Compilerbau – eine Einführung
SS1998
Version 2.0.0

Prof. Dr. Michael Jäger

21. März 1998

Inhaltsverzeichnis

1 Einführung **6**

1.1 Wozu Compilerbau ? 6

1.2 Phasenmodell eines Compilers 7

1.2.1 Verschränkung der Phasen 8

1.2.2 Arbeitsteilung bei der Compilerentwicklung 9

1.3 Bedeutung formaler Sprachbeschreibung 9

2 Einführung in Formale Sprachen **11**

2.1 Mathematische Grundlagen 11

2.2 Kontextfreie Grammatiken 12

2.2.1 Ableitbarkeit 13

2.2.2 Notationsvarianten für Grammatiken 16

2.3 EBNF 17

2.4 Transformation von EBNF in Grammatiken 18

2.5 Formale Sprachen und Compilerbau 20

3 Lexikalische Analyse (Scanner) **22**

3.1 Lexikalische Elemente einer Programmiersprache 22

3.2 Spezifikation der Token-Syntax 24

3.2.1 Reguläre Sprachen 24

3.2.2 Reguläre Ausdrücke 24

3.2.3 Reguläre Ausdrücke – UNIX-Notation 25
3.3 Scanner-Implementierung 27
3.4 Systematische Scanner-Implementierung 31
3.4.1 Endliche Automaten / Endliche Akzeptoren 31
3.4.2 Darstellung von endlichen Automaten 32
3.4.3 Automaten als Akzeptoren für formale Sprachen . . . 33
3.4.4 Vom regulären Ausdruck zum nichtdeterministischen Akzeptor – Thompson-Algorithmus 34
3.4.5 Vom nichtdeterministischen zum deterministischen Akzeptor 36
3.4.6 Minimierung endlicher Automaten 37
3.5 Scanner-Generatoren 39
3.5.1 Verwendung von *lex* 39

4 Syntaxanalyse **45**
4.1 Strategien zur Berechnung des Ableitungsbaums 45
4.2 Mehrdeutige Grammatiken, Präzedenz und Assoziativität . . 45
4.2.1 Präzedenzen und Assoziativitätsregeln in Programmiersprachen 46
4.2.2 Präzedenz und Assoziativität in der Grammatik . . . 48
4.2.3 Grammatik-Transformationen für Top-Down-Analyse 49
4.3 Top-Down-Verfahren 50
4.3.1 Nichtdeterministische Spezifikation einer Top-Down-Analyse: 50
4.3.2 Top-Down-Analyse durch rekursiven Abstieg 52
4.3.3 Top-Down-Analyse mit tabellengesteuertem LL(1)-Parser 53
4.3.4 LL(k) - Verfahren bei mehrdeutigen Grammatiken . . 56
4.4 Bottom-Up-Syntaxanalyse (LR-Verfahren) 57
4.4.1 LR-Parser-Schema 60
4.5 Fehlerbehandlung 61
4.6 Wertung der Analyseverfahren 62
4.7 Der Parser-Generator *yacc* 62
4.7.1 Das Eingabe-Dateiformat 63
4.7.2 „Debuggen“ einer Grammatik 65

4.7.3 „Debuggen“ des generierten Parsers 65
4.8 Fehlerbehandlung mit yacc/bison 67

5 Semantische Analyse **71**
5.1 Programmiersprachliche Grundlagen 71
5.1.1 Typen 71
5.1.2 Deklarationen 72
5.1.3 Sichtbarkeit von Bezeichnern 72
5.1.4 Polymorphismus, Überladungen, Konversionen 73
5.1.5 Typprüfung 74
5.2 Syntaxorientierte Spezifikation – Syntaxorientierte Übersetzung 74
5.3 Attributierte Grammatiken 75
5.3.1 Vererbte und synthetische Attribute 75
5.3.2 yacc-Notation für Attribute 76
5.3.3 Beispiel für eine attributierte Grammatik: Roboter . . 77
5.4 Semantische Aktionen und Übersetzungschemata 78
5.4.1 Ausführung der Aktionen durch Top-Down-Parser . . 78
5.4.2 Ein Übersetzer für Ausdrücke vom Infix- in das Postfixformat 79
5.4.3 Ausführung der Aktionen durch Bottom-Up-Parser . . 80
5.5 Syntax-orientierte Spezifikationstechniken im Vergleich 80
5.5.1 Implementierung des Roboter-Beispiels mit yacc . . . 83
5.6 Symboltabelle 86
5.6.1 Symboltabellen-Einträge 86
5.6.2 Operationen 88
5.6.3 Symboltabellen und Gültigkeitsbereiche 88
5.6.4 Implementierung 89

6 Laufzeitsysteme **90**
6.1 Programmiersprachliche Grundlagen 90
6.1.1 Das Environment in Pascal 90
6.2 Speicherverwaltung 92
6.2.1 Speicheraufteilungsschema 92
6.2.2 Speicher-Layout für lokale Daten 93

6.2.3 Aktionen beim Unterprogramm-Aufruf 93
6.3 Zugriff auf globale Adressen 94

7 Code-Erzeugung **95**
7.1 Einführung in die Maschinencode-Generierung und Optimierung 95
7.1.1 Abstrakte Maschinen und Zwischencode-Generierung . 95
7.1.2 Code-Erzeugung für Stack-Maschinen 95
7.1.3 Zielrechner-Architektur 97
7.1.4 Grundlagen der Codeerzeugung für die virtuelle RISC-Maschine 100
7.1.5 Erste Optimierungen – Verzögerte Code-Erzeugung . . 101

8 Literatur **104**

A Grammatik für Oberon-0 in *yacc*-Eingabeformat **104**

Vorbemerkung

Das vorliegende Skript wird ständig weiterentwickelt. Ich bitte darum, mich auf Fehler aufmerksam zu machen (am besten per E-Mail).

Die für die praktischen Übungen im SS98 verwendete Quellsprache Oberon-0 und die Definition der virtuellen Zielmaschine ist aus [Wirth96] entnommen.

Die Implementierung des Oberon-0-Compilers in den praktischen Übungen orientiert sich dagegen nicht an dem von Wirth vorgeschlagenen Weg (manuell programmierter rekursiver Abstieg), sondern verwendet einen Compilergenerator (*yacc* oder *bison*).

1 Einführung

Compilerbau gehört heute zum „Grund-Handwerkszeug“ jedes Informatikers. Es ist eine klassische Informatik-Disziplin, die sich bis in die achziger Jahre hinein rasant weiterentwickelt hat. Aufsehen erregende Weiterentwicklungen gibt es in den letzten Jahren weder bei den Programmiersprachen noch bei deren Implementierung.

Compilerbau ist sicher eines der am besten erforschten Gebiete innerhalb der Informatik. Solide theoretische Grundlagen, brauchbare formale Beschreibungsmodelle, effiziente, gut dokumentierte Algorithmen und brauchbare Implementierungs-Tools (Compiler-Generatoren) stehen dem Anwender zur Verfügung. War es 1975 noch einen Doktorhut wert, einen Pascal-Compiler zu entwickeln, konnte man dies 1980 schon als Praktikumsaufgabe jedem Informatik-Studenten zumuten.

1.1 Wozu Compilerbau ?

Obwohl nur wenige jemals Compilerbau im engeren Sinne betreiben, also einen Übersetzter für eine höhere Programmiersprache entwickeln, hat die Disziplin einen festen Platz in jedem Informatik-Lehrplan. Warum ?

Einerseits geht das Anwendungsspektrum des Stoffs weit über die Implementierung von Compilern hinaus. Insbesondere die für den Compilerbau entwickelten Techniken zur automatisierten Analyse hierarchisch strukturierter Dokumente haben unzählige Anwendungsgebiete.

Andererseits enthält das Wissen über die Implementierung einer Programmiersprache wichtige Aspekte zur Verwendung dieser Sprache. Beispielsweise sollte ein Programmierer, der immer mehrere Möglichkeiten zur Codierung hat, diese Möglichkeiten auch im Hinblick auf die Laufzeit-Effizienz gegeneinander abwägen können. Ohne gewisse Grundkenntnisse über den generierten Maschinencode ist diese Abwägung nicht praktikabel. (Man könnte theoretisch alle möglichen Varianten mit Zeitmessung ausprobieren).

