diesem Token wird fortgesetzt. Heuristiken zum Erstellen einer solchen Tabelle sind

1.) Zu einem $A \in V_N$ wähle alle $a \in \texttt{Follow}(A)$. Abbruch des Ersetzens von A, das heißt $A \longrightarrow \varepsilon$ wird ausgeführt.

2.) Besagte Tokens sind zum Beispiel keywords der Sprache, die vorherige Konstruktionen abschlie en (`;`, `else` oder `end` etc.).

3.) Solche Tokens können auch das Erreichen von Symbolen aus $\texttt{First}(A)$ sein. Dann kann das Ersetzen von A wiederaufgenommen werden.

4.) Ist $A \stackrel{*}{\Longrightarrow} \varepsilon \in R$, so verwende diese Regel.

5.) Entferne das oberste Stacksymbol, das nicht ersetzt werden kann. Dann besteht die Tabelle aus allen Terminalsymbolen.

b) Fallbedingte Fehlerbehandlung

Diese Fehlerbehandlung ermöglicht die bedingte Korrektur der falschen Eingabe. Sie ist mit Aufrufen von Unterprogrammen verbunden, weshalb die Gefahr von Rekursion besteht.

Bottom-up-Erstellung von Syntaxbäumen

Beispiel ("`if - then - else`")

Gegeben sei folgende Regelmenge

$$S \longrightarrow \texttt{i}\ B\ \texttt{t}\ R\ \texttt{e}\ S | \texttt{i}\ B\ \texttt{t}\ S | a$$
$$R \longrightarrow \texttt{i}\ B\ \texttt{t}\ R\ \texttt{e}\ R | a$$
$$B \longrightarrow b$$

wobei

$$a = \text{atomarer Ausdruck}$$
$$b = \text{boolscher Ausdruck}$$
$$\texttt{i} = \texttt{if}$$
$$\texttt{e} = \texttt{else}$$
$$\texttt{t} = \texttt{then}$$

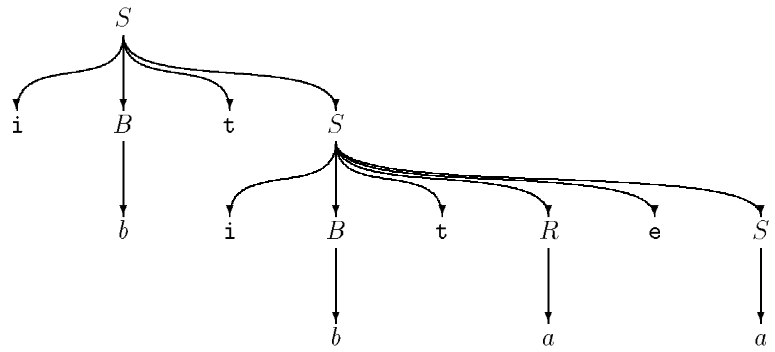
Zu erkennen ist der Ausdruck

$$\texttt{i}\ b\ \texttt{t}\ \texttt{i}\ b\ \texttt{t}\ a\ \texttt{e}\ a$$

Suche jeweils von *links* das erste Teilwort, das durch Anwenden einer Regel reduziert werden kann. Man erhält die Ableitungsfolge

$\texttt{i}\ \boxed{b}\ \texttt{t}\ \texttt{i}\ b\ \texttt{t}\ a\ \texttt{e}\ a$	verwende$(B \longrightarrow b, 2)$	Die “2” bezeichnet die Stelle, an der die Ersetzung ausgeführt werden soll.
$\texttt{i}\ B\ \texttt{t}\ \texttt{i}\ \boxed{b}\ \texttt{t}\ a\ \texttt{e}\ a$	verwende$(B \longrightarrow b, 5)$	
$\texttt{i}\ B\ \texttt{t}\ \texttt{i}\ B\ \texttt{t}\ \boxed{a}\ \texttt{e}\ a$	verwende$(R \longrightarrow a, 7)$	$(S \longrightarrow a, 7)$ führt nicht zum Ziel!
$\texttt{i}\ B\ \texttt{t}\ \texttt{i}\ B\ \texttt{t}\ R\ \texttt{e}\ \boxed{a}$	verwende$(S \longrightarrow a, 9)$	
$\texttt{i}\ B\ \texttt{t}\ \boxed{\texttt{i}\ B\ \texttt{t}\ R\ \texttt{e}\ S}$	verwende$(S \longrightarrow \texttt{i}B\texttt{t}R\texttt{e}S, 9)$	
$\boxed{\texttt{i}\ B\ \texttt{t}\ S}$	verwende$(S \longrightarrow \texttt{i}B\texttt{t}S, 4)$	
S		

Die folgende Abbildung zeigt den Aufbau des Syntaxbaumes von der Wurzel an.



Der Syntaxbaum stellt eine Rechtsableitung für den zu erkennenden Ausdruck dar.

Bemerkung

Bei einer Rechtsableitung werden die am weitesten rechts stehenden Nichtterminalsymbole beim Erstellen des Syntaxbaumes zuerst ersetzt.

3.10 Definition

Sei G eine kontextfreie Grammatik. Sei

$$w_0 \stackrel{*}{\Longrightarrow} \alpha A w \Longrightarrow \alpha\beta w$$

eine Rechtsableitung, wobei w_0 das Startsymbol ist und $A \in V_N$ und $w \in V_T^*$ gilt. Ist

$$\gamma = \alpha\beta w \quad \text{und} \quad i = \text{Länge}(\alpha\beta),$$

so heißt

$$(A \longrightarrow \beta, i)$$

ein *Handle* für γ.

Bei der Entscheidung, welche Regel $A \longrightarrow \alpha_1, A \longrightarrow \alpha_2, \ldots, A \longrightarrow \alpha_n$ anzuwenden ist, kann man eine feste Zahl k von Anfangsbuchstaben des nachfolgenden Wortes w verwenden.

Definition

Für $w \in V_T^*$ sei

$$\texttt{pref}_k(w) = \begin{cases} a_1 \ldots a_k & \text{wenn } w = a_1 \ldots a_k a_{k+1} \ldots a_n \text{ mit } n \geq k \\ w & \text{sonst.} \end{cases}$$

3.11 Definition

Sei $k \geq 0$ und G eine kontextfreie Grammatik ohne überflüssige Symbole. Aus dem Startsymbol w_0 läßt sich durch keine Ableitungsfolge w_0 nochmals produzieren, das heißt $w_0 \stackrel{+}{\Longrightarrow} w_0$ gilt nicht. G heißt dann *LR(k)-Grammatik*, wenn für jedes Wort γ aus V^*, das Handle eindeutig bestimmt ist, das heißt

$$\text{für alle } w, w', \alpha, \alpha', \beta, \beta', \delta \in V^*, x \in V_T^*, A \in V_N, A' \in V_N$$

gilt

$$\text{zu } \delta w \text{ und } \delta w' \text{ mit } \mathtt{pref}_k(w) = \mathtt{pref}_k(w')$$

mit

$$w_0 \stackrel{*}{\Longrightarrow} \alpha A w \Longrightarrow \alpha\beta w = \delta w$$

und

$$w_0 \stackrel{*}{\Longrightarrow} \alpha' A' x \Longrightarrow \alpha'\beta' x = \delta w'$$

ist das Handle $(A \longrightarrow \beta, \text{Länge}(\alpha\beta))$ eindeutig, das heißt

$$(A \longrightarrow \beta, \text{Länge}(\alpha\beta)) = (A' \longrightarrow \beta', \text{Länge}(\alpha'\beta')).$$

3.12 Satz

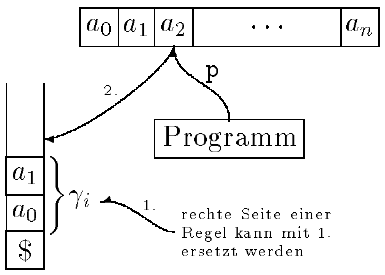
Jede LR(k)-Grammatik ist eindeutig, das heißt für jede LR(k)-Grammatik existiert genau ein Syntaxbaum.

Beweis

Da jeweils nur eine Rechtsableitung existiert, kann man den Syntaxbaum zu jeder anderen Ableitung als Rechtsableitung lesen und eindeutig identifizieren.

Ein Parser für die Bottom-up-Verarbeitung besitzt zwei Arten von Aktionen

1. Reduzieren anhand einer Regel (`reduce`).
2. Einlesen von Eingabezeichen in den Stack (`shift`).



Wenn oben auf dem Stack die rechte Seite einer Regel liegt, kann der Inhalt mit `reduce` ersetzt werden. Da die verarbeiteten Zeichen oben im Stack abgelegt werden, bewirkt die Strategie, jede erkannte rechte Seite sofort zu ersetzen, daß eine vollständige rechte Seite stets nur *oben* auf dem Stack liegen kann.

Beispiel

Gegeben sei die Regelmenge

$$E \longrightarrow E + E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow id$$

Die Zeichenfolge $id + id * id$ soll erkannt werden. Dann verdeutlicht die folgende Tabelle obige Vorgehensweise.

Stack	id	$+$	id	$*$	id	$\$$	Aktion
$\$$	$\uparrow$						`shift`
$\$id$		$\uparrow$					`reduce` $E \longrightarrow id$
$\$E$		$\uparrow$					`shift`
$\$E+$			$\uparrow$				`shift`
$\$E+id$				$\uparrow$			`reduce` $E \longrightarrow id$
$\$E+E$				$\uparrow$			`reduce` $E \longrightarrow E+E$
$\$E$				$\uparrow$			`shift`
$\$E*$					$\uparrow$		`shift`
$\$E*id$						$\uparrow$	`reduce` $E \longrightarrow id$
$\$E*E$						$\uparrow$	`reduce` $E \longrightarrow E*E$
$\$E$						$\uparrow$	

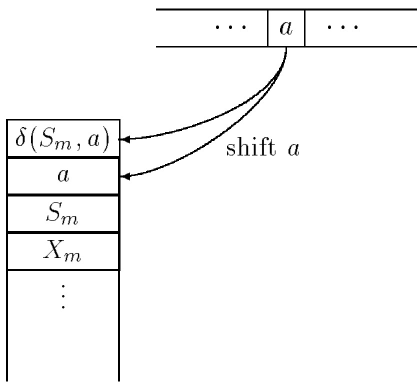
Im Stack liegt nur das Startsymbol E und der Einlesezeiger steht auf \$, das heißt das eingelesene Wort gehört zu der Sprache der Grammatik.

Eine Eingabefolge wie zum Beispiel $id***$ verursacht eine Fehlermeldung.

Es tritt allerdings das Problem auf, zu erkennen, ob die rechte Seite einer Regel oben auf dem Stack liegt. Aus Geschwindigkeitsgründen verbietet sich ein paarweiser Vergleich jeder rechten Seite mit dem Stack.

Man verwendet deshalb folgenden Trick. Notiere im Stack Zustände, die jeweils charakterisieren, welcher Inhalt darunter im Stack liegt. Konstruiere einen endlichen Automat, der die Wortanfänge, die zu rechten Seiten von Regeln gehören, als Zustände hat.

Die Zustände werden im Stack abgelegt. Bei Entscheidungen wird das betrachtete Eingabezeichen und der oben liegende Zustand verwendet, um den Nachfolgezustand zu berechnen.

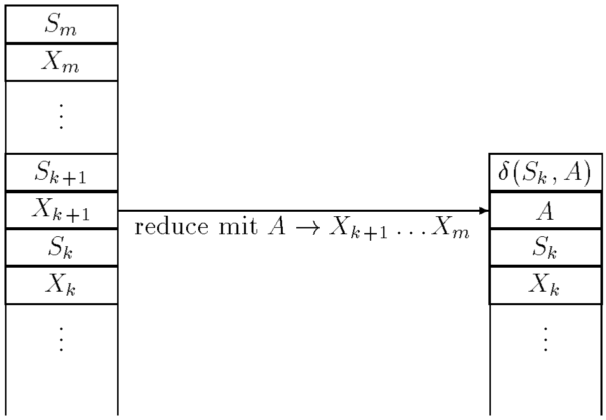


Die `shift`-Operation läßt sich also durch

$$\text{“neuer Zustand} = \delta(\text{alter Zustand}, a)\text{”}$$

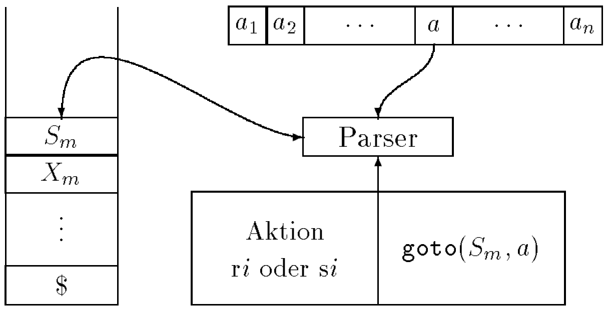
beschreiben.

Falls die rechte Seite einer Regel erkannt wird, wird zuerst reduziert. Die Abbildung veranschaulicht die Vorgehensweise bei einer `reduce`-Operation mit der Regel $A \longrightarrow X_{k+1} \ldots X_m$.



Dabei wird $\delta(S_k, A)$ auch als `goto`(S_k, A) bezeichnet. Die `reduce`-Operation läßt sich also durch eine `goto`-Anweisung beschreiben.

Der Aufbau eines Bottom-up-Parsers sieht in etwa folgenderma en aus



Dabei bezeichnet

ri: verwende die i-te Regel
si: lese ein Zeichen ein (`shift`) und lege den i-ten Zustand oben auf den Stack

Beispiel

Gegeben sei die Regelmenge

$$
\begin{array}{ll}
1 & E \longrightarrow E + T \\
2 & E \longrightarrow T \\
3 & T \longrightarrow T * F \\
4 & T \longrightarrow F \\
5 & F \longrightarrow (E) \\
6 & F \longrightarrow id
\end{array}
$$

und die zugehörige Parsertafel

	Aktion						goto		
Zustand	*id*	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		s2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5							9	3
7	s5								10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Erkannt werden soll wiederum das Wort $w = id * id + id$. Wir verdeutlichen den Ablauf anhand folgender Tabelle.

Stack	Input	Aktion
0	$id * id + id\$$	`shift`
0 *id* 5	$* id + id\$$	`reduce` $F \longrightarrow id$
0 F 3	$* id + id\$$	`reduce` $T \longrightarrow F$
0 T 2	$* id + id\$$	`shift`
0 T 2 * 7	$id + id\$$	`shift`
0 T 2 * 7 *id* 5	$+ id\$$	`reduce` $F \longrightarrow id$
0 T 2 * 7 F 10	$+ id\$$	`reduce` $T \longrightarrow T * F$
0 T 2	$+ id\$$	`reduce` $E \longrightarrow T$
0 E 1	$id\$$	`shift`
0 E 1 + 6	\$	`shift`
0 E 1 + 6 *id* 5	\$	`reduce` $F \longrightarrow id$
0 E 1 + 6 F 3	\$	`reduce` $T \longrightarrow F$
0 E 1 + 6 T 9	\$	`reduce` $E \longrightarrow E + T$
0 E 1	\$	wird akzeptiert

0 ist der Startzustand. Die neuen Zustände erhält man durch Nachsehen in der Parsertafel an der entsprechenden Stelle. Beachte, daß bei `reduce` der oberste Zustand *nach* Entfernen der rechten Seite für die `goto`-Auswertung entscheidend ist.

Wie stellt man nun allgemein eine Bottom-up-Parsertafel auf?

Die Zustände werden durch sogenannte Items beschrieben. Ist $A \longrightarrow w$ eine Regel, wobei $w = w_1 w_2$, so ist

$$A \longrightarrow w_1.w_2$$

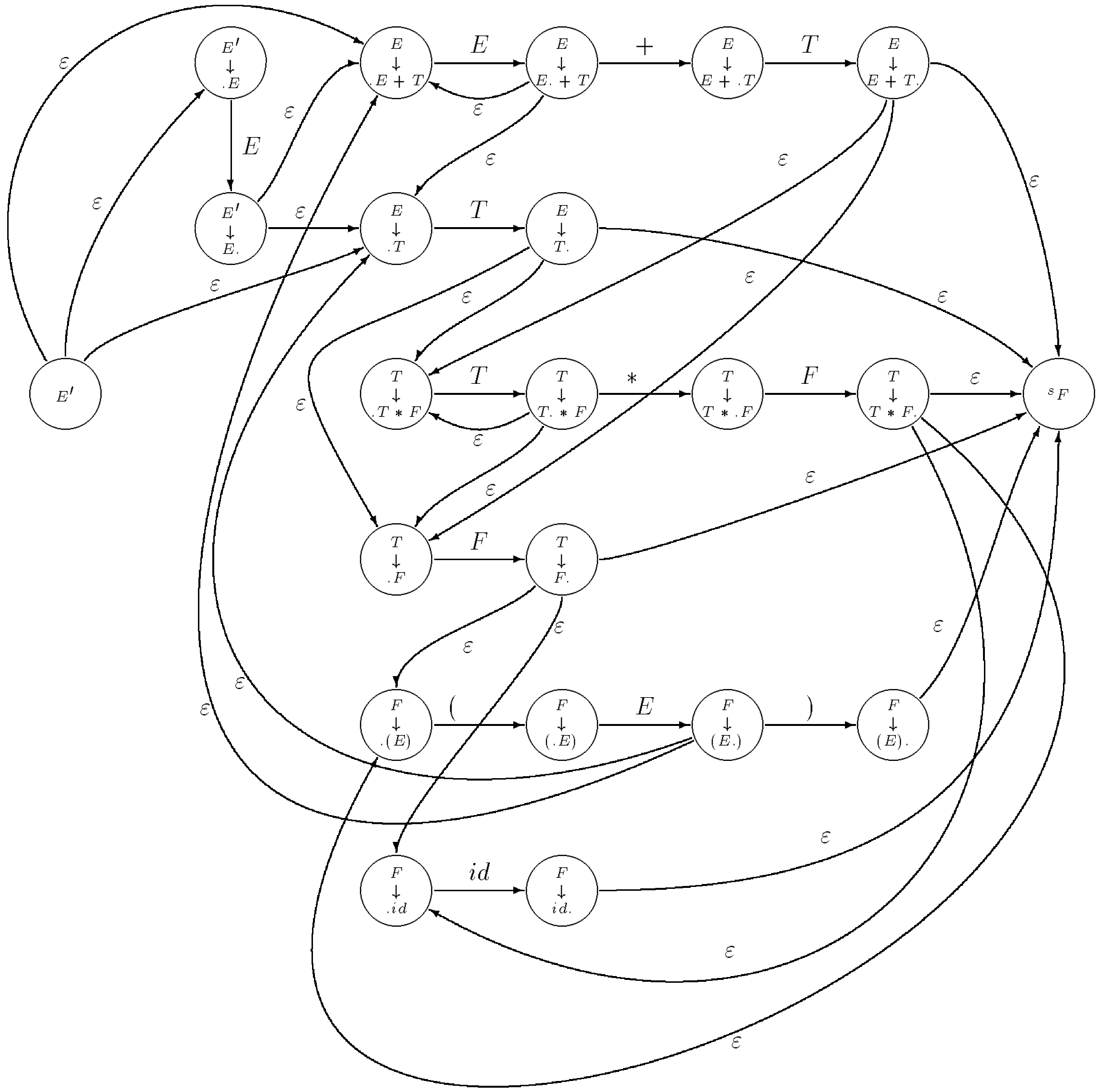
ein Item. Der Wortteil w_1 ist der bereits erkannte Teil der Regel, w_2 ist noch zu erkennen.

Zu $E \longrightarrow E * T$ gibt es also die Items

$$\begin{aligned} E &\longrightarrow .E * T \\ E &\longrightarrow E . * T \\ E &\longrightarrow E * .T \\ E &\longrightarrow E * T. \end{aligned}$$

Beispiel

Betrachte obige Regelmenge. Baue einen ε-Automaten für die Wortanfänge. Dabei wird E' als künstliches Startsymbol benötigt (vergleiche folgende Abbildung).



Durch die Teilmengen-Konstruktion erhält man aus den ε-Automaten einen determinierten Automaten mit derselben Sprache.

Sei I eine Item-Menge. Dann wird `Abschluß`(I) berechnet durch

```
1. Initialisiere
       J := I.
2. Wiederhole
    {
       Ist A ⟶ α.Bβ ∈ J und B ⟶ γ eine Regel
          füge B ⟶ .γ zu J hinzu
    } bis sich J nicht mehr ändert.
3. Setze Abschluß(I) = J.
```

Definition

Sei I eine Item-Menge und $X \in V$. Dann definieren wir $\texttt{goto}(I, X)$ durch

$$\texttt{goto}(I, X) := \texttt{Abschluß}(\{ A \longrightarrow \alpha X.\beta \mid A \longrightarrow \alpha.X\beta \in I \}).$$

Zu einer gegebenen Grammatik G mit Startsymbol S wird die Regel $S' \longrightarrow S$ hinzugenommen (erweiterte Grammatik). Dies ist nötig zur Konstruktion des ε-Automaten, aus dem durch die Teilmengenkonstruktion ein determinierter Automat erzeugt wird.

Damit lassen sich die kanonischen LR(0)-Item-Mengen konstruieren.

```
Initialisiere
   C := {Abschluß({S' ⟶ .S})}.
Wiederhole
{
   Für jedes I ∈ C und für jedes Grammatiksymbol X
      bilde J = goto(I, X)
   Ist J ≠ ∅
      füge J in C ein.
} bis sich C nicht mehr ändert.
```

Beachte, daß C eine Menge von Mengen ist!

Die Reduktion anhand einer Regel ist keine Automatenoperation. Daraus folgt, daß die Entscheidung über die Reduktionseinträge nicht mit der goto-Funktion zu finden sind.

Betrachte die Ableitungsfolge

$$\begin{aligned} S' &\overset{*}{\Longrightarrow} \alpha A w \\ &\Longrightarrow \alpha\beta_1\beta_2 w \end{aligned}$$

Sei der Zustand I erreicht mit $A \longrightarrow \beta_1.\beta_2 \in I$. Ist dann $\beta_2 = \varepsilon$, so liegt oben auf dem Stack die rechte Seite von $A \longrightarrow \beta_1 \in R$, und es sollte reduziert werden.

3.13 Konstruktion einer SLR-Parser-Tafel (S steht für simple)

Eingabe

Grammatik G', die aus G durch Hinzunahme von $S' \longrightarrow S$ erzeugt wurde.

Methode

```
1. Konstruiere die kanonischen LR(0)-Item-Mengen.
2. Ordne die Item-Mengen an und kodiere I_i durch i.
    Ist goto(I_i, X) = I_j
       setze goto(i, X) = j.
3. Ist A ⟶ α.aβ ∈ I_i und goto(i, a) = j mit a ∈ V_T
      setze action[i, a] = shift j
4. Ist A ⟶ α. ∈ I_i
    {
       setze für alle a ∈ Follow(A)
          action[i, a] = reduce mit A ⟶ α
    }
5. Ist S' ⟶ S. ∈ I_i
       setze action[i, $] = akzeptiere.
6. Fülle alle leeren Tabellenplätze mit Fehleraufrufen.
```

Definition

Hat für eine Grammatik G die SLR-Parsertafel keine mehrfachen Einträge, so heißt G eine SLR(0)-Grammatik.

In einer Item-Menge sind verschiedene Items enthalten. Zum selben Grammatiksymbol X könnten die verschiedenen Items für verschiedene Einträge in der Parsertafel sorgen. Wenn beispielsweise an einer Stelle gleichzeitig `shift` und `reduce`-Anweisungen stehen, spricht man von einem sogenannten `shift-reduce`-*Konflikt*.

