Compilerbau

Prof. Dr. W. Merzenich

2. April 2003

Inhaltsverzeichnis

Anmerkung

Dieses Skript wurde von Frank Katritzke in LaTeX gesetzt. Für den Inhalt des Skriptes und deren Prüfungsrelevanz kann keine Garantie gegeben werden. Für Anmerkungen bezüglich inhaltlicher Fehler bitte ich, den Setzer des Skriptes zu informieren. Wir bedanken uns bei den Korrekturlesern, wobei insbesondere Peter Fey hervorzuheben ist.

Kapitel 1

Einführung

1.1 Programmiersprachen, Compiler und Interpreter

Programmiersprachen stellen den sprachlichen Rahmen zur Verfügung, in dem der Programmierer die algroithmische Lösung von praktischen Problemen formal präzise ausdrücken kann. Die sprachlichen Mittel prägen die Form der Lösung und das (abstrakte) Denkmodell. Umgekehrt haben bestimmte Denkmodelle (Paradigmen) ganz bestimmte Sprachen hervorgebracht. So beeinflussen sich Sprache und Denkmodell gegenseitig. Die Struktur einer Programmiersprache wird aber auch stark durch die Forderung bestimmt, daß die Sprachkonstrukte durch Algorithmen analysiert werden können. Daß eine solche Analyse außerdem in annehmbarer Zeit und mit vertretbarem Speicheraufwand möglich sein muß, schränkt die sprachlichen Mittel zusätzlich ein. So ist der Entwurf einer problemorientierten Programmiersprache stets ein Kompromiß zwischen der Ausdruckskraft einerseits und der automatischen Analyse andererseits. Es sind heute einige hundert Programmiersprachen implementiert. Viele davon sind an Universitäten entstanden und dienen eher als Forschungsgegenstand denn als Produktions-Sprache.

Eine Klassifizierung der Sprachen und die Aufstellung von Kriterien, mit denen die Vor- und Nachteile für bestimmte Anwendungen beurteilt werden könnten ist eine interessante Aufgabe. Wir werden aber in dieser Vorlesung nicht die Vielfalt der vorhandenen Programmiersprachen untersuchen, sondern schwerpunktmäßig das Problem der Sprachübersetzung behandeln. Wir werden uns dabei außerdem auf die typischen Konstrukte der bekannten imperativen Sprachen beschränken. Diese Methoden sind teilweise auch für logische und funktionale (applikative) Programmiersprachen anwendbar. Andererseits erfordern natürlich die unterschiedlichen Sprachklassen auch jeweils ganz spezifische Methoden.

In dem Buch von Wilhelm und Maurer [19] wird auf diese Methoden genauer eingegangen. Zum Studium der verschiedenen Programmiersprachen im Vergleich gibt es einige Monographien.

Damit kann der Rahmen dieser Vorlesung genauer abgesteckt werden. Dazu sollen aber gleich einige wichtige Begriffe eingeführt werden.

Die wesentlichen Bestandteile einer Programmiersprache (PS) sind:

Syntax: Charakterisierung der Menge L von zulässigen Programmen. Ein Programm $p \in L$ ist ein Text, der nach ganz bestimmten Regeln erzeugt wurde.

Eingabemenge I.

Ausgabemenge O.

Semantische Abbildung $\mathrm{sem} : L \times I \rightharpoonup O$.
Diese Abbildung (Funktion) ordnet jedem Programm p und jeder Eingabe $x \in I$ einen Ausgabewert $y \in O$ zu: $y = \mathrm{sem}(p, x)$.

Im allgemeinen ist die semantische Abbildung sem nur partiell definiert, d.h. für bestimmte Werte $(p, x) \in L \times I$ ist kein Ausgabewert definiert. Man denke dabei an ein Programm p, das für die Eingabe x in eine Endlos-Schleife gerät und daher nicht (normal) terminiert.

Eingabe- und Ausgabemengen sind meistens auch Text-Mengen, so daß man annehmen kann: es gibt ein (gemeinsames) Alphabet A und A^* ist die Menge aller Texte (Zeichenreihen) mit dem Alphabet A und die angegebenen Mengen sind alle Teilmengen von A^*.

Definition 1.1.1 *Eine* Programmiersprache *ist ein* 4-*Tupel* $\underline{L} = (L, I, O, \mathrm{sem})$. *Dabei sind* L, I, O *Mengen und* $\mathrm{sem} : L \times I \rightharpoonup O$ *ist eine partielle Abbildung.*

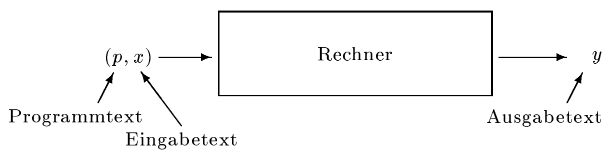
Bemerkung: Partielle Abbildungen werden meist durch den Halbpfeil „$\rightharpoonup$" gekennzeichnet. Die Definition ist noch sehr allgemein und abstrahiert noch von der genauen Ausprägung der Mengen L, I, O etwa als Text.

Sind A, B Mengen, dann bezeichnet $[A \to B]$ die Menge aller Abbildungen (Funktionen) von A nach B und $[A \rightharpoonup B]$ die Menge der partiellen Abbildungen.

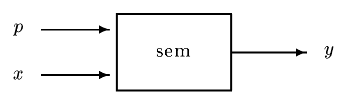
Damit die Definition nicht zu abstrakt bleibt betrachten wir die Programmiersprache PASCAL. A sei die Menge der Buchstaben und Interpunktionszeichen und Ziffern (etwa der ASCII-Zeichensatz). Ein PASCAL-Programm $p \in A^*$ ist eine Folge von Zeichen aus A, d.h. ein Text. Nicht jeder Text ist ein gültiges PASCAL-Programm. Etwa durch Syntax-Diagramme wird genau festgelegt, welche „Form" ein Text haben muß damit er ein PASCAL-Programm darstellt. Sei also $L \subset A^*$ die Menge der PASCAL-Programme. Wenn man als Ein-Ausgabe-Gerät ein herkömmliches Terminal verwendet, dann werden alle Eingaben und alle Ausgaben ebenfalls durch Text dargestellt, so daß man annehmen kann: $I, O \subseteq A^*$. Wir nehmen weiter an, wir verwenden einen Rechner, der so eingerichtet ist, daß er für eine Eingabe $(p, x) \in L \times I$ folgendes ausführt:

- Übersetzen und Laden von p.
- Anwenden auf die Eingabe x.
- Ausgabe von y.

Dann berechnet der Rechner die semantische Funktion sem, nämlich die Semantik der Programmiersprache PASCAL.



oder



Der Rechner ist eine Maschine, die die Funktion sem berechnet. Eine Maschine ist eine abstrakte mathematische Konstruktion, die aber prinzipiell rein mechanisch arbeiten könnte (z.B. Turingmaschine, Registermaschine, konkrete CPU mit Speicher und E/A-Einheit). Eine solche Maschine berechnet eine partielle Funktion $f : X \rightharpoonup Y$ (wobei man wieder $X, Y \subseteq A^*$ annehmen kann).

Ein Algorithmus für eine Funktion $f : X \rightharpoonup Y$ ist ein im intuitiven Sinn mechanisch ausführbares Verfahren, das zu jedem $x \in \mathrm{dom} f$ den Wert $y = f(x)$ „berechnet“, d.h. konstruktiv herstellt. Ein solches Verfahren kann durch ein Maschinenmodell präzise definiert werden. Eine Maschine, die $f : X \rightharpoonup Y$ berechnet ist also ein Algorithmus für f.

Sei $\underline{L} = (L, I, O, \mathrm{sem})$ eine Programmiersprache. Eine Maschine (also ein Algorithmus) die die Funktion $\mathrm{sem} : L \times I \rightharpoonup O$ berechnet, heißt *Interpreter für* $\underline{L}$. Wir sprechen dann auch einfach von dem Interpreter sem.

Ist $\mathrm{sem} : L \times I \rightharpoonup O$, dann kann zu jedem $p \in L$ die partielle Funktion $(\mathrm{sem}' p) : I \rightharpoonup O$ definiert werden durch

$$(\mathrm{sem}' p) x = \mathrm{sem}(p, x).$$

In diesem Sinn ist

$$\mathrm{sem}' : L \to [I \rightharpoonup O].$$

sem' ordnet also jedem Programm $p \in L$ eine partielle Ein-Ausgabe-Funktion zu. sem' ist eigentlich nur eine andere Betrachtung für die Funktion sem. Man sagt auch: $(\mathrm{sem}' p) : I \rightharpoonup O$ ist die partielle Auswertung von sem an der Stelle p. Wir werden im folgenden beide Schreibweisen für sem gleich bezeichnen:

$$\mathrm{sem} : L \to [I \rightharpoonup O]$$
$$\mathrm{sem} : L \times I \rightharpoonup O.$$

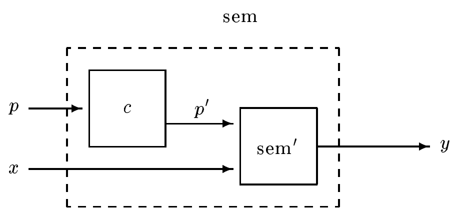
Sei $\underline{L}' = (L', I, O, \mathrm{sem}')$ eine weitere Programmiersprache mit gleichen Ein-Ausgabemengen:

$$\begin{tikzcd}
L \arrow[rr, dashed, "c"] \arrow[dr, "\mathrm{sem}"'] & & L' \arrow[dl, "\mathrm{sem}'"] \\
& {[I \rightharpoonup O]} &
\end{tikzcd}$$

Eine Abb. $c : L \to L'$ mit $\mathrm{sem}' \circ c = \mathrm{sem}$ heißt *semantikerhaltend*, und ein Algorithmus für c heißt ein *Übersetzer* oder *Compiler*. Der Compiler (für) c transformiert also jedes Programm $p \in L$ in ein Programm $c(p) \in L'$, so daß gilt:

$$\mathrm{sem}' c(p) = \mathrm{sem}\ p.$$

Man kann das auch so denken: Ein Interpreter für $\underline{L}'$ kann einen Interpreter für $\underline{L}$ simulieren:

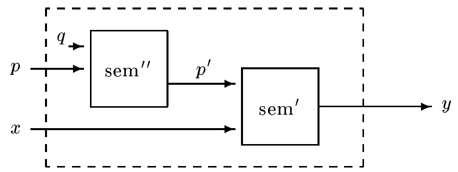


Es gilt also $p' = c(p)$ und $\text{sem}'(p', x) = \text{sem}(p, x) = y$.

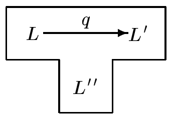
Da der Compiler c ebenfalls durch einen Algorithmus gegeben ist, könnte er durch ein Programm $q \in L''$ gegeben sein, so daß $c = \text{sem}''q$ und

$$\text{sem}'' : L'' \to [L \rightharpoonup L']$$

oder anschaulicher:

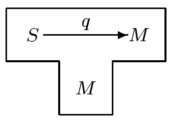


Man sagt dann: c ist ein Compiler von L nach L' geschrieben (implementiert) in der Sprache L''. Dieser Sachverhalt wird symbolisch durch sogenannte T-Diagramme dargestellt:

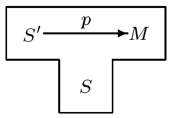


Solche T-Diagramme werden verwendet, um Sprachtransformationen und Portierungen von einer Maschine auf eine andere anschaulich darzustellen. Dafür ein

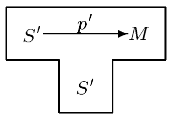
Beispiel: Sei S eine Programmiersprache und M eine Maschinensprache (ebenfalls eine Programmiersprache). Für M existiert ein Interpreter (Maschine). Sei $q \in M$ ein Compiler von S nach M:



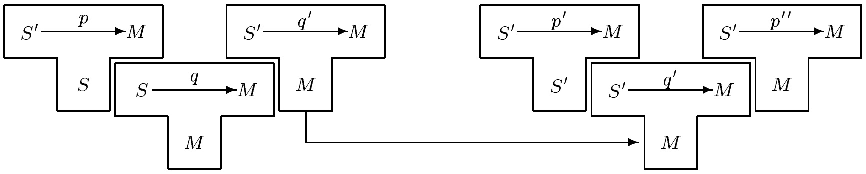
Sei nun S' eine Spracherweiterung von $S (S \subset S')$. Dann kann man einen Compiler von S' nach M in S formulieren (schreiben):



Dieser Compiler ist also in einer „höheren" Programmiersprache geschrieben. Man kann den Compiler nun in seiner eigenen Quellsprache S' schreiben:



Mit diesen Voraussetzungen kann man nun einen Compiler für S' in Maschinensprache erzeugen (berechnen):



Dieses Diagramm beschreibt den folgenden Prozess:

1. p wird mit dem Compiler q in ein (äquivalentes) Maschinenprogramm q' übersetzt.
2. Mit dem neuen Compiler q' wird der Compiler p' in das Maschinenprogramm p'' übersetzt.

`Ende Beispiel`

Auf diese Weise ist es möglich, daß man Compiler, die in ihrer eigenen Quellsprache geschrieben sind schrittweise auf eine bestimmte Zielmaschine zu portieren, indem man zunächst einen Compiler für eine Teilsprache in der Sprache der Zielmaschine schreibt und ihn dann nach dem obigen Vorgehen schrittweise „hochzieht“ (*Bootstrapping*).

Ein Compiler ist also ein Algorithmus, der Programme einer Programmiersprache L_1 in äquivalente Programme einer Sprache L_2 übersetzt. Der Compiler kann dabei selbst in einer Sprache L_3 formuliert sein, d.h. ein Programm der Sprache L_3 sein.

In dieser Vorlesung werden Methoden und theoretische Grundlagen behandelt, die dazu dienen einen Compiler zu konstruieren. Die Vorlesung steht somit in engem Zusammenhang mit den Vorlesungen über

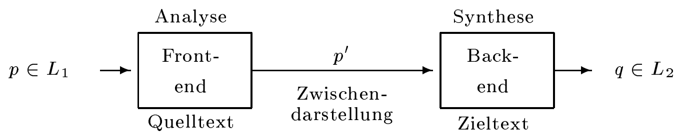
- Algorithmen (Compiler ist ein spezieller, komplexer Algorithmus).
- Formale Sprachen (Syntaktische Struktur der Programme, d.h. der Sprache L).
- Theorie der Programmierung (hier werden mathematische Methoden behandelt, um die Funktion sem zu charakterisieren).
- Berechenbarkeit und Komplexität (theoretische Grundlagen für berechenbare Funktionen, universelle Rechner und Leistungsgrenzen).

Diese Kenntnisse werden nicht vorausgesetzt, aber sie vertiefen das Verständnis für den Compilerbau. Die Voraussetzungen aus der Vorlesung „Formale Sprachen“ werden kurz wiederholt, wo sie benötigt werden; aber es werden hier keine Sätze bewiesen.

1.2 Struktur eines Compilers, Teilaufgaben

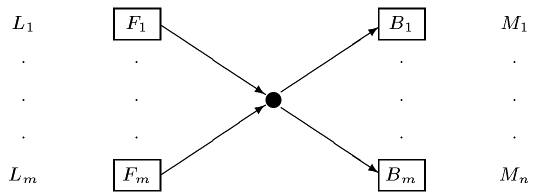
Ein Compiler ist ein Textübersetzungsprogramm, das genügend Information über die Semantik von Quell- und Zielsprache sowie deren syntaktische Struktur enthalten muß. Der Compiler transformiert den Eingabetext $p \in L_1$ in mehreren Schritten über gewisse Zwischenformen $p', p'', ..., p^{(n)} = q$ in einen Programmtext q der Zielsprache L_2.

Die Gesamtaufgabe läßt sich zunächst in zwei logisch getrennte Aufgaben zerlegen: Analyse und Synthese



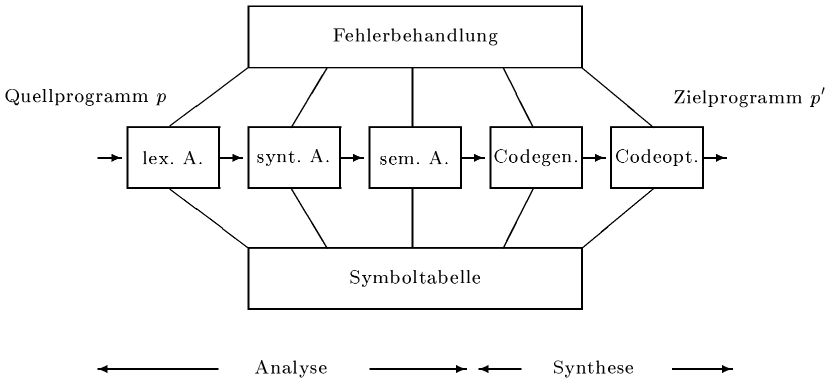
Die Zwischendarstellung repräsentiert die syntaktische Struktur von p, das nach bestimmten Regeln gemäß L_1 aufgebaut ist. p' ist dabei so aufbereitet, daß die anschließende Synthese des Zielprogramms viel einfacher zu behandeln ist als durch die Quell-Darstellung p. Analyse- und Syntheseteil heißen auch Front- bzw. Back-end des Compilers.

Die Zerlegung in Frontend und Backend hat einen großen praktischen Vorteil. Wenn man für eine Sprache L Compiler für mehrere Maschinensprachen $M_1, ..., M_n$ erstellen muß, dann bleibt das Frontend (Analyse der Quellsprache) erhalten und man muß für jede Sprache M_i nur jeweils ein neues Backend programmieren. Umgekehrt kann man bei festgehaltener Zwischendarstellung und einem gegebenen Backend für neue Programmiersprachen lediglich neue Frontends verwenden.



Für m Quellsprachen und n Zielsprachen benötigt man also nicht $m \cdot n$ verschiedene Compiler, sondern nur m Frontends und n Backends (also $m + n$ „halbe Compiler“).

Analyse und Synthese können noch weiter unterteilt werden, um die Gesamtaufgabe in überschaubare Teilaufgaben zu zerlegen. Diese Zerlegung bestimmt dann auch weitgehend die Stoffgliederung der Vorlesung.



Der Compiler kann als eine Folge von Transformationen betrachtet werden. Diese Einteilung ist z.T. willkürlich und nicht in jedem Compiler sind diese Transformationsschritte durch entsprechende Strukturierung der Programme in diese Komponenten realisiert. So können hier getrennte Transformationen in der Implementation zu einer Einheit zusammengefaßt werden.

Die einzelnen Transformationen produzieren eine Folge von Zwischendarstellungen bis hin zum Zielprogramm (in der Zielsprache). Die Transformationen müssen nicht streng nacheinander ausgeführt werden (Compiler-Phasen), sondern können nach dem „Pipeline-Prinzip“ zeitlich verzahnt ablaufen. Die Symboltabelle ist eine globale Datenstruktur für (fast) alle Komponenten und die Fehlerbehandlung kann jede einzelne Phase berücksichtigen. An einem einfachen Beispiel soll kurz der gesamte Übersetzungsprozeß, in die einzelnen Phasen zerlegt, vorgestellt werden. Dieses Beispiel ist also eine Art „Schnelldurchgang“ durch die gesamte Vorlesung. Jede Komponente wird als eine Transformation von einer Zwischendarstellung in die nächste beschrieben.

1.2.1 Lexikalische Analyse

Der Quelltext des Programms ist zunächst eine Folge von elementaren Zeichen (etwa ASCII). Diese Folge ist ein Zeichenstrom, in dem implizit eine syntaktische Struktur enthalten ist. Die beiden ersten Phasen sollen diese Struktur (Baum, Syntaxbaum) explizit aus dem Zeichenstrom konstruieren. Die Aufgabe wird aus praktischen Gründen in zwei Teilaufgaben zerlegt.

Die lexikalische Analyse zerlegt den Zeichenstrom in eine Folge von Teilfolgen. Jede Teilfolge gehört dabei zu einem Muster. Ein Muster ist einfach eine Klasse von bestimmten Folgen. In einer Programmiersprache wie z.B. PASCAL gehört etwa die Zeichenfolge „17395“ zu dem Muster „number“. Dieses Muster umfaßt alle Folgen, die nur aus Ziffern bestehen. Die Folge „startnode“ gehört dagegen zum Muster „identifier“. Jedes Muster ist also eine Sprache (Menge von Zeichenfolgen) und erhält einen Namen. Diese Namen heißen Token. Jede spezielle Zeichenfolge, die zu einem Muster gehört heißt *Lexem*. Die lexikalische Analyse zerlegt also die Eingabe-Zeichenfolge (den Zeichenstrom) in eine Folge von Lexemen und liefert als Ausgabe die Folge der zugehörigen Token, d.h. einen Tokenstrom. Die erste Zwischendarstellung ist also ein Tokenstrom.

Beispiel (PASCAL)

$$\text{Zeichenstrom}: \quad \underbrace{\text{startnode}}_{\text{ident}}\underbrace{:=}_{\text{op}}\underbrace{\text{x0}}_{\text{id}}\underbrace{+}_{\text{op}}\underbrace{\text{inode}}_{\text{ident}}\underbrace{;}_{\text{punct}}$$

Die Lexeme sind die unterklammerten Teilfolgen und unter den Klammern stehen die zugehörigen Token. Die Token sind wieder durch Wörter dargestellt, werden aber als einzelne Zeichen eines neuen Alphabets betrachtet. Leerzeichen (blank), Zeilenvorschub, Tabulatorzeichen usw. dienen nur der Lesbarkeit und werden für die lexikalische Analyse i.a. als Whitespace bezeichnet. Whitespace kann somit selbst als ein Muster betrachtet werden, das aber nicht in den Ausgabe-Tokenstrom eingeht.

Als laufendes praktisches Beispiel soll eine sehr einfache Programmiersprache (für Taschenrechner) definiert werden, an deren Struktur dann die einzelnen Compilerphasen erklärt werden können.

Definition der lexikalischen Struktur in EBNF-ähnlicher Notation:

<identifier> ::= <letter>{<letter> | <digit>},

<number> ::= <digit>{<digit>},

<operator> ::= +| − | ∗ | / | :=,

<punctuation> ::= ;| (|),

<digit> ::= 0|1|...|9,

<letter> ::= A|...|Z|a|...|z.

Token sind: IDENTIFIER, NUMBER, OPERATOR, PUNCTUATION.

- Whitespace ist ein Muster, das hier nicht explizit definiert wird und für das kein Token in die Ausgabe kommt.
- Jedes andere gelesene Symbol (Zeichen) soll in das Muster illegal aufgenommen werden. Man kann dafür ein Token, etwa ILLEGAL verwenden, oder den Fehler (Fehlerbehandlung) direkt durch die lexikalische Analyse anzeigen (Entwurfsentscheidung).

Auf der folgenden Seite ist die lexikalische Analyse an einem Beispiel dargestellt.

```
% { /* Lexikalische Syntax (K. Lemone) LEX-Format */
/* Definition der Token */
#define IDENTIFIER 257
#define NUMBER 258
#define OPERATOR 259
#define PUNCT 260
#define ILLEGAL 261
% }
/* Reguläre Ausdrücke */
Digit [0-9]
Letter [A-Za-z]
Operator [+ * -/] | „:=“
Punctuation [;()]
/* Regelteil */
%%
{Letter} ({Letter} | {Digit})*    { return IDENTIFIER;}
{Digit}+    { return NUMBER; }
{Operator}+    { return OPERATOR; }
{Punctuation} { return PUNCT; }
   {;}
[   \ n \ t \ f \ r]    {;}
.    { return ILLEGAL; }
% %
```

Eingabe:

x1 := a+ bb ∗ 12;

x2 := a/2 + (2-x1)

Lexikalische Analyse, Tokenstrom

IDENTIFIER 257 x1

OPERATOR 259 :=

IDENTIFIER 257 a

OPERATOR 259 +

IDENTIFIER 257 bb

OPERATOR 259 *

NUMBER 258 12

PUNCTUATION 260 ;

IDENTIFIER 257 x2

OPERATOR 259 :=

IDENTIFIER 257 a

OPERATOR 259 /

NUMBER 258 2

OPERATOR 259 +

PUNCTUATION 260 (

NUMBER 258 2

OPERATOR 259 −

IDENTIFIER 257 x1

PUNCTUATION 260)

Die Einheit für die lexikalische Analyse heißt auch *Scanner*. Der Scanner in dem obigen Beispiel würde aus der dargestellten Spezifikation automatisch generiert mit Hilfe des UNIX-Werkzeuges LEX. LEX ist ein Scanner-Generator und wird später noch ausführlicher beschrieben.

Zwei Bemerkungen sollen hier noch gemacht werden. Während der Scanner in der Ausgabe nur den jeweiligen Token ausgibt (meist eine Zahl, siehe Beispiel) wird für die weitere Bearbeitung manchmal auch das Lexem benötigt (etwa für identifier). Der Scanner sollte also Token und Lexem bereitstellen. Ist das Muster eine Zahl (⟨number⟩) so kann der Scanner die Zeichenfolge (das Lexem) gleich in die interne Zahldarstellung umformen. In diesem Fall berechnet der Scanner einen „Attributwert“ des Token, den er in der Ausgabe zur Verfügung stellen kann.

Ein Scanner, der mit LEX erzeugt wird leistet das und liefert:

- yylex() Funktion ruft den Scanner auf und liefert den nächsten Token.
- yytext das Lexem zum Token.
- yylval berechneter Wert.

1.2.2 Syntaktische Analyse

Die syntaktische Struktur eines Programms kann aus der Tokenfolge abgeleitet werden. Die syntaktische Struktur wird (i.d.R.) durch eine kontextfreie Grammatik definiert und daher gehört zu einem syntaktisch korrekten Programm ein Ableitungsbaum. Ziel der syntaktischen Analyse ist es, diesen Ableitungsbaum aus der Tokenfolge zu bestimmen. Die präzisen Definitionen für „kontextfreie Grammatik“ und „Ableitungsbaum“ folgen später, aber anhand des laufenden Beispiels können diese Begriffe intuitiv erklärt werden.

Die syntaktische Struktur unserer Beispielsprache in einer Form, die der BNF (Backus Naur Form) sehr ähnlich ist, ist auf der folgenden Seite definiert. Diese Form ist wieder von einem UNIX-Werkzeug YACC lesbar. YACC kann nach dieser Spezifikation ein Programm erzeugen, das die syntaktische Analyse durchführt. YACC wird uns auch später noch beschäftigen. Die Zeile:

$$\langle \text{statement} \rangle ::= IDENTIFIER\ \ ASSIGN \langle \text{expression} \rangle$$

bedeutet, daß ein Statement mit den Token IDENTIFIER und ASSIGN beginnt und dann folgt ein Ausdruck (expression).

```
%{
/* Parser (K. Lemone) YACC Format */
%}
%token IDENTIFIER NUMBER OPERATOR ASSIGN PUNCT
%token ILLEGAL
%start program
%%
program:
   statements
   ;
statements:
   statement ';' statements
   | statement
   ;
statement:
   IDENTIFIER ASSIGN expression
   ;
expression:
   expression '+' term
   |expression '-' term
   term
   ;
```

term:

term '*' factor

| term '/' factor

| factor

;

factor:

'(' expression ')'

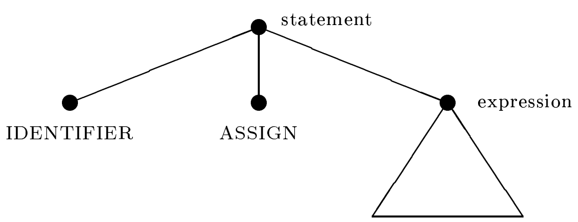
| IDENTIFIER

| NUMBER

;

%%

Der Syntaxanalyseteil eines Compilers heißt auch *Parser* (*Zerleger*, *Zerteiler*). Wie der Parser arbeitet, wird erst in einem späteren Kapitel genauer erklärt. Es wird durch eine geeignete interne Datenstruktur ein Baum erzeugt der die syntaktische Struktur des Tokenstromes beschreibt. Die inneren Knoten des Baumes stehen dabei für die linken Seiten der BNF (sogenannte *Nonterminals*) und die Söhne eines inneren Knoten stellen stets die rechte Seite einer Regel dar, deren linke Seite der Vaterknoten ist:



Die Blätter sind mit Token markiert. Wenn man zu jedem Token das Lexem schreibt das in der Eingabe gelesen wurde, dann liefert die Folge der Blätter von links nach rechts gelesen den ursprünglichen Eingabetext (ohne Formatierung, denn whitespace wurde nicht berücksichtigt). Auf der folgenden Seite ist der Syntaxbaum durch Einrückungen dargestellt, so wie Verzeichnisbäume manchmal dargestellt werden. Die Verbindungslinien wurden von Hand hinzugefügt.

x1 := a+ bb * 12;

x2 := a/2 + (2-x1)

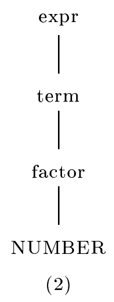
```
Syntaxanalyse, Ableitungsbaum

program
└─ stmts
   ├─ stmt
   │  ├─ x1
   │  ├─ :=
   │  └─ expr
   │     ├─ expr
   │     │  ├─ term
   │     │     ├─ factor
   │     │        └─ a
   │     ├─ +
   │     └─ term
   │        ├─ term
   │        │  ├─ factor
   │        │     └─ bb
   │        ├─ *
   │        └─ factor
   │           ├─ 12
   ├─ ;
   └─ stmts
      └─ stmt
         ├─ x2
         ├─ :=
         └─ expr
            ├─ expr
            │  └─ term
            │     ├─ term
            │     │  └─ term
            │     │     ├─ a
            │     ├─ /
            │     └─ factor
            │        └─ 2
            ├─ +
            └─ term
               ├─ factor
                  ├─ (
                  ├─ expr
                  │  ├─ expr
                  │  │  ├─ term
                  │  │     ├─ factor
                  │  │        ├─ 2
                  │  ├─ -
                  │  └─ term
                  │     ├─ factor
                  │        ├─ x1
                  └─ )

Parser erfolgreich beendet
```

Bemerkung: Die Token wurden durch die zugehörigen Lexeme ersetzt. Die Folge der Blätter liefert also den Eingabetext. `Ende Bemerkung`

Man beachte, daß der Ableitungsbaum eine viel „stärkere“ Struktur besitzt als die Eingabe-Zeichenfolge. Aber implizit konnte er aus dieser Folge berechnet werden. Wenn man in diesem Baum noch „Ketten“ der Art:



„zusammenzieht“ und auch Klammern wegläßt erhält man eine kompaktere Baumdarstellung, die aber die Reihenfolge von Berechnungsschritten immer noch korrekt widergibt. Eine solche abgemagerte Version ist dann der Syntaxbaum. Aus ihm ließe sich prinzipiell schon ein Assembler-Programm konstruieren (Codeerzeugung).

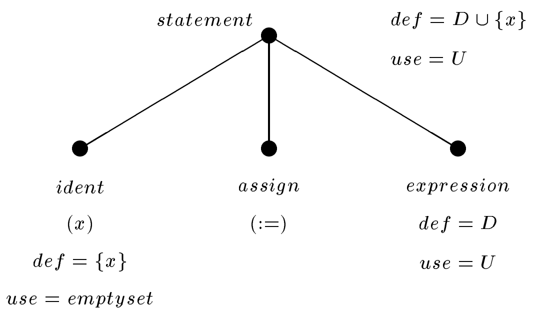
1.2.3 Semantische Analyse

Die semantische Analyse (manchmal auch statische Semantik genannt) soll weitere Eigenschaften eines Programms aus dem Text extrahieren. Genau genommen werden auch hier syntaktische Eigenschaften überprüft (da sie aus dem Text und nicht erst aus der dargestellten Funktion, also der Semantik hervorgehen). Es gibt aber syntaktische Konstruktionen, die sich nicht durch kontextfreie Grammatiken (sondern durch kontextsensitive Grammatiken) beschreiben lassen, die aber für reale Programmiersprachen sehr wichtig sind.

Um solche kontextsensitiven Eigenschaften zu überprüfen gibt es prinzipiell mehrere Möglichkeiten. Man kann einmal zur Festlegung der Syntax den mächtigeren Grammatiktyp (nämlich kontextsensitiv) verwenden. Dann werden aber automatische Analyseverfahren (Algorithmen) äußerst ineffizient. Daher ist dieser Weg eigentlich nur von theoretischem Interesse. Man kann aber auch die Syntax durch eine kontextfreie Grammatik definieren und dann durch zusätzliche Einschränkungen bestimmte (grammatisch korrekte) Programme wieder auschließen. Die semantische Analyse überprüft nun gerade solche (syntaktischen) Eigenschaften, die ein Programm haben soll, die aber nicht durch die BNF ausdrückbar sind. Als Beispiel sei hier nur genannt, die Verwendung von Variablenbezeichnern. In einem PASCAL-Programm etwa muß eine Variable deklariert werden, bevor sie verwendet wird. Eine solche „Fernwirkung“ kann man aber nicht durch eine kontextfreie Grammatik erzwingen. Ähnliches gilt für die Typkonsistenz in Ausdrücken und bei Wertzuweisungen.

Die Techniken zur semantischen Analyse bestehen darin, daß man den Syntaxbaum zusätzlich mit gewissen Attributen dekoriert, d.h. in jedem Knoten kann man sich eine endliche Menge von Variablen vorstellen (oder einen Record), die während der Analyse ganz bestimmte Werte erhalten (Attributauswertung). Die Verteilung dieser Attributwerte im Syntaxbaum liefert dann Kriterien für die statische Semantik.

Auch dazu soll unser Beispiel fortgesetzt werden. Jeder Knoten des Syntaxbaumes erhält zwei Attribute: use und def. Die Attributwerte sind Mengen von Bezeichnern.



Diese Mengen werden (rekursiv) von unten nach oben berechnet. Zu jeder Regel der Grammatik (BNF) definiert man durch Gleichungen, wie aus den Attributwerten der rechten Seite die Attributwerte der linken Seite berechnet werden. Für unser laufendes Beispiel ist unten die so „attributierte Grammatik" angegeben. Dazu noch einige Erklärungen. Die Menge def enthält immer die Menge der Variablen, die in dem entsprechenden Knoten definiert sind. Die Menge use enthält entsprechend Variablen, die schon in einem Ausdruck verwendet wurden. program → use und program → def geben also Auskunft über Variablen, die verwendet wurden ohne definiert zu sein, und solche, die definiert wurden aber nicht verwendet wurden.

Die folgende Spezifikation ist wie folgt zu lesen: In den Gleichungen hinter einer Regel

$$\alpha ::= \alpha_1 \alpha_2 \alpha_3$$

ist $$ eine Referenz auf die Attribute von α, $1, $2, $3 entspechend auf α_1, α_2 und α_3. In

$$statements := statement\$\$ \rightarrow use = \$1 \rightarrow use$$

bedeutet das: Die Statement-Folge, die nur aus einem Statement besteht erhält die gleiche use-Menge wie das Statement u.s.w.

```
%%
program:
  statements    $$ - >use = $1 - >use;
                $$ - >def = $1 - >def;
;
statements:
  statement ';' statements
                $$ - >use = set_union($1 - > use, $3 - > use);
                $$ - >def = set_union($1 - > def, $3 - > def);
    | statement $$ - > use = $1 - > use;
                  $$ - > def = $1 - > def;
;
statement:
IDENTIFIER ASSIGN expression $$ - > def = set_union($3 - > def,$1.lexem);
```

```
;
expression:
expression '+' term         $$ -> use = set_union($1 -> use, $3 -> use);
| expression '-' term       $$ -> use = set_union($1 -> use, $3 -> use);
| term                      $$ -> use = $1 -> use;
;
term:
term '*' factor             $$ -> use = set_union($1 -> use, $3 -> use);
| term '/' factor           $$ -> use = set_union($1 -> use, $3 -> use);
| factor                    $$ -> use = $1 -> use;
;
factor:
'(' expression ')'          $$ -> use = $2 -> use;
| IDENTIFIER                $$ -> use = { $1 -> lexem };
| NUMBER                    $$ -> use = $1 -> use;
;
%%
```

Mit dieser Spezifikation wurde der folgende dekorierte Syntaxbaum erzeugt. Das Ergebnis besagt:

- x2 wird definiert, aber nicht verwendet

$$x2 \in program{-}{>}\,def, x2 \notin program{-}{>}\,use.$$

- a, bb werden verwendet ohne definiert zu sein:

$$a, bb \in program{-}{>}\,use, a, bb \notin program{-}{>}\,def.$$

Beispiel:[Syntaxbaum mit ausgewerteten Attributen]

```
Eingabe:
x1 := a + bb * 12;
x2 := a/2 + (2-x1)
Parser initialisiert
program --> use: {x1, a, bb,}   --> def: { x1, x2,}
 stmts --> use: {x1, a, bb,}   --> def: { x1, x2,}
  stmt --> use: { a, bb,}   --> def: { x1,}
   x1 --> use: { x1,}
   :=
   expr --> use: { a, bb,}
```

```
    expr --> use: { a,}
      term --> use: { a,}
        factor --> use: { a,}
          a --> use: { a,}
    +
    term --> use: { bb,}
      term --> use: { bb,}
        factor --> use: { bb,}
          bb --> use: { bb,}
      *
      factor
        12
;
stmts --> use: {x1, a,}   --> def: { x2,}
  stmt --> use: {x1, a,}   --> def: { x2,}
    x2 --> use: {x2,}
    :=
    expr --> use: { x1, a,}
      expr --> use: { a,}
        term --> use: { a,}
          term --> use: { a,}
            factor --> use: { a,}
              a --> use: { a,}
          /
          factor
            2
      +
      term --> use: { x1,}
        factor --> use: { x1,}
          (
          expr --> use: { x1,}
            expr
              term
                factor
                  2
            -
            term --> use: { x1,}
              factor --> use: { x1,}
                x1 --> use: { x1,}
          )
parse erfolgreich
```

Ende Beispiel

1.2.4 Code-Erzeugung

Nach der Syntaxanalyse und der semantischen Analyse liegt ein mit Attributen dekorierter Syntaxbaum vor. Dieser kann eventuell weiter vereinfacht werden oder umgeformt werden, so daß aus dem Baum ein azyklischer Graph wird. Solche Transformationen können bereits eine „Optimierung“ der Zwischendarstellung bewirken. Wir gehen jetzt von einer solchen Zwischendarstellung aus, die logisch immer noch als attributierter Syntaxbaum betrachtet werden kann.

Die Code-Erzeugung definiert zunächst eine (abstrakte) Maschinensprache, die aber in ihrer Struktur der konkreten Maschinensprache vieler Zielmaschinen schon sehr ähnlich ist. Man denkt sich eine CPU, die bestimmte Operationen (arithmet. und logische u.s.w.) mit maximal je zwei Registern ausführen kann und die das Ergebnis in einem dritten Register ablegt. Die Register sind Speicherplätze, die durch Adressen referenziert werden können:

$$A := B + C.$$

Instruktionen dieser Form nennt man *Drei-Adreß-Code* (*TAC*). Die Codeerzeugung generiert anhand des dekorierten Syntaxbaumes (und ggf. mit der Hilfe der Symboltabelle) eine Folge solcher TACs. Unter den Instruktionen können natürlich auch Sprungbefehle und Unterprogramm-Aufrufe vorkommen. Die Technik der Code-Erzeugung soll erst in dem entsprechenden Kapitel behandelt werden. Auch soll unser laufendes Beispiel hier nicht fortgeführt werden, da zunächst eine Maschine und der Instruktionssatz definiert werden müßten.

Die Code-Erzeugung entdeckt bestimmte Teilstrukturen im Syntaxbaum und für diese stehen bestimmte Code-Sequenzen zur Verfügung, die dann entsprechend zusammengefügt werden.

1.2.5 Code-Optimierung

Das Wort Optimierung wird im Compilerbau etwas mißbraucht. Ein Optimum ist ein Maximum unter bestimmten Nebenbedingungen. Bei der Code-Optimierung wird versucht, die Code-Sequenzen so umzuformen, daß die Ausführungszeiten verkürzt werden. Nebenbedingung ist, daß die Semantik erhalten bleibt. Keines der bisher bekannten Verfahren kann hier ein Optimum garantieren. Optimierung bedeutet lediglich: Verbesserung. Die Code-Optimierung setzt eigentlich voraus, daß man Aussagen über alle möglichen Ausführungsfolgen eines Programms (oder eines Teiles davon) machen kann. Solche Aussagen erhält man durch eine Analyse der Datenabhängigkeiten (Datenfluß-Analyse) und der möglichen Ausführungsfolgen (Kontrollfluß-Analyse). Die Algorithmen zur Optimierung verwenden solche Analysen um z.B. unnötige Wiederholungen von Wertzuweisungen zu vermeiden, wenn erkannt wird, daß der jeweils zugewiesene Wert unverändert bleibt.

Code-Optimierung kann für den Zwischencode (TAC) durchgeführt werden, aber auch nochmals auf dem Programm für die Zielmaschine. Die Güte dieser Optimierungsalgorithmen bestimmt oft die Qualität eines Compilers. Während das Front-End (also der Analyseteil) sehr gut durch die Theorie der Formalen Sprachen verstanden wird, und daher auch gute Werkzeuge zum automatischen Generieren der Analysealgorithmen existieren, ist die Code-Optimierung noch Gegenstand der aktuellen Forschung. Die verwendeten Methoden sind meist heuristisch.

1.3 Grundbegriffe und Ergebnisse aus der Theorie der formalen Sprachen

Formale Sprachen sind Wortmengen über festgelegten Alphabeten. Man ist insbesondere an solchen Sprachen interessiert, die sich mit endlichen Mitteln konstruktiv erzeugen oder erkennen lassen.

Durch Wahl dieser endlichen Mittel sind einzelne Sprachklassen charakterisiert. Von besonderer Bedeutung sind die Klassen der Chomsky-Hierarchie und für den Compilerbau i.W. nur die beiden unteren Klassen (Typ 3 und Typ 2 Sprachen). In diesem Abschnitt sollen einige Grundbegriffe aus der Theorie der formalen Sprachen zusammengestellt werden. Weitere spezielle Ergebnisse werden dann eingeführt wenn sie gebraucht werden.

Ein *Alphabet* ist eine endliche Menge $A = \{a_1, ..., a_n\}$. Um den Begriff der lexikographischen Ordnung definieren zu können, nimmt man außerdem eine lineare Ordnung auf A an: $a_1 < a_2 < ... < a_n$ (Mathematisch ist ein Alphabet eine endliche Ordinalzahl).

Für $k \in I\!N$ ist A^k die Menge aller Folgen von k Zeichen aus A. $u \in A^k$ hat die Form: $u = u_1...u_k$ ($u_i \in A$). u heißt ein *Wort* der Länge k (über A). Die Menge der endlichen (endlich langen) Wörter über A ist dann

$$A^* = \bigcup_{k \in I\!N} A^k.$$

Die Vereinigung wird als disjunkt angenommen und $k = 0$ ist eingeschlossen: $A^0 = \{\epsilon\}$ als Element. ϵ hat die Länge 0 und heißt das *leere Wort*. Die *Wortlänge* ist eine Funktion

$$|\ | : A^* \to I\!N; |u| = k \iff u \in A^k.$$

Auf A^* wird ein Produkt (*Komplexprodukt* oder *Konkatenation*) eingeführt: $u, v \in A^*$, $u = u_1...u_k$, $v = v_1...v_m$ dann sei

$$uv = u_1...u_k v_1...v_m; u\epsilon = u = \epsilon u$$

einfach die Hintereinanderschreibung der Wörter u und v. Es gilt dann: $|uv| = |u| + |v|$. Eine Teilmenge $L \subseteq A^*$ heißt *formale Sprache*. Man definiert das Komplexprodukt auch für Sprachen: $L, M \subseteq A^*$

$$LM = \{uv | u \in L, v \in M\}.$$

Es gilt dann auch: $\{\epsilon\}L = L = L\{\epsilon\}$ und $L^0 = \{\epsilon\}$. Man definiert die *Iteration* (oder den *Kleene-Sternoperator*) für Sprachen:

$$L^* = \bigcup_{k \in I\!N} L^k.$$

Man beachte aber, daß diese Vereinigung i.a. nicht mehr disjunkt ist (Beispiel?).

Ist X eine beliebige Menge, dann sei $\mathcal{P}(X)$ die *Potenzmenge* von X, also die Menge aller Teilmengen von X. Somit ist $\mathcal{P}(A^*)$ die Menge aller Sprachen über dem festen Alphabet A. $\mathcal{P}(A^*)$ ist eine Boolesche Algebra mit den Operationen $\cup$, $\cap$, $\overline{}$ sowie $\emptyset$ (als 0) und A^* (als 1). Mit dem Komplexprodukt und $\{\epsilon\}$ als neutralem Element ist $\mathcal{P}(A^*)$ ein Monoid und mit der Iteration eine Kleene-Algebra. $\mathcal{P}(A^*)$ besitzt also eine sehr reiche algebraische Struktur. Außerdem ist $\mathcal{P}(A^*)$ eine Halbordnung (po) mit $\subseteq$ als Ordnungsrelation und diese ist sogar ein vollständiger Verband. Insbesondere hat jede aufsteigende Kette von Sprachen

$$L_0 \subseteq L_1 \subseteq ... \subseteq L_i \subseteq L_{i+1} \subseteq ...$$

ein (eindeutig bestimmtes) Supremum, nämlich:

$$L = \bigcup_{i \in I\!N} L_i.$$

Wir können auf die vielen interessanten Resultate, die sich aus diesen Strukturen ergeben nicht ausführlich eingehen. Wir werden daher nur solche Ergebnisse näher betrachten, die für die weitere Vorlesung ganz wesentlich sind.

Die Kleene-Operationen auf $\mathcal{P}(A^*)$ sind $\cdot$ (Komplexprodukt), $\cup$ (Vereinigung) und * (Iteration, Kleene-Stern). Sei $A = \{a_1, ..., a_n\}$ dann betrachten wir die folgende endliche Menge $\mathcal{B}$ von Sprachen

$$\mathcal{B} = \{\emptyset, \{a_1\}, ..., \{a_n\}\}.$$

$\emptyset$ ist die leere Sprache (enthält kein Element) und $\{a_i\}$ enthält genau ein Wort der Länge 1 nämlich $a_i \in A \subseteq A^*$.

Definition 1.3.1 *Eine Sprache $L \subseteq A^*$ heißt* regulär *dann und nur dann, wenn sie aus den Basissprachen aus $\mathcal{B}$ allein durch endlich viele Anwendungen der Kleene-Operationen erzeugt werden kann.*

So ist jede endliche Sprache regulär, denn jedes Wort läßt sich durch Konkatenation der Buchstaben (Alphabetzeichen) erzeugen und die Menge schließlich durch endliche Vereinigung. Auch $\{\epsilon\}$ ist regulär, denn $\emptyset^* = \{\epsilon\}$, wird also durch Iteration aus der leeren Menge erzeugt.

Übung: Kann man jede Sprache $L \subseteq A^*$ auf diese Weise erzeugen? Kann man unendliche Sprachen auf diese Weise erzeugen?

Jede reguläre Sprache hat eine endliche Beschreibung nämlich die Konstruktionsvorschrift und diese ist einfach ein algebraischer Term.

Definition 1.3.2 *Sei $A = \{a_1, ..., a_n\}$ ein Alphabet. Die Menge* Reg$A$ *der regulären Ausdrücke über A ist wie folgt definiert:*

1. $0, 1 \in \mathrm{Reg}A$
2. $A \subseteq \mathrm{Reg}A$ *(*$a \in A \Rightarrow a \in \mathrm{Reg}A$*)*
3. $\alpha, \beta \in \mathrm{Reg}A \Rightarrow (\alpha + \beta) \in \mathrm{Reg}A$ *und* $(\alpha\beta) \in \mathrm{Reg}A$
4. $\alpha \in \mathrm{Reg}A \Rightarrow (\alpha^*) \in \mathrm{Reg}A$
5. *Nur Ausdrücke, die nach 1. – 4. in endlich vielen Schritten gebildet werden können gehören zu* Reg$A$.

Die Ausdrücke bilden selbst eine Kleene-Algebra (nämlich die Termalgebra) und man kann jetzt jedem regulären Ausdruck eindeutig eine reguläre Menge (Sprache) zuordnen durch den (Homo-)Morphismus:

$$\varphi : \mathrm{Reg}A \to \mathcal{P}(A^*)$$

der induktiv über den Aufbau der Terme (Ausdrücke) definiert wird:

1. $\varphi(0) = \emptyset$; $\varphi(1) = \{\epsilon\}$
2. $\varphi(a) = \{a\}$ $(a \in A)$
3. $\varphi(\alpha + \beta) = \varphi(\alpha) \cup \varphi(\beta)$
   $\varphi(\alpha\beta) = \varphi(\alpha) \cdot \varphi(\beta)$
4. $\varphi(\alpha^*) = \varphi(\alpha)^*$

Reguläre Ausdrücke sind also endliche Zeichenreihen, die es gestatten, auch unendliche Sprachen (z.B. $\varphi(A^*)$) zu charakterisieren. Beispiele für reguläre Ausdrücke haben wir schon kennengelernt bei der Spezifikation der lexikalischen Syntax für den Scannergenerator LEX:

$$\{letter\}(\{letter\}|\{digit\})^* \hat{=} IDENTIFIER.$$

Die LEX-Syntax ist hier etwas anders als wir die regulären Ausdrücke definiert haben, aber man kann die Formen leicht ineinander übersetzen: $\{letter\}$ und $\{digit\}$ sind Abkürzungen, also Namen für bereits definierte Ausdrücke: $\{letter\}$ steht für $A+B+...+Z+a+b+...+z$. Analog ist $\{digit\}$ definiert. $\{letter\}|\{digit\}$ steht dann für $\{letter\}+\{digit\}$ und der Kleene-Stern ist so auch für reguläre Ausdrücke definiert.

Die Spezifikationssprache LEX ist eine erweiterte Klasse von regulären Ausdrücken, die sich aber alle auf die Kleene-Operationen also auf Reg$A$ zurückführen lassen. Daraus folgt dann auch:

Die lexikalischen Muster sind stets reguläre Mengen (Sprachen).

Wir können die Funktion eines Scanners jetzt etwas formaler beschreiben. Der Scanner wird spezifiziert durch eine endliche Menge von regulären Ausdrücken:

$$\alpha_1, ..., \alpha_p,$$

$L_i = \varphi(\alpha_i) \subseteq A^*$ seien die zugehörigen Muster. Zu jedem Muster L_i wird ein neues Zeichen (Token) $t_i \in T = \{t_1, ..., t_p\}$ festgelegt. Sei jetzt $u \in A^*$ ein Eingabewort (Zeichenstrom). Der Scanner zerlegt u in Teilwörter

$$u = w_1 w_2 ... w_s \quad (w_j \in A^*)$$

so daß jedes w_j einem Muster L_{i_j} zugeordnet ist: $w_j \in L_{i_j}$. Ausgabe ist die Folge der zugehörigen Token $t_{i_1}...t_{i_s} \in T^*$.

Übung: Ist die so definierte Aufgabe eines Scanners immer eindeutig lösbar, wenn man fordert, daß die L_i paarweise disjunkt sind?

Reguläre Ausdrücke sind nur eine Methode, um Sprachen mit endlichen Mitteln darzustellen. Es gibt zwei weitere wesentliche Methoden zur Sprachbeschreibung

- Erzeugung: Kombinatorisches Regelsystem mit dem man bestimmte Zeichenfolgen (Wörter) in endlich vielen Schritten ableiten kann (Grammatiken).
- Erkennung: Maschine, die ein vorgelegtes Wort $u \in A^*$ liest und dann entscheidet, ob sie das Wort akzeptiert oder nicht (Automaten).

Wir beschreiben zuerst generative Systeme. Sei V ein (endliches) Alphabet und $P \subseteq V^* \times V^*$ eine endliche, nichtleere Relation (Menge von Wort-Paaren). Ein Element $(u, w) \in P$ heißt auch eine *Ersetzungsregel* und $S = (V, P)$ heißt ein *Ersetzungssystem* (*rewrite-system*, *Semi Thue System*).

P erzeugt auf V^* eine Einschritt-Ableitungsrelation $\rightarrow_P \subseteq V^* \times V^*$ wie folgt:

$$x \underset{P}{\rightarrow} y \iff x = x'ux'', y = x'wx'' \text{ und } (u, w) \in P$$

d.h. wenn y aus x dadurch hervorgeht, daß ein Teilwort u in x, das linke Seite einer Regel $(u, w) \in P$ ist, durch die rechte Seite w ersetzt wird. Wir schreiben die Relation $\rightarrow_P$ ab jetzt meist in der Infixschreibweise. Mit $x' = x'' = \epsilon$ gilt dann $P \subseteq \rightarrow_P$, also $(u, w) \in P \Rightarrow u \rightarrow_P w$.

Durch endlich viele Ableitungsschritte hintereinander erhält man die Mehrschritt-Ableitungsrelation $\rightarrow_P^* \subseteq V^* \times V^*$ (als reflexive und transitive Hülle von $\rightarrow_P$):

$$x \overset{*}{\underset{P}{\rightarrow}} y \iff \text{ entweder } x = y \text{ oder ex. Folge } x_0, ..., x_n \in V^* \text{ mit } x = x_0 \underset{P}{\rightarrow} x_1 \underset{P}{\rightarrow} ... \underset{P}{\rightarrow} x_n = y.$$

Man kann jetzt Sprachen definieren durch die Menge aller Wörter, die man ausgehend von einem Startwort ableiten kann: Sei $w_0 \in V^*$, dann definiert man:

$$L(S, w_0) = \{u \in V^* | w_0 \overset{*}{\underset{P}{\rightarrow}} u\}.$$

Ist $u \in L(S, w_0)$, dann gibt es eine Ableitung

$$w_0 \underset{P}{\rightarrow} x_1 \underset{P}{\rightarrow} x_2 \underset{P}{\rightarrow} \ldots \underset{P}{\rightarrow} x_n = u.$$

Eine Ableitung ist also die explizite Auflösung von $w_0 \rightarrow_P^* u$ in Einzelschritte. Eine solche Ableitung kann als formaler Beweis aufgefaßt werden für $u \in L(S, w_0)$. Um zu beweisen $u \notin L(S, w_0)$ muß man zeigen, daß es keine Ableitung gibt. Ersetzungssysteme erzeugen rekursiv aufzählbare Mengen, und das bedeutet, daß das Auffinden eines Beweises für $u \in L(S, w_0)$ u.U. sehr aufwendig sein kann, prinzipiell aber durch einen Algorithmus gefunden werden kann. Beweise für $u \notin L(S, w_0)$ können aber nicht immer durch Algorithmen gefunden werden.

Ein Algorithmus, der einen formalen Beweis erzeugt für $u \in L(S, w_0)$ heißt ein *Parser*. Eine *Grammatik* ist ein spezielles Ersetzungssystem, bei dem das Alphabet V in zwei disjunkte Teile von *terminalen* und *nichtterminalen* Zeichen zerlegt ist. Außerdem wird ein spezielles nichtterminales Zeichen als Startwort (Länge 1) vorgegeben.

Definition 1.3.3 *Eine Grammatik ist ein 4-Tupel $G = (N, T, P, \sigma)$. Dabei sind N und T (endliche, nichtleere) Alphabete. Sei $V = N \cup T$, dann ist $P \subseteq V^* \times V^*$ eine endliche Relation und $\sigma \in N$.*

N heißt Alphabet der *Nichtterminalzeichen*.

T heißt Alphabet der *Terminalzeichen*.

P heißt Menge der *Produktionen* (*Regeln*).

σ heißt *Startsymbol*.

Sei $G = (N, T, P, \sigma)$ eine Grammatik, dann definiert man zwei Sprachen, die von G erzeugt werden:

$S(G) = \{w \in V^* | \sigma \rightarrow_P^* w\}$ *Menge der Satzformen*,

$L(G) = S(G) \cap T^*$, *Menge der terminalen Satzformen*.

Üblicherweise heißt $L(G)$ die von G *erzeugte Sprache*. Zwei Grammatiken G_1 und G_2 heißen *äquivalent* genau dann, wenn sie die gleiche (terminale) Sprache erzeugen:

$$G_1 \sim G_2 \iff L(G_1) = L(G_2).$$

Regeln werden oft in der *Backus-Notation* aufgeschrieben: Sei $P_u = \{v \in V^* | (u, v) \in P\}$. Diese Mengen der „rechten Seiten“ zu $u \in V^*$ sind stets endlich, da $P \subseteq V^* \times V^*$ endlich ist. Man faßt alle Regeln mit gemeinsamer linker Seite zusammen:

Wenn gilt: $P_u = \{v_1, v_2, ..., v_m\}$ dann schreibt man dafür:

$$u \rightarrow v_1 | v_2 | ... | v_m.$$

Beispiel: $N = \{\sigma, \alpha\}$, $T = \{a, b, c\}$

$$\begin{aligned} P: \quad \sigma &\rightarrow abc | a\sigma\alpha \quad (1,2) \\ b\alpha c &\rightarrow bbcc \quad (3) \\ c\alpha &\rightarrow \alpha c \quad (4) \end{aligned}$$

Ableitung:

$$\underline{\sigma} \underset{2}{\rightarrow} a\underline{\sigma}\alpha \underset{1}{\rightarrow} aab\underline{c\alpha} \underset{4}{\rightarrow} aa\underline{b\alpha c} \underset{3}{\rightarrow} aabbcc$$

Bei jedem Schritt ist hier die linke Seite der jeweils verwendeten Regel unterstrichen und die Regeln sind numeriert, so daß man für jeden Schritt auch die verwendete Regel angeben kann. Die Folge der Einzelschritte ist also ein Beweis für die Behauptung:

$$a^2b^2c^2 \in L(G).$$

Nach Regel 1: $\sigma \rightarrow abc$ ist auch

$$abc \in L(G).$$

Es gilt allgemeiner: $M = \{a^nb^nc^n | n \in I\!N, n \geq 1\}$ und $M = L(G)$. Diese Behauptung soll jetzt exemplarisch bewiesen werden:

Beweis:

1. $\boxed{M \subseteq L(G)}$ Wenn man Regel $2n$-mal anwendet, erhält man: $\sigma \rightarrow^* a^n\sigma\alpha^n \in S(G)$ und darauf wendet man Regel 1 an: $\sigma \rightarrow^* a^n\sigma\alpha^n \rightarrow a^{n+1}bc\alpha^n \in S(G)$. Das gilt somit für alle $n \in I\!N$.

   Es bleibt zu zeigen: $bc\alpha^n \rightarrow^* b^{n+1}c^{n+1}$. Für $n = 0$ ist das trivial. Angenommen diese Teilbehauptung gilt für $n-1$, also

   $$bc\alpha^{n-1} \overset{*}{\rightarrow} b^nc^n$$

   dann kann man aber ableiten:

   $$bc\alpha^n = bc\alpha^{n-1}\alpha \overset{*}{\rightarrow} b^nc^n\alpha \underset{4}{\overset{*}{\rightarrow}} b^n\alpha c^n \underset{3}{\rightarrow} b^{n+1}c^{n+1}.$$

   Damit ist die Teilbehauptung und somit auch $M \subseteq L(G)$ bewiesen.

2. $\boxed{L(G) \subseteq M}$ Diese Behauptung folgt aus der Beobachtung, daß die Regeln nur in einer bestimmten Folge angewendet werden können:

   Nachdem Regel 1 einmal angewendet wurde kann man nur noch die Regeln 3 und 4 anwenden, da σ nicht mehr in der Satzform vorkommt. Umgekehrt kann man Regeln 3 und 4 nur nach der einen Anwendung von 1 anwenden, da die linken Seiten von 3 und 4 nicht durch Regel 2 erzeugt werden können. Man kann also Regel 2 n-mal anwenden und muß dann genau einmal Regel 1 anwenden:

   $$\sigma \underset{2}{\overset{*}{\rightarrow}} a^n\sigma\alpha^n \underset{1}{\rightarrow} a^{n+1}bc\alpha^n.$$

   Jede Ableitung muß so beginnen. Ist die Satzform $a^{n+1}bc\alpha^n$ erreicht, so kann man nur Regel 4 anwenden:

   $$a^{n+1}bc\alpha^n \underset{3}{\rightarrow} a^{n+1}b\alpha c\alpha^{n-1}.$$

   Auf diese Satzform könnte man wieder Regel 4 anwenden und erhielte:

   $$a^{n+1}b\alpha^2c\alpha^{n-2}.$$

   Diese kann aber keine terminale Satzform mehr erzeugen, da dazu Regel 3 nötig wäre und die ist nicht mehr anwendbar. Es muß also nach Regel 4 zwingend Regel 3 angewendet werden:

   $$a^{n+1}b\alpha c\alpha^{n-1} \underset{3}{\rightarrow} a^{n+1}b^2c^2\alpha^{n-1}.$$

   Jedes α muß durch die Folge der c's nach links verschoben werden (Regel 4) bis es neben einem b (und vor einem c) steht. Jedes α erzeugt ein neues b und ein neues c (Regel 3). Wenn jedes α auf diese (einzige mögliche) Weise ersetzt würde erhält man die terminale Satzform: $a^{n+1}b^{n+1}c^{n+1}$. Jedes Wort aus $L(G)$ hat also diese Form.