Oft wird dem Programmierer auch die Semantik bestimmter Sprachkonzepte erst durch die Betrachtung der Implementierung vollends klar. (Zugegebenermassen ist die Semantik von Programmiersprachen oft genug nur durch deren Implementierung definiert: Die Wirkung des Sprachkonzepts „X“ ist das, was das übersetzte Programm bei Verwendung von „X“ tut !)

Einige Anwendungen für Übersetzerbau-Techniken:

- Compiler für höhere Programmiersprache
- Assembler
- Struktur-Editor (z.B. für HTML)
- WWW-Browser
- Implementierung bzw. Transformation von Text-Markup-Sprachen (SGML, LaTeX)
- SQL-Optimierer im DBMS
- Analysetools für Reverse-Engineering (automatisierte Gewinnung von Strukturinformation aus ungenügend oder gar nicht dokumentierter alter Software)
- Portierungstool zur (teil-) automatisierten Transformation einer Hochsprache in eine andere (oft bei Portierung im Zuge von „Downsizing"-Projekten)
- Interpreter Druckausgabe-Sprachen (Postscript, PDF)
- Interpreter für Roboter-Programme (oder andere Anwender-Sprachen)
- Test-Werkzeuge (zur Ermittlung von Überdeckungskriterien oder zur automatischen Instrumentierung von Programmen für Regressionstests)

1.2 Phasenmodell eines Compilers

Ein Compiler transformiert einen Quelltext einer höheren Programmiersprache in Maschinencode (oder Assembler-Code).

Dieser recht komplexe Prozess wird durch folgendes Phasenmodell beschrieben:

Die **Analysephase** wird unterteilt in

- Syntaxanalyse

  - lexikalische Analyse (Scanner)
    Identifikation von Operatoren, symb. Namen, Schlüsselwörtern usw.
  - Strukturanalyse (Parser)
    Analyse des hierarchischen Programmaufbaus

- Semantische Analyse
  Typprüfung, Berechnung von Typkonversionen u.a.

- Aufbau der internen Programmrepräsentation
  Die interne Darstellung benutzt meist folgende Datenstrukturen

  - Symboltabelle
    Hier wird zu jedem Bezeichner (symbolischer Name für Variable, Funktion o.a.) die vorhandene Information abgelegt
    Bei jeder Verwendung eines Bezeichners wird anhand des Symboltabelleneintrags geprüft, ob die Verwendung im Einklang mit den Bezeichner-Attributen steht.
  - Abstrakter Syntaxbaum
    Bäume werden in der Informatik immer dort verwendet, wo hierarchische Strukturen repräsentiert werden müssen. Der Anweisungsteil eines Pascal-Programms oder ein C-Funktionsrumpf etwa sind hierarchich strukturiert und werden als Baum repräsentiert.
    Ein abstrakter Baum enthält nur solche Informationen, die für die Weiterverarbeitung nötig sind. Das zu einer C-Anweisung gehörende abschließende Semikolon muss zwar im Quelltext stehen, in der internen Programmdarstellung ist es jedoch überflüssig.

Bei der **Synthesephase** erzeugt man oft aus dem Syntaxbaum zunächst einen maschinenunabhängigen Zwischencode. Das ist ein Programm für eine besonders einfach strukturierte virtuelle Maschine. Auf der Zwischcode-Ebene kann optimiert werden (z.B. Erkennung und Eliminierung bestimmter Rekursionsformen, Verkürzung von Sprungketten) bevor der eigentliche Zielcode generiert und danach (maschinenspezifisch) optimiert wird.

Der Zwischencode vereinfacht auch die Implementierung einer Sprache für unterschiedliche Zielmaschinen, denn nur der letzte Transformationsschritt hat maschinenspezifische Varianten.

1.2.1 Verschränkung der Phasen

Das beschriebene Phasenmodell könnte theoretisch in entsprechende Compiler-Phasen umgesetzt werden, die nacheinander aufgerufen werden,

wobei jede Phase beispielsweise über eine Datei ihre Ergebnisse an die nachfolgende Phase weiterleitet:

Der Scanner liest den Quelltext und erzeugt eine Token-Datei, der Parser liest die Token-Datei und erzeugt eine erste interne Darstellung, die semantische Analyse erzeugt durch Berechnung der Typinformation eine erweiterte interne Darstellung usw.

Um die Übersetzung schneller zu machen, werden die Teilfunktionen jedoch weitgehend zeitlich verschränkt. So ruft der Parser bei der Analyse jedesmal, wenn ein neues Token benötigt wird, den Scanner auf. Der Scanner ist demnach keine eigenständige Übersetzungsphase, sondern eine Funktion, die vom Parser regelmäßig aufgerufen wird, um nur jeweils ein einziges Token zu erkennen.

Auch die Typprüfung und die Zwischencode-Generierung lassen sich oft während der Syntaxanalyse durchführen. Noch einen Schritt weiter gehen Einphasen-Compiler, die in einem Durchgang den Zielcode erzeugen.

Zum Vergleich: Für Pascal gibt es Einphasen-Compiler, während der PL/1-Compiler noch 70 (!) Phasen hatte. Gebräuchliche C-Compiler haben bis zu 3 oder 4 Phasen.

1.2.2 Arbeitsteilung bei der Compilerentwicklung

Wenn man zunächst die interne Programmrepräsentation definiert, können Analyse- und Synthesephase parallel entwickelt werden.

Das gleiche Prinzip ist auch bei der Synthese über Zwischencode möglich: Während die Zwischencode-Generierung von einem Team entwickelt wird, kann ein zweites schon den Übersetzer vom Zwischen- in den Zielcode implementieren.

1.3 Bedeutung formaler Sprachbeschreibung

Ausgangspunkt sowohl für die Verwendung wie auch die Implementierung einer Programmiersprache ist eine Beschreibung der Syntax und der Semantik dieser Sprache.

Informale Beschreibungen, etwa Texte mit eingestreuten Diagrammen und Beispielen sind i.d.R. leicht verständlich und insbesondere zum Erlernen einer neuen Sprache gut geeignet.

Nachteile informaler Beschreibungen:

- Sie sind an vielen Stellen ungenau, lassen mehrere Interpretationsmöglichkeiten zu
- Sie sind lückenhaft

Formale Beschreibungen erfordern für den Leser zunächst die Vertrautheit mit dem Beschreibungsformalismus, der ggf. nur schwer verstehbar ist. Formale Sprachbeschreibungen sind für reale Programmiersprachen oft groß und kompliziert. Insbesondere werden für verschiedene Aspekte der Sprache unterschiedliche formale Modelle verwendet, z.B.

- reguläre Ausdrücke für lexikalische Ebene der Syntax
- kontextfreie Grammatiken für hierarchischen Programmaufbau
- Baumtransformations-Grammatiken zur Manipulation der internen Repräsentation
- Hardware-Beschreibungssprachen zur Spezifikation der Zielmaschine

Formale Beschreibungen sind zunächst einmal exakt, vollständig und widerspruchsfrei. (Die Exaktheit ist immanent, Vollständigkeit und Widerspruchsfreiheit erreicht man mit der gleichen Mühe, die es kostet, ein Programm zu entwickeln, das seinen Spezifikationen entspricht.)

Dann gibt es den einen entscheidenden Vorteil formaler Beschreibungen:

Formale Beschreibungen sind automatisierter Bearbeitung zugänglich.

Dies ist im Zusammenhang mit Programmiersprachen von essentieller Bedeutung. Tatsächlich lassen sich heutzutage **alle** Phasen eines Compilers aus den formalen Beschreibungen der zu übersetzenden Sprache und der Zielmaschine vollautomatisch generieren. Aus der Quellsprachbeschreibung alleine lassen sich z.B. Strukturediteren, Interpretierer und Debugger generieren.

Zumindest für die Analysephase sind geeignete Generatoren (z.B. **lex** und **yacc**) weit verbreitet und werden regelmäßig eingesetzt.

2 Einführung in Formale Sprachen

2.1 Mathematische Grundlagen

Definition

Sei A ein *Alphabet*, d.h. eine endliche Menge von Symbolen.

A^+ ist die *Menge aller Wörter über A*, die aus endlicher Aneinanderreihung (Konkatenation) von Symbolen aus A gebildet werden können.

ε bezeichne das *leere Wort*.