Beispiel

Gegeben sei die Regelmenge R durch

$$\begin{aligned} S &\longrightarrow L = R \\ S &\longrightarrow R \\ L &\longrightarrow *R \\ L &\longrightarrow id \\ R &\longrightarrow L \end{aligned}$$

Um die SLR-Parsertafel erstellen zu können benötigen wir zunächst die kanonischen LR(0)-Item-Mengen. Berechne dazu zunächst $I_0 = \texttt{Abschluß}(\{S' \longrightarrow .S\})$. Man erhält

$$\begin{array}{c} I_0 \\ S' \longrightarrow .S \\ S \longrightarrow .L = R \\ S \longrightarrow .R \\ L \longrightarrow .*R \\ L \longrightarrow .id \\ R \longrightarrow .L \end{array}$$

Die Betrachtung aller dieser Items liefert im ersten Durchgang (für jedes Item müssen alle Grammatiksymbole ($S', S, L, R, *, id, =$) beachtet werden)

$I_1 = \texttt{goto}(I_0, S)$	$I_2 = \texttt{goto}(I_0, L)$	$I_3 = \texttt{goto}(I_0, R)$	$I_4 = \texttt{goto}(I_0, *)$	$I_5 = \texttt{goto}(I_0, id)$
$S' \longrightarrow S.$	$S \longrightarrow L. = R$	$S \longrightarrow R.$	$L \longrightarrow *.R$	$L \longrightarrow id.$
	$R \longrightarrow L.$		$R \longrightarrow .L$	
			$L \longrightarrow . * R$	
			$L \longrightarrow .id$	

Für "=" ergibt sich keine Item-Menge, da in keinem Item aus I_0 der "." vor einem "=" auftritt. Gleiches gilt (immer) für S'. Der erste Durchgang ist somit beendet und ein weiterer Durchgang liefert

$$\begin{aligned} &I_6 = \texttt{goto}(I_2, =)\\ &S \longrightarrow L = .R\\ &R \longrightarrow .L\\ &L \longrightarrow . * R\\ &L \longrightarrow .id \end{aligned}$$

und

$$\begin{aligned} &I_7 = \texttt{goto}(I_4, R)\\ &L \longrightarrow *R. \end{aligned}$$

sowie

$$\begin{aligned} &I_8 = \texttt{goto}(I_4, L)\\ &R \longrightarrow L. \end{aligned}$$

$\texttt{goto}(I_4, *)$ beziehungsweise $\texttt{goto}(I_4, id)$ liefern die bereits vorhandenen Itemmengen I_4 beziehungsweise I_5. Aufgrund der Tatasache, daß in den Item-Mengen I_1, I_3 und I_5 der "." bereits hinter dem letzten Symbol der rechten Seite aller Items steht, liefern diese Mengen keine neuen Item-Mengen. In diesem Durchlauf haben wir neue Item-Mengen erhalten, die wir nun in einem erneuten Durchgang betrachten müssen. Wir erhalten

$$\begin{aligned} &I_9 = \texttt{goto}(I_6, R)\\ &S \longrightarrow L = R. \end{aligned}$$

$\texttt{goto}(I_6, L)$, $\texttt{goto}(I_6, *)$ beziehungsweise $\texttt{goto}(I_6, id)$ liefern bereits vorhandene Item-Mengen. Für I_7 und I_8 ist kein `goto` berechenbar.

Nun ist in einem weiteren Durchlauf das neu erzeugte I_9 zu betracten. Da aber im einzigen Item von I_9 der "." bereits ganz rechts steht, ist keine `goto`-Berechnung möglich. Weil sich keine neue Item-Mengen ergeben haben, ist die Berechnung der kanonischen LR(0)-Item-Mengen abgeschlossen.

Wir betrachten nun bei der Konstruktion der zugehörigen Parsertafel die Schlüsselstelle $[2, =]$ bei der ein `shift-reduce`-Konflikt auftreten wird. Nach Punkt 2. in der Konstruktionsmethode für eine SLR-Parsertafel ist

$$goto(2, =) = 6$$

zu setzen, da I_6 aus I_2 bei Einlesen von "=" folgt. Weiterhin ist $S \longrightarrow L. = R \in I_2$, das heißt wir müssen nach Punkt 3.

$$\texttt{action}[2, =] = \texttt{shift}\,6$$

setzen.

Wir zeigen nun zunächst, daß "=" $\in \texttt{Follow}(R)$ ist. Wir betrachten dazu die Ableitungsfolge

$$\begin{aligned} S &\Longrightarrow L = R\\ &\Longrightarrow *R = R \end{aligned}$$

Also ist "=" ein erstes Terminalsymbol, welches sich hinter R von S ausgehend erhalten läßt, das heißt "="$\in$ Follow(R).

Da nun auch $R \longrightarrow L. \in I_2$ und "="$\in$ Follow(R) müssen wir nach Punkt 4.

$$\texttt{action}[2,=] = \texttt{reduce}\ R \longrightarrow L$$

setzen.

Das bedeutet an der Stelle $[2,=]$ in der Parsertafel stehen zwei Einträge, also beschreibt die Regelmenge R keine SLR(1)-Grammatik.

Wir beschäftigen uns im Folgenden mit der Frage, wie man solche Konflikte vermeiden kann.

Die Menge Follow(A) wird durch Vereinigung (vieler) First(α)-Mengen erzeugt. Man kann nun besser differenzieren, indem man für die auf A in einem Wort folgenden β die Symbole a aus First(β) nur für Reduktionen bei action$[i,a]$ einträgt. Dazu muß bei den Items die Zusatzinformation für solche a's gespeichert werden.

Definition

Ist $A \longrightarrow \alpha\beta$ eine Regel, und $a \in V_T$ oder $a = \$$, so heißt

$$(A \longrightarrow \alpha.\beta, a)$$

ein *LR(1)-Item*.

Für $(A \longrightarrow \alpha., a)$ wird also nur dann eine Reduktion mit $A \longrightarrow \alpha$ wirksam, wenn das nächste Eingabezeichen a ist.

Wir erweitern nun die bisherigen Abschluß-Operationen auf LR(1)-Item-Mengen. Sei dazu

$$\begin{aligned} S &\stackrel{*}{\Longrightarrow} \delta A a x \\ &\Longrightarrow \delta\alpha B\beta a x \end{aligned}$$

eine Ableitungsfolge und

$$(A \longrightarrow \alpha.B\beta, a)$$

ein LR(1)-Item. Ist nun $B \longrightarrow \eta \in R$, so müssen wir in den Abschluß

$$(B \longrightarrow .\eta, b)$$

einfügen, wobei b das Anfangssymbol ist, das bei der Reduktion von $\beta a x$ auftreten kann. Das heißt ist

$$\begin{aligned} \beta = \varepsilon &\Longrightarrow b = a \\ \beta \neq \varepsilon &\Longrightarrow b \in \texttt{First}(\beta). \end{aligned}$$

Sei I eine LR(1)-Item-Menge, und G' die erweiterte Grammatik zu G. Dann wird Abschluß(I) berechnet durch

```
Wiederhole
{
   Für jedes (A ⟶ α.Bβ, a) ∈ I
   {
      Für jedes B ⟶ γ ∈ G'
      {
         Für jedes b ∈ First(βa)
            erweitere I um (B ⟶ .γ, b).
      }
   }
} bis I nicht mehr wächst.
```

Definition

Sei I eine LR(1)Item-Menge und $X \in V$. Dann definieren wir $\texttt{goto}(I, X)$ durch

$$\texttt{goto}(I, X) := \texttt{Abschluß}\,(\{(A \longrightarrow \alpha X.\beta, a) \mid (A \longrightarrow \alpha.X\beta, a) \in I\})\,.$$

Damit lassen sich die kanonischen LR(1)-Item-Mengen konstruieren.

```
Initialisiere
   C := {Abschluß({(S' ⟶ .S, $)})}.
Wiederhole
{
   Für jedes I ∈ C und für jedes Grammatiksymbol X
      bilde J = goto(I, X).
   Ist J ≠ ∅
      füge J in C ein.
} bis sich C nicht mehr ändert.
```

3.14 Konstruktion einer LR(1)-Parsertafel

Eingabe

Grammatik G', die aus G durch Hinzunahme von $S' \longrightarrow S$ erzeugt wurde.

Ausgabe

LR(1)-Parsertafel.

Methode

```
1. Konstruiere die kanonischen LR(1)-Item-Mengen.
2. Ordne die Item-Mengen an und codiere I_i durch i.
     Ist goto(I_i, X) = I_j
        setze goto(i, X) = j.
3. Ist (A ⟶ α.aβ, b) ∈ I_i und goto(i, a) = j mit a ∈ V_T
        setze action[i, a] = shift j.
4. Ist (A ⟶ α., a) ∈ I_i und A ≠ S'
        setze action[i, a] = reduce mit A ⟶ α
5. Ist (S' ⟶ S., $) ∈ I_i
        setze action[i, $] =  akzeptiere.
6. Fülle alle leeren Tabellenplätze mit Fehleraufrufen.
```

Falls keine mehrfachen Einträge auftreten, ist die Grammatik eine LR(1)-Grammatik.

Beispiel

Gegeben sei die Regelmenge

$$\begin{aligned} S' &\longrightarrow S \\ S &\longrightarrow CC \\ C &\longrightarrow cC \mid d \end{aligned}$$

Wir wollen die zugehörige Parsertafel aufstellen. Dazu benötigen wir die LR(1)-Item-Mengen, das heißt wir müssen zunächst

$$I_0 = \texttt{Abschluß}\,(\{(S' \longrightarrow .S, \$)\})$$

berechnen. Dazu bringen wir das Item $(S' \longrightarrow .S, \$)$ in das Schema $(A \longrightarrow \alpha.B\beta, a)$, somit entspricht $B = S$, $\alpha = \varepsilon$, $\beta = \varepsilon$ und $a = \$$. Das zu suchende $B \longrightarrow \gamma$ ist also $S \longrightarrow CC$. Nun ist für jedes

$$b \in \texttt{First}(\beta a) = \texttt{First}(\varepsilon\$) = \texttt{First}(\$) = \{\$\}$$

$(A \longrightarrow \alpha.B\beta, a)$—das heißt $(S \longrightarrow .CC, \$)$ — in I_0 einzufügen. Damit ist nun aber auch $C \longrightarrow cC$ und $C \longrightarrow d$ eine der zu betrachtenden Regeln der Form $B \longrightarrow \gamma$. Wir benötigen wiederum $\texttt{First}(\beta a)$, das heißt

$$\texttt{First}(C\$) = \texttt{First}(C) = \{c, d\}.$$

Damit sind die Regeln

$$(C \longrightarrow .cC, c) \quad (C \longrightarrow .cC, d) \quad (C \longrightarrow .d, c) \quad (C \longrightarrow .d, d)$$

in I_0 einzufügen. Nun stehen hinter dem "." nur Terminalsymbole und deshalb kann es keine Regel der Form $B \longrightarrow \gamma$ — das heißt keine Erweiterung — mehr geben. Wir erhalten also

$$\begin{array}{l} I_0 \\ (S' \longrightarrow .S, \$) \\ (S \longrightarrow .CC, \$) \\ (C \longrightarrow .cC, c) \\ (C \longrightarrow .cC, d) \\ (C \longrightarrow .d, c) \\ (C \longrightarrow .d, d) \end{array}$$

Die Betrachtung aller dieser Items liefert im ersten Durchgang (für jedes Item müssen alle fünf Grammatiksymbole (S', S, C, c, d) betrachtet werden)

$$\begin{array}{l} I_1 = \mathtt{goto}(I_0, S) \\ (S' \longrightarrow S., \$) \end{array}$$

und

$$\begin{array}{l} I_2 = \mathtt{goto}(I_0, C) \\ (S \longrightarrow C.C, \$) \\ (C \longrightarrow .cC, \$) \\ (C \longrightarrow .d, \$) \end{array}$$

und

$$\begin{array}{l} I_3 = \mathtt{goto}(I_0, c) \\ (C \longrightarrow c.C, c) \\ (C \longrightarrow c.C, d) \\ (C \longrightarrow .cC, c) \\ (C \longrightarrow .cC, d) \\ (C \longrightarrow .d, c) \\ (C \longrightarrow .d, d) \end{array}$$

sowie

$$\begin{array}{l} I_4 = \mathtt{goto}(I_0, d) \\ (C \longrightarrow d., c) \\ (C \longrightarrow d., d) \end{array}$$

$\mathtt{goto}(I_0, S')$ liefert wie bereits bei den SLR(0)-Parsern keine Item-Menge. Damit ist der erste Durchlauf vollständig.

Für den zweiten Durchgang sieht man, daß das Grammatiksymbol S überall nur links vom "." auftritt, es ist daher keine Berechnung von $\mathtt{goto}(?, S)$ möglich. Desweiteren erhalten wir

$$\begin{array}{l} I_5 = \mathtt{goto}(I_2, C) \\ (S \longrightarrow CC., \$) \end{array}$$

und

$$\begin{array}{l} I_6 = \mathtt{goto}(I_2, c) \\ (C \longrightarrow c.C, \$) \\ (C \longrightarrow .cC, \$) \\ (C \longrightarrow .d, \$) \end{array}$$

und

$$\begin{array}{l} I_7 = \mathtt{goto}(I_2, d) \\ (C \longrightarrow d., \$) \end{array}$$

sowie

$$\begin{array}{l} I_8 = \mathtt{goto}(I_3, C) \\ (C \longrightarrow cC., c) \\ (C \longrightarrow cC., d) \end{array}$$

$\mathtt{goto}(I_3, c)$ liefert I_3 und $\mathtt{goto}(I_3, d)$ liefert wieder I_4, und in I_4 ist keine goto-Berechnung möglich. Damit ist der zweite Durchgang beendet.

Für den nächsten Durchlauf ist nun in I_5 keine goto-Berechnung möglich. Allerdings erhalten wir

$$\begin{array}{l} I_9 = \mathtt{goto}(I_6, C) \\ (C \longrightarrow cC., \$) \end{array}$$

Aus $\mathtt{goto}(I_6, c)$ erhalten wir wieder I_6, und $\mathtt{goto}(I_6, d)$ liefert I_7. In I_7 und I_8 hingegen ist keine goto-Berechnung möglich. Somit ist auch dieser Durchgang vollständig.

Für einen erneuten Durchlauf sieht man sofort, daß in I_9 keine goto Berechnung möglich ist.

Da nun keine neuen Item-Mengen mehr hinzugekommen sind, ist die Berechnung der kanonischen LR(1)-Item-Mengen abgeschlossen. Damit können wir die zugehörige Parsertafel aufstellen.

	Aktion			goto	
Zustand	c	d	$\$$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r$C \longrightarrow d$	r$C \longrightarrow d$			
5			r$S \longrightarrow CC$		
6	s6	s7			9
7			r$C \longrightarrow d$		
8	r$C \longrightarrow cC$	r$C \longrightarrow cC$			
9			r$C \longrightarrow cC$		

Damit beenden wir die Betrachtung von LR(1)-Parsern und wenden uns im folgenden den sogenannten "Look-Ahead-LR-Parsern" zu.

Die Grundidee dieser Parser liegt darin, in einer LR(1)-Parsertafel Zustände zu fusionieren.

Im obigem Beispiel unterscheiden sich I_4 und I_7 nur in der zweiten Komponente. Ebenso die Item-Mengen I_3 und I_6. Untersuche nun, welche Konflikte entstehen, wenn solche Item-Mengen zu *einem* Zustand zusammengefa t werden.

Liegt oben auf dem Stack eine rechte Seite einer Regel, sagen wir β, wenn $B \longrightarrow \beta$ eine Regel der betrachteten Grammatik ist, dann berührt die Reduktion

$$B \longrightarrow \beta$$

den Zeiger auf das Eingabesymbol nicht!

Nach einer Folge von Reduktionsschritten folgt letztlich eine Shift-Operation. Die Reduktionen unterbrechen also die Worterkennung durch den Automaten.

Betrachten wir nochmals obige Beispielgrammatik

$$\begin{aligned} S' &\longrightarrow S \\ S &\longrightarrow CC \\ C &\longrightarrow cC|d \end{aligned}$$

deren Sprache

$$c^*dc^*d$$

ist. Verarbeitet werden soll die Eingabe $cc\ldots cd\$$. Dann verdeutlicht folgende Tabelle die Vorgehensweise.

Stack	c	c	$\ldots$	c	d	$\$$	Aktion
$\$$	$\uparrow$						`shift`
$\$c$		$\uparrow$					`shift`
$\$cc$			$\ddots$				$\vdots$
$\vdots$				$\uparrow$			`shift`
$\$cc\ldots c$					$\uparrow$		`shift`
$\$cc\ldots cd$						$\uparrow$	`reduce` $C \longrightarrow d$
$\$cc\ldots cC$						$\uparrow$	`reduce` $C \longrightarrow cC$
$\vdots$						$\vdots$	$\vdots$
$\$ccC$						$\uparrow$	`reduce` $C \longrightarrow cC$
$\$cC$						$\uparrow$	`reduce` $C \longrightarrow cC$
$\$C$						$\uparrow$	

Da am Ende der Abarbeitung nicht das Startsymbol S' der Grammatik erreicht wurde, muß ein Fehleraufruf ausgegeben werden.

Es ist zu prüfen, ob der Parser mit fusionierten Zuständen in genau der gleichen Weise auf jede Situation reagiert.

Wir wissen bereits, wie wir zu einer gegebenen Grammatik eine LR(1)-Parsertafel aufstellen müssen. Es stellt sich nun die Frage, welche Konflikte beim Verschmelzen von Zustandsmengen, die sich in der linken Seite der Items nicht unterscheiden, auftreten können. Prinzipiell sind natürlich möglich

1. `shift-shift`-Konflikte,
2. `shift-reduce`-Konflikte und
3. `reduce-reduce`-Konflikte.

zu 1.

Sei I eine Menge von LR(1)-Items. Dann ist

$$\texttt{goto}(I,X) := \texttt{Abschluß}(\{\,(A \longrightarrow \alpha X.\beta, a) \mid (A \longrightarrow \alpha.X\beta, a) \in I\,\})\,.$$

Die linken Seiten der entstehenden Item-Mengen sind nur von den linken Seiten der Item-Mengen im Ausgangszustand I abhängig. Daraus folgt, daß die Verschmelzung mit der Berechnung von `goto` verträglich ist.

zu 2.

Sei I Vereinigungsmenge zweier LR(1)-Item-Mengen, die als Zustände der LR(1)-Parsertafel auftreten, und deren linke Seiten der Items übereinstimmen.

Sei $(A \longrightarrow \alpha., a)$ in der ersten Menge, so daß bei a ein Reduzieren verlangt würde. Und $(B \longrightarrow \beta.a\gamma, b)$ in der zweiten Menge, so daß bei a ein `shift` verlangt würde.

Da die linken Seiten bei beiden Item-Mengen gleich sind, existiert in der ersten Menge ein Item

$$(B \longrightarrow \beta.a\gamma, d)$$

für irgendein d. Dann aber hatte bereits die LR(1)-Tafel denselben Konflikt.

zu 3.

Diese Konflikte können auftreten.

Beispiel

Wir betrachten die Regelmenge

$$\begin{aligned} S' &\longrightarrow S \\ S &\longrightarrow aAd|bBd|bAe|aBe \\ A &\longrightarrow c \\ B &\longrightarrow c \end{aligned}$$

mit der Sprache

$$L = \{acd, ace, bcd, bce\},$$

und einen Auszug aus der Item-Menge.

I_3 (für Anfang ac)	I_4 (für Anfang bc)
$(A \longrightarrow c., d)$	$(A \longrightarrow c., e)$
$(B \longrightarrow c., e)$	$(B \longrightarrow c., d)$

Bei dem Zeichen e wird also in I_3 verlangt, daß $B \longrightarrow c$ angewendet wird. Und in I_4 wird verlangt, daß $A \longrightarrow c$ angewendet wird. Nach der Fusion von I_3 und I_4 liegt also ein Konflikt vor!

3.15 Algorithmus zur Konstruktion einer LALR-Parsertafel

Eingabe

Grammatik G', die aus G durch Hinzunahme von $S' \longrightarrow S$ erzeugt wurde.

Ausgabe

LALR-Parsertafel.

Methode

```
1. Konstruiere die Menge C der kanonischen LR(1)-Item-Mengen.
2. Für jedes I ∈ C
   {
      suche alle J ∈ C dessen Items die gleichen linken Seiten haben
      wie die Items in I.
      Ersetze I durch die Vereinigung von I mit all diesen J.
   }
3. Ordne die in 2.entstandene Menge C' an und kodiere J_i durch i
   Definiere den action-Teil der Parsertafel wie im LR(1)-Algorithmus.
   Falls ein Konflikt eintritt, breche mit einer Fehlermeldung ab.
4. Sei J ∈ C', I ∈ C und I ⊆ J.
   Ist goto(I,X) = K mit K ⊆ L ∈ C'
      setze goto(J,X) = L
5. Fülle alle leeren Tabellenplätze mit Fehleraufrufen.
```

Falls kein Konflikt auftritt, ist die Grammatik eine LALR-Grammatik.

Ergänzungen

Falls man die leeren Plätze (Fehleraufrufe) wegläßt, ergibt sich eine Platzeinsparung von ungefähr 90%.

Um die Parsertafel eindeutig füllen zu können, wurden die LR-Grammatiken um Hilfssymbole erweitert.

Beispiel

Die Grammatikregeln

$$E \longrightarrow E + E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow (E)|id$$

werden ersetzt durch

$$E \longrightarrow E + T$$
$$T \longrightarrow T * F|F$$
$$F \longrightarrow (E)|id$$

Die neuen Symbole T und F müssen durch zusätzliche Reduktionen aus dem Stack entfernt werden. Das bedeutet der Parser ist langsam.

Abhilfe

Man verwendet weiterhin die mehrdeutige Grammatik! Dann müssen bei Konflikten im Einzelfall Algorithmen ergänzt werden. Dieses Konzept der Konfliktlösung von außen überläßt die Auflösung eines solchen Konfliktes dem Programmierer des Parsers.

Es gilt nicht mehr die Maxime, daß eine vollständige rechte Seite auf der Stackspitze reduziert werden muß. Der Zweck der Grammatik muß entscheiden, welcher Eintrag in der Parsertafel in welcher Situation den Konflikt in der gewünschten Weise löst.

Wir beziehen uns nochmals auf das letzte Beispiel, indem $*$ stärker bindet als $+$. Zu erkennen ist der Ausdruck $id + id * id$.

Sei der Wortanfang $id + id$ bereits gelesen, und $*\,id$ sei noch zu verarbeiten, das heißt der Zeiger $\uparrow$ steht momentan auf "$*$". Die Item-Mengen I_0, I_1, I_4 und I_7 (die übrigen Item-Mengen seien au er Acht gelassen) lauten wie folgt

$$\begin{array}{llll}
I_0 & I_1 & I_4 & I_7 \\
E' \longrightarrow .E & E' \longrightarrow E. & E \longrightarrow E+.E & E \longrightarrow E+E. \\
E \longrightarrow .E+E & E \longrightarrow E.+E & E \longrightarrow .E+E & E \longrightarrow E.+E \\
E \longrightarrow .E*E & E \longrightarrow E.*E & E \longrightarrow .E*E & E \longrightarrow E.*E \\
E \longrightarrow .(E) & & E \longrightarrow .(E) & \\
E \longrightarrow .id & & E \longrightarrow .id &
\end{array}$$

Im Stack liegt demnach $0\ E\ 1\ +\ 4\ E\ 7$.