Damit ist alles bewiesen. □

`Ende Beispiel`

Das Beispiel hat gezeigt, daß Beweise der Art $M \subseteq L(G)$ oder $L(G) \subseteq M$ unter Umständen einen mühsamen kombinatorischen Aufwand erfordern. Da M meist selbst durch eine Grammatik G' definiert wird ($M = L(G')$) oder aber durch eine Maschine (Automat) haben die meisten Äquivalenzbeweise diese Form.

Anders als in den BNF-Grammatiken für Programmiersprachen tauchen hier auf der linken Seite von Produktionen (Regeln) nicht nur einzelne (nichtterminale) Symbole auf, sondern auch Kombinationen, also ganze Wörter $u \in V^*$. Diese Eigenschaft der Grammatik im Beispiel ist ganz wesentlich. Wenn man fordert, daß auf der linken Seite jeder Produktion jeweils nur genau ein nichtterminales Zeichen stehen darf ($P \subseteq N \times V^*$), dann nennt man die Grammatik *kontextfrei* oder vom Typ 2. Ohne Beweis soll hier erwähnt werden, daß $\{a^n b^n c^n | n \in I\!N, n \geq 1\}$ nicht durch eine kontextfreie Grammatik erzeugt werden kann.

Wir wollen jetzt verschiedene Grammatiktypen und damit eine Hierarchie von Sprachklassen einführen. Die verschiedenen Typen ergeben sich aus bestimmten Einschränkungen bei der Form der Produktionen. Es sei $T_\epsilon = T \cup \{\epsilon\}$.

Definition 1.3.4 *Eine Grammatik $G = (N, T, P, \sigma)$ heißt*

1. allgemein $\iff P \subseteq V^* \times V^*$,
2. schwach normal $\iff P \subseteq V^+ \times V^*$,
3. normal $\iff P \subseteq V^* N V^* \times V^*$,
4. Chomsky $\iff P \subseteq N^+ \times V^*$,
5. separiert $\iff P \subseteq (N^+ \times N^*) \cup (N \times T_\epsilon)$.

Sei E eine Eigenschaft von Grammatiken, dann sei $\mathcal{L}_E$ die Klasse aller Sprachen, die durch eine Grammatik mit der Eigenschaft E erzeugt werden können. Wenn eine Eigenschaft E_1 eine Eigenschaft E_2 impliziert ($E_1 \Rightarrow E_2$), dann gilt natürlich auch: $\mathcal{L}_{E_1} \subseteq \mathcal{L}_{E_2}$. Nach der obigen Definition gilt:

$$\text{separiert} \Rightarrow \text{Chomsky} \Rightarrow \text{normal} \Rightarrow \text{schwach normal} \Rightarrow \text{allgemein}.$$

Man kann aber zeigen, daß alle diese Grammatiken die gleiche Sprachklasse erzeugen und diese Klasse nennt man $\mathcal{L}_0$, die Grammatiken haben den Typ 0. Eine Typ 0-Grammatik kann also stets o.B.d.A. als separiert angenommen werden. Man kann die Form sogar noch weiter einschränken, ohne die erzeugte Sprachklasse einzuschränken. Es sollen jetzt die verschiedenen Grammatiktypen definiert werden.

Definition 1.3.5 *Eine Grammatik $G = (N, T, P, \sigma)$ heißt:*

1. *ϵ-frei* $\iff P \subseteq V^* \times V^+$,
2. *ϵ-treu* $\iff$ *$((u, \epsilon) \in P \Rightarrow u = \sigma$ und für jede Regel $(v, w) \in P$ gilt: σ kommt nicht in w vor).*

Definition 1.3.6 *Eine Chomsky-Grammatik $G = (N, T, P, \sigma)$ heißt:*

1. Typ 0-Grammatik,

2. monoton $\iff$ *G ist ϵ-treu und für alle $(v, w) \in P$ mit $|w| \geq 1$ gilt:*
$$|v| \leq |w|,$$

3. Typ 1-Grammatik *oder* kontextsensitiv (c.s.) $\iff$ *G ist ϵ-treu und für alle $(v, w) \in P$ mit $|w| \geq 1$ gilt: Es gibt $\alpha \in N$, $v', v'' \in V^*$ und $u \in V^*$, so daß:*
$$v = v'\alpha v'' \text{ und } w = v'uv'',$$

4. Typ 2-Grammatik *oder* kontextfrei (c.f.) $\iff$ $P \subseteq N \times V^*$,

5. linear $\iff$ *G ist kontextfrei und auf der rechten Seite jeder Regel kommt höchstens ein Zeichen aus N vor,*

6. linkslinear (rechtslinear) $\iff$ $P \subseteq N \times N_\epsilon T^*$ $(P \subseteq N \times T^* N_\epsilon)$,

7. Typ 3-Grammatik (einseitig linear, rational) $\iff$ *G ist links- oder rechtslinear.*

Die Sprachen, die von Grammatiken vom Typ i $(i = 0, ..., 3)$ erzeugt werden können, werden in der Sprachklasse $\mathcal{L}_i$ $(= \mathcal{L}_{Typ\ i})$ zusammengefaßt. Es gilt dann:
$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0.$$

Man kann beweisen, daß alle diese Inklusionen echt sind. Für unsere Beispielsprache $M = \{a^n b^n c^n | n \in I\!N, n \geq 1\}$ gilt z.B.:
$$M \in \mathcal{L}_1 \text{ aber } M \notin \mathcal{L}_2.$$

Die Grammatik im Beispiel ist monoton und die monotonen Grammatiken erzeugen die gleichen Sprachen, wie die kontextsensitiven Grammatiken $(\mathcal{L}_1 = \mathcal{L}_{monoton})$, daher gilt $M \in \mathcal{L}_1$. Zum Beweis von $M \notin L_2$ benutzt man das sogenannte „pumping lemma" für kontextfreie Sprachen.

Die Hierarchie dieser vier Sprachklassen wird als *Chomsky-Hierarchie* bezeichnet und wird oft als Ordnungs-Diagramm dargestellt:

$\mathcal{L}_0$ Typ 0-Sprachen (rekursiv aufzählbar)
|
$\mathcal{L}_1$ Typ 1-Sprachen (kontextsensitiv)
|
$\mathcal{L}_2$ Typ 2-Sprachen (kontextfrei)
|
$\mathcal{L}_3$ Typ 3-Sprachen (regulär)

$\mathcal{L}_3$ heißt auch die Klasse der *regulären Sprachen*, weil es genau die Klasse der Sprachen ist, die durch reguläre Ausdrücke definiert werden.

Für jede Grammatik vom Typ i $(i = 0, ..., 3)$ gibt es bestimmte Normalformen, bei denen die Form der Produktionen „standardisiert" ist. Man kann zeigen, daß es zu jeder Grammatik vom Typ i eine *Chomsky-Normalform* vom Typ i gibt. Diese kann aus der gegebenen Grammatik konstruktiv hergestellt werden. Es sollen hier kurz diese Normalformen angegeben werden:

Typ 0: $P \subseteq (N \times N^2) \cup (N^2 \times N) \cup (N \times T_\epsilon)$ und $|P \cap V^* \times \{\epsilon\}| \leq 1$,

Typ 1: $P \subseteq (N \times N^2) \cup (N^2 \times N^2) \cup (N \times T_\epsilon)$ und $|P \cap V^* \times \{\epsilon\}| \leq 1$,

Typ 2: $P \subseteq (N \times N^2) \cup (N \times T_\epsilon)$ und G ist ϵ-treu,

Typ 3:

rechtslinear: $P \subseteq (N \times TN) \cup (N \times T_\epsilon)$
linkslinear: $P \subseteq (N \times NT) \cup (N \times T_\epsilon)$

Wenn man bestimmte Spracheigenschaften über Grammatiken beweisen will, dann kann man sich stets auf diese Normalformen beschränken.

Ist $G = (N, T, P, \sigma)$ eine Grammatik und existiert ein Algorithmus, der zu jedem $w \in T^*$ entscheidet ob gilt: $w \in L(G)$ oder $w \notin L(G)$, dann sagt man: Das *Wortproblem* von G ist lösbar (entscheidbar). Das Wortproblem ist für alle Grammatiken der Typen 1,2 und 3 lösbar; aber nicht für jede Grammatik vom Typ 0. Typ 0-Grammatiken können Sprachen erzeugen, die rekursiv aufzählbar sind, aber nicht entscheidbar.

Beispiel:

1. Typ 3-Grammatik für Bezeichner: $N = \{\langle\text{ident}\rangle, \langle\text{rest}\rangle\}$; $\sigma = \langle\text{ident}\rangle$; $T = \{A, ..., Z, 0, ..., 9\}$;

$$\begin{array}{rrcl} P: & \langle\text{ident}\rangle & \rightarrow & A\langle\text{rest}\rangle|B\langle\text{rest}\rangle|...|Z\langle\text{rest}\rangle \\ & \langle\text{rest}\rangle & \rightarrow & A\langle\text{rest}\rangle|B\langle\text{rest}\rangle|...|Z\langle\text{rest}\rangle \\ & & & |0\langle\text{rest}\rangle|1\langle\text{rest}\rangle|...|9\langle\text{rest}\rangle \\ & & & |\epsilon \end{array}$$

Ableitung für $CX3M$:

$$\langle\text{ident}\rangle \rightarrow C\langle\text{rest}\rangle \rightarrow CX\langle\text{rest}\rangle \rightarrow CX3\langle\text{rest}\rangle \rightarrow CX3M\langle\text{rest}\rangle \rightarrow CX3M$$

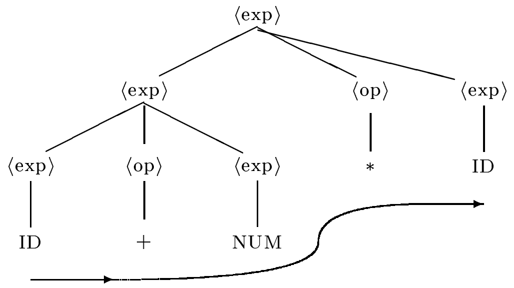
2. Typ 2-Grammatik für Ausdrücke:
$N = \{\langle\text{op}\rangle, \langle\text{exp}\rangle\}$; $\sigma = \langle\text{exp}\rangle$; $T = \{NUM, \text{id}, +, -, *, /,), (\}$;

$$\begin{array}{rrcl} P: & \langle\text{op}\rangle & \rightarrow & +|-|*|/ \\ & \langle\text{exp}\rangle & \rightarrow & \langle\text{exp}\rangle\langle\text{op}\rangle\langle\text{exp}\rangle|(\langle\text{exp}\rangle)|NUM|\text{id} \end{array}$$

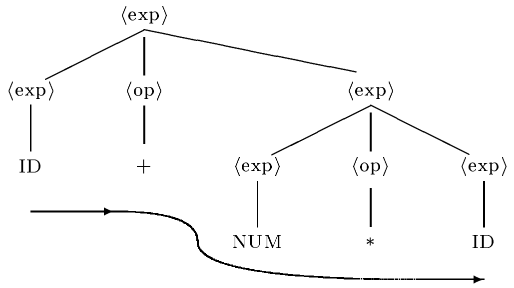
Ableitung für $\text{id} + NUM * \text{id}$:

$$\begin{array}{rcl} \langle\text{exp}\rangle & \rightarrow & \langle\text{exp}\rangle\langle\text{op}\rangle\langle\text{exp}\rangle \rightarrow \langle\text{exp}\rangle * \langle\text{exp}\rangle \rightarrow \langle\text{exp}\rangle * \text{id} \\ & \rightarrow & \langle\text{exp}\rangle\langle\text{op}\rangle\langle\text{exp}\rangle * \text{id} \rightarrow \langle\text{exp}\rangle + \langle\text{exp}\rangle * \text{id} \\ & \rightarrow & \text{id} + \langle\text{exp}\rangle * \text{id} \rightarrow \text{id} + NUM * \text{id} \end{array}$$

Da jedes nichtterminale Zeichen unabhängig vom Kontext in dem es steht ersetzt wird, kann man diese Ableitung auch durch einen Baum darstellen:



Die Folge der Blätter dieses Baumes ist die erzeugte Zeichenreihe (terminale Satzform). Für die gleiche Zeichenreihe ist aber auch der folgende Baum möglich:

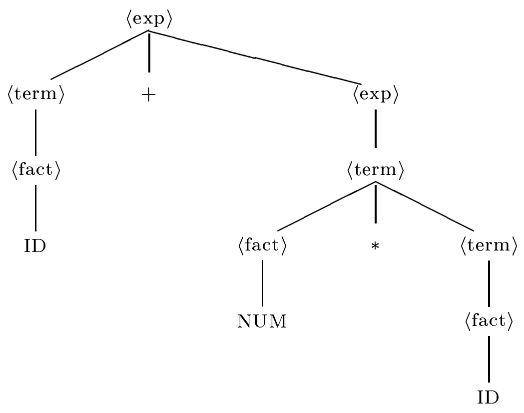


Diese unterschiedlichen Ableitungsbäume geben die unterschiedlichen Möglichkeiten für eine Auswertung (und damit: Semantik) dieses Ausdrucks wider. Durch eine Präzedenz (Rangfolge) der Operatoren kann diese Mehrdeutigkeit aufgehoben werden. Man kann aber die Eindeutigkeit in diesem Fall auch durch eine alternative Grammatik erzwingen.

3. Alternative Typ 2-Grammatik für Ausdrücke: $N = \{\langle\text{exp}\rangle, \langle\text{term}\rangle, \langle\text{fact}\rangle\}$; $\sigma = \langle\text{exp}\rangle$; $T = \{NUM, \text{id}, +, -, *, /,), (\}$;

$$\begin{array}{rrcl} P: & \langle\text{exp}\rangle & \rightarrow & \langle\text{term}\rangle | \langle\text{term}\rangle + \langle\text{exp}\rangle | \langle\text{term}\rangle - \langle\text{exp}\rangle \\ & \langle\text{term}\rangle & \rightarrow & \langle\text{fact}\rangle | \langle\text{fact}\rangle * \langle\text{term}\rangle | \langle\text{fact}\rangle / \langle\text{term}\rangle \\ & \langle\text{fact}\rangle & \rightarrow & (\langle\text{exp}\rangle) | \text{id} | NUM \end{array}$$

Ableitung (Baumdarstellung) für $\text{id} + NUM * \text{id}$:



Bei dieser Grammatik ist die Präzedenzregel „Punktrechnung geht vor Strichrechnung“ fest eingebaut. Der Ableitungsbaum ist eindeutig.

`Ende Beispiel`

Zum Schluß der Betrachtung von Grammatiken noch ein einfaches Beispiel für die Beziehung zwischen einseitig linearen Grammatiken und regulären Ausdrücken.

Beispiel: $T = \{a, b, c, d\}$, $\gamma \in \mathrm{Reg}T$ definiert durch $\gamma = (c + d)(a + b)^*$. Mit $N = \{\sigma, \alpha\}$ sei außerdem die rechtslineare Grammatik $G = (N, T, P, \sigma)$ gegeben durch

$$\begin{array}{rccl} P: & \sigma & \to & c\alpha | d\alpha \\ & \alpha & \to & a\alpha | b\alpha | \epsilon. \end{array}$$

Mit $R = \{c, d\}$ und $S = \{a, b\}$ gilt dann zunächst: $\varphi(\gamma) = RS^*$. Jedes Wort, das von G erzeugt wird, beginnt mit c oder d und dann folgt ein Rest, der von α abgeleitet werden kann: $L(G) = RM$. Von α kann man aber eine beliebige Folge aus a's und b's ableiten und somit gilt $M = S^*$, also: $L(G) = RS^* = \varphi(\gamma)$.

`Ende Beispiel`

Abstrakte Maschinen (*Automaten*) spielen in der Informatik eine große Rolle. Besonders wichtig sind diese mathematischen Modelle in der Theorie der formalen Sprachen. Die Automaten, die wir hier betrachten sind eingeschränkte Turing-Maschinen. Insbesondere sollen hier zunächst nur *Akzeptoren* (erkennende Automaten) betrachtet werden. Für Parser benötigt man aber auch Automaten mit Ausgabe (*Transduktoren*).

Definition 1.3.7 *Eine* abstrakte mathematische Maschine *ist ein 7-Tupel* $M = (I, O, K, \mathrm{in}, \mathrm{out}, R, E)$ *mit*

1. K *Menge von Konfigurationen,*
2. I, O *Mengen (*Ein- *und* Ausgabemenge*),*
3. $\mathrm{in} : I \rightharpoonup K$ Eingabefunktion *(partiell),*
   $\mathrm{out} : K \rightharpoonup O$ Ausgabefunktion *(partiell),*
4. $R \subseteq K \times K$ Überführungsrelation*,*
5. $E \subseteq K$ *Menge der* Endkonfigurationen*.*

M heißt *deterministisch* genau dann wenn die Relation R funktional ist (d.h. $c_1 R c_2 \wedge c_1 R c_3 \Rightarrow c_2 = c_3$) und sonst *nichtdeterministisch* (*n.d.*). Man kann eine solche Maschine durch ein Abbildungsdiagramm veranschaulichen:

$$\begin{tikzcd} I \arrow[r, dash, "\mathrm{in}"] & K \arrow[loop above] \arrow[r, dash, "\mathrm{out}"] \arrow[d, "\mathrm{ch}_E"] & O \\ & \{0,1\} & \end{tikzcd}$$

mit $\mathrm{ch}_E(c) = 1 \iff c \in E$ (*charakteristische Funktion*). R kann dabei als Relation oder als partielle Funktion aufgefaßt werden. Die Maschine M *berechnet* auf folgende Weise eine partielle Funktion (bzw. im nichtdeterministischen Fall eine Relation)

$$\varphi_M \subseteq I \times O.$$

Sei R^* der reflexive und transitive Abschluß von R, $x \in I$, $y \in O$: $x\varphi_M y \iff \exists c \in E : \mathrm{in}(x)R^*c \wedge y = \mathrm{out}(c)$, d.h. von der Konfiguration $\mathrm{in}(x) \in K$ gibt es eine endliche Folge von Konfigurationen:

$$\mathrm{in}(x) = c_0 \underset{R}{\rightarrow} c_1 \underset{R}{\rightarrow} \ldots \underset{R}{\rightarrow} c_m = c \in E$$

so daß c eine Endkonfiguration ist. Auf diese wendet man die Funktion out an und erhält die „Ausgabe“ y. Da man von $\mathrm{in}(x) = c_0$ i.a. mehr als eine Endkonfiguration erreichen kann, betrachtet man die „erste“ erreichbare Endkonfiguration: Für jedes $c \in K$ definiert man:

$$d(c,E) = \begin{cases} \min\{t \in I\!N \mid \exists c' \in E : cR^tc'\} & \text{falls diese Menge} \neq \emptyset \\ \text{undefiniert} & \text{sonst} \end{cases}$$

und damit (für deterministische Maschinen):

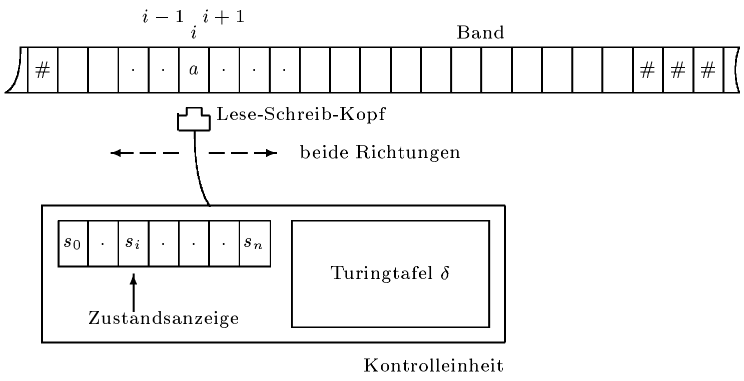
$$\varphi_M(x) = \begin{cases} y & \text{falls } d(\mathrm{in}(x), E) = t \text{ und } \mathrm{in}(x)R^tc \text{ und } y = \mathrm{out}(c) \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$\varphi_M : I \to O$ heißt die von M *berechnete Funktion*. Ist $O = \{0, 1\}$ und out die charakteristische Funktion von E, dann heißt M Akzeptor für $\varphi_M : I \to \{0, 1\}$. Die Wirkungsweise der im folgenden beschriebenen Maschinen wird stets auf dieses Modell zurückgeführt. Maschinen heißen hier auch *Automaten*, und alle Automaten, die wir betrachten, sind eingeschränkte (eventuell Mehrband- und Mehrkopf-) Turingmaschinen.

Die allgemeine Turingmaschine (TM) ist auf der folgenden Seite skizziert. Sie besteht aus einer endlichen Kontrolleinheit und einem unendlichen „Rechenband“. Das Band ist in Felder eingeteilt, und jedes Feld kann genau ein Zeichen aus dem (endlichen) Bandalphabet A aufnehmen (enthalten).

Ein Lese-Schreibkopf befindet sich auf einem Feld des Bandes und kann aus dieser Position jeweils um ein Feld nach links (L) oder rechts (R) bewegt werden. Die Kontrolleinheit enthält eine Zustandsanzeige, d.h. die Einheit befindet sich stets in genau einem der Zustände $s \in S = \{s_0, ..., s_e\}$. Eine (endliche) *Tafel* (Tabelle) $\delta : S \times A \to S \times Op$ ($Op = A \cup \{L, R\}$ definiert für jede Kombination (s, a) von innerem Zustand s und gelesenem Zeichen a einen neuen (nächsten) Zustand und eine Operation $\xi \in Op$. Ist $\delta(s, a) = (s', \xi)$ und $\xi \in A$, dann soll ξ auf das Band geschrieben werden (an der aktuellen Position des Kopfes). Ist $\xi \in \{L, R\}$, dann soll der Kopf nach links (L) bzw. rechts (R) bewegt werden. Der Rechenprozeß wird also durch die Tafel δ bestimmt. Er hängt ab vom Anfangszustand ($s_0 \in S$) und dem Inhalt des Bandes. Wenn der innere Zustand s_e (Endzustand) erreicht ist, soll die Maschine anhalten. Sie hat dann die anfängliche Bandinschrift (gemäß δ) modifiziert, also etwas „berechnet“.

Turingmaschine



Bandalphabet: $A = \{a_1, ..., a_p, \#\}$ (# ist Blank-Symbol)
Bandinhalt: $\beta : \mathbb{Z} \to A$ ($\mathbb{Z}$ = Menge der ganzen Zahlen)
Zustandsmenge: $S = \{s_0, ..., s_n\}$; s_0: Angangs-, s_n Endzustand
Turing-Tafel: $\delta : S \times A \to S \times Op$
Operationen:

$Op = A \cup \{L, R\}$.

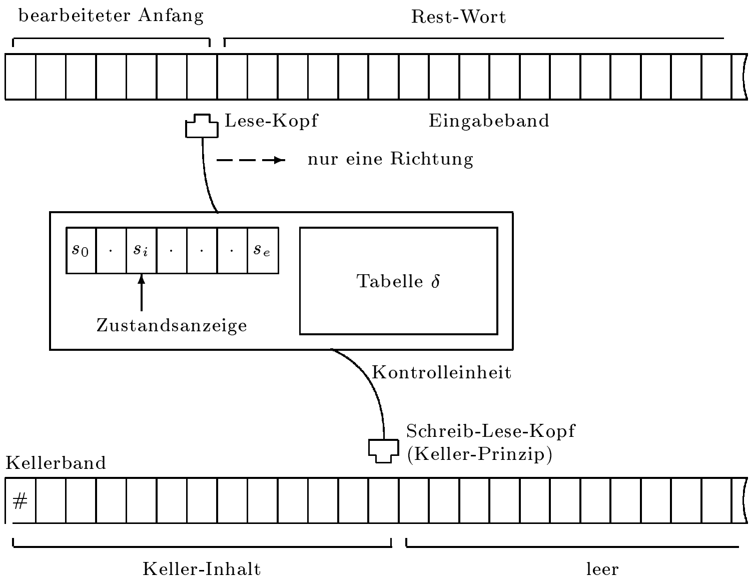
$\xi \in A$: Schreibe ξ auf das Feld.

$\xi = L$: Gehe ein Feld nach links.

$\xi = R$: Gehe ein Feld nach rechts.

$\delta(s, a) = (s', \xi)$ bedeutet: **Wenn** Zustand = s **und** gelesenes Zeichen = a, **dann** setze Zustand auf s' und führe Operation ξ aus.

Kellerautomat



Eingabealphabet: Σ, $\Sigma_\epsilon = \Sigma \cup \epsilon$
Kelleralphabet: Γ
Kellerbodensymbol: $\# \in \Gamma$, $\Gamma_0 = \Gamma - \{\#\}$
Zustandsmenge: S
Endzustände: $F \subseteq S$

Anfangszustand: $s_0 \in S$
Tabelle: $\delta \subseteq (S \times \Sigma_\epsilon \times \Gamma) \times (S \times \Gamma^*)$
mit: $(s, t, \#)\delta \subseteq S \times \#\Gamma_0^*$ $(t \in \Sigma)$
und: $(s, t, x)\delta \subseteq S \times \Gamma_0^*$ $(t \in \Sigma,\ x \in \Gamma_0)$
Deterministisch: Zusätzlich gilt:

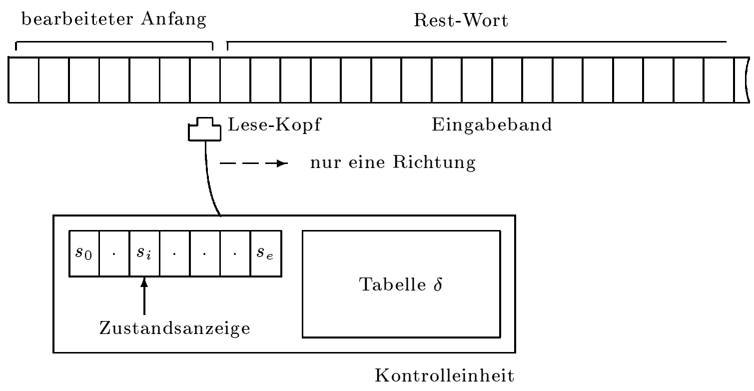
1. δ ist funktionale Relation.
2. $(s, \epsilon, x)\delta \neq \emptyset \Rightarrow \forall t \in \Sigma : (s, t, x)\delta = \emptyset$.

Konfigurationsraum: $K = \#\Gamma_0^* S \Sigma^*$

Endkonfiguration: $E = \#\Gamma_0^* F$
Eingabe: $\mathrm{in}(u) = \#s_0 u$
Dynamik: $R \subseteq K \times K$

$$(\#vxstu)R(\#vws'u) \iff (s, t, x)\delta(s'w)$$
$$(\#vxsu)R(\#vws'u) \iff (s, \epsilon, x)\delta(s'w)$$

Endlicher Automat



Eingabealphabet: Σ
Zustandsmenge: S
Endzustände: $F \subseteq S$
Anfangszustand: $s_0 \in S$
Tabelle: $\delta \subseteq (S \times \Sigma) \times S$
Deterministisch: Wenn zusätzlich gilt: δ ist funktionale Relation
Konfigurationsraum: $K = S\Sigma^*$
Endkonfiguaration: $E = F$
Eingabe: $\mathrm{in}(u) = s_0 u$
Dynamik: $R = \delta\Sigma^*$ also: $(stu)R(s'u) \iff (s, t)\delta s'$

Ist $L \subseteq A^*$ (nicht verwechseln mit $L \in Op$!) eine Typ-0-Sprache, dann gibt es eine Turingmaschine, die genau dann anhält, wenn man sie mit einem Anfangswort $u \in L$ im Zustand s_0 startet. Schreibt man aber ein Wort $u \in (A^* - L) = \overline{L}$ auf das Band, so hält die Maschine nicht an. Eine solche Maschine kann nicht entscheiden ob $u \in L$ oder $u \notin L$ gilt, sondern nur $u \in L$ bestätigen.

In diesem Sinne sagt man, daß Turingmaschinen gerade die Typ-0-Sprachen „erkennen", denn jede Sprache, die auf diese Weise durch eine Turingmaschine erkannt wird, ist auch vom Typ 0.

Wenn man als Turingtafel allgemeiner eine Relation $\delta \subseteq (S \times A) \times (S \times Op)$ (δ endlich) zuläßt, dann kann die Maschine u.U. einen Folgeschritt „auswählen". Eine solche Turingmaschine heißt *nichtdeterministisch*. Man kann beweisen, daß deterministische und nichtdeterministische Turingmaschinen die gleiche Sprachklasse akzeptieren, nämlich $\mathcal{L}_0$. Nicht bei allen Automatenarten erkennen die deterministischen Versionen genau die gleichen Sprachen, wie die nichtdeterministischen Versionen. Eine eingeschränkte Turingmaschine erhält man, wenn man den Speicherplatzbedarf für die (Zwischen-) Berechnungen der Maschine auf dem Band begrenzt.

Ein *linear beschränkter Automat* (*LBA*) ist eine Turingmaschine, die so modifiziert ist, daß die Kopfbewegung auf den Teil des Bandes eingeschränkt wird, den das Eingabewort belegt hatte. Ein langes Eingabewort liefert also viel Speicherplatz und ein kurzes wenig. Die nichtdeterministischen Versionen (NLBA) dieser Automaten erkennen genau die kontextsensitiven Sprachen. Ein entsprechender Äquivalenzbeweis stellt eine Beziehung her zwischen den Berechnungsfolgen eines NLBA und den Ableitungen einer monotonen Grammatik.

Es ist noch ein offenes Problem, ob die deterministischen LBA's (DLBA) die gleiche Sprachklasse erkennen oder nur eine echte Teilklasse. Wir wenden uns jetzt den für die Vorlesung wichtigsten Automatentypen zu, nämlich den *Kellerautomaten* (*Push-Down-Automaton*) und den *endlichen Automaten* (*Finite Automaton*). Während der endliche Automat einfach ein Kellerautomat ohne Kellerspeicher ist, ist die Beschreibung des Kellerautomaten etwas aufwendiger.

Ein Kellerautomat (zunächst nur erkennender Automat: *Push-Down-Acceptor*, *PDA*) ist eine Turingmaschine mit zwei Bändern und zwei Köpfen (siehe Seite 34). Die Verwendung der Bänder ist aber stark eingeschränkt. Zum *Eingabeband* gehört ein (Nur-) Lesekopf, der auch nur Feld für Feld von links nach rechts bewegt werden darf. Diese Einrichtung liest also das Eingabewort Zeichen für Zeichen. Felder links vom Kopf sind nicht mehr erreichbar.

Das Band für Zwischenrechnungen kann gelesen und beschrieben werden, aber nur nach dem *Keller-Prinzip*. Der Kopf kann das „oberste" Keller-Zeichen lesen und dieses durch ein Wort (auch ϵ) nach „oben" (im Bild nach rechts) ersetzen. Danach steht der Kopf auf dem letzten Zeichen des neu geschriebenen Wortes. Nur wenn dieses Wort ϵ ist, bewegt sich der Kopf eine Position nach „unten" (im Bild links), da er das gelesene Zeichen gelöscht hat.

Definition 1.3.8 *Ein* nichtdeterministischer Kellerakzeptor *(*NPDA*) ist ein 7-Tupel* $M = (\Sigma, \Gamma, S, \delta, s_0, \#, F)$ *mit:*

1. Σ Eingabealphabet*,*
2. Γ Kelleralphabet *und* $\# \in \Gamma$*;* $\Gamma_0 = \Gamma - \{\#\}$*,*
3. S *endliche* Zustandsmenge*,* $s_0 \in S$*,*
4. $F \subseteq S$ *Menge der* Endzustände*,*
5. $\delta \subseteq (S \times \Sigma_\epsilon \times \Gamma) \times (S \times \Gamma^*)$ *endliche* Relation *mit:*
   (a) $(s, t, \#)\delta \subseteq S \times \#\Gamma_0^*$
   (b) $(s, t, c)\delta \subseteq S \times \Gamma_0^*$

$\#$ ist das spezielle *Kellerboden-Symbol*, und die Bedingungen unter 5. besagen, daß dieses Symbol nicht gelöscht werden darf (a) und daß kein weiteres davon auf das Kellerband geschrieben werden darf (b).

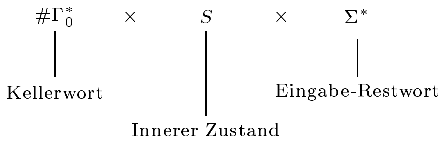
Definition 1.3.9 *Ein* Kellerakzeptor $M = (\Sigma, \Gamma, S, \delta, s_0, \#, F)$ *heißt* deterministisch *(*DPDA*), wenn zusätzlich gilt: δ ist funktional (partielle Funktion) und:* $(s, \epsilon, c)\delta \neq \emptyset \Rightarrow \forall t \in \Sigma : (s, t, c)\delta = \emptyset$.

Bemerkung zur Notation: Ist $R \subseteq A \times B$ eine beliebige 2-stellige Relation, dann sei für Teilmengen $A' \subseteq A$ und $B' \subseteq B$:

$$\begin{aligned} A'R &= \{b \in B | \exists a \in A' : aRb\} \\ RB' &= \{a \in A | \exists b \in B' : aRb\} \end{aligned}$$

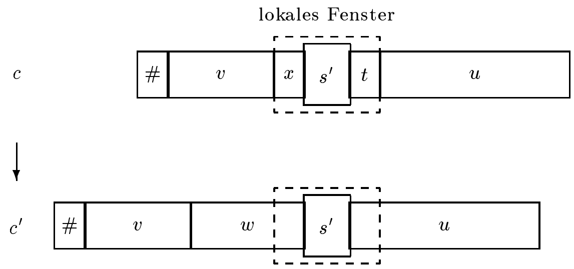
Die Funktionsweise eines PDA kann jetzt als abstrakte Maschine beschrieben werden.

Eine *Konfiguration* ist eine „globale Zustandsbeschreibung". Dazu gehören der innere Zustand der Maschine, die Bandinhalte und die Positionen der beiden Köpfe. Da der Lesekopf nur nach links bewegt werden kann, reicht für das Eingabeband die Angabe des *Restwortes* $u \in \Sigma^*$, das bei der Kopfposition beginnt. Der *Kellerinhalt* ist stets ein Wort $\#v \in \#\Gamma_0^*$ und der Lese-Schreibkopf des Kellerbandes steht auf dem letzten Zeichen dieses Wortes. Außerdem benötigt man den *inneren Zustand* $s \in S$: Der Konfigurationsraum ist also ein Produkt



Wenn man o.B.d.A. annimmt: $\Gamma \cap S = \emptyset = \Sigma \cap S$, dann kann man die Konfiguration (x, s, u) auch als Wort xsu über dem Alphabet $\Gamma \cup S \cup \Sigma$ betrachten. Daher definiert man den Konfigurationsraum wie folgt: $K = \#\Gamma_0^* S \Gamma^*$ (Bem.: Das ist eine reguläre Sprache). Eine *Endkonfiguration* liegt vor, wenn das Eingabe-Restwort leer ist und $s \in F$: $E = \#\Gamma_0^* F$, und mit $I = \Sigma^*$ und $O = \underline{2} = \{0, 1\}$ ist dann: $\mathrm{in}(u) = \#s_0 u$, also leerer Keller und u steht auf dem Eingabeband und Anfangszustand ist s_0. Die Ausgabefunktion out ist einfach die charakteristische Funktion von $E \subseteq K$, wie bei allen Akzeptoren.

Die *Dynamik* $R \subseteq K \times K$ wird als lokale Ersetzungsregel durch δ definiert:



Konfigurationsübergang als lokale Ersetzung

$$(\#vxstu)R(\#vws'u) \quad \Longleftrightarrow \quad (s, t, x)\delta(s', w)$$

$$(\#vxsu)R(\#vws'u) \quad \Longleftrightarrow \quad (s,\epsilon,x)\delta(s',w)$$

Die zweite Zeile stellt den Fall dar, daß auf dem Keller eine Ersetzung stattfindet, ohne daß der Eingabekopf weiterbewegt wird (ϵ-Übergang).

Beispiel: DPDA $M = (\Sigma, \Gamma, S, \delta, s_0, \#, F)$ mit $\Sigma = \{a, b\}$; $\Gamma = \{\#, a\}$; $S = \{s_0, s_1, s_2, s_3\}$; $F = \{s_3\}$. δ wird als Ersetzungssystem beschrieben: Für $(s,t,x)\delta(s',w)$ schreiben wir $xst \rightarrow ws'$ (siehe Lokales Fenster).

$$\begin{array}{rrcl}
\delta: & \#s_0a & \rightarrow & \#as_1 \text{ (shift) (1)} \\
 & as_1a & \rightarrow & aas_1 \text{ (shift) (2)} \\
 & as_1b & \rightarrow & s_2 \text{ (reduce) (3)} \\
 & as_2b & \rightarrow & s_2 \text{ (reduce) (4)} \\
 & \#s_2 & \rightarrow & \#s_3 \text{ (stop, (accept)) (5)}
\end{array}$$

Behauptung: Sei $L = \{a^nb^n | n \geq 1\}$ und $L(M)$ die von M akzeptierte Sprache, dann gilt: $L(M) = L$.

Beweis:

1. Zeige: $\#s_0a^nb^nR^*\#s_3$ $(n \geq 1)$

$$\#s_0a^nb^n \underset{(shift)}{\overset{*}{\rightarrow}} \#a^ns_1b^n \underset{(reduce)}{\overset{*}{\rightarrow}} \#s_2 \underset{(accept)}{\rightarrow} \#s_3.$$

2. Zeige: $L(M) \subseteq L$. Sei also $u \in L(M)$. Wegen (1) muß u mit a beginnen: $u = au'$ und wegen (3) und (4) enthält u' mindestens ein b, also gilt $u = a^ibv$ $(i \geq 1,\ v \in \Sigma^*)$. Für $i = 1$ ist $u = abv$ und man erhält:

$$\#s_0abv \rightarrow \#as_1bv \rightarrow \#s_2v \rightarrow \#s_3v$$

das ist aber nur dann Endkonfiguration, wenn gilt: $v = \epsilon$ also $u = ab \in L$. Sei jetzt $i \geq 2$:

$$\#s_0a^ibv \underset{(shift)}{\overset{*}{\rightarrow}} \#a^is_1bv \underset{(reduce)}{\rightarrow} \#a^{i-1}s_2v.$$

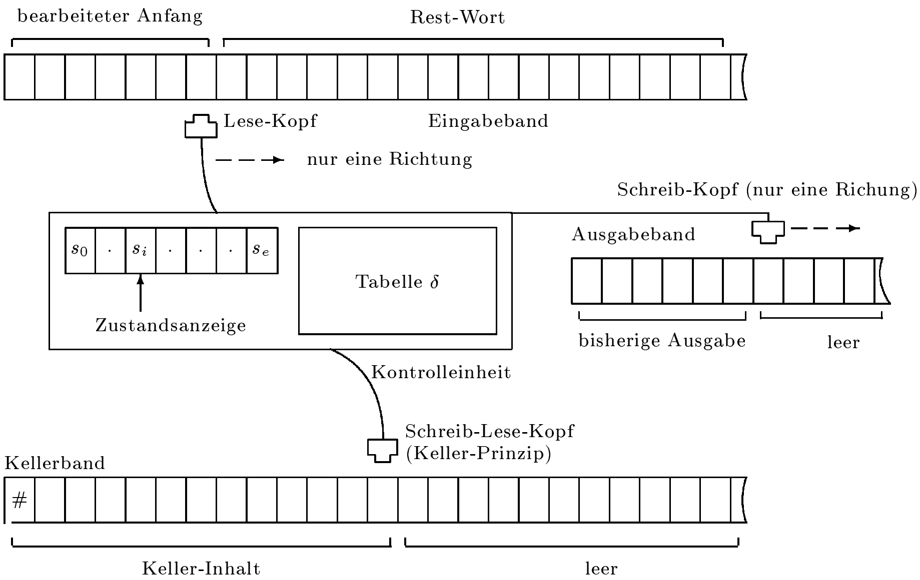
Jetzt kann man nur noch Regel (4) anwenden und dazu muß gelten: $v = b^{i-1}$, um $\#s_2$ und somit eine Endkonfiguration zu erreichen. Es ist also $u = a^ibv = a^ib^i \in L$ $(i \geq 2)$. Also gilt: $L(M) \subseteq L$.

$\square$

`Ende Beispiel`

Ein Kellerautomat mit Ausgabe besitzt ein zusätzliches Ausgabeband mit einem Schreibkopf, der ebenfalls nur in einer Richtung bewegt werden darf. Bei jedem Übergang (δ-Schritt) wird ein Wort $z \in \Delta^*$ auf das Ausgabeband geschrieben. Δ ist ein geeignetes Ausgabealphabet.

Kellerautomat mit Ausgabeband



Ein von YACC erzeugter Parser hat i.W. diese Struktur. Die Tabelle δ wird dabei automatisch generiert.

Ein *endlicher Akzeptor* (*FA*) ist einfach zu betrachten, wie ein Keller-Akzeptor ohne Keller.

Definition 1.3.10 *Ein* endlicher (n.d.) Akzeptor *(*NFA*) ist ein 5-Tupel* $M = (\Sigma, S, \delta, s_0, F)$ *mit:*

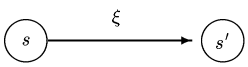
1. Σ Eingabealphabet*,*
2. S *endliche* Zustandsmenge*;* $s_0 \in S$*,*
3. $F \subseteq S$ *Menge der* Endzustände*,*
4. $\delta \subseteq (S \times \Sigma_\epsilon) \times S$ Übergangs-Relation*.*

M heißt deterministisch *(*DFA*), wenn δ eine partielle Funktion ist.*

Die zugehörige abstrakte Maschine ist leicht zu beschreiben: $K = S\Sigma^*$ (Zustand, Restwort), $E = F \subseteq K$ und $\text{in}(u) = s_0 u$ $(u \in \Sigma^*)$. R wird wieder durch δ definiert:

1. $(stu)R(s'u) \iff (s,t)\delta s'$,
2. $(su)R(s'u) \iff (s,\epsilon)\delta s'$.

Ein Übergang nach 2. heißt wieder *ϵ-Übergang*. Man kann einen endlichen Automaten (Akzeptor) sehr gut durch einen *Zustands-Graphen* darstellen. Die Zustände sind die Knoten und jeder Übergang $(s,\xi)\delta s'$ wird durch eine markierte Kante dargestellt ($\xi \in \Sigma_\epsilon$):

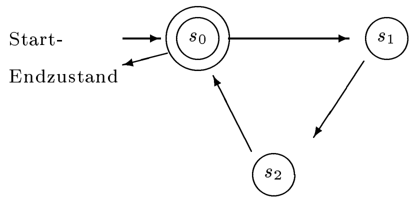


Anfangs- und Endzustände werden geeignet markiert. Dafür ein Beispiel:

Beispiel:[DFA] Sei $L = (abb)^*$ (also eine reguläre Menge), $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, $F = \{s_0\}$,

$$\begin{array}{rlcll} \delta: & s_0 a & \rightarrow & s_1 & (1) \\ & s_1 b & \rightarrow & s_2 & (2) \\ & s_2 b & \rightarrow & s_0 & (3) \end{array}$$

als Zustands-Graph:



Sei $L(M)$ die von M erkannte Sprache.
Behauptung: $L(M) = L$.

Beweis:

1. Zeige $L \subseteq L(M)$, also $\forall n \in I\!N : (abb)^n \in L(M)$.
   (a) $n = 0$: $s_0\epsilon = s_0 \in F \Rightarrow (abb)^0 = \epsilon \in L(M)$.
   (b) Angenommen $s_0(abb)^n \rightarrow^* s_0$ gilt bereits. Zeige: $s_0(abb)^{n+1} \rightarrow^* s_0$.
   $$s_0(abb)^{n+1} = s_0abb(abb)^n \rightarrow s_1bb(abb)^n \rightarrow s_2b(abb)^n \rightarrow s_0(abb)^n \stackrel{*}{\rightarrow} s_0.$$
2. Zeige $L(M) \subseteq L$: Sei $u \in L(M)$, dann gilt entweder $u = \epsilon$ (wegen $s_0 \in F$) und damit $u \in L$ oder u beginnt mit abb, hat also die Form $(abb)v$ mit $v \in L(M)$. Zeige: $\forall i \in I\!N : L(M) \cap \Sigma^i \subseteq L$.
   (a) Für $i = 0$ schon gezeigt.
   (b) Sei also $u = (abb)v$ und gelte $\forall i < |u| : L(M) \cap \Sigma^i \subseteq L$ (Induktionsvoraussetzung) dann ist $|v| < |u|$ und $v \in L(M)$, somit $v \in L$. v hat also die Form $v = (abb)^k$ für $3k < |u|$ und somit:
   $$u = (abb)(abb)^k = (abb)^{k+1} \in L.$$

□

`Ende Beispiel`

Sei τ eine Bezeichnung für einen Maschinentyp, dann sei $\mathcal{L}_\tau$ die Klasse der Sprachen, die von Maschinen (Akzeptoren) dieses Typs erkannt werden. Die folgenden Aussagen sollen hier ohne Beweis zitiert werden.

Satz 1.3.11

1. *$\mathcal{L}_0 = \mathcal{L}_{NTA} = \mathcal{L}_{DTA}$ (Turing-Akzeptoren),*

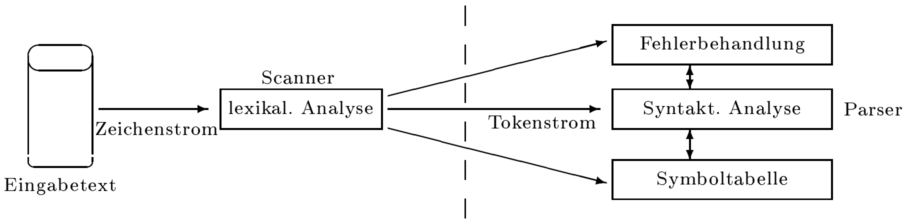
2. $\mathcal{L}_1 = \mathcal{L}_{monoton} = \mathcal{L}_{NLBA}$,
3. $\mathcal{L}_{DLBA} \subseteq \mathcal{L}_{NLBA}$ *(Echtheit unbekannt!)*,
4. $\mathcal{L}_2 = \mathcal{L}_{NPDA}$,
5. $\mathcal{L}_{DPDA} \subsetneq \mathcal{L}_{NPDA}$,
6. $\mathcal{L}_3 = \mathcal{L}_{\mathrm{Reg}} = \mathcal{L}_{R-lin} = \mathcal{L}_{L-lin} = \mathcal{L}_{NFA} = \mathcal{L}_{DFA}$,
7. $\mathcal{L}_3 \subsetneq \mathcal{L}_{DPDA}$.

Kapitel 2

Lexikalische Analyse

2.1 Aufgaben der Lexikalischen Analyse

Die lexikalische Analyse ist der syntaktischen Analyse vorgeschaltet. Aber eigentlich sind beide nur verschiedene Stufen der Syntaxanalyse. So ist auch die Grammatik für Programmiersprachen meist als eine kontextfreie Grammatik gegeben, deren terminale Zeichen die ASCII-Zeichen sind. Die Zerlegung in zwei Stufen entspricht zum einen der Identifizierung von regulären Mustern in der Sprache, zum anderen aber auch dem Prinzip der Zerlegung einer Aufgabe in logisch abgegrenzte Teilaufgaben.



- der Parser fordert vom Scanner jeweils den nächsten Token, d.h. die Bezeichnung für ein Muster, dem das gerade identifizierte Lexem entspricht.

- Wenn ein Token einen zusätzlichen Attributwert besitzt, der aus dem Lexem abgeleitet werden kann, dann wird dieser in einer Symboltabelle abgelegt.

- Wenn lexikalische Fehler entdeckt werden, dann soll eine Fehlermeldung erfolgen. Auch syntaktische Fehler, die der Parser entdeckt, können dazu führen, daß die Fehlerbehandlung den Scanner zum „weiterlesen“ veranlaßt, um dann die Analyse später im Text wieder aufzunehmen. Der Scanner kennt auch stets die Position im Eingabetext und kann so den Fehlerort (oder besser den Ort, an dem der Fehler zuerst bemerkt wird) angeben.

- Wenn eine Programmiersprache bestimmte Schlüsselwörter verwendet, wie z.B. PASCAL die Wörter <u>begin</u> , <u>end</u> , <u>if</u> , <u>then</u> , <u>else</u> u.s.w. dann kann es Aufgabe des Scanners sein, zwischen einem gewöhnlichen Bezeichner (identifier) und einem Schlüsselwort (keyword) zu unterscheiden. Das kann leicht durch einen Suchvorgang in einer Tabelle erfolgen (diese Tabelle könnte auch die Symboltabelle sein).

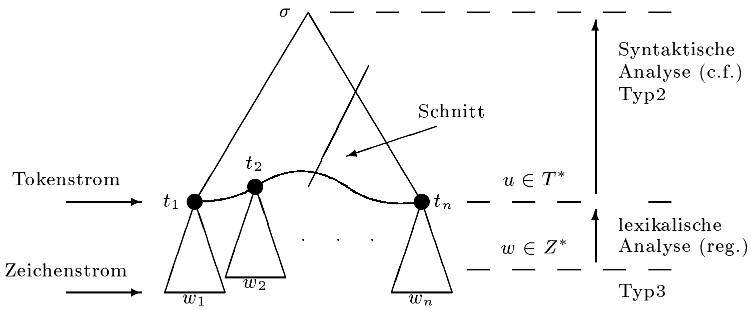
Die Hauptaufgabe der lexikalischen Analyse wurde bereits im ersten Kapitel formuliert:

Gebeben: Menge (Alphabet) T von Token und zu jedem $t \in T$ ein Muster $L_t \subseteq Z^*$. L_t ist dabei eine reguläre Sprache über dem Zeichenalphabet Z.

Gesucht: Zu jedem Eingabewort $w \in Z^*$ eine Zerlegung $w = w_1 ... w_n$ mit $w_i \in Z^*$ und ein Wort $u = u_1 ... u_n$ mit $u_i \in T$ so daß gilt: $\forall i \in \{1, ..., n\} : w_i \in \mathcal{L}_{u_I}$.

Das Eingabewort $w \in Z^*$ wird also in ein Wort $u \in T^*$ und damit ein Zeichenstrom in einen Tokenstrom transformiert. Die syntaktische Analyse kann sich daher auf die (reduzierten) Wörter $u \in T^*$ beschränken.

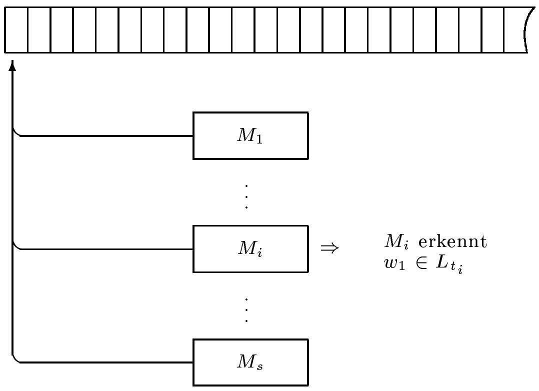
Diese Transformation kann man leicht an den Ableitungsbäumen veranschaulichen:



Der gesamte Ableitungsbaum bis auf die terminalen Zeichen aus Z wird zerschnitten. So daß man einen Schnitt erhält, dessen Nichtterminalzeichen gerade die neuen Terminalzeichen $t \in T$ sind. Die Syntaxanalyse muß dann die obere Hälfte des Schnittes (den Baum oberhalb des Schnittes) analysieren (also aus dem Schnitt-Wort $u \in T^*$ generieren). Die lexikalische Analyse muß nur die Abschnitte w_i des Eingabewortes finden, die sich jeweils auf t_i reduzieren lassen, also aus t_i ableiten lassen.

Da die Sprachen $(L_t | t \in T)$ alle regulär sind, kann man diese Sprachen (Muster) charakterisieren durch einseitig lineare Grammatiken, durch endliche Automaten oder reguläre Ausdrücke.

Wenn man für jedes der endlich vielen Muster einen Automaten M_t hat, dann kann man das Eingabewort „parallel" in jeden Automaten eingeben. Sobald einer der Automaten in einen Endzustand übergeht, ist ein Abschnitt erkannt und gehört zum Muster des zugehörigen Automaten. Mann kann dann alle Automaten wieder in den Anfangszustand versetzen und so das nächste Muster suchen u.s.w.:



Dies ist im Prinzip die Arbeitsweise eines Scanners, der mit LEX erzeugt wurde. Die Details dazu werden weiter unten behandelt.

Zum Schluß dieses Abschnittes soll die Transformation, die die lexikalische Analyse vornimmt durch den Begriff der Substitution beschrieben werden. Seien T und Z zwei Alphabete und $\varphi : T \rightarrow \mathcal{P}(Z^*)$ eine Abbildung. φ liefert also zu jedem $t \in T$ eine Sprache $\varphi(t) \subseteq Z^*$. Diese Abbildung läßt sich nun auf T^* und sogar auf $\mathcal{P}(T^*)$ wie folgt fortsetzen:

1. Sei $u = u_1...u_n \in T^*$ $(u_i \in T, i \in \{1, ..., n\})$ dann definiert man $\overline{\varphi}(u) = \varphi(u_1)\varphi(u_2)...\varphi(u_n)$. Auf der rechten Seite steht die Konkatenation von Sprachen und somit ist auch $\overline{\varphi}(u) \subseteq Z^*$.

2. Sei $L \subseteq T^*$, dann definiert man
$$\hat{\varphi}(L) = \bigcup_{u \in L} \overline{\varphi}(u)$$
und damit ist auch $\hat{\varphi}(L) \subseteq Z^*$. Alle diese Abbildungen werden dann einfach mit φ bezeichnet und $\varphi : T \rightarrow \mathcal{P}(Z^*)$ heißt eine *(Sprach-)Substitution*. Sie bewirkt eine Sprachtransformation
$$\varphi : \mathcal{P}(T^*) \rightarrow \mathcal{P}(Z^*).$$

Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik. Für jedes $\alpha \in N$ definiert man:
$$L_\alpha(G) = \{w \in T^* | \alpha \overset{*}{\underset{p}{\rightarrow}} w\}$$
also die terminalen Wörter, die aus α abgeleitet werden können.

Man nennt $L_\alpha(G)$ (kurz L_α) auch eine *syntaktische Kategorie*. L_α beschreibt bestimmte Teilwörter der Sprache $L(G)$; aber L_α ist i.a. keine Teilsprache (Teilmenge) von $L(G)$.

Beispiel: Man betrachte die Grammatik G auf Seite 14. Sei $\alpha =$<term> und $\sigma =$<program>, dann gilt:

id$*$ NUM $\in L_\alpha$,

id$*$ NUM $\notin L_\sigma = L(G)$,

id$*$ NUM ist ein Term aber kein (vollständiges) Programm.

`Ende Beispiel`

Jedes nichtterminale Zeichen definiert also selbst eine Sprache und die Wörter dieser Sprache kommen in den Wörtern aus $L(G)$ als Teilwörter vor.

Wenn man für jedes $t \in T$ eine kontextfreie Grammatik $G_t = (N_t, Z, P_t, \sigma_t)$ hat und jede dieser Grammatiken definiert eine Sprache $\varphi(t) = L(G_t)$, dann ist $\varphi(L(G)) \subseteq Z^*$ eine Sprache mit den terminalen Zeichen Z. Diese kann man auch mit einer neuen kontextfreien Grammatik $\tilde{G} = (\tilde{N}, Z, \tilde{P}, \sigma)$ erzeugen, wobei gilt: $\tilde{N} = N + T$, $\tilde{P} = P \cup \bigcup_{t \in T} P_t$. Jedes terminale Zeichen $t \in T$ wird zu einem neuen nichtterminalen Zeichen in $\tilde{N}$ und man nimmt zu den Produktionen P noch alle Produktionen der Grammatiken G_t hinzu. Somit ist die Sprache $\varphi(L(G)) = L(\tilde{G})$ und damit selbst kontextfrei.

Eine Ableitung für $\tilde{G}$:
$$\sigma \overset{*}{\underset{\tilde{P}}{\rightarrow}} w \in Z^*$$

kann also stets über ein Zwischenwort $u \in T^*$ „faktorisiert“ werden:

$$\sigma \xrightarrow[P]{*} u \xrightarrow[\bigcup_{t\in T} P_t]{*} w \text{ mit } u \in T^* \text{ und } w \in Z^*.$$

Es ist dies genau die Zerlegung in die syntaktische Analyse ($\sigma \rightarrow_P^* u$) und die lexikalische ($u \rightarrow^*_{\bigcup_{t\in T} P_t} w$).

Man wählt diese Zerlegung aber außerdem so, daß alle Sprachen $\varphi(t) = L(G_t)$ regulär sind.

Nach diesen theoretischen Betrachtungen wollen wir jetzt die praktische Programmierung solcher Scanner genauer betrachten. Dabei werden die endlichen Automaten als Akzeptoren für reguläre Mengen (Sprachen) eine wesentliche Rolle spielen.

Die Spezifikation dieser regulären Mengen (Muster) wird dabei meistens (aber nicht zwingend) durch reguläre Ausdrücke vorgenommen.

2.2 Methoden zur Programmierung eines Scanners

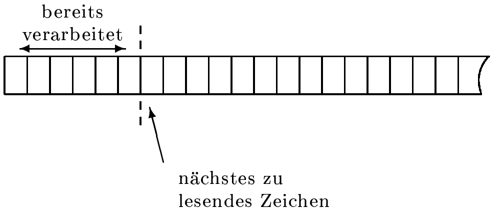
Es soll hier kurz auf drei Methoden eingegangen werden:

- Ad hoc Programmierung: Man versucht, die Syntaxdiagramme (also die Regeln einer Grammatik) direkt in ein Programm zu übertragen. Die lexikalischen Konstrukte haben oft eine sehr einfache Form, so daß man entsprechende Funktionen leicht programmieren kann.
- Übergangsdiagramme: Die syntaktischen Definitionen werden in gerichtete Graphen übersetzt. Die Knoten stellen dann die Information über den bereits gelesenen Abschnitt einer Folge dar. Wenn ein Zeichen gelesen wird, wird auf einer gerichteten Kante ein neuer Zustand erreicht. Ausgezeichnete Start- und Endknoten markieren den Beginn und das Ende eines Lexems.
- Scanner-Generator: Ein Scanner-Generator (wie z.B. LEX) erhält als Eingabe die Beschreibung der regulären Muster in der Form regulärer Ausdrücke. Man kann jedem Muster dann eine Aktion (C-Programmtext) zuordnen, die ausgeführt werden soll, wenn das Muster erkannt wurde. Der Generator erzeugt dann den Programmtext für den Scanner.

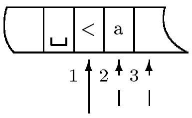
2.2.1 Ad hoc-Programmierung

Es wird nur auf einige typische Teilaufgaben eingegangen.

Eine wichtige Grundaufgabe ist das zeichenweise Lesen der Eingabe.