$A^* = A \cup \{\varepsilon\}$

Beispiel

Sei $A = \{a, B, \$\}^*$

aaaB$BB ist ein Wort über A, d.h. ein Element von A^*

Für u, v $\in A^*$ ist auch die Konkatenation uv $\in A^*$, z.B. u=$$, v=aBBa, uv=$$aBBa. ε ist das neutrale Element der Konkatenation:

εw = wε = w ,

für alle w $\in A^*$

Anmerkung

Eine Algebra mit einer Menge M und einer assoziativen Verknüpfung $\circ$ auf M heißt Halbgruppe. Falls darüber hinaus noch ein neutrales Element bezüglich $\circ$ existiert, nennt man die Algebra ein Monoid.

Betrachten Sie die Menge aller (nichtleeren) Wörter über einem Alphabet A: A^+ bzw. A^* .

Beides sind Halbgruppen, denn die Konkatenation ist eine assoziative Verknüpfung,

z.B. (ab)c = a(bc) = abc.

A^* enthält mit dem leeren Wort ε darüberhinaus noch ein neutrales Element, A^* ist also ein Monoid. (A^* ist übrigens keine Gruppe, denn bezügl. der Konkatenation existieren keine inversen Elemente).

A^+ ist die freie von A frei erzeugte Halbgruppe, A^* das freie Monoid über A.

2.2 Kontextfreie Grammatiken

Grammatiken dienen zur endlichen Beschreibung formaler Sprachen, etwa von Programmiersprachen. Für den Übersetzerbau sind insbesondere kontextfreie Grammatiken wichtig.

Definition

Eine *kontextfreie Grammatik* besteht aus

- einer endlichen Menge T von *Terminalsymbolen* (oder Tokens)
- einer endlichen Menge N von *Nonterminalsymbolen* (oder Variablen), $N \cap T = \{\}$
- einer endlichen Menge P von Ableitungsregeln (oder Produktionen) der Form

  $\alpha \to w$, wobei $a \in N, w \in (N \cup T)^*$

  a heißt linke Seite, w rechte Seite der Regel.
- einem Startsymbol S ∈ N.

Beispiel

G = (T, N, P, A) mit T = { a,b,c } , N = { A,B } , P = $\{A \to aAb, A \to B, B \to c\}$

und Startsymbol A ist eine kontextfreie Grammatik.

Anmerkung

Kontextfreie Grammatiken sind insofern gegenüber allgemeineren Grammatiken eingeschränkt, als die linke Regelseite immer aus genau einem Nonterminal besteht. Produktionen, wie Ba→aB oder abcd→ ε sind nicht erlaubt.

Beispiel für eine „richtige“ Grammatik:

(in Anlehnung an die Pascal-Programmstruktur)

Programm	$\rightarrow$	Programmkopf Deklarationen Anweisung .
Programmkopf	$\rightarrow$	"`program`" Id OptProgParListe ";"
OptProgParListe	$\rightarrow$	"(" IdListe ")"
OptProgParListe	$\rightarrow$	ϵ
IdListe	$\rightarrow$	Id
IdListe	$\rightarrow$	Id "," IdListe
Deklarationen	$\rightarrow$	OptConstDekl OptTypDekl OptVarDekl OptUpDekl
OptConstDekl	$\rightarrow$	ϵ
OptConstDekl	$\rightarrow$	"`const`" ConstDeklListe
	...	
VarDekl	$\rightarrow$	IdListe ":" Typ
	...	

2.2.1 Ableitbarkeit

Zur Beschreibung der Syntax formaler Sprachen gibt man in der Regel eine Grammatik an und definiert die Sprache dann als Menge aller Wörter, die sich aus der Grammatik erzeugen (oder ableiten) lassen. Was heißt das aber genau ?

Definition

Sei G = (T,N,P,S) eine Grammatik, u, x, y, w $\in$ (T $\cup$ N)*.

Aus uxw ist uyw **direkt ableitbar** (Notation: $uxw \rightarrow_G uyw$), wenn $(x \rightarrow y) \in$ P.

Aus uxw ist uyw **ableitbar** (Notation: $uxw \stackrel{*}{\rightarrow}_G uyw$), wenn für ein $n > 0$ Wörter $v_0, \ldots, v_n$ existieren, so daß

$uxw = uv_0w \rightarrow_G uv_1w \ldots \rightarrow_G uv_nw = uyw$

(Die Ableitbarkeitsrelation ist demzufolge der transitive und reflexive Abschluß der direkten Ableitbarkeit.)

Definition

Die von G erzeugte Sprache L(G) ist die Menge aller aus dem Startsymbol ableitbaren Wörter, die nur aus Terminalsymbolen bestehen:

$L(G) = \{w \in T^* \mid S \stackrel{*}{\rightarrow}_G w\}$

Anmerkung

Man beachte, daß mit Grammatiken dem Informatiker also eine endliche Beschreibungsmöglichkeit für unendliche Mengen - insbesondere die Menge aller Programme einer bestimmten Programmiersprache - zur Verfügung

steht.

Anmerkung

Es gibt Sprachen, die sich nicht durch kontextfreie Grammatiken, wohl aber mittels allgemeinerer Grammatiken definieren lassen,

z.B. $\{a^n b^n c^n \mid n \in N\}$

Definition

Sei G = (N, T, P, S) eine Grammatik. Ein Wort $w \in (N \cup T)^*$, das sich aus S ableiten läßt, $S \overset{*}{\rightarrow}_G w$, heißt **Satzform** (zu G).

Falls in einer Satzform mehrere Nonterminal-Zeichen vorkommen, ist die Reihenfolge der Ableitungsschritte nicht mehr eindeutig festgelegt. Wir definieren zwei kanonische Reihenfolgen für die Ableitungschritte:

Definition

Eine Ableitung, bei der in jedem Ersetzungsschritt das äußerst linke (rechte) Nonterminal-Zeichen ersetzt wird, heißt **Linksableitung** (bzw. **Rechtsableitung**).

Eine Darstellung, die von der Reihenfolge der Ableitungsschritte abstrahiert ist der Ableitungsbaum (auch Syntaxbaum oder Zerlegungsbaum).

Definition

Ein **Ableitungsbaum** ist ein geordneter Baum, der aus einer Ableitung A eines Worts w bezüglich einer Grammatik G = (N, T, P, S) wie folgt konstruiert wird:

- Der Wurzelknoten wird mit dem Startsymbol S markiert
- Zu jeder in der Ableitung angewandten Regel
  $X \overset{*}{\rightarrow}_G x_1 ... x_n$ werden dem mit X markierten Knoten im Baum, der das abgeleitete Nonterminal repräsentiert, n neue, mit $x_1, ..., x_n$ markierte Knoten als Nachfolgeknoten zugeordnet.

Der Baum hat folgende Eigenschaften:

- Die inneren Knoten des Baums sind mit Nonterminalsymbolen aus N markiert.

- Die Blätter sind mit Terminalsymbolen aus T markiert.
- Die Blätter ergeben von links nach rechts angeordnet das abgeleitete Wort w.

Beispiel

Wir betrachten die Grammatik

G = (T, N, P, A) mit T = { a,b,c } , N = { A,B,C }.

Die Regelmenge P enthalte folgende Regeln:

1. A→ aBC
2. B→ Bb
3. B→ ϵ
4. C→ c

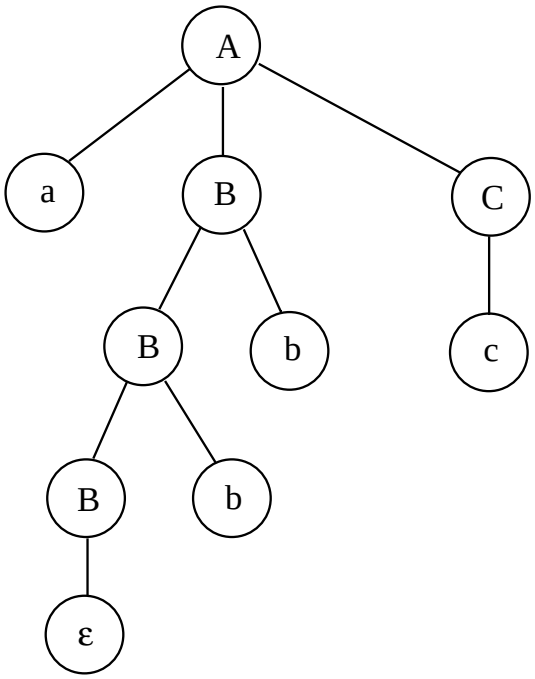
Das Wort w=abbc liegt in L(G). Die Linksableitung für w ist

$A \xrightarrow{1} aBC \xrightarrow{2} aBbC \xrightarrow{2} aBbbC \xrightarrow{3} abbC \xrightarrow{4} abbc$

Die Rechtsableitung:

$A \xrightarrow{1} aBC \xrightarrow{4} aBc \xrightarrow{2} aBbc \xrightarrow{2} aBbbc \xrightarrow{3} abbc$

Der (eindeutig bestimmte) Ableitungsbaum:



2.2.2 Notationsvarianten für Grammatiken

Für Grammatik-Regeln gibt es mehrere gebräuchliche Schreibweisen, die insbesondere der Tatsache Rechnung tragen, daß man Grammatiken mit dem Rechner verarbeitet, auf gebräuchlichen Tastaturen aber weder „$\rightarrow$“ noch „ϵ“-Tasten zu finden sind.