Es entsteht also folgender Konflikt

$E+E.$	die rechte Seite einer Regel liegt im Stack, das heißt `reduce` mit $E \longrightarrow E+E$
$E.*E$	E liegt oben auf den Stack und $*$ soll eingelesen werden, das heißt `shift`

`reduce` bedeutet aber, daß $id+id*id$ wie $(id+id)*id$ ausgewertet würde, `shift` hingegen würde bedeuten, daß es wie $id+(id*id)$ ausgewertet würde, also ist die `shift`-Operation erwünscht.

Ähnliches gilt für das `if then else`-Problem.

Zu erkennen ist das Wort

$$\texttt{if}\ e\ \texttt{then}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s$$

Dann liegt der Konflikt darin, entweder

$$1.\ s \longrightarrow \texttt{if}\ e\ \texttt{then}\ s$$

oder

$$2.\ s \longrightarrow \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s$$

anzuwenden.

Dabei entspricht 1. anzuwenden, daß das `else` zum ersten `if` gehört, und 2. anzuwenden, daß das `else` zum zweiten `if` gehört.

Da wir weiter vorne bereits per Konvention festgelegt haben, daß ein `else` immer zum letzten `if` gehört, wäre also 2. anzuwenden.

Fehlerbehandlung

Die LR-Parser lesen im Fehlerfall nie ein weiteres Zeichen ein. Um weiter hinten liegende Fehler im selben Durchlauf aufdecken zu können, geht man wie folgt vor

1. Entnehme solange Symbole aus dem Stack, bis ein Symbol A aus einer festen Liste oben liegt.
2. Wandere mit dem Zeiger beim Eingabewert solange weiter, bis für das angezeigte Zeichen a in der Parsertafel $M[A, a]$ wohldefiniert ist. Fahre an dieser Stelle fort.

4 Theoretische Ergänzungen zu kontextfreien Grammatiken

Einsichten

Gewisse Sprachen sind nicht kontextfrei definierbar. Jede kontextfreie Sprache läßt sich automatisch erkennen! Das gilt auch für nicht eindeutige!

Nur für theoretische Beweise werden im folgenden spezielle Normalformen für kontextfreie Grammatiken ausgezeichnet.

4.1 Definition

Sei $G = (V_N, V_T, R, S)$ eine kontextfreie Grammatik. G ist in *Chomsky-Normalform*, wenn jede Regel von der Form

$$X \longrightarrow YZ \quad \text{oder} \quad X \longrightarrow a \qquad \text{mit } X, Y, Z \in V_N,\ a \in V_T$$

ist.

Bemerkung

Ist $S \stackrel{*}{\Longrightarrow} w$ für eine Grammatik in Chomsky-Normalform mit $w \in V^*$, so ist für $w \Longrightarrow w'$ stets

$$l(w') \geq l(w),$$

wobei l als Funktion die Wortlänge liefert. Insbesondere ist $\epsilon \notin L(G)$.

Betrachte im folgenden nur Grammatiken G mit $\epsilon \notin L(G)$.

4.2 Satz

Sei G eine kontextfreie Grammatik und $\epsilon \notin L(G)$. Dann gibt es eine Grammatik G' in Chomsky-Normalform mit

$$L(G) = L(G').$$

Beweis

Der Beweis erfolgt in zwei Schritten.

1) Entfernen aller Regeln $A \longrightarrow \epsilon$ ist möglich.

Nehme das Einsetzen jeder solchen Regel vorweg, dann ist $U := \{ X \in V_N \mid X \stackrel{*}{\Longrightarrow} \epsilon \}$ zu berechnen.

```
Initialisiere
    U = {ε}.
Wiederhole
{
    U = U ∪ { X | X ⟶ α ∈ R,  α ∈ U* }
} bis sich U nicht mehr ändert.
```

Diese Schleife ist endlich, da $|V_N| < \infty$ und $|R| < \infty$. Die ursprüngliche Regelmenge R wird somit ersetzt durch die Regelmenge R'

$$R' = \left\{ X \longrightarrow \alpha' \;\middle|\; \begin{array}{l} \alpha' \neq \epsilon, \text{ zu } \alpha' \text{ existiert } \alpha \text{ mit } X \longrightarrow \alpha \in R \text{ und} \\ \alpha' \text{ entsteht aus } \alpha \text{ durch Fortlassen von Symbolen aus } U \end{array} \right\}.$$

2) Betrachte alle Regeln der Form $X \longrightarrow X_1 \ldots X_n$.

- Ziel ist es zunächst alle Regeln der Form $X \longrightarrow X_1$ zu entfernen.

  Bilde dazu die Menge $U(X) := \{Y \in V_N \cup V_T \mid X \stackrel{*}{\Longrightarrow} Y\}$.

Initialisiere
    $U(X) = \{X\}$.
Wiederhole
{
    $U(X) = U(X) \cup \{ Y \in V_N \cup V_T \mid$ Es existiert $Z \in U$ mit $Z \longrightarrow Y \in R\}$
} bis sich $U(X)$ nicht mehr ändert.

Ergänze $X \longrightarrow X_1 \dots X_n$ durch alle $Y \longrightarrow Y_1 \dots Y_n$ mit $X \in U(Y)$ und $Y_i \in U(X_i)$ für alle i.

Nun sind keine Regeln der Form $X \longrightarrow X_1$ mehr nötig, und werden deshalb entfernt. Stets ist bei $X \longrightarrow X_1 \dots X_n$ nun $n \geq 2$.

$$\begin{tikzcd}
 & Y \arrow[d] & & \\
 & X \arrow[dl] \arrow[d] \arrow[drr] & & \\
X_1 \arrow[d] & X_2 \arrow[d] & \cdots & X_n \arrow[d] \\
Y_1 & Y_2 & \cdots & Y_n
\end{tikzcd}$$

- Ist $a \in L(G)$, so füge die Regel $S \longrightarrow a$ in R ein.
- Ist $X \longrightarrow X_1 \dots x_i \dots X_n \in R$, wobei $x_i \in V_T$ ist, so erfinde ein neues Symbol $\overline{X}_i \in V_N$ und ersetze x_i durch $\overline{X}_i$ und füge
$$\overline{X}_i \longrightarrow x_i$$
in R ein.
- Betrachte nun die Regeln der Form $X \longrightarrow X_1 \dots X_n$ wobei $X_i \in V_N$ für alle i und $n \geq 3$ ist.

  Erfinde neue Symbole $Z_i \in V_N$ und ersetze die obige Regel durch
$$\begin{aligned}
X &\longrightarrow X_1 Z_1 \\
Z_1 &\longrightarrow X_2 Z_2 \\
&\vdots \\
Z_{n-2} &\longrightarrow X_{n-1} X_n
\end{aligned}$$

4.3 Satz (Pumping-Lemma)

Sei G eine kontextfreie Grammatik. Dann gibt es $p, q \in I\!N$, so daß für $Z \in L(G)$ mit $l(Z) > p$ gilt
$$Z = UVWXY, \quad l(VWX) \leq q, \quad VX \neq \epsilon$$
und für alle $i \in I\!N$ ist
$$UV^iWX^iY \in L(G).$$

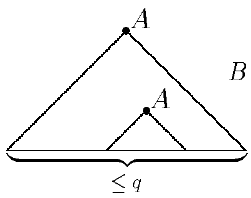
Beweis

Ohne Beschränkung der Allgemeinheit sei G in Chomsky-Normalform, das heißt es liegen nur Regeln der Form $X \longrightarrow AB$ oder $X \longrightarrow a$ vor.

Ein Ableitungsbaum für ein Wort ist ein binärer Baum, denn jeder Knoten hat höchstens zwei Söhne. Ist T ein Ableitungsbaum der Höhe $h(T)$, so hat T höchstens $2^{h(T)}$ Blätter. Bei Wortlänge $l > 2^n$ ist $h(T) > n$.

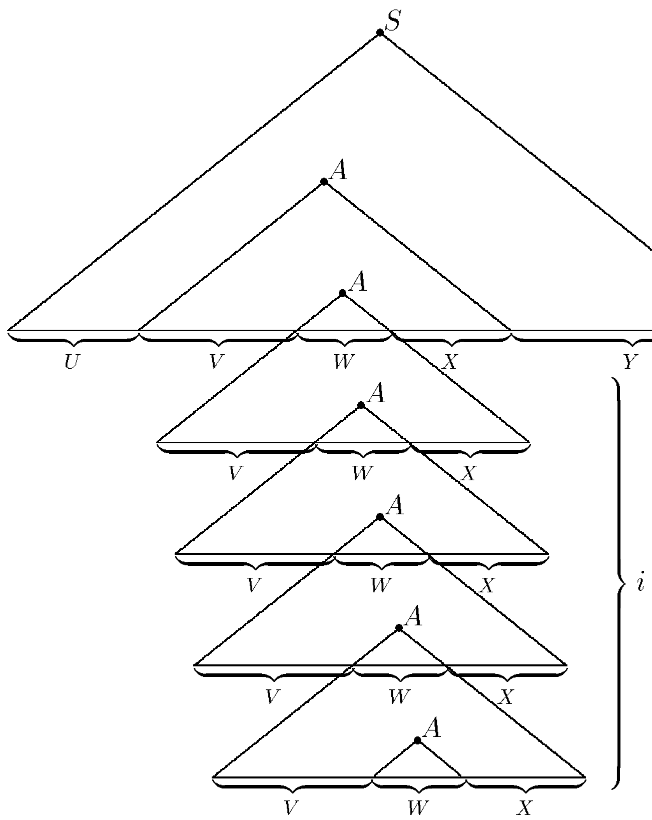
Sei $|V_N| = n$, $p = 2^n$, $q = 2^{n+1}$, $Z \in L(G)$ und $l(Z) > p = 2^n$. Dann hat der Ableitungsbaum mindestens die Höhe $n+1$. Das heißt es existiert ein Weg von der Wurzel zu einem Blatt, der mindestens $n+2$ Knoten durchläuft, wovon einer ein Blatt ist, und $n+1$ Knoten mit Symbolen aus V_N. Die Anzahl der Nichtterminalsymbole $|V_N|$ ist aber gerade n, das heißt es existiert ein $A \in V_N$ welches doppelt auftritt.

Betrachte unter all solchen Situationen eine solche, bei der die Höhe des Teilbaums mit dem oberen Auftreten von A minimal ist, B bezeichnet den gesamten Ableitungsbaum ab dem A zum erstenmal auftritt.



B enthält außer A nur paarweise verschiedene Symbole aus V_N auf jedem Wege zu einem Blatt. Daraus folgt $h(B) \leq n+1$ und daraus folgt $l(VWX) \leq q$. Falls $VX = \epsilon$ wäre, so würde damit folgen $A \stackrel{*}{\Longrightarrow} A$, das heißt aus A läßt sich A nichttrivial ableiten, was aber nicht sein kann, da eine Grammatik in Chomsky-Normalform vorliegt.

Somit ist der Satz bewiesen, denn unter den gemachten Voraussetzungen des Beweises gilt für alle $Z = UVWXY \in L(G)$, daß auch $Z = UV^iWX^iY \in L(G)$ für alle i ist (vergleiche dazu folgende Abbildung).



Satz

$L := \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist keine kontextfreie Sprache.

$M := \{a^n b^n \mid n \in \mathbb{N}\}$ ist eine kontextfreie Sprache.

Beweis

Zum Beweis sei $Z := a^n b^n c^n$ wobei n genügend groß ist, so daß mit $Z = UVWXY$ auch

$$UV^iWX^iY \in L \quad \text{für alle } i$$

ist.

Weiterhin muß (vergleiche Pumping-Lemma) $VX \neq \epsilon$ sein.

- Wir betrachten zunächst $V \neq \epsilon$.
  - i) $V = a^k b^l c^m$ wobei $k, l, m \in \mathbb{N} \setminus \{0\}$. In $V^2 = a^k b^l c^m a^k b^l c^m$ tritt a hinter b und c auf, daraus folgt, daß V^2 kein Teilwort eines Wortes aus L ist.
  - ii) $V = d^k e^l$ wobei $d, e \in \{a, b, c\}$ daraus folgt, daß V^2 kein Teilwort eines Wortes aus L ist.
  - iii) $V = e^k$ wobei $e \in \{a, b, c\}$ nur dann ist V^i ein Teilwort eines Wortes aus der Sprache von L für alle i.
- Ebenso schließt man aus $X \neq \epsilon$, daß nur für $X = d^l$ wobei $d \in \{a, b, c\}$ X^i ein Teilwort eines Wortes aus der Sprache von L für alle i ist.

Damit folgt insgesamt, daß $Z = Ue^kWd^lY$. Daraus folgt die Existenz eines $f \in \{a, b, c\}$ mit $f \neq e$ und $f \neq d$, das in $U(e^k)^iW(d^l)^iY$ mit der gleichen Potenz wie in Z auftritt. Daraus folgt

$$U(e^k)^iW(d^l)^iY \notin L$$

für alle $i > 1$.

Folgerung

Mit kontextfreien Grammatiken läßt sich $L = \{a^n b^n c^n \mid n \in I\!N\}$ nicht formulieren!

Man kann eine kontextfreie Grammatik nicht dadurch erhalten, daß man den Schnitt von kontextfreien Sprachen bildet. Dies ist die Aussage des folgenden Satzes.

Satz

Seien L_1 und L_2 kontextfreie Sprachen. Dann braucht $L = L_1 \cap L_2$ keine kontextfreie Sprache zu sein.

Beweis

Sei $L_1 = \{a^n b^n c^m \mid n \in I\!N,\ m \in I\!N\}$ und $L_2 = \{a^m b^n c^n \mid m \in I\!N, n \in I\!N\}$ dann ist

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \in I\!N\}$$

nicht kontextfrei nach vorherigem Satz, L_1 und L_2 hingegen sind kontextfrei, denn

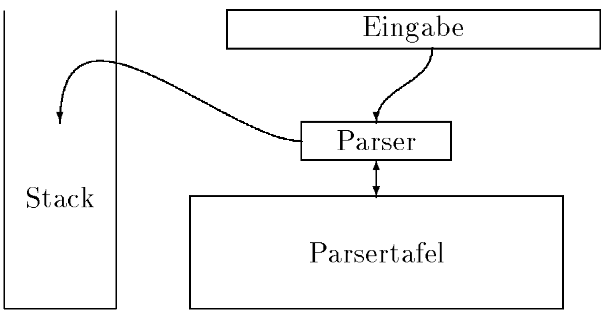
$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb|\varepsilon \\ C &\longrightarrow CC|c|\varepsilon \end{aligned}$$

ist eine kontextfreie Grammatik G mit $L(G) = L_1$. Eine Ableitungsfolge dieser Grammatik G sieht folgendermaßen aus

$$\begin{aligned} S &\longrightarrow AC \\ &\Longrightarrow aAbC \\ &\Longrightarrow aAbCC \\ &\Longrightarrow aaAbbCC \\ &\Longrightarrow aaAbbCCC \\ &\Longrightarrow aaAbbCCc \\ &\vdots \\ &\Longrightarrow a^i b^i c^j \end{aligned}$$

Mit einer ähnlichen Grammatik läßt sich zeigen, daß auch L_2 kontextfrei ist.

Push-Down-Automaten



Der in obiger Abbildung dargestellte Push-Down Automat kann eine schnelle Syntaxanalyse vornehmen. Man kann aber nicht für jede kontextfreie Sprache einen solchen determinierten Push-Down-Automaten bilden, der genau diese Sprache erkennt. Dies ist nur bei Verzicht auf Eindeutigkeit möglich.

4.5 Satz (Coke, Kasami, Younger)

Ist L eine kontextfreie Sprache, $\epsilon \notin L$ und $L \subseteq \Sigma^*$, so gibt es einen Algorithmus, der entscheiden kann, ob ein gegebenes Wort $w \in \Sigma^*$ in L liegt.

Beweis

Ohne Beschränkung der Allgemeinheit sei L Sprache einer Grammatik G, G sei dabei kontextfrei und in Chomsky-Normalform. Im folgenden konstruieren wir zu

$$w = a_1 a_2 \ldots a_n$$

alle Teilwörter die mittels Regeln aus G zu w abgeleitet werden können. Die Realisierung dieser Konstruktion erfolgt rekursiv.

Definiere dazu

$$M_{ij} := \{A \in V_N \mid A \stackrel{*}{\Longrightarrow} a_{i+1}a_{i+2}\ldots a_j\} \qquad \text{für } 1 \leq i < j \leq n.$$

Diese M_{ij} lassen sich rekursiv beschreiben, denn es gilt

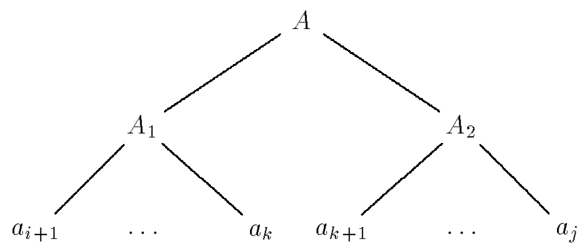
$$A \in M_{ij}$$

genau dann, wenn eine Regel $A \longrightarrow A_1 A_2 \in R$ existiert mit

$$A_1 \in M_{ik} \qquad \text{und} \qquad A_2 \in M_{kj}$$

mit einem $k \in I\!N$, das heißt

$$M_{ij} = \{A \in V_N \mid A \longrightarrow A_1 A_2 \in R \text{ und es existiert ein } k \in I\!N \text{ mit } A_1 \in M_{ik} \text{ und } A_2 \in M_{kj}\}.$$



Konstruiere nun zunächst $M_{i\,i+1}$ für $0 \leq i \leq n-1$. Es ist

$$M_{i\,i+1} = \{\, A \in V_N \mid A \longrightarrow a_{i+1} \in R \,\}.$$

Mittels obiger Äquivalenz konstruiere nun hieraus rekursiv M_{0n}.

Es ist

$$S \in M_{0n} \Longleftrightarrow w \in L(G).$$

Beispiel

Betrachte die folgende Grammatik

$$R = \{S \longrightarrow AB,\ A \longrightarrow a,\ A \longrightarrow CD,\ B \longrightarrow CD,\ C \longrightarrow b,$$
$$B \longrightarrow c,\ C \longrightarrow a,\ A \longrightarrow AB,\ B \longrightarrow AA,\ D \longrightarrow c,\ D \longrightarrow b\}.$$

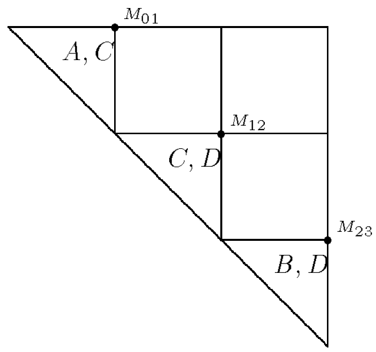
Wir wollen entscheiden, ob

$$w = abc \in L(G)$$

ist. Es ist

$$M_{01} = \{A, C\}, \quad M_{12} = \{C, D\}, \quad M_{23} = \{B, D\}.$$

Siehe Abbildung.

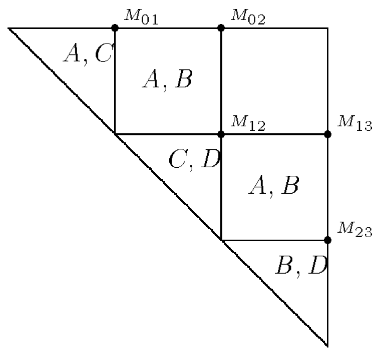


Aus M_{01} und M_{12} konstruieren wir M_{02}. Die Regeln $A \longrightarrow CD$ und $B \longrightarrow CD$ liefern

$$M_{02} = \{A, B\}.$$

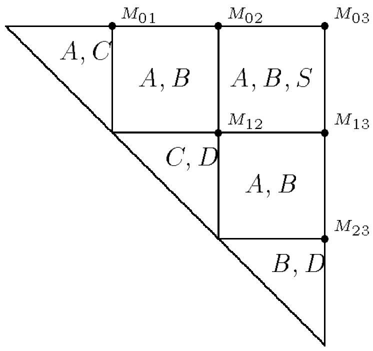
Aus M_{12} und M_{23} erhalten wir

$$M_{13} = \{A, B\}$$

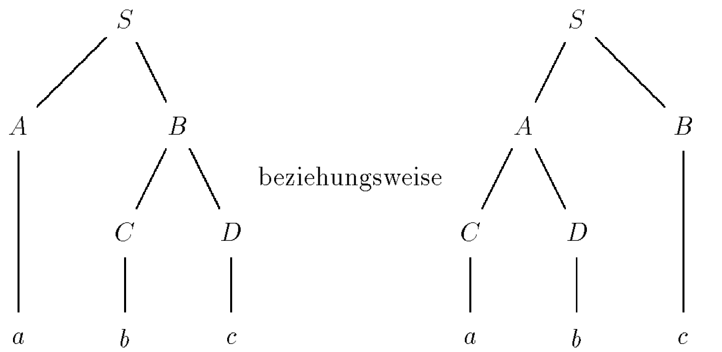


Aus M_{01} und M_{13} sowie M_{02} und M_{23} erhalten wir schließlich

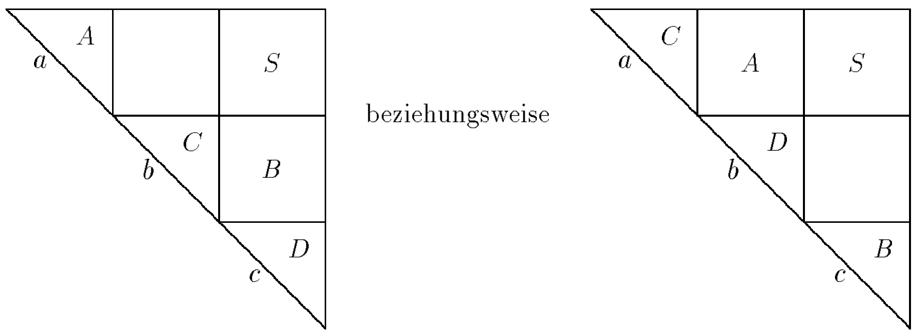
$$M_{03} = \{S, A, B\}$$



Aus diesen Konstruktionsschritten erhalten wir die beiden Ableitungsbäume in der folgenden Abbildung



die den beiden Rekursionsschemata



entsprechen.

4.6 Definition

Eine Sprache L heißt entscheidbar, wenn es einen Algorithmus gibt, der zu jedem gegebenen w entscheidet, ob $w \in L$ gilt oder nicht.

Die kontextfreien Sprachen — das heißt Sprachen, die sich in Chomsky-Normalform bringen lassen — sind nach **4.5** entscheidbar.

Es gibt jedoch auch Sprachen, die nicht entscheidbar sind. Wir betrachten dazu das "Wortproblem" aus der *Gruppentheorie*.

Wortproblem

Das Wortproblem ist nicht entscheidbar.

G ist eine Gruppe definiert durch Erzeuger $X_1, X_2, \ldots, X_n$ und Relationen. (Zum Beispiel

$$X_1X_2X_1X_2X_2 = id;$$
$$X_iX_j = X_iX_{j+1}X_{j+2};)$$

Sei nun w_0 als Wort in den X_i gegeben. Die Frage, ob $w_0 = id$, ist nicht entscheidbar. Es gibt aber eine

Allgemeine Entscheidungsstrategie

Liste alle $w \in L$ auf und prüfe, ob das gegebene Wort w_0 vorkommt.

- falls w_0 auftritt, so sind dazu nur endlich viele Schritte nötig.
- falls w_0 nicht auftritt, so bricht dieses Verfahren nicht ab.

Wenn man außer L auch $\Sigma^* \setminus L$ auflisten kann, so sind in jedem Fall nur endlich viele Schritte nötig. Es ist zwar keine Schranke bekannt, aber das Verfahren terminiert.