Eine Funktion `next_char()` liefert das nächste Zeichen der Eingabe und setzt dann den (gedachten) Lesekopf um eine Position nach rechts. Der Scanner liest bei Bedarf ein Zeichen aus der Eingabe und entscheidet dann, ob dieses Zeichen noch zu dem bearbeiteten Lexem gehören kann, oder ob es bereits das erste Zeichen eines neuen Lexems ist. Dabei tritt das Problem auf, daß der Scanner eine gewisse Vorausschau (lookahead) haben müßte:



Wenn in der gezeichneten Situation ein neues Lexem mit dem Zeichen „<“ beginnt dann könnte ein „=“ folgen und so das Lexem „<=“ bilden. Um das festzustellen muß das nächste Zeichen gelesen werden (2) danach rückt der Lesekopf aber in die Position (3).

Ist das gelesene Zeichen ein „=“, dann kann das Lexem abgeschlossen werden und der Kopf steht auf dem ersten Zeichen eines neuen Lexems. Ist das Zeichen aber kein „=“, sondern etwa ein „a“ wie in der Zeichnung, dann war „<“ ein Lexem und „a“ ist erstes Zeichen eines neuen Lexems; aber der Lesekopf steht nicht an dieser Position (2), sondern schon eine Position zu weit rechts.

In der IO-Bibliothek von C gibt es dazu die Funktion `ungetch()`, die den Lesekopf wieder genau eine Position nach links versetzt. Wenn der Scanner aber mit der Position (1) jeweils auch schon die Position (2) lesen könnte, dann wäre ein solches Zurücksetzen nicht nötig.

Es gibt Programmiersprachen, wie z.B. FORTRAN, bei denen bestimmte lexikalische Konstrukte auch noch weitere Vorausschau erfordern, aber darauf soll hier nicht weiter eingegangen werden.

In der Praxis wird die Eingabe ohnehin meist „gepuffert“ gelesen. Es werden also bei jedem physikalischen Lesevorgang gleich ganze Abschnitte einer definierten Länge in einen internen Vektor (Array) geladen. Darin kann man auch längere Lookahead-Distanzen realisieren.

Es gibt oft bestimmte lexikalische Vereinbarungen, die nicht explizit durch Produktionen der Grammatik angegeben werden (obwohl das prinzipiell möglich ist):

- Kommentar: Die meisten Programmiersprachen definieren ein Muster für Kommentare. Das sind Lexeme, die der Übersetzer nicht verwendet und daher soll sie der Scanner ohne Ausgabe von Token überlesen (ausfiltern).
- Trenn- und Formatierungszeichen: Solche Zeichen sollen lexikalische Einheiten voneinander trennen und dem Programmtext eine gut lesbare Form geben:

  Zeilenvorschub, Tabulator, Blank u.s.w.

  Der Scanner kann solche Zeichen als spezielle Muster behandeln, die aber kein Token liefern. Mit dem Zeilenvorschubzeichen <LF> kann der Scanner einen Zähler inkrementieren, der zur Fehlerbehandlung benötigt wird.

Spezielle Muster können weitere Aktionen des Scannners erfordern:

- Zahlen: Wenn ein Lexem eine Zahl darstellt, dann kann der Scanner die interne Zahldarstellung berechnen und diesen Zahlenwert als Attribut an den Token binden.
- Bezeichner: Wenn ein Lexem ein Bezeichner ist, dann kann der Scanner das Lexem, also den Bezeichner selbst als String in einer Tabelle ablegen und diesen String als Attribut an den Token binden.
  Der Parser kann später weitere Attribute hinzufügen, die erst aus der syntaktischen Struktur auf der Tokenebene abgeleitet werden können (etwa ob der Bezeichner eine Variable (und damit Speicherplatz) bezeichnet oder eine Konstante (einen Wert)).

- Schlüsselwörter: Die meisten Programmiersprachen definieren bestimmte Zeichenfolgen als neue (Super-)Zeichen, die eine ganz bestimmte syntaktische Bedeutung haben (keywords). Diese haben meist die lexikalische Struktur von Bezeichnern: begin , while , for , do u.s.w. Diese speziellen Bezeichner können in eine Tabelle eingetragen werden (Initialisierung des Scanners) und dann muß der Scanner feststellen ob ein gelesener Bezeichner in dieser Tabelle vorkommt oder nicht. Diese Tabelle kann auch die ohnehin vorhandene Symboltabelle sein.

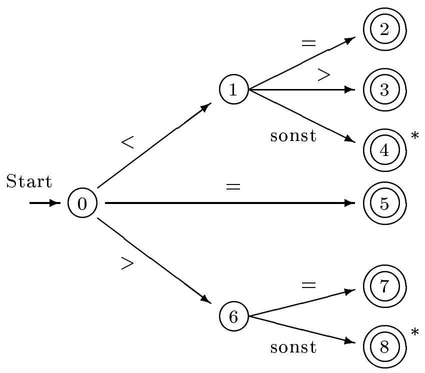
Weitere Hinweise zur ad hoc Programmierung finden sich in der Aufgabenstellung zum Begleitprojekt: PL0-Scanner.

2.2.2 Übergangsdiagramme

Bei dieser Methode entwirft man einen gerichteten Graphen mit Knoten und beschrifteten Kanten. Die Knoten heißen auch Zustände und die Kanten sind mit Zeichen des Eingabealphabets beschriftet. Ein Knoten wird als Startknoten (Startzustand) ausgezeichnet und bestimmte Knoten sind Endknoten (Endzustände).

Bevor ein neues Lexem gesucht wird, steht die Eingabe auf dem ersten möglichen Zeichen dieses Lexems und es wird der Startzustand eingenommen. Durch das erste gelesene Zeichen wird diejenige Kante ausgewählt, die mit diesem Zeichen beschriftet ist, und es findet ein Zustandsübergang statt zum Zustand am Ende dieser Kante. Das wird wiederholt, bis man einen Endknoten erreicht hat.

Beispiel:(nach [1]) Vergleichsoperatoren: Token sind `EQL`, `LEQ`, `NEQ`, `LE`, `GEQ`, `GR` (mit offensichtlicher Bedeutung)



In jedem Endzustand wird ein bestimmter Vergleichsoperator erkannt und damit ein Token bestimmt. Danach geht man zurück in den Startzustand. Die Zustände 4 und 8 sind mit einem $*$ gekennzeichnet. In diesen Zuständen wurde ein Zeichen („sonst“) zuviel gelesen. Dieses muß in die Eingabe zurückgegeben werden. `Ende Beispiel`

Allgemeiner kann man für jeden Endzustand gewisse Aktionen definieren, die beim Erreichen dieses Zustandes ausgeführt werden sollen.

Wenn man ein solches Übergangsdiagramm für alle lexikalischen Elemente entworfen hat, dann kann man dieses Diagramm durch eine geeignete case-Anweisung leicht programmieren.

2.2.3 Verwendung eines Scannergenerators

Wir beziehen uns hier auf den Scannergenerator LEX. Im Anhang findet man eine Kurzbeschreibung von LEX, so daß hier nicht auf die syntaktischen Details der Spezifikationssprache für LEX eingegangen werden muß. Man sollte dazu auch enschlägige Bücher wie z.B. [5] heranziehen. Wir wollen hier den prinzipiellen Aufbau einer LEX-Spezifikation betrachten.

Man kann in LEX Namen für reguläre Ausdrücke verwenden. Diese Namen dürfen dann selbst in anderen regulären Ausdrücken vorkommen. Solche „Abkürzungen“ werden durch Paare:

$$\begin{array}{ll} \mathrm{name}_1 & \alpha_1 \\ \mathrm{name}_2 & \alpha_2 \\ \vdots & \vdots \\ \mathrm{name}_n & \alpha_n \end{array}$$

definiert. name_i ist dabei ein selbstgewählter Name und α_i ein regulärer Ausdruck, in dem die Namen name_1, ..., name_{i-1} (also die bereits definierten Namen) vorkommen dürfen. Nach diesem Definitionsteil folgt der Regelteil. Dieser besteht aus Paaren der Form

$$\begin{array}{ll} \beta_1 & S_1 \\ \beta_2 & S_2 \\ \vdots & \vdots \\ \beta_n & S_m \end{array}$$

Die β_i sind reguläre Ausdrücke, in denen die definierten Namen für reguläre Ausdrücke aus dem Definitionsteil verwendet werden können. Die S_i sind sogenannte Aktionen. Diese Aktionen sind aber einfach C-Anweisungen und die Anweisung S_i wird ausgeführt, wenn ein Lexem gelesen wurde, das dem durch β_i definierten Muster genügt.

Nach dem Definitionsteil folgt ein Teil, in dem C-Funktionen definiert werden können. Das wird insbesondere benutzt, um diese Funktionen dann in den Anweisungen S_i aufrufen zu können.

Noch vor dem Definitionsteil für reguläre Ausdrücke kann man Konstanten, Variablen und Datentypen definieren, die man in den Funktionen oder auch global verwenden will.

Als Beispiel kann man nochmals die LEX-Spezifikation von S. 12,b genauer betrachten. Eine solche Spezifikation wird dann vom Scannergenerator (also LEX oder FLEX) in ein C-Programm übersetzt. Dieses C-Programm enthält insbesondere die Funktion `yylex()`, die bei jedem Aufruf den nächsten Token zurückgibt. Wenn man an ein Token Attributwerte binden will, die man aus dem zugehörigen Lexem berechnen kann, dann muß man diese Bindung explizit programmieren und mit der entsprechenden Aktion dann ausführen.

Die oben erwähnten Bücher enthalten sehr viele Beispiele für LEX.

2.3 Endliche Automaten zur Mustererkennung

Für die lexikalische Analyse werden zu jedem Token $t \in T$ (aus dem Tokenalphabet) reguläre Muster (Sprachen) $L_t \subseteq Z^*$ definiert. Diese Sprachen L_t können durch reguläre Ausdrücke α_t $(t \in T)$ spezifiziert werden. Dabei ist jedes $\alpha_t \in \mathrm{Reg}(Z)$. Die lexikalische Analyse soll dann ein Wort $w \in Z^*$ in Abschnitte zerlegen: $w = w_1 ... w_n$ $(w_i \in Z^*)$ so daß zu jedem w_i ein Token $u_i \in T$ gehört mit: $\forall i \in \{1, ..., n\} : w_i \in L_{u_i}$.

Um dieses Problem algorithmisch zu lösen, kann man zunächst zu jedem $t \in T$ einen nichtdeterministischen Akzeptor (NFA) konstruieren, der die Sprache $L_t = \varphi(\alpha_t) \subseteq Z^*$ erkennt. Sei M_t ein solcher NFA für jedes $t \in T$. Diese Automaten kann man dann, wie auf Seite 42 beschrieben, zur Analyse verwenden.

Mehrdeutigkeiten können ebenfalls entstehen durch:

- Ein Lexem gehört zu zwei (oder mehreren) verschiedenen Mustern.
- Ein Anfangsstück (Präfix) eines Lexems gehört zu einem Muster. Soll dann der Token für das Anfangsstück oder für das „längere" Lexem ausgegeben werden?

Um solche Mehrdeutigkeiten (Konfikte) aufzulösen, führt man geeignete Zusatzregeln ein. Man kann die regulären Ausdrücke (oder einfach die Tokenmenge T) in eine lineare Anordnung bringen: $t_1 < t_2 < ... < t_s$. Wenn jetzt zwei Automaten M_{t_i} und M_{t_j} gleichzeitig in einen Endzustand übergehen, dann wird der „kleinere" Token ausgegeben. Die Präfix-Mehrdeutigkeit kann man durch die Regel beseitigen, daß man ein Lexem erst dann akzeptiert, wenn keine Verlängerung zu einem gültigen Lexem führt (jeweils ein Lexem maximaler Länge). Mit diesen Verabredungen wird die Zerlegung der Eingabe eindeutig.

Wenn sich die Eingabe nicht auf diese Weise in eine Folge von Token transformieren läßt, liegt ein lexikalischer Fehler vor, der dann angezeigt werden kann (Fehlerbehandlung).

Es soll jetzt beschrieben werden, wie man zu regulären Ausdrücken α entsprechende NFA M_α konstruiert so daß gilt: M_α akzeptiert $\varphi(\alpha)$. Diese Automaten werden dann in einem zweiten Schritt durch einen Algorithmus simuliert, der auch die Mehrdeutigkeiten nach den obigen Regeln auflöst.

Zunächst soll auf den regulären Ausdrücken eine „Norm" (eine Längenabschätzung) definiert werden.

Definition 2.3.1 *Sei Z ein Alphabet. Auf* $\text{Reg}(Z)$ *definiert man die Abbildung* $||.|| : \text{Reg}(Z) \to I\!N$ *durch*

1. $||\underline{0}|| = 1$,
2. $\forall z \in Z : ||z|| = 1$,
3. $||(\alpha + \beta)|| = ||(\alpha \cdot \beta)|| = ||\alpha|| + ||\beta|| + 1$,
4. $||(\alpha^*)|| = ||\alpha|| + 1$.

$||\alpha||$ ist die Länge der Zeichenreihe, die diesen Ausdruck darstellt, wobei die Klammern nicht mitgezählt werden.

Ein NFA ist ein 5-Tupel

$$M = (Z, S, \delta, s_0, F)$$

mit $s_0 \in S$ und $F \subseteq S$ und $\delta : S \times Z_\epsilon \to \mathcal{P}(S)$ (oder äquivalent: $\delta \subseteq (S \times Z_\epsilon) \times S$).

Die Übergangsrelation eines solchen Automaten ist eine Relation $R \subseteq K \times K$ und $K = SZ^*$. R ist auf Seite 38 definiert durch:

1. $(szw)R(s'w) \iff (s,z)\delta s'$
   oder

2. $(sw)R(s'w) \iff (s,\epsilon)\delta s'$

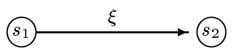
für $s, s' \in S$, $z \in Z$ und $w \in Z^*$. Eine Endkonfiguration liegt vor, wenn das Eingaberestwort leer ist und der Zustand in F liegt: $E = F$. Man definiert daher:

$$w \in L(M) \iff \exists s \in F : (s_0 w)R^* s.$$

M akzeptiert die Sprache $L(M)$. Zu einem NFA kann man den Übergangsgraphen $\Gamma = (Q, S)$ definieren mit $Q \subseteq S \times Z_\epsilon \times S$ und

$$(s_1, \xi, s_2) \in Q \iff (s_1, \xi)\delta(s_2) \text{ mit } \xi \in Z_\epsilon.$$

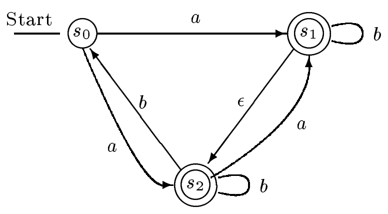
Jedes Element $q = (s_1, \xi, s_2)$ ist eine gerichtete Kante von s_1 nach s_2, die mit $\xi \in Z_\epsilon$ beschriftet ist:



Beispiel: $Z = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, $F = \{s_1, s_2\}$ und δ ist als Abbildung $\delta : S \times Z_\epsilon \to \mathcal{P}(S)$ definiert durch die Tabelle:

	a	b	ϵ
s_0	$\{s_1, s_2\}$	$\emptyset$	$\emptyset$
s_1	$\emptyset$	$\{s_1\}$	$\{s_2\}$
s_2	$\{s_1\}$	$\{s_0, s_2\}$	$\emptyset$

Der Übergangsgraph ist dann:



`Ende Beispiel`

Ein NFA mit genau einem Endzustand soll im folgenden symbolisch dargestellt werden durch den Anfangs- und den Endzustand:

Solche Automaten können leicht zusammengesetzt werden zu neuen Automaten.

Satz 2.3.2 *Sei $\alpha \in \mathrm{Reg}(Z)$, dann kann man einen NFA $M = (Z, S, \delta, s_0, F)$ explizit konstruieren, so daß gilt:*

1. $L(M) = \varphi(\alpha)$,

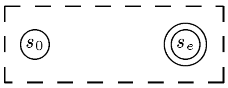
2. $|S| \leq 2||\alpha||$ *und* $|F| = 1$,

3. $x \in Z_\epsilon$, $s \in F \Rightarrow \delta(s, x) = \emptyset$,

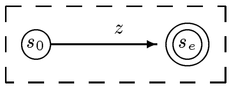
4. $\forall s \in S : \left|\bigcup_{x \in Z_\epsilon} \delta(s, x)\right| \leq 2$.

Beweis:(Skizze): Der Beweis wird induktiv über den Aufbau der regulären Ausdrücke geführt.

1. $||\alpha|| = 1$, also $\alpha = \underline{0}$ oder $\alpha \in Z$. Die NFA's werden explizit durch ihre Übergangsgraphen angegeben:



$\alpha = \underline{0}$
$\varphi(\alpha) = \emptyset$

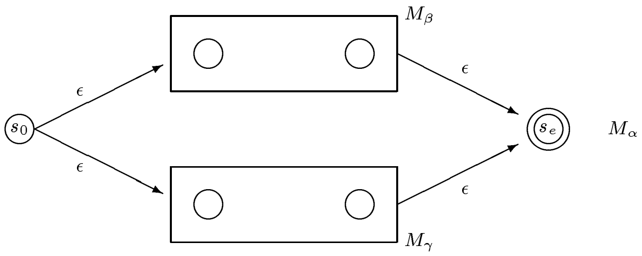


$(z \in Z)$
$\alpha = z$, $\varphi(\alpha) = \{z\}$

Man prüft leicht, daß alle vier Aussagen des Satzes auf diese Automaten zutreffen.

2. Ein Ausdruck α mit $||\alpha|| > 1$ hat genau eine von drei möglichen Formen:

(a) $\alpha = (\beta + \gamma)$
Wegen $||\beta|| < ||\alpha||$ und $||\gamma|| < ||\alpha||$ existieren die Automaten M_β und M_γ, die $\varphi(\beta)$ bzw. $\varphi(\gamma)$ akzeptieren und auch die anderen Bedingungen des Satzes erfüllen. Man konstruiert dann M_α wie folgt:



Dann gilt offensichtlich:

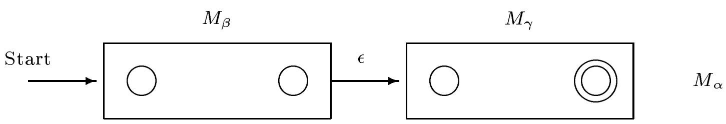
i. $L(M_\alpha) = \varphi(\beta + \gamma)$,

ii. $|S_\alpha| = |S_\beta| + |S_\gamma| + 2 \leq 2||\beta|| + 2||\gamma|| + 2 = 2(||\beta|| + ||\gamma|| + 1) = 2||\alpha||$ und $|F_\alpha| = 1$,

iii. S_e hat keinen Nachfolgezustand,

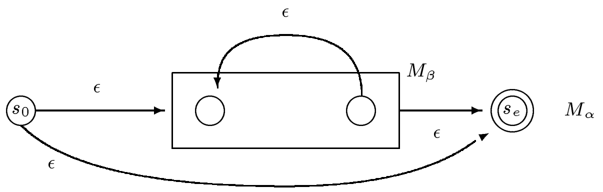
iv. kein Zustand hat mehr als 2 Nachfolger.

(b) $\alpha = (\beta \cdot \gamma)$
Voraussetzungen wie unter 2a. Konstruktion:



M_α akzeptiert die Konkatenation von $L(M_\beta)$ und $L(M_\gamma)$, also $\varphi(\beta \cdot \gamma)$ und die restlichen Eigenschaften zeigt man analog wie in 2a.

(c) $\alpha = (\beta^*)$, Konstruktion:



Der Automat M_α erkennt ein Wort genau dann, wenn es eine endliche Folge von Wörtern aus $L(M_\beta)$ ist. Somit gilt $L(M_\alpha) = \varphi(\beta^*)$

$$|S_\alpha| = |S_\beta| + 2 \leq 2||\beta|| + 2 = 2(||\beta|| + 1) = 2||\alpha||.$$

Der Rest gilt wieder offensichtlich.

□

Der Beweis ist konstruktiv, d.h. er definiert einen Algorithmus, mit dem man zu jedem regulären Ausdruck $\alpha \in \mathrm{Reg}(Z)$ den NFA M_α mit den obigen Eigenschaften konstruiert.

Wenn man zu einem regulären Ausdruck α den NFA M_α nach obiger Konstruktion erzeugt hat, dann ist dieser wegen des nichtdeterministischen Übergangsverhaltens zunächst nicht unmittelbar als Algorithmus zu verwenden. Aber man kann diesen Nichtdeterminismus „simulieren", indem man nach jeder Eingabe die Menge aller möglichen Zustände berechnet, in denen der nichtdeterministische Automat sein könnte. Das sind alle unmittelbar unter dem Eingabezeichen erreichbaren Zustände die man aus diesen durch endliche Folgen von ϵ-Übergängen erreichen kann. Man definiert daher zu einer Teilmenge der Zustandsmenge den ϵ-Abschluß.

Definition 2.3.3 *Sei $M = (Z, S, \delta, s_0, F)$ ein NFA und $T \subseteq S$, dann sei*

$$\mathrm{cl}(T) = \{s \in S | \exists s' \in T : s' \overset{*}{\underset{R}{\rightarrow}} s\}.$$

R^* ist dabei wieder die reflexive und transitive Hülle der globalen Übergangsrelation $R \subseteq K \times K$ im Konfigurationsraum $K = SZ^*$ des Automaten.

Der Algorithmus (von Thompson) ermittelt zu jedem Anfangsstück v des gelesenen Wortes $w \in Z^*$ (also $w = vu$) die Menge $H(v) \subseteq S$ der Zustände, in denen der Automat nach Eingabe von v sein kann:

$$H(\epsilon) = \mathrm{cl}(\{s_0\}),$$

$$H(vz) = \mathrm{cl}\left(\bigcup_{s \in H(v)} \delta(s, z)\right).$$

Der folgende Algorithmus simuliert das Verhalten eines NFA $M = (Z, S, \delta, s_0, F)$, indem ständig die Menge $H(v)$ für jedes Anfangsstück v der Eingabe w berechnet wird. Wenn dann gilt: $H(w) \cap F \neq \emptyset$, dann wird w akzeptiert.

Algorithmus zur Simulation eines NFA

$M = (Z, S, \delta, s_0, F)$

begin

$T := \{s_0\}$;
$T := \mathrm{cl}(T)$;
$z :=$ get_next_char;
while $z \neq$ eof do
  begin
  $T := \bigcup_{s \in T} \delta(s, z)$;
  $T := \mathrm{cl}(T)$;
  $z :=$ get_next_char;
  end;
if $T \cap F \neq \emptyset$ then accept
  else reject;
end ;

Die Details des Algorithmus, also die Umsetzung in ein konkretes Programm (C, PASCAL, MODULA) sei dem Leser empfohlen. Dabei sind die Mengenoperationen und die Operation $\mathrm{cl}(T)$ geeignet zu implementieren.

Satz 2.3.4 *Sei $M = (Z, S, \delta, s_0, F)$ ein NFA mit der Eigenschaft*

$$\forall s \in S : |\{(s', x) \in S \times Z_\epsilon | s' \in \delta(s, x)\}| \leq k,$$

dann benötigt der Algorithmus von Thompson für ein Wort $w \in Z^*$ $f(|w|)$ Schritte mit

$$f \in O(k \cdot |w| \cdot |s|).$$

Beweis: Um für eine Teilmenge $T \subseteq S$ den ϵ-Abschluß zu bestimmen, betrachtet man zu jedem $s \in T$ alle direkten ϵ-Nachfolger. Das sind wegen der Zusatzbedingung höchstens k Stück. Jeder Nachfolger der noch nicht in T liegt wird neu hinzugenommen. Da man also für höchstens $|s|$ Zustände jeweils höchstens k Nachfolger aufsuchen muß, benötigt man höchstens $|s| \cdot k$ Schritte (Nachsehen in der Tabelle), um $\mathrm{cl}(T)$ zu berechnen.

Die while-Schleife wird genau $|w|$-mal durchlaufen und im Rumpf der while-Schleife benötigt man für

$$T := \bigcup_{s \in T} \delta(s, z) \text{ und } T := \mathrm{cl}(T)$$

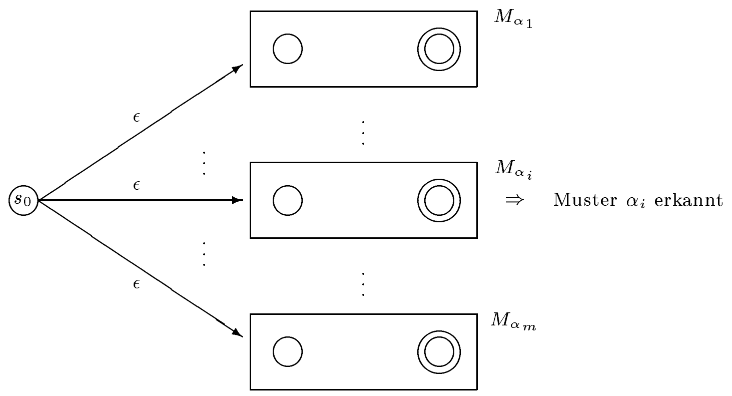
jeweils höchstens $k \cdot |s|$ Schritte. Somit ist die Laufzeit aus $O(k \cdot |s| \cdot |w|)$. □

Ist also $\beta \in \mathrm{Reg}(Z)$ und $w \in Z^*$, dann benötigt der Algorithmus von Thompson zum Erkennen (oder Ablehnen) von w höchstens $O(2 \cdot |s| \cdot |w|) = O(||\beta|| \cdot |s|)$ Schritte, denn es gilt $|s| \leq 2 \cdot ||\beta||$. Der Algorithmus ist also linear in $|w|$.

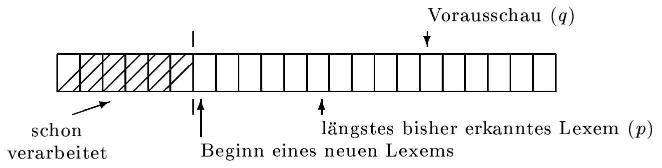
Für die lexikalische Analyse sind nun mehrere reguläre Ausdrücke gegeben:

$$\alpha_1, \alpha_2, ..., \alpha_m \in \mathrm{Reg}(Z).$$

Zu jedem α_i kann man den NFA M_{α_i} konstruieren und daraus kann man einen neuen NFA $M = (Z, S, \delta, s_0, F)$ wie folgt konstruieren:



Diesen Automaten für $\alpha = \alpha_1 + \alpha_2 + ... + \alpha_m$ kann man nun mit dem Algorithmus von Thompson simulieren. Dabei wird der Algorithmus so modifiziert, daß man die weiter oben dargestellten Konfliktsituationen entsprechend auflöst. Sei dazu das Eingabewort w wie folgt zerlegt:



Nachdem ein Lexem erkannt wurde steht die Eingabe auf dem Beginn eines neuen Lexems. Diese Position wird geeignet markiert (gespeichert). Der Automat wird in den Anfangszustand s_0 versetzt. Ein Vorausschau-Zeiger q liest die folgenden Zeichen und steuert so den Automaten M. Wird nun ein Muster α_i erkannt (also ein Lexem aus $\varphi(\alpha_i)$), dann sind folgende Aktionen auszuführen:

1. Ein Zeiger p für das längste bisher erkannte Lexem wird auf die Position von q gesetzt.
2. α_i wurde erkannt weil in der aktuellen Zustandsmenge T der Endzustand von M_{α_i} enthalten ist. Falls weitere Endzustände in T enthalten sind, dann wird das Muster mit dem kleinsten Index zusammen mit der Position p gespeichert. Sei also α_i das Muster mit dem kleinsten Index.
3. Der Vorausschau-Zeiger q läuft aber weiter und steuert M in neue Zustandsmengen.
4. Wenn eine Zustandsmenge T erreicht wird, in der es unter dem aktuellen Eingabezeichen keinen Nachfolger mehr gibt, dann ist das längste Lexem durch p und α_i charakterisiert.

   Der Token für α_i wird ausgegeben und die Eingabe wird auf die Position von $q + 1$ gesetzt.

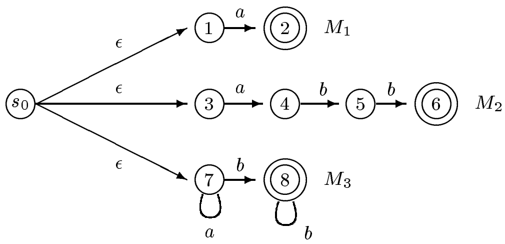
Damit ist der Algorithmus hinreichend genau beschrieben. Die Organisation mit den Zeigern muß durch eine geeignete Pufferung der Eingabe implementiert werden. Durch geeignete Wahl der lexikalischen Syntax kann man erreichen, daß die Lexeme eine beschränkte Länge haben, so daß man mit einem Puffer fester Länge arbeiten kann.

Lexikalische Fehler können durch das Muster $Z^* \setminus \varphi(\alpha)$ erkannt und behandelt werden. Das Verfahren soll an einem (sehr einfachen) Beispiel erläutert werden.

Beispiel:(nach [1]): Es werden drei Muster spezifiziert für die Token t_1, t_2, t_3 und $Z = \{a, b\}$. Die zugehörigen regulären Ausdrücke:

t_1:	a	(α_1)
t_2:	abb	(α_2)
t_3:	a^*bb^*	(α_3)

Zugehöriger NFA als Graph:



Eingabe	Zustandsmenge T		
$\underline{a}\ a\ b\ a\ b\ b\ \#$	$\{s_0, 1, 3, 7\}$		
$\underline{a}\ a\ b\ a\ b\ b\ \#$	$\{\underline{2}, 4, 7\}$		(α_1 erkannt)
$\underline{a}\ a\ b\ a\ b\ b\ \#$	$\{7\}$		
$\underline{a}\ a\ b\ a\ b\ b\ \#$	$\{\underline{8}\}$		(α_3 erkannt)
$\underline{a}\ a\ b\ a\ b\ b\ \#$	$\emptyset$	$\Rightarrow$	Lexem: aab, Token: t_3
$a\ a\ b\ \underline{a}\ b\ b\ \#$	$\{s_0, 1, 3, 7\}$		
$a\ a\ b\ \underline{a}\ b\ b\ \#$	$\{\underline{2}, 4, 7\}$		(α_1 erkannt)
$a\ a\ b\ \underline{a}\ b\ b\ \#$	$\{\underline{8}, 5\}$		(α_3 erkannt)
$a\ a\ b\ a\ b\ b\ \#$	$\{\underline{8}, \underline{6}\}$		(α_2, α_3 erkannt)
	$\emptyset$	$\Rightarrow$	Lexem: abb, Token: t_2

`Ende Beispiel`

Man kann für einen Scanner natürlich auch einen deterministischen Automaten (DFA) konstruieren und diesen dann direkt für den Algorithmus verwenden. Der Vorteil ist, daß man zu jedem Zeitpunkt nur in genau einem Zustand ist. Allerdings erkauft man sich diesen Vorteil durch eine Zustandsmenge, die nicht linear, sondern exponentiell mit der Länge der regulären Ausdrücke wachsen kann.

Es soll hier gezeigt werden, wie man aus einem NFA $M = (Z, S, \delta, s_0, F)$ einen DFA $M' = (Z, S', \delta', s'_0, F')$ konstruiert, der äquivalent zu M ist, d.h. $L(M) = L(M')$: beide erkennen die gleiche Sprache.

Konstruktion des DFA:

1. Man ersetzt alle ϵ-Übergänge: Sei $\Gamma = (Q, S)$ der Übergangsgraph von M, dann sei $\overline{\Gamma} = (\overline{Q}, S)$ definiert durch:
$$\overline{Q} = Q \cup \{(s, z, s') | sz \xrightarrow[R]{*} s'\} \setminus \{(s, \epsilon, s') | (s, \epsilon, s') \in Q\}.$$
Auf diese Weise werden alle ϵ-Übergänge (Kanten) entfernt und ihre Wirkung durch neue Kanten, die nur noch in $S \times Z \times S$ liegen, ersetzt.

   $\overline{F} = \mathrm{cl}(F)$ ist die neue Endzustandsmenge. Man kann zeigen, daß dieser ϵ-freie Automat die gleiche Sprache erkennt wie M.

2. Sei $\overline{M} = (Z, S, \overline{\delta}, s_0, \overline{F})$ der zu M konstruierte ϵ-freie äquivalente NFA.

   Man konstruiert dazu den Potenzautomaten, dessen Zustände die Teilmengen von S sind:

$$S' = \mathcal{P}(S), s'_0 = \{s_0\}, \delta'(T, z) = \bigcup_{t \in T} \overline{\delta}(t, z), F' = \{T \subseteq S | T \cap \overline{F} \neq \emptyset\}.$$

   Dieser Automat ist deterministisch und leistet im Prinzip das gleiche, wie der Algorithmus von Thompson. Wegen $S' = \mathcal{P}(S)$ gilt aber $|S'| = 2^{|S|}$ und mit $|S| \leq 2||\alpha||$ gilt $|S'| \leq 2^{2||\alpha||} = 4^{||\alpha||}$.

   Da der Automat die Eingabe schrittweise abarbeitet ist die Laufzeit der Analyse weiterhin linear in der Länge der Eingabe, aber der Speicherverbrauch wächst unter Umständen exponentiell mit der Länge der regulären Ausdrücke.

   Bei dieser Konstruktion des Potenzautomaten entstehen meist aber viel zu viele (unnötige) Zustände. Man kann zu jedem deterministischen endlichen Akzeptor einen äquivalenten Automaten konstruieren der unter allen äquivalenten Automaten die kleinste Anzahl von Zuständen hat.

Satz 2.3.5 *Zu jedem DFA $M = (Z, S, \delta, s_0, F)$ existiert ein äquivalenter DFA $M' = (Z, S', \delta', s_0, F')$, so daß gilt: Wenn M'' ein DFA ist mit $L(M'') = L(M)$ dann gilt $|S'| \leq |S''|$.*

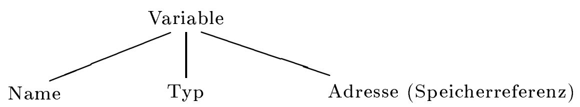
Der Beweis soll hier nicht geführt werden. Er ist konstruktiv, enthält also einen Algorithmus zur Erzeugung des minimalen Automaten.

Kapitel 3

Symboltabellen

3.1 Die Aufgaben der Symboltabelle

Wenn in einem Programm eine Variable (explizit, wie in PASCAL oder C; oder implizit wie in FORTRAN) definiert wird, dann wird eine Beziehung (Bindung) zwischen der Variablen und ihren Attributen hergestellt.



Diese Attribute können zu einer Variablenbeschreibung als Record zusammengefaßt werden, z.B.

Attribut	Wert des Attributes
Art:	Variable
Name:	'X5'
Typ:	integer
Adresse:	128051

Das Attribut „Art“ zeigt an, daß es sich um eine Variable handelt. Andere Bezeichner (identifier) könnten etwa an Konstante, Funktionen oder Datentypen gebunden sein, die dann gegebenenfalls auch andere Attribute besitzen.

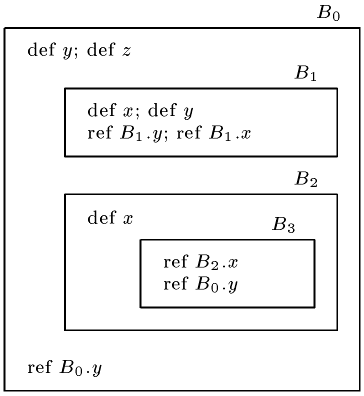
Der Compiler legt eine Tabelle mit solchen Einträgen an. Diese Tabelle heißt auch *Symboltabelle*. Die Einträge in diese Tabelle werden meist schon vom Scanner vorgenommen, aber die konkreten Attributwerte können erst während der semantischen Analyse (wenn der dekorierte Syntaxbaum vorliegt) bestimmt werden. Die vollständige Symboltabelle wird dann für die Codeerzeugung benötigt und sie wird in einer geeigneten Form sogar in das Zielprogramm eingebunden, wenn zur Laufzeitanalyse ein symbolischer Debugger verwendet werden soll. Die meisten Compiler besitzen dazu eine entsprechende Compiler-Option.

Die Einträge in dieser Tabelle müssen gesucht werden. Die Suche wird nach den Namen (Bezeichnern) vorgenommen durch einen Vergleich. Der Tabelle müssen neue Einträge zugefügt werden, wenn ein Bezeichner entdeckt wird (vom Scanner), der noch nicht in der Tabelle vorkommt, und Einträge können aus der Tabelle entfernt werden, wenn die Bearbeitung (einschließlich Codeerzeugung) einen lokalen Block verlassen hat. Für den Debugger muß allerdings auch zum Schluß eine vollständige Symboltabelle zur Verfügung stehen. Diese letzte Forderung wollen wir aber in diesem Kapitel nicht weiter berücksichtigen.

Da der Compiler sehr häufig auf die Symboltabelle zugreifen muß, muß man auf eine effiziente Implementierung der Zugriffsfunktionen besonderes Gewicht legen.

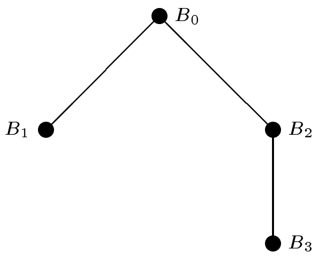
Die Symboltabelle muß die Gültigkeitsregeln der Programmiersprache berücksichtigen. Bei einer Programmiersprache mit Blockstruktur kann der gleiche Bezeichner zu einem Zeitpunkt mehrfach in der Tabelle mit verschiedenen Bedeutungen (Attributen) vorkommen.

Die folgende schematische Skizze soll das verdeutlichen. Es werden geschachtelte Blöcke B_0, B_1, B_2, B_3 durch geschachtelte Rechtecke dargestellt. def x ist die Definitionsstelle des Bezeichners x und ref x eine Verwendung (Referenz) des Bezeichners x. Die Referenzen sind durch Voranstellen des Blocknamens mit der zuständigen Definition aufgelöst.



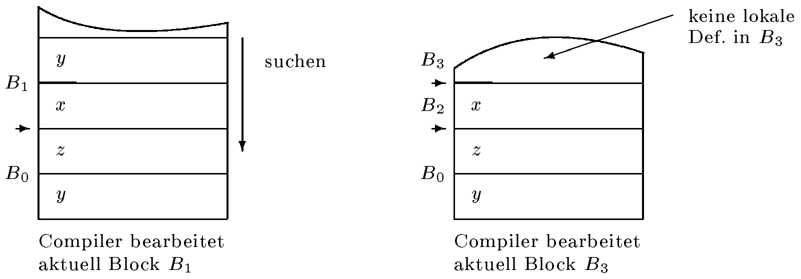
Die Blockbezeichnungen sind in den Referenzen implizit und hier explizit hinzugefügt worden (In der Programmiersprache ADA kann man auf diese Weise sogar auf „verdeckte“ Variablen zugreifen).

Die Blockstruktur im obigen Bild kann (wie jede geschachtelte Struktur) durch einen Baum dargestellt werden:

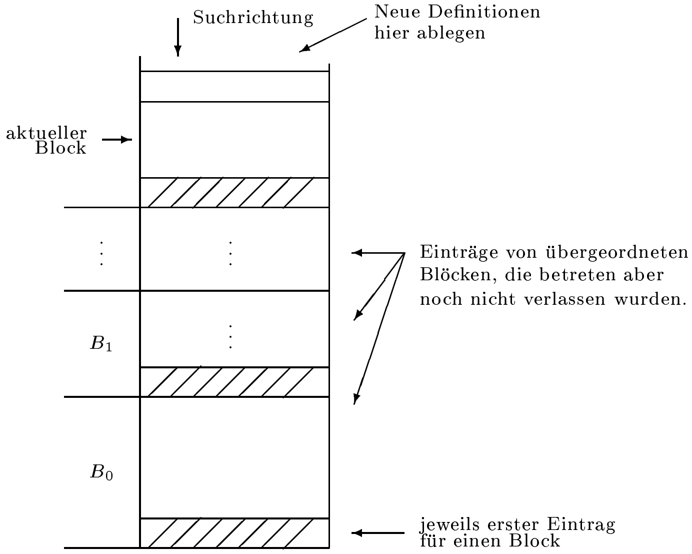


Wenn der Compiler einen Block (z.B. B_3) bearbeitet, dann sind nur die Definitionen derjenigen Blöcke relevant, die auf dem eindeutigen Pfad zur Wurzel des Baumes liegen (hier also B_3, B_2, B_0).

Betrachtet man die lokalen Definitionen in einem Block als lokale Symboltabelle, dann kann man die gesamte Symboltabelle als einen Stapel (Stack) von lokalen Symboltabellen betrachten. Das soll auch an dem obigen Beispiel verdeutlicht werden:



Wenn nun die Definition zu einer Referenz gesucht wird, dann sucht man den Bezeichner einfach von oben beginnend nach unten. Das erste Vorkommen bei dieser Suche ist die korrekte Definition. Die Symboltabelle kann also abstrakt als ein Stapel von lokalen Symboltabellen aufgefaßt werden. Wenn der Compiler einen neuen Block betritt, dann wird die Position auf dem Stapel geeignet markiert. Für jede Definition in diesem Block wird ein Eintrag (Bezeichner, Attribute) auf dem Stapel abgelegt. Bei jeder Referenz wird die Definition von oben beginnend nach unten gesucht. Wird der Block verlassen, dann können alle lokalen Einträge für diesen Block vom Stapel entfernt werden.



Die Symboltabelle für blockstrukturierte Sprachen kann logisch als ein Stapel von lokalen Symboltabellen betrachtet werden. Jede lokale Symboltabelle ist eine Folge von Einträgen.

Die Operationen, die auf einer Symboltabelle ausgeführt werden müssen sind (mindestens):

- insert(eintrag): Neuen Eintrag einfügen.
- look_up(bezeichner): Finde Eintrag mit: bezeichner.
- delete(bezeichner): Entferne Eintrag mit: bezeichner.
- new_block: Markiere Blockanfang.
- remove_block: Entferne alle Einträge des aktuellen Blockes.

3.2 Implementierungen von Symboltabellen

Sei Id die Menge der möglichen Bezeichner. Die Attribute zu einem Bezeichner sind Records, deren Struktur aber vom bezeichneten Objekt abhängt. Dafür kann man einen varianten Record-Typ verwenden (union in C). Der Datentyp für Attribute hat also die Struktur

$$T = \bigcup_{i=1}^{m} T_i \quad m \text{ verschiedene Varianten,}$$

$$T_i = T_i^1 \times T_i^2 \times ... \times T_i^{n_i} \text{ die Struktur der Variante } T_i.$$

Die Menge der Einträge ist dann:

$$E = Id \times T$$

und eine Symboltabelle ist eine Folge solcher Einträge: $e_1, ..., e_k$. Der Bezeichner ist das Suchargument (Schlüsselfeld). Die Operation look_up(id) liefert einen Zeiger auf den korrekten Eintrag e_i. insert(e) fügt einen neuen Eintrag in die Tabelle ein. delete(id) löscht den entsprechenden Eintrag. Betrachten wir zunächst eine lokale Symboltabelle (ohne Berücksichtigung von Blockstrukturen).

Die Einträge können wie oben durch einen varianten Record-Typ dargestellt werden. Während aber die Attributwerte meist wenige Bytes für die Speicherung benötigen (integer oder Aufzählungstypen), kann der Platzbedarf für Bezeichner erheblich schwanken. Daher verwendet man für den Bezeichner-String oft einen dynamisch angeforderten Speicherplatz (new, malloc) und speichert in dem Eintrag nur einen Zeiger (Pointer) auf den String.

Die Tabelle selbst sollte als dynamische Datenstruktur implementiert werden, da die Zahl der Einträge für verschiedene Quelltexte stark schwanken kann.

Für eine effiziente Implementierung der Tabellenoperationen kann man die bekannten Suchverfahren und ihre Datenstrukturen verwenden:

1. Linear verkettete Listen.
2. Hash-Tabellen.
3. Binäre Suchbäume.

Wir wollen hier nur kurz auf 1. und 2. eingehen, da die Verfahren als bekannt vorausgesetzt werden können.

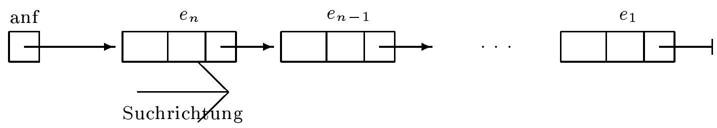
Methode 1. ist einfach zu implementieren, aber nicht so effizient wie 2. (oder 3.), denn die Suchzeit ist aus $O(n)$, also linear in der Zahl der Einträge.

Methode 2. ist aufwendiger zu implementieren, aber dafür ist die mittlere Suchzeit günstiger als für Methode 1.

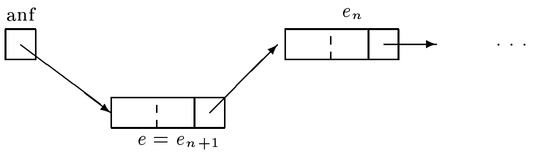
Methode 1:

Die Einträge erhalten noch ein weiteres Feld für einen Zeiger auf Einträge:

Diese Einträge lassen sich zu einer Liste verketten:



- Look_up(id): ein Zeiger durchläuft die Liste, beginnend bei anf.
  Wenn eine Übereinstimmung (String-Vergleich) gefunden wird, dann wird der Zeiger auf diesen Eintrag als Funktionswert zurückgegeben. Wird kein passender Eintrag gefunden, wird der nil-Zeiger zurückgegeben.
- insert(e): Der neue Eintrag e wird einfach vorn in der Liste eingehängt:



  So ist der zuletzt definierte Eintrag der erste für die Suche (Stack-Prinzip!) Die Suchzeit ist offensichtlich linear in n.

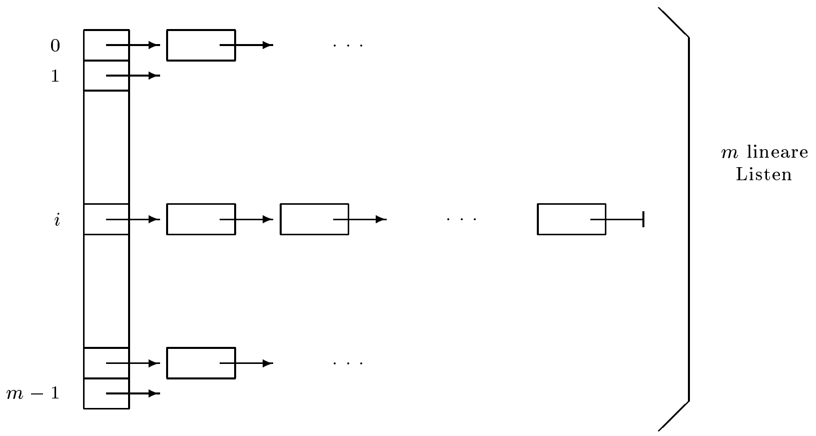
Methode 2: Hash-Tabellen

Die Idee besteht darin, die lange lineare Liste von Methode a in m möglichst gleich lange Listen zu zerlegen, so daß man aus dem Suchargument (String) direkt den Index $i \in \{0, ..., m-1\}$ der betreffenden Liste berechnen kann. Die Rechenzeit zur Berechnung eines Index sollte unabhängig sein von der Zahl der vorkommenden Bezeichner.

Man wählt dazu eine Zahl $m \in \mathbb{N}$ und eine Array-Variable H mit m Zeigern auf Einträge:

$$\underline{\text{var}}\ H\colon\ \underline{\text{array}}\ [0..m-1]\ \ \underline{\text{of}} \uparrow E.$$

Außerdem definiert man eine Hash-Funktion $h : \text{id} \to \{0, ..., m-1\}$, die jedem möglichen Bezeichner einen Array-Index zuordnet.



Die Funktionen look_up(id) und insert(e) können einfach auf die entsprechenden Funktionen list_lookup(id) und list_insert(e) von Methode a abgestützt werden. Da es jetzt mehrere Listen gibt, werden diese um den Anfangszeiger der Liste zu z.B. list_insert(p, e) modifiziert, wobei p ein Zeiger auf Einträge ist.

- look_up(id) = list_lookup($H[h(\text{id})], \text{id}$)
- insert(e) :
  $k := h(e.\text{id})$;
  list_delete($H[k], \text{id}$);

H wird zu Beginn mit nil-Zeigern initialisiert.

Sei r die Zahl der Einträge die für ein gegebenes Programm benötigt werden. Wenn die Hash-Funktion die Bezeichner (einigermaßen) gleichmäßig auf die Indizes $\{0, ..., m-1\}$ verteilt, dann haben die Listen eine Länge von jeweils etwa $\alpha = r/m$. Die Laufzeit für h wird als konstant vorausgesetzt also als Funktion aus $O(1)$. Die Suchzeit in einer Liste beträgt im Mittel $\alpha/2$ Vergleiche. Damit erhält man eine mittlere Laufzeit aus $O(1 + \alpha/2)$.

Man beachte allerdings, daß die Verteilung der Bezeichner in einem Programm (Block) so ungünstig sein kann (worst-case), daß h jeden Bezeichner auf den gleichen Index i abbildet. Dann geht die Laufzeit über in die nach Methode a.

Die Wahl der Hash-Funktion ist für dieses Verfahren sehr wichtig. Ausführliche Diskussionen dazu findet man in der Literatur über effiziente Algorithmen.

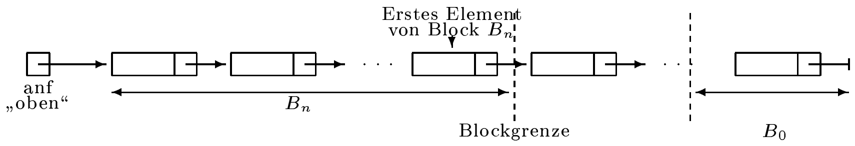
Eine Möglichkeit zur Berechnung von Hash-Werten soll hier kurz vorgestellt werden. Ein Bezeichner (gemeint ist hier das Lexem) ist eine Folge $u = u_1...u_s \in Z^*$. Jedes Zeichen $u_i \in Z$ ist intern durch eine (kleine) Zahl dargestellt. Für ASCII-Zeichen ist etwa $|Z| = 128$ (oder 256) und so entspricht jedes u_i einer Zahl $\varphi(u_i) \in \{0, ..., 127\}$. Sei $|Z| = p$, dann kann man $u = u_1...u_s$ als die p-adische Darstellung einer Zahl $\sigma(u) = \sum_{i=1}^{s} \varphi(u_i) p^{i-1}$ betrachten. Die Zahl $\sigma(u)$ wird aber bereits für kurze Bezeichner einen sehr großen Wert annehmen, und daher muß dieser Wert effizient in eine Zahl $q(u) = \sigma(u) \bmod 2^l \in \mathbb{Z}_{2^l}$ umgerechnet werden, wobei l meist die Zahl der Bit-Positionen für einen integer-Wert (z.B. 16 oder 32) ist. Anschließend kann man dann $h(u) = q(u) \bmod m \in \mathbb{Z}_m$ berechnen und erhält so einen zulässigen Array-Index:

$$\begin{tikzcd}
\mathrm{Id} \arrow[r, "\sigma"] \arrow[d, dashed, "h"'] & \mathbb{N} \arrow[d, "\bmod 2^l"] \\
\mathbb{Z}_m & \mathbb{N} \arrow[l, "\bmod m"]
\end{tikzcd}$$

Da $\sigma(u)$ eine sehr große Zahl sein kann, wird dieses Zwischenergebnis nicht explizit berechnet.

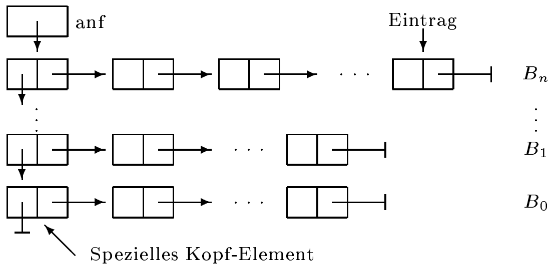
Wir haben bisher lokale Symboltabellen betrachtet. Um die Blockstruktur zu berücksichtigen muß man lokale Symboltabellen für jeden Block anlegen und diese (logisch) auf einen Stapel anordnen. Der aktuelle Block liegt dabei „oben" auf dem Stapel und die ihn umfassenden Blöcke „darunter".

Für lineare Listen gibt es wieder verschiedene Möglichkeiten, dieses Konzept zu implementieren.



1. Man ordnet alle Einträge in der Reihenfolge des Auftretens der Definitionen in die Liste ein. Den letzten Eintrag jeweils „oben" oder am Listenanfang. Das jeweils erste Element eines neuen Blocks erhält eine besondere Markierung. Wenn ein aktueller Block verlassen wird, können alle Elemente bis einschließlich dem markierten Element entfernt werden. Die Suche erfolgt von „oben" nach „unten".

2. Man ordnet die lokalen Symboltabellen in übereinanderliegenden Schichten an. Dazu versieht man entweder jeden Eintrag mit einem weiteren Zeiger für die senkrechte Verkettung, oder man verwendet lineare Listen mit einem speziellen Kopfelement, wie in der folgenden Skizze:

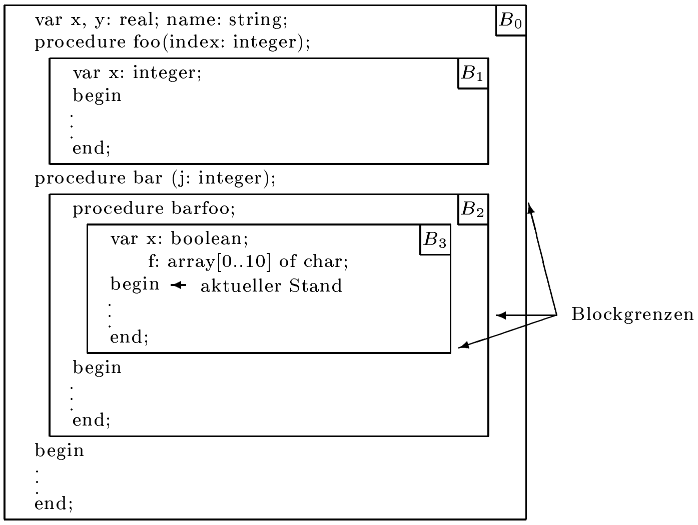


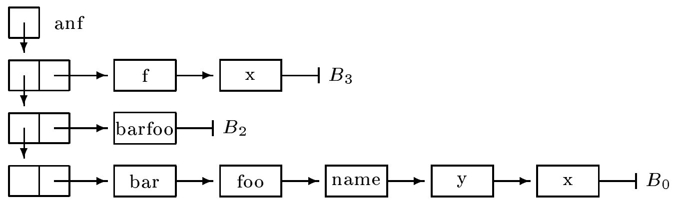
anf zeigt stets auf die obere Ebene (aktueller Block). Der Abwärts-Zeiger zeigt auf die nächst tiefere Ebene (umfassender Block). Wenn ein neuer Block betreten wird, dann wird eine neue Ebene angelegt, die dann „oben" auf diesem Stapel abgelegt wird. Wird ein Block verlassen, dann wird eine ganze Ebene (die „oberste") entfernt. Bei der Suche nach einem Bezeichner werden die Ebenen „von oben nach unten" durchsucht. Man kann natürlich statt der linearen Listen für die lokalen Symboltabellen auch wieder die anderen Datenstrukturen verwenden: binäre Suchbäume, Hashtabellen.

Bei der Verwendung von Hash-Tabellen kann man die Stapel-Struktur auch mit einer einzigen Hash-Tabelle realisieren, wenn man die Einträge, die zu einem Block gehören durch eine zusätzliche Verkettung kennzeichnet.

Beispiel: Symboltabelle mit linearen Listen und Blockstruktur

Programm-Fragment (nur Definitionen)



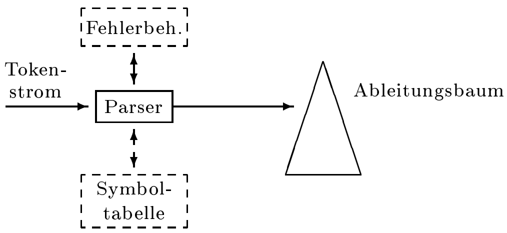


`Ende Beispiel`

Kapitel 4

Syntaxanalyse

4.1 Einführung



Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik. Ein Wort $u \in T^*$ heißt hier auch ein Tokenstrom. Der Parser hat die folgende Aufgabe:

Gegeben: $u \in T^*$ (Tokenstrom)

Gesucht:

Falls $u \in L(G)$: Erzeugung einer schrittweisen Ableitung: $\sigma \rightarrow_P v_1 \rightarrow_P v_2 \rightarrow_P ... \rightarrow_P v_m = u$ oder (äquivalent): Ableitungsbaum.

Falls $u \notin L(G)$: Ausgabe einer Fehlerbeschreibung.

Die Fehlerbeschreibung sollte möglichst viele Fehler in einem Lauf erfassen. Der Parser sollte also die Syntaxanalyse nicht beim Auftreten des ersten Fehlers abbrechen (Fehlerbehandlung). Zur genaueren Beschreibung der verschiedenen Verfahren sind einige Begriffe aus der Theorie zu behandeln.

Zunächst soll der schon häufiger verwendete Begriff Ableitungsbaum präzisiert werden.

Ist M eine Menge und sind $M_1, ..., M_s \subseteq M$ mit

1. $M = \bigcup_{i=1}^{s} M_i$ (Überdeckung),
2. $\forall i \in \{1, ..., s\} : M_i \neq \emptyset$,

3. $\forall i, j \in \{1, ..., s\} : i \neq j \Rightarrow M_i \cap M_j = \emptyset$ (disjunkt)

dann heißt die Menge $\{M_1, ..., M_s\}$ eine *Partition* von M. Wenn man die Folge dieser Teilmengen berücksichtigen muß, dann nennt man die Folge $(M_1, ..., M_s)$ eine *geordnete Partition.*

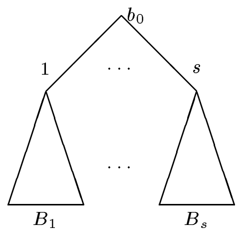
Definition 4.1.1 *Ein endlich geordneter Baum ist eine endliche Menge B zusammen mit einer geordneten Partition $(B_0, B_1, ..., B_s)$ von B, so daß gilt*

1. $|B_0| = 1$,
2. $\forall i \in \{1, ..., s\} : B_i$ *ist geordneter Baum.*

Da $|B_0| = 1$ gilt $B_0 = \{b_0\}$ mit $b_0 \in B$ und man schreibt dann auch kurz:

$$B = (b_0, B_1, ..., B_s)$$

s kann dabei auch 0 sein: $B = (b_0)$. Ein Baum kann als Graph betrachtet werden:

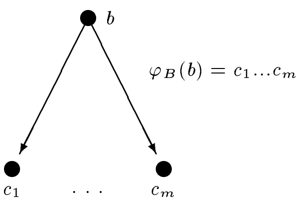


b_0 heißt auch Wurzel von B: $b_0 = \text{root}(B)$.

Sei $B = (b_0, B_1, ..., B_s)$ ein geordneter Baum, dann definiert man die Nachfolger-Abbildung $\varphi : B \to B^*$. Sei dazu $b_i = \text{root}(B_i)$ $(1 \leq i \leq s)$. φ_B wird rekursiv definiert:

1. $\varphi_B(b_0) = b_0$,
2. $b \in B_i$ dann sei $\varphi_B(b) = \varphi_{B_i}(b)$.

$\varphi_B(b)$ ist die (geordnete) Folge der Söhne oder Nachfolger von b



Für φ_B schreibt man auch einfach φ. Die Elemente von B heißen auch *Knoten.* Die Abbildung $\varphi : B \to B^*$ kann auf beliebige Wörter über B zu $\varphi : B^* \to B^*$ fortgesetzt werden:

1. $\varphi(\epsilon) = \epsilon$,
2. $\varphi(ub) = \varphi(u)\varphi(b)$ $(u \in B^*, b \in B)$.

Ein Knoten $b \in B$ heißt *Blatt* g.d.w.: $\varphi(b) = \epsilon$. Der *Grad eines Knotens* ist die Zahl der Söhne: $\deg(b) = |\varphi(b)|$. Der *Rand* von B ist die Folge aller Blätter: $\partial B \subseteq B^*$ wird rekursiv definiert:

1. $B = (b_0) : \partial B = b_0$,
2. $\partial(b_0, B_1, ..., B_s) = \partial B_1 \partial B_2 ... \partial B_s \quad (s \geq 1)$.