- Um linke und rechte Seite einer Regel zu trennen wird statt $\rightarrow$ auch "`::=`" oder (vgl. *yacc/bison*) einfach nur "`:`"verwendet.
- Oft gibt es für ein Nonterminalsymbol X mehrere syntaktische Varianten $V_1 \ldots V_k$. Statt für jede Variante eine eigene Regel zu formulieren

  $$\begin{array}{lcl} \mathrm{X} & \rightarrow & V_1 \\ & \ldots & \\ \mathrm{X} & \rightarrow & V_k \end{array}$$

  schreibt man auch

  $\mathrm{X} \rightarrow V_1 \mid \ldots \mid V_k$
- Zur Abgrenzung zwischen Terminal- und Nonterminalsymbolen werden einzelne Terminalsymbole oder Terminalsymbolfolgen auf der rechten Seite einer Grammatik in Apostrophe eingeschlossen.

Beispiel:

Die oben definierte Grammatik

G = (T, N, P, A) mit T = { a,b,c } , N = { A,B,C } mit den Regeln

A$\rightarrow$ aBC
B$\rightarrow$ Bb
B$\rightarrow$ ϵ
C$\rightarrow$ c

sieht in yacc-Notation so aus:

```
A : "a" B C ;
B : B "b" | /* epsilon */ ;
C : "c" ;
```

„`/* epsilon */`“ ist dabei nur ein Kommentar (C-Syntax), der für den Leser klar macht, daß B bei dieser Regelalternative ganz entfällt.

2.3 EBNF

Kontextfreie Grammatiken werden auch als BNF-Notation (Backus-Naur-Form) bezeichnet.

EBNF steht für „erweiterte Backus-Naur-Form“. EBNF bietet gegenüber der einfacheren BNF oder der „klassischen“ Notation für kontextfreie Grammatiken einige Schreibabkürzungen. Insbesondere gibt es einen Wiederholungsoperator ($\{\ldots\}$), der die umständlichere rekursive Definition von Listen unnötig macht.

Eine EBNF-Beschreibung besteht aus Regeln der Form

Symbol ::= *rechte Seite*

Auf der rechten Seite können Terminalzeichenketten stehen, die in Apostrophe eingeschlossen sind, z.B. `"hallo"` oder `">="`, (Nonterminal-)Symbole oder komplexe Ausdrücke.

Für die Bildung komplexer Ausdrücke verwendet EBNF diverse Meta-Zeichen, z.B. eckige, geschweifte und runde Klammern. Falls diese Zeichen auch in der zu beschreibenden Sprache vorkommen, sind sie dort in Apostrophe einzuschließen, um Verwechslungen zu vermeiden.

- Für die Auslassung eines Elements verwenden wir **EMPTY**.
- die rechte Seite kann mehrere durch den Alternativ-Operator `|` getrennte syntaktische Varianten enthalten, z.B.

```
declaration ::= label_decl | const_decl | type_decl |
                var_decl | function_decl | procedure_decl
```

  Die Präzedenz diese Operators ist schwächer als die der Konkatenation, ggf. ist zu klammern :

```
name ::=  "HANS"  | "PETER" "MAIER"
```

  ist äquivalent zu

```
name ::=  "HANS"  | ( "PETER" "MAIER" )
```

- auf der rechten Seite kann eine beliebige (auch eine 0-fache) Wiederholung eines Elements durch Einschließen dieses Elements in geschweifte Klammern spezifiziert werden

```
function_call  ::= identifier  ( parameter_list | EMPTY )
parameter_list ::= "(" expression  { "," expression } ")"
```

- für eine Beschränkung der Wiederholungen nach unten durch *min* oder nach oben durch *max* verwendet man

  $$\{ \quad \ldots \quad \}_{min}^{max}$$

- statt $\{ \quad \ldots \quad \}_0^1$ schließt man optionale Elemente in eckige Klammern ein.

2.4 Transformation von EBNF in Grammatiken

Die Transformation einer EBNF-Syntax in eine kontextfreie Grammatik läßt sich sowohl von Hand als auch per Programm ohne große Probleme bewerkstelligen.

Im wesentlichen wird man dabei folgende Ersetzungen benötigen:

- Einen Listenbestandteil einer rechten Regelseite der Form $\{Listenelement\}$ ersetzt man durch ein neues Nonterminalsymbol, hier z.B. *Liste*, und definiert rekursiv:

  $Liste \rightarrow ListenelementListe \mid \epsilon$ oder

  $Liste \rightarrow ListeListenelement \mid \epsilon$

- Für eine optionale syntaktische Struktur der Form $[X]$ wird gleichfalls ein neues Nonterminalsymbol definiert, hier z.B. $OptionalesX$. Die Regeln lauten dann

  $OptionalesX \rightarrow X \mid \epsilon$

- Auch für geklammerte komplexe Bestandteile der rechten Regelseite wird ein neues Nonterminalsymbol definiert, dessen Definition dann abhängig ist von den in den Klammern stehenden Operatoren.

Das EBNF-Beispiel von oben

```
function_call  ::= identifier  ( parameter_list | EMPTY )
parameter_list ::= "(" expression  { "," expression } ")"
```

kann in folgende Grammatik transformiert werden:

function_call	$\rightarrow$	*identifier OptParameterList*
OptParameterList	$\rightarrow$	*parameter_list* $\mid \epsilon$
parameter_list	$\rightarrow$	*"(" expression Expressions ")"*
Expressions	$\rightarrow$	*"," expression Expressions* $\mid \epsilon$

Listing 2.1 EBNF-Beispiel: Syntax von Oberon-0

```
ident = letter {letter|digit}.
integer = digit {digit}.

selector = { "." ident |"[" expression "]"}.
number = integer.
factor = ident selector | number
         | "(" expression ")" | "~" factor.
term = factor {("*" | "DIV" | "MOD" | "&") factor}.
SimpleExpression = ["+"|"-"] term {("+"|"-"|"OR") term}.
expression = SimpleExpression
             ("=" | "#" | "<" | "<=" | ">" | ">=")
             SimpleExpression.
assignment = ident Selector ":=" expression.
ActualParameters = "(" [expression {"," expression }]")".
ProcedureCall = ident [ActualParameters].
IfStatement = "IF" expression "THEN" StatementSequence
             {"ELSIF" expression "THEN" StatementSequence}
             [ELSE StatementSequence] "END".
WhileStatement = "WHILE" expression "DO" StatementSequence "END".
statement = [ assignment | ProcedureCall
            | IfStatement | WhileStatement ].
StatementSequence = statement {";" statement }.

IdentList = ident {"," ident}.
ArrayType = "ARRAY" expression "OF" type.
FieldList = [IdentList ":" type].
RecordType = "RECORD" FieldList {";" FieldList} "END".
type = ident | ArrayType | RecordType.
FPSection = ["VAR"] IdentList ":" type.
FormalParameters = "(" [FPSection {";" FPSection}] ")".
ProcedureHeading = "PROCEDURE" ident [FormalParameters].
ProcedureBody = declarations
                ["BEGIN" StatementSequence]
                "END" ident.
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody.
declarations = ["CONST" {ident "=" expression ";"}]
               ["TYPE"  {ident "=" type ";"}]
               ["VAR" {IdentList ":" type ";"}]
               {ProcedureDeclaration ";"}.
module = "MODULE" ident ";" declarations
         [ "BEGIN" StatementSequence ] "END" ident ":".
```

2.5 Formale Sprachen und Compilerbau

Die Theorie der Formalen Sprachen, die Automaten- und die Komplexitätstheorie haben entscheidend die Compilerbau-Theorie geprägt.