Ein Aufzählungsalgorithmus realisiert eine Funktion

$$f: L \longrightarrow I\!N; \;\; w \mapsto i.$$

Die Funktion f muß daher eine *berechenbare Funktion* sein.

4.7 Satz

Sei L eine abzählbare Sprache. Jedes $w \in L$ definiere eine partielle Funktion

$$f_w: I\!N \dashrightarrow I\!N.$$

Dann gibt es eine totale Funktion

$$f: I\!N \longrightarrow I\!N,$$

so daß kein $w \in L$ diese totale Funktion definiert.

Beweis

Sei $w_1, w_2, \ldots, w_n, \ldots$ eine Aufzählung von L. Sei

$$f(n) := \begin{cases} 1 & \text{falls } f_{w_n}(n) = 0 \\ 0 & \text{sonst} \end{cases}$$

Wäre $f = f_{w_i}$, so wäre $f(i) = 1$. Es ist aber $f(i) = 1$ nur für $f_{w_i}(i) = 0$, also $f \neq f_{w_i}$.

Satz 4.7 gilt insbesondere für kontextfreie Sprachen.

Das Halteproblem

Gesucht ist ein Programm A, das zu jedem Programm P und jeder Eingabe E entscheidet, ob P bei E terminiert.

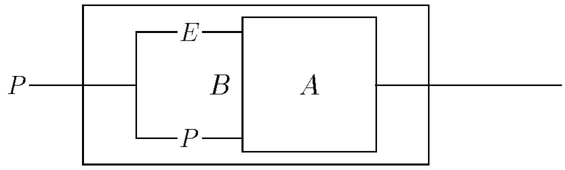
Satz

Es gibt kein solches A.

Beweis

Sei angenommen, daß A existiert.

Wir konstruieren ein spezielles Programm $P =$ "verrückt" und ein Programm B. Vergleiche dazu Abbildung 4.11.



Wir erhalten als Ergebnis "ja" falls $P(E)$ hält, und "nein" sonst. Ist $B(P)$ = ja, so läuft `verrückt`(P) in eine Endlosschleife.

Ist $B(P)$ = nein, so hält `verrückt`(P) an.

Berechne nun `verrückt(verrückt)`!

Angenommen `verrückt(verrückt)` hält, daraus folgt B(`verrückt`) = nein. Das heißt allerdings `verrückt(verrückt)` hält nicht an —Widerspruch.

Angenommen `verrückt(verrückt)` hält nicht, daraus folgt B(`verrückt`) = ja. Das heißt aber `verrückt(verrückt)` hält an —Widerspruch.

Damit folgt insgesamt, daß es kein solches A geben kann.

4.8 Satz

Jede Chomsky-Sprache ist aufzählbar.

Beweis

Sei S das Startsymbol, und es gebe keine Regel der Form $\epsilon \longrightarrow w$. Auf S werden iteriert Regeln angewandt. Entsteht eine Folge, die nur aus Terminalsymbolen besteht, so gehört das Wort zur Sprache der Grammatik. Andererseits wird auch jedes Wort der Sprache so erhalten. Bilde Mengen

$M_k = \{w \mid w \in V^*$ und w läßt sich aus S durch Anwenden von k Regeln erhalten, d.h. $S \overset{k}{\Longrightarrow} w\}$

für alle $k \in I\!N$ (wobei $V = V_N \cup V_T$).

Die Regelmenge wird angeordnet, das heißt die Wörter in M_k werden angeordnet erzeugt. Wird bei der Erzeugung von M_k ein Wort gefunden, das in V_T^* liegt, so gebe dies als Wort der Sprache aus.

Im Allgemeinen sind

$$ABCD \longrightarrow XYZXYZXYZX$$
$$ABC \longrightarrow X$$

mögliche Regeln. Die rechten Seiten können also sowohl kürzer als auch länger als die linken Seiten sein!

Das bedeutet bei der Aufzählung werden die Wörter fester Länge in keiner geordneten Reihenfolge erzeugt. Wir haben keine Schranke für die Anzahl der Schritte, die erforderlich sind, um zu entscheiden, ob ein gegebenes Wort zu der Sprache gehört oder nicht.

Motivation für das folgende Kapitel

Wir können aus dem Pumping-Lemma schließen, daß kontextfreie Grammatiken nicht ausreichen, um alle Eigenschaften von Programmiersprachen zu definieren.

5 Attributierte Grammatiken

Probleme

Die Variablen bekommen Typen zugewiesen, dabei sind Typprüfungen vorzunehmen. Das bedeutet, daß zum Beispiel die Operationen *Menge* + *Menge*, *Int* + *Int* oder *Int* + *Real* erlaubt sind, eine Operation wie *Menge* + *Int* jedoch nicht.

Beim Aufruf von Unterprogrammen sind ebenfalls die Typen zu vergleichen. Für Variablen muß ein Geltungsbereich festgelegt werden und vieles mehr.

Um solche Bedingungen zu formulieren werden *semantische Regeln* eingeführt.

Beispiel einer Grammatik mit semantischen Regeln

$$E \longrightarrow num\ \{E.type = int\}$$

$$E \longrightarrow num.num\ \{E.type = real\}$$

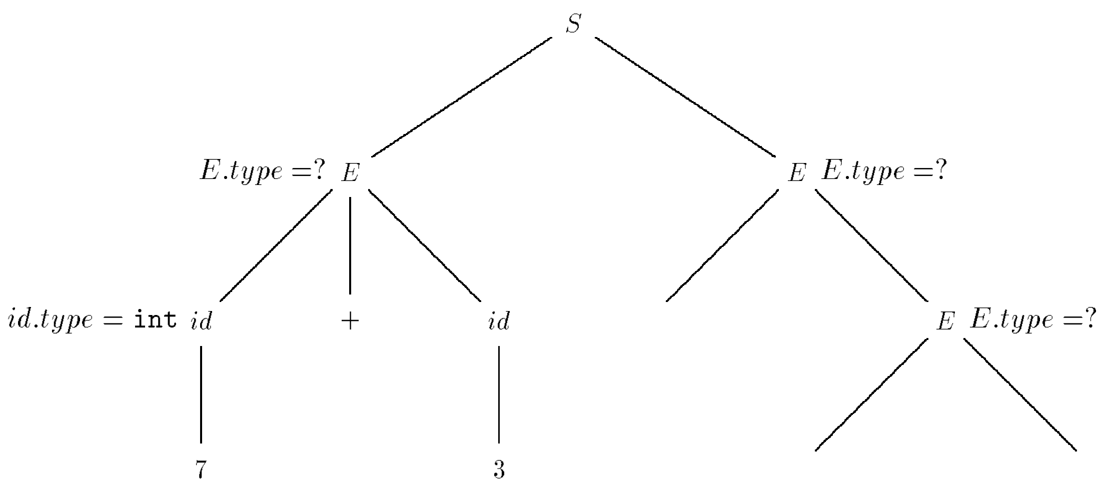
$$E \longrightarrow id\ \{E.type = \texttt{lookup}(id.type)\}$$

$$E \longrightarrow E_1 op E_2 \left\{ \begin{array}{l} E.type = \texttt{if}\ E_1.type == int\ \texttt{and}\ E_2.type == int \\ \qquad\qquad int \\ \qquad\qquad \texttt{if}\ E_1.type == real\ \texttt{and}\ E_2.type == real \\ \qquad\qquad real \\ \qquad\qquad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

Dabei stehen links die Grammatikregeln und in geschweiften Klammern die semantischen Regeln. Die Funktion lookup($id.type$) liest in der Symboltabelle den Typeintrag für den Identifier id.

Den Grammtiksymbolen sind Attribute zugeordnet —im obigen Beispiel war dies das Attribut *type*. Jede Eigenschaft (Attribut) besitzt einen Wertebereich. Eine semantische Regel schreibt vor, wie ein Attributwert zu berechnen ist.

Im Ableitungbaum für ein Wort der Sprache stehen an den Knoten Grammatiksymbole. Bei jedem Knoten sind die Attributwerte für dieses Grammatiksymbol an dieser Stelle zu berechnen.



Ein Attribut heißt *abgeleitet*, wenn sein Wert stets in jedem Ableitungsbaum aus den Attributwerten der Knoten im unter ihm liegenden Teilbaum bestimmt werden kann.

Attribute, deren Werte sich aus Attributwerten von Brüdern und des Vaterknotens bestimmen, heißen *vererbt*. Die vererbten Attribute beschreiben Kontextabhängigkeiten.

Beispiel für Typvereinbarung über vererbte Attribute

Zu Erkennen ist die Eingabe

$$\texttt{real} \quad id_1, id_2, id_3$$

Dies leistet die folgende attributierte Grammatik

$$\begin{aligned}
&D \longrightarrow TL\ \{L.in = T.type\}\\
&T \longrightarrow int\ \{T.type = int\}\\
&T \longrightarrow real\ \{T.type = real\}\\
&L \longrightarrow L_1, id \begin{Bmatrix} L_1.in = L.in \\ \texttt{addtype}(id.type, L.in) \end{Bmatrix}\\
&L \longrightarrow id\ \{\texttt{addtype}(id.type, L.in)\}
\end{aligned}$$

Dabei bedeutet "*in*" inherited, das heißt vererbt. Bei der Grammatikregel $L \longrightarrow L_1, id$ wurde L_1 mit einem Index versehen, um bei den semantischen Regeln das L links der Grammatikregel von dem rechts der Grammatikregel unterscheiden zu können. Die Funktion addtype$(id.type, L.in)$ legt für id auf $id.type$ den Wert $L.in$ in der Symboltabelle ab.

Den Ableitungsbaum für dieses Grammatikbeispiel ziegt die folgende Abbildung.

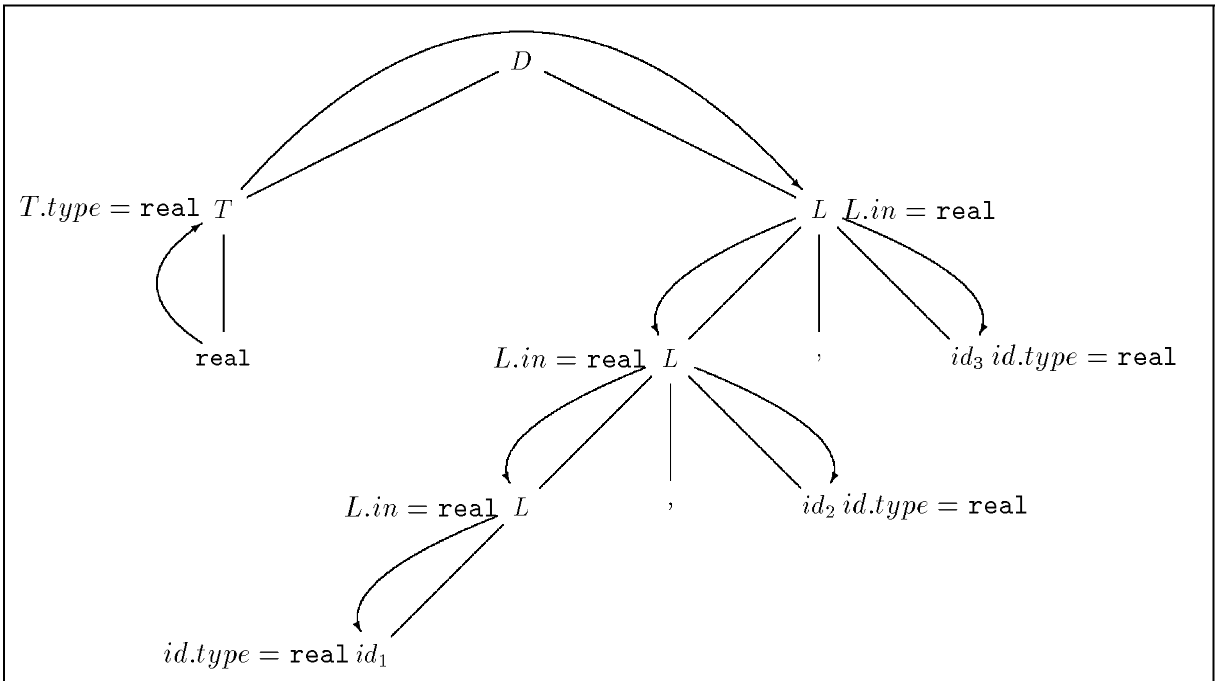


Abb. 5.2

5.1 Definition

Eine *attributierte Grammatik* ist ein Vier-Tupel

$$AG = (G, A, SR, B)$$

mit

1. $G = (V_T, V_N, R, S)$ ist eine kontextfreie Grammatik ohne überflüssige Symbole. Ist

$$r \colon x_0 \longrightarrow x_1 x_2 \ldots x_{n(r)}$$

eine Regel, so werden darin mehrfach vorkommende Grammatiksymbole durch Indizes unterschieden.

2. Die Attributmenge A ist definiert durch

$$A := \dot{\bigcup_{x \in V}} A(x).$$

Dabei ist $A(x)$ die Attributmenge von x. Jedes $a \in A$ besitzt einen Wertebereich $W(a)$. Jeder Regel

$$r\colon x_0 \longrightarrow x_1 x_2 \ldots x_{n(r)}$$

ist die Menge

$$A(r) = \bigcup_{i=0}^{n(r)} A(x_i)$$

zugeordnet.

3. Die endliche Menge von semantischen Regeln SR ist definiert durch

$$SR := \bigcup_{r \in R} SR(r).$$

Dabei gilt für die $SR(r)$

$$SR(r) \subseteq \Big\{ f\colon W(a_0) \times \cdots \times W(a_t) \longrightarrow W(a) \ \Big| \ t \geq 0 \text{ und für alle } i \text{ ist } a_i \in A(r) \text{ und } a \in A(r) \Big\}.$$

Die Funktionen f beschreiben, wie Attribute zu berechnen sind.

4. Die endliche Menge von Kontextbedingungen B ist definiert durch

$$B := \bigcup_{r \in R} B(r),$$

wobei

$$B(r) \subseteq \Big\{ f\colon W(a_0) \times \cdots \times W(a_t) \longrightarrow \{\texttt{wahr}, \texttt{falsch}\} \ \Big| \ t \geq 0 \text{ und für alle } i \text{ ist } a_i \in A(r) \Big\}.$$

Ist eine vorliegende Kombination von Attributwerten für eine Regel nicht erlaubt, das heißt es existiert ein $f \in B(r)$ das für diese Wertekombination `falsch` ausgibt, so erfolgt beim Berechnen der Attributwerte ein Fehlerabbruch.

Attributwerte können durch ein

$$f\colon W(a_0) \times \cdots \times W(a_t) \longrightarrow W(a)$$

nur dann berechnet werden, wenn alle Argumente bekannt sind. Somit muß die Reihenfolge der Attributberechnungen so organisiert werden, daß diese Bedingung erfüllt ist.

Um dies zu gewährleisten steht als Hilfsmittel der sogenannte Abhängigkeitsgraph zur Verfügung.

Definition (Abhängigkeitsgraph)

Die Knoten im Abhängigkeitsgraph sind Tupel

$$(K, a),$$

wobei K ein Knoten im Ableitungsbaum und a ein Attribut ist. Ein Pfeil

$$(K_1, a_1) \longrightarrow (K_2, a_2)$$

wird dann gezeichnet, wenn zur Berechnung des Wertes von a_2 bei K_2 der Wert von a_1 bei K_1 nötig ist.

Der Abhängigkeitsgraph ist so zu durchlaufen, daß jeder Knoten nach allen Vorgängern besucht wird. Falls der Abhängigkeitsgraph einen Zyklus enthält, ist keine Berechnung der Attributwerte möglich.

Topologisches Sortieren

Gegeben ist ein gerichteter azyklischer Graph (DAG). Gesucht ist eine Reihenfolge $(K_1, \ldots, K_n)$ der Knoten, für die gilt

$$\text{existiert ein } K_i \longrightarrow K_j, \text{ dann muß } i < j \text{ gelten.}$$

Dazu geht man wie folgt vor

1. Bestimme zu jedem Knoten die Anzahl der Vorgänger.

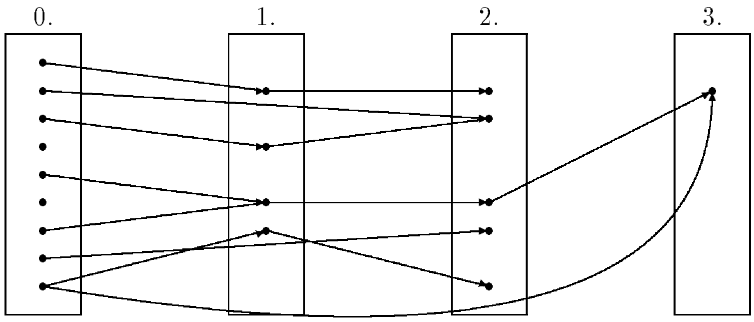
2. Gebe zuerst alle Knoten ohne Vorgänger aus.

3. Betrachte dabei alle Nachfolger zu jedem einzelnen dieser Knoten, und zähle bei jedem die Vorgängerzahl um 1 herunter.

Bilde die nächste Knotenschicht aus den Knoten, die nun Vorgängerzahl 0 haben. Gebe diese aus, falls noch Knoten existieren und gehe zu 3.. Sonst terminiere.

Für den Zugriff auf diese Knotenmengen verwalte Listen für Knoten gleicher Vorgängerzahl.

Die folgende Abbildung verdeutlicht die Methode beim topologischen Sortieren: In jeder Schicht liegen nur Knoten die mindestens einen Vorgänger in der vorherigen Schicht besitzen.



Nach dem topologischen Sortieren des Abhängigkeitsgraphen erhält man also eine Auswertreihenfolge für das konfliktfreie Berechnen der Attributwerte im Ableitungsbaum.

Heuristiken erlauben es, bereits beim Aufbau des Ableitungsbaumes die Auswertreihenfolge für die Attributwerte festzulegen, allerdings nicht in jedem Fall. Der Abhängigkeitsgraph wird bei diesen Heuristiken nicht explizit erstellt.

Datenstruktur im Ableitungsbaum

Der Ableitungsbaum enthält folgende Knoten

```
struct
{
  Grammatiksymbol  G;
  Nachfolgerzeiger N1 ... Nn;
  Attributwerte    A1 ... Am;
} node;
```

Die Datenstruktur sieht also wie folgt aus

$$(G, N_1, \ldots, N_n, A_1, \ldots, A_n)$$

Bei der statischen Implementierung hat man ein Feld von Strukturen `node`. Die Zeiger werden durch Platznummern in diesem Feld ersetzt.

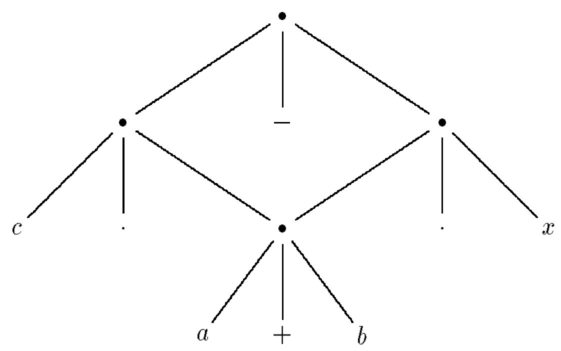
Bei der dynamischen Implementierung liegen aktuelle Zeiger auf die für die einzelnen Knoten belegten Speicherplätze vor. Dies hat natürlich den Vorteil, daß für jeden Knoten nur der notwendige Speicherplatz belegt wird.

Ausdrücke

Manche Teilausdrücke können natürlich mehrfach auftreten. Betrachten wir zum Beispiel

$$c \cdot (a + b) - (a + b) \cdot x,$$

so erhalten wir den in der Abbildung dargestellten gerichteten azyklischen Graphen.



Beachte, daß der in Abbildung 5.5 dargestellte Graph *kein* Baum ist, sondern ein gerichteter azyklischer Graph ist.

Falls der Knoten

$$(G, N_1, \ldots, N_n, A_1, \ldots, A_n)$$

bereits vorhanden ist, so kann er für jeden weiteren "Vaterknoten" als Sohn benutzt werden, der dieses Symbol G mit diesen Söhnen $N_1, \ldots, N_n$ und Attributwerten $A_1, \ldots, A_n$ braucht.

Dabei taucht das Problem auf, die Suche nach dem Auftreten von $(G, N_1, \ldots, N_n, A_1, \ldots, A_n)$ möglichst schnell zu gestalten. Deshalb verwendet man für die Verwaltung der Knoten die Hash-Technik, womit sich ein Suchaufwand $O(1)$ ergibt.

5.3 Bottom-up Auswertung für die Berechnung der Attributwerte

Wie uns aus **Kapitel 3.** bekannt, wird in der Auswertung bei Vorliegen einer Regel

$$A \longrightarrow X_1 \ldots X_k$$

bei aktueller Stackspitze

$$X_k \ldots X_1$$

die Folge der X_i durch A ersetzt.

Die X_i besitzen nun also zusätzlich Attributwerte. Diese können im Stack mit verwaltet werden. Abgeleitete Attribute von A lassen sich aus den Attributwerten berechnen, die unterhalb im Stack liegen.

Ist die rechte Seite nur teilweise bekannt, so liegt ein Wortanfang $X_i \ldots X_2 X_1$ oben im Stack.

Wird ein weiteres Zeichen X_{i+1} des Wortanfangs erkannt und in den Keller gelegt, so können zur Berechnung der Attributwerte dieses neuen Symbols bereits die Attributwerte der vorher im Wortanfang stehenden Symbole verwendet werden.

Dadurch sind einige vererbte Attribute bestimmbar.

5.4 Definition

Sei $A \longrightarrow X_1 \dots X_n$ eine Regel und zur Berechnung jedes Attributwertes von X_i existieren nur semantische Regeln, die von den Attributwerten der X_j mit $j < i$ abhängen. Dann heißt diese Regel *links-attributiert*.

Ist jede Regel eine Grammatik links-attributiert, so heißt die Grammatik *links-attributiert*.

Schreibweise

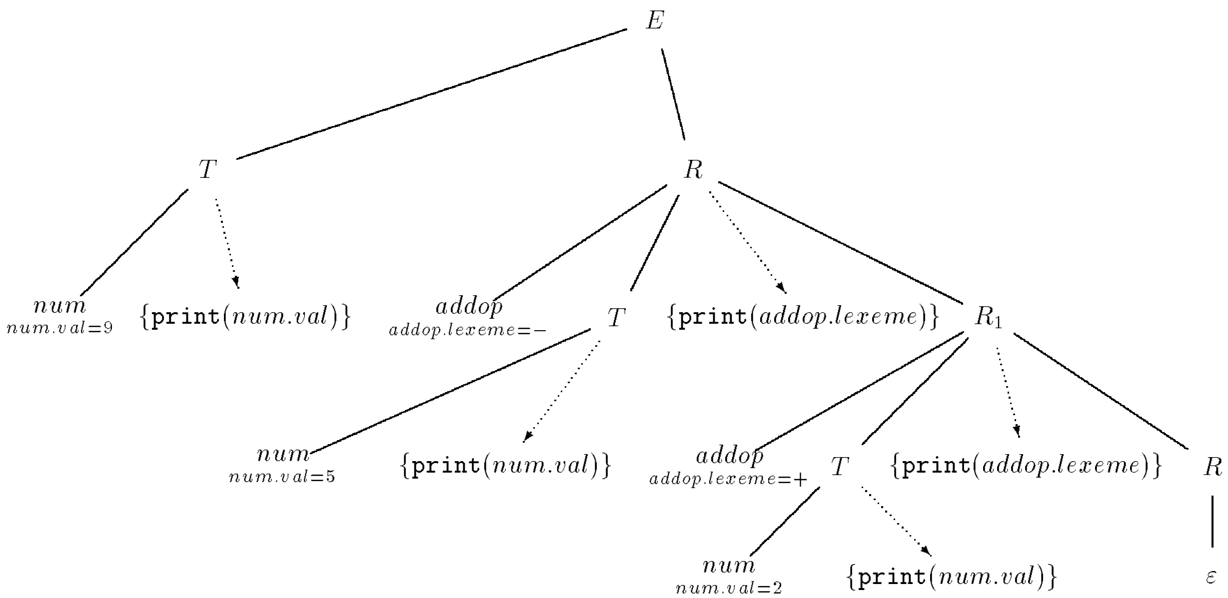
Um die Plätze festzuhalten, bei deren Erreichen eine spezielle semantische Regel jeweils auszuführen ist, werden diese Regeln direkt zwischen die Grammatiksymbole eingefügt.