Für $u \in B^*$ sei $[u] \subseteq B$ definiert durch

1. $[\epsilon] = \emptyset$,
2. $[ub] = [u] \cup \{b\} \quad (u \in B^*, b \in B)$.

$[u]$ ist die Menge der Elemente, die in u vorkommen.

Zu jedem Knoten $b \in B$ existiert ein eindeutig bestimmter Pfad, der bei b_0 beginnt. Dieser Pfad ist die Folge der Knoten auf denen man von b_0 zu b gelangt. Er wird mit $\pi_B(b) \in B^*$ bezeichnet und wie folgt definiert:

$$\pi_B(b) = \begin{cases} b_0 & \text{falls } b = b_0 \\ b_0 \pi_{B_i}(b) & \text{falls } b \in B_i \ (1 \leq i \leq s) \end{cases}$$

Ist $C \subseteq B$ eine beliebige Teilmenge, dann sei $\overline{C}$ die Menge aller Knoten, die von C aus erreichbar sind:

$$b \in \overline{C} \iff \exists c \in C, i \in I\!N : b \in \varphi^i(c).$$

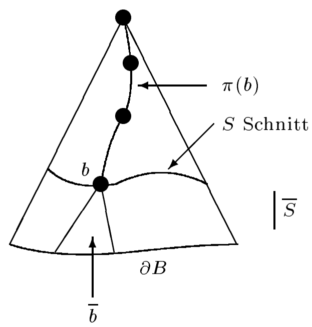
Eine Teilmenge $C \subseteq B$ heißt *präfixfrei* g.d.w. gilt:

$$\forall c \in C : b \in [\pi(c)] \cap C \Rightarrow b = c$$

d.h. kein Vorgänger eines Knotens aus C liegt in C.

Definition 4.1.2 *Eine Teilmenge $S \subseteq B$ heißt* Schnitt $\iff$

1. *S ist präfixfrei,*
2. *$[\partial B] \subseteq \overline{S}$.*



Jeder Knoten $b \in B$ erzeugt einen Teilbaum, in dem b die Wurzel ist:

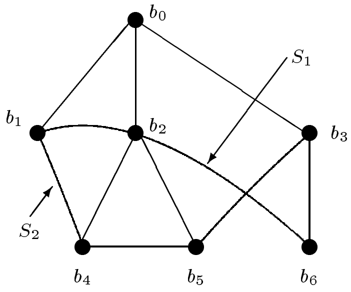
$$\overline{b} = (b, C_1, ..., C_m) \text{ mit } C_i = \overline{c}_i \text{ und } \varphi b = c_1 ... c_m.$$

Ein Schnitt S kann daher auch immer als Wort $S \subseteq B^*$ aufgefaßt werden, wenn man die Elemente von S in der Folge der Randabschnitte anordnet: $S = b_1 ... b_m \in B^*$ mit

$$\partial B = \partial \overline{b}_1 \partial \overline{b}_2 ... \partial \overline{b}_m.$$

Jeder Schnitt erzeugt den Rand und der Rand läßt sich zu jedem Schnitt reduzieren. Man kann Schnitte auch ordnen:

Seien S_1, S_2 zwei Schnitte von B, dann definiert man: $S_1 \sqsubseteq S_2 \iff S_2 \subseteq \overline{S}_1$. In dieser Halbordnung ist der Schnitt ∂B maximal und der Schnitt b_0 minimal. Es gibt auch Schnitte, die in dieser Ordnung nicht miteinander vergleichbar sind:



$\partial B = b_1 b_4 b_5 b_6$
$S_1 = b_1 b_2 b_6$
$S_2 = b_1 b_4 b_5 b_3$

dann ist $\overline{S}_1 = \{b_1 b_2 b_4 b_5 b_6\}$ und $\overline{S}_2 = \{b_1 b_4 b_5 b_3 b_6\}$.

Da $b_2 \in \overline{S}_1$, $b_2 \in \overline{S}_2$ und $b_3 \in \overline{S}_2$, $b_3 \notin \overline{S}_1$ ist weder $\overline{S}_1$ in $\overline{S}_2$ enthalten noch umgekehrt.

Man kann in der Notation die [ ]-Klammern auch einsparen, wenn aus dem Kontext klar ist, ob das Wort u gemeint ist oder die Menge $[u]$.

Lemma 4.1.3 *Sei $b \in \partial B$, dann gilt $\pi(b) \cap S \neq \emptyset$ für jeden beliebigen Schnitt S.*

Beweis: Wegen $b \in \partial B$ existiert ein $c \in S$ mit $b \in \partial \overline{c}$ und dieses c ist eindeutig bestimmt. Also existiert auch der eindeutig bestimmte Pfad $\pi_{\overline{c}}(b) = cc_1 ... c_i = b$. Außerdem ist $\pi_B(c)$ eindeutig bestimmter Pfad von b_0 nach c und somit gilt:

$$\pi_B(b) = \pi_B(c) c_1 ... c_i \text{ mit } c \in S \cap \pi_B(b).$$

□

Übung: Man zeige: Für einen geordneten Baum B gilt:

$$b \in \partial B \iff \varphi(b) = \epsilon.$$

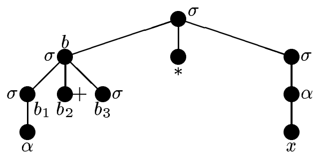
Ist B ein geordneter Baum und $\mu : B \to M$ eine beliebige Abbildung, dann sagt man *der Baum ist durch μ markiert* (*dekoriert*) und (B, μ) heißt *markierter Baum.*

Es soll jetzt eine Beziehung zwischen Bäumen und den Ableitungen einer kontextfreien Grammatik hergestellt werden.

Definition 4.1.4 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik, B ein geordneter Baum, $V = N \cup T$ und $\mu : B \to V_\epsilon$. (B, μ) heißt ein* Ableitungsbaum $\iff$

$$\forall b \in B : \varphi b = b_1 ... b_m \wedge m \geq 1 \Rightarrow (\mu(b), \mu(b_1 b_2 ... b_m)) \in P.$$

Beispiel: $G = (N, T, P, \sigma)$ mit $T = \{a, b, ..., z\}$, $N = \{\sigma, \alpha\}$ und $P = \{\sigma \to \alpha | \sigma * \sigma | \sigma + \sigma, \alpha \to a|b|...|z\}$.



Ableitungsbaum von G: Jeder innere Knoten (kein Blatt) ist so markiert, daß er mit der linken Seite einer Produktion und seine Nachfolger mit der rechten Seite markiert sind wie z.B. für $b \in B$:

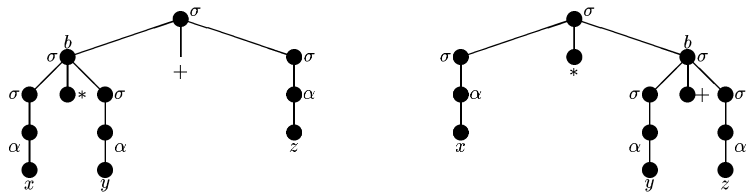
$$\varphi(b) = b_1 b_2 b_3 \Rightarrow \mu(b) \underset{P}{\to} \mu(b_1)\mu(b_2)\mu(b_3).$$

`Ende Beispiel`

Wenn (B, μ) ein Ableitungsbaum ist für die kontextfreie Grammatik G, und wenn gilt $\mu(b_0) = \sigma$, dann ist $\mu(\partial B) \in S(G)$, also eine Satzform. Ist außerdem $[\mu(\partial B)] \subseteq T_\epsilon$, dann ist das Wort $\mu(\partial B)$ aus $L(G)$. Ist $\mu(b_0) = \alpha \in N$ und $[\mu(\partial B)] \subseteq T_\epsilon$, dann ist $\mu(\partial B) \in L_\alpha$. Da jeder Schnitt von B selbst Rand eines Baumes ist, der in B enthalten ist, und die gleiche Wurzel hat wie B, gilt außerdem: Wenn $\mu(b_0) = \sigma$, dann ist für jeden Schnitt $S : \mu(S)$ eine Satzform.

Wenn gilt $\mu(b_0) = \sigma$ und $\mu(\partial B) = u \in T^*$, dann heißt (B, μ) ein *Ableitungsbaum* für u.

Beispiel:(Fortsetzung) G wie oben



Man erkennt außerdem, daß α hier eigentlich überflüssig ist. Man kann die Produktion $\sigma \to \alpha$ ersetzen durch: $\sigma \to a|b|...|z$ `Ende Beispiel`

Da man die Semantik aus der Struktur der Ableitungsbäume bestimmt, ist es wichtig bei der Wahl der Grammatik darauf zu achten, daß es zu jedem Wort $u \in L(G)$ nur einen Ableitungsbaum gibt.

Definition 4.1.5 *Eine kontextfreie Grammatik heißt* eindeutig*, wenn es zu jedem Wort der erzeugten Sprache genau einen Ableitungsbaum gibt, sonst heißt sie* mehrdeutig.

Leider ist diese Eigenschaft nicht entscheidbar, d.h. es existiert kein Algorithmus, der für jede kontextfreie Grammatik berechnet (entscheidet), ob sie eindeutig ist oder nicht.

Wenn $G = (N, T, P, \sigma)$ eine eindeutige kontextfreie Grammatik ist, dann gibt es zu jedem $u \in L(G)$ nur einen Ableitungsbaum, aber es kann viele mögliche Ableitungen

$$\sigma \underset{P}{\to} v_1 \underset{P}{\to} v_2 \underset{P}{\to} ... \underset{P}{\to} v_m = u$$

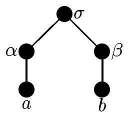
geben, wie z.B.:

$$\sigma \to \alpha\beta \to a\beta \to ab$$

und

$$\sigma \to \alpha\beta \to \alpha b \to ab.$$

Welches nichtterminale Symbol in einer Satzform als nächstes ersetzt wird ist willkürlich. Der Ableitungsbaum vermeidet diese Willkür, da dort quasi „parallel“ ersetzt wird:



Man kann aber auch die Ableitungen so normieren, daß die Ableitungsfolge (für eine eindeutige Grammatik) eindeutig wird. Die folgende Regel kann dazu dienen:

Wenn in einer Satzform mehrere nichtterminale Zeichen ersetzt werden können, dann wird stets dasjenige zuerst ersetzt, das am weitesten links (rechts) steht.

Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik, dann definiert man zu $\to_P$ die Teilrelation $\to_L \subseteq \to_P$ der Links-Ableitungsschritte wie folgt:

$$u \underset{L}{\to} v \iff u = w\alpha u'', v = wu'u'' \text{ und } \alpha \underset{P}{\to} u' \text{ und } w \in T^*.$$

Analog definiert man Rechtsableitungsschritte. Zu jedem Wort $u \in L(G)$ gibt es eine Linksableitung, die nur aus Links-Ableitungsschritten besteht:

$$\forall u \in L(G) : \sigma \overset{*}{\underset{L}{\to}} u \quad (\sigma \overset{*}{\underset{R}{\to}} u).$$

Linksableitungen und Ableitungsbäume entsprechen sich:

Lemma 4.1.6 *Zu jedem Ableitungsbaum (B, μ) einer kontextfreien Grammatik gibt es genau eine Linksableitung (Rechtsableitung) für $\mu(\partial B)$.*

Beweis: Übung □

Beispiel:(Fortsetzung) Zu den beiden Ableitungsbäumen gehören die folgenden Linksableitungen:
$B : \sigma \to \sigma + \sigma \to \sigma * \sigma + \sigma \to \alpha * \sigma + \sigma \to x * \sigma + \sigma \to x * \alpha + \sigma \to x * y + \sigma \to x * y + \alpha \to x * y + z$
$B' : \sigma \to \sigma * \sigma \to \alpha * \sigma \to x * \sigma \to x * \sigma + \sigma \to x * \alpha + \sigma \to x * y + \sigma \to x * y + \alpha \to x * y + z$
`Ende Beispiel`

Es gibt zwei wesentliche (aber das sind nicht die einzigen) Strategien, um zu einem gegebenen Wort $u \in T^*$ (oder besser: $u \in L(G)$) einen Ableitungsbaum zu konstruieren.

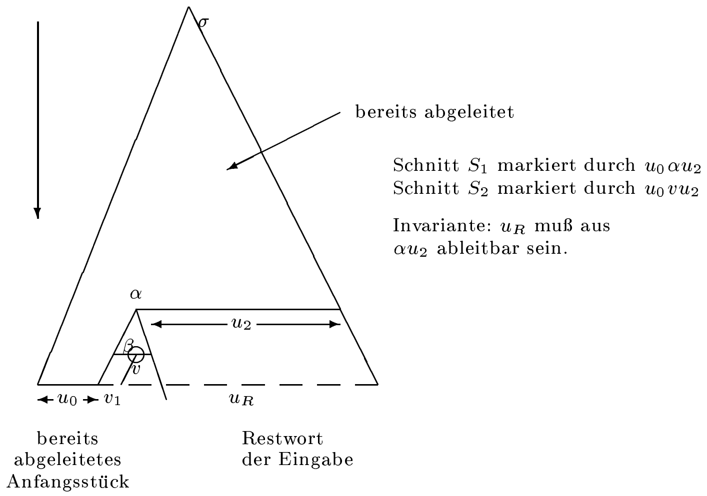
- *Top-Down-Analyse* Ausgehend von der Wurzel des Ableitungsbaumes versucht man den ersten Ableitungsschritt zu raten (nichtdeterministisch). Wenn v_1 terminale Zeichen enthält, müssen diese zum gegebenen Wort u (auf dem Rand) passen.

  Man verfährt so weiter bis man u erzeugt hat.

- *Bottom-Up-Analyse* Ausgehend vom Rand des Baumes wird versucht, die rechte Seite der zuletzt angewendeten Produktion einer Rechtsableitung zu erkennen. Sei das die Produktion $\alpha \to v$ mit $v \in T^*$ und $u = vu'$. Dann wird v zu α reduziert und man erhält die Satzform $\alpha u'$. Mit dieser verfährt man ebenfalls nichtdeterministisch weiter, bis man u auf die Satzform σ reduziert hat.

  Man erhält so eine Rechtsableitung in umgekehrter Reihenfolge.

Für beide Verfahren kann man eine Folge von Schnitten im Ableitungsbaum betrachten, die bei der Analyse durchlaufen wird. Jedem dieser Schnitte entspricht dann eine Situation eines (nichtdeterministischen) Kellerautomaten.

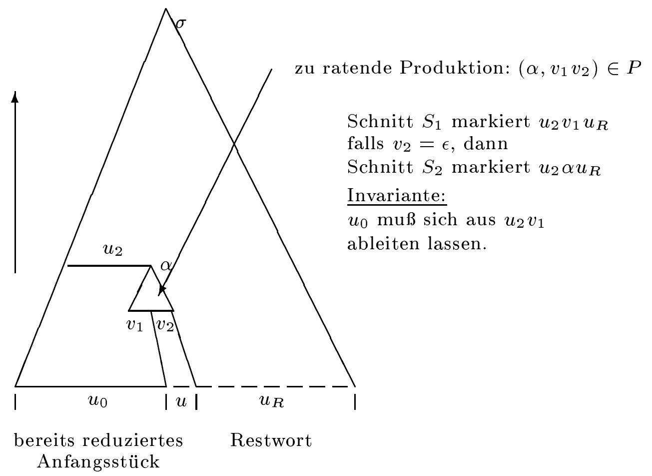


In der Situation, die durch S_1 gegeben ist, wird der L-Ableitungsschritt $\alpha \to v \in P$ „geraten" (nichtdeterministisch). Ist $v = v_1\beta v_2$ mit $v_1 \in T^*$, dann muß gelten: $u_R = v_1 u'_R$, d.h. u_R beginnt mit v_1. u'_R ist dann das neue Restwort und man erhält den neuen Schnitt S_3, der mit $u_0 v_1 \beta v_2 u_2$ markiert ist. Dieser Prozeß wird solange fortgesetzt, bis gilt: $u_R = \epsilon$. Der Schnitt S_1 entspricht der folgenden Situation des Kellerautomaten:

Restwort Eingabe:	u_R
Kellerinhalt:	αu_2
	↑ „oben"

α wird dann auf dem Keller durch v ersetzt.

4.1.1 Bottom-up Analyse



In der Situation, die durch S_1 gegeben ist, muß zunächst aus einem Anfangsstück von u_R das Teilwort v_2 reduziert werden. Dann kann $v = v_1 v_2$ auf α reduziert werden, so daß die Markierung des Schnittes S_2 dann lautet: $u_2 \alpha u'_R$. Dabei ist $u_R = u u'_R$ und es gilt $v_2 \rightarrow^* u$.

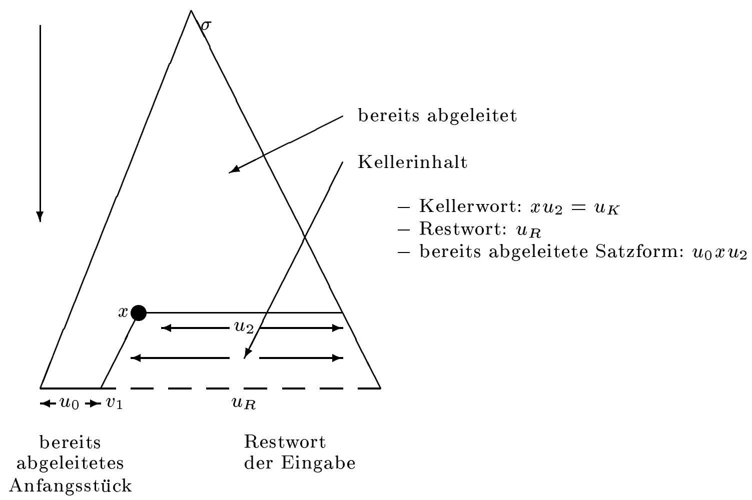
Der Prozeß wird solange fortgesetzt, bis der minimale Schnitt mit Markierung σ erreicht ist. Für den Kellerautomaten gilt:

Restwort Eingabe:	u_R
Kellerinhalt:	αu_2
	↑ „oben“

Ist das Kellerwort $u_2 v_1 v_2$, dann kann $v_1 v_2$ durch α ersetzt werden zu $u_2 \alpha$.

4.1.2 Top-down Analyse

Eine *Situation* ist gegeben durch einen bestimmten *Schnitt* im *Ableitungsbaum*



Der Schnitt S hat die Markierung $u_0xu_2 = u_0u_K$. Die Situation für einen Kellerautomaten ist das Paar $(u_K, u_R) \in V^* \times T^*$. In jeder Situation muß der Automat *entscheiden*, wie der nächste Schnitt im Ableitungsbaum erreicht wird:

- Wenn das oberste Kellersymbol $x = t \in T$ ist, dann muß auch u_R mit t beginnen: $u_R = tu'_R$. In diesem Fall wird x vom Keller genommen: $u'_K = u_2$ und u'_R ist das neue Restwort.
- Wenn das oberste Kellersymbol $x = \alpha \in N$ ist, dann wählt man (nichtdeterministisch) eine Produktion $(\alpha, v) \in P$ und ersetzt α auf dem Keller durch v: $u'_K = vu_2$, $u'_R = u_R$. So wird ein neuer Linksableitungsschritt gewonnen:

$$\sigma \overset{*}{\underset{L}{\rightarrow}} u_0xu_2 = u_0\alpha u_2 \underset{L}{\rightarrow} u_0vu_2.$$

Die Situation bestimmt also die nächste Aktion.

Man betrachtet daher die Menge aller Situationen, in denen eine bestimmte Produktion $(\alpha, v) \in P$ angewendet werden kann.

Definition 4.1.7 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik und $(\alpha, v) \in P$. Dann heißt*

$$A(\alpha, v) = \left\{(\alpha u, u_R) \in V^+ \times T^* | \exists u_0 \in T^* : \sigma \overset{*}{\underset{L}{\rightarrow}} u_0\alpha u \wedge vu \overset{*}{\underset{L}{\rightarrow}} u_R\right\}$$

die Ableitungsklasse *von (α, v).*

$(\alpha u, u_R)$ ist in $A(\alpha, v)$ also genau dann, wenn es eine Satzform $u_0\alpha u$ gibt, die durch eine Linksableitung aus σ hergeleitet werden kann, und wenn aus vu das Restwort u_R abgeleitet werden kann. Die Ableitungsklassen enthalten nur Situationen, bei denen ein nichtterminales Zeichen oben auf dem Keller steht. Ist die Ableitungsklasse zu einer Situation eindeutig bestimmt, dann ist auch der nächste Linksableitungsschritt eindeutig bestimmt. Dazu aber noch eine technische Definition:

Definition 4.1.8 *Eine kontextfreie Grammatik $G = (N, T, P, \sigma)$ heißt* reduziert $\iff$

$$\forall \alpha \in N : L_\alpha \neq \emptyset \wedge \exists u_1, u_2 \in V^* : \sigma \overset{*}{\rightarrow} u_1\alpha u_2.$$

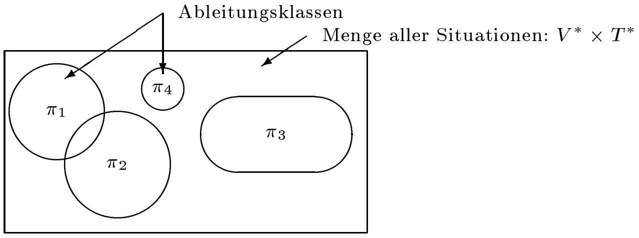
In einer reduzierten Grammatik produziert jedes $\alpha \in N$ terminale Wörter und kommt in mindestens einer Satzform vor.

Es ist entscheidbar, ob eine kontextfreie Grammatik reduziert ist und man kann zu jeder kontextfreien Grammatik eine äquivalente reduzierte Grammatik effektiv konstruieren (Algorithmus: Siehe Übungen).

Satz 4.1.9 *Eine reduzierte kontextfreie Grammatik ist genau dann eindeutig, wenn ihre Ableitungsklassen paarweise disjunkt sind.*

Beweis: Ohne Beweis, siehe Formale Sprachen. □

Man kann die Bedeutung der Ableitungsklassen durch eine geometrische Analogie (Mengendiagramm) veranschaulichen:



$P = \{\pi_1, ..., \pi_4\}$

Es gibt hier Situationen, die sowohl in $A(\pi_1)$ als auch in $A(\pi_2)$ liegen. π_1 und π_2 haben dann die gleiche linke Seite $\alpha \in N$ aber unterschiedliche rechte Seiten. In einer solchen Situation ist der nächste Ableitungsschritt nicht eindeutig festgelegt. Sind die Ableitungsklassen dagegen paarweise disjunkt, dann ist der nächste L-Ableitungsschritt stets durch die Ableitungsklasse der Situation eindeutig bestimmt.

Ein Algorithmus, der zu jeder Situation $(\alpha u, u_R)$ den nächsten Ableitungsschritt berechnet kann für die Top-down Analyse verwendet werden. Wenn man in $(\alpha u, u_R)$ die Klammern wegläßt und das trennende Komma als neues Symbol $\iota \in T' = T \cup \{\iota\}$ betrachtet, dann bilden die Situationen und damit die Ableitungsklassen Sprachen. Ein Automat, der die jeweilige Ableitungsklasse erkennt, wäre also ein geeignetes Mittel für die Top-down Analyse. Leider sind die Ableitungsklassen i.a. selbst wieder kontextfreie Sprachen und damit ist der Aufwand für ihre Erkennung zu groß.

Ein Ausweg ist die Definition geeigneter *Obermengen* (*Oberklassen*) zu den Ableitungsklassen, die

- paarweise disjunkt,
- leicht analysierbar (z.B. regulär)

sind.

Ein solches System von Obermengen heißt *akzeptabel*, wenn gilt:

1. $\forall \pi_1, \pi_2 \in P : \pi_1 \neq \pi_2 \Rightarrow \Omega(\pi_1) \cap \Omega(\pi_2) = \emptyset$,
2. es gibt einen effizienten Algorithmus, der zu jeder Situation $(\alpha u, u_R)$ dasjenige $\pi \in P$ berechnet mit $(\alpha u, u_R) \in \Omega(\pi)$.

Wenn die Obermengen $\Omega(\pi)$ $(\pi \in P)$ insbesondere reguläre Mengen sind, dann kann man für die Klassifizierung 2. einen endlichen Automaten konstruieren, der π jeweils durch einen Endzustand bestimmt.

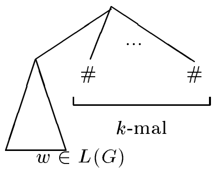
Eine wesentliche Idee, solche Obermengen zu finden, besteht darin, die Situationen nur nach den ersten k Zeichen von u_R $(k \in \mathbb{N})$ zu unterscheiden:

$$\Omega^{(k)}(\alpha, v) = \{(\alpha u, z\tilde{u}) |\ |z| = k \wedge \exists w \in T^* : (\alpha u, zw) \in A(\alpha, v)\}.$$

Alle Situationen $(\alpha u, u_R)$, bei denen u_R das feste Anfangsstück $z \in T^k$ hat, fallen somit in die gleiche Klasse. Zur Beurteilung der Situationen ist daher von u_R nur das Anfangsstück der Länge k wesentlich. Damit dieses Restwort u_R auch stets ein Anfangsstück der festen Länge k hat, wird die Grammatik zunächst technisch modifiziert. Sei $\# \notin T$, dann sei für $G = (N, T, P, \sigma)$ die *k-Erweiterung* definiert durch:

$$G^{(k)} = (\hat{N}, \hat{T}, \hat{P}, \sigma_0) \text{ mit } \sigma_0 \notin N, \hat{N} = N + \{\sigma_0\}, \hat{T} = T + \{\#\}, \hat{P} = P + \left\{(\sigma_0, \sigma\#^k)\right\}.$$

Dann gilt: $L\left(G^{(k)}\right) = L(G)\#^k$, d.h. jedes Wort aus $L(G^{(k)})$ ist ein Wort aus $L(G)$, das noch einen „Schweif“ $\#^k$ angehängt hat:



$w\#^k \in L\left(G^{(k)}\right)$

Für die so erweiterte Grammatik definiert man:

$$\Omega^{(k)}(\alpha, v) = \left\{\alpha uz | z \in \hat{T}^k \wedge \exists w \in \hat{T}^* : (\alpha u, zw) \in \hat{A}(\alpha, v)\right\} .$$

Es ist jetzt wegen $|z| = k$ die Zerlegung von αuz eindeutig bestimmt, und daher kann man $\alpha uz \in V^*$ annehmen. $\Omega^{(k)}(\alpha, v) \subseteq V^*$ heißt jetzt auch die *LL(k)-Oberklasse* für $(\alpha, v) \in P$ und man nennt αuz auch eine *LL(k)-Situation*.

Definition 4.1.10 *Eine reduzierte kontextfreie Grammatik heißt* LL(k)-Grammatik *genau dann, wenn ihre LL(k)-Oberklassen paarweise disjunkt sind.*

Eine LL(k)-Grammatik ist natürlich immer eindeutig. Die Bezeichnung LL(k) setzt sich folgendermaßen zusammen: L steht für Eingabe von links nach rechts abarbeiten. Das zweite L steht für die Ausgabe einer Linksableitung (bei der Analyse) und k ist der Parameter für die Ableitungsklassen. Analog steht später LR(k) für eine Bottom-up Analyse, die eine Folge von Rechtsableitungsschritten erzeugt.

Definition 4.1.11 *Eine Sprache $L \subseteq T^*$ heißt* LL(k) *genau dann, wenn es eine LL(k)-Grammatik gibt, die L erzeugt.*

Das folgende einfache Beispiel zeigt, daß der Durchschnittstest für Ableitungsklassen und auch LL(k)-Oberklassen recht mühsam ist.

Beispiel: G sei durch die folgenden Produktionen definiert:

$$\begin{array}{rcl} \sigma & \to & a\alpha \\ \alpha & \to & a\alpha b \,|\, b \end{array}$$

Man liest für die Satzformmengen direkt die folgenden Gleichungen ab:

$$\begin{array}{rcl} S_\sigma & = & aS_\alpha + \sigma \\ S_\alpha & = & aS_\alpha b + b + \alpha \\ & \vdots & \\ & = & a^n S_\alpha b^n + b + \alpha \quad ; (n \geq 0) \\ & = & \{a^n b b^n, a^n \alpha b^n | n \geq 0\} \\ S_\sigma & = & \{a^{n+1} b^{n+1}, a^{n+1} \alpha b^n | n \geq 0\} + \sigma \end{array}$$

Damit kann man leicht die Ableitungsklassen bestimmen:

$$\begin{array}{rcl} A(\sigma, a\alpha) & = & \left\{(\sigma u, w) | u_0 \sigma u \in S_\sigma \wedge a\alpha u \overset{*}{\underset{L}{\to}} w\right\} \\ & = & \{(\sigma, w) | w \in aL_\alpha\} \end{array}$$

$$\begin{aligned}
&= \{(\sigma, a^n b^n) | n \geq 1\} \\
A(\alpha, a\alpha b) &= \left\{(\alpha u, w) | u_0 \alpha u \in S_\sigma \wedge (a\alpha b)u \overset{*}{\underset{L}{\rightarrow}} w\right\} \\
&= \left\{(\alpha b^n, w) | a\alpha b^{n+1} \overset{*}{\rightarrow} w\right\} \\
&= \left\{(\alpha b^n, w) | w \in aL_\alpha b^{n+1}\right\} \\
&= \left\{(\alpha b^n, a^{m+1} b^{m+1} b^{n+1}) | m, n \geq 0\right\} \\
&= \left\{(\alpha b^n, a^m b^m b^{n+1}) | n \geq 0, m \geq 1\right\} \\
A(\alpha, b) &= \left\{(\alpha u, w) | u_0 \alpha u \in S_\sigma \wedge bu \overset{*}{\underset{L}{\rightarrow}} w\right\} \\
&= \left\{(\alpha b^n, w) | bb^n \overset{*}{\rightarrow} w\right\} \\
&= \left\{(\alpha b^n, b^{n+1}) | n \geq 0\right\}
\end{aligned}$$

Man sieht leicht, daß die Ableitungsklassen paarweise disjunkt sind, und damit ist die Grammatik eindeutig.

$$\begin{aligned}
\Omega^{(0)}(\sigma, a\alpha) &= \{\sigma u | (\sigma u, w) \in A(\sigma, a\alpha)\} \\
&= \{\sigma\} \\
\Omega^{(0)}(\alpha, a\alpha b) &= \{\alpha u | (\alpha u, w) \in A(\alpha, a\alpha b)\} \\
&= \{\alpha b^n | n \geq 0\} \\
\Omega^{(0)}(\alpha, b) &= \{\alpha u | (\alpha u, w) \in A(\alpha, b)\} \\
&= \{\alpha b^n | n \geq 0\}
\end{aligned}$$

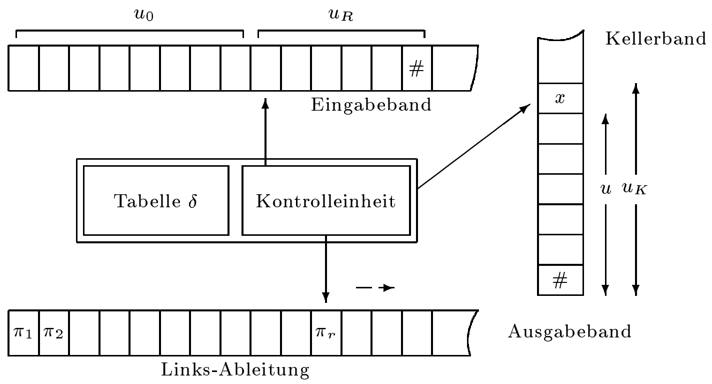
Wegen $\Omega^{(0)}(\alpha, a\alpha b) = \Omega^{(0)}(\alpha, b)$ ist G keine LL(0)-Grammatik.

$$\begin{aligned}
\Omega^{(1)}(\sigma, a\alpha) &= \{\sigma ut | (\sigma u, tw) \in A(\sigma, a\alpha)\} \\
&= \{\sigma a\} \\
\Omega^{(1)}(\alpha, a\alpha b) &= \{\alpha ut | (\alpha u, tw) \in A(\alpha, a\alpha b)\} \\
&= \{\alpha b^n a | n \geq 0\} \\
\Omega^{(1)}(\alpha, b) &= \{\alpha ut | (\alpha u, tw) \in A(\alpha, b)\} \\
&= \{\alpha b^n b | n \geq 0\}
\end{aligned}$$

Somit ist G eine LL(1)-Grammatik. `Ende Beispiel`

Eine LL(k)-Situation $\alpha uz \in \Omega^{(k)}(\pi)$ legt eindeutig den nächsten Links-Ableitungsschritt $\pi \in P$ fest. Ein LL(k)-Parser muß also die LL(k)-Situationen der jeweiligen LL(k)-Obermenge zuordnen können (Klassifizierung).

Wir wollen hier nur den Fall $k = 1$ betrachten, der für die Praxis wesentlich ist. Eine Situation hat die Form αut mit $t \in T$, und man kann zeigen, daß die LL(1)-Oberklasse nur von α und t abhängt. Der Parser kann daher durch eine Tabelle $\delta : N \times T \rightarrow P$ gesteuert werden:



Man betrachtet $x \in V$ auf dem Kellerband und t auf dem Eingabeband:

Falls $x = t \in T$ dann wird x gegen t „gekürzt“: $u'_K = u$ und $u_R = tu'_R$ definiert u'_R.

Falls $x = \#$ (leerer Keller): Erfolgreich beendet.

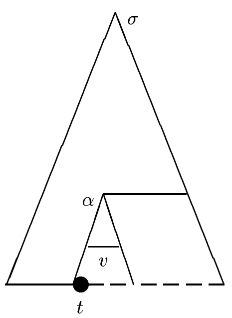
Falls $x = \alpha \in N$, dann wird in der Tabelle δ unter α und t die Produktion $\delta(\alpha, t) = (\alpha, v) = \pi$ gefunden, die den nächsten L-Ableitungsschritt ergibt. Man ersetzt $x = \alpha$ durch $v : u'_K = vu$ und schreibt π auf das Ausgabeband. Ist $\delta(\alpha, t) = \emptyset$, dann liegt ein Fehler vor.

Es muß dann nur noch die Tabelle δ konstruiert werden. Um zu zeigen, wie diese Tabelle konstruiert wird, sind wieder einige Begriffe zu definieren, die dann aber auch ein sehr wirkungsvolles Kriterium liefern für die LL(1)-Eigenschaft von Grammatiken.

Definition 4.1.12 *Sei $G = (N, T, P, \sigma)$ eine reduzierte kontextfreie Grammatik. Für $u \in V^*$ und $\alpha \in N$ definiert man die Mengen:* First*(u),* Follow*(α) $\subseteq T_\epsilon$ durch:*

1. $t \in \text{First}(u) \iff \exists u' \in V^* : u \rightarrow^* tu'$
   $\epsilon \in \text{First}(u) \iff u \rightarrow^* \epsilon,$
2. $t \in \text{Follow}(\alpha) \iff \exists u_1, u_2 \in V^* : u_1 \alpha t u_2 \in S(G).$

Bedeutung dieser Mengen:



Wenn man aus α und t die nächste Produktion $(\alpha, v) \in P$ eindeutig bestimmen will, dann muß gelten:

1. $(\alpha, v_1) \neq (\alpha, v_2) \Rightarrow \text{First}(v_1) \cap \text{First}(v_2) = \emptyset$. t bestimmt dann die α-Produktion eindeutig.

Es ist aber auch möglich, daß t gar nicht durch α produziert wurde und dennoch eine α-Produktion angewendet werden muß, nämlich dann, wenn α zu ϵ abgeleitet werden kann. Dann ist auch $\epsilon \in$ First(α). Die Ableitung

$$\sigma \stackrel{*}{\to} u_0 \alpha t u_2 \stackrel{*}{\to} u_0 t u_2$$

hat dann auch eine α-Produktion verwendet. Um auch in diesem Fall Eindeutigkeit zu erreichen fordert man also

2. $\epsilon \in$ First(α) $\Rightarrow$ First(α)$\cap$ Follow(α) $= \emptyset$

Satz 4.1.13 *Eine reduzierte kontextfreie Grammatik* $G = (N, T, P, \sigma)$ *ist LL(1)* $\iff \forall \alpha \in N :$ $(\alpha, v_1), (\alpha, v_2) \in P \wedge v_1 \neq v_2 \Rightarrow$

1. First(v_1)$\cap$ First(v_2) $= \emptyset$*,*

2. $\epsilon \in$ First(α) $\Rightarrow$ First(α)$\cap$ Follow(α) $= \emptyset$*.*

Beweis: Ohne Beweis. □

Diese endlichen Mengen First(v) und Follow(α) spielen also eine wichtige Rolle bei der Beurteilung einer Grammatik. Es soll noch skizziert werden, wie man die Mengen First(v) berechnet:

1. Sei $x \in N \cup T$ (einzelnes Symbol, $|v| = 1$).
   - $x \in T$: First(x) $= \{x\}$,
   - $x \in N$ und $x \to \epsilon \in P$: $\epsilon \in$ First(x),
   - $x = \alpha \in N$ und $\alpha \to v_1 ... v_n \in P$ ($v_i \in V$):
     $t \in$ First(α) falls: $t \in$ First(v_i) $\wedge \forall 1 \leq j < i : \epsilon \in$ First(v_j)
     $\epsilon \in$ First(α) falls: $\forall 1 \leq j \leq n : \epsilon \in$ First(v_j)

   Nach diesen Regeln wird First(x) solange erweitert, bis kein neues Element mehr hinzu kommt.
2. Sei $v = v_1 ... v_n \in V^*$:
   - $v = \epsilon$: First(ϵ) $= \{\epsilon\}$,
   - $t \in$ First(v) falls: $t \in$ First(v_i) $\wedge \forall 1 \leq j < i : \epsilon \in$ First(v_j),
   - $\epsilon \in$ First(v) falls: $\forall 1 \leq j \leq n : \epsilon \in$ First(v_j).

Damit ist das Verfahren skizziert. Da die First-Mengen für eine gegebene Grammatik $G = (N, T, P, \sigma)$ nur für alle $\alpha \in N$ und alle $v \in V^*$ mit $(\alpha, v) \in P$ benötigt werden, sind das nur endlich viele Mengen.

Die Follow-Mengen werden wie folgt berechnet (Man beachte, daß # hier ein spezielles terminales Symbol ist):

- $\# \in$ Follow(σ).
- Sei $\alpha \to u\beta v \in P$ dann: (First(v) $- \epsilon$) $\subseteq$ Follow(β).
- Sei $\alpha \to u\beta v \in P$ und $\epsilon \in$ First(v) dann Follow(α) $\subseteq$ Follow(β).

Beispiel: (Grammatik für vollständig geklammerte Ausdrücke) G ist gegeben durch die Produktionen:

$$\begin{array}{rcl} E & = & T + E|T \quad (\sigma = E) \\ T & = & F * T|F \\ F & = & \text{id} \quad (\text{id terminal}) \end{array}$$

(E: Ausdruck, Expression; T: Term; F: Faktor)
$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{\text{id}\}$
$\text{Follow}(E)$: $\# \in \text{Follow}(E)$
$E \to T + E \Rightarrow \text{Follow}(E) \subseteq \text{Follow}(E) \Rightarrow \text{Follow}(E) = \{\#\}$
$\text{Follow}(T)$: $E \to T + E \Rightarrow \text{First}(+E) \subseteq \text{Follow}(T)$
$E \to T \Rightarrow \text{Follow}(E) \subseteq \text{Follow}(T) \Rightarrow \text{Follow}(T) = \{+, \#\}$ $\text{Follow}(F)$: $T \to F * T \Rightarrow \text{First}(*T) \subseteq \text{Follow}(F)$
$T \to F \Rightarrow \text{Follow}(T) \subseteq \text{Follow}(F) \Rightarrow \text{Follow}(F) = \{+, *, \#\}$
Wenn man nun den obigen Satz anwendet, dann betrachtet man:

$$E \to E + T \text{ und } E \to T \quad (\in P).$$

Es ist aber:
$\text{First}(E + T) = \text{First}(E) = \{\text{id}\}$
$\text{First}(T) = \{\text{id}\}$
und daher ist G keine LL(1)-Grammatik. `Ende Beispiel`

First und Follow werden auch verwendet, um die Tabelle für den LL(1)-Parser zu konstruieren.

Gegeben: $G = (N, T, P, \sigma)$ red. c.f.-Grammatik

Gesucht: Parsertabelle $\delta : N \times T \to \mathcal{P}(P)$

<u>begin</u>
  <u>for</u> $(\alpha, v) \in P$ <u>do</u> <u>begin</u>
    <u>for</u> $t \in \text{First}(v)$ <u>do</u>
     $\delta(\alpha, t) := \delta(\alpha, t) + (\alpha, v)$;
    <u>if</u> $\epsilon \in \text{First}(v)$ <u>then</u>
      <u>for</u> $t \in \text{Follow}(\alpha)$ <u>do</u> $\delta(\alpha, t) := \delta(\alpha, t) + (\alpha, v)$;
  <u>end</u>
<u>end</u> .

Diese Tabelle kann immer konstruiert werden, auch wenn G nicht LL(1) ist. G ist nicht LL(1), wenn es ein Paar $(\alpha, t) \in N \times T$ gibt mit $|\delta(\alpha, t)| > 1$.

Wenn $\delta(\alpha, t) = \emptyset$, dann bedeutet das für den Parser, daß in dieser Situation keine Produktion anwendbar ist, d.h. es liegt eine fehlerhafte Eingabe vor.

Es gibt verschiedene Strategien dafür, wie der Parser auf Fehler reagieren kann, ohne gleich abzubrechen. Man definiert dafür sogenannte synchronisierende Token: syn $\subseteq T$. Wenn ein Fehler auftritt, dann wird in der Eingabe bis zu einem Zeichen aus syn weitergegangen. Man nennt dieses Verfahren auch die Panik-Methode. Es handelt sich um ein heuristisches Verfahren und stellt kein Optimum der Fehlerbehandlung dar. Die folgenden Strategien sollen hier nur als Hinweis dienen:

Panik-Reaktion auf Fehler:

- Oberstes Kellersymbol $x = \alpha \in N$ und $\delta(\alpha, t) = \emptyset$:
  - Eingabe weiterrücken bis ein Zeichen aus Follow(α) gefunden wird. Dann α vom Keller entfernen.
  - Oder α auf dem Keller belassen und die Eingabe weiterrücken, bis ein Zeichen aus First(α) gefunden wird.
  - Falls $\epsilon \in$ First(α) kann man α einfach vom Keller entfernen.
- Oberstes Kellersymbol ist $x = t' \in T$ und $t' \neq t$: Man entfernt t' vom Keller und rückt mit der Eingabe um eine Position nach rechts vor. t wird in der Eingabe als unzulässig markiert.

Ein elegantes Verfahren für die LL(1)-Syntaxanalyse heißt *Rekursiver Abstieg* (*recursive descent*). Das Verfahren verwendet den Kellerspeicher nicht explizit, sondern implizit durch Rekursion. Dieses Verfahren wird auch für den PL0-Compiler im Buch von Wirth angewendet.

Rekursiver Abstieg: Grammatik $G = (N, T, P, \sigma)$; LL(1).

Für jedes $\alpha \in N$ schreibt man eine Funktion (Prozedur) F_α. Diese Funktion bearbeitet gerade Konstrukte (Teilwörter der Eingabe) aus L_α. Sie wird direkt aus dem Syntaxdiagramm für α programmiert:
F_α bearbeitet die α-Produktionen,

$$\alpha \to v_1|v_2|...|v_n.$$

- Die Auswahl der Alternative v_i wird durch den nächsten Token $t \in T$ der Eingabe eindeutig bestimmt. Man erhält eine case-Anweisung für diese Alternativen.
- Sei $v_i = x_1...x_r \in V^*$ die gegebene Alternative:
  1. Ist $x_1 \in T$ dann muß $x_1 = t$ sein. t ist dabei das aktuelle Zeichen der Eingabe. Wenn $x = t$ gilt, wird die Position weitergerückt. Danach wird $x_2...x_r$ weiter bearbeitet.
  2. Ist $x_1 \in N$ und $x_1 = \beta$, dann wird die Funktion F_β aufgerufen.

Das Verfahren ist recht übersichtlich zu programmieren, da die Programmstruktur an der syntaktischen Struktur der Grammatik orientiert ist. Allerdings sind Grammatiken für praktische Analyseaufgaben oft nicht LL(1)-Grammatiken. Um das Verfahren dennoch anwenden zu können, kann man versuchen, die Grammatik G in eine äquivalente Grammatik G' zu transformieren, so daß G' dann eine LL(1)-Grammatik ist.

Definition 4.1.14 *Eine kontextfreie Grammatik $G = (N, T, P, \sigma)$ heißt* links-rekursiv

$$\iff \exists \alpha \in N : \alpha \xrightarrow{+} \alpha u \quad (u \in V^*).$$

Eine links-rekursive Grammatik ist niemals LL(1) (Übung). Aber in der Praxis treten oft links-rekursive Regeln der Form $(\alpha, \alpha u) \in P$ auf.

Beispiel:(Arithmetische Ausdrücke) G sei gegeben durch die Produktionen

$E \to E + T|T$

$T \to T * F|F$

$F \to (E)|\text{id}.$

Diese Grammatik ist offensichtlich links-rekursiv und damit nicht LL(1). `Ende Beispiel`

Links-Rekursion läßt sich durch eine (systematische) Grammatik-Transformation vermeiden. Das allgemeine Verfahren findet man z.B. in [1]. Es soll hier nur gezeigt werden, wie man direkte Links-Rekursion (also links-rekursive Regeln) transformiert.

Beseitigung links-rekursiver Regeln (Produktionen):
Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik und $\alpha \in N$, so daß zu α linksrekursive Regeln existieren:

$$\alpha \to \alpha u_1 | \alpha u_2 | ... | \alpha u_m | v_1 | ... | v_k$$

mit $u_i, v_j \in V^*$ $(1 \leq i \leq m)$ $(1 \leq j \leq k)$ und kein v_j beginnt mit α.

Man führt ein neues nichtterminales Symbol $\overline{\alpha}$ ein und die neuen Produktionen:

$$\begin{array}{lcl} \alpha & \to & v_1\overline{\alpha}|...|v_k\overline{\alpha} \\ \overline{\alpha} & \to & u_1\overline{\alpha}|...|u_m\overline{\alpha}|\epsilon \end{array}$$

Mit diesen neuen Produktionen kann man die alten α-Produktionen simulieren:

Ableitung in G:

$$\alpha \to \alpha u_{i_1} \to \alpha u_{i_2} u_{i_1} \to ... \to \alpha u_{i_n} ... u_{i_1} \to v_j u_{i_n} ... u_{i_1}$$

Das gleiche Endresultat erreicht man mit der folgenden Ableitung in G':

$$\alpha \to v_j\overline{\alpha} \to v_j u_{i_n}\overline{\alpha} \to ... \to v_j u_{i_n} ... u_{i_1}\overline{\alpha} \to v_j u_{i_n} ... u_{i_1}.$$

Beispiel:(Fortsetzung) Die Grammatik G aus dem vorhergehenden Beispiel ist links-rekursiv. Durch die Anwendung der obigen Transformation erhält man:

$E \to E + T|T$ wird ersetzt durch:

$E \to T\overline{E}$
$\overline{E} \to +T\overline{E}|\epsilon$

$T \to T * F|F$ wird ersetzt durch:

$T \to F\overline{T}$
$\overline{T} \to *F\overline{T}|\epsilon$

damit erhält man eine Grammatik G' ohne Links-Rekursion, die aber äquivalent zu G ist:

$E \to T\overline{E}$

$\overline{E} \to +T\overline{E}|\epsilon$

$T \to F\overline{T}$

$\overline{T} \to *F\overline{T}|\epsilon$

$F \to (E)|\text{id}$

Wir untersuchen G' auf die Eigenschaft LL(1):

$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{(, \text{id}\}$

$\text{First}(\overline{E}) = \{+, \epsilon\}$

$\text{First}(\overline{T}) = \{*, \epsilon\}$

$\text{Follow}(E) = \text{Follow}(\overline{E}) = \{), \#\}$

$\text{Follow}(T) = \text{Follow}(\overline{T}) = \{+,), \#\}$

$\text{Follow}(F) = \{+, *,), \#\}$

Die Produktionen für $\overline{E}$, $\overline{T}$ und F haben mehr als eine rechte Seite.

Kriterium 1: $\text{First}(+T\overline{E}) = \{+\}$, $\text{First}(\epsilon) = \{\epsilon\}$, also $\text{First}(+T\overline{E}) \cap \text{First}(\epsilon) = \emptyset$
Analog für $\overline{T}$ und F.

Kriterium 2: $\epsilon \in \text{First}(\overline{E})$: $\text{First}(\overline{E}) \cap \text{Follow}(\overline{E}) = \emptyset$
Analog für $\overline{T}$: $\text{First}(\overline{T}) \cap \text{Follow}(\overline{T}) = \emptyset$

Somit ist G eine LL(1)-Grammatik. `Ende Beispiel`

Manchmal kann man aus einer kontextfreien Grammatik $G = (N, T, P, \sigma)$ eine äquivalente LL(1)-Grammatik durch die Transformation der Links-Faktorisierung gewinnen: Das soll an einem bekannten Beispiel erklärt werden.

Beispiel: „hängendes“ else
<stmt>→ <u>if</u> <exp> <u>then</u> <stmt> | <u>if</u> <exp> <u>then</u> <stmt> <u>else</u> <stmt>
Bei der LL(1)-Analyse muß die Prodution durch den ersten Token (hier <u>if</u>) festgelegt sein. Das ist hier aber nicht möglich. Erst das auftretende <u>else</u> kann die Entscheidung liefern. Aber das else kann beliebig weit vom <u>if</u> entfernt stehen, so daß kein noch so großes (aber festes) Lookahead diese Entscheidung liefern kann.

Transformation: <stmt>→ <u>if</u> <exp> <u>then</u> <stmt> <if_rest>
<if_rest> → <u>else</u> <stmt> | ϵ
Die Entscheidung wird also erst nötig, wenn der <if_rest> analysiert wird. `Ende Beispiel`

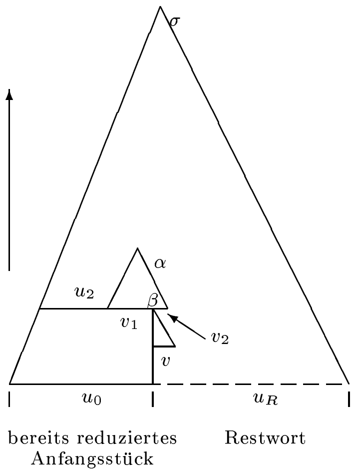
Diese Transformation heißt *Links-Faktorisierung*: Sei $\alpha \to uu_1|uu_2$ und $u \to_L^* tu'$, dann kann man aus t nicht auf die Produktion für α schließen.

Ersetzung: $\alpha \to u\overline{\alpha}, \overline{\alpha} \to u_1|u_2$.

4.2 Bottom-up Syntaxanalyse

4.2.1 Einleitung

In 4.1 wurde bereits ein allgemeines Verfahren für einen NPDA skizziert, das den Ableitungsbaum von unten nach oben (bottom-up) erzeugt.



Situation: u_0 wurde bereits zu $u_2v_1 \in V^*$ reduziert. u_R ist das Restwort in der Eingabe und u_2v_1 der Kellerinhalt. Wenn man ein Anfangsstück von u_R auf v_2 reduzieren kann und $(\alpha, v_1v_2) \in P$, dann kann man (danach) v_1v_2 zu α reduzieren.

- Wenn $v_2 = tv_2'$ mit $t \in T$ und $u_R = tu_R'$ dann wird t auf den Keller verschoben und u_R' ist neues Restwort.
- Wenn $v_2 = \beta v_2'$ mit $\beta \in N$, dann sucht man eine Produktion $(\beta, v) \in P$ und versucht, ein Anfangsstück von u_R zu v zu reduzieren.

Diese Regeln werden solange angewendet, bis $u = u_0u_R$ zu σ reduziert ist. Immer wenn man eine Reduktion (α, v) vornimmt, schreibt man die Produktion auf ein Ausgabeband. Dort entsteht eine Rechtsableitung in umgekehrter Reihenfolge (der letzte Rechtsableitungsschritt zuerst).

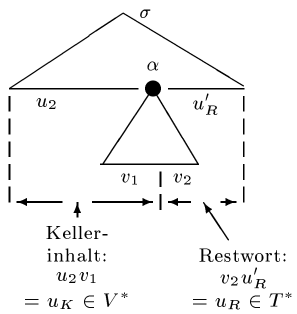
Dieses Verfahren ist zunächst nur von theoretischem Interesse, da es nichtdeterministisch ist und daher sehr ineffizient sein kann. In diesem Abschnitt sollen effiziente Algorithmen für die LR-Analyse vorgestellt werden.

Bemerkungen:

1. Es gibt einen Algorithmus von Earley (1970), der beliebige kontextfreie Grammatiken verwenden kann und eine Laufzeit aus $O(n^3)$ und einen Speicherplatzbedarf aus $O(n^2)$ hat. Dabei ist n die Länge des Eingabewortes.
2. Die frühen Compiler (z.B. Fortran II) haben i.W. das Problem behandelt, arithmetische Ausdrücke (<u>FOR</u>MELN) zu transformieren (<u>TRAN</u>SLATE). Die Ausdrücke dürfen dabei Klammern einsparen. Die Eindeutigkeit der (impliziten) Klammerung ergibt scih aus bestimmten Regeln der *Operator-Präzedenz* und der Assoziativität 2-stelliger Operatoren. Aus dieser Problemstellung entstand eine spezielle Form der LR-Analyse, die man auch Operator-Präzedenz-Analyse nennt. Diese spezielle Form der Analyse, die heute nur in speziellen Anwendungen vorkommt, ist aber recht effizient und kann speziell für die Analyse von Ausdrücken verwendet werden. Das läßt sich auch für die Top-down Analyse bei Verfahren des rekursiven Abstiegs anwenden. In diesem Abschnitt wird dieses Verfahren aber nur skizziert.

4.2.2 Der SR-Parser

Wir wollen die Situation eines Kellerautomaten für die Bottom-up Analyse nochmals genauer betrachten:



Wir nehmen an, daß $(\alpha, v) \in P$ die nächste mögliche Reduktion ist. Auf dem Kellerband befindet sich ein Anfangsstück $v_1 \leq v = v_1v_2$. Wenn $v_2 \neq \epsilon$ ist, dann muß zunächst das Wort v_2 ganz auf den Keller „verschoben" werden (shift). Erst dann kann das oberste Kellerwort $v = v_1v_2$ vermöge $(\alpha, v) \in P$ zu α reduziert werden, so daß dann v_1v_2 durch α ersetzt wird (reduce). Es ist also in jeder Situation (u_K, u_R) zu entscheiden, ob

entweder ein weiteres Zeichen von u_R auf den Keller zu schieben ist (shift)

oder der Kellerinhalt u_K eine Zerlegung $u_K = u_2v$ besitzt mit $(\alpha, v) \in P$, so daß v durch α ersetzt werden kann (reduce).

Ein solcher Parser heißt daher auch ein *Shift-Reduce-Parser.*

Probleme:

1. Wie findet man die Zerlegung des Kellerwortes $u_K = u_2v$. Muß dazu der gesamte Kellerinhalt u_K gelesen werden?
2. Wenn es eine solche Zerlegung für keine Produktion $(\alpha, v) \in P$ gibt, liegt dann ein Fehler in der Eingabe vor, oder kann man durch Schiebeoperationen wieder zu einer Reduktion gelangen?

Wir wollen wieder solche kontextfreien Grammatiken betrachten, bei denen diese Fragen durch eine Vorausschau (Lookahead) von k Zeichen von u_R entschieden werden können. Dazu sind wieder einige technische Definitionen für kontextfreie Grammatiken nötig. Im folgenden sei stets $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik.

Satz 4.2.1 *Eine reduzierte kontextfreie Grammatik ist genau dann eindeutig, wenn die Reduktionsklassen paarweise disjunkt sind.*

Beweis: Ohne Beweis (siehe Formale Sprachen). □

Wie bei den Ableitungsklassen kann man aus diesem Ergebnis schließen, daß die Reduktionsklassen nicht immer regulär sein können, denn die Disjunktheit von regulären Mengen ist entscheidbar, während die Eindeutigkeit von kontextfreien Grammatiken nicht entscheidbar ist.

Ein Shift-Reduce-Parser für eindeutige kontextfreie Grammatiken:

```
begin uK := ε; read (uR); error := false ;
  while not error and (uK, uR) ∉ R(σ) do
    begin
    if (uK, uR) = (u0v, uR) ∈ R(α, v)
      then begin { reduce }
       uK := u0α; write ((α, v))
      end
    else if (uK, uR) = (uK, tw) ∈ R(shift)
      then begin { shift }
       uK := uKt; uR := w
      end
    else error := true
    end
  if (uK, uR) ∈ R(σ) then accept
                          else { Fehlerbehandlung }
end
```

Sei $S_R(G) = \{u \in V^* | \sigma \rightarrow_R^* u\} \subseteq S(G)$ die Menge der Rechts-Satzformen (R-Satzformen). Jedem Schnitt bei der Bottom-up Analyse entspricht eine R-Satzform.

Definition 4.2.2 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik und $u \in S_R(G)$. Ein* Handle *(*Henkel*,* Griff*) ist eine Zerlegung $u = u_1 v w$, so daß gilt:*

$$\exists (\alpha, v) \in P : u_1 \alpha w \in S_R(G).$$

Anschaulich: (α, v) ist letzter R-Ableitungsschritt:

$$\begin{tikzcd}
\sigma \arrow[rr, "*", "R"'] \arrow[dr, "*", "R"'] & & u_1 v w \\
& u_1 \alpha w \arrow[ur, "R"'] &
\end{tikzcd}$$

Jede solche Faktorisierung ist ein Henkel von u.

Wenn G eindeutig ist, dann hat jede R-Satzform genau einen Henkel, sonst kann es mehrere geben.

Beispiel:(Grammatik für Ausdrücke)

$$\begin{tikzcd}
 & \sigma * \sigma \arrow[r, "R"'] & \sigma * \mathrm{id} \arrow[dr, "R"'] & \\
\sigma \arrow[ur, "R"'] \arrow[dr, "R"'] & & & \sigma + \sigma * \mathrm{id} \\
 & \sigma + \sigma \arrow[r, "R"'] & \sigma + \sigma * \sigma \arrow[ur, "R"'] &
\end{tikzcd}$$

Die R-Satzform $\sigma + \sigma * \mathrm{id}$ hat die Henkel:

1. $u_1 = \epsilon$; $v = \sigma + \sigma$; $w = *\mathrm{id}$ mit $(\sigma \to \sigma + \sigma)$,
2. $u_1 = \sigma + \sigma *$; $v = \mathrm{id}$; $w = \epsilon$ mit $(\sigma \to \mathrm{id})$.

`Ende Beispiel`

Ist die Grammatik $G = (N, T, P, \sigma)$ eindeutig, dann besitzt $w \in L(G) \subseteq S_R(G)$ einen eindeutig bestimmten Henkel

$$u^{(0)} = w = uvw_1 \text{ mit } (\alpha, v) \in P$$
$$u^{(1)} = u\alpha w_1 \in S_R(G)$$

besitzt dann wieder eine eindeutig bestimmte R-Ableitung (in umgekehrter Reihenfolge).

Das Verfahren liefert einen Algorithmus zur Bottom-up Analyse durch sukzessive Ersetzung (Reduktion) von Henkeln (*Handle-Pruning*). Es ist aber nicht immer einfach, in einer R-Satzform den Henkel zu finden.

Analog zu den Ableitungsklassen der Top-down Analyse werden jetzt für die Bottom-up Analyse Reduktionsklassen definiert.