Innerhalb der Theorie der Formalen Sprachen klassifiziert die Chomsky-Hierarchie verschiedene Komplexitätsstufen für Grammatiken und die dadurch definierten Sprachen:

- Bei Chomsky-0-Grammatiken gibt es keine Restriktionen für die Ableitungsregeln. Ob ein Wort w mit den Regeln der Grammatik aus dem Startsymbol ableitbar ist, ist **nicht entscheidbar**.
- Chomsky-1-Grammatiken nennt man auch **kontextsensitive** Grammatiken. Für diese Grammatiken gibt es bestimmte Restriktionen für die Ableitungsregeln und die theoretische Aussage, daß die Zugehörigkeit eines Wort zu einer Chomsky-1-Sprache entschieden werden kann.

  Für den Compilerbau sind Chomsky-1-Grammatiken zu komplex.
- Chomsky-2-Grammatiken sind die **kontexfreien** Grammatiken.

  Die kontexfreien Sprachen eignen sich zur Spezifikation der hierarchischen Programmstruktur für alle gängigen Programmiersprachen. Zur Syntaxanalyse können Automaten mit einem Stack (Kellermaschinen) verwendet werden.
- Chomsky-3-Grammatiken nennt man auch **reguläre** Grammatiken. Hier sind nur links- oder rechtsrekursive Regeln ($X \to Xw$ bzw. $X \to wX$) erlaubt, aber keine allgemeineren Rekursionsformen.

  Die regulären Sprachen eignen sich zur Spezifikation der lexikalischen Ebene der Sprachsyntax. Zur Erkennung können Endliche Automaten verwendet werden, ein Stack ist im Gegensatz zu den Chomsky-2-Grammatiken nicht notwendig

Eine wesentliche Rolle spielen im Compilerbau die kontextfreien Sprachen, deren Syntax durch kontextfreie Grammatiken beschrieben werden kann. Dies mag bei näherem Hinsehen verwundern, denn Kontextabhängigkeiten gibt es in den gebräuchlichen Programmiersprachen genügend viele. Ein Beispiel:

In C gilt das Prinzip „declare before use“: Jeder *Bezeichner* muss vor seiner ersten Verwendung deklariert werden (vgl. 3.1, S. 22).

Diese Regel ist syntaktischer Natur, geht aber aus keiner Grammatik für die Sprache hervor. Warum nicht ?

Der Spezifikationsmechanismus der kontextfreien Grammatiken ist dafür nicht mächtig genug. Nur mit komplizierten kontextabhängigen Regeln ließe sich das „declare before use“-Prinzip spezifizieren. Für kontextabhängige

Grammatiken gibt es aber keine praktikablen Analyseverfahren (zu kompliziert, zu langsam). Der einzige ernsthafte Versuch, kontextabhängige Grammatiken für eine reale Programmiersprache zu verwenden, die ALGOL68-Definition, war nicht gerade erfolgreich.

Daher beschreibt man die Syntax einer Programmiersprache wie folgt: In einer kontextfreien Grammatik beschreibt man den Großteil der syntaktischen Anforderungen. Alles, was sich nicht mit kontextfreien Regeln beschreiben läßt, wird außerhalb der Grammatik, quasi als „Nebenabrede“ (meist informal) ergänzt. Manchmal werden auch Dinge, die sich an für sich kontextfrei beschreiben liessen, aus der Grammatik herausgenommen, um diese nicht zu komplex zu machen.

3 Lexikalische Analyse (Scanner)

Die Lexikalische Analyse ist ein Teil der Syntaxanalyse. Ziel ist die Erkennung von lexikalischen Quelltextbestandteilen wie Literalen, Schlüsselwörtern, Bezeichnern, Operatoren, Kommentaren.

Wir nennen diese syntaktischen „Grundbausteine“ eines Programms nachfolgend **Tokens**. Ein Tokenerkennungsprogramm heißt auch **Scanner**.

Tokens haben eine besonders einfache Syntax, die mit **regulären Ausdrücken** beschrieben wird. Diese sind kompakter als Grammatiken.

Der Grund für die Abgrenzung zwischen Lexikalischer Analyse und Syntaxanalyse liegt darin, dass für die Tokens einfachere und effizientere Analyseverfahren eingesetzt werden.

3.1 Lexikalische Elemente einer Programmiersprache

In höheren Programmiersprachen muß die Lexikalische Analyse typischerweise folgende Tokens erkennen:

- **Bezeichner** (engl.: Identifier)

  Ein Bezeichner ist ein üblicherweise vom Programmierer definierter symbolischer Name. Er wird nach festen Regeln gebildet, z.B. Buchstabe oder Unterstrich gefolgt von beliebigen Buchstaben, Ziffern oder Unterstrichen.

  Der Bezeichner kann im Programmtext sehr unterschiedliche Dinge repräsentieren, z.B.

  - Variablen
  - Konstanten
  - Typen
  - Funktionen
  - Sprungziele
  - Klassen

  In vielen Programmiersprachen muss ein Bezeichner vor der ersten Verwendung **deklariert** werden.

- **Operator**

  Ein Operator repräsentiert in kompakter Schreibweise eine häufig benötigte „Operation“. Syntaktisch besteht er meist nur aus einem oder zwei Sonderzeichen, z.B. `+, +=, ->, ., :=`.

  In vielen Sprachen stehen Operatoren immer für vordefinierte Funktionen, in anderen (z.B. C++) können Sie auch Programmierer-definierte Funktionen repräsentieren.

  - **Präfix-Operatoren** werden ihren Operanden vorangestellt, z.B. `- x`
  - **Postfix-Operatoren** werden hinter ihre Operanden gestellt, z.B. `p ++`
  - **Infix-Operatoren** stehen zwischen ihren Operanden, z.B. `a += b`
  - **Mixfix-Operatoren** nennt man andere syntaktische Formen, z.B. in C der Index-Operator `[]`, der eine Feldkomponente selektiert, oder der `? ... :`-Operator

- **Literal**

  Ein Literal ist eine vordefinierte spezielle Schreibweise zur Repräsentation eines bestimmten Wertes, z.B. `-123e5, 0Xff, 'x', "hallo\n"`

- **Kommentar**

  Ein Kommentar mag aus der Sicht seines Verfassers oder seines Lesers ein langer, komplexer, strukturierter (und wichtiger) Programmbestandteil sein.

  Vom Compiler wird er allerdings i.d.R. nicht weiter analysiert (sondern meist schlichtweg ignoriert) und daher als Token behandelt.

- **Schlüsselwort**

  Schlüsselwörter sind von der Sprache fest vorgegebene Wörter, die dem Compiler die Erkennung bestimmter Strukturen ermöglichen (z.B. **if**, **repeat**, **typedef**).

  Es gibt Sprachen mit extrem vielen Schlüsselwörtern (z.B. COBOL) und Sprachen völlig ohne Schlüsselwörter (z.B. LISP).

  Schlüsselwörter sind in der Regel **reserviert**, es ist also verboten einen syntaktisch identischen Bezeichner zu verwenden.

  (In PL/I sind sie nicht reserviert, was einer einfachen Syntaxanalyse nicht gerade dienlich ist, vor allem aber nicht der Lesbarkeit von Programmen:)

  ```
  IF IF=THEN THEN THEN:=IF ELSE ELSE:=THEN ...
  ```

3.2 Spezifikation der Token-Syntax

Zur Spezifikation der Token-Syntax ist EBNF geeignet. Es gilt (ohne Beweis): **Eine Sprache ist genau dann regulär, wenn man sie durch eine einzige nichtrekursive EBNF-Regel beschreiben kann.**

Compilerbau-Werkzeuge und andere UNIX-Tools verarbeiten statt EBNF-Regeln allerdings **reguläre Ausdrücke**. Diese Ausdrücke sind im Prinzip zu den nichtrekursiven EBNF-Regeln äquivalent. In der UNIX-Notation gibt es allerdings zusätzliche Operatoren, die die Syntaxbeschreibung noch weiter vereinfachen.