Beispiel

Wir betrachten die folgende attributierte Grammatik

$$\begin{aligned}
&E \longrightarrow TR\\
&R \longrightarrow T\ \{\texttt{print}(addop.lexeme)\}\ R_1 | \varepsilon\\
&T \longrightarrow num\ \{\text{print}(num.val)\}
\end{aligned}$$

und erstellen den Ableitungsbaum für `9 - 5 + 2` (Infix-Notation).



Die "depth-first-Reihenfolge" liefert `9 5 - 2 +` (Postfix-Notation).

Diese Notation ist direkt umsetzbar in ausführbaren Code, indem jeder Operator auf die davor stehenden Operanden angewendet wird.

Die folgende Prozedur erledigt den Durchlauf und die Auswertung des Ableitungsbaumes in depth-first Reihenfolge.

```
void dfs(node n)
{
  durchlaufe die Söhne von n von links nach rechts mit m
  {
    berechne die vererbten Attribute von m;
    dfs(m);
  }
  berechne die Werte der abgeleiteten Attribute von n;
}
```

Um die Darstellung der Regeln zu vereinfachen, kann man zusätzliche Symbole einführen, die an die Stelle der semantischen Regeln treten. Bei der Ersetzung dieser Symbole durch ε wird die semantische Regel angewendet.

5.5 Berechnung von Attributwerten bei der Top-Down-Erstellung von Ableitungsbäumen

Erinnerung

In der gegebenen Grammatik war zuerst jede Linksrekursion zu entfernen, das heißt die Regel

$$A \longrightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_m,$$

wobei kein β_i mit A beginnt wird zu

$$\begin{aligned} A &\longrightarrow \beta_1 A'|\beta_2 A'|\ldots|\beta_m A' \\ A' &\longrightarrow \alpha_1 A'|\ldots|\alpha_n A'|\varepsilon \end{aligned}$$

wobei A' ein neues Symbol ist.

Beispiel für das Entfernen von Linksrekursionen in einer attributierten Grammatik

Die Regelmenge

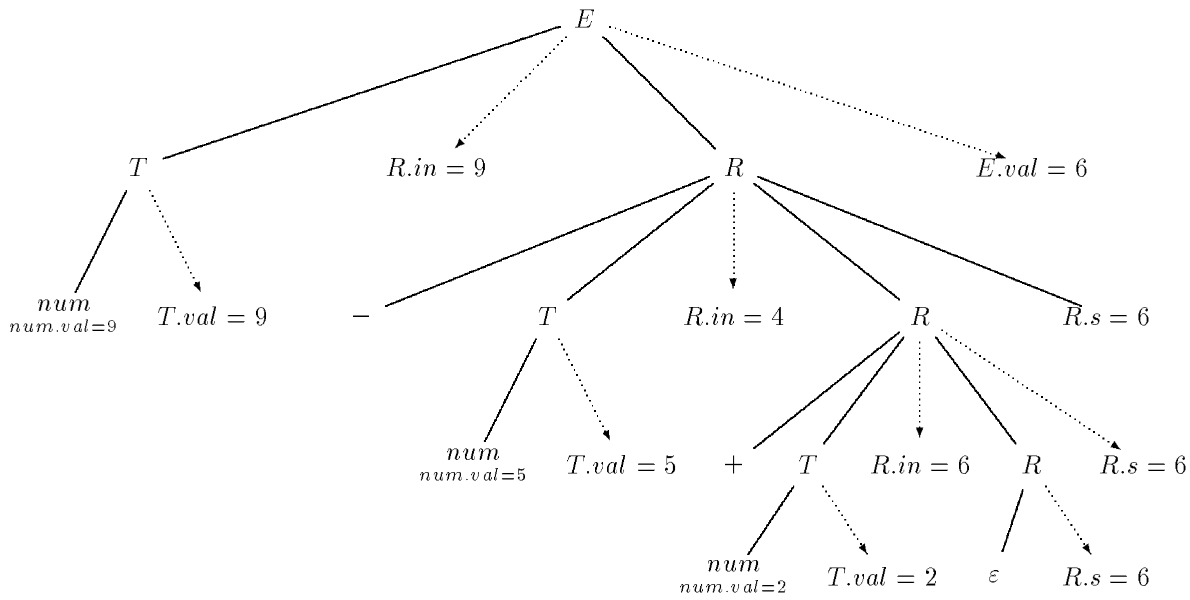
$$\begin{aligned} E &\longrightarrow E_1 + T\ \{E.val = E_1.val + T.val\} \\ E &\longrightarrow E_1 - T\ \{E.val = E_1.val - T.val\} \\ E &\longrightarrow T\ \{E.val = T.val\} \\ T &\longrightarrow (E)\ \{T.val = E.val\} \\ T &\longrightarrow num\ \{T.val = num.val\} \end{aligned}$$

mit Startsymbol E wird ersetzt durch die Regelmenge

$$\begin{aligned} E &\longrightarrow T\ \{R.in = T.val\}\ R\ \{E.val = R.s\} \\ R &\longrightarrow +T\ \{R_1.in = R.in + T.val\}\ R_1\ \{R.s = R_1.s\} \\ R &\longrightarrow -T\ \{R_1.in = R.in - T.val\}\ R_1\ \{R.s = R_1.s\} \\ R &\longrightarrow \varepsilon\ \{R.s = R.in\} \\ T &\longrightarrow (E)\ \{T.val = E.val\} \\ T &\longrightarrow num\ \{T.val = num.val\} \end{aligned}$$

mit dem Startsymbol E. Dabei steht das s in $R.s$ für "synthesized" (abgeleitet) und das in in $R.in$ für "inherited" (vererbt).

Die Abbildung zeigt den Ableitungsgraph für das Zahlenbeispiel `9 - 5 + 2`.



Sei $A \longrightarrow X_1X_2X_3 \ldots X_n$ eine beliebige Regel ohne Semantik, dann kann man sie wie folgt zu einer Regel mit Semantik auffüllen

$$A \longrightarrow \{\uparrow\}X_1\{\uparrow\}X_2 \ldots \{\uparrow\}X_n\{\uparrow\}$$

Dabei kann an jeder mit $\uparrow$ bezeichneten Stelle eine semantische Regel eingefügt werden.

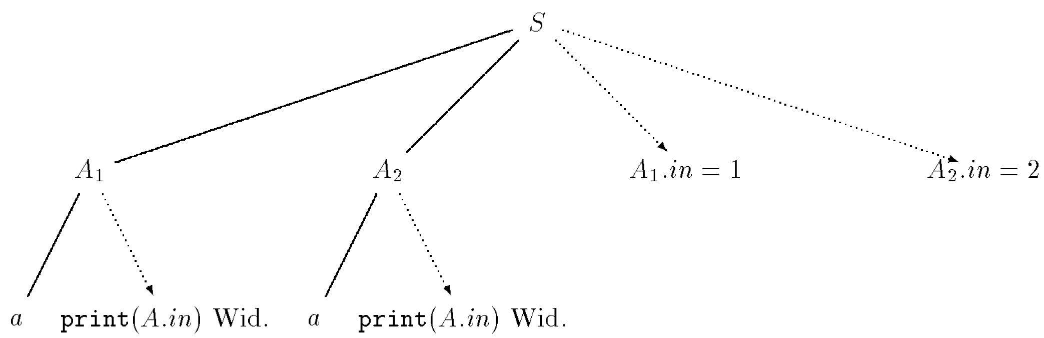
Den Symbolen sind Attribute zugeordnet. Semantische Regeln bestehen aus Programmcode zu deren Berechnung. Die Auswertreihenfolge wird über den Abhängigkeitsgraphen festgelegt, die Attribute werden bei der Erstellung des Ableitungsbaumes mitberechnet.

Stets müssen alle Werte bekannt sein, die für die Ausführung des Programmcodes für eine semantische Regel benötigt werden. Ist X_i auf der rechten Seite einer Regel, so stehen semantische Regeln zur Berechnung von vererbten Attributen vor X_i, und von abgeleiteten Attributen hinter X_i.

Wir betrachten die folgende attributierte Grammatik:

$$\begin{aligned} &S \longrightarrow A_1A_2\ \{A_1.in = 1\}\ \{A_2.in = 2\} \\ &A \longrightarrow a\ \{\texttt{print}(A.in)\} \end{aligned}$$

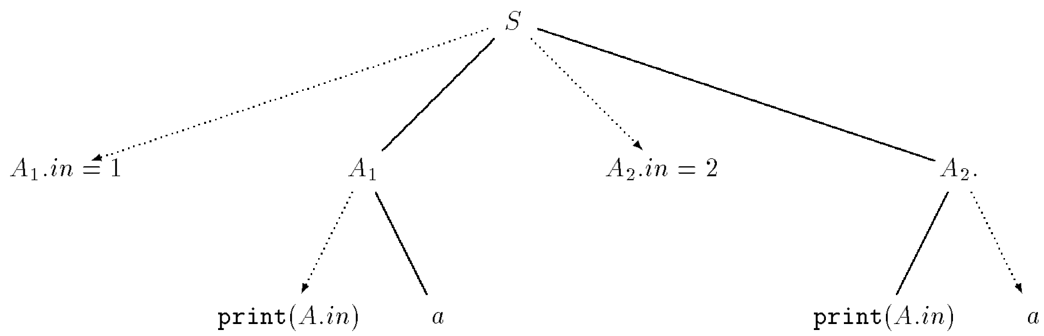
Bei der Ausgabe von $A.in$ ist dieser Wert noch gar nicht berechnet, was zu einem Fehler führt.



Korrekt könnte diese Grammatik lauten

$$\begin{aligned} &S \longrightarrow \{A_1.in = 1\}\ A_1\ \{A_2.in = 2\}\ A_2 \\ &A \longrightarrow \{\texttt{print}(A.in)\}\ a \end{aligned} .$$

Der Ableitungsbaum zu dieser Grammatik ist in der folgenden Abbildung dargestellt.



Nun ist beim Anwenden der Regel

$$A \longrightarrow \{\texttt{print}(A.in)\}\ a$$

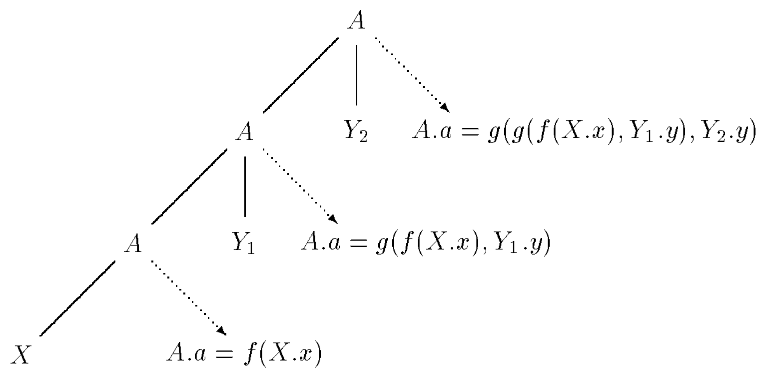
der Wert $A.in$ bekannt, die Grammatik ist also korrekt.

Beispiel für das Entfernen der Rekursion in einer Grammatik mit semantischen Regeln

Gegeben sei die folgende Grammatik

$$\begin{aligned} &A \longrightarrow A_1 Y\ \{A.a = g(A_1.a, Y.y)\} \\ &A \longrightarrow X\ \{A.a = f(X.x)\} \end{aligned}$$

daß heißt es liegt eine Linksrekursion vor, wodurch auch der Attributwert $A.a$ rekursiv zu berechnen ist.



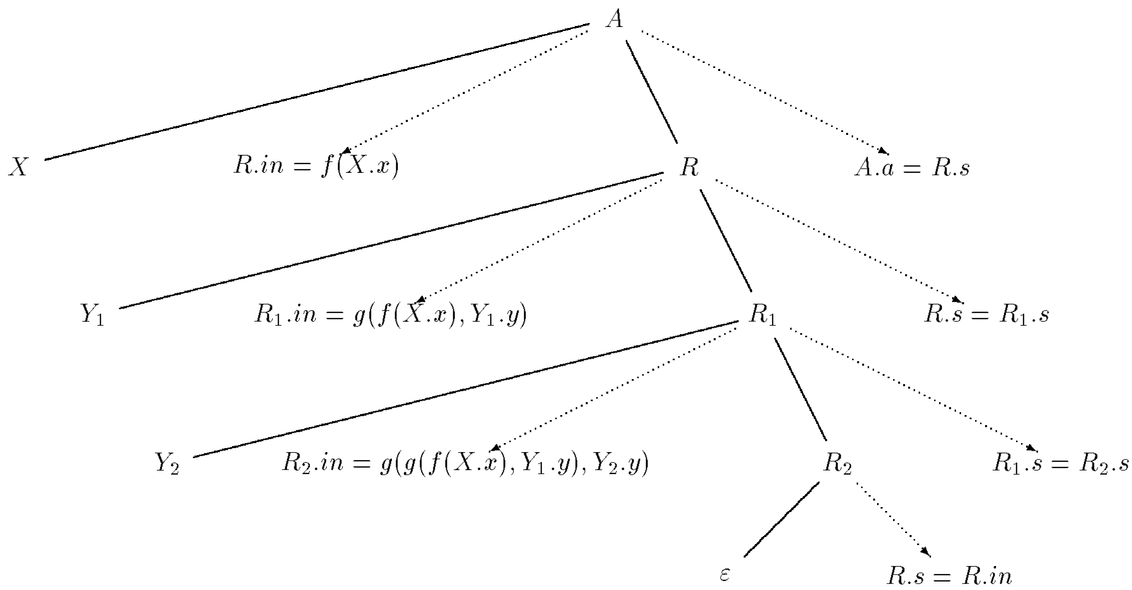
Die Linksrekursion wird zunächst auf übliche Weise durch Hilfsvariablen entfernt, das heißt wir erhalten folgende Grammatik (zunächst ohne semantische Regeln)

$$\begin{aligned} &A \longrightarrow XR \\ &R \longrightarrow YR_1 \mid \varepsilon \end{aligned} .$$

Die semantischen Regeln behandelt man nun wie folgt

$$A \longrightarrow X\ \{R.in = f(X.x)\}\ R\ \{A.a = R.s\}$$
$$R \longrightarrow Y\ \{R_1.in = g(R.in, Y.y)\}\ R_1\ \{R.s = R_1.s\}$$
$$R \longrightarrow \varepsilon\ \{R.s = R.in\}$$

Vergleiche dazu den entsprechenden Ableitungsbaum.



Die rekursiv zu berechnenden Attributwerte werden als vererbte Attribute der Hilfsvariablen berechnet. Wenn die Hilfsvariable verschwindet, wird jedes vererbte Attribut zu einem abgeleiteten Attribut. Dadurch werden die Attributwerte bei der Rückkehr im Baum an die Wurzel weitergegeben; die Rekursion ist entfernt.

5.6 Aufbau eines Parsers für Top-Down-Übersetzung mit Berechnung von Attributwerten

Eingabe

Linksattributierte LL(1)-Grammatik, bei der die semantischen Regeln in der Position auf der rechten Seite der Grammatikregeln angefügt sind, an der ihre Ausführung erfolgen soll.

Ausgabe

Computerprogramm.

Methode

1. Zu jedem $A \in V_N$ bilde eine Funktion, die in jedem vererbten Attribut von A einen formalen Eingabeparameter besitzt und die die Werte der abgeleiteten Attribute zurückgibt. (Ohne Beschränkung der Allgemeinheit existiere zu A genau ein abgeleitetes Symbol).
2. Entscheide wie bisher anhand vom vorliegenden Symbol im Stack und betrachteten Eingabezeichen, welche Grammatikregel zu wählen ist oder ob ein *Shift* auszuführen ist.
3. Ausführung der Attributberechnung:
   Die rechten Seiten der Regeln werden von links nach rechts durchlaufen. Es können Terminal-, Nichtterminalsymbole und semantische Regeln auftreten.

i) Das Terminalsymbol X habe das Attribut x. Dann lege den Wert von x auf den Platz der Variable $X.x$ ab. Der Zeiger auf den Eingabewert wird um eine Position weitergerückt.

ii) Ein Nichtterminalsymbol B habe die vererbten Attribute $b_1 \dots b_n$ und das abgeleitete Attribut c sei durch $c = f_B(b_1 \dots b_n)$ zu berechnen. Dann führe diese Berechnung durch.

iii) Bei einer semantischen Regel kopiere das Programm und ersetze jede Referenz auf ein Attribut durch die Variable für das Attribut. Führe das Programm aus.

5.7 Bottom-up-Verarbeitung vererbter Attribute

Beim Aufstellen eines Ableitungsbaumes wird dieser von links her erstellt, das heißt alle Werte aus dem linken Teilbaum sind bekannt. Diese Werte liegen im Stack an entsprechenden Plätzen. Formal existiert je ein vererbtes und ein abgeleitetes Attribut für jedes Symbol.

Wenn man jede Regel der Form

$$A \longrightarrow X_1 S_1 X_2 S_2 \dots,$$

wobei X_i für ein Grammatiksymbol und S_i für eine semantische Regel steht, ersetzt durch

$$\begin{aligned} A &\longrightarrow X_1 M_1 X_2 M_2 \dots \\ M_1 &\longrightarrow \varepsilon S_1 \\ M_2 &\longrightarrow \varepsilon S_2 \\ &\vdots \end{aligned}$$

stehen zwischen den Grammatiksymbolen keine semantischen Regeln mehr.

Eingabe

Linksattributierte LL(1)-Grammatik, bei der die semantischen Regeln in die Position auf der rechten Seite der Grammatikregeln eingefügt sind, an denen ihre Ausführung erfolgen soll.

Ausgabe

Computerprogramm

Methode

Jedes $A \in V_N$ besitze ein vererbtes Attribut $A.in$ und jedes Grammatiksymbol X ein abgeleitetes Attribut $X.s$.

Ist $X \in L$, so ist $X.s$ der von der lexikalischen Analyse hierfür ermittelte Wert. In jeder Regel $A \longrightarrow X_1 \dots X_n$ entferne die semantischen Regeln wie oben. $X_j.s$ wird im Stack direkt unterhalb von X_j abgelegt.

Zur Berechnung der Attributwerte:

1. Reduktion auf ein M_i. Die Produktionen $A \longrightarrow M_1 X_1 \dots M_j X_j \dots X_n$, zu der M_j gehört ist bekannt, das heißt die Position der Attributwerte im Stack sind bekannt, die zur Ausführung der semantischen Regel nötig ist, für die M_j steht. Daraus folgt, daß $X_j.in$ berechnet werden kann.
2. Reduktion mittels $A \longrightarrow M_1 X_1 \dots M_n X_n$. Nun wird nur $A.s$ berechnet, da $A.in$ bereits bekannt ist. Wiederum sind die benötigten Attributwerte dem Stack aus wohlbestimmten Positionen zu entnehmen.

Das heißt bei `shift`-Operationen werden vererbte und bei `reduce`-Operationen werden abgeleitete Attribute berechnet.

6 Typprüfungen

Man muß grundsätzlich unterscheiden zwischen statischen und dynamischen Typprüfungen. Die statischen Typprüfungen finden bereits zur Compilierzeit statt, die dynamischen erst zur Laufzeit des Programms. Deshalb sollten die statischen Typprüfungen möglichst gut sein.

Das Problem ist aber, daß statische Typprüfungen nicht immer möglich sind, etwa bei dynamisch vereinbarten Feldern.

6.1 Typen

Variablen besitzen Typen, Ausdrücke erhalten Typen, Programmsequenzen können Typen haben, wie zum Beispiel "korrekt" oder falsch.

Die Typen werden als Attribute behandelt. Der Wertebereich für das Attribut `Typ` ist rekursiv definiert anhand der Möglichkeiten der Sprache, komplexe Typen aus einfacheren zusammenzusetzen. Jeder unterschiedliche Typ wird durch einen Ausdruck beschrieben: *type_expression*.

1. Grundtypen sind type_expressions: boolean, integer, real, char, type_error, void, ... (von der jeweiligen Sprache abhängig).

2. Der Name für ein type_expression ist wieder type_expression.

3. Konstruktoren bilden aus type_expressions neue type_expressions

Felder

Sei der Indexbereich I und der Typ für Feldeinträge T, dann ist

$$\mathrm{Feld}(I, T)$$

eine type_expression. Zum Beispiel

```
LONG a[10];
```

Dann ist `LONG` der Typ für Feldeinträge und 10 der Indexbereich.

Produkte

Seien T_1 und T_2 type_expressions. Dann ist

$$T_1 \times T_2$$

eine type_expression.

Records

Hier haben die Komponenten eines Produkts Namen erhalten. Das heißt, wir erhalten Produkte der Form

$$(name_1 \times T_1) \times (name_2 \times T_2) \times \cdots \times (name_n \times T_n).$$

Dabei stehen die $name_i$ für einen Namen und die T_i für einen Typ. In der jeweiligen Programmiersprache formuliert man dies durch

$$\mathtt{record(name\_1 : T\_1, name\_2 : T\_2, \ldots)}$$

Zeiger

Ist T eine type_expression, so ist

$$\text{pointer}(T)$$

eine type_expression.

Funktionen

Ist der Urbildbereich D einer Funktion F eine type_expression, und ihr Bildbereich R eine type_expression, dann ist auch die Funktion

$$F\colon D \longrightarrow R$$

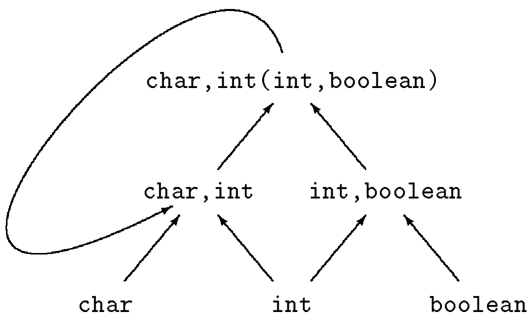
eine type_expression. Wir betrachten dazu das Beispiel

```
int *f( int a, int b)
```

Somit ist $D = int \times int$ und $R = int\ *$, und der Typ von f ist $int \times int \longrightarrow int\ *$. Dies kann natürlich beliebig kompliziert werden, wenn man zum Beispiel eine Abbildung auf eine Abbildung abbildet $((int \to int) \to (int \to int))$.

4. type_expressions dürfen Variablen beinhalten, die als Werte type_expressions annehmen. Es gibt Sprachen, die aus der Verwendung einer Variablen zurückschließen auf den Typ, den diese haben muß. Es wird nicht verlangt explizit einen Typ zu vereinbaren. Um den Typ zu erschließen wird er zunächst mit einer Typvariablen bezeichnet.

Zur Veranschaulichung und zur Implementation eignet sich ein gerichteter Graph. Der Aufbau des Graphen beginnt bei Grundtypen. Falls ein Konstruktor verwendet wird, so wird ein neuer Knoten für diesen erzeugt. Es werden Pfeile gelegt auf die Typen, die als Argumente des Konstruktors verwendet werden. Man erhält einen Typen-“Baum” beziehungsweise besser gesagt einen gerichteten Typengraph.