Definition 4.2.3 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik. Man definiert die folgenden Reduktionsklassen:*

1. $(\alpha, v) \in P$:
$$R(\alpha, v) = \left\{ (u_0 v, w) | w \in T^* \wedge \sigma \overset{*}{\underset{R}{\to}} u_0 \alpha w \right\},$$
$$S(\alpha, v) = \{(u_0 v_1, v_2 w_2) | (u_0 v_1 v_2, w_2) \in R(\alpha, v) \wedge |v_2| \geq 1\},$$
2. $R(\mathrm{shift}) = \bigcup_{\pi \in P} S(\pi)$ *(Shift-Klasse),*
3. $R(\sigma) = \{(\sigma, \epsilon)\}$ *(σ-Klasse).*

Leider ist der Test auf die Zugehörigkeit zu den Reduktionsklassen i.a. viel zu aufwendig, da diese selbst i.a. kontextfreie Sprachen sind.

Man definiert daher wieder geeignete (reguläre) Obermengen wie bei der Top-down Analyse. Sei zu $G = (N, T, P, \sigma)$ wieder $G^{(k)}$ die k-Erweiterung mit $L\left(G^{(k)}\right) = L(G)\#^k$ wie in 4.2.

Für $\gamma \in \{\mathrm{shift}, \sigma\} \cup P$ sei:

$$Q_\gamma^{(k)} = \{(u, w_1 w_2) || w_1| = k \wedge \exists w_2' \in T^* : (u, w_1 w_2') \in R(\gamma)\}$$

und

$$K_\gamma^{(k)} = \{u w_1 || w_1| = k \wedge \exists w_2' \in T^* : (u, w_1 w_2') \in R(\gamma)\}.$$

$K_\gamma^{(k)}$ heißt die *Kellerklasse der Ordnung k* für $\gamma \in P \cup \{\mathrm{shift}, \sigma\}$ von G.

Definition 4.2.4

1. *Eine reduzierte kontextfreie Grammatik* $G = (N, T, P, \sigma)$ *heißt* LR(k)-Grammatik *g.d.w. die Kellerklassen der Ordnung* k *paarweise disjunkt sind.*
2. *Eine Sprache* $L \subseteq T^*$ *heißt* LR(k)-Sprache, *wenn es eine LR(k)-Grammatik* G *gibt, die* L *erzeugt.*

Korollar 4.2.5 *Jede LR(k)-Grammatik ist eindeutig.*

Übung:

1. $P : \sigma \to a^2\sigma|a$ ist LR(1) aber nicht LR(0).
2. $P' : \sigma \to a\sigma a|a$ erzeugt die gleiche Sprache, ist aber für kein $k \in I\!N$: LR(k)!

Die Kellerklassen einer LR(k)-Grammatik kann man nun wieder als Test für die Aktionen eines Shift-Reduce Parsers verwenden.

Satz 4.2.6 *Die Kellerklassen einer kontextfreien Grammatik sind stets regulär. Man kann die zugehörigen linkslinearen Grammatiken effektiv konstruieren.*

Beweis: Ohne Beweis. □

Ein triviales Beispiel zeigt, wie man einen endlichen Automaten zur Klassifizierung von Situationen in die Kellerklassen verwenden kann.

Beispiel: P:

$\sigma \to \alpha \quad (1)$

$\alpha \to \beta b|\epsilon \quad (2,3)$

$\beta \to \alpha a \quad (4)$

$L_\sigma = L_\alpha = L_\beta b + \epsilon = L_\alpha ab + \epsilon = (ab)^*$

- Reductionsklassen:

  $R(\sigma, \alpha) = \{(\alpha, \epsilon)\}$
  $R(\alpha, \beta b) = \{(\beta b, (ab)^n)|n \geq 0\}$
  $R(\alpha, \epsilon) = \{(\epsilon, (ab)^n)|n \geq 0\}$
  $R(\beta, \alpha a) = \{(\alpha a, b(ab)^n)|n \geq 0\}$
  $R(\text{shift}) = \{(\beta, b(ab)^n), (\alpha, (ab)^{n+1})|n \geq 0\}$

  Diese Klassen sind paarweise disjunkt und somit ist G eindeutig.

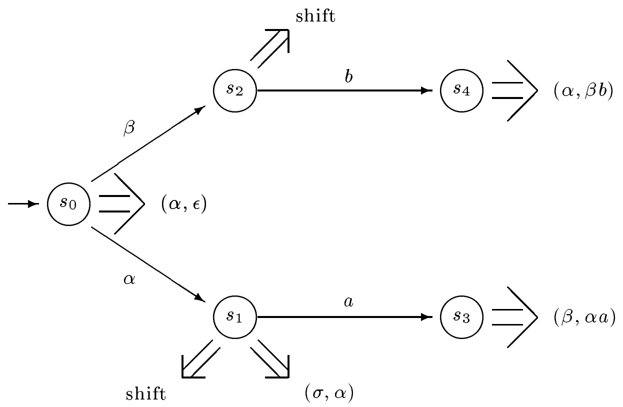
- Kellerklassen $K_\gamma^{(k)}$ für $k = 0$:

  $K^{(0)}(\sigma, \alpha) = \{\alpha\};$
  $K^{(0)}(\alpha, \beta b) = \{\beta b\};$

$K^{(0)}(\alpha, \epsilon) = \{\epsilon\};$
$K^{(0)}(\beta, \alpha a) = \{\alpha a\};$
$K^{(0)}(\text{shift}) = \{\alpha, \beta\};$

Da $K^{(0)}(\sigma, \alpha) \cap K^{(0)}(\text{shift}) \neq \emptyset \Rightarrow G$ nicht LR(0).

Man kann trotzdem einen Akzeptor (endlichen Automaten) für diese trivialen Klassen verwenden:



Kellerklassen für $k = 1$:

$K^{(1)}(\sigma, \alpha) = \{\alpha\#\};$

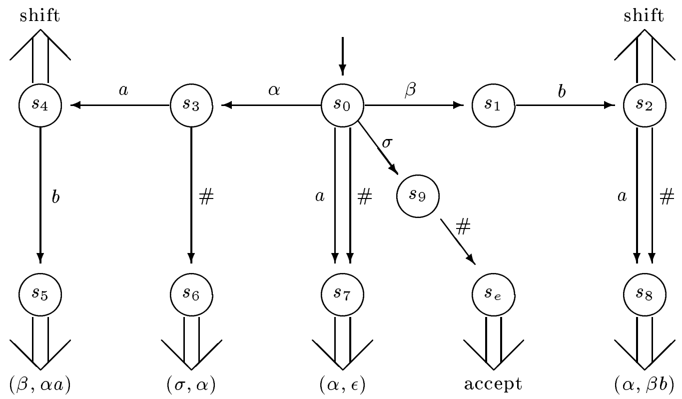
$K^{(1)}(\alpha, \beta b) = \{\beta ba, \beta b\#\};$

$K^{(1)}(\alpha, \epsilon) = \{a, \#\};$

$K^{(1)}(\beta, \alpha a) = \{\alpha ab\};$

$K^{(1)}(\text{shift}) = \{\alpha a, \beta b\};$

Man sieht jetzt, daß G LR(1) ist.



`Ende Beispiel`

Man kann also geeignete endliche Automaten zur Klassifizierung der Situationen verwenden. Das Kellerwort $u_K = u_0 v_1$ und ein Lookahead der Länge k: $z \in T^k$ definieren die jeweils nächste Aktion eines Shift-Reduce-Parsers (SR-Parser).

Sei $A = (S, \delta, F, s_0)$ ein endlicher Automat, der die Situationen (u_K, z) mit $z \in T^k$ klassifizieren kann. Das Wort $u_K z \in V^* T^k \subseteq V^*$ liefert dann jeweils eine der Aktionen aus

$$\text{Act} = P \cup \{\text{shift}, \text{accept}, \text{error}\}.$$

Eine Aktion $\pi \in P$ ist dann eine *Reduce-Aktion*. Sei $\delta : S \times V \to S$ die Zustandsabbildung des Automaten A, dann kann man diese auf $S \times V^*$ fortsetzen zu $\delta^* : S \times V^* \to S$ durch

1. $\delta^*(s, \epsilon) = s \quad (s \in S)$,
2. $\delta^*(s, ux) = \delta(\delta^*(s, u), x) \quad (u \in V^*, x \in V)$.

Für $\delta^*(s, u) = s'$ schreiben wir auch einfach: $su = s'$. Jedes Anfangsstück von u definiert somit einen Zustand des Automaten:

$$u = u_1 ... u_m \text{ und } u_1 ... u_i \leq u \text{ (Präfix)}$$

dann sei $s_i = s_0(u_1 ... u_i)$. Das Wort $u = u_1 ... u_m$ definiert ein zugehöriges Zustandswort: $s_0 s_1 ... s_m$. Der Zustand s_m ist die gesamte Information, die der Automat nach der Eingabe von $u = u_1 ... u_m$ gespeichert hat (der Automat hat ein endliches Gedächtnis). Die Auswahl einer Aktion aus Act ist also nur abhängig von dem Zustand $s_0 u_K$ und dem Lookahead $z \in T^k$. Daher kann man eine entsprechende Tabelle

$$\text{action} : S \times T^k \to \text{Act}$$

berechnen. δ ist die Tabelle des endlichen Automaten und heißt hier next_state:

$$\text{next\_state} : S \times V \to S.$$

Sie ist i.a. nur partiell definiert.

Die wesentliche Idee des SR-Parsers besteht nun darin, daß man nicht mehr das Kellerwort auf dem Kellerspeicher ablegt, sondern das zugehörige Zustandswort:

Ist $u_K = u_1 ... u_m$ das (eigentliche) Kellerwort, dann wird auf dem Kellerspeicher abgelegt

$$s_0 s_1 ... s_m \in S^*,$$

wobei gilt: $s_i = s_0 u_1 ... u_i$ $(0 \leq i \leq m)$. Wir nennen jetzt weiterhin u_K das *Kellerwort* und $s_0 s_1 ... s_m$ den *Kellerinhalt*.

Ist $u_K = uv$ Kellerwort und $(\alpha, v) \in P$, dann kann man v zu α reduzieren. Dabei wird v durch α ersetzt: $u'_K = u\alpha$. Der Kellerinhalt zu diesem Kellerwort ergibt sich wie folgt: Sei $u_K = u_1 ... u_r v_1 ... v_q$ und $s_0 s_1 ... s_r s_{r+1} ... s_{r+q}$ der zugehörige Kellerinhalt. Es ist dann $u'_K = u_1 ... u_r \alpha$ und dazu gehört der Kellerinhalt: $s_0 s_1 ... s_r s'_{r+1}$ mit $s'_{r+1} = \text{next\_state}(s_r, \alpha)$. Bei einem Reduktionsschritt mit (α, v) und $|v| = q$ werden also einfach die obersten q Elemente $s_{r+1} ... s_{r+q}$ vom Keller entfernt. Dann ist s_r oberstes Element und man berechnet dann $s'_{r+1} = \text{next\_state}(s_r, \alpha)$ und legt diesen Zustand auf dem Keller ab.

Bei einer Shift-Aktion wird ein Terminalsymbol $t \in T$ vom Eingabeband an das Kellerwort „geschoben“: $u'_K = u_K t$ und der zugehörige Kellerinhalt ist dann der alte Kellerinhalt „auf“ dem der durch die Eingabe t und den obersten Zustand s des Kellerinhaltes berechnete Zustand $s' = \text{next\_state}(s, t)$.

Damit kann man einen allgemeinen SR-Parser formulieren, der die beiden Tabellen interpretiert:

```
SR-Parser: (für LR(1)-Grammatiken)
push(s0); { initialer Kellerinhalt }
done := false ;
repeat
  s := top; t := current_token; { Lookahead }
   case action[s, t] of
    shift: begin
         push(next_state[s, t]);
         t := get_next_token
          end ;
    (α, v): begin
      q := |v|; pop^q; s := top;
      push(next_state[s, α])
       end ;
    accept: done := true ;
    error: [Fehlerbehandlung]
     end { case }
 until done;
```

Beispiel:(Fortsetzung) Für die weiter oben untersuchte LR(1)-Grammatik kann man die Tabellen next_state und action direkt aus dem konstruierten Automaten ablesen:

action						
	s_0	s_1	s_2	s_3	s_4	s_9
a	(α, ϵ)	–	$(\alpha, \beta b)$	shift	–	–
b	–	shift	–	–	$(\beta, \alpha a)$	–
#	(α, ϵ)	–	$(\alpha, \beta b)$	(σ, α)	–	accept

next_state						
	s_0	s_1	s_2	s_3	s_4	s_9
a	s_7	–	s_8	s_4	–	–
b	–	s_2	–	–	s_6	–
#	s_7	–	s_8	s_6	–	s_e
α	s_3	–	–	–	–	–
β	s_1	–	–	–	–	–
σ	s_9	–	–	–	–	–

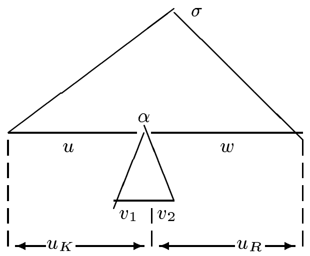
Ablauf für die Eingabe $abab\#$			
Stack	Restwort	Action	Neue Situation
s_0	$abab\#$	(α, ϵ)	$(\alpha, abab\#)$
s_0s_3	$abab\#$	shift	$(\alpha a, bab\#)$
$s_0s_3s_4$	$bab\#$	$(\beta, \alpha a)$	$(\beta, bab\#)$
s_0s_1	$bab\#$	shift	$(\beta b, ab\#)$
$s_0s_1s_2$	$ab\#$	$(\alpha, \beta b)$	$(\alpha, ab\#)$
s_0s_3	$ab\#$	shift	$(\alpha a, b\#)$
$s_0s_3s_4$	$b\#$	$(\beta, \alpha a)$	$(\beta, b\#)$
s_0s_1	$b\#$	shift	$(\beta b, \#)$
$s_0s_1s_2$	$\#$	$(\alpha, \beta b)$	$(\alpha, \#)$
s_0s_3	$\#$	(σ, α)	$(\sigma, \#)$
s_0s_e	$\#$	accept	–

`Ende Beispiel`

4.2.3 Die Tabellen des SR-Parsers

Es soll jetzt noch darauf eingegangen werden, wie man die Tabellen für LR(0) und LR(1)-Parser systematisch erzeugt. Es wird sich zeigen, daß diese Tabellen für LR(1)-Parser bei praktisch vorkommenden Programmiersprachen sehr groß werden können, so daß man noch Zwischenformen definiert, die mit kleineren Tabellen auskommen.

Betrachten wir nochmals eine Situation als Schnitt im Ableitungsbaum:



Bekannt: Situation (u_K, u_R) mit $u_K \in V^*, u_R \in T^*$. Sei $(\alpha, v) \in P$; $v = v_1v_2$ der nächste mögliche Reduktionsschritt.

Gesucht also: Zerlegung $u_K = uv_1$, $u_R = v_2w$ um daraus auf (α, v) schließen zu können. Das Kellerwort u_K hat die folgende Eigenschaft: $\sigma \rightarrow^*_R u_K u_R = uv_1v_2w$ und uvw ist Henkel von $u_K u_R$. Ein Henkel uvw ist durch das Anfangsstück $uv \leq uvw$ und die Zerlegung in u und v bestimmt. Ein Henkel ist also ein Präfix der gegebenen Rechtssatzform, der mit der rechten Seite v einer Produktion $(\alpha, v) \in P$ endet. Man kann daher auch den Henkel (vereinfachend) durch uv bezeichnen (statt korrekterweise: (u, v)).

Ist uv Henkel einer R-Satzform von G, also existiert ein $w \in T^*$ mit $uvw \in S_R(G)$ und $(\alpha, v) \in P$, und ist $\hat{u} \leq uv$, dann heißt $\hat{u}$ ein *gültiger Präfix*. Die gültigen Präfixe einer Grammatik G sind also genau die möglichen Kellerwörter der LR-Analyse. Man kann diese Menge $\mathrm{Pr}(G)$ der gültigen Präfixe von G auch so kennzeichnen:

$$\hat{u} \in \mathrm{Pr}(G) \iff \exists w \in T^* : \hat{u}w \in S_R(G)$$

$$\mathrm{Pr}(G) \subseteq V^*.$$

Ist G eindeutig, dann hat jede R-Satzform einen eindeutig bestimmten Henkel und wenn $\hat{u} \in \mathrm{Pr}(G)$ mit $\hat{u}w \in S_R(G)$, dann ist $\hat{u}$ auch Präfix des Henkels uv von $\hat{u}w$. Ist $\hat{u} = uv$ (also Henkel) dann heißt $\hat{u}$ *maximal*. Die Menge $\mathrm{Pr}(G) \subseteq V^*$ ist regulär.

Statt nun die gültigen Präfixe auf dem Keller zu speichern, legt man dort nur das zugehörige Zustandswort des erkennenden Automaten für $\mathrm{Pr}(G)$ ab. Für eine LR(0)-Grammatik wird die nächste Aktion des SR-Parsers allein durch den gültigen Präfix auf dem Keller, und daher nur durch den „obersten" Zustand auf diesem Keller bestimmt.

Ein gültiger Präfix u_K endet mit einem Anfangsstück (Präfix) einer rechten Seite einer Produktion $(\alpha, v_1 v_2)$ und läßt sich so zum Henkel $u_K v_2 = u'_K v_1 v_2$ fortsetzen. Man definiert daher alle Zerlegungen der rechten Seiten von Produktionen aus P als die Menge der *LR(0)-Konfigurationen* (auch *kontextfreie items* oder *0-Konfigurationen*). Eine 0-Konfiguration ist also ein Tripel (α, v_1, v_2) mit $(\alpha, v_1 v_2) \in P$. Dabei ist auch $v_1 = \epsilon$ oder $v_2 = \epsilon$ eingeschlossen. Sei also

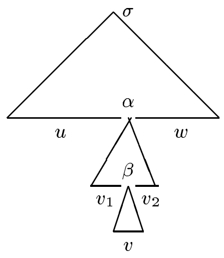
$$K_0(G) = K_0 = \{(\alpha, v_1, v_2) | (\alpha, v_1 v_2) \in P\}.$$

K_0 ist eine endliche Menge, denn P ist endlich. Statt (α, v_1, v_2) schreibt man in der Literatur meistens $(\alpha, v_1.v_2)$ oder $\alpha \to v_1.v_2$. Wir wollen diese Notation auch hier übernehmen.

Ist $\hat{u}$ ein gültiger Prafix ($u \in \mathrm{Pr}(G)$) und $c = (\alpha, v_1.v_2) \in K_0(G)$, dann sagt man: *$c$ ist gültig für $\hat{u}$* falls gilt: $\hat{u} = uv_1$, also $\hat{u}$ endet mit v_1. In diesem Fall läßt sich $\hat{u} = uv_1$ zu dem Henkel uv_1v_2 fortsetzen.

Eine gültige Konfiguration c für $\hat{u}$ ist also ein partielles Matching zwischen einem Endstück von $\hat{u}$ und einem Anfangsstück der rechten Seiten einer Produktion $(\alpha, v_1 v_2)$. Ist insbesondere $(\alpha, v.)$ gültig für $\hat{u}$, dann ist das Matching vollständig ($(\alpha, v.)$ heißt dann auch *akzeptierende Konfiguration* für $\hat{u}$). $(\alpha, .v)$ ist dagegen für jeden gültigen Präfix $\hat{u}$ gültig (*triviales Matching*, *voraussagende Konfiguration*). Wenn G eine LR(0)-Grammatik ist, dann liegt jeder gültige Präfix in genau einer Kellerklasse der Ordnung 0. Man konstruiert daher einen endlichen Automaten $A_0(G) = (S, \delta, s_0, F)$ (Eingabealphabet ist immer V und wird daher nicht mehr explizit angegeben), der in jedem Zustand eine Kellerklasse der Ordnung 0 klassifiziert. In einem Zustand $s \in S$ soll dieser Automat alle gültigen 0-Konfigurationen „kennen", die für den bis dahin eingegebenen gültigen Präfix gültig sind.

Die Zustände von A_0 sind also Teilmengen von K_0: $S \subseteq \mathcal{P}(K_0)$. Ist $(\alpha, v_1.\beta v_2) \in S$ $s \in S$ und $(\beta, .v) \in K_0$, dann muß auch $(\beta, .v)$ zu s gehören:



Damit β einmal in das partielle Matching $(\alpha, v_1\beta.v_2)$ aufgenommen wird, muß zuvor die Reduktion nach (β, v) stattfinden. Also wird die Konfiguration $(\beta, .v)$ „erwartet". Diese Hinzunahme von *voraussagenden Konfigurationen* nennt man die *Abschlußbildung* (*0-Hülle*, *closure*).

Definition 4.2.7 *Sei* $s \subseteq K_0(G)$, $G = (N,T,P,\sigma)$ *eine kontextfreie Grammatik. Die* 0-Hülle $\mathrm{cl}_0(s)$ *ist definiert durch*

1. $s \subseteq \mathrm{cl}_0(s)$,
2. $(\alpha, v_1.\beta v_2) \in \mathrm{cl}_0(s) \wedge (\beta, v) \in P \Rightarrow (\beta, .v) \in \mathrm{cl}_0(s)$,
3. cl_0 *ist die kleinste Menge mit 1. und 2.*

Algorithmus (Bildung der 0-Hülle):

Gegeben: $s \subseteq K_0(G)$, $G = (N,T,P,\sigma)$ kontextfreie Grammatik

Gesucht: $\mathrm{cl}_0(s) \subseteq K_0(G)$

```
begin
 closure := s;
  repeat
   done := true ;
    for (α, v1.βv2) ∈ closure do
     if (β, v) ∈ P and (β, .v) ∉ closure then
       begin
        closure := closure +(β, .v);
        done := false
       end ;
  until done
end ;
```

$s \subseteq K_0$ heißt *abgeschlossen* wenn gilt: $\mathrm{cl}_0(s) = s$. Jetzt kann der endliche Automat $A_0(G)$ konstruiert werden. Man betrachtet G als zu $G^{(1)}$ erweitert.

Definition 4.2.8 *Sei* $G = (N,T,P,\sigma)$ *eine kontextfreie Grammatik, dann sei* $A_0(G) = (S, \delta_0, s_0, F)$ *der endliche Akzeptor mit* $\delta_0 : S \times V \to S$, $F \subseteq S$, $s_0 \in S$ *und* $S \subseteq \mathcal{P}(K_0)$. *Für* $x \in V$ *ist* $\delta_0(s,x) = s'$ *definiert durch:*

1. $(\alpha, v_1.xv_2) \in s \Rightarrow (\alpha, v_1x.v_2) \in s'$,
2. $s' = \mathrm{cl}_0(s')$ *(abgeschlossen).*

$s_0 = \mathrm{cl}_0(\sigma_0, .\sigma\#)$
S ist gegeben durch:

1. $s_0 \in S$,
2. $s \in S \Rightarrow \forall x \in V : \delta_0(s,x) \in S$,

3. *S ist kleinste Menge mit 1. und 2.*

$F = S - \{\emptyset\}$

Die Funktion $\delta_0 : S \times V \to S$ heißt manchmal auch *Nachfolger-Funktion* ($succ : S \times V \to S$). Wenn in den Automaten A_0 ein gültiger Präfix u_K eingegeben wird, dann ist A_0 in einem Zustand, der die Kellerklasse dieser LR(0)-Situation festlegt. In jedem Zustand des Automaten ist daher genau eine Aktion des Parsers möglich. Diese Zuordnung wird in der Tabelle action: $S \to \text{Act}$ festgelegt.

Dazu definieren wir zunächst die Abbildung $\lambda : S \to \mathcal{P}(\text{Act})$ durch

1. $(\alpha, v.) \in s \Rightarrow (\alpha, v) \in \lambda(s)$,
2. $(\alpha, v_1.tv_2) \in s \wedge t \in T \Rightarrow \text{shift} \in \lambda(s)$,
3. $\lambda(s)$ ist die kleinste Menge mit 1. und 2.

Für eine LR(0)-Grammatik ist dann

$$\forall s \in S : |\lambda(s)| \leq 1.$$

Damit kann man dann die action-Tabelle definieren:

$$\text{action}(s) = \begin{cases} \text{accept} & \text{falls } \lambda(s) = \{(\sigma_0, \sigma\#)\} \\ (\alpha, v) & \text{falls } \lambda(s) = \{(\alpha, v)\} \\ \text{shift} & \text{falls } \lambda(s) = \{\text{shift}\} \\ \text{error} & \text{falls } \lambda(s) = \emptyset \end{cases}$$

Ist G nicht LR(0), dann gibt es Zustände $s \in S$ mit $|\lambda(s)| \geq 2$. Man nennt einen solchen Zustand auch einen *Konflikt*. Ein Konflikt s heißt *reduce/reduce-Konflikt* falls $|\lambda(s) \cap P| \geq 2$ und *shift/reduce-Konflikt* falls $\text{shift} \in \lambda(s)$ und $\lambda(s) \cap P \neq \emptyset$.

Beispiel: LR(0)-Parser Tabellen (aus Fisher/LeBlanc)

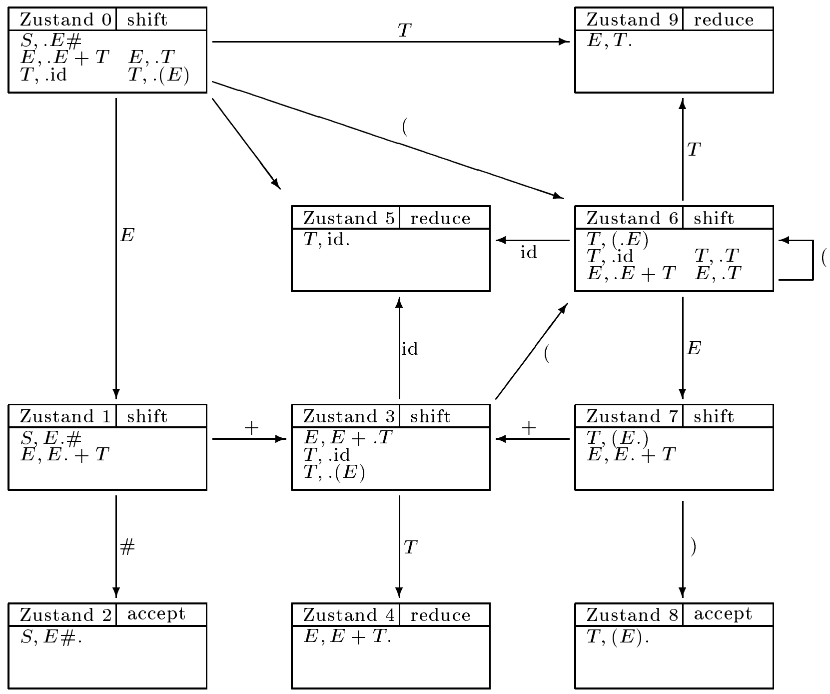
Grammatik G

$S \to E\#$

$E \to E + T | T$

$T \to \text{id} | (E)$

Zustandsgraph des Automaten $A_0(G)$:



Ende Beispiel

Damit ist die Konstruktion des Automaten $A_0(G)$ abgeschlossen. $A_0(G)$ erkennt nach Konstruktion genau die gültigen Präfixe: $\Pr(G) = L(A_0(G))$, und das ist die Vereinigung aller Kellerklassen der Ordnung 0.

Ist $p = |P|$ die Zahl der Produktionen von G und für jedes $\pi = (\alpha, v) \in P$ sei $|\pi| = |v| + 1$, dann gilt: $q_0 = |K_0| = \sum_{\pi \in P} |\pi|$ und damit ist $|\mathcal{P}(K_0)| = 2^{q_0}$. 2^{q_0} ist die maximale Zahl möglicher Zustände, aber i.a. ist $|S|$ deutlich kleiner als 2^{q_0}.

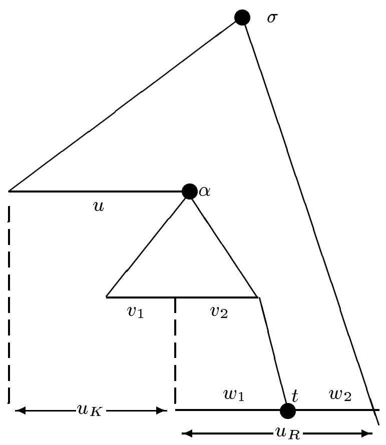
Die Tabellen für einen LR(1)-Parser sind wesentlich aufwendiger, denn zu einer 1-Konfiguration gehört jetzt zusätzlich noch ein Eingabezeichen:

$$K_1(G) = K_0(G) \times T_\epsilon.$$

Eine 1-Konfiguration hat jetzt die Form:

$$(\alpha, v_1.v_2; t) \quad (t \in T_\epsilon).$$

t ist der jeweilige *Lookahead*. Wenn die Eingabe schon auf dem Ende-Zeichen # steht, dann sei das Lookahead (nächstes Zeichen) ϵ. Die Bedeutung einer 1-Konfiguration liegt darin, daß man t aus Follow(α) wählen kann:



Kellerwort $u_K = uv_1$ ist gültiger Präfix. u_R hat die Zerlegung $u_R = w_1tw_2$ und $v_2 \rightarrow_R^* w_1$. Wenn w_1 zu v_2 reduziert ist, ist $t \in \text{Follow}(\alpha)$ das nächste Eingabezeichen. Dieses zusätzliche Lookahead-Symbol kann dann für eine LR(1)-Grammatik Konflikte auflösen, die für eine LR(0)-Tabelle eventuell noch bestehen. Analog zur Konstruktion von $A_0(G)$ wird jetzt der Automat $A_1(G)$ konstruiert. Mit $q_0 = |K_0|$ und somit $|\mathcal{P}(K_1)| = 2^{|K_0|\cdot|T_\epsilon|} = |\mathcal{P}(K_0)|^{|T_\epsilon|}$. Die Menge der möglichen Zustände wächst also exponentiell mit $|T|$. Entsprechend kann auch die Zustandsmenge von $A_1(G)$ sehr groß werden. Bei der Konstruktion von A_1 $(= A_1(G))$ werden wieder die Funktionen 1-Hülle und Nachfolger benötigt.

Definition 4.2.9 *Sei* $G = (N, T, P, \sigma)$ *eine kontextfreie Grammatik und* $s \in K_1(G)$. *Die* 1-Hülle *von* s *ist* $\text{cl}_1(s)$ *und definiert durch*

1. $s \subseteq \text{cl}_1(s)$,
2. $(\alpha, v_1.\beta v_2; t) \in \text{cl}_1(s)$, $(\beta, v) \in P$ *und* $t' \in \text{First}(v_2t) \Rightarrow (\beta, .v, t') \in \text{cl}_1(s)$,
3. $\text{cl}_1(s)$ *ist die kleinste Menge mit 1. und 2.*

Notation: Wenn mehrere 1-Konfigurationen die gleiche linke Seite $c \in K_0$ aber unterschiedliche Lookahead-Symbole $t_1, ..., t_r$ haben, dann kürzt man das wie folgt ab:

$$(c, \{t_1, ..., t_r\}) \text{ steht für } \{(c_i, t_i) | 0 \leq i \leq r\}.$$

Beispiel:(1-Hülle): Sei G die gleiche Grammatik wie im vorigen Beispiel, und $(S,\ .E\#;\ \epsilon) \in K_1$. In $(S, .E\#; \epsilon)$ ist $E \in N$, also müssen alle E-Produktionen mit dem . am Anfang aufgenommen werden. Hier ist $v_2 = \#$ und $t = \epsilon$ $\text{First}(v_2t) = \text{First}(\#) = \{\#\}$. Somit muß auch $(E, .E + T; \#)$ und $(E, .T; \#)$ in der 1-Hülle liegen.

In $(E, .E + T; \#)$ ist wieder $E \in N$, diesmal aber mit $v_2 = +T$ und $t = \#$. Daher müssen die gleichen 0-Konfigurationen nochmals mit dem Lookahead $+ \in \text{First}(v_2t) = \text{First}(+T\#) = \{+\}$ aufgenommen werden. Analog müssen die vorhersagenden 0-Konfigurationen für T mit den gleichen Lookahead-Symbolen aufgenommen werden:

$$\begin{aligned} \text{cl}_1(S, .E\#; \epsilon) \quad = \quad & \{(S, .E\#; \epsilon)\} \\ & \cup(E, .E + T; \{\#, +\}) \\ & \cup(E, .T; \{\#, +\}) \\ & \cup(T, .\text{id}; \{\#, +\}) \\ & \cup(T, .(E); \{\#, +\}). \end{aligned}$$

`Ende Beispiel`

Erkenntlich ist der Aufwand zur Berechnung der 1-Hülle größer als für die Berechnung der 0-Hülle. Diese Arbeit kann aber ein Algorithmus erledigen.

Die Nachfolgerfunktion $succ(s, x)$ für $s \subseteq K_1$ und $x \in V$ ist unabhängig vom Lookahead definiert durch:

Definition 4.2.10 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik und $s \subseteq K_1$ und $x \in V = N \cup T$, dann sei $succ(s, x)$ definiert durch*

1. $(\alpha, v_1.xv_2; t) \in s \Rightarrow (\alpha, v_1x.v_2; t) \in succ(s, x)$,
2. *$succ(s, x)$ ist abgeschlossen: $s' = succ(s, x) \Rightarrow \mathrm{cl}_1(s') = s'$,*
3. *$succ(s, x)$ ist die kleinste Menge mit 1. und 2.*

Damit ist der Automat $A_1(G) = (S, \delta_1, s_0, F)$ wie folgt zu konstruieren.

Definition 4.2.11 *Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik. Der Automat $A_1(G) = (S, \delta_1, s_0, F)$ ist gegeben durch:*

1. $s_0 = \mathrm{cl}_1(\sigma_0, .\sigma\#; \epsilon) \subseteq K_1$; $s_0 \in S$,
2. *wenn $s \in S$, $x \in V$ dann ist $succ(s, x) \in S$,*
3. *S ist die kleinste Teilmenge von $\mathcal{P}(K_1)$ mit 1. und 2.*
4. $\delta_1(s, x) = succ(s, x)$,
5. $F = S - \{\emptyset\}$.

Die action-Tabelle action: $S \times T \to \mathcal{P}(\mathrm{Act})$ berücksichtigt jetzt das Lookahead-Symbol. Zunächst die Funktion $\lambda : S \times T \to \mathcal{P}(\mathrm{Act})$:

1. $(\alpha, v.; t) \in s \Rightarrow (\alpha, v) \in \lambda(s, t)$,
2. $(\alpha, v_1.tv_2; t') \in s \Rightarrow \mathrm{shift} \in \lambda(s, t)$,
3. $\lambda(s, t)$ ist die kleinste Menge mit 1. und 2.

Es gilt dann analog zum Fall LR(0):

$$G \text{ ist LR(1)} \iff \forall s \in S, t \in T : |\lambda(s, t)| \leq 1.$$

Damit kann man auch die action-Tabelle analog zum Fall LR(0) definieren:

$$\mathrm{action}(s, t) = \begin{cases} \text{accept} & \text{falls } \lambda(s, \#) = \{\text{shift}\}; t = \# \\ \text{shift} & \text{falls } \lambda(s, t) = \{\text{shift}\}; t \neq \# \\ (\alpha, v) & \text{falls } \lambda(s, t) = \{(\alpha, v)\} \\ \text{error} & \text{falls } \lambda(s, t) = \emptyset \end{cases}$$

Beispiel:(nach Fisher; LeBlanc): Grammatik für Ausdrücke:

$S \to E\$,$

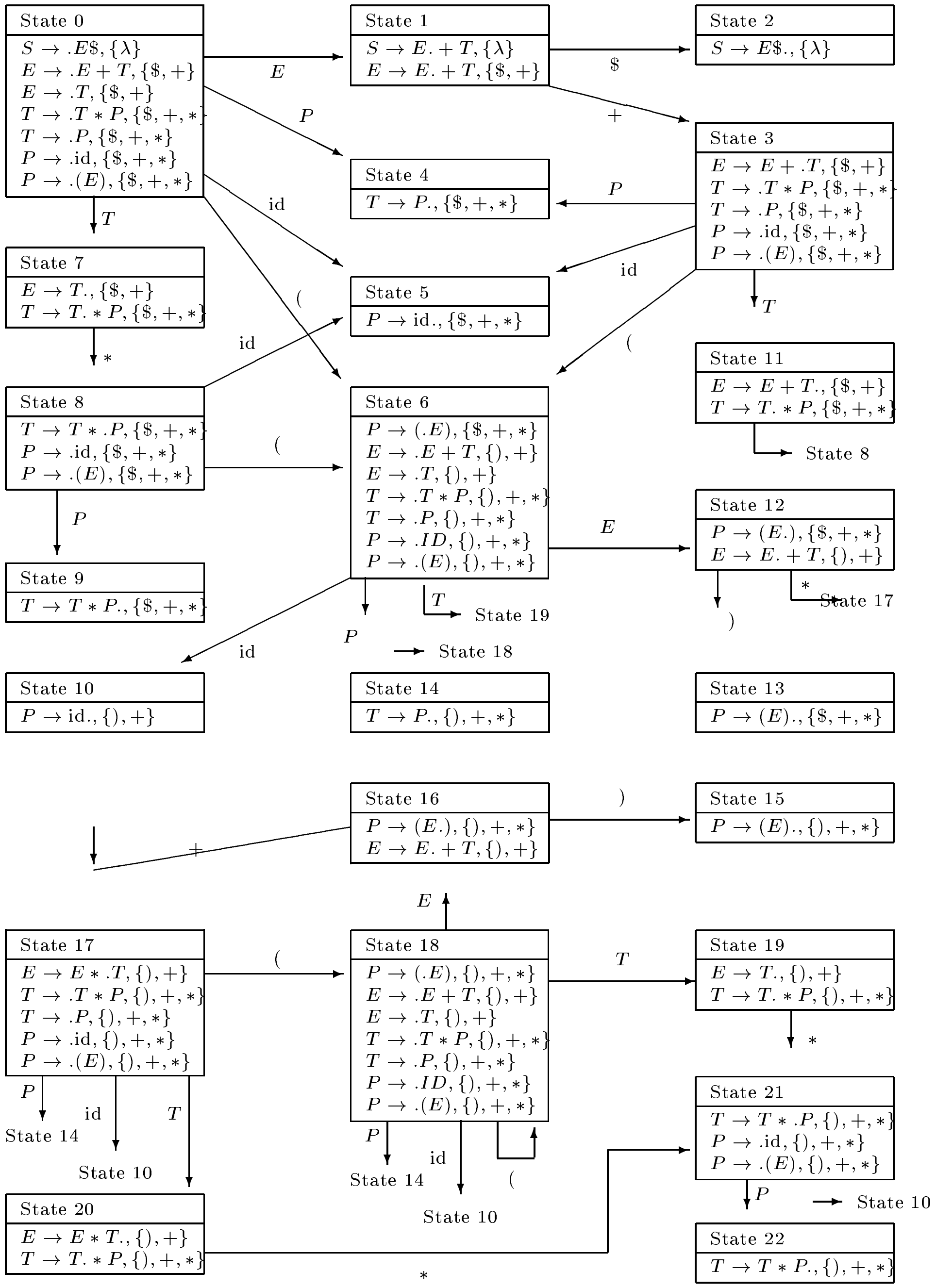
$E \to E + T|T,$

$T \to T * P|P,$

$P \to \mathrm{id}|(E).$

Bemerkung: Hier wurde das Symbol # durch das Symbol \$ ersetzt und in dem folgenden Automaten-Graphen bedeutet λ das leere Wort, also das $\epsilon \in V^*$. λ ist also nicht mit unserer Funktion λ zu verwechseln. Diese notationelle Änderung war nötig, um die aufwendigen Graphen nicht neu editieren zu müssen.

Die action-Funktion kann dem Graphen implizit entnommen werden: Etwa im Zustand 7 ist eine shift- und eine reduce-Konfiguration enthalten. Wenn das Lookahead „$*$“ ist, dann erfolgt die shift-Operation und wenn das Lookahead Symbol „\$“ oder „+“ ist dann erfolgt die Reduce-Operation. Sonst liegt ein Fehler vor.



Ende Beispiel

4.2.4 Alternativen zum SR(1)-Parser

Der Aufwand für eine LR(1)-Tabelle ist für reale Programmiersprachen meist unvertretbar hoch. So können leicht Automaten mit $2 \cdot 10^4$ Zuständen entstehen, und bei $5 \cdot 10^1$ Token ergeben sich dann 10^6 Tabelleneinträge für die action-Tabelle.

Es werden daher für reale Programmiersprachen Grammatiken verwendet, die in ihrer Ausdruckskraft zwischen LR(0) und LR(1) angesiedelt sind. Man versucht dabei, den Automaten $A_0(G)$ beizubehalten. Wenn dieser Automat Konflikt-Zustände enthält, dann werden diese durch das Lookahead-Symbol aufgelöst. Man verwendet zwei unterschiedliche Techniken, die auch zu unterschiedlichen Sprachklassen führen.

Ein SLR(1)- (Simple LR(1)) Parser verwendet die next-State Funktion des Automaten $A_0(G)$. Ist ein Konflikt-Zustand gegeben, dann wird zur Bestimmung der Aktion (action-Funktion) das Lookahead-Symbol betrachtet. Wenn alle Konflikt-Zustände des Automaten $A_0(G)$ auf diese Weise zu eindeutigen Aktionen führen, heißt die Grammatik G eine *SLR(1)-Grammatik*. Als Unterschied zu einem LR(0)-Parser wird also nur eine Funktion

$$\lambda : S \times T \to \mathcal{P}(\mathrm{Act})$$

neu definiert:

Sei also $A_0(G)$ gegeben und $s \in S$ und $t \in T$ dann ist $\lambda(s,t)$ definiert durch:

1. $(\alpha, v.) \in s \wedge t \in \mathrm{Follow}(\alpha) \Rightarrow (\alpha, v) \in \lambda(s,t)$,
2. $(\alpha, v_1.tv_2) \in s \Rightarrow \mathrm{shift} \in \lambda(s,t)$,
3. $\lambda(s,t)$ ist die kleinste Teilmenge von Act mit 1. und 2.

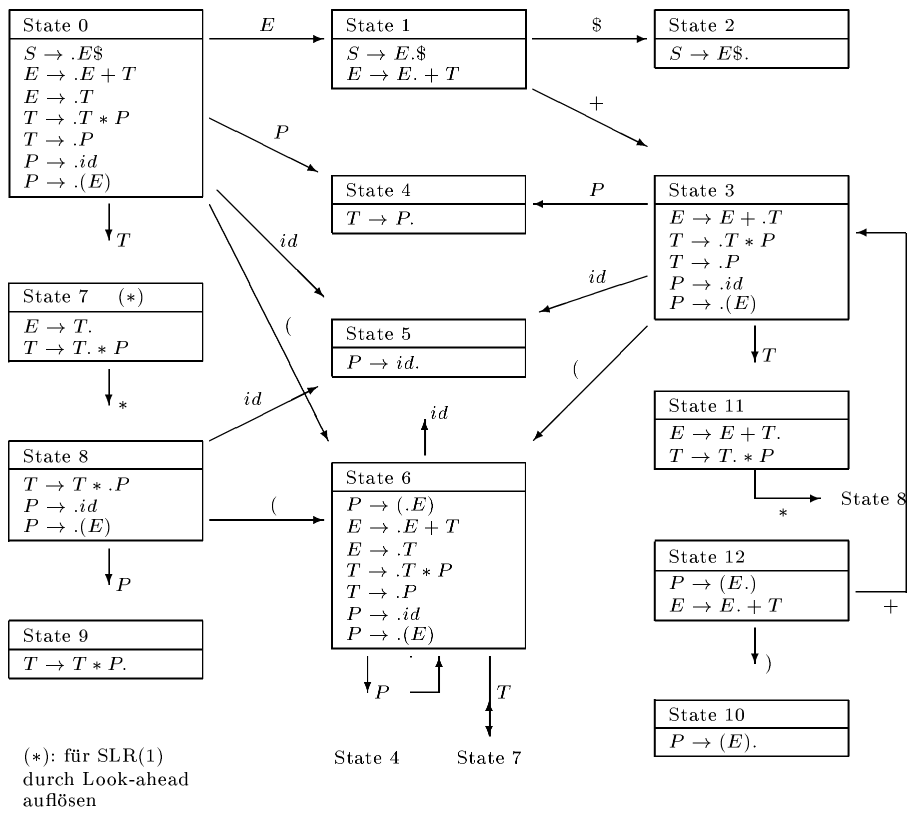
G ist also eine SLR(1)-Grammatik genau dann wenn:

$$\forall (s,t) \in S \times T : |\lambda(s,t)| \leq 1.$$

Für eine SLR(1)-Grammatik erhält man so die action-Funktion wie für eine LR(1)-Grammatik:

$$\mathrm{action}(s,t) = \begin{cases} \text{accept} & \text{falls } \lambda(s,t) = \{\text{shift}\}; t = \# \\ \text{shift} & \text{falls } \lambda(s,t) = \{\text{shift}\}; t \neq \# \\ (\alpha, v) & \text{falls } \lambda(s,t) = \{(\alpha, v)\} \\ \text{error} & \text{falls } \lambda(s,t) = \emptyset \end{cases}.$$

Beispiel:(Fortsetzung, nach Fisher, LeBlanc) Grammatik wie im LR(1)-Beispiel. Es gilt die gleiche Bemerkung wie im LR(1)-Beispiel.



Ende Beispiel

Beim LALR- (Look Ahead LR(1)) Parser verwendet man zwar auch einen Automaten $\hat{A}_0(G)$, der als Graph mit dem Automaten $A_0(G)$ übereinstimmt ($A_0(G)$ und $\hat{A}_0(G)$ sind isomorph); aber die Zustände von $\hat{A}_0(G)$ sind Teilmengen von $K_1(G)$, also 1-Konfigurationen. Wegen $K_1 = K_0 \times T_\epsilon$ gibt es die Projektion auf die erste Komponente:

$$p : K_0 \times T_\epsilon \to K_0; p(c;t) = c.$$

Sei $S^{(i)}$ $(i = 0, 1)$ die Zustandsmenge des Automaten $A_i(G)$, dann ist für $s \in S^{(1)} : p(s) = \{p(\overline{c}) | \overline{c} \in s\}$. Man sagt auch $p(\overline{c}) \in K_0$ ist der Kern der 1-Konfiguration $\overline{c} \in K_1$ und $p(s)$ ist die Menge der Kerne der 1-Konfigurationen aus s. $A_1(G)$ enthält nun verschiedene Zustände, die die gleichen Kern-Mengen besitzen. Diese werden unter der Abbildung p also auf die gleiche Menge abgebildet. Wenn man diese Zustände aus $A_1(G)$ zusammenfaßt, so erhält man einen Automaten, der zu $A_0(G)$ isomorph ist, dessen Zustände aber Mengen von 1-Konfigurationen sind wie in $A_1(G)$.

Diese Konstruktion kann man auch so beschreiben, daß man den Zuständen von $A_0(G)$ die entsprechende Menge von 1-Konfigurationen zuordnet:

$$q : S^{(0)} \to \mathcal{P}(K_1),$$

$$q(s) = \bigcup_{p(\overline{s})=s, \overline{s} \in S^{(1)}} \overline{s}.$$

Man erhält $\hat{A}_0(G)$ indem man jeden Zustand $s \in S^{(0)}$ durch den Zustand $q(s) \subseteq K_1$ ersetzt. Jeder Zustand enthält jetzt die Lookahead Information von K_1 und somit können die Aktionen davon abhängig gemacht werden: Man definiert

$$\lambda : S^{(0)} \times T \to \mathcal{P}(\mathrm{Act})$$

durch

1. $(\alpha, v.;t) \in q(s) \Rightarrow (\alpha, v) \in \lambda(s,t)$,
2. $(\alpha, v_1.tv_2) \in s \Rightarrow \mathrm{shift} \in \lambda(s,t)$,
3. $\lambda(s,t)$ ist minimal mit 1. und 2.

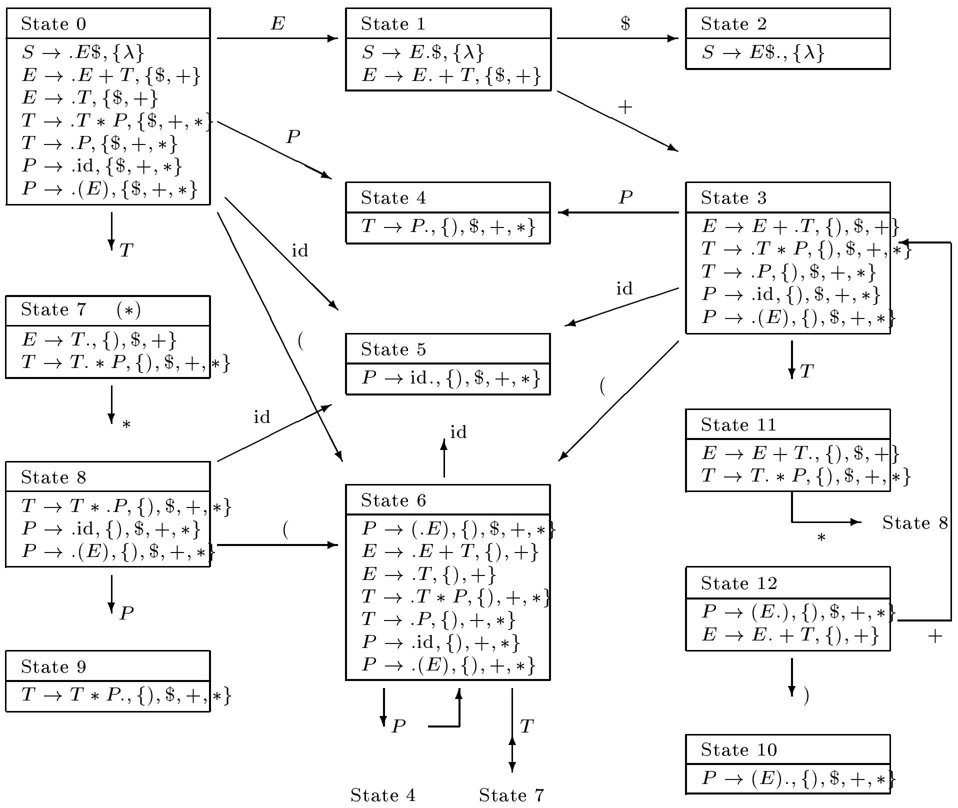
Eine Grammatik G heißt *LALR(1)-Grammatik* genau dann, wenn gilt:

$$\forall (s,t) \in S^{(0)} \times T : |\lambda(s,t)| \leq 1.$$

Für eine LALR(1)-Grammatik definiert man die action-Funktion:

$$\mathrm{action}(s,t) = \begin{cases} \text{accept} & \text{falls } \lambda(s,t) = \{\text{shift}\}; t = \# \\ \text{shift} & \text{falls } \lambda(s,t) = \{\text{shift}\}; t \neq \# \\ (\alpha, v) & \text{falls } \lambda(s,t) = \{(\alpha, v)\} \\ \text{error} & \text{falls } \lambda(s,t) = \emptyset \end{cases} .$$

Beispiel:(Fortsetzung):



Die action-Funktion ist die gleiche, wie für den SLR(1)-Parser, da die Grammatik bereits eine SLR(1)-Grammatik ist. `Ende Beispiel`

Das folgende einfache Beispiel zeigt eine Grammatik G, die LR(1) ist, aber nicht LALR(1):

Beispiel: G ist gegeben durch

$S \to (E)|[E],$

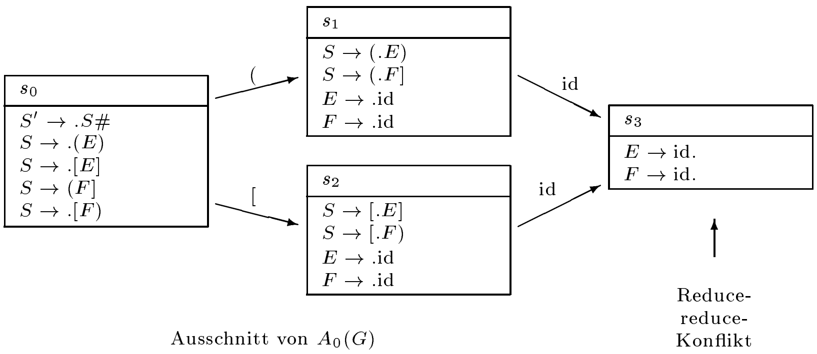
$S \to (F]|[F),$

$E \to \mathrm{id},$

$F \to \mathrm{id}.$

$L(G)$ ist endlich (und damit auch regulär):

$$L(G) = \{(\mathrm{id}), [\mathrm{id}], (\mathrm{id}], [\mathrm{id})\}.$$



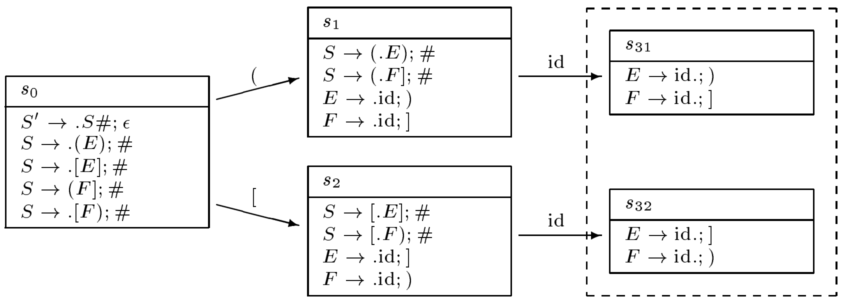
Ausschnitt von $A_0(G)$

Wenn man jetzt die 1-Konfigurationen zu s_3 betrachtet, dann ist

$$q(s_3) = \{(E, \mathrm{id}.;)), (E, \mathrm{id}.;]), (F, \mathrm{id}.;)), (F, \mathrm{id}.;])\}$$

d.h. durch das Lookahead-Symbol ist der Konflikt in s_3 auch jetzt nicht aufzulösen, und somit ist G nicht $LALR(1)$.

Betrachtet man aber $A_1(G)$, dann enthält man für s_3 zwei verschiedene Zustände:



Ausschnitt von $A_0(G)$

Diese beiden Zustände in A_1
werden in A_0 identifiziert zu s_3

Die Grammatik G ist somit LR(1), aber nicht LALR(1). `Ende Beispiel`

Die Sprachklassen zu diesen Grammatik-Typen bilden eine echte Hierarchie:

$$\begin{array}{c} \mathcal{L}_{LR(1)} \\ | \\ \mathcal{L}_{LALR(1)} \\ | \\ \mathcal{L}_{SLR(1)} \\ | \\ \mathcal{L}_{LR(0)} \end{array}$$

4.3 Der Compiler Compiler YACC

YACC (<u>Y</u>et <u>a</u>nother <u>C</u>ompiler <u>C</u>ompiler) ist ein Programm, das eine YACC-Spezifikation liest und daraus einen tabellengesteuerten Bottom-up Parser als C-Quellprogramm erzeugt. Die Spezifikation folgt ähnlichen syntaktischen Regeln wie LEX, aber YACC akzeptiert die formale Beschreibung einer kontextfreien Grammatik. Es werden die next_state-Funktion und die Funktion λ nach den Regeln für einen LALR(1)-Parser erzeugt. Wenn λ funktional ist, dann ist die vorgelegte Grammatik LALR(1) und es wird ein Shift-reduce-Parser erzeugt. Ansonsten berichtet YACC über bestehende Konflikte.

Die Konflikte werden von YACC allerdings nach bestimmten Regeln aufgelöst. Der Anwender muß daher die angezeigten Konflikte genau untersuchen und entscheiden, ob die Auflösung durch YACC seinen Vorstellungen entspricht. Andernfalls muß die Grammatik geeignet modifiziert werden. Es ist üblich, YACC zusammen mit LEX zu verwenden, wobei LEX die lexikalische Analyse übernimmmt und an den von YACC erzeugten Parser einen Tokenstrom übergibt. Auf Einzelheiten der Syntax kann hier nicht ausführlich eingegangen werden (siehe dazu: Anhang YACC oder das Buch von Herold).

An dem Beispiel auf der folgenden Seite sollen nur einige Eigenschaften der Spezifikationssprache erläutert werden.

Die Spezifikation ist in drei Abschnitte gegliedert:

%{

C-Deklarationen und Definitionen

%}

YACC-Definitionen

%%

Regelteil

%%

C-Programmteil

Im C-Deklarationsteil werden globale Variablen, Typen und Funktionen deklariert.

YACC-Definitionen legen die Token fest und charakterisieren Präzedenz und Assoziativität von Operatoren.
%left '+' '−' bedeutet also daß diese Operatoren in ungeklammerten Ausdrücken von links nach rechts angewendet werden.
%right UNARYMINUS bedeutet, daß das am weitesten rechtsstehende unäre Minuszeichen zuerst angewendet wird.

Der Regelteil enthält eine angereicherte EBNF-Form einer kontextfreien Grammatik. Nach jeder Produktion kann hier in {...} eine C-Anweisung stehen, die ausgeführt wird, wenn nach dieser Produktion reduziert wird.

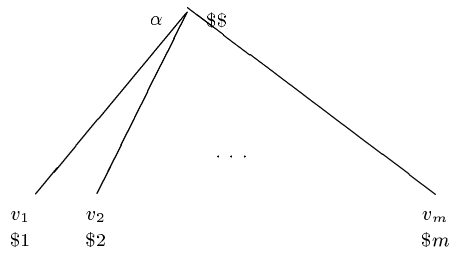
```
%{
/* Ein einfacher Tischrechner für arithmetische Ausdrücke;
YACC-Format nach Aho, Sethi, Ullman */
#define YYSTYPE double
%}
%token NUMBER
%left '+' '−'
%left '*' '/'
%right UNARYMINUS
%%
list : /* epsilon */
    | list '\n'
    | list expr '\n' {printf("\t%8g\n", $2);}
;
expr : expr '+' expr {$$ = $1 + $3;}
    | expr '-' expr {$$ = $1 - $3;}
    | expr '*' expr {$$ = $1 * $3;}
    | expr '/' expr {$$ = $1 / $3;}
```

```
    | '(' expr ')' {$$ = $2;}
    | '-' expr
    %prec UNARYMINUS {$$ = - $2;}
    | NUMBER {$$ = $1;}
    ;
%%
#include <stdio.h>
#include <ctype.h>
char *progname;
int lineno = 1;
main ( int argc, char *argv[]){
    progname = argv [0];
    yyparse();
}
yylex(){
  int c;
  while ((c = getchar()) == ' ' || c == '\t');
  if (c == EOF)
  return 0;
  if (c == ',' || isdigit(c)){
  ungetc(c, stdin);
  scanf("%lf", &yylval);
  return NUMBER;
  }
  if (c == '\ n')
  lineno++;
  return c;
}
yyerror(char *s){
  fprintf(stderr, "%s: %s", progname, s);
  fprintf(stderr, "near line %d\n", lineno);
}
```

In diesen Anweisungen tauchen spezielle Variablenbezeichnungen auf wie: $\$\$$, $\$1$, $\$2$... Diese Variablen haben stets den Typ, der im Deklarationsteil durch YYSTYPE definiert wird. Im Beispiel ist das double. Meist ist das aber ein strukturierter union-Typ. Ist nun

$$\alpha \to v_1 v_2 ... v_m \in P$$

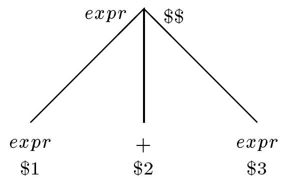
eine Produktion, dann bezeichnet in der zugehörigen Anweisung (Aktion in {...}) $\$\$$ eine Variable, die man als Attribut von α im Ableitungsbaum auffassen kann. Entsprechend bezeichnet $\$i$ das zugehörige Attribut für v_i:



Im Beispiel ist jedes Attribut vom Typ double, und die Anweisung (Aktion)

$$expr \to expr' +' expr\{\$\$ = \$1 + \$3;\}$$

bewirkt, daß das Attribut von expr auf der linken Seite den Wert der Summe der Attribute von expr und expr auf der rechten Seite erhält:



Der C-Programmteil ist aufgebaut wie eine Datei für einen C-Quelltext. Darin werden dann Funktionen definiert, die im Regelteil aufgerufen werden können.