Nachfolgend werden reguläre Ausdrücke und reguläre Sprachen formal definiert.

3.2.1 Reguläre Sprachen

Operationen für Sprachen

Seien L, L_1 und L_2 Sprachen. Wir definieren vier Operationen für Sprachen:

1. Vereinigung
   $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ oder } w \in L_2\}$
2. Konkatenation
   $L_1 L_2 = \{w \mid w = uv, u \in L_1, v \in L_2\}$
3. Kleenesche Hülle
   $L^* = \bigcup_{i=0}^{\infty} L^i$
4. positive Hülle
   $L^+ = \bigcup_{i=1}^{\infty} L^i$

Sprachen, die aus endlichen Wortmengen nur mittels dieser Operationen konstruiert werden, nennt man reguläre Sprachen. Sie sind besonders einfach strukturiert und eignen sich für die Spezifikation der lexikalischen Ebene einer Programmiersprache.

3.2.2 Reguläre Ausdrücke

Mit den oben definierten Sprach-Operationen können reguläre Ausdrücke als formale Beschreibungsmethode für die lexikalische Ebene der Syntax von Programmiersprachen (und andere Zwecke) definiert werden.

Definition

Sei A ein Alphabet. Die Menge der regulären Ausdrücke über A ist induktiv definiert. Jeder reguläre Ausdruck r repräsentiert eine Sprache $L(r) \subseteq A^*$.

1. Für jedes Symbol $a \in A$ ist a ein regulärer Ausdruck. $L(a) = \{a\}$
2. ϵ ist ein regulärer Ausdruck. $L(\epsilon) = \{\epsilon\}$
3. Seien r_1 und r_2 reguläre Ausdrücke (über A). Dann ist $r_1 r_2$ ein regulärer Ausdruck. $L(r_1 r_2) = L(r_1)L(r_2)$
4. Seien r_1 und r_2 reguläre Ausdrücke. Dann ist $r_1 \mid r_2$ ein regulärer Ausdruck. $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$
5. Seien r ein regulärer Ausdruck. Dann ist r^* ein regulärer Ausdruck. $L(r^*) = L(r)^*$

Definition

Seien r und s reguläre Ausdrücke über A. r und s heißen **äquivalent** (Notation: r=s), genau dann, wenn $L(r) = L(s)$.

Notation

Um eine eindeutige Operator-Operanden-Zuordnung zu ermöglichen, lassen sich reguläre Ausdrücke klammern. Für ungeklammerte Ausdrücke wird vereinbart:

- $*$ ist linksassoziativ und hat höchste Präzedenz
- Die Konkatenation ist linksassoziativ und hat zweithöchste Präzedenz
- $\mid$ ist linksassoziativ und hat niedrigste Präzedenz

3.2.3 Reguläre Ausdrücke – UNIX-Notation

Für den praktischen Umgang mit Programmen, die reguläre Ausdrücke verarbeiten, sind weitere Operatoren nützlich, die eine kompaktere Notation ermöglichen. (Daneben gibt es auch noch das Problem mit der fehlenden ϵ-Taste an der Rechnertastatur!)

Wir benutzen die in UNIX-Systemen übliche Schreibweise, wie sie vom Scannergenerator *lex*, aber (zumindest annähernd) auch von einer Menge anderer Programme (*grep*, *egrep*, *sed*, *emacs* usw.) benutzt wird. Man beachte,

daß in vielen Betriebssystemen auch eine POSIX-Programmierschnittstelle zur Verwendung dieser regulären Ausdrücke verfügbar ist, vgl. *regcomp*, *regexec* u.a.

Die Notation

Man unterscheidet Sonderzeichen und normale Zeichen. Sonderzeichen sind z.B. : `* + [ ] ? ^ ( ) . $`

- ein „normales“ Zeichen steht für sich selbst
- `.` steht für ein beliebiges Zeichen außer `'\n'`
- Falls ein Sonderzeichen nicht als solches interpretiert werden soll, ist ein „\“ voranzustellen
- Auch in Apostrophe eingeschlossene Strings werden verbatim interpretiert.
- Klammerung erfolgt durch (und)
- Konkatenation zweier reg. Ausdrücke erfolgt ohne expliziten Operator
- Alternativen werden mittels | gebildet
- ein nachgestellter * steht für beliebige Wiederholung
- ein nachgestelltes + steht für nichtleere Wiederholung
- ein nachgestelltes ? bezeichnet einen optionalen Anteil
- `^` am Anfang eines regulären Ausdrucks steht für Zeilenanfang
- $ am Ende eines regulären Ausdrucks steht für Zeilenende
- eine **Zeichenklasse** steht für ein Zeichen.

  Sie kann durch **Zeichen-Aufzählung** $x_1 x_2 \ldots x_n$ und **Bereichsangaben** $x_1 - x_n$ gebildet werden:

  - $x_1 - x_n$ steht für ein Zeichen aus dem Bereich, z.B. `[0-9]`
  - $x_1 x_2 \ldots x_n$ steht für ein Zeichen aus der Menge der angegebenen Zeichen, z.B. `[abcx]`
  - beide Schreibweisen können kombiniert werden z.B. `[0-9a-zA-Z_]`
  - Eine `^`-Zeichen am Anfang einer Zeichenklasse `[^...]` spezifiziert die komplementäre Zeichenmenge, z.B. steht `[^0-9]` für ein beliebiges Zeichen außer einer Ziffer

Beispiele:

a) Alle mit kleinem „a“ beginnenden Zeichenketten: `a.*`

b) Alle nichtleeren Dezimalziffernfolgen: `[0-9]+`

c) Alle Wörter, die aus genau 3 Zeichen bestehen und nicht mit einer Ziffer enden: `..[^0-9]`

d) Pascal-Bezeichner = `[A-Za-z][A-Za-z0-9]*`

e) C-Float-Literale = `-?[0-9]+((\.[0-9]+)|((\.[0-9]+)?[eE]-?[0-9]+)`

C-Float-Literale bestehen aus

- einem „Vorkomma-Anteil“ (ggf. mit Minuszeichen) (Syntax: `-?[0-9]+`)
- einem optionalen Nachkomma-Anteil (Syntax: `\.[0-9]+`)
- und einem ebenfalss optionalen Exponenten-Anteil (Syntax: `[eE]-?[0-9]+`).

Dabei ist zu beachten, daß entweder der Nachkomma-Anteil oder der Exponent vorhanden sein muß. Wenn beides fehlt, liegt eine Integer-Konstante vor. Daher ist die folgende Spezifikation nicht korrekt:

`-?[0-9]+(\.[0-9]+)?([eE]-?[0-9]+)?`

(Viel komplizierter wird es in der Praxis nur selten !)

3.3 Scanner-Implementierung

Scanner werden i.d.R. entweder

- „ad hoc“ programmiert oder
- mittels Scanner-Generator automatisch erzeugt

Systematische Scanner-Implementierung (im Gegensatz zur „ad hoc“-Vorgehensweise) kann basierend auf endlichen Akzeptoren nach dem im Abschnitt 3.4, S. 31 erklärten Schritten durchgeführt werden. Wegen der Größe der Automaten ist dies allerdings sehr aufwendig.

Alle Schritte der systematischen Scanner-Implementierung sind voll automatisierbar. Wenn man diese Schritte programmiert, hat man einen Scanner-Generator implementiert.

Betrachten wir zunächst „ad hoc“-Verfahren.

Folgende Punkte sind zu beachten:

- Typischerweise wird der Scanner als Funktion implementiert, die vom Parser aufgerufen wird, um das nächste im Quelltext stehende Token zu erkennen.

  Als Resultat muß der Scanner zumindest eine eindeutige Token-Identifikation liefern. Bei Operatoren und Schlüsselwörtern genügt diese Information dem Parser, die konkrete Zeichenfolge, aus denen die Tokens bestehen ist irrelevant.

  Hat der Scanner einen Bezeichner gefunden, genügt die Aussage, daß es sich um einen solchen handelt, natürlich nicht. Hier wird der Bezeichner selbst benötigt. Das gleiche gilt für Literale aller Art.

  Ein von *yacc* aufgerufener Scanner erwartet die Token-Klassifikation (Typ: int) als Funktionsresultat. Die Zeichenkette, aus der das Token besteht, muß in der globalen Variable *yytext* stehen und deren Länge in *yylen*.