Attribute bei attributierten Grammatiken eignen sich für die statische Typprüfung. Die semantischen Regeln müssen dazu die Typzuweisungen vornehmen und jeweils prüfen, ob die gewünschte Verwendung einer Variablen mit der Typvereinbarung verträglich ist.

Beispiel für eine attributierte Grammatik mit Typprüfung

$$E \longrightarrow literal\ \{E.type = char\}$$
$$E \longrightarrow num\ \{E.type = int\}$$
$$E \longrightarrow id\ \{E.type = \text{lookup}(id.type)\}$$

$$E \longrightarrow E_1 \bmod E_2 \quad \left\{ \begin{array}{l} E.type = \texttt{if } E_1.type == int \texttt{ and } E_2.type == int \\ \qquad\qquad int \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$$E \longrightarrow E_1[E_2] \quad \left\{ \begin{array}{l} E.type = \texttt{if } E_2.type == int \texttt{ and } E_1.type == array(s,t) \\ \qquad\qquad t \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

Hier ist es möglich zu prüfen, ob der Wert von E_2 kleiner als der Wert von s ist, wenn eine feste Zahl für s auftritt. Wird dies nicht getan, so erhält man eventuell einen Laufzeitfehler.

$$E \longrightarrow *E_1 \quad \left\{ \begin{array}{l} E.type = \texttt{if } E_1.type == pointer(t) \\ \qquad\qquad t \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$*E_1$ ist der Inhalt von E_1 (Dereferenzierung). Im Folgenden steht S für Statement.

$$S \longrightarrow id = E \quad \left\{ \begin{array}{l} S.type = \texttt{if } id.type == E.type \\ \qquad\qquad void \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$$S \longrightarrow \text{if } E \text{ then } S_1 \quad \left\{ \begin{array}{l} S.type = \texttt{if } E.type == boolean \\ \qquad\qquad S_1.type \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$$S \longrightarrow \text{while } E \text{ do } S_1 \quad \left\{ \begin{array}{l} S.type = \texttt{if } E.type == boolean \\ \qquad\qquad S_1.type \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$$S \longrightarrow S_1; S_2 \quad \left\{ \begin{array}{l} S.type = \texttt{if } S_1.type == void \texttt{ and } S_2.type == void \\ \qquad\qquad void \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

$$\begin{array}{l} P \longrightarrow D; E \\ D \longrightarrow D; D | id : T \\ T \longrightarrow char | int | array[num] \ of \ T | *T \\ E \longrightarrow literal | num | id | E \bmod E | E[E] | *E \end{array}$$

Grundtypen sind $char$, int, $type\_error$ und Konstruktoren $array[num] \ of \ T$, $T*$.

$$D \longrightarrow id : T \ \{\text{addtype}(id.type, T.type)\}$$

Das heißt in der Symboltabelle wird bei id der Typ von T abgelegt

$$\begin{array}{l} T \longrightarrow char \ \{T.type = char\} \\ T \longrightarrow T_1 * \ \{T.type = pointer(T_1.type)\} \\ T \longrightarrow array[num] \ of \ T_1 \ \{T.type = array(1 \ldots num.val, T_1.type)\} \end{array}$$

Typprüfung bei Funktionen

$E(E)$ stehe für die Anwendung eines Ausdrucks auf einen Ausdruck und T sei ein Funktionstyp. Dann sieht die Typprüfung bei Funktionen wie folgt aus

$$T \longrightarrow T_1 \mapsto T_2 \; \{T.type = T_1.type \mapsto T_2.type\}$$

$$E \longrightarrow E_1(E_2) \; \left\{ \begin{array}{l} E.type = \texttt{if } E_2.type == s \texttt{ and } E_1.type == s \mapsto t \\ \qquad\qquad t \\ \qquad\quad \texttt{else} \\ \qquad\qquad type\_error \end{array} \right\}$$

Beispiel

```
real root(real f(real),real x)
```

Somit ist die Funktion `root` definiert durch

$$\texttt{root}: ((real \to real) \times real) \to real$$

6.2 Wann sind zwei Typen gleich?

Sollen die Indexgrenzen bei Feldern übereinstimmen, damit zwei Typen gleich sind? Man kann Typen Namen geben. Sind gleich aufgebaute Ausdrücke von Typen mit verschiedenen Namen gleich?

Beispiel

```
struct node
{
   struct node *next;
   struct node *last;
};

struct node *p;
struct node *q,*r;
```

Es stehen nun drei Alternativen zur Auswahl

1. Alle Variablen haben den gleichen Typ.
2. Jeder Typname („`struct node`") bezeichnet einen eigenen Typ, das heißt `p`,`q`,`r` haben den gleichen Typ.
3. Intern werden für jeden auftretenden Typausdruck Typennamen erzeugt. Das heißt der Typ von `p` ist verschieden vom Typ von `q`,`r` und wiederum verschieden von `struct node`.

6.3 Test auf strukturelle Äquivalenzen

Die folgende Funktion `sequiv`() vergleicht Typen-Bäume.

```
boolean sequiv(S_1,S_2)
{
   if S_1 and S_2 are of the same basistype
      return (TRUE);
   else if S_1 = array(I_1,T_1) and S_2 = array(I_2,T_2)
      return (sequiv(T_1,T_2) and sequiv(I_1,I_2));          (*)
   else if S_1 = K × L and S_2 = M × N
      return (sequiv(K,M) and sequiv(L,N));
   else if S_1 = pointer(K) and S_2 = pointer(L)
      return (sequiv(K,L));
   else if S_1 = K ↦ L and S_2 = M ↦ N
      return (sequiv(K,M) and sequiv(L,N));
   ⋮
   else
      return (FALSE);
}
```

Statt $(*)$ kann auch

```
      return (sequiv(T_1,T_2));
```

verwendet werden, wenn bei Array die Indexgrenzen nicht geprüft werden sollen.

6.4 Handhabung variabler Typen

1. Explizite Fallunterscheidung der verschiedenen Kombinationsmöglichkeiten von Typen etwa bei "+"

$$int + int; \quad int + real; \quad \text{Menge} + \text{Menge}; \quad int + \text{Menge}$$

Letzteres führt natürlich zu einem Fehler.

2. Man kann einem Attribut *type* auch eine Menge von möglichen erlaubten Typwerten zuweisen. Bei Konstruktoren werden die Typmengen auf die zulässigen Kombinationen eingeschränkt. Ein abgeschlossener Ausdruck, der einer Variablen zugewiesen wird, muß allerdings einen eindeutigen Typ haben. Beispielweise in der Programmiersprache *Ada* verwendet.

3. Typvariable einsetzen. Es gibt folgende Implementierungsmöglichkeiten für den Typenvergleich

"Fest verdrahtet"

Der Typenvergleich erfolgt durch einer Reihe von `if`-Anweisungen und `case`-Anweisungen.

"Dynamischer Baum"

Die Kombination der Typen sind Werte, der Ergebnistyp ist der Infoteil eines bestimmten Knotens.

Für Expertensysteme ist es wichtig, daß zur Laufzeit neue Typen erzeugt und Typprüfungen vorgenommen werden.

6.5 Typ-Variable

Programmausschnitt zur Berechnung der Länge einer Liste in C.

```
struct Node
{
   struct Node *next;
           int  info;
};

int length(struct node *lptr)
{
   int len = 0;

   while (lptr)
   {
      len++;
      lptr = lptr->next;
   }
   return (len);
}
```

Hierbei ist die Typprüfung so streng, daß `length()` nur für Listen angewendet werden kann, bei denen im Info-Teil `int`-Werte stehen.

Für Listen von Namen müßte die Funktion also neu vereinbart werden.

Es ist aber auch möglich die Typprüfung völlig zu ignorieren, womit man natürlich wiederverwendbare Routinen und Programmteile erhält, falls die Typen jedoch nicht passen, erhält man einen Laufzeitfehler. Dazu das

Beispiel

```
void qsort(void *base, size_t count, size_t size, int (*compare());
```

Dabei ist `base` der Zeiger auf das erste Element des zu sortierenden Feldes. `count` enthält die Anzahl der Elemente im Feld, `size` die Größe eines einzelnen Elements in Bytes. Und `compare` ist der Zeiger auf die Vergleichsfunktion die aufgerufen wird, und wie folgt definiert sein muß

```
int ⟨name⟩(e_type *elem1,e_type *elem2);
```

Der Rüchkgabewert r der Vergleichsfunktion ist definiert durch

$$r := \begin{cases} < 0, & \text{falls } \texttt{*elem1 < *elem2} \\ = 0, & \text{falls } \texttt{*elem1 == *elem2} \\ > 0, & \text{falls } \texttt{*elem1 > *elem2} \end{cases}$$

Damit ist es zwar möglich verschiedene Felder mit derselben Funktion zu sortieren, aber falls die angegebene Vergleichsfunktion Typen benötigt, die nicht den Feldtypen entsprechen, so erfolgt ein Laufzeitfehler.

Bei Verwendung einer Typvariablen könnte dagegen bereits statisch geprüft werden, ob die Typen stimmen.

Funktionen, die für viele Typen von Argumenten verwendbar sind, heißen *polymorph*. Solche Funktionen kann man mit der 1984 entwickelten Programmiersprache ML schreiben.

```
func length(lptr)=
   if null(lptr) then
      0
   else
      length(tl(lptr))+1;
```

Dabei testet die Funktion `null()` auf leer und `tl` entfernt das erste Element einer Liste und gibt den Rest zurück.

Beispiel einer Grammatik für eine Sprache für polymorphe Funktionen

$$
\begin{aligned}
&P \longrightarrow D; E\\
&D \longrightarrow D; D | id : Q\\
&Q \longrightarrow \forall \text{type\_variable}.Q | T\\
&T \longrightarrow T \mapsto T || T \times T | \text{unary\_constructor}(T) | \text{basic\_type} | \text{type\_variable} | (T)\\
&E \longrightarrow E(E) \ | \ E, E \ | \ id
\end{aligned}
$$

Typ-Variable müssen bei jedem Auftreten durch aktuelle Typen ersetzt werden. Dabei kann die Ersetzung jeweils anders sein. Falls dieselbe Variable in einem Ausdruck mehrfach auftritt, so muß sie dort gleich ersetzt sein. (Gültigkeitsbereich der Typvariablen ist festzulegen).

Typvariable können einen Wert durch die Verwendung erhalten, daraus folgt automatische Deduktion (Erkennung) des Typs ist möglich. So ist zum Beispiel die Bestimmung des Rückgabewertes einer Funktion anhand der Ausführung des Codes bei bestimmten Daten möglich.

Prüfung auf Konsistenz bei der Verwendung eines Types in einem Ausdruck

```
deref(p)
{
   return (*p)
}
```

Dabei muß p vom Pointer-Typ sein, mit anderen Worten wenn α eine Typvariable ist, dann ist `p.type = pointer(`α`)`. Somit wird

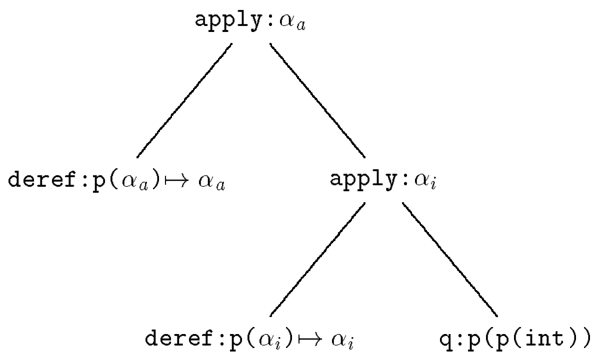
```
deref.type = pointer(α) ↦ α
```

für alle Typvariablen α.

Sei die Funktion `deref()` wieder wie oben, das heißt sie liefert zu jeder Typvariablen vom Pointertyp den Basistyp auf den diese Typvariable zeigt. Beispielsweise sei nun die Typvariable `q` wie folgt definiert

```
q: pointer(pointer(int));
```

dann ist `deref(deref(q))` den Typ `int`. Diese zweimalige Anwendung der Dereferenzierung (im folgenden mit `apply` abgekürzt) wird mithilfe des Ableitungsbaumes verdeutlicht. Weil hier `apply` zweimal ausgeführt wird, werden zwei Typvariablen benötigt: α_a (aussen) und α_i (innen). Wir erhalten als Ableitungsgraphen für `deref(deref(q))`:



Es wird also zunächst `q` durch `apply` einmal dereferenziert, das heißt α_i ist der Typ `pointer(int)`. Durch das zweite `apply` wird α_a der Typ `int`.

Was heißt Äquivalenz von Typen?

Angenommen E_1 hat Typ $s \mapsto s'$ und wird auf E_2 vom Typ t angewandt.

Da nun s, t Typvariable sein können, müssen wir diese *unifizieren*, das heißt ihren größten gemeinsamen Typ bestimmen. In obigem Beispiel ist α und `int` zu unifizieren zu `int`. Eine erfolgte Typersetzung muß bei jedem Auftreten der Typvariablen durchgeführt werden.

Wenn man jeder Typ-Variablen eine type_expression zuweist, so nennt man dies eine *Substitution*. Die Substitution kann auf einen Ausdruck angewendet werden, indem die Variablen bei jedem Auftreten substituiert werden.

```
type_expression subst(type_expression t)
{
   if t is a basis-type
      return (t);
   else if t is a variable
      return (s(t));
   else if t is a t1 ↦ t2
      return (subst(t1) ↦ subst(t2));
}
```

Dabei ist $s(\cdot)$ die Substitution. $s(t)$ heißt dann *Instanz* (Beispiel) für t.

$$
\begin{array}{rl}
int \mapsto int & \text{ist Instanz für } \alpha \mapsto \alpha \\
int \mapsto real & \text{ist keine Instanz für } \alpha \mapsto \alpha \\
\alpha \mapsto real & \text{ist ebenso keine Instanz für } \alpha \mapsto \alpha
\end{array}
$$

Man kann t_1 und t_2 *unifizieren*, wenn es eine Substitution s gibt mit

$$s(t_1) = s(t_2).$$

Beispiel

Gegeben seien die beiden *type_expressions* t_1 und t_2 wie folgt

$$t_1 : \Big(((\alpha_1 \mapsto \alpha_2)) \times list(\alpha_3) \Big) \mapsto list(\alpha_2)$$

$$t_2 : \Big(((\alpha_3 \mapsto \alpha_4)) \times list(\alpha_3) \Big) \mapsto \alpha_5$$

Man untersuche die beiden *type_expressions* auf strukturelle Äquivalenz, das heißt man finde eine Substitution s, so daß $s(t_1) = s(t_2)$.

Definiere dazu s_1 und s_2 wie folgt

Unifikatoren	$s_1(\alpha)$	$s_2(\alpha)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	$list(\alpha_2)$	$list(\alpha_1)$

Einsetzen liefert sofort $s_1(t_1) = s_1(t_2)$ und $s_2(t_1) = s_2(t_2)$. Man kann also t_1 und t_2 unifizieren, da wir sogar zwei Substitutionen gefunden haben. Man beachte, s_1 "allgemeiner" als s_2 ist.

Definition (Allgemeinster Unifikator)

Man bezeichnet die Substitution, die am wenigsten Einschränkungen vornimmt als den *allgemeinsten Unifikator*. Genauer heißt das

1) $s(t_1) = s(t_2)$
2) Für alle anderen Substitutionen s' mit $s'(t_1) = s'(t_2)$ ist s' ein Beispiel für s, das heißt für alle t ist $s'(t)$ ein Beispiel für $s(t)$.

Der allgemeinste Unifikator ist somit eindeutig bis auf Namensgebung.

Es gibt einen Algorithmus um den Allgemeinsten Unifikator zu berechenen. Dazu werden benötigt

1) `fresh(t)`

   Dabei ist t eine Typ-Variablen ist. Einige dieser Typ-Variablen sind möglicherweise durch $\forall$-Kennzeichnungen an diesen Ausdruck gebunden. (Beispiel: in $\sum_{i=1}^{n} i$ ist das i an die Summe gebunden, außerhalb dieser Summe ist eine Zuweisung wie $l = i$ Unsinn).

   Solche gebundenen Variablen werden durch eine neue bisher noch nicht aufgetretene Variablen ersetzt. Danach werden die $\forall$-Symbole weggelassen.

   Beispiel

$$\forall\alpha_1\forall\alpha_2((\alpha_1 \mapsto \alpha_2) \times list(\alpha_3)) \quad \text{beziehungsweise} \quad \forall\alpha_1\forall\alpha_2((\alpha_2 \mapsto \alpha_1) \times list(\alpha_4))$$

   wird ersetzt durch

$$((\alpha_5 \mapsto \alpha_6) \times list(\alpha_3)) \quad \text{beziehungsweise} \quad ((\alpha_7 \mapsto \alpha_8) \times list(\alpha_4)).$$

2) `unify(m,n)`

   Diese Funktion unifiziert die type_expressions auf die die Knoten n und m des Ableitungsbaumes für die Typen zeigen. Die vorgenommene Ersetzung s wird abgelegt. Falls keine Substitution existiert, wird ein Fehler zurückgegeben.

   Für jede Typ-Variable gibt es ein eindeutiges Blatt. Ansonsten brauchen strukturell äquivalente Typen nicht dem selben Knoten zugewiesen zu sein.

Der allgemeinste Unifikator faßt äquivalente Typen zu einer Klasse zusammen. Da die Typen rekursiv aufgebaut sind, kann auch der allgemeinste Unifikator nicht ohne Rekursion auskommen.

Grammatikumsetzungsbeispiel

$$E \longrightarrow E_1(E_2) \;\; \left\{ \begin{array}{l} p = \texttt{mkleaf}(newtypevar); \\ \texttt{unify}(E_1.type, \texttt{mknode}(\mapsto, E_2.type, p)); \\ E.type = p \end{array} \right\}$$

Dabei erzeugt $\texttt{mkleaf}(v)$ ein neues Blatt mit der neuen Variablen v, und $\texttt{mknode}(op, arg_1, arg_2)$ einen neuen Knoten mit Operator op und linken Sohn arg_1 und rechten Sohn arg_2. Sei nun angenommen $E_1.type = \alpha$ und $E_2.type = \beta$, dann ist $\alpha = \beta \mapsto \gamma$. Für γ wird nun zunächst mit $\texttt{mkleaf}()$ eine neue Variable erzeugt, mit $\texttt{mknode}()$ ein neuer Knoten angelegt, dann wird $E_1.type$ mit $E_2.type \mapsto \gamma$ unifiziert. Damit erhält γ einen bereits bekannten Typ.

$$E \longrightarrow E_1, E_2 \; \{E.type = \texttt{mknode}(\times, E_1.type, E_2.type)\}$$
$$E \longrightarrow id \; \{E.type = \texttt{fresh}(id.type)\}$$

Algorithmus zur Unifikation

Eingabe: Ein Graph und ein Knotenpaar (n, m), das zu unifizieren ist

Ausgabe: ja, falls n und m unufizierbar
nein, sonst

Methode

Zunächst werden die Knoten als Records dargestellt, in denen der Konstruktor, der linke Knoten, der rechte Knoten und die Menge der äquivalenten Knoten stehen.

Initialisierung

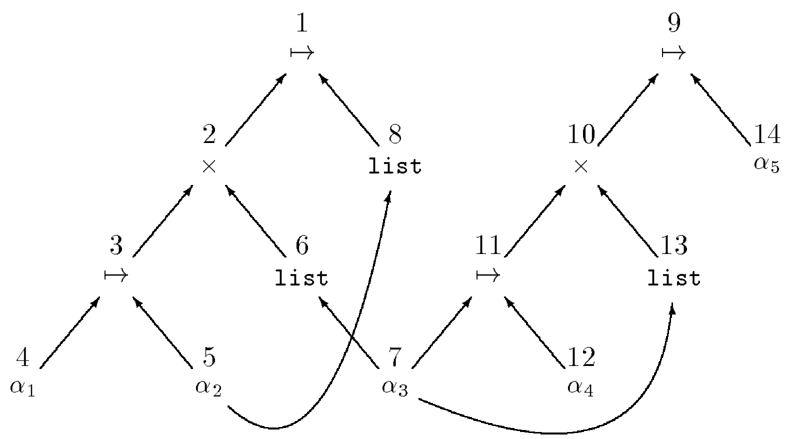
Jeder Knoten bildet eine eigene Menge. Zu jedem Zeitpunkt bilden die Menge der Klassen eine Klasseneinteilung auf der Menge aller Knoten.

Die Funktion $\texttt{unify}(n, m)$ sieht wie folgt aus:

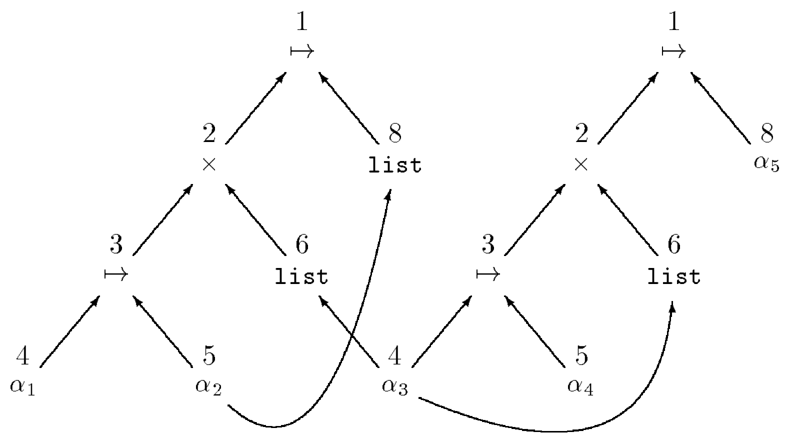
```
BOOL unify(node n, node m)
{
  s =find(m);  // find() liefert für die Menge m
  t =find(n);   // einen ausgezeichneten Repräsentanten
  if s == t
     return (TRUE);
  else if s,t represent the same basic type
     return (TRUE);
  else if (s is an op-node with children s1,s2 and
           t is an op-node with children t1,t2 and
           the operator is the same)
  {
     union(s,t);  // Die Menge, in denen s,t liegen werden vereinigt
     return (unify(s1,t1) and unify(s2,t2));
  }
  else if (s or t represents a variable)
  {
     union(s,t);
     return (TRUE);
  }
  else
     return (FALSE);
}
```

Wir betrachten nochmals

$$\Big(((\alpha_1 \mapsto \alpha_2)) \times list(\alpha_3)\Big) \mapsto list(\alpha_2)$$
$$\Big(((\alpha_3 \mapsto \alpha_4)) \times list(\alpha_3)\Big) \mapsto \alpha_5$$



Wir wollen feststellen, ob diese beiden Ausdrücke äquivalent sind. Führe dazu `unify`$(1,9)$ durch. Es wird also "$\mapsto$" mit "$\mapsto$"verglichen, das heißt `1` und `9` werden fusioniert. Jetzt wird rekursiv `unify`$(2,10)$ und `unify`$(8,14)$, und dann `unify`$(3,11)$ und `unify`$(6,13)$ und so weiter aufgerufen. Als Resultat erhalten wir



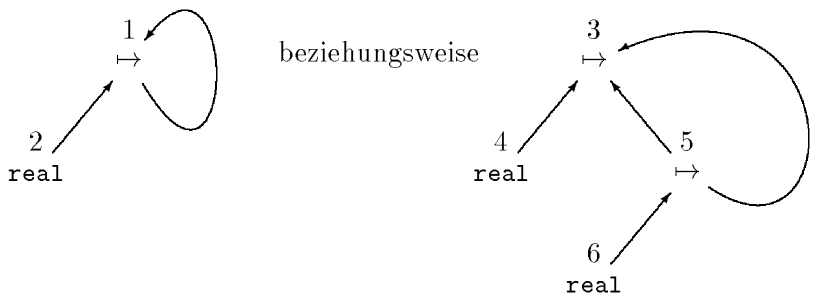
Also liegt strukturelle Äquivalenz der beiden Ausdrücke vor.