Im Beispiel wird hier nur eine main-Funktion definiert, die dann die von YACC erzeugte Funktion yyparse() aufruft. Der einfache Scanner wird hier direkt (d.h. ohne LEX) als Funktion yylex() in C programmiert. Er hat i.W. die Aufgabe, Zahlen zu erkennen (NUMBER) und die speziellen Zeichen, die direkt Token sind, wie '+', '−', '∗' und '/'. Außerdem zählt er in einer globalen Variablen (lineno) die Zeilen für die Fehlerlokalisierung. Wesentlich ist die Zeile

scanf("%lf'", & yylval).

yylval ist eine Variable vom Typ YYSTYPE und erhält den Attributwert für den jeweils erkannten Token. Damit erhalten auch die Blätter des Ableitungsbaumes ihre Attributwerte. Die Funktion yyerror(char ∗) wird vom Parser aufgerufen, wenn der Parser einen Fehler erkennt. Es wird dann eine einfache Fehlermeldung ausgegeben, die die Zeilennummer ausgibt.

Das Begleitprojekt „Pl0-Compiler“ stellt eine umfangreiche Übung zum Umgang mit LEX und YACC dar.

4.4 Fehlerbehandlung

Je nachdem, wo ein Fehler erkannt werden kann, unterteilt man die Menge der Fehler in folgende Klassen:

- Lexikalische Fehler (kein passendes Muster).
- Syntaktische Fehler (kein Wort der Sprache).
- Statisch semantische Fehler (Typ-Fehler, Verwendung: Namen).
- Dynamisch semantische Fehler (Laufzeit).

Die Fehler der ersten drei Klassen muß der Compiler erkennen, geeignet analysieren und anzeigen.

Laufzeitfehler, die nicht aus den ersten drei Klassen verursacht werden, kann der Compiler nicht erkennen, aber er kann geeigneten Zusatz-Code erzeugen, der bestimmte Laufzeitfehler zur Laufzeit entdeckt und dann einen geeigneten Kommentar ausgibt. Dazu gehören Bereichsüberschreitungen oder Zugriffe auf nicht vorhandene Dateien u.s.w.

Der Compiler kann allerdings in bestimmten Fällen Warnungen ausgeben wenn zwar kein Fehler der ersten drei Klassen vorliegt, aber ein Fehler der vierten Klasse erwartet werden kann, wie z.B. die Verwendung nicht initialisierter Variablen.

Die Fehlermeldungen des Compilers sollten für den Programmierer so ausführlich und verständlich sein wie möglich. Man vergleiche dazu etwa die drei folgenden Meldungen:

- Zeile 43; Spalte 62: Semikolon erwartet,
- Shift in Zustand 346 nicht möglich,
- Syntaxerror! Compilation aborted!

Im Buch von Aho, Sethi und Ullmann [1] findet man eine Untersuchung über die Statistik von Programmierfehlern. Danach sind bei routinierten Programmierern (Studenten!)

60% der Programme syntaktisch korrekt.

Rest:

80% ein Fehler,

13% zwei Fehler,

7% mehr als zwei Fehler.

90 % der syntaktischen Fehler betreffen dabei genau ein Zeichen.

Die Fehler bei der lexikalischen Analyse werden durch den endlichen Automaten für die Mustererkennung entdeckt, wenn ein Eingabezeichen keinen weiteren Zustandsübergang mehr zulässt. Die sogenannte Panik-Methode versetzt den Automaten dann in den Anfangszustand und versucht, das nächste mögliche Muster zu erkennen.

Eine aufwendigere Methode besteht darin, daß man versucht, den Zeichenstrom lokal an der Fehlerstelle zu korrigieren durch geeignete Transformationen der Eingabe

- Löschen eines Zeichens.

- Einfügen eines neuen Zeichens.
- Ersetzen eines Zeichens.
- Vertauschen benachbarter Zeichen.

Eine solche lokale Reparatur kann aus mehreren Transformationen bestehen. Diejenige Reparatur, die mit den wenigsten Schritten wieder ein erkennbares Muster (genauer: Lexem!) liefert, nennt man dann auch Fehlerkorrektur mit minimalem Abstand (analog zu fehlerkorrigierenden Codes). Die Methode wird aber i.a. aus Effizienzgründen nicht angewendet. Man kann bei der lexikalischen Analyse auch ein weiteres Muster „error“ als Komplement zu allen anderen Mustern einführen. Das ist dann ebenfalls eine reguläre Menge und kann somit durch einen speziellen Zustand des Automaten erkannt werden. In diesem Fall erhält der Parser den Token „error“ und muß dann die Fehlerbehandlung organisieren.

Es soll daher jetzt kurz auf mögliche Fehlerbehandlungs-Strategien des Parsers eingegangen werden. Diese Strategien sind heuristisch motiviert und stellen nicht optimale Methoden dar. Die Fehlerbehandlung soll so organisiert werden, daß der Analyseprozeß an einer geeigneten Stelle im Text wieder aufgenommen werden kann (error-recovery, Erholung von einem Fehler).

Bei der Top-down-Analyse wurde bereits die Panik-Methode erwähnt. Bei dieser Methode werden Token der Eingabe solange „überlesen“, bis ein synchronisierendes Token gefunden wird. Synchronisierende Token sind dabei meist solche, die eine syntaktische Konstruktion abschließen, wie z.B. ein Semikolon in der Sprache C. Synchronisierende Token werden so gewählt, daß man verläßliche Annahmen machen kann über die Fortsetzung der Analyse. Auf Details zur Auswahl der synchronisierenden Token und der Aktionen die auszuführen sind nach ihrer Entdeckung kann aber hier nicht weiter eingegangen werden.

Bei der lokalen Fehlerkorrektur werden lokale Transformationen im Tokenstrom vorgenommen so daß mit möglichst wenigen solcher Transformationen wieder ein analysierbarer Abschnitt im Tokenstrom entsteht. Die Transformationen sind ganz analog zur Fehlerbehandlung bei der lexikalischen Analyse: Einfügen, Löschen, Ersetzen und Vertauschen von Token. Allerdings muß man dabei beachten, daß eine solche Folge von Transformationen auch abbricht, denn beim Einfügen wird die ursprüngliche Folge von Token verlängert, so daß man in eine Endlos-Schleife gelangen könnte. Die lokale Fehlerreparatur ist sehr aufwendig und außerdem wenig wirksam, wenn die Fehlerursache viel weiter vorne im Programmtext liegt, wie z.B. bei einer fehlenden geöffneten Klammer.

Eine interessante Methode ist die Erweiterung der gegebenen Grammatik um spezielle Fehlerproduktionen, so daß bestimmte Fehler dann durch eine ganz normale Reduktion erkannt werden und durch die zugehörige Aktion geeignet behandelt werden können. In diesem Fall kann ein sehr genauer Kommentar für die Fehlerursache gegeben werden. Die Analyse ist in ihrem Fortgang dann gar nicht durch den Fehler betroffen. Der Entwurf dieser Fehlerproduktionen ist allerdings nicht einfach. Es fließen hier statistische Erhebungen über häufig auftretende Fehler ein.

Am aufwendigsten ist eine globale Fehlerreparatur. Man versucht dabei, das Quellprogramm durch elementare Transformationen auf der Basis von bestimmten Heuristiken so zu transformieren, daß ein syntaktisch korrektes Programm entsteht. Dabei versucht man immer diejenige Reparatur mit der kleinsten Zahl von elementaren Transformationen zu finden. Das Verfahren ist nicht nur extrem aufwendig, sondern es ist auch fraglich, ob nach solchen Reparaturen die intendierte Semantik des Programmes überhaupt noch einigermaßen getroffen wird.

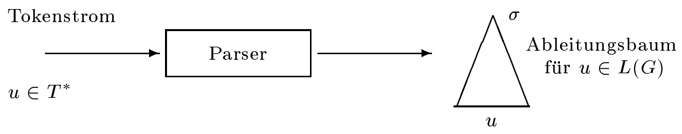
Meistens verwendet man in der Praxis die Panik-Methode, die bei geringem Aufwand meistens auch ganz brauchbare Ergebnisse liefert.

Kapitel 5

Syntaxgesteuerte Übersetzung und attributierte Grammatiken

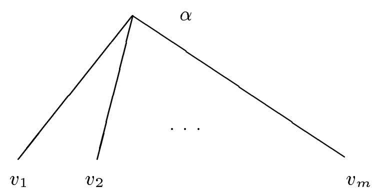
5.1 Einführung

Der Parser erzeugt einen Ableitungsbaum oder zumindest eine äquivalente Darstellung wie z.B. eine R-Ableitung.



Die Struktur des Ableitungsbaumes enthält die wesentliche „semantische Information“ für die Übersetzung. Die Übersetzung orientiert sich i.W. an der Struktur dieses Baumes. Sie wird durch die syntaktische Struktur gesteuert.

Zu diesem Zweck werden die Knoten des Baumes mit gewissen Attributwerten versehen und es werden lokale semantische Regeln (Aktionen) festgelegt. Lokal bedeutet hier, daß sich eine Regel immer nur auf einen Teil des Ableitungsbaumes bezieht, der durch eine Produktion der Grammatik markiert ist:



$(\alpha, v_1 ... v_m) \in P$

Die semantischen Regeln werden daher für die einzelnen Produktionen der Grammatik $G = (N, T, P, \sigma)$ definiert. Diese Regeln können funktionale Beziehungen zwischen bestimmten Attributwerten festlegen. Sie können aber allgemein auch Prozeduren mit Seiteneffekten sein. Semantische Regeln können etwa schon direkt den Code für eine (abstrakte oder konkrete) Zielmaschine generieren.

Die semantischen Regeln (Aktionen) und ihre gesteuerte Ausführung leisten also bereits wesentliche Übersetzungsarbeit für den Syntheseteil eines Compilers. Diese Regeln verwenden auch wesentlich die Eintragungen der Symboltabelle.

Wir wollen hier einen formalisierten Zugang zur Steuerung der Übersetzung betrachten, der logisch auf einer vollständigen Darstellung des Ableitungsbaumes beruht, aber natürlich in konkreten Implementationen stärker mit dem Parser verzahnt werden kann.

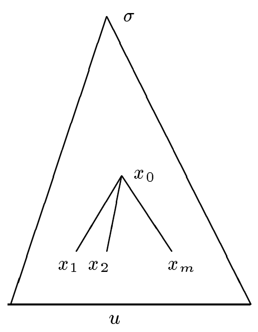
Es werden zwei sehr ähnliche Formen der syntaxgesteuerten Übersetzung verwendet:

- Syntaxgesteuerte Definitionen.
- Übersetzungsschemata.

Die erste Form ist dabei etwas abstrakter und geht daher nicht so stark auf die konkrete Implementation ein, während Übersetzungsschemata schon recht nah an der Implementation orientiert sind.

5.2 Attributierte Grammatiken

Mit den Produktionen einer kontextfreien Grammatik kann man bestimmte, lokal begrenzte semantische Regeln assoziieren. Man betrachtet dazu den Ableitungsbaum (B, μ) für ein Wort $u \in L(G)$ mit $G = (N, T, P, \sigma)$.



Die semantischen Regeln beziehen sich auf einen solchen lokalen Bereich im Ableitungsbaum, der gerade durch eine Produktion $(x_0,\, x_1...x_m) \in P$ markiert ist.

Dabei ist $x_0 \in N$ und $x_1...x_m \in V^*$. x_0 ist Markierung eines Knotens $b \in B$: $\mu(b) = x_0$ und es ist $\mu\varphi(b) = x_1...x_m \in V^*$. Es kann also verschiedene lokale Bereiche geben, die mit der gleichen Produktion markiert sind.

Ein Attribut ist das gleiche wie eine Variable in Programmiersprachen. Ein Attribut hat einen Namen und einen Datentyp (einfach Typ) und kann Werte des zugeordneten Typs annehmen. Wir bezeichnen ein solches Attribut auch kurz durch ein Paar:

$$a : s,$$

wobei a der Name und s der Typ des Attributes ist. Eine Menge solcher Attribute

$$\{a_1 : s_1, a_2 : s_2, ..., a_k : s_k\}$$

entspricht dann einem Record-Typ (struct). Eine solche Menge von Attributen ist selbst wieder ein Typ (Record-Typ):

$$s = \{a_1 : s_1, a_2 : s_2, ..., a_k : s_k\}.$$

Sei nun $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik. Dann definiert man für jedes $x \in V = N \cup T$ eine Attributmenge:

$$s_x = \{a_1^x : s_1^x, ..., a_{k_x}^x : s_{k_x}^x\}$$

d.h. man ordnet jedem Symbol $x \in V$ einen geeigneten Record-Typ zu. s_x beschreibt die semantische Information, die ein Symbol im Ableitungsbaum tragen kann als Datenstruktur.

Semantische Regeln stellen lokale Abhängigkeiten zwischen den Attributwerten einer Produktion her. Sei also $(x_0, x_1...x_m) \in P$ eine Produktion der Grammatik G. Wir betrachten zunächst jedes Symbol x_i in dieser Produktion als eine Variable des Typs s_{x_i}, also $x_i : s_{x_i}$ $(0 \leq i \leq m)$.

Eine semantische Regel definiert einen der Attributwerte als Funktion der anderen

$$x_i.a_j = f(Y_1.a_{j_1}, ..., Y_r.a_{j_r}). \qquad (*)$$

Mit $Y_t \in \{x_0, x_1, ..., x_m\}$. Die hochgestellten Indizes an den Attributnamen wurden weggelassen, da sie jeweils durch den Variablennamen bestimmt sind. Der Attributwert darf dabei nicht von sich selbst abhängen: $x_i.a_j$ kommt nicht in der Argumentliste von f vor. f kann allgemeiner als eine Funktion im Sinne der (imperativen) Programmierung betrachtet werden, d.h. f kann Seiteneffekte haben, wie z.B. die Ausgabe von Code-Zeilen für eine Zielmaschine. Wenn f keine Seiteneffekte hat, also eine Funktion im mathematischen Sinne ist, dann heißt f *Berechnungsregel*.

In der Berechnungsregel $(*)$ sagt man auch, daß das Attribut $x_i.a_j$ von den Attributen $Y_1.a_{j_1}, ..., Y_r.a_{j_r}$ abhängt und $x_i.a_j$ heißt das *abhängige Attribut* dieser Regel.

Definition 5.2.1 *Eine* syntaxgesteuerte Definition *(*SD*) besteht aus:*

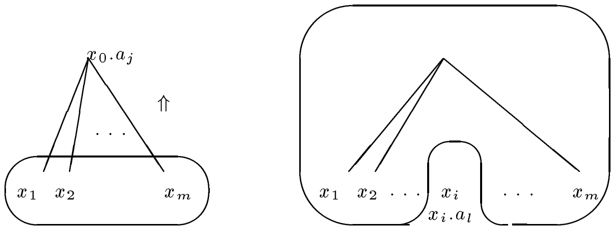
1. *Einer kontextfreien Grammatik* $G = (N, T, P, \sigma)$.
2. *Einer endlichen Menge* A *von Attributnamen und einer Typ-Zuordnung* $\tau : A \to S$*, wobei* S *eine Menge von Datentypen ist. Für* $\tau(a) = s$ *schreibt man* $a : s$.
3. *Einer Zuordnung, die jedem* $x \in V$ *eine endliche Attributmenge zuordnet:*

   $$s_x = \{a_1^x : s_1^x, ..., a_{k_x}^x : s_{k_x}^x\}$$

   s_x *ist disjunkt zerlegt in synthetische und geerbte Attribute.*
4. *Zu jeder Produktion* $(x_0, x_1...x_m) \in P$ *gibt es semantische Regeln, in denen nur Attribute der Symbole* $x_0, x_1, ..., x_m$ *vorkommen, und es gilt:*
   - *Zu jedem synthetischen Attribut von* x_0 *gibt es genau eine Regel, in der es abhängig ist.*
   - *Zu jedem geerbten Attribut der* x_j $(1 \leq j \leq m)$ *gibt es genau eine Regel, in der es abhängig ist.*

Die terminalen Symbole haben nur synthetische Attribute, die ihre Werte extern erhalten (i.a. durch den Scanner) und σ hat keine geerbten Attribute.

Synthetische Attribute für die linke Seite einer Produktion $x_0 \to x_1...x_m$ werden also stets aus den Attributwerten der rechten Seiten berechnet.



Ein geerbtes Attribut eines Symboles auf der rechten Seite kann von Attributwerten seiner „Geschwister“ und der linken Seite („Vater“) abhängen.

Eine syntaxgesteuerte Definition heißt eine *attributierte Grammatik*, wenn alle semantischen Regeln ohne Seiteneffekte sind.

Zunächst wieder das Standard-Beispiel für arithmetische Ausdrücke. Dieses Beispiel wurde bereits als YACC-Beispiel weiter oben angegeben.

Beispiel:

$E \to T; E.\text{val} = T.\text{val},$

$E_0 \to E_1 + T; E_0.\text{val} = E_1.\text{val} + T.\text{val},$

$T \to F; T.\text{val} = F.\text{val},$

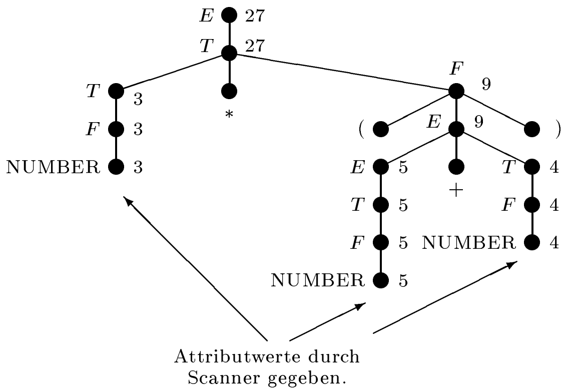
$T_0 \to T_1 * F; T_0.\text{val} = T_1.\text{val} * F.\text{val},$

$F \to (E); F.\text{val} = E.\text{val},$

$F \to \text{NUMBER}; F.\text{val} = \text{NUMBER.val}$ (Wert vom Scanner).

Es gibt nur ein Attribut val : *int* und jedes Symbol besitzt dieses Attribut. Die Indizes in den Produktionen sind jetzt notwendig wenn ein Symbol in einer Produktion mehrmals auftritt.

Man betrachtet jetzt einen Ableitungsbaum für den Ausdruck $3 * (5 + 4)$:



Die Attribute sind alle synthetisch und die Werte an jedem Knoten wurden durch die semantischen Regeln bestimmt. `Ende Beispiel`

Das Beispiel zeigt, wie diese semantischen Regeln verwendet werden. Das Ziel der syntaxgesteuerten Übersetzung ist es, eine Bewertung aller Attribute zu bestimmen, so daß alle (lokalen) Regeln erfüllt werden. Wenn alle Attribute synthetisch sind, dann ist das recht einfach. Die Werte für die Blätter (terminale Symbole) werden als gegeben vorausgesetzt und jeder Attributwert eines Vaterknotens im Baum wird nach einer eindeutig bestimmten Regel aus Attributwerten der Söhne berechnet. Das entspricht der üblichen Auswertung von algebraischen Termen. Wenn aber auch geerbte Attribute vorkommen, dann kann es sein, daß zunächst bestimmte synthetische Attribute „bottom-up“ berechnet werden müssen und erst danach können geerbte Attribute „top-down“ berechnet werden. Es wird also darauf ankommen, die Reihenfolge der Ausführung der semantischen Regeln geeignet festzulegen.

Sind alle Attribute konsistent mit den Regeln bewertet, dann sagt man auch, daß ein dekorierter (oder mit Attributwerten annotierter) Ableitungsbaum vorliegt.

Es soll jetzt noch ein Beispiel betrachtet werden, in dem synthetische und geerbte Attribute vorkommen.

Beispiel:[Bewertung von Bit-Strings als Dualzahl] Sei $b_{n-1}b_{n-2}...b_0$ eine Folge von Bits, d.h. $b_i \in \{0,1\}$ und die *Bewertung* ist:

$$\mathrm{val}(b_{n-1}b_{n-2}...b_0) = \sum_{i=0}^{n-1} b_i 2^i.$$

Diese Bewertung soll durch eine attributierte Grammatik berechnet werden. Jedes Bit (binäre Ziffer) soll dabei mit dem Gewicht 2^{pos} zur Bewertung (Summe) beitragen, wobei pos die Position im Bit-String ist.

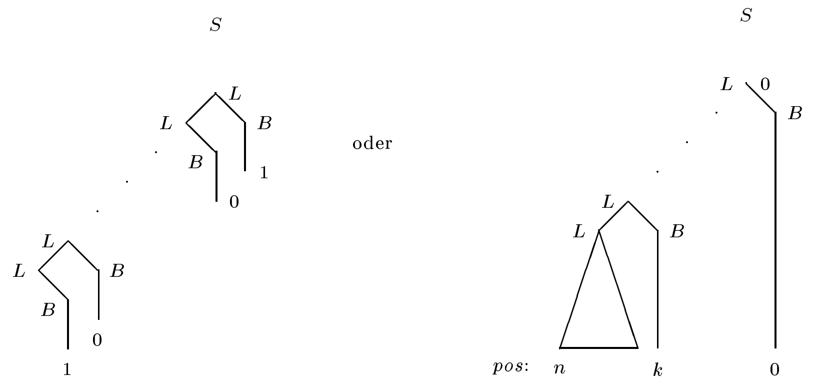
Grammatik zur Erzeugung der Strings (Folgen):

$S \to L,$

$L \to B|LB,$

$B \to 0|1.$

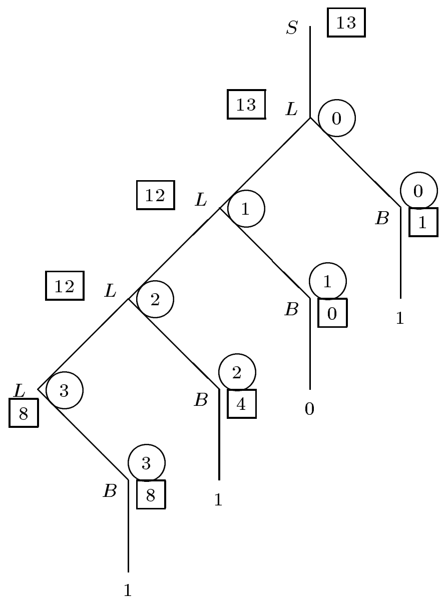
Die Ableitungsbäume haben folgende Struktur:



pos ist ein Attribut für $L, B \in \mathbb{N}$ und val ein Attribut für alle Symbole aus $V = \{S, L, B, 0, 1\}$. pos für B soll die Position im Bitstring sein und pos für L die niederwertigste Position im String, der von L erzeugt wird. Damit erhält man folgende attributierte Grammatik:
$A = \{\text{val}, \text{pos}\}$ mit val : *int*; pos : *int*
val: synthetisch; pos: geerbt

1. $S \to L$; $S.\text{val} = L.\text{val}, L.\text{pos} = 0$,
2. $L \to L_1 B$; $L.\text{val} = L_1.\text{val} + B.\text{val}$, $L_1.\text{pos} = L.\text{pos} + 1$, $B.\text{pos} = L.\text{pos}$,
3. $L \to B$; $L.\text{val} = B.\text{val}$, $B.\text{pos} = L.\text{pos}$,
4. $B \to 0$; $B.\text{val} = 0$,
5. $B \to 1$; $B.\text{val} = 2^{B.\text{pos}}$.

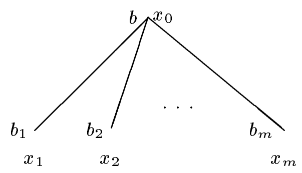
Damit soll ein Ableitungsbaum dekoriert werden: pos : ○; val : □



Es wird zuerst das Attribut pos „top-down“ berechnet: Mit 1. erhält man den ersten Wert dieses Attributes für den am weitesten oben stehenden Knoten, der mit L markiert ist. Mit 2. durchläuft man dann die „L-Kette“ nach unten und erhöht dabei jeweils das pos-Attribut um 1. Die B-Knoten erben das pos-Attribut der L-Knoten. Danach werden die val-Attribute „bottom-up“ ausgewertet. An jedem B-Knoten wird die zugehörige Binärziffer mit $2^{B.\text{pos}}$ multipliziert (nach den Produktionen 4. und 5.). Die L-Kette summiert dann (nach den Produktionen 2. und 3.) die „gewichteten“ Bits auf. `Ende Beispiel`

Das Beispiel zeigt, daß die semantischen Regeln (Funktionen) im Ableitungsbaum in einer ganz bestimmten Reihenfolge ausgeführt werden müssen.

Sei (B, μ) ein Ableitungsbaum. Für $b \in B$ ist $\mu(b) = x_0 \in V$ und $\varphi(b) = b_1 ... b_m \in B^*$ ist die Folge der Söhne von b und $\mu\varphi(b) = x_1 ... x_m \in V^*$ ist die Markierung der Söhne:



Wir betrachten jeden Knoten $b \in B$ als eine Variable vom Typ $s_{x_0} = s_{\mu(b)} = \{a_1 : s_1, ..., a_k : s_k\}$.

Sei $a : s \in s_{\mu(b)} = s_{x_0}$ und a synthetisch, dann gibt es genau eine Regel:

$$b.a = f(b_{j_1}.a_{j_1}, ..., b_{j_r}.a_{j_r}).$$

Dabei sind die $j_i \in \{1, ..., m\}$. Der Wert $b.a$ kann also erst dann berechnet werden, wenn die Attributwerte $b_{j_1}.a_{j_1}, ..., b_{j_r}.a_{j_r}$ der Söhne bereits feststehen. In diesem Fall geht der „Informationsfluß" von unten (den Söhnen) nach oben (zum Vater). Daher werden synthetische Attribute bottom-up berechnet.

Ist dagegen $a : s \in s_{\mu(b_i)}$ ein geerbtes Attribut eines Sohnknoten von b, dann gibt es wieder genau eine semantische Regel

$$b_i.a = g(b_{l_1}.a_{l_1}, ..., b_{l_q}.a_{l_q}).$$

Dabei sind die $b_{l_i} \in \{b, b_1, ..., b_m\}$ $(1 \leq i \leq q)$, d.h. $b_i.a$ hängt ab von den Attributwerten der Nachbarknoten von b_i, also sowohl von linken und rechten Nachbarn, wie auch vom Vaterknoten. Bevor $b_i.a$ berechnet werden kann, müssen alle diese Attributwerte vorliegen. Das läuft auf eine Sortierung der einzelnen Attribute hinaus.

Sei $\mathcal{A}(B, \mu) = \{b.a | b \in B \wedge a : s \in s_{\mu(b)}\}$ die Menge aller in B annotierten Attribute und Φ eine Abbildung, die jedem Attribut $b.a \in \mathcal{A}(B, \mu)$ einen Wert $\Phi(b.a)$ vom Typ s für $a : s \in s_{\mu(b)}$ zuordnet, so daß alle semantischen Funktionen „erfüllt" sind:

$$\Phi(b.a) = g(\Phi(b_{j_1}.a_{j_1}), ..., \Phi(b_{j_r}.a_{j_r})) \qquad (*)$$

(g ist die zu $b.a$ gehörige Funktion). Φ heißt eine *vollständige, konsistente Attributbewertung.* Sind die Bedingungen $(*)$ stets erfüllt, falls alle Argumente von g definiert sind, dann heißt eine partielle Bewertung *konsistent.* Die (synthetischen) Attribute der Blätter des Baumes werden dabei als gegeben vorausgesetzt.

Der folgende Algorithmus liefert zumindest eine konsistente Bewertung:

1. Φ_0 sei die partielle Funktion, die die Attribute der Blätter bewertet.

2. Ist $b.a = g(b_{j_1}.a_{j_1}, ..., b_{j_r}.a_{j_r})$ die semantische Regel für $b.a$ und sind alle $b_{j_i}.a_{j_i}$ $(1 \leq i \leq r)$ durch Φ bewertet, dann sei $\Phi'(b.a) = g(\Phi(b_{j_1}.a_{j_1}), ..., \Phi(b_{j_r}.a_{j_r}))$ und für $b'.a'$ sei $\Phi'(b'.a') = \Phi(b'.a')$.

3. Wiederhole 2. solange, bis kein weiteres Attribut mehr berechnet werden kann.

Dieser Algorithmus liefert eine vollständige Bewertung, wenn es für die Attributwerte keine zyklischen Abhängigkeiten gibt. Ob solche Zyklen vorliegen, läßt sich algorithmisch testen. Dazu definiert man für die Attribute $b.a \in \mathcal{A}(\mathcal{B}, \mu)$ eine Abhängigkeits-Relation E, die man auch als Di-Graph betrachten kann

$$(b_{j_i}.a_{j_i}, b.a) \in E \iff b.a = g(b_{j_1}.a_{j_1}, ..., b_{j_r}.a_{j_r}) \wedge (1 \leq i \leq r).$$

Man kann dafür die gerichtete Kante zeichnen:

$$b_{j_i}.a_{j_i} \to b.a.$$

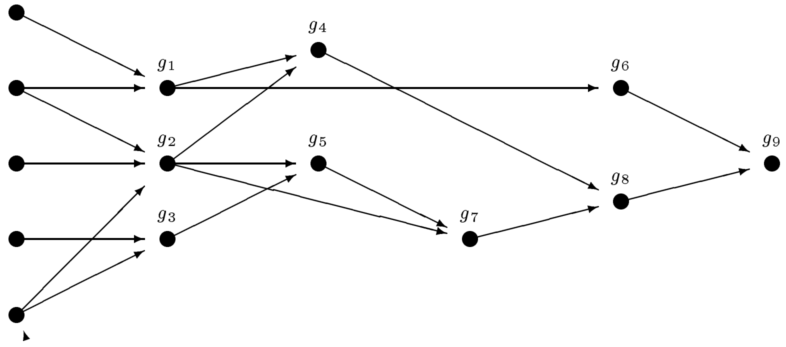
Es ist klar, daß die Reihenfolge der Bewertung einzelner Attribute mit dieser Relation konsistent sein muß:

Wenn gilt: $b_j.a_j \to b_i.a_i$

dann muß $b_j.a_j$ vor $b_i.a_i$ bewertet werden.

Die (synthetischen) Attribute der Blattknoten haben bezüglich dieser Relation keine Vorgänger. Ihre Bewertung wird als gegeben vorausgesetzt. Die Menge $\mathcal{A}(\mathcal{B},\mu)$ ist endlich und somit ist der Abhängigkeitsgraph E endlich. Daher kann er auf Zyklen getestet werden.

Wir setzen jetzt voraus, daß der Abhängigkeitsgraph keine Zyklen besitzt. Da zu jedem Attribut $b.a \in \mathcal{A}(\mathcal{B},\mu)$ eine Funktion g gehört, die den Wert von $b.a$ berechnet, kann man diese Funktionen an die Knoten des Graphen schreiben:



Um die Funktionen in eine lineare Ordnung zu bringen, in der man sie berechnen kann so daß eine vollständige Bewertung entsteht, benötigt man eine ordnungstreue Abbildung des Abhängigkeitsgraphen auf eine (gleichmächtige) lineare Ordnung. Eine solche Abbildung (bzw. einen Algorithmus dafür) heißt *Topologische Sortierung* (*Top-Sort*). Jede solche Sortierung ordnet die Funktionen linear so an, daß aus den Anfangswerten für die Blatt-Attribute sämtliche anderen Attribute konsistent berechnet werden. Algorithmen für das topologische Sortieren findet man in vielen Büchern über Algorithmen.

Beispiel: Eine Deklaration von Variablen des gleichen Typs hat etwa in der Programmiersprache C die folgende syntaktische Struktur (hier vereinfacht):

$\underline{\text{int}}$ x, y, z

Die Variablennamen bilden eine Liste und der vorangestellte Typbezeichner ist dann jeder Variablen der Liste zuzuordnen. Syntaxgesteuerte Definition:

$D \to TL \quad L.in = T.type,$

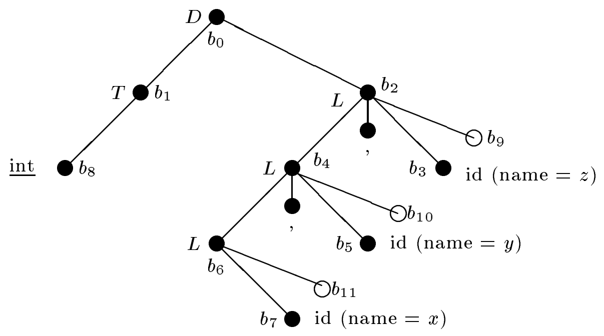
$T \to int \quad T.type = integer,$

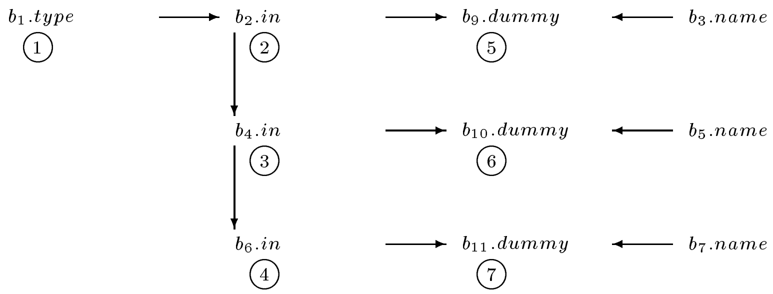
$T \rightarrow real \quad T.type = real,$

$L \rightarrow \mathrm{id} \quad insert(\mathrm{id}.name, L.in),$

$L \rightarrow L_1, \mathrm{id} \quad L_1.in = L.in; insert(\mathrm{id}.name, L.in).$

Die Funktion $insert(\mathrm{id}.name, L.in)$ soll einen Eintrag in die Symboltabelle ausführen, bei dem $\mathrm{id}.name$ (der Bezeichner der Variablen) zusammen mit dem Typbezeichner der gesamten Liste ($L.in$) eingetragen wird: An den Variablennamen $\mathrm{id}.name$ wird der in der Deklaration auftretende Typbezeichner ($L.in$) gebunden. Diese Funktion liefert keinen Funktionswert an ein Attribut. Man führt für solche Funktionen ein Schein-Attribut ein: $dummy = insert(\mathrm{id}.name, L.in)$. Das Scheinattribut wird i.a. nicht explizit erwähnt; aber es muß im Abhängigkeitsgraphen berücksichtigt werden:





Die Scheinknoten wurden immer da eingefügt, wo für eine Regelanwendung der Grammatik ein Scheinattribut berechnet (ausgeführt) werden muß. Die durch 1, 2, ... angegebene Reihenfolge der Attributbewertung ist aus dem Abhängigkeitsgraphen durch topologisches Sortieren entstanden.

`Ende Beispiel`

Scheinattribute werden also da notwendig, wo nur eine bestimmte Aktion mit Seiteneffekten ausgeführt werden soll, wobei aber kein Funktionswert für ein echtes Attribut berechnet wird. Ein solches Scheinattribut wird dann an einen zusätzlich eingefügten Scheinknoten im Ableitungsbaum annotiert.

Der gesamte Vorgang der Attributbewertung kann nochmals zusammengefaßt werden:

- Konstruktion des Abhängigkeitsgraphen $(\mathcal{A}(B,\mu), E)$ zum Ableitungsbaum (B,μ).
- Test auf Zyklen in diesem Graphen.
- Bei Zyklenfreiheit: Topologisches Sortieren.

- Ausführen der semantischen Funktionen in dieser Reihenfolge.

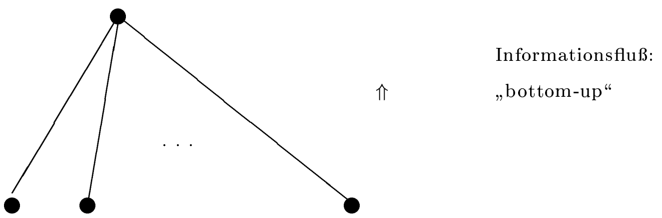
Dieses Verfahren ist prinzipiell stets möglich, aber doch recht aufwendig. Es erfordert mehrere Arbeitsgänge nach der Erstellung des Ableitungsbaumes.

Wenn man die Abhängigkeiten der Attributwerte geeignet einschränkt, kann das Verfahren direkt in die Syntaxanalyse integriert werden, so daß kein zweiter Arbeitsgang nach der Erstellung des Ableitungsbaumes erforderlich ist. Man betrachtet daher:

- S-attributierte Definitionen (Grammatiken).
- L-attributierte Definitionen (Grammatiken).

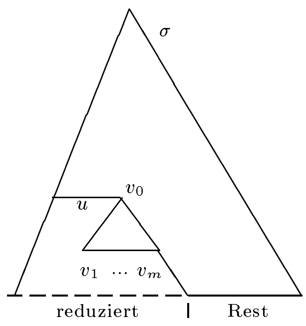
Definition 5.2.2 *Eine syntaxgesteuerte Definition heißt* S-attributiert $\iff$ *jedes Attribut ist synthetisch.*

Die Abhängigkeit von Attributwerten nennt man manchmal auch „Informationsfluß", und dieser hat bei S-attributierten Definitionen immer die Richtung: von den Sohnknoten zum Vaterknoten.



$x.a = f(x_{i_1}.a_{i_1}, ..., x_{i_r}.a_{i_r})$ für $i_1, ..., i_r \in \{1, ..., m\}$

Für S-attributierte Definitionen kann die Attributbewertung zusammen mit der bottom-up Syntaxanalyse erfolgen. Das kann man sehr einfach einsehen, wenn man an die Situationen eines Kellerautomaten denkt:



Bei diesem Reduktionsschritt werden die Elemente $v_1...v_m$ vom Keller entfernt und durch die linke Seite v_0 ersetzt. Wenn man die Attributwerte zu jedem Symbol $x \in V$ mit auf dem Keller ablegt, dann sind also bei diesem Reduktionsschritt sämtliche Attributwerte der „Söhne" $v_1....v_m$ schon berechnet und man kann dann nach Voraussetzung alle Attributwerte des Vaterknotens v_0 berechnen und diese zusammen mit v_0 auf dem Keller ablegen.

Mit der Voraussetzung, daß die Attributwerte der terminalen Symbole (Token) stets schon gegeben sind ergibt sich somit die vollständige Attributbewertung während der bottom-up Analyse.

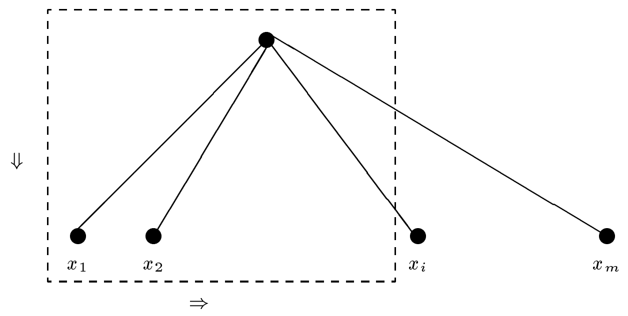
S-attributierte Definitionen sind gut geeignet, um arithmetische Ausdrücke auszuwerten. Andere, kontextsensitive Beziehungen, wie z.B. die typgerechte Verwendung von Bezeichnern lassen sich aber besser durch geerbte Attribute ausdrücken.

L-attributierte Definitionen lassen auch geerbte Attribute zu, sind aber auch so eingeschränkt, daß die Attributbewertung zusammen mit der Syntaxanalyse erfolgen kann.

Definition 5.2.3 *Eine syntaxgesteuerte Definition heißt* L-attributiert *genau dann, wenn gilt: Ist* $x_i.a_i$ *ein geerbtes Attribut von* x_i *und* $(x_0, x_1 ... x_m) \in P$ $(1 \leq i \leq m)$ *und* $x_i.a_i = g(x_{j_1}.a_{j_1}, ..., x_{j_r}.a_{j_r})$ *die semantische Regel für* $x_i.a_i$*, dann gilt:*

1. $\forall 1 \leq k \leq r : 0 \leq j_k < i$ *und*
2. *wenn* $j_k = 0$*, dann ist* $x_{j_k}.a_{j_k}$ *ein geerbtes Attribut von* x_0.

Der Informationsfluß für geerbte Attribute ist also von oben nach unten und nur von links nach rechts:



Geerbte Attibute von x_i hängen nur ab von Attributen der $x_1, ..., x_{i-1}$ und von geerbten Attributen von x_0.

Für L-attributierte Definitionen gibt es einen einfachen Algorithmus zur Sequentialisierung der semantischen Regeln, der eine Traversierung des Ableitungsbaumes verwendet. Die Tiefensuche (Depth First Search) oder DFS für einen geordneten Baum wird wie folgt rekursiv definiert:

Algorithmus DFS (für geordnete Bäume):

Gegeben: Geordneter Baum $B = (b_0, B_1, ..., B_m)$

Gesucht: Systematischer Besuch jedes Knotens

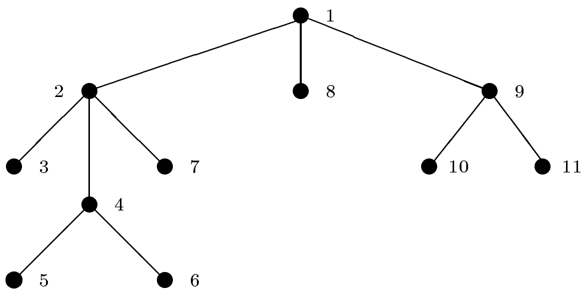
von B genau einmal (Traversierung)

DFS(B):

besuche b_0;

for $i := 1$ to m do DFS(B_i)

Beispiel:



Die Knoten werden hier in der Reihenfolge ihrer Numerierung besucht. `Ende Beispiel`

Die Söhne eines Knotens werden bei dieser Reihenfolge stets von links nach rechts besucht, wie man unmittelbar an der for-Schleife erkennt. Man kann diesen Algorithmus nun leicht erweitern, um die Attributbewertung von L-attributierten Ableitungsbäumen vorzunehmen.

Algorithmus DFV (Attributbewertung von Ableitungsbäumen zu

L-attributierten Definitionen)

Gegeben: Ableitungsbaum (B, μ) und beliebiger Knoten $b \in B$ mit $\varphi(b) = b_1 ... b_m$.

Für b seien die geerbten Attribute bereits bewertet.

Gesucht: Attributbewertung des Teilbaums T_b, der von b erzeugt wird.

DFV(b):

<u>begin</u>

<u>for</u> $i := 1$ <u>to</u> m <u>do</u>

<u>begin</u>

bewerte die geerbten Attribute von b_i;

DFV(b_i)

<u>end</u> ;

bewerte die synthetischen Attribute von b

<u>end</u>

Diese Bewertung der Attribute kann man in den Syntaxanalyse-Prozeß sowohl eines LR(1)- als auch eines LL(1)-Parsers integrieren. L-attributierte syntaxgesteuerte Definitionen sind daher für die praktischen Anwendungen von großer Bedeutung. Sie stellen einen guten Kompromiß dar zwischen Ausdrucksstärke und Effizienz.

5.3 Übersetzungsschemata

Übersetzungsschemata (ÜS) basieren ebenfalls auf dem Begriff der attributierten Grammatik und sind den syntaxgesteuerten Definitionen sehr verwandt. Die semantischen Regeln (Aktionen) können jetzt aber beliebig in die rechten Seiten von Produktionen eingefügt werden. Damit legt ein ÜS die Reihenfolge der Anwendung der semantischen Regeln eindeutig fest.

Sei $G = (N, T, P, \sigma)$ eine kontextfreie Grammatik und F eine Menge von semantischen Funktionen. Wie bei syntaxgesteuerten Definitionen sei zu jedem Symbol $x \in V = N \cup T$ ein Recordtyp s_x definiert. Die semantischen Funktionen definieren auch hier Beziehungen zwischen Attributwerten innerhalb des lokalen Bereiches von Produktionen von G. Auch hier werden geerbte und synthetische Attribute unterschieden. Soweit ist alles definiert wie für syntaxgesteuerte Definitionen. Bei einem ÜS werden die Produktionen (α, v) einfach erweitert durch Einfügen von semantischen Funktionen $f \in F$ in die rechten Seiten. Eine F-erweiterte Produktion ist ein Paar $(\alpha, v') \in N \times (V + F)^*$ so daß man nach Streichen von allen Elementen aus F eine Produktion $(\alpha, v) \in P$ entsteht.

Definition 5.3.1 *Ein Übersetzungsschema besteht aus einer kontextfreien Grammatik $G = (N, T, P, \sigma)$, einer Menge F von semantischen Funktionen und einer F-Erweiterung P' von P. Die semantischen Funktionen sind so definiert wie bei syntaxgesteuerten Definitionen.*

Ein Übersetzungsschema definiert zu jedem Eingabewort $u \in L(G)$ eine Folge von Funktionsausführungen wie folgt:

1. Es wird der Ableitungsbaum für u berechnet.
2. Die semantischen Funktionen werden als zusätzliche Blattknoten in diesen Baum eingefügt, so daß ein Ableitungsbaum der erweiterten Grammatik $G' = (N,\, T,\, P',\, \sigma)$ entsteht.
3. Der so erweiterte Ableitungsbaum wird mit dem Algorithmus DFS durchlaufen.
4. Immer wenn ein Knoten b besucht wird mit $\mu(b) \in F$, dann wird die Funktion $\mu(b)$ ausgeführt.

Dabei ist allerdings zu beachten, daß die Abhängigkeiten der Attribute mit dem DFS-Durchlauf verträglich sind. Das ist stets der Fall, wenn die Abhängigkeiten den Regeln der L-attributierten Definitionen genügen: Bei der Ausführung einer semantischen Funktion müssen alle Attributwerte, die als Argumente vorkommen, schon gegeben sein.

YACC akzeptiert solche erweiterten Produktionen. Dabei werden die Aktionen (C-Anweisungen) einfach in {}-Klammern eingefügt und in die rechten Seiten eingesetzt. Dabei ist die Numerierung der Referenzen zu beachten: Z.B.:

x_0	$\to$	x_1	$\{g_1\}$	x_3	$\{g_2\}$	x_4	x_5	erw. Produktion
\$\$		\$1	\$2	\$3	\$4	\$5	\$6	Referenzen

Beispiel:[Umsetzung von Infix-Ausdrücken zu Postfix-Ausdrücken] Übersetzungsschema:

$E \to TR,$

$R \to op\, T\{g_1\}R_1|\epsilon,$

$T \to num\{g_2\}.$

Token sind: *op* und *num*.

op hat das Attribut *name*.

num hat das Attribut *value*.

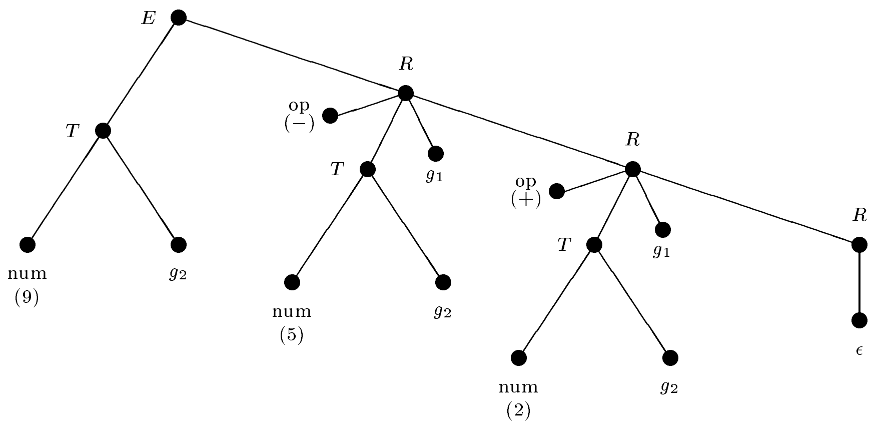
Die Attributwerte für Token liegen nach Voraussetzung vor.

Die semantischen Funktionen sind:

g_1: $print(op.name)$,

g_2: $print(num.value)$.

Sie beziehen sich nur auf Attributwerte von Token. Erweiterter Ableitungsbaum für Eingabe $9 - 5 + 2$:



Die DFS-Folge liefert den Ausdruck (Print)

$$9 \quad 5 \quad - \quad 2 \quad + \,.$$

`Ende Beispiel`

Kapitel 6

Semantische Analyse und Erzeugung von Zwischencode

Die typischen Aufgaben der semantischen Analyse sind:

- Typüberprüfung.
- Bezeichneridentifikation.
- Erzeugung von Zwischencode.

Diese Aufgaben werden durch die syntaxgesteuerte Übersetzung und z.T. durch bestimmte Funktionen der Symboltabelle erledigt. Man kann diese Aufgaben durch eine attributierte Grammatik spezifizieren und durch geeignete Attributauswerte-Algorithmen ausführen. Für die semantische Analyse besitzt man somit auch gewisse Standardverfahren. Diese lassen sich in bestimmten Fällen auch vollständig in die Syntaxanalyse integrieren.

6.1 Typüberprüfung

Wir behandeln zunächst die Typenüberprüfung. In einer Wertzuweisung

$$e_1 := e_2$$

sind e_1 und e_2 Ausdrücke. Der Wert von e_1 wird als L-Wert bestimmt, d.h. e_1 muß einen Speicherplatz bezeichnen für Werte eines Typs T_1. Der Wert von e_2 dagegen wird als R-Wert bestimmt, d.h. als ein Wert vom Typ T_2. Dabei könnte e_2 durchaus auch einen L-Wert haben (Speicherplatz vom Typ T_2); aber da e_2 auf der rechten Seite der Wertzuweisung steht wird dann der Inhalt dieses Speicherplatzes als Wert bestimmt. Durch die Stellung in der Wertzuweisung werden beide Ausdrücke ausgewertet.

Beispiel: Der Ausdruck $3+5$ hat den R-Wert 8 vom Typ integer; aber $3+5$ hat keinen L-Wert. Ist x Variable vom Typ integer, dann hat x einen L-Wert und einen R-Wert, nämlich den Speicherplatz (vom Typ integer) und den Inhalt dieses Speicherplatzes.

Ist a Variable vom Typ <u>array</u> [1..10] <u>of</u> <u>integer</u> , dann hat der Ausdruck $a[3 + 5]$ einen L-Wert und einen R-Wert. `Ende Beispiel`

In der Wertzuweisung $e_1 := e_2$ muß der L-Wert von e_1 ein Speicherplatz sein für Werte vom Typ T_2. In einer if - then - else Anweisung

$$\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$$

muß der Typ des Ausdrucks e_1 (zumindest für die Sprache PASCAL) *boolean* sein. In der Sprache C sind an dieser Stelle auch *integer*, *char* und *pointer* zugelassen, wobei der Wert 0 jeweils den Booleschen Wert false und jeder andere Wert den Wert true darstellt.

In dem Ausdruck $a + x$ sind für a bzw. x unterschiedliche Typen zugelassen. Es muß nur möglich sein, diese mit einer geeigneten Rechenoperation „+“ zu verknüpfen: z.B.

$$a : integer,\ x : real.$$

Beide Operanden werden als *real* betrachtet und auch so addiert. Das Ergebnis ist *real*.

Alle Regeln dieser Art machen das Typsystem einer Programmiersprache aus. Es gibt Programmiersprachen, die den Datentyp von Variablen nicht zur Übersetzungszeit (statische Typbindung) sondern erst zur Laufzeit (dynamisch Typbindung) binden, wie z.B. Smalltalk. Hier können zur Laufzeit Fehler auftreten, die in der Unverträglichkeit von Datentypen begründet sind. Eine Programmiersprache heißt *stark getypt* oder besitzt ein *starkes Typsystem*, wenn alle möglichen Unverträglichkeiten von Datentypen zur Übersetzungszeit entdeckt werden können (z.B. PASCAL).

Ein Typsystem besteht aus gewissen Basistypen (z.B. {*integer*, *real*, *boolean*, *char*}) und aus Konstruktoren, die es erlauben, aus vorhandenen Typen neue strukturierte Typen zu konstruieren (z.B. *array*, *record*, *set*...).

Eine solche Konstruktion wird durch einen Typausdruck spezifiziert:

- Die Namen $\beta_1, ..., \beta_m$ für die Basistypen sind Typausdrücke.
- Seien $\tau_1, ..., \tau_k$ Typausdrücke und Φ ein Konstruktor, der auf $\tau_1, ..., \tau_k$ anwendbar ist, dann ist auch $\Phi(\tau_1, ..., \tau_k)$ ein Typausdruck.
- Ist α ein Bezeichner und τ ein Typausdruck, dann ist $\alpha = \tau$ eine Typdefinition und α kann in Typausdrücken verwendet werden wie τ.

Typausdrücke bezeichnen also Datentypen und es kann daher unterschiedliche Typausdrücke geben, die den gleichen Datentyp bezeichnen.

Beispiel:[PASCAL]

```
type s1 = array [1..10] of char ;
     s2 = array [1..5] of s1;
     s3 = array [1..5] of array [1..10] of char ;
     s4 = array [1..5, 1..10] of char ;
```

Die Typausdrücke

s2 und array [1..5] of s1

sind syntaktisch verschieden, aber sie bezeichnen den gleichen Typ (per Definition). Die rechten Seiten von s2 und s3 sind syntaktisch verschieden aber bezeichnen den gleichen Typ. Die rechten Seiten von s3 und s4 sind syntaktisch verschieden und bezeichnen auch zwei verschiedene Typen, die aber inhaltlich gleichwertig sind. `Ende Beispiel`

Diese Beispiele sollten zeigen, daß die Definitionen für die Äquivalenz von Typausdrücken nicht ganz unproblematisch ist. Man unterscheidet daher die Prüfung auf syntaktische Gleichheit der Typausdrücke und auf strukturelle Gleichheit. Zwei Typausdrücke sollen dabei *strukturell gleich* (*äquivalent*) heißen, wenn sie syntaktisch gleich sind, nachdem sämtliche definierten Typbezeichner durch die rechte Seite ihrer Definition ersetzt wurden.

Beispiel:[Fortsetzung] Im obigen Beispiel sind s2 und s3 syntaktisch verschieden, aber nach Ersetzung aller Namen erhält man

$$\text{s2} = \underline{\text{array}}\ [1..5]\ \underline{\text{of}}\ \text{s1} = \underline{\text{array}}\ [1..5]\ \underline{\text{of}}\ \underline{\text{array}}\ [1..10]\ \underline{\text{of}}\ \underline{\text{char}} = \text{s3}$$

s2 und s3 lassen sich auf die gleiche volle expandierte (Normal-) Form transformieren und sind daher strukturell gleich. `Ende Beispiel`

Es ist natürlich viel aufwendiger, die strukturelle Gleichheit zu überprüfen, als die rein syntaktische.

Beispiele für Typkonstruktoren sind:

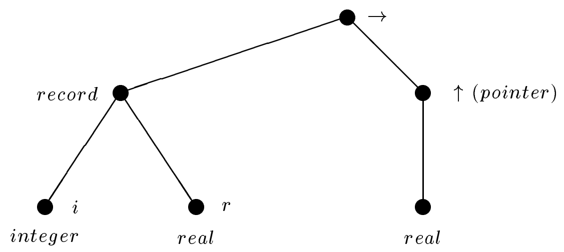
- Aufzählungstyp: $(a_1, a_2, ..., a_m)$.
- Array: τ_1 ordinaler Typ, τ_2 (fast) beliebig: $\underline{\text{array}}\ [\tau_1]\ \underline{\text{of}}\ \tau_2$, „abstrakter“: array($\tau_1$, τ_2)
- Record: $\tau_1, ..., \tau_m$ Typausdrücke.
  $n_1, ..., n_m$ Bezeichner.
  $\underline{\text{record}}\ n_1 : \tau_1;$
  $\vdots$
  $n_m : \tau_m \quad \underline{\text{end}}$
  oder record $(n_1 : \tau_1, ..., n_m : \tau_m)$
- Pointer (Zeiger) τ Typausdruck
  $\uparrow \tau$ (Zeiger auf Typ τ).

Man kann auch die Funktionen von einem Typ T_1 in den Typ T_2, also die Menge $[T_1 \to T_2]$ als neuen (funktionalen) Typ betrachten. So könnte man auch einen entsprechenden Typausdruck: $\tau_1 \to \tau_2$ für beliebige Typausdrücke τ_1 und τ_2 einführen.

Für die meisten strukturierten Typen existieren Selektoren. Das sind Funktionen vom strukturierten Typ auf die Komponententypen. Für Zeigervariablen var x : $\uparrow$ T gibt es die Dereferenzierung: x $\uparrow$ ist eine Variable vom Typ T.

Typausdrücke sind strukturell genauso aufgebaut wie die Terme einer Algebra und können daher auch durch Bäume dargestellt werden, deren Blätter dann die (Bezeichner der) Basistypen sind.

Beispiel: $\tau = record(i : integer \times r : real) \to \uparrow real$ wird durch folgenden Baum dargestellt:



Ende Beispiel

Man führt einen speziellen Typ ein, um damit Typ-Fehler zu bezeichnen: type_error. Da Typausdrücke auch definierte Namen enthalten können, kann man diese auch an den Blättern von Typ-Bäumen verwenden. Ein Typbaum, bei dem sämtliche Blätter Basistypen bezeichnen heißt *vollständig entwickelt*. Der strukturelle Typvergleich vergleicht jeweils die vollständig entwickelten Typbäume. Um nun die Typinformation im Ableitungsbaum zu berücksichtigen, kann man den Knoten ein Attribut type zuordnen. Der Wert dieses Attributes ist dann ein Typausdruck oder ein Typbaum, den man geeignet darstellen muß.

Die Typüberprüfung geschieht dann durch geeignete semantische Regeln. Das folgende ausführliche Beispiel soll das prinzipielle Vorgehen verdeutlichen.

Beispiel:[für eine syntaxgesteuerte Definition zur Typüberprüfung einer sehr einfachen Sprache (nach [1])] **Teil 1:** Produktionen für Deklarationen von Variablen mit Typ und einfachen Ausdrücken. Ein „Programm" besteht zunächst nur aus einem Deklarationsteil D und einem Ausdruck E

$P \to D; E,$

$D \to D; D,$

id : $T\{$add_type(id.$entry, T.type)\}$.

add_type ist eine Funktion, die ein Element der Symboltabelle aufsucht und dort den Attributwert für einen Typausdruck (oder Baum) einfügt.

id.*entry* ist ein Attribut, das den Symboltabelleneintrag für id liefert.

$T.type$ ist der Typausdruck der zum nichtterminalen Symbol T gehört.

T stellt entweder einen elementaren Typ (Basistyp) dar, einen Array-Typ oder einen Zeigertyp:

$$T \to char\{T.type = char\},$$
$$T \to integer\{T.type = integer\},$$
$$T \to array[num]of T_1\{T.type = array(1..num.val, T_1.type)\},$$
$$T \to\uparrow T_1\{T.type = pointer(T_1.type)\}.$$

Bei der Konstruktion von Ausdrücken kann jetzt die Typverträglichkeit überprüft werden. Bei unverträglichen Typen erhält der Gesamtausdruck den Typ: type_error. Der Token *lit* steht für ein Druckzeichen. Die Funktion *look_up*(id.*entry*) sucht das Element der Symboltabelle und liefert den dort abgelegten Typausdruck. Es wird hier nur die arithmetische Operation mod verwendet und die Selektoren für Arrays und Dereferenzierung für Zeiger.

$E \to lit\{E.type = char\}$,

$E \to num\{E.type = integer\}$,

$E \to \mathrm{id}\{E.type = look\_up(\mathrm{id}.entry)\}$,

$E \to E_1 \bmod E_2\{E.type = \underline{\text{if}}\ E_1.type = \text{integer}\ \underline{\text{and}}\ E_2.type = \text{integer}\ \underline{\text{then}}\ \text{integer}\ \underline{\text{else}}\ \text{type\_error}\}$,

$E \to E_1[E_2]\{E.type = \underline{\text{if}}\ E_2.type = \text{integer and } E_1.type = \text{array}(S,T)\ \underline{\text{then}}\ T\ \underline{\text{else}}\ \text{type\_error}\}$,

$E \to \uparrow E_1\{E.type = \underline{\text{if}}\ E_1.type = pointer(T)\ \underline{\text{then}}\ T\ \underline{\text{else}}\ \text{type\_error}\}$.