- Der Scanner muß sehr effizient programmiert werden, da große Programme aus hunderttausenden von Tokens bestehen können und entsprechend viele Scanner-Aufrufe im Lauf einer Compilierung erfolgen. Insbesondere darf er pro eingelesenem Zeichen im Durchschnitt nicht allzuviel Bearbeitungszeit benötigen.

- Es ist unbedingt darauf zu achten, das beim Einlesen des Quelltexts von einer Plattendatei mit Pufferung gearbeitet wird und nicht pro eingelesenem Zeichen ein physikalischer Plattenzugriff erfolgt.

  Bei Verwendung der „Streams“-Funktion der Standard-C-Bibliothek (*getchar*, *fgetc*, *fread* usw.) ist Pufferung verfügbar (ggf. mit *setbuf*, *setvbuf* einstellen).

- In der Regel wird der Scanner beim Aufruf das erste Zeichen des nächsten Tokens lesen. Anhand dieses Zeichens kann er eine Fallunterscheidung treffen zwischen Operatoren, Literalen, Bezeichnern oder Schlüsselwörtern usw. In jedem der Fälle wird dann das Token vollständig gelesen.

  Man beachte, daß manche Operatoren mit dem gleichen Zeichen beginnen.

- Zur Erkennung des Token-Endes ist oft eine Vorausschau erforderlich.

  Um das Ende eines Bezeichners zu erkennen, muß der Scanner das Zeichen hinter dem Bezeichner gelesen haben, also das erste Zeichen nach dem Bezeichner-Anfang, das nach den Syntaxregeln kein Bezeichnerbestandteil mehr sein darf (z.B. Space, Newline, Operator).

  Für viele Sprachen genügt eine Vorausschau um ein Zeichen.

Am einfachsten wird die Einleseschleife dann, wenn folgende Vor- und Nachbedingung für den Scanneraufruf erfüllt werden:

Beim Scanner-Aufruf und beim Rücksprung aus dem Scanner ist das aktuelle Zeichen im Eingabestrom entweder

- das erste Zeichen des nächsten Token oder
- ein Whitespace-Zeichen vor dem nächsten Token oder
- das Eingabeende.

Um dies zu erfüllen, ist immer ein Token und das dahinter stehende Zeichen zu lesen.

Schlüsselwörter und Bezeichner

Der Scanner selbst wird sehr viel einfacher, wenn alle Schlüsselwörter von ihm vorläufig als „Bezeichner“ klassifiziert werden. In der Symboltabelle steht zu jedem (echten oder vermeintlichen) Bezeichner dann die korrekte Klassifikation.

Dazu initialisiert man vor dem Scannen einfach die Tabelle mit je einem Eintrag für jedes Schlüsselwort.

Findet der Scanner einen „Bezeichner“, wird in der Tabelle ein passender Eintrag gesucht. Falls keiner vorhanden ist, handelt es sich um einen echten Bezeichner, der mit genau dieser Klassifikation neu eingetragen wird.

Ist dagegen schon ein Eintrag vorhanden, so kann die korrekte Klassifikation der Tabelle entnommen werden, einerlei, ob es sich um ein bei der Initialisierung eingetragenes Schlüsselwort oder einen während der Analyse vorher schon einmal gefundenen echten Bezeichner handelt.

Wer sucht in der Symboltabelle nach der korrekten Klassifikation ?

Für die Aufgabe ist es egal, ob der Scanner selbst oder die aufrufende Parser-Simulation dies tut. Bei einem vollständigen System wird man dies besser dem Parser überlassen, der ja bei echten Bezeichnern ohnehin Symboltabellenzugriffe durchführen muß.

Grobskizze für „ad hoc“-Scanner:

```
int gettoken(){
  ...
  /* Leerzeichen, TABs usw. ueberlesen: */
  do
    c=lieszeichen();
  while ( IS_WHITE_SPACE (c) );

  switch (c) {
  /* Operatoren, die aus einem Zeichen bestehen,
     und am ersten Zeichen erkannt werden koennen */
  case '.':
  case ',':
     ...    token=c;
            c=lieszeichen();
            return token;
  /* sonstige Operatoren: */
  case ':': /* Test ob ":" oder ":=" */
            if((c=lieszeichen())=='=') {
              c=lieszeichen();
              return TK_WERTZUWEISUNG;
              }
            else
              return ':';
  ...
  /* Kommentare */
  case '{':  ...

  /* Zahlen */
  case 0:
   ...
  case 9:  ...

  /* Bezeichner (bzw. Schluesselwoerter) */
  default:
           if (BEZEICHNER_ZEICHEN(c)) {
              ...
              }
           else
              /* weitere Möglichkeiten ?, Fehler ? */
  }
```

Die einzelnen Fälle der switch-Anweisung können dabei direkt ausprogrammiert werden. Eine Alternative ist ein kleiner endlicher Automat, der mehrere Zustände unterscheidet, für die jeweils ein separates Verhalten programmiert wird.

3.4 Systematische Scanner-Implementierung

Grundlage für die Automatisierung der Scanner-Entwicklung ist das Konzept der Endlichen Akzeptoren, die zur Prüfung der Token-Syntax bzw. Erkennung von Tokens verwendet werden.

3.4.1 Endliche Automaten / Endliche Akzeptoren

Definition

Ein (nichtdeterministischer) endlicher Automat (NEA) besteht aus

- einer endlichen Menge S von Zuständen
- einem Eingabealphabet Σ
- einer Übergangsrelation
  $next \subseteq (S \times (\Sigma \cup \{\varepsilon\})) \times S$
- einem Startzustand $s_0 \in S$
- einer Menge von Endzuständen $F \subseteq S$

Ist $(s, x, s') \in next$, so nennen wir s' Folgezustand zu s bei Eingabe x.

Definition

Ein deterministischer endlicher Automat (DEA) ist ein endlicher Automat $(S, \Sigma, next, s_0, F)$, bei dem $next$ eine Funktion

$$next : (S \times \Sigma) \to S$$

ist.

Anmerkung

Bei einem DEA gibt es also keine Zustandsübergänge mit ε und zu jedem Zustand und jedem Eingabesymbol nur einen Folgezustand.

3.4.2 Darstellung von endlichen Automaten

Die Zustandsübergänge eines EA lassen sich durch eine Zustandsübergangstabelle oder durch ein Zustandsübergangs- diagramm repräsentieren.

Während das Diagramm der menschlichen Auffasungsgabe besser Rechnung trägt, ist die Tabellen-Darstellung für die maschinelle Verarbeitung gut geeignet.

Beispiel

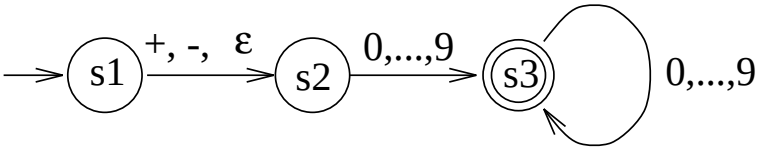
Sei S={s1, s2, s3}die Zustandsmenge, s1 der Startzustand und s3 der einzige Endzustand, d.h. F={s3}

Repräsentation durch die Übergangstabelle

Zustand	Eingabe	Folgezustand
s1	+	s2
s1	-	s2
s1	ε	s2
s2	0	s3
s2	1	s3
	⋮	
s2	9	s3
s3	0	s3
s3	1	s3
	⋮	
s3	9	s3

Repräsentation durch Zustandsübergangsdiagramm

Das Diagramm ist ein Graph, in dem die Zustände durch Knoten und die Übergange durch Kanten repräsentiert werden, die mit den Eingabesymbolen bzw. mit ε beschriftet sind.



Endzustände und Anfangszustand sind besonders markiert.

3.4.3 Automaten als Akzeptoren für formale Sprachen

Definition

Ein NEA $A = (S, \Sigma, next, s_0, F)$ akzeptiert eine Eingabe $w \in S^*$ genau dann, wenn ein Pfad $k_0, \ldots, k_n$ mit Kantenmarkierungen $x_1, \ldots, x_n$ im Übergangsdiagramm von A existiert, so daß

$k_0 = s_0$
$k_n \in F$ und
$x_1 \ldots x_n = w$

Die Menge aller von A akzeptierten Wörter ist die von A akzeptierte Sprache.