Beispiel (rekursiv definiert)

Gegeben seien die beiden folgenden Typenvariablen e und f

$$e\colon real \mapsto e$$
$$f\colon real \mapsto (real \mapsto f)$$

mit zugehörigen Typenbäumen



Mit `unify`$(1,3)$ werden `1` und `3` in dieselbe Klasse gelegt. Dann erfolgt ein rekursiver Aufruf von `unify`$(2,4)$ und `unify`$(1,5)$. Die Knoten `2` und `4` sind vom selben Basis-Typ, das heißt `unify`$(2,4)$ liefert "`TRUE`" zurück.

`unify`$(1,5)$ ruft rekursiv `unify`$(2,6)$ und `unify`$(1,3)$ auf. `unify`$(2,6)$ liefert "`TRUE`", da `2` und `6` vom gleichen Basis-Typ sind, und da `1` und `3` in der selben Klasse liegen liefert auch `unify`$(1,3)$ "`TRUE`".

Insgesamt folgt also, daß

$$e\colon real \mapsto e$$

und

$$f\colon real \mapsto (real \mapsto f)$$

strukturell äquivalent, das heißt austauschbar sind.

7 Berechenbarkeit

Wir beschäftigen uns im Wesentlichen mit der Frage, ob `while`-Schleifen nötig sind, oder ob `for`-Schleifen ausreichen.

7.1 Theorie der Berechenbarkeit

Die Theorie der Berechenbarkeit beschreibt alle Funktionen

$$f\colon \mathbb{N}^k \longrightarrow \mathbb{N},$$

die mit Computern zu berechnen sind ("Church'sche These").

Die Beschreibung dieser Funktionen geschieht implizit, indem man einige Grundfunktionen, Kompositionskonstruktoren und ein Rekursionsschema angibt.

7.1.1 Grundfunktionen

Für das Weitere sei $k \in \mathbb{N}$, $0 \in \mathbb{N}$, $\mathbb{N}^k = \underbrace{\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N}}_{k}$, $\mathbb{N}^0 = ()$. Definiere die Grundfunktionen ν, $C_0^{(0)}$ und $C_0^{(1)}$ durch

$$\begin{aligned}
\nu\colon \mathbb{N} \to \mathbb{N};\ \nu(n) &= n+1 \quad \text{(Nachfolgefunktion)}\\
C_0^{(0)}\colon \mathbb{N}^0 \to \mathbb{N};\ C_0^{(0)}() &= 0 \quad \text{(nullstellige Nullfunktion)}\\
C_0^{(1)}\colon \mathbb{N}^1 \to \mathbb{N};\ C_0^{(1)}(n) &= 0 \quad \text{(einstellige Nullfunktion)}
\end{aligned}$$

und für $1 \leq i \leq k$ die Grundfunktion $\Pi_i^{(k)}$ durch

$$\Pi_i^{(k)}\colon \mathbb{N}^k \to \mathbb{N};\ \Pi_i^{(k)}(n_1, \ldots, n_k) = n_i \quad \text{(Projektion auf die } i\text{-te Komponente)}$$

somit ist also $\Pi_1^{(1)}(n) = n$, das heißt $\Pi_1^{(1)} \equiv \mathrm{id}$.

7.1.2 Komposition

Seien $r, s \in \mathbb{N}$, $g\colon \mathbb{N}^s \longrightarrow \mathbb{N}$ und für $i = 1, \ldots, s$ sei $f_i\colon \mathbb{N}^r \longrightarrow \mathbb{N}$. Für $n_1, \ldots, n_r$ schreiben wir n. Dann ist

$$f := g \circ (f_1, \ldots, f_s)\colon \mathbb{N}^r \longrightarrow \mathbb{N}$$

mit

$$f(n) := \begin{cases} g(f_1(n), \ldots, f_s(n)) & \text{falls jedes } f_i \text{ für } n \text{ definiert ist}\\ \perp & \text{sonst}\end{cases}$$

die Komposition von f_i und g.

7.1.3 Primitive Rekursion

Sei $k \in \mathbb{N}$ und $g\colon \mathbb{N}^k \to \mathbb{N}$, $h\colon \mathbb{N}^{k+2} \to \mathbb{N}$. Dann erhält man durch das folgende Rekursionsschema eine Funktion

$$f\colon \mathbb{N}^{k+1} \to \mathbb{N}$$

mit

$$f(x,y) := \begin{cases} g(x) & \text{für } y = 0,\\ h(x, y-1, f(x, y-1)) & \text{für } y > 0.\end{cases}$$

7.1.4 Definition

Alle aus den Grundfunktionen durch endliches Anwenden von Komposition und primitiver Rekursion konstruierbaren Funktionen heißen *primitiv rekursiv*.

7.1.5 Beispiele für Funktionen, die mit diesen Hilfsmitteln beschrieben werden können

i) Seien die Funktionen

$$g\colon I\!N \to I\!N;\ x \mapsto x$$

und

$$h\colon I\!N^3 \to I\!N;\ (x_1, x_2, x_3) \mapsto x_3 + 1$$

gegeben, das heißt $h \equiv \nu \circ \Pi_3^{(3)}$. Diese Funktionen definieren durch primitive Rekursion die Funktion

$$f\colon I\!N^2 \to I\!N;\ (x, y) \mapsto x + y,$$

das heißt die Addition ist primitiv rekursiv.

Beweis

Es gilt

$$\begin{aligned} f(x,0) &= g(x) = x \\ f(x,1) &= h(x,0,f(x,0)) = h(x,0,x) = x+1 \\ f(x,2) &= h(x,1,f(x,1)) = h(x,1,x+1) = x+2 \\ f(x,y) &= x+y, \end{aligned}$$

da $f(x,y+1) = h(x,y,f(x,y)) = f(x,y)+1 \overset{\mathrm{IV}}{=} x+y+1$.

ii) Seien die Funktionen

$$g\colon I\!N \to I\!N;\ n \mapsto 0$$

und

$$h\colon I\!N^3 \to I\!N;\ (x_1, x_2, x_3) \mapsto x_1 + x_3$$

gegeben. Diese Funktionen definieren durch primitive Rekursion die Funktion

$$f\colon I\!N^2 \to I\!N;\ (x, y) \mapsto x \cdot y,$$

das heißt die Multiplikation ist primitiv rekursiv.

Beweis

Es gilt

$$\begin{aligned} f(x,0) &= g(x) = 0 \\ f(x,1) &= h(x,0,f(x,0)) = h(x,0,0) = x \\ f(x,2) &= h(x,1,f(x,1)) = h(x,y,x) = x+x = 2x \\ f(x,y) &= x \cdot y, \end{aligned}$$

da $f(x,y+1) = h(x,y,f(x,y)) \overset{\mathrm{IV}}{=} x + (x \cdot y) = x(y+1)$.

iii) Seien die Funktionen

$$g\colon I\!N^0 \to I\!N;\ () \mapsto 0$$

und

$$h\colon I\!N^2 \to I\!N;\ (x, y) \mapsto x$$

gegeben. Diese Funktionen definieren durch primitive Rekursion die Funktion

$$f\colon I\!N \to I\!N;\ y \mapsto y \dot{-} 1,$$

das heißt $y \dot{-} 1$ ist primitiv rekursiv. Dabei ist $y \dot{-} 1 := \begin{cases} 0 & \text{falls } y = 0, \\ y-1 & \text{sonst.} \end{cases}$

Beweis

Es gilt

$$\begin{aligned} f(0) &= g() = 0 \\ f(y+1) &= h(y, f(y)) = y \\ f(y) &= y \dot{-} 1. \end{aligned}$$

Analog zu oben läßt sich

$$x \dot{-} y := \begin{cases} 0 & \text{falls } y \geq x, \\ x - y & \text{sonst.} \end{cases}$$

berechnen.

7.1.6 Bemerkung

Alle primitiv rekursiven Funktionen sind total.

7.1.7 Satz

Die folgenden Funktionen sind primitiv rekursiv

$$\begin{aligned} &f_k\colon \mathbb{N} \to \mathbb{N};\ f_k(x) = k \cdot x, \quad \text{mit } k \in \mathbb{N} \\ &f\colon \mathbb{N}^2 \to \mathbb{N};\ f(x, y) = x^y \\ &f\colon \mathbb{N}^2 \to \mathbb{N};\ f(x, y) = \left.x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\right\}y \\ &f_k\colon \mathbb{N} \to \mathbb{N};\ f_k(x) = k^x \quad \text{mit } k \in \mathbb{N} \\ &f\colon \mathbb{N} \to \mathbb{N};\ f(x) = \operatorname{sign}(x) = \begin{cases} 0 & \text{falls } x = 0, \\ 1 & \text{falls } x > 0. \end{cases} \\ &f\colon \mathbb{N} \to \mathbb{N};\ f(x) = \overline{\operatorname{sign}}(x) = \begin{cases} 1 & \text{falls } x = 0, \\ 0 & \text{falls } x > 0. \end{cases} \end{aligned}$$

Beweis

Die primitive Rekursion ist bei allen Funktionen leicht zu beweisen.

Wir definieren für das Weitere $x := x_1, \ldots, x_k$.

7.1.8 Satz

Seien $r \in \mathbb{N}$, $k \in \mathbb{N}$ und $g\colon \mathbb{N}^k \to \mathbb{N}$.

i) Fallunterscheidung

Sei $x \in \mathbb{N}^k$ und seien die Funktionen $g_1, \ldots, g_r, h_1, \ldots, h_r\colon \mathbb{N}^k \to \mathbb{N}$. Definiere

$$g\colon \mathbb{N}^k \to \mathbb{N};\ g(x) := \begin{cases} g_1(x) & \text{falls } h_1(x) = 0, \\ g_2(x) & \text{falls } h_2(x) = 0, \\ \vdots & \\ g_r(x) & \text{falls } h_r(x) = 0, \end{cases}$$

wobei jeweils genau ein $h_i(x) = 0$ ist. Sind dabei die h_i und g_i primitiv rekursiv, so ist auch g primitiv rekursiv.

ii) Beschränkte Summe

Die Funktion $f\colon \mathbb{N}^{k+1} \to \mathbb{N}$ sei primitiv rekursiv, dann ist auch

$$f_1\colon \mathbb{N}^{k+1} \to \mathbb{N};\ f_1(x, y) := \sum_{0 \leq i \leq y} f(x_1, \ldots, x_k, i)$$

primitiv rekursiv.

iii) Beschränktes Produkt

Die Funktion $f: I\!N^{k+1} \to I\!N$ sei primitiv rekursiv, dann ist auch

$$f_2: I\!N^{k+1} \to I\!N;\ f_2(x, y) := \prod_{0 \le i \le y} f(x_1, \ldots, x_k, i)$$

primitiv rekursiv.

iv) Beschränkte Minimalisierung

Die Funktion $f: I\!N^{k+1} \to I\!N$ sei primitiv rekursiv, dann ist auch

$$f_3: I\!N^{k+1} \to I\!N;\ f_3(x, y) := \begin{cases} j & \text{falls } j = \arg\min\limits_{i \le y}\{\, i \mid f(x_1, \ldots, x_k, i) = 0 \,\} \\ y + 1 & \text{sonst} \end{cases}$$

primitiv rekursiv. Mit f_3 kann man somit eine endliche Schleife (das heißt `for`-Schleife) simulieren.

Beweis

i) $g(x) = g_1(x) \cdot \overline{\mathrm{sign}}(h_1(x)) + \ldots + g_r(x) \cdot \overline{\mathrm{sign}}(h_r(x))$ und damit primitiv rekursiv.

ii) Es ist

$$f_1(x, 0) = f(x, 0) =: g(x)$$
$$f_1(x, y + 1) = f_1(x, y) + f(x, y + 1) =: h(x, y, f_1(x, y))$$

und somit f_1 primitiv rekursiv.

iii) Produkt analog.

iv) Aus i) folgt, daß eine Fallunterscheidung primitiv rekursiv ist, falls die Entscheidungsfunktionen primitiv rekursiv sind. Wegen

$$f_3(x, 0) := \begin{cases} 0 & \text{falls } f(x, 0) = 0 \\ 1 & \text{sonst} \end{cases}$$

kann man die Funktion $g: I\!N^k \to I\!N$ definieren als

$$g(x) := \mathrm{sign}(f(x, 0)) \equiv f_3(x, 0).$$

Somit ist g primitiv rekursiv. Es bleibt nun noch ein h zu bestimmen:

$$f_3(x, y + 1) = \begin{cases} f_3(x, y) & \text{falls } f_3(x, y) \le y \text{ oder } f(x, y + 1) = 0 \\ y + 2 & \text{sonst} \end{cases}.$$

Da nun

$$f_3(x, y) \le y \quad \text{oder} \quad f(x, y + 1) = 0$$

äquivalent ist zu

$$(f_3(x, y) \dot{-} y) \cdot \mathrm{sign}(f(x, y + 1)) = 0$$

ist es sinnvoll die beiden Funktionen $h_1: I\!N^{k+2} \to I\!N$ und $h_2: I\!N^{k+2} \to I\!N$ folgendermaßen zu definieren

$$h_1(x, y, z) = (z \dot{-} y) \cdot \mathrm{sign}(f(x, y + 1)) \quad \text{primitiv rekursiv}$$
$$h_2(x, y, z) = \overline{\mathrm{sign}}(h_1(x, y, z)) \quad \text{primitiv rekursiv}$$

Da nun die Fallunterscheidung primitiv rekursiv ist, ist auch die Funktion $h: I\!N^{k+2} \to I\!N$ definiert durch

$$h(x, y, z) = \begin{cases} z & \text{falls } h_1(x, y, z) = 0 \\ y + 2 & \text{falls } h_2(x, y, z) = 0 \end{cases}$$

primitiv rekursiv, da die Entscheidungsfunktionen primitiv rekursiv sind. Insgesamt folgt also, daß f_3 primitiv rekursiv ist.

7.1.9 μ-Rekursion

Gegeben sei ein $f\colon \mathbb{N}^{k+1} \longrightarrow \mathbb{N}$ für ein $k \in \mathbb{N}$. Dann ist

$$\mu(f)\colon \mathbb{N}^k \longrightarrow \mathbb{N}$$

definiert durch

$$\mu(f)(x) := \begin{cases} y & \text{falls } f(x,y) = 0 \text{ und für alle } i \text{ gilt: } f(x,i) \text{ ist definiert und } f(x,i) \neq 0 \\ \perp & \text{sonst} \end{cases}$$

Für den Wert y ist dabei keine obere Schranke vorhanden. Für den Fall $k = 0$ stellt sich $\mu(f)()$ wie folgt dar:

$$\mu(f)() = \begin{cases} y & \text{falls } f(y) = 0 \text{ und } f(i) \in \mathbb{N},\ f(i) \neq 0 \ \text{ für } \ 0 \leq i < y \\ \perp & \text{sonst} \end{cases}$$

7.1.10 Definition

Alle aus den Grundfunktionen durch endliche Anwendung von Komposition, primitiver Rekursion und von μ-Rekursion erzeugbaren Funktionen heißen *μ-rekursiv*.

7.1.11 Chursch'sche These

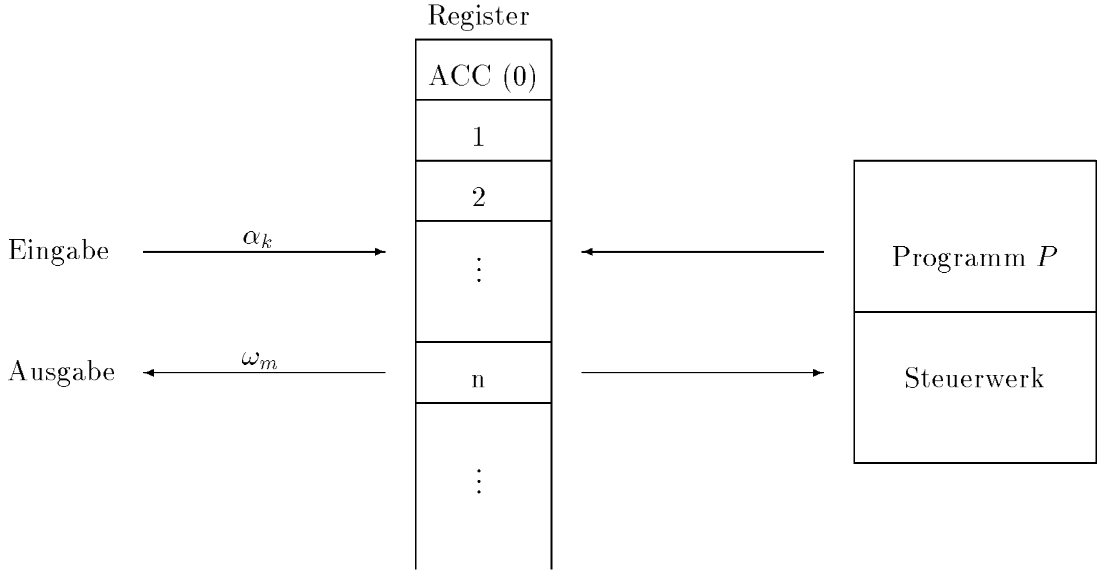
Alle "intuitiv berechenbaren" Funktionen sind μ-rekursiv.

Vorgehensweise zum Beweis dieser These

Wir betrachten nun URM-Maschinen und kommen über URM-Programme zur URM-Berechenbarkeit. Alle URM-berechenbaren Funktionen sind mit GOTO-Programmen berechenbar. Alle mit GOTO-Programmen berechenbaren Funktionen wiederum sind μ-rekursiv.

URM-Programme

URM steht für **U**nbeschränkte **R**egister **M**aschine. Bei einer URM gibt es abzählbar viele Register, wobei jedes eine beliebig große natürliche Zahl aufnehmen kann. Eine Registerbelegung ist ein Element $z \in \mathbb{N}^{\mathbb{N}}$, $z = (z_0, z_1, z_2, \ldots)$ wobei $z_i \in \mathbb{N}$ für alle $i \in \mathbb{N}$.



Für $k \in I\!N$ belegt α_k die Registerplätze $1, \ldots, k$ mit Eingabewerten. Das Steuerwerk arbeitet das Programm P ab und verändert dabei den Inhalt der Registerplätze. ω_m gibt den Inhalt der Plätze $1, \ldots, m$ aus. Mit $\|P\|$ bezeichnen wir die Semantik des Programmes P und mit $|P|$ seine Wirkung.

$$\|P\| \colon I\!N^k \xrightarrow{\alpha_k} I\!N^{I\!N} \xrightarrow{|P|} I\!N^{I\!N} \xrightarrow{\omega_m} I\!N^m$$

Befehle

Arithmetische Befehle

$$|A_i| \colon I\!N^{I\!N} \to I\!N^{I\!N}; \quad |A_i|(z) = z' : \ \forall j \ z'_j = z_j + \delta_{ij}$$
$$|S_i| \colon I\!N^{I\!N} \to I\!N^{I\!N}; \quad |S_i|(z) = z' : \ \forall j \ z'_j = z_j \dot{-} \delta_{ij}$$

Ablaufkontrolle

$(P)_i$ bewirkt, daß $|P|$ solange wiederholt wird, bis auf dem i-ten Registerplatz 0 steht. $|(P)_i|$ ist nicht überall definiert, da der i-te Platz nie 0 zu werden braucht.

Hintereinanderausführung

$P_1; P_2$ bewirkt die Hintereinanderausführung von P_1 und P_2, das heißt $|P_1; P_2| = |P_2| \circ |P_1|$.

Ein URM-Programm läßt sich rekursiv aus folgenden Bauteilen zusammensetzen:

1. Für alle $i \in I\!N$ ist A_i ein URM-Programm.
2. Für alle $i \in I\!N$ ist S_i ein URM-Programm.
3. Sind P_1 und P_2 URM-Programme, so auch $P_1; P_2$.
4. Ist P ein URM-Programm und $i \in I\!N$, so ist $(P)_i$ ein URM-Programm.

Nur die mit 1.– 4. erzeugbaren Zeichenfolgen sind URM-Programme. Zu einem URM-Programm P läßt sich

$$|P| \colon I\!N^{I\!N} \to I\!N^{I\!N}$$

rekursiv anhand der Konstruktion von P aus den zugehörigen Befehlen bestimmen. Dabei bewirkt

$$A_i \text{ stets } |A_i|$$
$$S_i \text{ stets } |S_i|$$
$$(P)_i \text{ stets } |(P)_i|$$
$$P_1; P_2 \text{ stets } |P_2| \circ |P_1|$$

Definition

Wenn es zu $f \colon I\!N^k \to I\!N^m$ ein URM-Programm P gibt mit $\|P\| = f$, so heißt f URM-berechenbar.

Probleme

1. Finde P.
2. Weise nach, daß $\|P\| = f$ ist.

zu 1.

Für Funktionen, die konstruktiv beschrieben sind, speziell durch μ-Rekursion, können wir P konstruieren. (Dann muß die Funktion μ-rekursiv dargestellt sein! (erneutes Problem)).

zu 2.

Mit formaler Logik läßt sich nachweisen, daß geeignete Programme Funktionen berechnen, für die sie geschrieben sind. Dazu später mehr.

Satz

Jede μ-rekursive Funktion ist URM-berechenbar.

Beweis

Für die Nachfolgerfunktion ν ist A_1 das Programm P mit $\|P\| = \nu$.

Für die nullstellige Nullfunktion $C_0^{(0)}$ ist $A_0; S_0$ das Programm P mit $\|P\| = C_0^{(0)}$.

Für die einstellige Nullfunktion $C_0^{(1)}$ ist $(S_1)_1$ das Programm P mit $\|P\| = C_0^{(1)}$.

Für die Projektion auf die i-te Komponente $\Pi_i^{(k)}$ ist $(S_0)_0; (A_0; S_i)_i; (S_1)_1; (A_1; S_0)_0$ das Programm P mit $\|P\|$ ist $\Pi_i^{(k)}$. Dabei kopiert hier das Programm P zunächst z_i in den Accumulator (Platz 0), dann wird der Inhalt von Register 1 gelöscht, und anschließend der Inhalt von Platz 0 (das ist gerade z_i) auf den Platz 1 kopiert.

Ist $f\colon I\!N^{k+1} \longrightarrow I\!N$ URM-berechenbar, dann ist auch $\mu(f)\colon I\!N^k \longrightarrow I\!N$ URM-berechenbar. Sei dazu

$$|P|\colon (0, x_1, \ldots, x_k, y, 0, 0, \ldots) \longmapsto \begin{cases} 0, x_1, \ldots, x_k, y, \underbrace{f(x,y)}_{x_{k+2}}, 0, \ldots & (x,y) \in D(f), \\ \bot & \text{sonst.} \end{cases}$$

wobei $|P|$ durch f beschrieben wird. Da f URM-berechenbar ist folgt, daß P ein URM-Programm ist. Wir müssen also nur noch ein Programm P' finden, welches der μ-Rekursion entspricht. Dazu definieren wir das Programm P' durch

$$P' := P; (A_{k+1}; (S_{k+2})_{k+2}; P)_{k+2}.$$

Die Definition von P' entspricht folgendem Programmcode:

```
P;
while (x_{k+2} > 0)
{
   x_{k+1}++;
   x_{k+2}=0;
   P;
}
```

Nach Beendigung des Programms P' steht auf Platz $k+1$ der kleinste Index, für den $f(x,y) = 0$ gilt.

Bemerkung

URM-Programme sind `while`-Programme.