Teil 2: Da Ausdrücke auch in Anweisungen vorkommen, muß auch dort ihre korrekte Verwendung überprüft werden. Dazu wird wieder ein spezieller Typ (Typausdruck) eingeführt: *void*. Wenn eine Anweisung (statement) den Typ *void* hat, dann ist sie typgerecht konstruiert, andernfalls erhält sie das Typattribut: *type_error*. Für *if-then* und *while*-Anweisungen nehmen wir jetzt noch boolean zu den Basis-Typen hinzu: Ein Programm hat jetzt die Form $P \to D; S$, wobei D der Deklarationsteil und S der Anweisungsteil ist. Anweisungen haben folgende Struktur:

$S \to \mathrm{id} := E\{S.type = \underline{\text{if}}\ \mathrm{id}.type = E.type\ \underline{\text{then}}\ void\ \underline{\text{else}}\ type\_error\}$,

$S \to \underline{\text{if}}\ E\ \underline{\text{then}}\ S_1\{S.type = \underline{\text{if}}\ E.type = \underline{\text{boolean}}\ \underline{\text{then}}\ S_1.type\ \underline{\text{else}}\ type\_error\}$,

$S \to \underline{\text{while}}\ E\ \underline{\text{do}}\ S_1\{S.type = \underline{\text{if}}\ E.type = \underline{\text{boolean}}\ \underline{\text{then}}\ S_1.type\ \underline{\text{else}}\ type\_error\}$,

$S \to S_1; S_2\{S.type = \underline{\text{if}}\ S_1.type = \underline{\text{void}}\ \underline{\text{and}}\ S_2.type = \underline{\text{void}}\ \underline{\text{then}}\ \underline{\text{void}}\ \underline{\text{else}}\ type\_error\}$.

Für Funktionsanwendungen kann man die folgenden Regeln hinzufügen. Sei dazu $E_1(E_2)$ die syntaktische Form der Funktionsanwendung und $T_1 \Rightarrow T_2$ der Typ von Funktionen von T_1 nach T_2:

$T \to T_1 \Rightarrow T_2\{T.type = T_1.type \to T_2.type\}$,

$E \to E_1(E_2)\{E.type = \underline{\text{if}}\ E_2.type = s\ \underline{\text{and}}\ E_1.type = s \to t\ \underline{\text{then}}\ t\ \underline{\text{else}}\ type\_error\}$.

`Ende Beispiel`

Die Typenprüfung im Beispiel verwendet die Abfrage auf Gleichheit. Dabei wurde offengelassen, ob man die syntaktische Gleichheit der Typausdrücke oder die strukturelle Gleichheit der (vollständig expandierten) Ausdrücke verwendet. Während bestimmte Operationen Typgleichheit voraussetzen, kann man ein Typsystem aber auch weniger streng festlegen. Man kann auf den Typausdrücken eine Relation $\sqsubseteq$ definieren

$$\tau_1 \sqsubseteq \tau_2,$$

die bedeuten soll, daß überall da wo ein Objekt mit R-Wert vom Typ τ_2 erwartet wird, auch ein Objekt mit R-Wert vom Typ τ_1 stehen kann.

Beispiel: In der Sprache PASCAL gilt z.B.

$$integer \sqsubseteq real.$$

An der Stelle einer *real*-Variablen, -Konstanten oder *real*-Ausdrucks allgemein kann auch ein integer-Ausdruck verwendet werden, wie z.B. in folgendem Programmstück:

```
var r: real ;
  i: integer ;
procedure foo (x: real );
   begin end ;
function bar(y: integer ): integer ;
   begin end ;
begin
  i := 1; r := 3.14 * i; r := i;
  foo(r); foo(i); r := bar(i)
end .
```

Ende Beispiel

Wenn $D(\tau)$ die Menge ist, die durch den Typausdruck τ bezeichnet wird und wenn man die Beziehung $\tau_1 \sqsubseteq \tau_2$ in einem Typsystem zuläßt, dann gibt es Funktionen

$$D(\tau_1) \stackrel{C_{\tau_1 \tau_2}}{\rightarrow} D(\tau_2)$$

die Werte vom Typ τ_1 in solche vom Typ τ_2 umformen (cast-Operation). Zu der Halbordnung der Datentypen muß eine entsprechende Menge solcher cast-Operatoren definiert sein. Wenn dann verschiedene Typen als Argumente zu einem Operator oder einer Funktion auftreten, werden (implizit) zunächst die cast-Operatoren auf die Argumente angewendet und dann der typgerechte Operator. Ein solches System der impliziten Typanpassung kann sehr komplex werden und es erfordert u.U. auch einen hohen Aufwand, die Typ-Verträglichkeit zu testen. Viele Programmiersprachen verfügen auch über *explizite cast-Operatoren*, mit denen der Programmierer die Typ-Verträglichkeit selbst herstellen kann.

Ein weiteres wichtiges Konzept in Programmiersprachen wie z.B. ADA oder C++, ist das Überladen von Operatoren. Ein bekanntes Beispiel dafür ist das Überladen etwa des Operatorsymbols „+“:

$$\begin{array}{lcl} + & : & int \times int \to int \\ + & : & real \times real \to real \\ + & : & set \times set \to set \end{array}$$

Die Bedeutung des Operatorzeichens ist eine konkrete Operation (im Rechner ein bestimmter Algorithmus, der entweder in der CPU fest „verdrahtet“ ist, oder auch eine Bibliotheksfunktion). Je nach Typ der Operanden werden aber ganz unterschiedliche Algorithmen verwendet. Die Multiplikation von Fließkommazahlen (real) ist i.a. wesentlich verschieden von der Multiplikation von Ganzzahlen.

Seien etwa p verschiedene Typkombinationen und Operatoren gegeben:

$$O_i : T_1^{(i)} \times T_2^{(i)} \to T_3^{(i)} (1 \leq i \leq p)$$

dann kann man u.U. ein einziges Symbol für alle diese Operationen verwenden, etwa ω und ω bezeichnet dann einen Operator

$$O : \sum_{i=1}^{p} \left(T_1^{(i)} \times T_2^{(i)} \right) \to \sum_{i=1}^{p} T_3^{(i)}$$

und es gilt $O(x,y) = O_i(x,y)$ falls $x \in T_1^{(i)}$ und $y \in T_2^{(i)}$. ω ist dann ein (p-fach) *überladenes Operatorsymbol* (oder man nennt ω auch einen *überladenen Operator*) und die Typen der Argumente legen für den Compiler zur Übersetzungszeit den auszuwählenden Operator (Algorithmus dafür) O_i fest. So ist in PASCAL der „+“-Operator überladen. In C++ aber kann auch der Programmierer Operatoren überladen.

Um die Mehrdeutigkeit solcher überladenen Operatoren aufzulösen, kann man eine Tabelle anlegen, die zu jeder Typ-Kombination der Argument-Ausdrücke den korrekten Operator festlegt. Eine Funktion

$$get\_type(\omega, \tau_1, \tau_2)$$

kann dann in der syntaxgesteuerten Übersetzung jeweils den korrekten Operator bestimmen.

Ein weiterer Bereich, in dem die Typenbindung eine wichtige Rolle spielt, sind die *polymorphen Funktionen*. Ist T ein Datentyp, dann ist die Menge T^* als Datentyp die Menge aller Listen mit Elementen aus T. Mathematisch definiert man nun Operationen (Funktionen) auf T^*, deren Definition vom Typ T überhaupt nicht abhängt:

$$\text{first} : T^+ \to T$$

$$\text{rest} : T^+ \to T^*$$

definiert durch

$$\text{first}(tu) = t(t \in T, u \in T^*)$$

$$\text{rest}(tu) = u(t \in T, u \in T^*)$$

(dabei ist $T^+ = TT^*$).

Die Funktionen first und rest sollten auch in der Programmierung auf diese Weise unabhängig vom Grundtyp T formulierbar sein. Die Funktionen first und rest sind *polymorph* da sie einheitlich definiert sind für verschiedene (in diesem Fall sogar für beliebige) Typen T.

In der Sprache PASCAL müßte man diese Funktionen für jeden Typ T neu definieren. Zur Realisierung eines solchen Konzeptes von polymorphen Funktionen sind einerseits *objektorientierte* Programmiersprachen mit *später Bindung* und andererseits auch Sprachen mit einem *Typ-Inferenzsystem* geeignet. Auf beide kann aber in dieser Vorlesung nicht weiter eingegangen werden.

6.2 Bezeichneridentifikation

Bezeichner werden in Programmen an vielen Stellen verwendet. Ihre Bedeutung wird in Definitionen festgelegt und bei ihrer Verwendung werden diese Definitionen benötigt. Bezeichner können durchaus mehrfach definiert werden, wenn durch geeignete *Bereichsregeln* (*scope-rules*) zu jeder Verwendung die korrekte Definition bestimmt ist.

In Block-strukturierten Sprachen gilt meist die *Verdeckungsregel*: Die Blöcke sind durch die syntaktische Struktur halbgeordnet: $B \sqsubseteq B'$ bedeutet, daß der Block B ganz innerhalb des Blockes B' liegt. Die Blöcke bilden eine Klammerstruktur und daher ist die Halbordnung sogar ein Baum. Wenn ein Name (Bezeichner) x im Block B verwendet wird, dann bestimmt man in der Halbordnung den kleinsten Block, der B umfaßt und in dem x definiert ist. Diese Definition ist dann für x gültig. Weiter darüber liegende Blöcke, in denen x ebenfalls definiert wird bleiben bei der Suche verdeckt.

Diese Bereichsregel nennt man auch die *lexikalische Bereichsregel*. Man betrachtet dabei die statische (oder textuelle, lexikalische) Anordnung der Blöcke. Im Gegensatz dazu könnte man auch die

Schachtelung der Blöcke betrachten, die zur Laufzeit durch geschachtelte Funktionsaufrufe entsteht (dynamische Bereichsregel: Z.B. LISP).

Bei der Definition eines Bezeichners werden ganz bestimmte Attribute (Attributwerte) an diesen Bezeichner gebunden. Eine solche Bindung ist formal ein Paar

$$(name, descriptor)$$

wobei der *descriptor* die Attributwerte enthält. Eine solche Bindung wird durch einen Eintrag in die Symboltabelle realisiert. Dabei werden meist variante Recordtypen verwendet, die je nach der Art des bezeichneten Objekts bestimmte Attribute enthalten, wie z.B.

- kind (Art des Objekts: Variable, Konstante, Prozedur)
- type (Typausdruck)
- address (Speicheradresse, L-Wert)
- block (ggf. Nr. des def. Blockes oder Schachtelungstiefe)

Die Attributwerte werden entweder durch den Scanner (für Token) bestimmt, oder durch die semantischen Regeln einer syntaxgesteuerten Übersetzung berechnet.

Für die Behandlung der Bereichsregeln verwendet man den Begriff der Umgebung. Eine *Umgebung* (*environment*) ist einfach eine Liste von Bindungen. Wenn ein Bezeichner verwendet wird, dann ist für ihn eine ganz bestimmte Umgebung gültig. Diese wird durch die Bereichsregeln festgelegt. Umgebungen werden durch Definition von Bezeichnern verändert.

Die lexikalische Bereichsregel kann durch eine stack-orientierte Symboltabelle realisiert werden, so wie das bereits in Kapitel 3 ausführlicher dargestellt wurde. Diese Realisierung wird auch für das Begleitprojekt vorgeschlagen. Das folgende Beispiel zeigt schematisch eine Attributierung, bei der zu einem verwendeten Bezeichner (applied_ident) in einem Ausdruck die für ihn zuständige Umgebung als Attributwert berechnet wird.

Beispiel:

$\langle$Block$\rangle.loc$: Lokale Definitionen (Bindungen),
$.glob$: Globale Definitionen (Bindungen),
$.env$: Lokale Bindungen verdecken globale Bindungen (Stack).

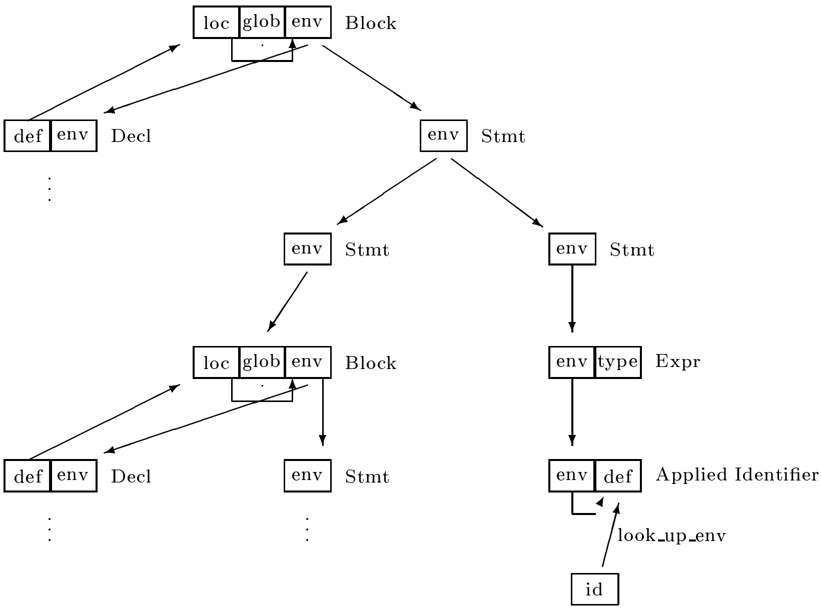
$\langle$Decl$\rangle.def$: Liste der lokalen Bindungen,
$.env$: Vom Block geerbte Umgebung zur Auswertung der Definitionen.

$\langle$stmt$\rangle.env$: Die Umgebung zur korrekten Interpretation der Bezeichner.

Die Pfeile im folgenden Diagramm zeigen die Abhängigkeiten dieser Attributwerte, also den Informationsfluß.

$\langle\text{Applied\_ident}\rangle \rightarrow \langle\text{Ident}\rangle$
$\{\langle\text{Applied\_ident}\rangle.def = \textit{look\_up\_env}(\langle\text{Ident}\rangle.name, \langle\text{Applied\_ident}\rangle.env\}$,

$\langle\text{Block}\rangle \rightarrow \langle\text{Decl}\rangle\langle\text{Stmt}\rangle$
$\{\langle\text{Block}\rangle.local = \langle\text{Decl}\rangle.def;$
$\langle\text{Block}\rangle.env = \textit{add\_env}(\langle\text{Block}\rangle.local, \langle\text{Block}\rangle.global);$
$\langle\text{Decl}\rangle.env = \langle\text{Block}\rangle.env;$
$\langle\text{Stmt}\rangle.env = \langle\text{Block}\rangle.env; \}$



Ende Beispiel

Die fehlerhafte Verwendung von Bezeichnern kann durch geeignete semantische Regeln genauer bezeichnet werden:

- Mehrfachdefinition in einem Block.
- Anwendung ohne vorhergehende Definition.

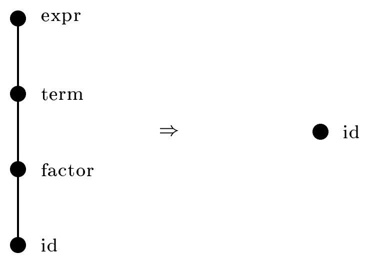
Ein Beispiel für eine solche Attributierung wurde bereits im Kapitel 1 (18) skizziert.

6.3 Erzeugung von Zwischencode

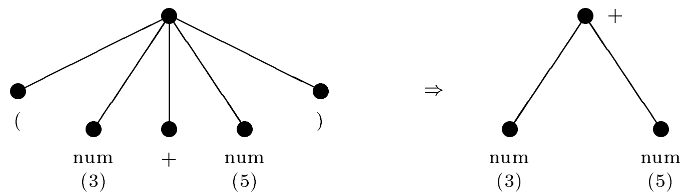
Der Ableitungsbaum zusammen mit den bewerteten Attributen enthält die semantische Information des Quellprogramms in einer neuen Form. Die Eingabe wurde auf diese Zwischenform transformiert.

Aus dieser Form (oder auch anstatt dieser Form) kann man weitere Formen ableiten, die der Erzeugung des Maschinencodes in der Zielsprache näher sind. Diese Nähe zur Zielmaschine bedeutet, daß zusammengesetzte arithmetische (oder logische) Ausdrücke zerlegt werden müssen in eine Folge von 2- oder 1-stelligen Operationen, deren Ergebnisse als Zwischenergebnisse in Speicherplätzen abgelegt werden können. Der Kontrollfluß des Quellprogramms muß auf primitive Operationen des Kontrollflusses, wie etwa bedingte und unbedingte Programmsprünge, zurückgeführt werden. Der Speicher einer realen Zielmaschine ist oft eine lineare Folge von homogenen Zellen (Byte, Wort u.s.w.). Für Variablen mit strukturiertem Datentyp muß eine geeignete Speicherabbildung systematisch erzeugt werden. Es sind verschiedene Formen eines Zwischencodes gebräuchlich.

Eine noch relativ abstrakte Form ist der abstrakte Syntaxbaum (AST). Dieser wird aus dem Ableitungsbaum gewonnen, indem man alle Knoten wegläßt, die nur notwendig waren, um die Baumstruktur eindeutig herzustellen (Parser), die aber überflüssig sind, wenn der Baum schon besteht. Dazu gehört etwa das „Zusammenziehen von Ketten“:



oder das Weglassen der Klammersymbole und die Verlagerung der Operatorsymbole an die Wurzel eines Teilbaumes:

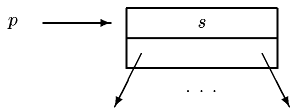


Man kann die Ausdrücke, die strukturierten Anweisungen und die Deklarationslisten alle als algebraische Ausdrücke auffassen, die man klammerfrei in die Postfix-Form überführen kann. Diese Postfix-Form definiert dann wiederum einen Baum und das ist der abstrakte Syntaxbaum.

Ein sehr ausführliches Beispiel dazu findet man im Begleitprojekt. Dort findet sich eine spezielle Version des Parsers im YACC-Format. Dieser Parser ist direkt als syntaxgesteuerte Definition mit semantischen Regeln versehen, die statt des Ableitungsbaumes direkt den abstrakten Syntaxbaum generieren. Die zugehörige Spezifikation ist auf den folgenden Seiten abgedruckt, und anschließend ist einem (sinnlosen) PL0-Programm der zugehörige AST gegenübergestellt.

Beispiel:[AST-Erzeugung für PL0-Programme] Zur Erklärung ist nur auf die Funktionen des Modules s_tree.c kurz hinzuweisen. Mit diesen Funktionen kann ein Baum systematisch (bottom-up) erzeugt werden.

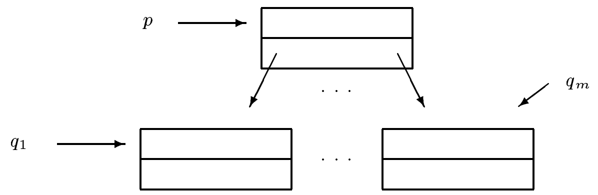
$p = make\_node(char * s)$ erzeugt einen Knoten mit der Textmarkierung s und gibt einen Zeiger darauf zurück.



Ist p ein solcher Zeiger auf einen Knoten und sind $q_1, ..., q_m$ ebensolche Zeiger, dann können diese an den Knoten p angehängt werden und zwar der Reihe nach:

$$add\_tree(p, q_1); add\_tree(p, q_2); ...add\_tree(p, q_m)$$

erzeugt dann die Verkettung:



Die Funktion *print_tree*$(p, 0)$ druckt dann die Markierungen eines solchen Baumes so aus, daß Knoten, die im Baum auf einer Ebene liegen durch die gleiche Einrückung gekennzeichnet werden. So läßt sich die Baumstruktur einfach ausdrucken. Jedes Symbol im Ableitungsbaum hat jetzt nur ein Attribut und das ist ein Zeiger auf einen Baum, nämlich den dazu gehörenden abstrakten Syntaxbaum. Der Startknoten ⟨program⟩ enthält schließlich einen Zeiger auf den vollständigen AST und dieser kann dann ausgedruckt werden. `Ende Beispiel`

```
/* PL0 Parser; YACC Format */
program : block T_PERIOD
    {$$ = make_node("program");
    add_tree($$,$1); printf("\n\n");
    print_tree($$,0);}
    ;
block :
   const_declaration_part
   var_declaration_part
   procedure_declaration_part
   statement
      {$$ = make_node("block");
      add_tree($$,$1);add_tree($$,$2);
      add_tree($$,$3);add_tree($$,$4);}
    ;
const_declaration_part :
   empty
      {$$ = make_node("const_decl_part");
      add_tree($$,$1);}
    | T_CONST const_decl_list T_SEMICOLON
```

```
      {$$ = $2;}
    ;
const_decl_list:
   const_declaration
      {$$ = make_node("const_decl"); add_tree($$,$1);}
    | const_decl_list T_COMMA const_declaration
      { add_tree($1,$3); $$ = $1); }
    ;
const_declaration :
   T_IDENT T_EQL T_NUMBER
      {$$ = make_node("const_decl");
      add_tree($$,$1); add_tree($$,$3);}
    ;
var_declaration_part :
   empty
      {$$ = make_node("var_decl_part"); add_tree($$,$1); }
   | T_VAR var_list T_SEMICOLON
      { $$ = $2; }
   ;
var_list :
   T_IDENT
      {$$ = make_node("var_list"); add_tree($$,$1);}
   | var_list T_COMMA T_IDENT
      {add_tree($1,$3);$$=$1;}
    ;
procedure_declaration_part:
   empty
      {$$ = make_node("proc_decl_part"); add_tree($$,$1);}
    | procedure_declaration_part procedure_declaration
      {add_tree($1,$2); $$=$1;}
    ;
procedure_declaration:
    T_PROCEDURE T_IDENT T_SEMICOLON block T_SEMICOLON
      {$$ = make_node("proc_decl");
      add_tree($$,$2); add_tree($$,$4);}
```

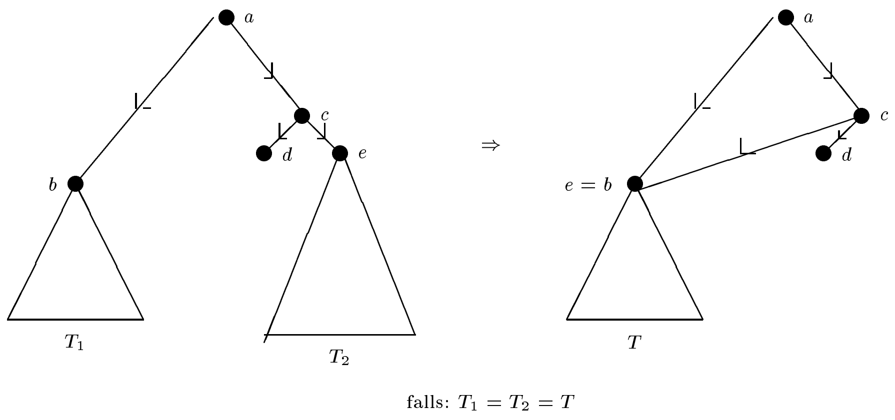
```
  ;
statement :
  empty
  | T_IDENT T_ASSIGN expression
    {$$ = make_node(“stmt“); add_tree($$,$1);add_tree($$,$3);}
  | T_CALL T_IDENT
    {$$ = make_node(“call“); add_tree($$,$2);}
  | T_READ T_IDENT
    {$$ = make_node(“read“); add_tree($$,$2);}
  | T_WRITE expression
   {$$ = make_node(“write“); add_tree($$,$2);}
  | T_BEGIN statement_sequence T_END
    {$$ = $2; }
  | T_IF condition T_THEN statement
    {$$ = make_node(“if_then“); add_tree($$,$2);add_tree($$,$4); }
  | T_WHILE condition T_DO statement
    {$$ = make_node(“while“); add_tree($$,$2);add_tree($$,$4); }
  ;
statement_sequence:
  statement
    {$$ = make_node(“seq“); add_tree($$,$1);}
  | statement_sequence T_SEMICOLON statement
    {add_tree($1,$3); $$=$1;}
  ;
condition :
  T_ODD expression
    {$$ = make_node(“odd“); add_tree($$,$2);}
  | expression comp_operator expression
    {$$ = make_node($2->symbol); add_tree($$,$1); add_tree($$,$3);}
  ;
comp_operator :
  T_EQL
    {$$ = make_node(“=“);}
  | T_NEQ
    {$$ = make_node(“<>“);}
```

```
  | T_LSS
    {$$ = make_node(“<“);}
  | T_LEQ
    {$$ = make_node(“<=“);}
  | T_GTR
    {$$ = make_node(“>“);}
  | T_GEQ
    {$$ = make_node(“>=“);}
  ;
expression :
  signed_term
    {$$ = $1; }
  | expression add_op term
    {$$ = make_node($2− >symbol); add_tree($$,$1); add_tree($$,$3);}
  ;
signed_term :
  term
    {$$ = $1; }
   | unary_op term
    {$$ = make_node($1− >symbol); add_tree($$,$2); }
   ;
unary_op :
  T_PLUS
     {$$= make_node(“+“); }
  | T_MINUS
     {$$= make_node(“−“); }
  ;
add_op :
  T_PLUS
     {$$= make_node(“+“); }
  | T_MINUS
      {$$= make_node(“−“); }
  ;
term :
  factor
```

```
    {$$ = $1; }
  | term mult_op factor
    {$$ = make_node($2->symbol); add_tree($$,$1);add_tree($$,$3); }
  ;
mult_op :
   T_MULT
    {$$= make_node("*"); }
   | T_DIV
    {$$= make_node("/"); }
   ;
factor :
  T_IDENT
    {$$= $1; }
  | T_NUMBER
    {$$= $1; }
  | T_LPAREN expression T_RPAREN
    {$$= $2; }
   ;
empty :
    {$$= make_node("");}
  ;
```

```
program                                   var x, y, z;
└ block
  ├ const decl part
  ├ var list
  │ ├ x
  │ ├ y
  │ └ z
  ├ proc decl part                        procedure multiply;
  │ ├ proc decl
  │ │ ├ multiply
  │ │ └ block
  │ │   ├ const decl
  │ │   │ ├ const decl                       const one = 23; two = 32;
  │ │   │ │ ├ one
  │ │   │ │ └ 23
  │ │   │ └ const decl
  │ │   │   ├ two
  │ │   │   └ 32
  │ │   ├ var list                           var a,b;
  │ │   │ ├ a
  │ │   │ └ b
  │ │   ├ proc decl part
  │ │   │ └ proc decl
  │ │   │   ├ foo                            procedure foo;
  │ │   │   └ block
  │ │   │     ├ const decl part
  │ │   │     ├ var list                         var c;
  │ │   │     │   c
  │ │   │     ├ proc decl part                   begin // foo
  │ │   │     └ assign                             z := a * c / one;
  │ │   │       ├ z                              end; // foo
  │ │   │       └ /
  │ │   │         ├ *
  │ │   │         │ ├ a
  │ │   │         │ └ c
  │ │   │         └ one
  │ │   └ seq                                begin // muliply
  │ │     ├ assign                         a := x;
  │ │     │ ├ a
  │ │     │ └ x
  │ │     ├ assign                         b := y;
  │ │     │ ├ b
  │ │     │ └ y
  │ │     └ call                           call foo
  │ │       └ foo                          end; // multiply
  │ └ proc decl
  │   ├ bar                                procedure bar;
  │   └ block
  │     ├ const decl part                      var k,l;
  │     ├ var list
  │     │ ├ k
  │     │ └ l
  │     ├ proc decl part                       begin // bar
  │     └ seq                                  y := k
  │       └ assign                             end; // bar
  │         ├ y
  │         └ k
  └ seq                                   begin // main program
    ├ x                                   ? x
    │ └ x
    ├ while                               while x > y do
    │ ├ >
    │ │ ├ x
    │ │ └ y
    │ └ seq                               begin
    │   ├ assign                          x := 2 * x + 3 * y;
    │   │ ├ x
    │   │ └ +
    │   │   ├ *
    │   │   │ ├ 2
    │   │   │ └ x
    │   │   └ *
    │   │     ├ 3
    │   │     └ y
    │   └ assign                          y := y + two;
    │     ├ y
    │     └ +
    │       ├ y
    │       └ two                         end;
    └ write                               ! y
      └ y
                                          end.
```

Abstrakte Syntaxbäume lassen sich noch weiter reduzieren, falls gleiche Teilbäume mehrfach vorkommen, solche werden dann auch nur einmal verwendet durch die Identifizierung der Wurzelknoten dieser Teilbäume. Es entstehen dann aber Graphen, die keine Bäume sind, aber *azyklische gerichtete Graphen* (directed acyclic graph = *DAG*).



Es soll aber auf die Erzeugung solcher DAG's hier nicht weiter eingegangen werden.

Deutlich näher an der Zielsprache von realen Rechnern liegt eine andere Form der Zwischendarstellung. Diese Form ist bereits eine Folge von Instruktionen für eine idealisierte Registermaschine. Der Instruktionssatz enthält Instruktionen für:

- Datentransporte.
- Arithmetische und logische Operationen.
- Programmsprünge.
- Unterprogrammaufruf.
- Ein-/Ausgabe.

Eine standardisierte Form ist der *3-Adreß-Code* (three address code = *TAC*). Das sind Instruktionen der Form:

$$a := b \text{ op } c.$$

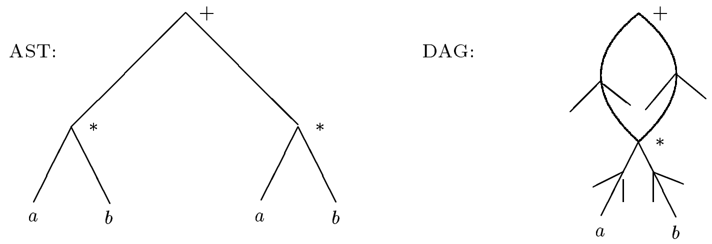
Dabei sind a, b und c Namen für Register und op ist der Name einer arithmetischen oder logischen Operation. Die Register können dabei einen bestimmten (oder auch verschiedene) verabredete Basistypen (z.B. integer, real, char, ...) aufnehmen. Diesen symbolischen Namen sind noch keine konkreten Speicheradressen zugeordnet. Das macht erst der nachfolgende Code-Generator, der den Ziel-Code für eine konkrete Maschine erzeugt.

Ein *TAC-Programm* ist eine Folge von TAC-Instruktionen. Eine Instruktion kann außerdem noch eine *Marke* (ein *label*) haben, so daß man diese Instruktionen in einer Sprunganweisung referenzieren kann. Eine TAC-Instruktion hat höchstens 3 Adressen, kann aber auch weniger haben:

- $a := b$ op c (*binäre Operation*),
- $a :=$ op b (*unäre Operation*),
- $a := b$ (*Move-Instruktion*),
- $m : a := b$ op c (*Instruktion mit Label*),
- goto m (*unbedingter Sprung*),
- ifzero b goto m,

- ifnotzero b goto m (*bedingte Sprünge*):
- arg a_1
  arg a_2
  $\vdots$
  arg a_m (*Argumentenliste für einen Unterprogramm-Aufruf*)
  $a :=$ call p, m (a: Ergebnis, p: Unterprogrammname, m: Zahl der Parameter).
- return b (*Rücksprung aus Unterprogramm*, b: Rückgabewert).

Beispiel:



TAC-Programmteil:
$t_1 := a * b$
$t_2 := a * b$
$t_3 := t_1 + t_2$

$t_1 := a * b$
$t_2 := t_1 + t_1$
oder
$t_1 := a * b$
$t_1 := t_1 + t_1$

t_1, t_2 und t_3 sind dabei interne Registernamen die bei der Zwischencode-Erzeugung geeignet generiert werden. `Ende Beispiel`

Die Variablennamen in den TAC-Instruktionen sind Variablennamen des Quellprogramms oder auch vom Compiler erzeugte Namen. Insbesondere erzeugt der Compiler stets neue Namen für temporäre Register $(t_1, t_2, ...)$. Argumente in TAC-Instruktionen können auch Konstanten sein:

$$a := b + 12$$

oder Marken, wie in

$$\text{goto } m.$$

Die Marken werden ebenfalls vom Compiler erzeugt. Es sind numerische oder symbolische Bezeichnungen von TAC-Instruktionen in der Ausgabefolge. Eine TAC-Instruktion könnte die folgende Struktur haben:

tac_t = record

$op : op\_code_t;$

$a1, a2, a3 : arg\_t;$

end;

arg_t = record

case $kind$ of
$const\_c$: (val: integer);
var_c: (ref: $sym\_tab\_entry\_t$);
$label\_c$: (lab: $\uparrow tac\_t$);
end

Die Symboltabelle enthält die notwendigen Informationen über die Variablen. Die TAC-Instruktionsfolge kann durch die syntaxgesteuerte Übersetzung erzeugt werden. Das soll wieder an einfachen Beispielen gezeigt werden. Es werden geeignete Attribute definiert, die den erzeugten Code, Registernamen, Marken u.s.w. enthalten. Es seien die folgenden Attribute gegeben:

- code (Zeiger auf) TAC-Sequenz (Liste),
- loc (Registername für Werte),
- begin
- end (Marken für Sprünge).

Damit kann die Erzeugung von Zwischencode für die einfachen syntaktischen Konstrukte angegeben werden.

new_temp erzeugt eine neue Bezeichnung für ein temporäres Register,

$c_1||c_2$ steht für die Konkatenation von Codesequenzen c_1 und c_2,

$make\_code$ erzeugt eine TAC-Instruktion,

new_label erzeugt eine neue Marke,

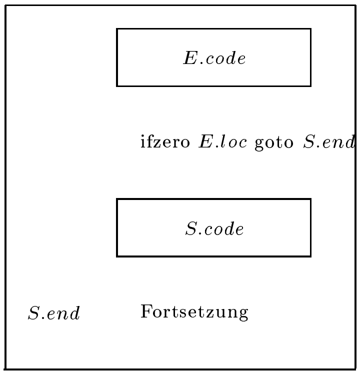
$make\_label$ setzt eine Marke,

$E \rightarrow E_1 + E_2 \; \{E.loc = new\_temp; \; E.code = E_1.code|| \; E_2.code|| \; makecode(+, E.loc, E_1.loc, E_2.loc)\}$,

$S \rightarrow \mathrm{id} := E \; \{S.code = E.code||make\_code(assign, \mathrm{id}.loc, E.loc)\}$.

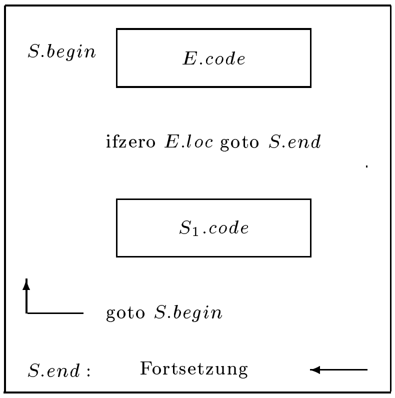
Für if-$then$ und $while$-do werden jetzt Marken benötigt, die auch konsistent in den Code eingefügt werden müssen.

$S \rightarrow$ if E then S_1



$\{S.end = new\_label;$
$S.code = E.code||$
$make\_code(ifz\_goto, E.loc, S.end)||$
$S_1.code||$
$make\_label(S.end)\}$

$S \rightarrow \text{while } E \text{ do } S_1$



$\{S.begin = new\_label;$
$S.end = new\_label;$
$S.code =$
$make\_label(S.begin) || E.code ||$
$make\_code(ifz\_goto, E.loc, S.end) ||$
$S_1.code || make\_code(goto, S.begin) ||$
$make\_label(S.end)\}$

Diese Techniken können auch verwendet werden, um den Ziel-Code für eine Register- oder Stack-Maschine direkt zu erzeugen. Das soll an dem sehr ausführlichen Beispiel des Begleitprojektes gezeigt werden.

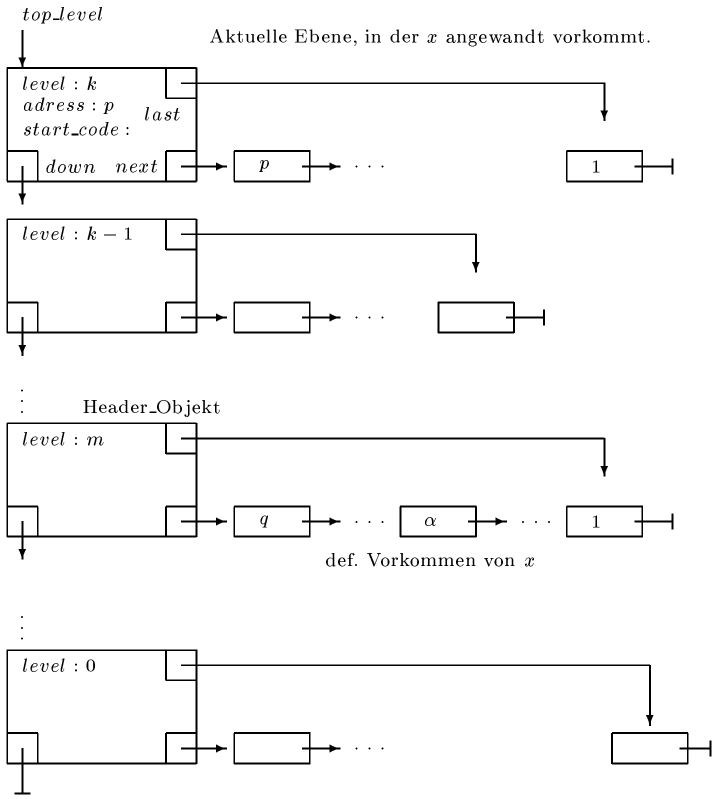
Vorab sind aber noch einige Erklärungen zu machen. Für die Bezeichneridentifikation wird eine Symboltabelle mit Stack-Struktur verwendet. Wenn der Compiler den Text eines neuen Blockes beginnt, dann wird eine neue Ebene der Symboltabelle für die lokalen Definitionen angelegt (Funktion: $new\_level()$). Wenn der Block verlassen wird, wird diese Ebene vollständig entfernt (Funktion: $remove\_level()$). Der erste Block (program) erhält die Ebenen-Zahl 0 und für jede neue Ebene wird diese Zahl (level) um 1 erhöht gegenüber der darunterliegenden Ebene.

Auf diese Weise drückt die Symboltabelle stets die Relation der statischen Blockschachtelung aus ($B_n \sqsubseteq B_{n-1} \sqsubseteq ... \sqsubseteq B_0$).

Tritt ein Bezeichner x angewandt auf, dann wird er zunächst in der aktuellen Ebene (top_level), also lokal im aktuellen Block gesucht. Wenn er hier nicht gefunden wird dann wird er eine Ebene tiefer gesucht u.s.w. (siehe Kapitel 3).

Angenommen die aktuelle Ebene des gerade bearbeiteten Blockes ist k und die Definition von x wird auf der Ebene $m \leq k$ gefunden, dann hat x auf der Ebene m eine (relative) Adresse α und das angewandte Vorkommen im Block k erhält die Adresse: $\alpha : l$ mit $l = k - m$.

Die Bedeutung dieser Paare (α, l) zur Adressierung von Variablen wird erst richtig verständlich, wenn man die Funktion der Zielmaschine (SM1) genauer kennt. Der level-Parameter drückt aber in jedem Fall die Ebenendifferenz aus zwischen der Ebene, in der eine Variable verwendet wird und der Ebene in der sie definiert ist.



Struktur der Symboltabelle

Die genaue Definition dieser Struktur findet man in der Datei: `symbtab.h`. Die Bedeutung der weiteren Einträge (Attribute) im header-Element einer Ebene der Symboltabelle ist:

- level: Ebene des Blockes (statisch),
- address: Anzahl der in dieser Ebene definierten Variablen (relative Adresse der zuletzt definierten Variablen dieses Blockes),
- start_code_address: Label für die erste Instruktion des Blockes.

Die anderen Objekte in der Symboltabelle sind: Variablen, Konstanten und Prozeduren. Diese Objekte besitzen spezifische Attribute, die durch den union-Typ definiert sind (`Symbtab.h`).

- Eine Variable hat eine (bezüglich des Blockes) relative Adresse. Die zuerst definierte Variable erhält die Adresse 1 ($\alpha = 1$) und die weiteren Variablen des Blockes werden fortlaufend numeriert. Der level-Wert ist vom header-objekt übernommen.
- Eine Konstante hat nur einen Wert und keine Adresse.
- Eine Prozedur erhält als Adresse die Programmadresse der ersten Instruktion und erbt ebenfalls den level-Wert vom zugehörigen header-Objekt.

/*———————————————

Datei: symtab.h

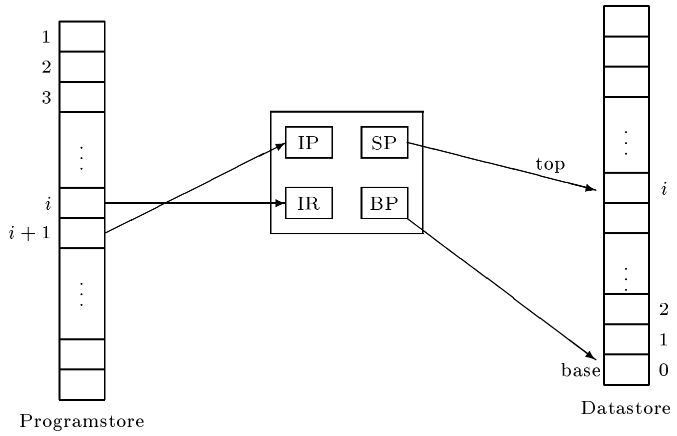
Projekt: PL0 Compiler

Modul: Symbol-Tabelle

```
------------------------------------/*
/* Datentypen */
typedef
    enum { Constant, Variable, Proc, Header } object_class_t;
typedef
    struct {
        unsigned int lev, addr, size;
    } proc_t;
typedef
    struct {
        int val;
    }const_t;
typedef
    struct {
        struct obj *last, *down;
        int level, address, start_code_address;
    }head_t;
typedef
    struct obj{
        char name[32];
        object_class_t kind;
struct obj *next;
        union {
            const_t c;
            var_t v;
            proc_t p;
            head_t h;
        }cont;
    }object_t;
/* Externe Definitionen
------------------------------------*/
extern object_t *top_scope, *undefined_object;
/* Prototypen, Export
```

```
----------------------------------*/
object_t *insert_new(object_class_t, char*);
object_t *lookup(char*);
void init_symtab( void );
void new_level( void );
void remove_level( void );
/* Ende partab.h
----------------------------------*/
```

6.4 Die Stackmaschine SM1



Instruktionsformat:			
	opcode	address	level

Instruktionssatz:

Organisation: halt, alloc n,

Arithmetik: add, sub, mul, div, minus,

Vergleiche: odd, eql, neq, less, leq, gtr, geq,

Sprünge: jump m, jumpcd m, call m, ret,

Datentransport: loadi c, load $a : l$, store $a : l$,

Ein-Ausgabe: read, write.

6.4.1 Der Instruktionssatz

Ein- und Ausgabe

read liest Eingabe und legt sie auf dem Stack ab,

write schreibt oberstes Element des Stacks in die Ausgabe.

Arithmetische Operationen:
Die benötigten Operanden werden vom Stack geholt, dann wird die Operation ausgeführt und das Ergebnis wird wieder auf den Stack abgelegt. Man beachte, daß das oberste Stackelement jeweils der zweite Operand ist.

add: temp = pop; push(pop + temp),

sub: temp = pop; push(pop − temp),

mult: analog,

minus: push(-pop).

Vergleichsoperationen:
Die Operanden werden vom Stack geholt und miteinander verglichen. Das Ergebnis des Vergleichs ist 1 wenn der Vergleich zutrifft und 0 sonst. Das Ergebnis wird wieder auf dem Stack abgelegt. Man beachte auch hier, daß das oberste Stackelement der zweite Operand ist.

eql: temp = pop; push(pop == temp),

less: temp = pop; push(pop < temp),

neq, leq, gtr, geq, odd: analog.

Sprünge:
Der bedingte Sprung erfolgt, wenn die vorhergehende Vergleichsoperation nicht erfüllt war, also den Wert 0 lieferte.

jump m: goto m,

jumpcd m: if (!pop) goto m.

Datentransporte von und zu adressierbaren Speicherplätzen:

loadi c: push (c),

load $a : l$: push (⟨Data[address(a,l)⟩),

store $a : l$: ⟨Data[address(a,l)⟩ = pop.

Instruktionen zur Organisation von Unterprogramm-Aufrufen:

alloc n: $SP = SP + n$,

call $p : l$: siehe unten,

ret $p : l$: siehe unten,

halt: beendet Instruktionszyklus.

Instruktionszyklus:

do

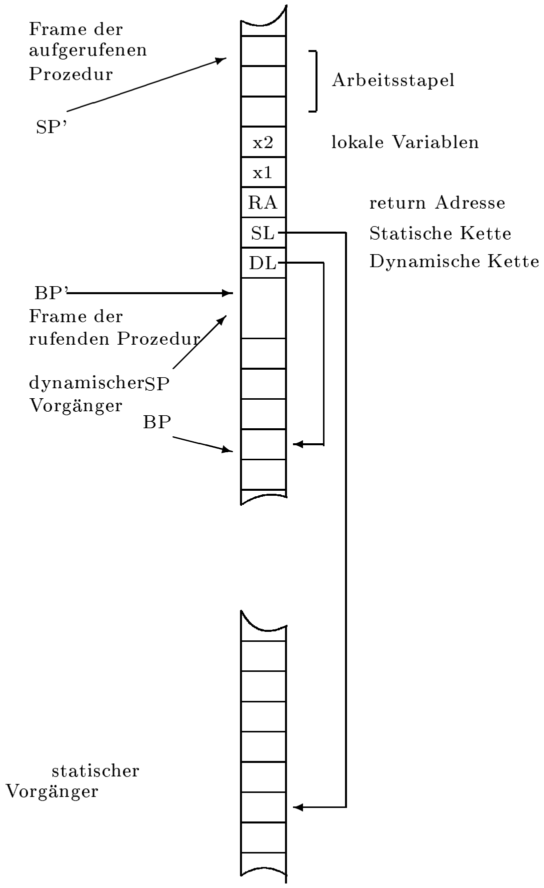
- hole Instruktion l auf die IP zeigt ins IR: IR = Prog[ip];

- erhöhe IP: $ip++$;
- interpretiere Inhalt von IR: execute();

while ir.op != halt;

Operationscodes					
halt	0	eql	7	ret	15
add	1	neq	8	loadi	16
sub	2	less	9	load	17
mul	3	leq	10	store	18
div	4	gtr	11	alloc	19
minus	5	geq	12	call	20
odd	6	jump	13	read	21
		jumpcd	14	write	22

Beispielprogramm: Summe 1..100			
1	alloc	2	zwei Plätze reservieren
2	loadi	100	Schleifenzähler = 100
3	store	1	ablegen in Platz 1
4	loadi	0	laufende Summe = 0
5	store	2	ablegen in Platz 2
6	load	2	hole Summe auf Stack
7	load	1	hole Schleifenzähler auf Stack
8	add		bilde Summe
9	store	2	neue Summe speichern
10	load	1	hole Schleifenzähler
11	loadi	1	hole 1
12	sub		Schleifenzähler - 1
13	store	1	neuer Schleifenzähler
14	load	1	hole Schleifenzähler
15	loadi	0	hole Konstante 0
16	leq		Schleifenzähler <= 0?
17	jumpcd	6	falls nein: gehe zu 6
18	load	2	hole Endsumme
19	write		Endsumme ausgeben
20	halt		



Anordnung der Stackframes (Rahmen) zur Laufzeit.

- Die dynamische Kette entspricht der Folge der Prozeduraufrufe, dynamischer Vorgänger ist der Rahmen der aufrufenden Prozedur.
- Die statische Kette entspricht der textuellen Schachtelung der Blöcke, statischer Vorgänger ist der Rahmen des Blockes in den der aktuelle Block unmittelbar textuell eingebettet ist.

Die Adresse einer Variablen wurde vom Compiler durch ein Paar $a : l$ definiert. Dabei ist:

l die Zahl der Stufen, um die die statische Kette zurückverfolgt werden muß, um die Basis des Rahmens zu erhalten, in dem die Variable lokal definiert ist.

a die relative Adresse bezogen auf die Basis. Zu a wird außerdem der Frame-offset (hier 3) addiert, damit die Zeiger DL, SL, RA übersprungen werden.

Organisation der Stackframes

/* initialize first stack frame */

bp = 0;

Data[bp + STATIC_LINK_POS] = 0;

```
  Data[bp + DYNAMIC_LINK_POS] = 0;
  Data[bp + RETURN_ADDR_POS] = 0;
call:
  Data[sp + DYNAMIC_LINK_POS] = bp;
  Data[sp + RETURN_ADDR_POS] = ip;
  Data[sp + STATIC_LINK_POS] = base(level);
  bp = sp;
  sp = bp + FRAME_OFFSET;
  ip = address;
ret:
  sp = bp;
  ip = Data[bp + RETURN_ADDR_POS];
  bp = Data[bp + DYNAMIC_LINK_POS];
```

Adressrechnung

address(a,l): Wenn $a : l$ der Adreßteil einer Instruktion ist, dann gibt die Funktion address(a,l) eine errechnete absolute Adresse des Datenspeichers zurück.

Der Levelwert 1 wurde vom Compiler berechnet als die Differenz der Blockebenen zwischen dem definierenden und dem angewandten Vorkommen der referenzierten Variablen.

```
int base( int l) {
  int b,i;
  b= bp;
  for (i=0; i<1; i++){
    b = memory.data[b+STATIC_LINK_POS];
  }
  return (b);
}
/*----------------------*/
int address(adr, lev) {
  return (base(lev) + FRAME_OFFSET + adr);
}
```

```
/*______________________
Datei: codegen.h
Projekt: PL0 Compiler
Modul: Code Erzeugung fuer SM1
*/
/*_Includes____________________*/
#include "../../include/sm1.h"
/*_____________________*/
/* Prototypen, Export */
 void init_code_generator( void );
 void fixup( unsigned int );
 void make_code(op_code_t, unsigned int , unsigned char );
 unsigned int label( void );
/*_Ende codegen.h____________________*/
```

```
program : block T_PERIOD
    ;
block :
      {new_level();
      top_scope->cont.h.start_code_address=label();
      make_code(jump(0,0);}
      const_declaration_part
      var_declaration_part
   procedure_declaration_part
      {fixup(top_scope->cont.h.start_code_address);
      make_code(alloc, top_scope->cont.h.address,0);}
   statement
       {make_code(ret,0,0);
       remove_level();}
   ;
const_declaration_part :
   empty
   | T_CONST const_decl_list T_SEMICOLON
```

```
  ;
const_decl_list:
  const_declaration
  | const_decl_list T_COMMA const_declaration
  ;
const_declaration :
  T_IDENT T_EQL T_NUMBER
    { p = insert_new(Constant,$1.name);
    p− >cont.c.val=$3.value;}
   ;
var_declaration_part :
  empty
   | T_VAR var_list T_SEMICOLON
   ;
var_list :
  T_IDENT
    {p = insert_new(Variable, $1.name);}
  | var_list T_COMMA T_IDENT
    {p = insert_new(Variable, $3.name);}
  ;
procedure_declaration_part:
  empty
  | procedure_declaration_part procedure_declaration
  ;
procedure_declaration:
  T_PROCEDURE T_IDENT T_SEMICOLON
    {p=insert_new(Proc,$2.name);
    p− >cont.p.addr = label();
    p− >cont.p.lev = top_scope− >cont.h.level;}
block T_SEMICOLON
  ;
statement :
  empty
  | T_IDENT T_ASSIGN expression
    {p=lookup($1.name);
```

```
    make_code(store, p- >cont.v.addr,
    top_scope- >cont.h.level- p- >cont.v.lev);}
  | T_CALL T_IDENT
    {p=lookup($2.name);
    make_code(call, p- >cont.p.addr,
    top_scope- >cont.h.level- p- >cont.p.lev);}
  | T_READ T_IDENT
    {p=lookup($2.name);
    make_code(read, 0, 0);
    make_code(store, p- >cont.v.addr,
    top_scope- >cont.h.level- p- >cont.v.lev);}
  | T_WRITE expression
    {make_code(write, 0,0);}
  | T_BEGIN statement_sequence T_END
  | T_IF condition
    {push_label(); make_code(jumpcd,0,0);}
   T_THEN statement
    {fixup(pop_label());}
  | T_WHILE
    {push_label();}
   condition
    {push_label(); make_code(jumpcd,0,0);}
T_DO statement
    { lab= pop_label(); make_code(jump, pop_label(),0);
    fixup(lab);}
  ;
statement_sequence:
  statement
   | statement_sequence T_SEMICOLON statement
   ;
condition :
  T_ODD expression
    {make_code(odd, 0,0);}
  | expression comp_operator expression
    {make_code($2.operator,0,0);}
```

```
  ;
comp_operator :
  T_EQL
    {$$.operator = eql;}
  | T_NEQ
    {$$.operator = neq;}
  | T_LSS
    {$$.operator = less;}
  | T_LEQ
    {$$.operator = leq;}
  | T_GTR
    {$$.operator = qtr;}
  | T_GEQ
    {$$.operator = geq;}
  ;
expression :
  signed_term
  | expression add_op term
    {make_code($2.operator,0,0);}
  ;
signed_term :
  term
  | unary_op term
    {if($1.operator>=0) make_code($1.operator,0,0);}
  ;
unary_op :
  T_PLUS
    {$$.operator = -1; }
  | T_MINUS
    {$$.operator = minus; }
  ;
add_op :
  T_PLUS
    {$$.operator = add; }
  | T_MINUS
```

```
    {$$.operator = sub; }
  ;
term :
  factor
  | term mult_op factor
    {make_code($2.operator,0,0);}
  ;
mult_op :
  T_MULT
    {$$.operator = mul; }
  | T_DIV
    {$$.operator = divi; }
  ;
factor :
  T_IDENT
    {p = lookup($1.name);
    if(p- >kind == Variable)
    make_code(load, p- >cont.v.addr,
    top_scope- >cont.h.level -p- >cont.v.lev);
    if(p- >kind == Constant)
    make_code(loadi, p- >cont.c.val,0;}
  | T_NUMBER
    {make_code(loadi, $1.value, 0);}
  | T_LPAREN expression T_RPAREN
   ;
empty :;
```

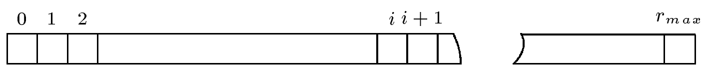
```
//PL0; Groesster gemeinsamer Teiler
VAR a,b;
BEGIN
  ?a;
  WHILE a > 0 DO
  BEGIN ?b;
    WHILE a <> b DO
    BEGIN
      IF a > b THEN a := a - b;
      IF a < b THEN b := b - a;
      !a; !b
    END;
    ?a
  END
END.
```

1	jump	0	0
0	fix	1	2
2	alloc	2	0
3	read	0	0
4	store	1	0
5	load	1	0
6	loadi	0	0
7	gtr	0	0
8	jumpcd	0	0
9	read	0	0
10	store	2	0
11	load	1	0
12	load	2	0
13	neq	0	0
14	jumpcd	0	0
15	load	1	0
16	load	2	0
17	gtr	0	0
18	jumpcd	0	0
19	load	1	0
20	load	2	0
21	sub	0	0
22	store	1	0
0	fix	18	23
23	load	1	0
24	load	2	0
25	less	0	0
26	jumpcd	0	0
27	load	2	0
28	load	1	0
29	sub	0	0
30	store	2	0
0	fix	26	31
31	load	1	0
32	write	0	0
33	load	2	0
34	write	0	0
35	jump	26	0
0	fix	14	36
36	read	0	0
37	store	1	0
38	jump	26	0
0	fix	14	39
39	ret	0	

Zwischencode-Erzeugung für strukturierte Datentypen

Es sollen jetzt noch einige Aspekte der Zwischencode-Erzeugung etwas genauer betrachtet werden. Strukturierte Datentypen benötigen je nach Konstruktion unterschiedlich viel Speicherplatz. Dieser Platzbedarf kann bereits bei der Zwischencode-Erzeugung ermittelt werden.

Register sind bisher für die Aufnahme von Basistypen (integer, char, real, ...) undifferenziert verwendet worden. Man kann die Größe eines Registers (z.B. in Byte oder Bit) festlegen und dann für Objekte mit Speicherplatzbedarf jeweils eine Größe als Zahl der benötigten Register berechnen. Der Speicher wird dabei als eine lineare Folge von adressierbaren (numerierten) Registern betrachtet:

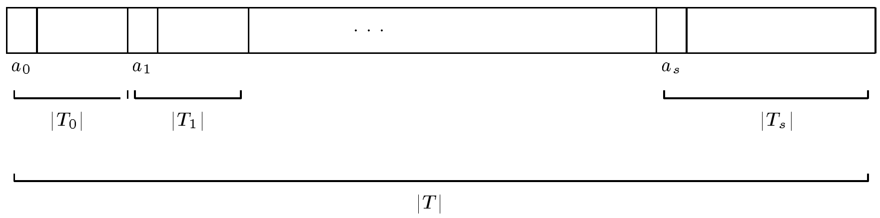


Sei T ein Datentyp, dann soll $|T|$ die Zahl der Register sein, die man benötigt, um Werte des Typs T darzustellen (zu speichern). Da Datentypen durch Typausdrücke beschrieben werden, kann man auch diesen Ausdrücken die entsprechende Größe zuordnen:

- Für Basistypen wird die Größe direkt festgelegt:

$|\text{integer}| = 1,$
$|\text{char}| = 1,$
$|\text{real}| = 2$

- und für zusammengesetzte Typausdrücke wird die Größe rekursiv definiert. Außerdem definiert man dazu auch gleich Funktionen zur Selektion der Komponenten. Ist T ein strukturierter Typ mit Komponenten: $T_0, T_1, ..., T_s$, dann müssen diesen Komponenten relative Adressen innerhalb des Bereiches von T zugeordnet werden:



Diese Adressen werden als offset bezüglich a_0 berechnet.

Speicherplatzabbildung für strukturierte Datentypen:

1. $T = \text{array}[1..k] \text{ of } T_0$
   $|T| = k|T_0|$
   $offset(i) = (i-1) \cdot |T_0|$
2. $T = \text{record } x_1 : T_1; ...; x_n : T_n \text{end}$
   $|T| = \sum_{i=1}^{n} |T_i|; \; offset(x_i) = \sum_{j=1}^{i-1} |T_j|$
3. $T =$ <u>record</u>
   <u>case</u> *kind* <u>of</u>
   $k_1 : (x_{11} : T_{11}, ..., x_{1p_1} : T_{1p_1});$

⋮

   $k_m : (x_{m1} : T_{m1}, ..., x_{mp_m} : T_{mp_m});$
   <u>end</u> ;
   $|T| = \max_j \left\{ \sum_{i=1}^{p_j} |T_{ji}| \right\}, \; offset(x_{ij} = \sum_{l=1}^{j-1} |T_{il}|$

Wenn ein Objekt deklariert wird, das Speicherplatz benötigt, dann wird in die Symboltabelle eingetragen:

Name, Typ(ausdruck), Anfangsadresse

dieses Objekts. Das kann wieder durch die semantischen Regeln erfolgen. Als Beispiel betrachten wir einen entsprechenden Ausschnitt aus einer attributierten Grammatik, die hier als Übersetzungsschema formuliert ist.

Beispiel: Nichtterminalzeichen: B (Block), D (Deklaration), S (Anweisung), T (Typausdruck)

- $enter(name, type, address)$ macht einen Eintrag in die Symboltabelle,
- adr ist eine globale Variable, die die laufende Anfangsadresse enthält.

Attribute:

- *size*: Größe eines Typs,
- *type*: Typausdruck,
- *name*: Lexem eines Identifiers,
- *val*: Wert eines num-Tokens.

$B \to \{adr = 0\} D; S$

$B \to D; D$

$D \to \text{id} : T\{enter(\text{id}.name, T.type, adr); adr = adr + T.size;\}$

$T \to integer\ \{T.type = integer; T.size = 1;\}$

$T \to real\ \{T.type = real; T.size = 2;\}$

$T \to array[num] of T_1\ \{T.type = array(num.val, T_1.type);\ T.size = num.val * T_1.size;\}$

$T \to \uparrow T_1\ \{T.type = pointer(T_1.type);\ T.size = 1;\}$

`Ende Beispiel`

Übung: Man gebe semantische Regeln an, um Record-Typen (oder variante Record-Typen) zu berücksichtigen.

Mehrdimensionale Arrays

Etwas ausführlicher soll jetzt der Fall mehrdimensionaler Arrays behandelt werden. Dabei ist besonders interessant, daß ein Teil der Adreßrechnung zur Compile-Zeit ausgeführt werden kann. Dadurch wird der Zugriff auf Array-Komponenten zur Laufzeit effizienter.

Sei zunächst $T = array[a..b] of T_0$, wobei a und b natürliche Zahlen sind mit $a < b$. Die Zahl der Komponenten ist dann $k = (b - a) + 1$. Die Anfangsadresse im Speicher sei l_0, und es sei $q = |T_0|$, dann gilt für die Komponente i mit $a \le i \le b$ die Adreßrechnung:

$$\begin{aligned} loc(i) &= l_0 + (i - a) \cdot q \\ &= l_0 + iq - aq \\ &= iq + (l_0 - aq) \end{aligned}$$

Der Term $l_0 - aq$ kann bereits vom Compiler ausgewertet werden: $c = l_0 - aq$ und damit ist die Adreßrechnung zur Laufzeit: $loc(i) = iq + c$. Allgemein erhält man für die Adreßrechnung bei m-dimensionalen Arrays die folgenden Beziehungen. Sei $1 \le i \le m$ und $I_i = a_i ... b_i$ das Indexintervall in der i-ten Dimension. Weiter sei $k_i = b_i - a_i + 1 = |I_i|$ und $q = |T_0|$. Dann sei $\text{rloc}(x_1 : k_1, ..., x_m : k_m, q)$ die relative Position (bezogen auf die Anfangsadresse) der Komponente $A[x_1, ..., x_m]$ wenn gilt $A : array[I_1, ..., I_m]\ of\ T_0$.

Behauptung:

$$\text{rloc}(x_1 : k_1, ..., x_m : k_m, q) = q \sum_{i=1}^{m} (x_i - a_i) \alpha_i^{(m)} \text{ mit } \alpha_i^{(m)} = \prod_{j=i+1}^{m} k_j.$$

Beweis:

1. $m = 1$: $\mathrm{rloc}(x_1 : k_1, q) = (x_1 - a_1)q$ wegen $\alpha_1^{(1)} = 1$ gilt die Formel für $m = 1$.
2. $m > 1$:

$$\underbrace{\mathrm{rloc}(x_1 : k_1, ..., x_{m-1} : k_{m-1}, k_m q)}_{\text{rel. Anfangsadr eines Arrays mit } k_m \text{ Komp. vom Typ } T_0} + \underbrace{(x_m - a_m)q}_{\text{Komponente } x_m}$$

Nach Induktionsvoraussetzung:

$$\begin{aligned} \mathrm{rloc}(x_1 : k_1, ..., x_m : k_m, q) &= k_m q \sum_{i=1}^{m-1} (x_i - a_i)\alpha_i^{(m-1)} + (x_m - a_m)q \\ &= q \sum_{i=1}^{m} (x_i - a_i)\alpha_i^{(m)} \end{aligned}$$

□

Damit erhält man:

$$\mathrm{loc}(x_1, ..., x_m) = l_0 + q \sum_{i=1}^{m} x_i \alpha_i^{(m)} - q \sum_{i=1}^{m} a_i \alpha_i^{(m)}.$$

Das ist noch einfacher durch Skalarprodukte von Vektoren zu formulieren:

Sei $x = (x_1, ..., x_m)$, $a = (a_1, ..., a_m)$, $u = q(\alpha_1^{(m)}, ..., \alpha_m^{(m)})$, dann gilt:

$$\mathrm{loc}(x) = l_0 + xu - au,$$

und davon kann man $c = l_0 - au$ zur Compile-Zeit berechnen. Zur Laufzeit wird also nur Code benötigt, der das Skalarprodukt xu berechnet. Da die Komponenten x_i in einem Selektor $A[x_1, ..., x_m]$ selbst Ausdrücke sein können, ist die Code-Erzeugung für solche Selektor-Ausdrücke nicht ganz trivial.

Da der Code in der DFS-Folge erzeugt werden soll, kann man das Produkt xu auch rekursiv definieren für jedes Anfangsstück von x. Sei $s \leq m$, dann sei $e_s = \sum_{i=1}^{s} x_i \alpha_i^{(s)}$ oder:

$$\begin{aligned} e_1 &= x_1 \\ e_s &= e_{s-1} k_s + x_s \text{ und } q e_m = xu \end{aligned}$$

Mit der Adreßrechnung $\mathrm{loc}(x)$ wird also der L-Wert für einen Selektorausdruck $A[x_1, ..., x_m]$ ermittelt.

Wir betrachten im folgenden Beispiel einen Ausschnitt aus einer Grammatik mit der solche Selektorausdrücke für mehrdimensionale Arrays erzeugt werden können.

Beispiel: Nichtterminalzeichen:

L: L-Wert (durch syntakt. Position),

E: Ausdruck,

$Elist$: Liste von Ausdrücken (Selektor).

Grammatik (Ausschnitt):

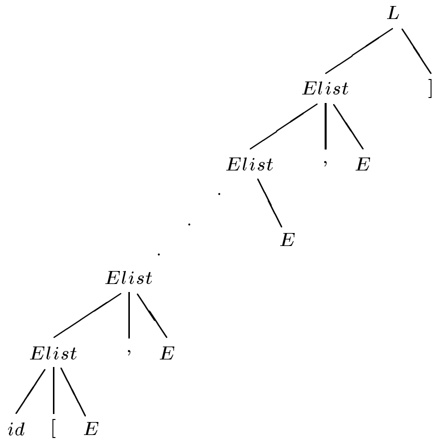
$L \to \mathrm{id}[Elist] | \mathrm{id}$,

$Elist \rightarrow Elist, E|E$.

Um die notwendigen Attribute bequemer berechnen zu können wird die Grammatik etwas modifiziert:

$L \rightarrow Elist]|\mathrm{id}$,

$Elist \rightarrow Elist, E|\mathrm{id}[E$.



Für $Elist$ kann man daher synthetische Attributwerte aus Attributwerten von id berechnen.

Attribute:

$Elist.ndim$: Zahl der Indexausdrücke (m),

$Elist.array$: Zeiger auf Symboltabelleneintrag zum Array-Bezeichner,

$E.place$: (temp.) Register für den Wert des Ausdrucks,

$L.place$: (symbolischer) Speicherplatz der den L-Wert darstellt. Bei Array: Anfangsadresse,

$L.offset$: 0 für einfache Variablen und $offset$ für Array-Variable.

Funktionen:

$limit(p, i) = k_i$, p: Zeiger auf Symboltabellen-Eintrag für Array-Objekt, i: Komponente,

$q(p) = |T_0|$, p wie oben,

$c(p)$ die vom Compiler berechnete Konstante.

Es folgt eine attributierte Grammatik zu diesem Beispiel.

Grammatik-Auszug:

$S \rightarrow L := E$

$E \rightarrow E + E|(E)|L$

$L \rightarrow Elist]|\mathrm{id}$

$Elist \rightarrow Elist, E|\mathrm{id}|E$

Attributierung:

$S \to L := E$

$\{$ <u>if</u> $L.offset =$ <u>nil</u> <u>then</u> $make\_code(assign, L.place, E.place)$

<u>else</u> $make\_code(assign, L.place[L.offset], E.place)\}$

$E \to E_1 + E_2$

$\{E.place = new\_temp; make\_code(+, E.place, E_1.place, E_2.place)\}$

$E \to (E_1)$

$\{E.place = E_1.place\}$

$E \to L$

$\{$ <u>if</u> $L.offset =$ <u>nil</u> <u>then</u> $E.place = L.place$

<u>else</u> $E.place = new\_temp; make\_code(assign, E.place, L.place[L.offset]\}$

$L \to Elist]$

$\{L.place = new\_temp; L.offset = new\_temp;$

$make\_code(assign, L.place, c(Elist.array));$

$make\_code(*, L.offset, Elist.place, q(Elist.array))\}$

$L \to \mathrm{id}$

$\{L.place = \mathrm{id}.place; L.offset = nil\}$

$Elist \to Elist_1, E$

$\{t = new\_temp, s = Elist_1.ndim + 1;$

$make\_code(*, t, Elist_1.place, limit(Elist_1.array, s));$

$make\_code(+, t, t, E.place);$

$Elist.array = Elist_1.array;$

$Elist.place = t;$

$Elist.ndim = s; \}$

$Elist \to \mathrm{id}[E$

$\{Elist.array = \mathrm{id}.place;$

$Elist.place = E.place;$

$Elist.ndim = 1\}$

`Ende Beispiel`

Zwischencode-Erzeugung für Boolesche Ausdrücke

Boolesche Werte werden ganz ähnlich behandelt wie numerische Werte, und Ausdrücke mit Booleschen Werten werden ganz analog zu arithmetischen Ausdrücken behandelt. Dazu muß lediglich eine Darstellung der Booleschen Werte für true und false durch Registerinhalte festgelegt werden, wie z.B.:

$$\text{true} \mathrel{\widehat{=}} 1;\ \text{false} \mathrel{\widehat{=}} 0$$

oder auch: false $\widehat{=}$0 und jeder andere Wert stellt true dar. Oft werden Boolesche Ausdrücke (BA) so verwendet, daß der Kontrollfluß in Abhängigkeit vom Wert des Ausdrucks unterschiedlich verzweigt, wie etwa in *if-then-* oder *while-do*-Anweisungen. Man kann Boolesche Werte daher auch durch Programm-Adressen darstellen. Das soll bedeuten, daß bei der Berechnung eines BA zwei verschiedene Programm-Adressen (Sprung-Adressen) berechnet werden (oder zur Compile-Zeit bestimmt werden) und ein Programmsprung je nach dem Wert des Ausdrucks erfolgt.

Der folgende Ausschnitt aus einer attributierten Grammatik zeigt zunächst die übliche Bewertung in Analogie zur Bewertung von arithmetischen Ausdrücken. Darin ist *relop* ein Token und das Attribut *relop.op* ist der Vergleichsoperator, der direkt in TAC verwendet werden kann. *E.place* ist wieder ein Register für den R-Wert des Ausdrucks.

$E \to E_1$ and E_2 $\{E.place = new\_temp;$

$\quad make\_code(and, E.place, E_1.place, E_2.place)\},$

$E \to \mathrm{id}_1$ *relop* id_2 $\{$

$\quad E.place = new\_temp;$

$\quad u = new\_temp;$

$\quad make\_code(relop.op, u, \mathrm{id}_1.place, \mathrm{id}_2.place);$

$\quad$ (label) $make\_code(ifzero, u, goto\ label + 3);$

$\quad +1\ make\_code(assign, E.place, 0);$

$\quad +2$ (label) $make\_code(goto\ label + 2);$

$\quad +3\ +1\ make\_code(assign, E.place, 1);$

$\quad +2\ \}$

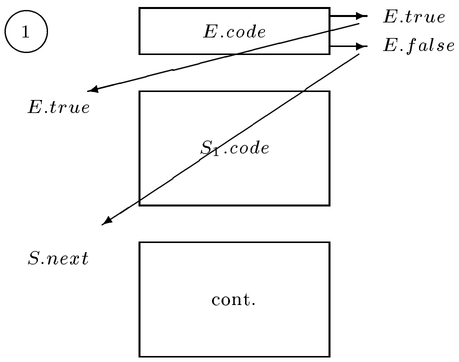
$E \to true\ \{E.place = new\_temp;\ make\_code(assign, E.place, 1);\}$

label ist dabei eine Funktion, die den Wert der gerade erzeugten Code-Zeile liefert. Man kann daher relativ zu label adressieren. In der ersten Regel oben wird davon ausgegangen, daß der Code für E_1 und E_2 bereits ausgegeben wurde und daß $E_1.place$ und $E_2.place$ die Werte dieser (Teil-) Ausdrücke enthalten.

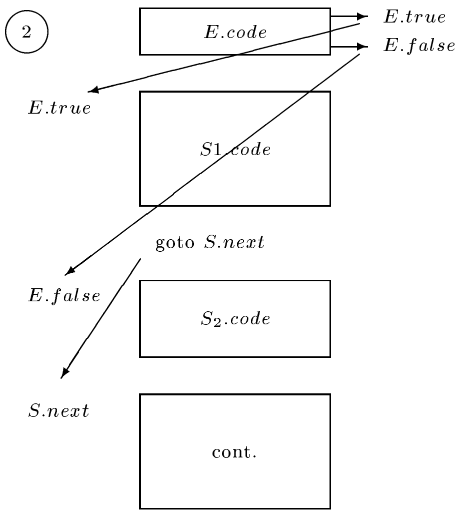
Alternativ kann man BA auch derart auswerten, daß je nach Wert an eine andere Sprungadresse verzweigt wird. In diesem Fall definiert der „Ort der Kontrolle" den Booleschen Wert. Mit dieser Technik lassen sich Kontrollfluß-Anweisungen effizient übersetzen. Um diese Form der Code-Erzeugung zu beschreiben, werden mit jedem Booleschen Ausdruck E die Attribute $E.true$ und $E.false$ verbunden, deren Wert eine Programm-Adresse (ein Label) ist. Jetzt werden mit Anweisungen wieder TAC-Sequenzen (*S.Code*) verbunden, die dann geeignet zusammengesetzt werden können. *S.next* ist ein Attibut für Anweisungen und hat als Wert ein Label (wie $E.true$, $E.false$), also eine symbolische Programm-Adresse. *S.next* soll dabei die Adresse der Code-Zeile sein, die direkt (oder indirekt bei goto) nach dem Code von S ausgeführt wird.

Das soll jetzt für einige Konstrukte exemplarisch gezeigt werden. Es werden dazu auch einfache „Block-Schaltungen" angegeben, die den Kontrollfluß darstellen sollen:

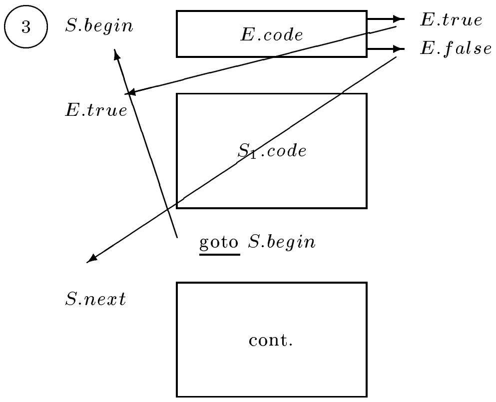
soll die TAC-Sequenz $E.code$ darstellen, wobei die Sprungadressen als Ausgänge mit Pfeilen dargestellt sind.



1. $S \rightarrow$ <u>if</u> E <u>then</u> S_1
$\{E.true = new\_label;$
$E.false = S.next;$
$S_1.next = S.next;$
$S.code = E.code ||$
$make\_label(E.true) ||$
$S_1.code; \}$

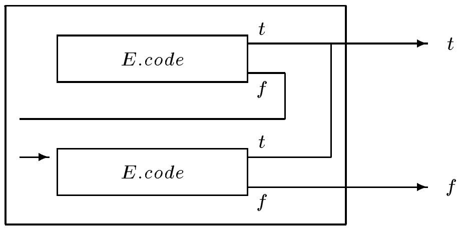


1. $S \rightarrow$ <u>if</u> E <u>then</u> S_1 <u>else</u> S_2
$\{E.true = new\_label;$
$E.false = new\_label;$
$S_1.next = S.next;$
$S.code = E.code || makelabel(E.true) ||$
$S_1.code || make\_code(goto\ S.next) ||$
$make\_label(E.false) || S_2.code; \}$



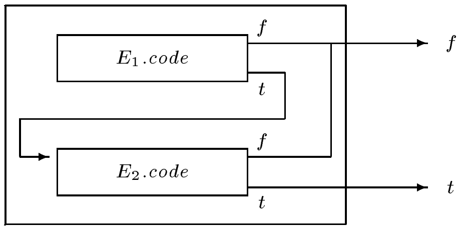
1. $S \rightarrow$ <u>while</u> E <u>do</u> S_1
$\{S.begin = new\_label;$
$E.true = new\_label;$
$E.false = S.next;$
$S_1.next = S.begin;$
$S.code = E.code ||$
$make\_label(E.true) || S_1.code ||$
$make\_code(goto\ S.begin); \}$

4



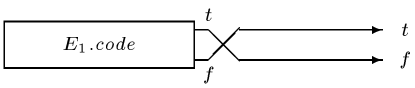
$E \rightarrow E_1$ or E_2
$\{E_1.true = E.true;$
$E_1.false = new\_label;$
$E_2.true = E.true;$
$E_2.false = E.false;$
$E.code = E_1.code||$
$make\_label(E_1.false)||$
$E_2.code; \}$

5



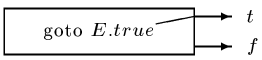
$E \rightarrow E_1$ and E_2
$\{E_1.false = E.false;$
$E_1.true = new\_label;$
$E_2.false = E.false;$
$E_2.true = E.true;$
$E.code = E_1.code||$
$make\_label(E_1.true)||$
$E_2.code; \}$

6



$E \rightarrow$ not E_1
$\{E_1.true = E.false;$
$E_1.false = E.true;$
$E.code = E_1.code; \}$

7



$E \rightarrow true$
$E.code = make\_code(goto\ E.true)\}$

Es ist zu beachten, daß Sprungadressen wie $E.true$, $E.false$, $S.next$ geerbte Attribute sind, so daß diese Sprungadressen „in die Konstrukte hinein“ vererbt werden. Boolesche Ausdrücke werden so ausgewertet, daß die Auswertung abgebrochen wird, sobald der Wert feststeht. So wird in 5. $E_2.code$ nicht durchlaufen, wenn E_1 den Wert false ergibt.

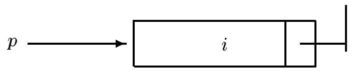
Boolesche Ausdrücke — Effizienter mit Backpatching

Als letzte Technik in diesem Abschnitt soll das *Backpatching* (Nachflicken, Nachbessern) oder auch *Fixup* behandelt werden. Backpatching ist ein Verfahren, bei dem die Sprungadressen für Vorwärtssprünge zunächst undefiniert bleiben und erst dann ersetzt („gepatcht“) werden, wenn die Sprungadresse bekannt ist, also dann wenn die angesprungene Code-Zeile erzeugt wird. Wenn der erzeugte Code im Speicher gehalten wird, dann kann das Nachbessern im gleichen Durchlauf erfolgen. Wurde der (unvollständige) Code aber bereits ausgegeben, dann muß das Nachbessern in einem weiteren Durchlauf erfolgen (vergl. hierzu die Lösung beim Begleitprojekt: Pl0-Compiler).

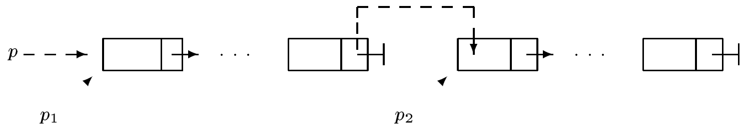
Die Idee dieser Technik besteht darin, daß man Listen anlegt mit denjenigen Code-Zeilennummern, die nachgebessert werden müssen. Man legt für jede noch nicht bekannte Sprungadresse die Liste derjenigen Zeilen-Nummern an, die mit genau dieser Sprungadresse korrigiert werden müssen.

Funktionen: Die Code-Zeilennummern werden als Code-Index $i \in I\!N$ betrachtet.

- $p = make\_list(i)$ liefert einen Zeiger p auf eine neu erzeugte Liste, die i als ersten (und einzigen) Code-Index enthält:



- Verkettung von zwei Listen: Seien p_1 und p_2 Zeiger auf Code-Index-Listen, dann sei $p = cat(p_1, p_2)$ Zeiger auf die Verkettung (Konkatenation) dieser beiden Listen:



- Nachbessern: Sei p Zeiger auf eine Liste und j ein Code-Index, dann bewirkt $backpatch(p, j)$, daß jede Codezeile der Liste p mit der Sprungadresse j nachgebessert wird.

Die folgende syntaxgesteuerte Übersetzung verwendet diese Technik. Statt $E.true$, $E.false$, $S.next$ u.s.w. werden jetzt ähnliche Attribute verwendet, deren Werte aber jeweils Listen von Code-Indizes sind: $E.truelist$, $E.falselist$, $S.nextlist$ u.s.w. Um die Indizes der TAC-Instruktionen zu bestimmen, wird ein spezielles Nonterminal M eingeführt, das auch nur zu ϵ abgeleitet werden kann ($M \to \epsilon \in P$). M hat das Attribut $M.label$ dessen Wert ein Code-Index ist, und die Funktion $label$ liefert stets den Index der nächsten zu erzeugenden Instruktion.

Grammatik-Auszug:

Boolesche Ausdrücke mit Verzweigung und Backpatching

$E \to E_1\ \underline{\text{or}}\ ME_2 | E_1\ \underline{\text{and}}\ ME_2 |\ \underline{\text{not}}\ E_1 | (E_1)$

$\quad |\text{id}_1\ relop\ \text{id}_2|\ \underline{\text{true}}\ |\ \underline{\text{false}}$

$M \to \epsilon$

Attributierung:

$E \to E_1\ \underline{\text{or}}\ ME_2$

$\quad \{backpatch(E_1.falselist, M.label);$

$\quad E.truelist = cat(E_1.truelist, E_2.truelist);$

$\quad E.falselist = E_2.falselist\}$

$E \to E_1\ \underline{\text{and}}\ ME_2$

$\quad \{backpatch(E_1.truelist, M.label);$

$\quad E.falselist = cat(E_1.falselist, E_2.falselist);$

$\quad E.truelist = E_2.truelist\}$

$E \to\ \underline{\text{not}}\ E_1$

$\quad \{E.falselist = E_1.truelist; E.truelist = E_1.falselist\}$

$E \to (E_1)$

$\quad \{E.falselist = E_1.falselist; E.truelist = E_1.truelist\}$

$E \rightarrow \mathrm{id}_1 relop \mathrm{id}_2$

$\{E.truelist = make\_list(label);$

$\{E.falselist = make\_list(label + 1);$

(label:) $make\_code(if\_goto, \mathrm{id}_1.place, relop.op, id_2.place, nil);$

(label+1:) $make\_code(goto, nil); \}$

$E \rightarrow$ <u>true</u>

$\{E.truelist = make\_list(label);$

$make\_code(goto, nil); \}$

$E \rightarrow$ <u>false</u>

$\{E.falselist = make\_list(label);$

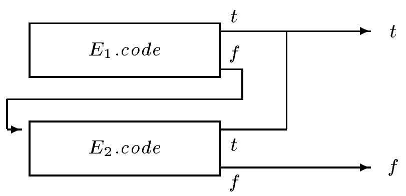
$make\_code(goto, nil); \}$

$M \rightarrow \epsilon$

$\{M.label = label\}$

Einige der oben aufgeführten Konstrukte sollen noch genauer erklärt werden:

1. $E \rightarrow E_1$ <u>or</u> $M E_2$



- Nachdem $E_1.code$ erzeugt wurde, wird die erste Instruktion von $E_2.code$ mit dem Index *label* markiert. Dieser wurde aber als Attributwert $M.label$ zugewiesen als nach $M \rightarrow \epsilon$ reduziert wurde.
- Im Code von E_1 (also $E_1.code$) können jetzt alle Instruktionen, die auf der Liste $E_1.falselist$ stehen mit diesem Code-Index nachgebessert werden:

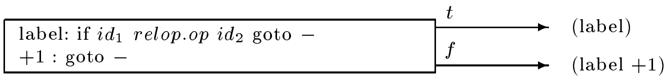
$$backpatch(E_1.falselist, M.label).$$

- Da die für den true- und false-Wert von E noch nicht bekannt sind, werden die Listen aus den Teilausdrücken übernommen:

$$\begin{array}{rcl} E.falselist & = & E_2.falselist \\ E.truelist & = & cat(E_1.truelist, E_2.truelist). \end{array}$$

Die beiden true-Listen werden konkateniert und die false-Liste von E_2 wird übernommen. Die anderen Booleschen Ausdrücke werden analog behandelt.

2. $E \rightarrow \mathrm{id}_1$ *relop* id_2



- Die nächste zu erzeugende Code-Zeile (Index) hat die Nummer *label* und die darauf folgende Nummer $label + 1$. Es werden zwei neue Listen mit diesen Indizes angelegt:

$$E.truelist = make\_list(label)$$

$$E.falselist = make\_list(label + 1)$$

- Dann werden zwei Code-Zeilen mit undefinierten Sprungadressen erzeugt. Der Index der ersten steht dann also in der $E.truelist$ und der Index der zweiten in der $E.falselist$. Erst wenn die zugehörigen Sprungadressen bekannt sind werden die *goto*-Instruktionen entsprechend nachgebessert.

Diese Technik kann natürlich auch auf Anweisungen angewendet werden, wie die folgenden Attributierungen zeigen.

Kontrollstrukturen mit Verzweigungen und Backpatching

$S \rightarrow \underline{\text{if}}\ E\ \underline{\text{then}}\ S |\ \underline{\text{if}}\ E\ \underline{\text{then}}\ S\ \underline{\text{else}}\ S$

$\quad |\ \underline{\text{while}}\ E\ \underline{\text{do}}\ S |\ \underline{\text{begin}}\ L\ \underline{\text{end}}\ | A$

$L \rightarrow L; S | S$

Attributierung:

$N \rightarrow \epsilon$

$\quad \{N.nextlist = make\_list(label);$

$\quad make\_code(goto, nil)\}$

$M \rightarrow \epsilon$

$\quad \{M.label = label\}$

$S \rightarrow \underline{\text{if}}\ E\ \underline{\text{then}}\ M_1 S_1 N\ \underline{\text{else}}\ M_2 S_2$

$\quad \{backpatch(E.truelist, M_1.label);$

$\quad backpatch(E.falselist, M_2.label);$

$\quad S.nextlist = cat(S_1.nextlist, cat(N.nextlist, S_2.nextlist))\}$

$S \rightarrow \underline{\text{if}}\ E\ \underline{\text{then}}\ M S_1$

$\quad \{backpatch(E.truelist, M.label);$

$\quad S.nextlist = cat(E.falselist, S_1.nextlist)\}$

$S \rightarrow \underline{\text{while}}\ M_1 E\ \underline{\text{do}}\ M_2 S_1$

$\quad \{backpatch(S_1.nextlist, M_1.label);$

$\quad backpatch(E.truelist, M_2.label);$

$\quad S.nextlist = E.falselist;$

$\quad make\_code(goto, M_1.label)\}$

$S \rightarrow$ $\underline{\text{begin}}$ L $\underline{\text{end}}$

$\{S.nextlist = L.nextlist\}$

$S \rightarrow A$

$\{S.nextlist = nil\}$

$L \rightarrow L_1; MS$

$\{backpatch(L_1.nextlist, M.label);$

$L.nextlist = S.nextlist\}$

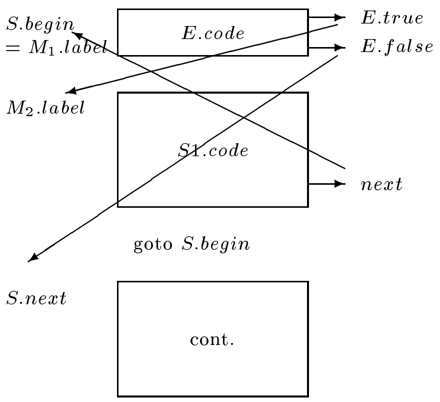
$L \rightarrow S$

$\{L.nextlist = S.nextlist\}$

Exemplarisch soll hier nur die while-Anweisung genauer erklärt werden.

$$S \rightarrow \ \underline{\text{while}}\ M_1 E\ \underline{\text{do}}\ M_2 S$$

- M_1 und M_2 liefern über das Attribut *.label* jeweils den Index der ersten Code-Zeile von $E.code$ bzw. $S.code$.



- Alle Sprünge in $S_1.nextlist$ können jetzt mit $M_1.label$ nachgebessert werden, ebenso alle Sprünge in $E.truelist$ mit $M_2.label$:

$$backpatch(S_1.nextlist, M_1.label);$$

$$backpatch(E.truelist, M_2.label);$$

- Die Liste $S.nextlist$ ist die Liste $E.falselist$.
- Da $E.code$ und $S_1.code$ schon erzeugt worden sind, muß nur noch der Rücksprung erzeugt werden:

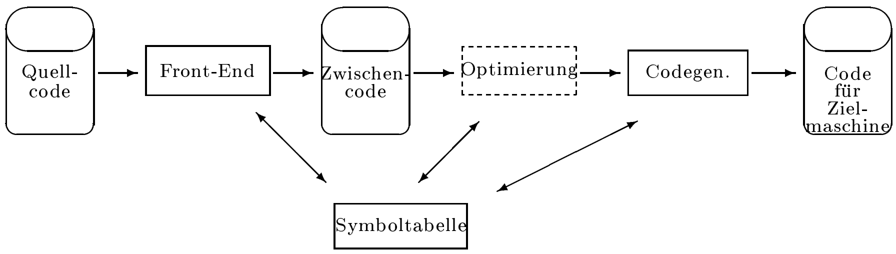
$$S.nextlist = E.falselist;$$

$$make\_code(goto\ M_1.label)$$

Kapitel 7

Code-Erzeugung

7.1 Aufgaben und Teilprobleme der Code-Erzeugung



Das Compiler Front-End hat eine Zwischendarstellung und eine Symboltabelle erzeugt. Auf der Basis dieser Daten soll das Back-End Code für eine gegebene Zielmaschine (Target Machine, *TM*) erzeugen. Die *Optimierung* auf der Ebene der Zwischendarstellung wird in dieser Vorlesung nicht weiter behandelt. Der Codegenerator kann selbst schon gewisse Vorteile zur Erzeugung eines effizienten Codes ausnutzen.

Ausgabe des Codegenerators:

Gegeben: Zwischendarstellung (TAC, Postfix, DAG, AST, Stackmaschinen-Code, ...) und eine Symboltabelle.

Gesucht: Für eine konkret definierte Zielmaschine soll eine Code-Sequenz erzeugt werden, die

- die gleiche Semantik hat wie die Zwischendarstellung,
- die möglichst effizient ist.

Die Ausgabe des Codegenerators kann verschiedene Formen haben:

- Absoluter Maschinencode (direkt ladbar).
- Verschiebbarer Maschinencode (Object für Linker).
- Assembler (symbolischer Maschinencode).

Die Symboltabelle enthält die nötigen Informationen, um jede dieser Formen erzeugen zu können. Das Ziel, effizienten Code zu erzeugen, ist weiter zu präzisieren:

- Kleinste mögliche Laufzeit.
- Geringsten Speicherplatzbedarf für Datenstrukturen und Programm.

Wenn man diese Ziele genau definiert, ergibt sich ein Optimierungsproblem, das nicht entscheidbar, also nicht algorithmisch lösbar ist. Aber auch die meisten Teilprobleme der Code-Erzeugung führen auf Optimierungsprobleme, die meistens NP-vollständig sind, und das bedeutet, daß es extrem unwahrscheinlich ist, dafür effiziente Algorithmen zu finden.

Die Konsequenz aus diesen Ergebnissen ist, daß man nicht erwarten kann, wirklich optimale Lösungen zu berechnen; aber daß es sich lohnt, heuristische Verfahren aufzufinden und miteinander zu vergleichen. Solche Verfahren beruhen auf gewissen Annahmen und Strategien, die als erfolgreich angesehen werden. Das Gebiet der Code-Optimierung (die ja nur suboptimale Lösungen liefert) läßt daher viel Spielraum für Kreativität und lokale Verbesserungen.

Es sollen jetzt kurz die wesentlichen Teilaufgaben der Code-Erzeugung angegeben werden.

1. Speicherabbildung:
   - Organisation der Datenstrukturen in der gegebenen Speicherstruktur.
   - Größe und Selektoren für Datenstrukturen sind in der Symboltabelle auf etwas abstrakterer Ebene schon gegeben.
   - Die Speicherstruktur der Zielmaschine ist wesentlich geprägt durch die Adressierungsarten der CPU und deren Registerstruktur.
2. Auswahl von Maschinen-Instruktionen:
   - Für jede TAC-Instruktion gibt es i.a. viele Möglichkeiten, diese in eine Sequenz von Maschinen-Instruktionen abzubilden, die die gewünschte Semantik haben. Insbesondere Maschinen mit umfangreichem komplexem Instruktionssatz (CISC-Maschine) bieten hier eine kombinatorisch sehr große Zahl von Möglichkeiten. Etwas eingeschränkt sind diese Möglichkeiten bei den RISC (<u>r</u>educed <u>i</u>nstruction <u>s</u>et <u>c</u>omputer)-Maschinen.
   - Wenn man ganze TAC-Sequenzen auf Maschinencode abbildet, werden die Möglichkeiten nochmals stark erweitert.
3. Registerzuweisung (eigentlich ein Spezialfall der Speicherabbildung).

   Da Operationen mit Maschinen-Registern sehr schnell ausgeführt werden können, sollten häufig verwendete TAC-Register (Variablen) und insbesondere temporäre Register so weit wie möglich auf Maschinenregister (CPU-Register) abgebildet werden. Auch dieses Problem ist kombinatorisch sehr aufwendig (exponentieller Suchraum) und für optimale Lösungen ist das Problem NP-vollständig. Hinzu kommen oft noch besondere Einschränkungen der Registerbenutzung (Stack-Pointer, Index-Register u.s.w.).
4. Lokale Transformationen auf TAC-Sequenzen. Durch semantikerhaltende Transformationen auf Abschnitten der TAC-Sequenz kann man u.U. zu besseren Maschinencode-Sequenzen gelangen. Auch für dieses Teilproblem ist die Optimierung ein NP-vollständiges Problem.

7.2 Struktur einer Zielmaschine

Eine Zielmaschine wird durch folgende Eigenschaften spezifiziert:

- Registerstruktur:

  - Anzahl, Größe, Verwendung.

- Instruktionssatz und -format:
  - Funktionalität.
  - Operationscodes.
  - Adressierungsarten.
  - Ausführungszeiten.
- Speicherstruktur (Adressierung):
  - Linear oder segmentiert.
  - Stack- und Heap-Organisation.
  - Wortlänge.
  - Datenausrichtung (alignment).

Für die exemplarische Behandlung von bestimmten Methoden der Code-Erzeugung wird hier vereinfachend eine (hypothetische) Zielmaschine eingeführt. Die Spezifikation ist nur so ausführlich wie notwendig.

Spezifikation:

- Register: n Register: $R(0), ..., R(N-1)$
  Größe: 4 Bytes (32 Bit).
- Speicherstruktur:
  - Linearer Adreßraum: 2^{32} Bytes (4 GB).
  - Adressierbare Einheit: 1 Byte.
  - Adresse benötigt 32 Bit= 4 Byte Speicherplatz.
- Instruktionssatz und Format:

  Format: ⟨op_code⟩ ⟨source⟩ ⟨destination⟩.

  Notation: $op\ s, d$.

  op ist ein Operationscode.

  s Adresse Quelle.

  d Adresse Ziel.

Die Adressen und ihre Interpretation werden bei den Adressierungsarten genauer beschrieben. Es werden nur einige Instruktionen erklärt und weitere werden dann eingeführt, wenn sie benötigt werden:

- Datentransport:

$$MOV\ s, d \qquad cont(s) \rightarrow d$$

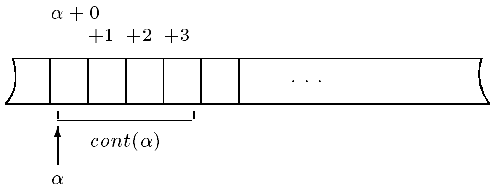
- Arithmetik:

$$\begin{array}{rl} ADD\ s, d & cont(s) + cont(d) \rightarrow d \\ SUB\ s, d & cont(s) - cont(d) \rightarrow d \\ \vdots & \end{array}$$

- Sprünge:

$GOTO\ p$	(p ist Programmadresse)
$JCD\ p$	bedingter Sprung hängt vom Statusregister ab
$CALL\ p$	Unterprogrammaufruf
RET	Rücksprung

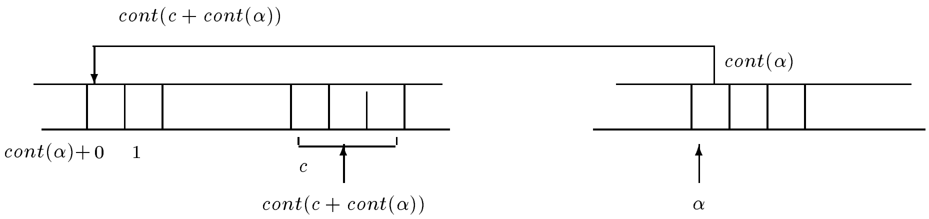
- Adressierung: Sei α eine Adresse (32-Bit-Wert), dann sei $cont(\alpha)$ der Wert des Wortes (4 Bytes), das im Speicher bei der Adresse α beginnt:



$cont(\alpha)$ ist also wieder ein 32-Bit-Wert. α kann ein 32-Bit-Wert sein oder ein Registername. Ist α Registername, dann ist $cont(R_i)$ der 32-Bit-Wert des Registers R_i.

Der Inhalt unter einer Adresse kann selbst wieder als Adresse betrachtet werden: $cont(cont(\alpha))$ ist also der Inhalt des Speicherwortes an der Adressse $cont(\alpha)$. Ist c eine (32-Bit) Konstante und α eine Adresse, dann bedeutet:

$$cont(c + cont(\alpha))$$



Eine Instruktion besteht mindestens aus einem 32-Bit-Wort, das den Operationscode und bis zu zwei Registernamen (Nummern) enthält. Eine volle 32-Bit-Adresse für source und/oder destination erfordert dann ein oder zwei zusätzliche 32-Bit-Wörter. Aus dieser Darstellung ergeben sich dann auch die unterschiedlich langen Ausführungszeiten (*Kosten*).

Damit werden die folgenden Adressierungsarten und Kosten definiert:

Adressierung	Notation	Adresse	zus. Kosten	Ref. Stufe
immediate	$\#C$	$-$	1	0
memory	M	M	1	1
register	R	R	0	1
indexed	$c(R)$	$c + cont(R)$	1	1
ind. register	$*R$	$cont(R)$	0	2
ind. indexed	$*c(R)$	$cont(c + cont(R))$	1	2

Beispiel:

$MOV\ *4(R0), M$

Bedeutung: $cont(cont(4 + cont(R0))) \rightarrow M$

Kosten: 3 Speicherwörter für:

1. MOV und $R0$
2. $c = 4$ (32-Bit Konstante)
3. M (32-Bit Speicheradresse)

Die TAC-Instruktion:

$$a := b + c \quad (+, a, b, c)$$

kann auf verschiedene Weise in kurze Sequenzen von Maschinen-Instruktionen übersetzt werden:

1. Kosten insgesamt 6.

   $MOV\ b, R0$

   $ADD\ c, R0$

   $MOV\ R0, a$

2. Kosten insgesamt 6.

   $MOV\ b, a$

   $ADD\ c, a$

3. Kosten insgesamt 2.

   $MOV\ *R1, *R0$

   $ADD\ *R2, *R0$

Für die Version 3. wurde von folgender Voraussetzung ausgegangen:

$cont(R0) = l\_val(a)$,

$cont(R1) = l\_val(b)$,

$cont(R2) = l\_val(c)$ (Adressen).

`Ende Beispiel`

Das Beispiel zeigt, daß es i.a. viele Möglichkeiten geben kann, eine einzelne TAC-Instruktion zu übersetzen. Wenn man zusätzliche Voraussetzungen schaffen kann (möglichst viele Operanden in CPU-Registern), dann kann man auch die Kosten (Ausführungszeiten) gering halten. Dabei sind die Antworten auf die folgenden Fragen wichtig:

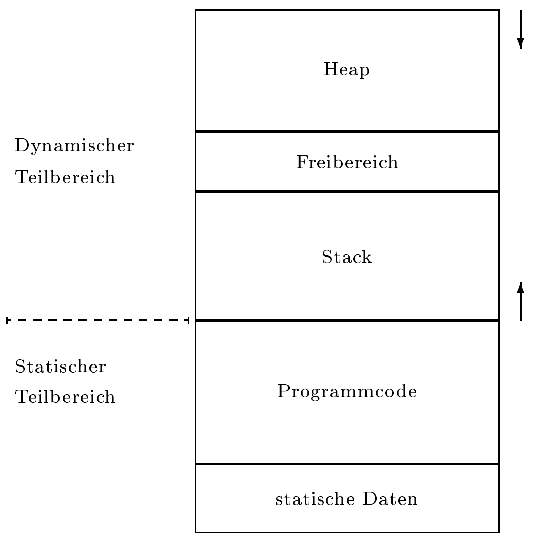
- Wie sind die Register gerade geladen, d.h. welche aktuellen L- oder R-Werte von TAC-Namen (Variablen) sind gerade in welchen Registern?
- Welche dieser aktuellen Registerinhalte werden in der Zukunft noch benötigt und welche definitiv nicht?

Auf die Behandlung dieser Informationen werden wir in Abschnitt 7.3 genauer eingehen.

Die Organisation von Unterprogrammen ist am Modell der Stackmaschine im Begleitprojekt ausführlich behandelt worden. Daher soll hier noch kurz eine andere Technik für Unterprogramme erklärt werden.

Wir gehen aus von einer Einteilung des Speichers in Bereiche für:

- Statische Daten.
- Programm-Code.
- Dynamische Daten:
  - Stack,
  - Heap.



Der Heap ist dynamisch (zur Laufzeit) vergebener Speicherplatz (etwa in C durch malloc() oder in PASCAL durch new()) für dynamische Datenstrukturen (Listen, Bäume, ...). Der Stack ist Arbeitsspeicher der zur Speicherung von temporären Daten verwendet werden kann. Die statischen Anteile sind der Maschinencode und statisch (zur Compile-Zeit) vergebener Speicherplatz für globale Daten.

Man kann aber auch die lokalen Variablen von Prozeduren im statischen Bereich anordnen. Diese Technik schränkt allerdings den Gebrauch von rekursiven Prozeduren drastisch ein.

Wir gehen von TAC-Instruktionen für Unterprogramme aus:

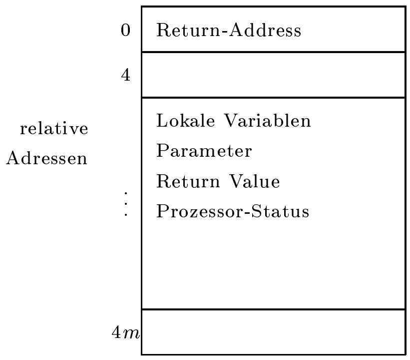
- *call* p: p symb. Prozedurname, Verweis auf Symboltabelle,
- *return*: Rückkehr zur TAC-Instruktion nach dem *call*.

Zu jeder deklarierten Prozedur wird ein Datenbereich definiert, der für diese Prozedur spezifisch ist. Dieser Datenbereich heißt auch *activation record* (*AR*) und entspricht den Stackframes der Maschine SM1 im Begleitprojekt.

Ein AR enthält i.a. die folgenden Daten:

- Lokale Variablen,
- Parameter,
- Rückkehradresse,
- Prozessor-Status,
- Rückgabewert (Funktion).

AR-Struktur (Beispiel):



Man kann nun zu jeder Prozedur p einen AR im Bereich der statischen Daten zur Compile-Zeit anlegen (z.B. FORTRAN-Compiler). Ist p der symbolische Prozedurname auf TAC-Ebene, dann hat p die Attribute:

$p.ar$ Anfangsadresse des AR,

$p.code\_addr$ Anfangsadresse der Code-Sequenz von p.

Damit kann man jetzt die Übersetzung von *call* und *return* einfach beschreiben.

TAC	Maschinencode
	(here:) $MOV\ \#here + c, p.ar$
	$\vdots$
	$GOTO\ p.code\_addr$
call p	$\vdots$
	$\vdots$
	$(+c:)$ (Rücksprungadresse)
return	$GOTO\ {*p.ar}$

Zur Übung sollte man sich überlegen, wie die etwas aufwendigere Form im TAC:

$arg\ a_1$

$\vdots$

arg a_n

$a :=$ *call* p, n

zu übersetzen ist, wenn außerdem noch ein Prozessor-Statusregister R_S existiert, das beim Aufruf in den AR transportiert werden muß, und das beim Rücksprung wieder geladen werden muß.

Die statische Zuordnung der AR's erlaubt natürlich nur bedingt rekursive Aufrufe, da die lokalen Variablen nur einmal existieren und nicht bei jedem Aufruf neuer Platz für diese Variablen angelegt wird. Die Stackorganisation ist daher für rekursive Prozeduren besser geeignet.

7.3 Kontrollflußgraph und Code-Erzeugung für Basisblöcke

TAC ist der Drei-Adreß-Code für die Zwischendarstellung und entsprechend sei TC der Machinen-Code der Zielmaschine (Targetcode). Wir betrachten TAC und TC als die Mengen der TAC-Instruktionen oder die Menge der TC-Instruktionen. Außerdem nehmen wir an, daß man die gesamte Information der Symboltabelle als ein Element $s \in ST$ der Menge aller möglichen Inhalte der Symboltabelle auffassen kann.

Der Codegenerator ist dann (semantisch) eine Abbildung:

$$\varphi :\ \mathrm{ST}\ \times\ \mathrm{TAC}^* \rightarrow\ \mathrm{TC}^*$$

und für eine gegebene Symboltabelle $s \in ST$

$$\varphi_s : \mathrm{TAC}^* \rightarrow\ \mathrm{TC}^*; \varphi_s(u) = \varphi(s, u).$$

Eine sehr einfache Möglichkeit, eine solche Abbildung zu realisieren, besteht darin, daß man sie zunächst nur auf TAC definiert und dann auf TAC* kanonisch forsetzt:

$$\tilde{\varphi} : \mathrm{ST} \times \mathrm{TAC} \rightarrow \mathrm{TC}^*$$

wird definiert und dann setzt man:

$$\varphi(s, \sigma_1, ..., \sigma_r) = \tilde{\varphi}_s(\sigma_1)...\tilde{\varphi}_s(\sigma_r); \quad \varphi(s, \epsilon) = \epsilon.$$

Man übersetzt also jede TAC-Instruktion nach einem festen Muster, ohne daß man den Kontext betrachtet, in dem die Instruktion vorkommt. Die Kosten kann man ebenfalls für jede einzelne Maschineninstruktion $\gamma \in$ TC festlegen:

$$\tau : \mathrm{TC} \rightarrow \mathbb{R}_{\geq 0}$$

und dann auf Code-Sequenzen fortsetzen:

$$\overline{\tau} : \mathrm{TC}^* \rightarrow \mathbb{R}$$

durch $\overline{\tau}(\epsilon) = 0$, $\overline{\tau}(\gamma_1...\gamma_m) = \sum_{i=1}^{m} \tau(\gamma_i)$. Man beachte aber, daß dieses Maß nicht die Laufzeit korrekt beschreibt, da der Kontrollfluß (z.B. Schleifen) nicht berücksichtigt wird. Wenn die Kostenfunktion aber den Speicherplatz je Instruktion beschreibt, dann erhält man mit $\overline{\tau}$ den vom Maschinenprogramm belegten Speicherplatz (statische Komplexität).

Um den Kontext einer zu übersetzenden TAC-Instruktion besser zu berücksichtigen kann man eine gegebene TAC-Sequenz

$$\overline{\sigma} = \sigma_1\sigma_2...\sigma_n$$

in geeignete Abschnitte zerlegen:

$$\overline{\sigma} = \underbrace{\sigma_1...\sigma_{m_1}}_{Block:1} \; \underbrace{\sigma_{m_1+1}...\sigma_{m_2}}_{2} \cdots \underbrace{\sigma_{m_{k-1}+1}...\sigma_{m_k}}_{k} .$$

Dann kann man versuchen, solche Abschnitte jeweils insgesamt möglichst effizient zu übersetzen. Man nennt die Abschnitte *Blöcke*, und diese Blöcke sollen die Struktur des Kontrollflusses geeignet respektieren. Die Blöcke sollen daher die folgenden Eigenschaften haben:

- Ein Block wird stets an der ersten Instruktion begonnen (*Eingang*, Anfang).
- Ein Block kann nur an der letzten Instruktion verlassen werden (*Ausgang*).
- Der Block sollte mit diesen Eigenschaften eine maximale Länge haben.

Das führt zu dem Begriff der *Basisblöcke*. Sei $\overline{\sigma} = \sigma_1...\sigma_n \in \text{TAC}^*$.

σ_i heißt *Eingang* (bzgl. $\overline{\sigma}$) wenn gilt: $i = 1$

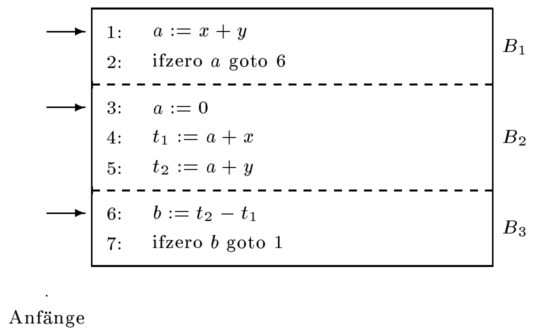
oder: σ_i ist Ziel einer Sprunginstruktion von $\overline{\sigma}$

oder: σ_{i-1} ist Sprunginstruktion.

Jeder Eingang markiert den Beginn eines *Basisblockes* und dieser erstreckt sich bis zum nächsten Eingang (ausschließlich). Mit dieser Definition ist auch ein einfacher Algorithmus skizziert, wie man eine TAC-Sequenz in Basisblöcke zerlegen kann. Basisblöcke sind maximale Teilsequenzen, in denen die Kontrolle nicht verzweigt, sondern streng linear verläuft.

Wir betrachten dazu zunächst ein einfaches Beispiel.

Beispiel: Zerlegung in Basisblöcke



1: ist immer ein Anfang,

3: ist Anfang weil 2: Sprunginstruktion,

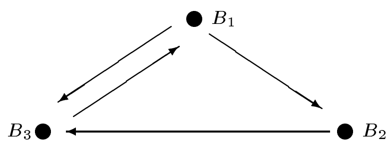
6: ist Anfang weil 2: nach 6 verzweigt.

`Ende Beispiel`

Durch die Sprunginstruktion und die Reihenfolge, in der die Blöcke angeordnet sind ergibt sich zwischen den Blöcken eine Relation, die durch den Kontrollfluß definiert ist. Seien B_i und B_j Blöcke und sei σ_{i,m_i} die letzte Instruktion von B_i und $\sigma_{j,1}$ die erste Instruktion von B_j (Anfang), dann definiert man

$$B_i \to B_j \iff \sigma_{j,1} \text{kann unmittelbar nach } \sigma_{i,m_i} \text{ ausgeführt werden.}$$

Beispiel:(Fortsetzung)



`Ende Beispiel`

Die Relation $\to$ auf den Blöcken definiert einen gerichteten Graphen, dessen Knoten die Basisblöcke sind. Dieser Graph heißt der *(Kontroll-) Flußgraph* zur TAC-Sequenz $\overline{\sigma}$. Er beschreibt auf abstrakter Ebene den Kontrollfluß innerhalb der TAC-Sequenz. Dieser Graph liefert Informationen darüber, welche Werte in einem Block B_i berechnet werden und welche später in einem „nachfolgenden" Block B_j noch verwendet werden. Diese Struktur ist für die Zuweisung von Registern wichtiger als die lineare Struktur von $\overline{\sigma}$.

Wir wollen jetzt das Problem betrachten, die Basisblöcke möglichst günstig auf TC-Sequenzen abzubilden. Insofern ist die Sicht immer noch „lokal" auf die Basisblöcke beschränkt. Der Kontext der Basisblöcke wird noch nicht berücksichtigt. Eine solche „globale" Sicht wird bei der Code-Optimierung eingenommen, die aber im Rahmen dieser Vorlesung nicht mehr behandelt wird.

Ein erster Ansatz zur verbesserten Code-Erzeugung für Basisblöcke besteht darin, den jeweiligen Block nach bestimmten Regeln so umzuformen, daß die Semantik erhalten bleibt aber effizienterer Code erzeugt werden kann, weil z.B. die Register besser genutzt werden können. Einige dieser Transformationen sollen kurz angedeutet werden.

- Eliminieren gemeinsamer Teilausdrücke:

  $a := b + c$ ← gleiche Form aber zwei
  $b := a - d$ ← verschiedene Werte, da
  $c := b + c$ ← b verändert wurde
  $d := a - d$ ← gleiche Berechnung, muß nur einmal ausgeführt werden

  wird daher transformiert zu:

  $a := b + c$
  $b := a - d$
  $c := b + c$

$d := b$ (Rechenoperation gespart).

- Entfernen von nutzlosem Code (Dead Code):

  $i : x := y + z$
  und x kommt in keiner nachfolgenden Instruktion des Blockes auf der rechten Seite vor: dann kann man die ganze Zeile streichen.

- Vertauschen von TAC-Instruktionen:

  $t_1 := b + c \; x \neq t_1 \neq y,$

  $t_2 := x + y \; b \neq t_2 \neq c.$

  Wenn die Namen die aufgeführten Bedingungen erfüllen, kann man die Zeilen vertauschen (Ziel: eventuell bessere Nutzung von Registern).

- Umbenennung von temporären Variablen (Namen): Eine temporäre Variable kann in einem Block überall durch einen anderen neuen Namen benannt werden.

Die Variablennamen in TAC-Instruktionen sollen jetzt einfach „Namen" heißen. Sie haben L- und R-Werte. Den Namen werden Speicheradressen zugeordnet, die wir aber symbolisch dann mit diesen Namen bezeichnen. R-Wert und L-Wert von Namen können temporär in Registern gespeichert werden. Ein Name kommt in einer TAC-Instruktion „definiert" und „verwendet" vor: In

$$a := x + y$$

kommt a *definiert* vor und es kommen x und y *verwendet* vor.

Ein Name x heißt in einem Block B in der Zeile (Instruktions-Index) i *lebendig* wenn er in einer nachfolgenden Zeile des Blocks ($j > i$) noch verwendet wird. Ein Name heißt *lebendig* beim Verlassen eines Blockes wenn er in einem erreichbaren Block verwendet wird.

Für die Zuteilung von Registern ist es wichtig zu wissen, wann ein Name, der in einer Instruktion σ_i vorkommt als nächstes wieder verwendet wird. Diese Eigenschaften (lebendig, nächste Verwendung) können durch Analysealgorithmen bestimmt werden. Eine solche Analyse kann für Basisblöcke getrennt (lokal) oder aber blockübergreifend (global) durchgeführt werden. Für die globale Analyse spielt der Flußgraph eine wichtige Rolle.

Zum Schluß dieses Abschnittes soll ein Verfahren zur Code-Erzeugung skizziert werden, das die Basisblöcke in möglichst effizienten Maschinencode übersetzen soll.

Verfahren zur Code-Erzeugung (Skizze):

Annahmen:

- Die Operationen der TAC-Instruktionen haben jeweils eine Entsprechung bei den TC-Instruktionen (Maschinen-Instruktionen),
- Werte können in Registern gehalten werden, bis ein Basisblock verlassen wird.

Strategie:
Wenn ein Block verlassen wird, werden alle R-Werte von Namen, die aktuell nur in einem Register gespeichert sind in die Speicherplätze (L-Werte) übertragen.

Datenstrukturen:
Man legt Tabellen an (das kann in der Symboltabelle geschehen), die für einen Block jeweils folgende Informationen enthalten:

- Zu jedem Register wird angegeben, welcher Name gerade in diesem Register dargestellt wird. Der Inhalt dieses Registerdeskriptors kann sich von Code-Zeile zu Code-Zeile verändern.
- Zu jedem Namen wird angegeben, in welchen Speicherplätzen oder Registern der aktuelle R-Wert zu diesem Namen gespeichert ist: *Adressen-Deskriptor*.

Algorithmus:

Gegeben: TAC-Sequenz $\overline{\sigma}$ und ein Basisblock B.

Gesucht: TC-Sequenz für diesen Block (möglichst effizient).

Es werden nur die Schritte für eine TAC-Instruktion der Form:

$$x := y \; op \; z$$

beschrieben. (Die Übersetzung anderer Formen bleibt dem Leser überlassen).

1. Bestimme einen Speicherplatz (L-Wert) für das Resultat. Falls ein Register frei ist, verwende ein freies Register. Sei L_x dieser Speicherplatz (Speicheradresse oder Register).
2. Der Adreß-Deskriptor beschreibt die L-Werte (Speicherplätze) in denen die R-Werte von y und z zu finden sind. Sei y' der L-Wert für y, der möglichst ein Register sein sollte. Falls $y' \neq L_x$, dann erzeuge die TC-Instruktion

$$MOV \; y', L_x.$$

3. Sei z' der L-Wert für z (möglichst Register) dann erzeuge:

$$OP \; z', L_x.$$

4. Modifiziere Adreß-Deskriptor:

   Füge hinzu: L_x enthält R-Wert von x.

   Falls L_x ein Register ist modifiziere Register-Deskriptor: L_x enthält aktuellen R-Wert von x.
5. Falls der Wert von y im Block zukünftig nicht mehr gebraucht (verwendet) wird, modifiziere Register-Deskriptor:
   Wenn y einem Register zugeordnet ist, dann ist dieses Register jetzt frei. Analog für z.

Die anderen Formen von TAC-Instruktionen werden ähnlich behandelt. Sprunginstruktionen werden durch eine Compare-Instruktion (CMP) mit anschließendem Sprung realisiert. Aber zuvor müssen alle R-Werte, die aktuell nur in Registern gespeichert sind in die zugehörigen Speicherplätze zurückgeschrieben werden.

Literaturverzeichnis

[1] Aho, A.V., Sethi, R., Ullman, J.D. *Compilerbau; Teil 1 und Teil 2.* Addison-Wesley, Bonn, 1989.

[2] Bennett, J.P. *Introduction to Compiling Techniques.* Teubner, Stuttgart, 1990.

[3] Doberkat, E.-E., Fox, D. *Praktischer Übersetzerbau.* Teubner, Stuttgart, 1990.

[4] Grune, D., Jacobs, C.J.H. *Parsing Techniques, a practical guide.* Ellis Horwood, New York, 1990.

[5] Herold, H. *lex und yacc — Lexikalische und syntaktische Analyse.* Addison-Wesley, Bonn, 1992.

[6] Holmes, J. *Object-Oriented Compiler Construction.* Prentice Hall, Englewood Cliffs, 1995.

[7] Jobst, F. *Compilerbau, Von der Quelle zum professionellen Assemblertext.* Hanser, München, 1992.

[8] Kastens, U. *Übersetzerbau.* Oldenbourg, München, 1990.

[9] Klöppel, B.; Paul, M.; Rauch, J.; Ruhland, J. *Compilerbau, Am Beispiel der Sprache SIMPL.* Vogel, Würzburg, 1991.

[10] Koskimies, K.; Paakki, J. *Automating Language Implementation, a pragmatic approach.* Ellis Hoewood, New York, 1990.

[11] Lee, P. *Realistic Compiler Generation.* MIT-Press, Cambridge, MA., 1989.

[12] Lemone, K.A. *Design of Compilers, Techniques of Programming Language Transl.* CRC Press, Boca Raton, 1992.

[13] Naumann, S.; Langer, H. *Parsing.* Teubner, Stuttgart, 1994.

[14] Penner, V. *Konzepte und Praxis des Compilerbaus, eine Einführung mit Diskette.* Vieweg, Braunschweig, 1994.

[15] Schmitz, L. *Syntaxbasierte Programmierwerkzeuge.* Teubner, Stuttgart, 1995.

[16] Schreiner, A.T.; Friedmann, G. *Compiler bauen mit UNIX, Eine Einführung.* Hanser, München, 1985.

[17] Tremblay, J.-P.; Sorensen, P.G. *The Theory and Practice of Compilerwriting.* McGraw-Hill, New York, 1985.

[18] Waite, W.M.; Goos, G. *Compiler Construction.* Springer, New York, 1989.

[19] Wilhelm, R.; Maurer, D. *Übersetzerbau, Theorie, Konstruktion, Generierung.* Springer, New York, 1985.

[20] Wirth, N. *Compilerbau.* Teubner, Stuttgart, 1984.

[21] Zima, H. *Compilerbau; Band 1 und Band 2.* BI, Mannheim, 1989.