Alle primitiv rekursiven Funktionen sind total, das Programm $\|(A_1)_1\|\colon I\!N \rightarrow I\!N$ ist jedoch nur an der Stelle Null definiert, das heißt nur dann wenn auf dem Registerplatz 1 eine Null steht.

Endlos-Schleifen sind nicht möglich mittels primitiv rekursiver Funktionen, man benötigt μ-rekursive Funktionen.

GOTO-Programme

Jedes GOTO-Programm hat die folgende prinzipielle Gestalt

$$l_i : \text{Befehl};$$

wobei l_i eine Marke ist, und Befehl für die an dieser Stelle auszuführenden Anweisung steht. Da bei der Marke l_i keine Kontrolle der Daten mit denen weiterzuarbeiten ist, stattfindet, haben diese Programme natürlich eine erhebliche Fehlerwahrscheinlichkeit. Sie werden mit "Spaghetti-Programmen" bezeichnet, da keine modulare Programmierung und keine strukturierte Aufteilung in Unterprogramme möglich ist.

Für das Weitere sei L eine nichtleere, endliche Menge, die Menge der Marken. Ohne Beschränkung der Allgemeinheit sei $0 \in L \subset I\!N$. Weiterhin sei V eine nichtleere, endliche Menge, die Menge der Variablen. Ohne Beschränkung der Allgemeinheit sei $V = \{x_1 \dots x_n\}$ für ein $n \in I\!N$. A sei eine nichtleere endliche Menge von Anweisungen der Form

$$l : x_i = x_i + 1; l' \quad \text{oder} \quad l : x_i = x_i \dot{-} 1; l' \quad \text{oder} \quad l : \texttt{if } x_i = 0; l', l''$$

wobei $l, l', l'' \in L$ und $x_i \in V$ für alle i ist.

Schematisch läßt sich ein GOTO-Programm somit wie folgt darstellen:

$$l : \boxed{\text{Anweisung}}; l'$$

wobei l die Marke der Anweisungen und l' die Nachfolgemarke ist beziehungsweise

$$l : \boxed{\text{Bedingung}}; l', l''$$

wobei l' und l'' die Verzweigungsmarken sind.

Eine Folge $P = a_0; a_1; \dots; a_k$ von Anweisungen a_i heißt *GOTO-Programm*, wenn zu jedem $l \in L$ höchstens ein a_i mit Marke l existiert.

Semantik von GOTO-Programmen

Es gibt Registerplätze $x_0, x_1, \dots, x_n$. Der Registerplatz x_0 dient als Befehlszähler. Dort wird stets die Marke der nächsten auszuführenden Anweisung abgelegt. Die Semantik von GOTO-Programmen kann mit den Funktionen

$$|a_i| : I\!N^{n+1} \to I\!N^{n+1} \qquad i = 1, 2, 3$$

beschrieben werden.

Erhöhung des i-ten Registerplatzes um 1

Dies bewirkt die Funktion a_1

$$l : x_i = x_i + 1; l'.$$

x_0 erhält den Wert l', der Inhalt von x_i wird um eins erhöht und die restlichen Registerplätze bleiben unverändert.

Verminderung des i-ten Registerplatzes um 1

Dies bewirkt die Funktion a_2

$$l : x_i = x_i \dot{-} 1; l'.$$

x_0 erhält den Wert l', der Inhalt von x_i wird um eins vermindert, falls möglich und die restlichen Registerplätze bleiben unverändert.

Bedingte Verzweigung

Dies bewirkt die Funktion a_3

$$l : \texttt{if}\ x_i = 0, l'; l''.$$

Falls $x_i = 0$ erhält x_0 den Wert l', sonst l''.

Definition eines GOTO-Programmes

Gegeben sei ein GOTO-Programm P, dann ist die Semantik von P durch

$$\|P\|: \mathbb{N}^n \to \mathbb{N};\ (x_1, \ldots, x_n) \in \mathbb{N}^n \xrightarrow{\alpha_n} \mathbb{N}^{n+1} \xrightarrow{|P|} \mathbb{N}^{n+1} \xrightarrow{\omega_1} \mathbb{N} \ni x_1$$

definiert.

Start

Die Registerplätze $x_1, \ldots, x_n$ werden mit Eingabewerten belegt. x_0 erhält den Wert 0, ω_1 gibt den Wert von x_1 aus.

Abarbeitung eines Programmes P

Zuerst wird der Befehl aus P mit der Marke 0 ausgeführt. Falls dieser nicht existiert, bricht die Bearbeitung ab. Nach i Schritten liege die Marke l auf dem Platz x_0. Dann wird der Befehl mit Marke l ausgeführt, falls ein solcher existiert. Existiert kein Befehl mit Marke l, so wird die Abarbeitung beendet.

Die Wirkung $|P|$ des Programmes P ist folglich

$$|P| = \cdots |(l' : \ldots ; l'')| \circ |(l : \ldots ; l')| \circ |(0 : \ldots ; l)|$$

Falls die Abarbeitung nicht nach endlich vielen Schritten abbricht, ist $|P|$ für den vorliegenden Datensatz nicht definiert.

Satz

Ist eine Funktion URM-berechenbar, so ist sie auch GOTO-berechenbar.

Beweis

Sei P ein URM-Programm, das die gegebene Funktion berechnet. Jeder einzelne Befehl aus P läßt sich dann durch GOTO-Befehle simulieren, das heißt die gleiche Änderung der Registerbelegung wird erreicht. Das heißt A_i entspricht $l : x_i = x_i + 1; l'$ und S_i entspricht $l : x_i = x_i \dot{-} 1; l'$.

$M_1; M_2$ entspricht $P_1; P_2$ wobei die Folgemarke von P_1 die Startmarke von P_2 ist.

$(M)_i$ entspricht $l : \texttt{if}\ x_i = 0; l', l''; P$ wobei M durch P simuliert wird und die erste Anweisung von P die Marke l'' besitzt. Die Folgemarke der letzten Anweisung von P ist l. Und l' ist die Startmarke des Programmteils, das nach Abarbeitung von $(M)_i$ abzuarbeiten ist. Die Vergabe der Marken muß so erfolgen, daß keine Marke doppelt auftritt.

Satz

Jede GOTO-berechenbare Funktion ist μ-rekursiv.

Beweis

Wir betrachten die Abfolge der einzelnen Befehlsausführungsschritte. Ist P ein GOTO-Programm, so sei E_P eine Einzelschrittfunktion, die aus einer Registerbelegung durch Anwendung der entsprechenden Anweisung aus P die Nachfolgebelegung berechnet, das heißt

$$E_P: \mathbb{N}^{n+1} \to \mathbb{N}^{n+1};\ (z_0, \ldots, z_n) \mapsto (z'_0, \ldots, z'_n).$$

wobei für $z' := (z'_0, \ldots, z'_n)$ wenn $z_0 = l$ die Marke eines Befehls $a \in P$ ist

$$z' = \begin{cases} z'_0 = l' \wedge z'_i = z_i + 1 \wedge \forall i \neq j: \ z'_j = z_j & \text{falls } a = (l : x_i = x_i + 1; l') \\ z'_0 = l' \wedge z'_i = z_i \dot{-} 1 \wedge \forall i \neq j: \ z'_j = z_j & \text{falls } a = (l : x_i = x_i \dot{-} 1; l') \\ z'_0 = l' \ \forall j \neq 0: \ z'_j = z_j & \text{falls } a = (l : \texttt{if } x_i = 0; l', l'') \wedge x_i = 0 \\ z'_0 = l'' \ \forall j \neq 0: \ z'_j = z_j & \text{falls } a = (l : \texttt{if } x_i = 0; l', l'') \wedge x_i \neq 0 \\ \forall i: \ z'_i = z_i & \text{sonst} \end{cases}$$

gelten muß.

Da P nur endlich viele Anweisungen enthält, ist die Fallunterscheidung endlich. Da P ein GOTO-Programm ist, ist die Fallunterscheidung disjunkt. Die einzelnen Operationen sind primitiv rekursiv, das heißt die Berechnung jedes einzelnen Z'_i ist primitiv rekursiv. Und damit ist auch

$$\Pi_j^{n+1} \circ E_P,$$

die Projektion von E_P auf die j-te Komponente für alle j primitiv rekursiv.

Beschreibung des Ablaufs

Berücksichtigung eines Schrittzählers t. Definiere dazu die Funktion

$$E'_P: \mathbb{N}^{n+2} \to \mathbb{N}^{n+1}$$

durch

$$\begin{aligned} E'_P(z_0, \ldots, z_n, 0) &:= E_P(z_0, \ldots, z_n) \\ E'_P(z_0, \ldots, z_n, t) &:= E_P(E'_P(z_0, \ldots, z_n, t-1)) \quad \text{für } t > 0. \end{aligned}$$

Und damit ist auch

$$\Pi_j^{n+2} \circ E'_P(z, t)$$

für alle j primitiv rekursiv.

Zähle die Gesamtzahl der Schritte, die das GOTO-Programm zu durchlaufen hat. Diese Zahl braucht nicht definiert zu sein, es ist auch keine obere Schranke dafür bekannt. Daher wird hier μ-Rekursion erforderlich. Wir bezeichnen mit $T_p(z_0, \ldots, z_n)$ die Anzahl der Schritte, die das Programm P bei Eingabe von $z_1, \ldots, z_n$ ausführt. Damit ist

$$T_P(z_0, \ldots, z_n) := \begin{cases} \min\limits_{t \in \mathbb{N}} \{\Pi_0^{n+2} \circ E'_P(z, t) \quad \text{ist keine Marke eines Befehls aus } P\} \\ \perp \quad \text{falls} \quad \forall t \in \mathbb{N}: \ \Pi_0^{n+2} \circ E'_P(z, t) \quad \text{ist Marke eines Befehls aus } P \end{cases}$$

Hat P die Markenmenge $L(P) = \{0 \ldots m\}$, so gilt

$$\Pi_0^{n+2} \circ E'_P(z, t) > m$$
$$\Updownarrow$$
$$z_0 \text{ ist keine Marke eines Befehls aus } P$$
$$\Updownarrow$$
$$\overline{\text{sign}}(\Pi_0^{n+2} \circ E'_P(z, t) \dot{-} m) = 0$$

Daraus folgt $T_P(z) = \mu\Big(\overline{\text{sign}}(\Pi_0^{n+2} \circ E'_P(z, t) \dot{-} m)\Big)$, das heißt der Schrittzähler wurde minimiert.

Somit bleibt nur noch die Beschreibung der Ausgabe. Definiere dazu

$$R_P(z) := \begin{cases} E'_P(z, T_P(z)) & \text{falls } z \text{ aus dem Definitionsbereich von } T_P(\cdot), \\ \perp & \text{sonst.} \end{cases}$$

$R_P(z) = E'_P\Big(z, \mu\Big(\overline{\text{sign}}(\Pi_0^{n+2} \circ E'_P(z, t) \dot{-} m)\Big)\Big)$ ist Komposition μ-rekursiver Funktionen, also μ-rekursiv in jeder Komponente.

Ergebnis

$\|P\| = \Pi_1^{n+1} \circ R_P$ ist μ-rekursiv.

Folgerung

μ-rekursiv $\iff$ GOTO-berechenbar $\iff$ URM-berechenbar (das heißt `while`-berechenbar)

Berechenbarkeit mit `for`-Schleifen

Im folgenden Abschnitt beschäftigen wir uns mit der Frage, ob man totale Funktionen mit `for`-Schleifen berechnen kann, das heißt sind totale Funktionen primitiv rekursiv.

Beispiel einer Funktion, die total und URM-berechenbar, aber nicht primitiv rekursiv ist

Seien dazu $B_0, B_1, B_2, \ldots, B_n, \ldots$ Familien von Funktionen, die primitiv rekursiv sind. Durch Diagonalisierung wird eine Funktion A definiert, die jeweils für ein n entsprechende Funktionswerte annimmt. Dieses A ist dann nicht mehr primitiv rekursiv. Dazu sei die Funktion B_0 zunächst wie folgt definiert

$$\begin{aligned} B_0(x) &:= \begin{cases} 1 & \text{falls } x = 0 \\ 2 & \text{falls } x = 1 \\ x+2 & \text{falls } x > 1 \end{cases} \\ B_{n+1}(x) &:= B_n^x(1) \qquad (= \underbrace{B_n(B_n(\cdots(B_n(1))\cdots))}_{x}\ ) \quad \text{für } n \in I\!N \end{aligned}$$

Damit ist $B_1(0) = 1$, $B_1(1) = 2$, $B_1(2) = B_0(B_0(1)) = 4$, $B_1(3) = 6$ und $B_1(x) = 2x$ da $B_1(x+1) = B_0(B_1(x)) = B_0(2x) = 2x + 2$.

Weiterhin ist $B_2(x) = 2^x$.

Schwieriger zu berechnen ist bereits $B_3(4) = B_2(B_2(B_2(B_2(1)))) = 2^{2^{2^2}}$ und $B_3(x) = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}\Big\}x$.

Und schließlich $B_4(x) = 2^{\cdot^{\cdot^{\cdot}}}$ ist schon gar nicht mehr so leicht anzugeben.

Definition (Große Ackermann-Funktion)

Die Funktion

$$A\colon I\!N^2 \to I\!N;\ (n, x) \mapsto B_n(x)$$

heißt die *Große Ackermann-Funktion.*

Satz

Die Große Ackermann-Funktion ist nicht primitiv rekursiv.

Beweis

Das Rekursionsschema für die Berechnung von $A(n, x)$ lautet

$$\begin{aligned} A(0, x) &= B_0(x) \\ A(n+1, 0) &= 1 \\ A(n+1, x+1) &= B_{n+1}(x+1) = B_n^{x+1}(1) = B_n(B_n^x(1)) \\ &= B_n(B_{n+1}(x)) = A(n, B_{n+1}(x)) = A(n, A(n+1, x)) \end{aligned}$$

Die Große Ackermann-Funktion ist eine totale Funktion, die sich mit einem URM-Programm berechnen läßt, jedoch nicht primitiv rekursiv ist. Alle $B_n(x)$ hingegen sind für festes n primitiv rekursiv. Die $B_n(x)$ klassifizieren die mit `for`-Schleifen der Schachtelungstiefe n berechenbaren Funktionen.

Vollständige Beweise hierzu finden sich in Informatik I.

Beweisideen

B_n **ist primitiv rekursiv**

$n = 0$

Wenn man

$$g_1(x) := \nu \circ C_0^{(0)}(x) \qquad g_2(x) := \nu \circ \nu \circ C_0^{(1)}(x) \qquad g_3(x) := \nu \circ \nu \circ \Pi_1^{(1)}(x)$$

definiert, läßt sich $B_0(x)$ schreiben als

$$B_0(x) = \begin{cases} g_1(x) & \text{falls } \operatorname{sign}(x) = 0, \\ g_2(x) & \text{falls } \operatorname{sign}(x+1) + \overline{\operatorname{sign}}(x) = 0, \\ g_3(x) & \text{falls } \overline{\operatorname{sign}}(x \dot{-} 1) = 0. \end{cases}$$

Somit folgt, daß B_0 primitiv rekursiv ist.

$n > 0$

Definiere

$$g\colon \mathbb{N}^0 \to \mathbb{N};\ () \mapsto 1$$

und

$$h\colon \mathbb{N}^2 \to \mathbb{N};\ (x, y) \mapsto B_n \circ \Pi_2^{(2)}(x, y).$$

Damit ist

$$B_{n+1}(0) = g(()) = 1$$

und

$$B_{n+1}(x+1) = h(x, B_{n+1}(x)) = B_n(B_{n+1}(x)) = \underbrace{B_n(B_n \ldots (B_n(1)) \ldots)}_{x+1},$$

das heißt B_n ist primitiv rekursiv.

Konstruktion der P_i

Wir definieren Mengen von Programmen, dazu wird P_0 wie folgt charakterisiert

(1) $A_i \in P_0 \quad \forall i \in \mathbb{N}$

(2) $(S_i)_i \in P_0 \quad \forall i \in \mathbb{N}$

(3) $M_1, M_2 \in P_0 \Longrightarrow M_1; M_2 \in P_0$

(4) Nur die mit (1), (2) und (3) konstruierten Programme sind in P_0.

Ist P_n bereits definiert, so sei P_{n+1} definiert durch

(1) $P_n \subseteq P_{n+1}$

(2) $M_1, M_2 \in P_{n+1} \Longrightarrow M_1; M_2 \in P_{n+1}$

(3) $M \in P_n$, $i \in \mathbb{N}$, i tritt in M nicht auf $\Longrightarrow (M; S_i)_i \in P_{n+1}$, das heißt

```
for k = 1 to k = Inhalt des i-ten Registers
    M;
```

(4) Nur die mit (1),(2) und (3) konstruierten Programme sind in P_{n+1}

Es gilt

$$P_0 \subsetneq P_1 \subsetneq P_2 \subsetneq \ldots \subsetneq P_n \subsetneq \ldots$$

In P_n definiere

$$F_n := \{f\colon \mathbb{N}^k \longrightarrow \mathbb{N} \mid k \in \mathbb{N},\ \exists P \in P_n \wedge f = \|P\|\}$$

Satz

Für alle natürlichen Zahlen n gilt

$$B_n \in F_n \quad \text{und} \quad B_{n+1} \notin F_n$$

Beweis

Auf dem Registerplatz 1 wird x übergeben. Wir zeigen zunächst, daß $B_n \in F_n$ gilt.

$n = 1$

Wir wissen bereits, daß

$$B_1(x) = \begin{cases} 1 & \text{falls } x = 0 \\ 2x & \text{falls } x > 0 \end{cases}.$$

Setzt man nun

$$X_1 = \underbrace{(\underbrace{A_0; A_2}_{\in P_0}; S_1)_1}_{\in P_1}; A_1; \underbrace{((S_1)_1; S_0)_0}_{\substack{\text{leert 1. Platz} \\ \text{falls } x \neq 0}}; \underbrace{(A_1; A_1; S_2)_2}_{\substack{\text{berechnet } 2x \\ \text{falls } x \neq 0}},$$

so ist $\|X_1\| = B_1$.

$n > 1$

Sei $X_n \in P_n$ und $\|X_n\| = B_n$. Ohne Beschränkung der Allgemeinheit sei nach Ausführung von X_n jedes für eine Zählschleife verwendete Register auf 0 heruntergezählt. Sei r ein noch nicht benutzter Registerplatz. Es gilt

$$B_{n+1}(x) = B_n(\cdots(B_n(1))\cdots)$$

Das bedeutet das Programm X_n muß mit einer 1 auf dem Registerplatz 1 so oft ausgeführt werden, wie das Argument von B_{n+1} angibt, welches ja zunächst auf dem Registerplatz 1 steht. Für X_{n+1} ergibt sich also

$$X_{n+1} := (A_r; S_1)_1; A_1; (X_n; S_r)_r$$

Damit ist $\|X_{n+1}\| = B_{n+1}$.

Man zeigt nun

1. Ist $f: \mathbb{N}^k \longrightarrow \mathbb{N} \in F_n$, so gibt es eine feste Potenz p von B_n, so daß für alle $(x_1, \ldots, x_k) \in \mathbb{N}^k$ gilt
$$f(x_1, \ldots, x_k) \leq B_n^{(p)}\left(\max_{1 \leq i \leq k}(x_i)\right)$$
2. Für alle n gilt $B_n < B_{n+1}$, daß heißt für alle natürlichen Zahlen p existiert ein $x_0 \in \mathbb{N}$, so daß für alle $x > x_0$ gilt
$$B_n^{(p)} < B_{n+1}(x).$$
3. Wäre $B_{n+1} \in F_n$, so wäre nach 1. B_{n+1} durch ein $B_n^{(p)}$ beschränkt, dies steht aber im Widerspruch zu 2.

Weiterhin kann man durch Induktion zeigen, daß für alle $P \in P_n$ die Semantik von P, $\|P\|$ primitiv rekursiv ist.

Satz

A ist nicht primitiv rekursiv.

Beweis

Sei angenommen,daß A primitiv rekursiv ist.

Dann wäre A URM-berechenbar, und damit wäre $A \in F_i$ für ein geeignet großes i. Das heißt aber, es existiert ein $p \in I\!N$ mit

$$A(n,x) \leq B_i^{(p)}(\max(n,x)).$$

Daraus folgt $A(i+1,x) \leq B_i^{(p)}(x)$, falls $x \geq i+1$, es ist aber

$$B_i^{(p)}(x) < B_{i+1}(x) = A(i+1,x) \quad \text{falls} \quad x > x_0 \geq i+1.$$

Somit ist der Widerspruch gefunden und der Satz bewiesen.

Folgerung

`for`-Schleifen reichen nicht aus um die Große Ackermann-Funktion zu berechnen, man braucht Programme, die `while`-Schleifen verwenden.

Beachte

Die Große Ackermann-Funktion ist auf ganz $I\!N^2$ definiert.

7.2 Korrektheit von Programmen

Das Testen von Programmen erlaubt die Anwesenheit von Fehlern zu belegen. Außer beim Testen *aller* möglichen Eingaben, kann so nicht geschlossen werden, daß kein Fehler vorhanden ist.

7.2.1 Programm-Verifikation

Man versucht mit logischen Formeln die Korrektheit eines Programmes zu beweisen.

Dazu werden dem Programm logische Formeln beigefügt. Sei A ein Programmteil. Dann gehört zu A eine Vorbedingung P und eine Nachbedingung Q. Dann ist

$$\{P\}A\{Q\} \qquad P \text{ und } Q \text{ Prädikate}$$

eine *logische Formel* mit dem Wert "wahr" oder "falsch". Ist $P(X) =$"wahr" für einen Datensatz X, so ist nach Anwendung von A auf X und Ausgabe Y auch $Q(Y)$ "wahr".

Anhand des Aufbaus von Programmen mittels Ablaufkonstruktoren ist für jeden Schritt einzeln nachzuweisen, daß für das Ergebniskonstrukt die Korrektheit gegeben ist.

Einschränkung partielle Koreektheit

$\{P\}A\{Q\}$ ist *partiell korrekt*, wenn gilt:

Unter der Vorraussetzung, daß A terminiert, ist die Korrektheit gegeben.

Verkettungsbedingung

Seien die Konstrukte A_1 und A_2 gegeben, und

$$\{P_1\}A_1\{Q_1\}, \{P_2\}A_2\{Q_2\}$$

partiell korrekt. Dann lautet die *Verkettungsbedingung*

Wird Q_1 "wahr", dann auch P_2.

Damit folgt, daß $\{P_1\}A_1; A_2\{Q_2\}$ partiell korrekt ist.

Verzweigung

Zuweisung

Schleifen

Ein wichtiges Hilfsmittel zum Beweis der Korrektheit von Programmen sind die sogenannten *Schleifeninvarianten*. Diese müssen vom Entwerfer des Programms gefunden werden. Damit kann man die partielle Korrektheit nachweisen.

Um die *absolute Korrektheit* nachzuweisen, verwendet man eine Variable, deren Wert sich von Durchlauf zu Durchlauf ändert. Falls man nachweisen kann, daß dieser Wert streng monotn fällt und eine natürliche Zahl ist, so terminiert der Algorithmus.

Zusammenfassung

Es ist

$$\{P\}A\{Q\}$$

zu erstellen. Dabei ist Q von der Aufgabenstellung her gegeben. Ziel ist es nun die schwächste Vorbedingung P zu suchen, für die $\{P\}A\{Q\}$ korrekt ist.

In der Praxis werden nur die Schleifeninvarianten tatsächlich verfolgt, und die Invarianzen nachgewiesen.

Kapitel 8. beschäftigt sich mit Formalen Hilfsmitteln zur Formulierung von Wissen und zur Ableitung von Aussagen.