Für den oben angegebenen Automaten A ist die akzeptierte Sprache durch folgenden regulären Ausdruck gegeben: `[+|-]?[0-9]+`

Satz

Zu jeden regulären Ausdruck r gibt es effektiv einen endlichen Akzeptor A, so daß $\mathrm{L}(r) = \mathrm{L}(A)$

Es gibt diverse Algorithmen zur Konstruktion passender (nichtdeterministischer oder deterministischer) Akzeptoren, wir stellen in 3.4.4 den Thompson-Algorithmus vor.

Systematische Vorgehensweise:

Sei T die Menge der lexikalischen Atome der Sprache (Tokens).

Sprach-abhängig:

1. Beschreibe für alle Tokens $t \in T$ die Syntax durch je einen regulären Ausdruck r_t
2. Berechne zu jedem Token t aus r_t einen nichtdeterministischen endlichen Akzeptor A_t (Thompson-Algorithmus). Die Zustandsmengen aller Akzeptoren seien disjunkt.
3. Berechne den NEA, der sich durch Parallelschaltung aller nach Schritt 2 berechneten Akzeptoren A_t ergibt. Dabei wird die Zuordnung eines Tokens t zu jedem Endzustand von A_t festgehalten.
4. Berechne den äquivalenten DEA (Die Teilmengen, die Endzustände repräsentieren, müssen einelementig sein, sonst ist die Token-Syntax mehrdeutig)
5. Minimiere den nach 4 berechneten DEA, behalte aber die Unterscheidung solcher Endzustände bei, die an unterschiedliche Tokens gebundenen sind

6. Repräsentiere den minimalen DEA mittels komprimierter Zustands-Übergangs-Matrix

Sprach-unabhängig:
Implementiere einen Tabellen-gesteuerten DEA-Simulator

Pseudocode-Schema für DEA-Simulation (ohne vernünftige Fehlerbehandlung):

```
zustand := startzustand ;
token := scanner();
while (nexttoken <> EOF) do
  zustand := folgezustand [ zustand, token ];
  if (zustand = fehlerzustand) then
    break;
  nexttoken := scanner();
done

if (zustand = fehlerzustand) or
   (zustand nicht Endzustand) then
  Fehlerbehandlung
else
  return tokenklasse[zustand];
fi
```

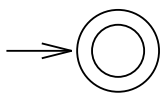
3.4.4 Vom regulären Ausdruck zum nichtdeterministischen Akzeptor – Thompson-Algorithmus

Eingabe: ein regulärer Ausdruck r über dem Alphabet Σ

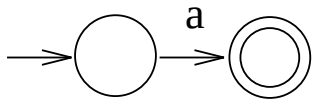
Ausgabe: NEA A für $\mathrm{L}(r)$

Fallunterscheidung:

1. $r = \varepsilon$



2. $r = a \in \Sigma$



3. $r = r_1 \mid r_2$

   Parallelschaltung:

   (a) Konstruiere NEA A_1 zu r_1, A_2 zu r_2

   (b) Schalte A_1 und A_2 zu A wie folgt parallel:

   - alle Zustände und Übergänge aus A_1 und A_2 werden übernommen
   - ein neuer Startzustand s für A wird eingeführt
   - von s aus werden ε-Übergänge zu den beiden Zuständen hinzugefügt, die in A_1 und A_2 Startzustände sind.
   - die Endzustände von A_1 und A_2 bleiben Endzustände

4. $r = r_1 r_2$

   Serienschaltung:

   (a) Konstruiere NEA A_1 zu r_1, A_2 zu r_2

   (b) Schalte A_1 und A_2 zu A wie folgt hintereinander:

   - alle Zustände und Übergänge aus A_1 und A_2 werden übernommen
   - Startzustand von A ist der Startzustand von A_1.
   - Alle Endzustände von A_2 sind Endzustände von A
   - ein Endzustand von A_1 ist nicht länger Endzustand
   - zu jedem Endzustand von A_1 gibt es einen ε-Übergang in den Startzustand von A_2

5. $r = r_1^*$

   Rückführung:

   (a) Konstruiere NEA A_1 zu r_1

   (b) Konstruiere A aus A_1 wie folgt:

   - alle Zustände und Übergänge aus A_1 werden übernommen
   - Startzustand von A ist der Startzustand von A_1.
   - Alle Endzustände von A_1 sind Endzustände von A
   - Der Startzustand ist auch Endzustand
   - zu jedem Endzustand von A_1 gibt es einen ε-Übergang in den Startzustand von A

Anmerkung

- Zur Implementierung benötigt man natürlich einen Parser für reguläre Ausdrücke
- Die Implementierung wird einfacher, wenn man nur einen Endzustand zuläßt (ggf. neuen Endzustand und ε-Übergänge von den ursprünglichen Endzuständen in den neuen einführen)
- Man kann aus dem NEA durch Teilmengen-Konstruktion einen DEA ableiten
- Es gibt Verfahren, die direkt einen DEA konstruieren

3.4.5 Vom nichtdeterministischen zum deterministischen Akzeptor

Satz

Zu jedem (nichtdeterministischen) endlichen Akzeptor A existiert ein äquivalenter deterministischer endlicher Akzeptor A' (d.h. A und A' akzeptieren die gleiche Sprache).

(ohne Beweis)

Konstruktion deterministischer endlicher Automaten (Teilmengenkonstruktion)

Eingabe: Ein NEA $A = (S, \Sigma, next, s_0, F)$

Ausgabe: Ein DEA $A' = (S', \Sigma, next', s_0, F')$ mit $\mathrm{L}(A) = \mathrm{L}(A')$.

Idee: Der Zustand, den A' mit der Eingabe w erreicht, repräsentiert die Menge aller Zustände, die A mit der Eingabe w erreichen kann. Jeder Zustand T des DEA A' ist also eine Teilmenge von S.

Basis-Operationen:

zu jedem Zustand $s \in S$, zu jeder Zustandsmenge $T \in 2^S$ und jedem $a \in \Sigma$ sei definiert:

- ε-Abschluß(s)=$\{s' \in S \mid s' = s$ oder s' ist von s aus durch einen oder mehrere ε-Übergänge zu erreichen $\}$
- ε-Abschluß(T)=$\bigcup_{s \in T}$ ε-Abschluß(s)

- $move(T, a) = \{s' \in S \mid \exists_{s \in T} : (s, a, s') \in next\}$

Algorithmus:

Die Zustandsmenge S und die Übergangsfunktion *next* werden sukzessive erweitert, alle in S übernommenen Zustände seien anfangs unmarkiert.

```
S := ε-Abschluß    while (S enthält unmarkierten Zustand T ) do
  markiere T
  for a ∈ Σ do
    U:=ε-Abschluß(move(T,a));
    if not U ∈ S then
      S := S ∪ {U}
    end if;
    next(T,a) := U;
  end for;
end while;
```

3.4.6 Minimierung endlicher Automaten

Satz

Zu jedem (nichtdeterministischen) endlichen Akzeptor existiert ein äquivalenter deterministischer endlicher Akzeptor mit minimaler Zustandsmenge. Bis auf die Zustandsbenennung ist der minimale Akzeptor eindeutig bestimmt.

(ohne Beweis)

Konstruktion minimaler endlicher Automaten

Eingabe: Ein DEA $A = (S, \Sigma, next, s_0, F)$
(o.B.d.A sei *next* auf $(S \times \Sigma)$ totale Funktion – gegebenenfalls „tote“ Zustände einführen)

Ausgabe: Der minimale DEA $A' = (\mathcal{S}, \Sigma, next, s_0, F)$ mit $\mathrm{L}(A') = \mathrm{L}(A)$.

Idee: Ein Wort $w \in S^*$ unterscheidet zwei Zustände, falls A mit Eingabe w aus genau einem dieser Zustände einen Endzustand erreicht.

Zwei Zustände sind äquivalent, falls kein $w \in S^*$ diese unterscheidet. Der Algorithmus prüft im i-ten Schritt für alle Zustände und für alle Eingabewörter der Länge i, ob zwei bislang nicht unterscheidbare Zustände durch das i-te Zeichen der Eingabe unterschieden werden.

Der Zustand, den A mit der Eingabe w erreicht, repräsentiert die Äquivalenzklasse der nicht unterscheidbaren Zustände, die A mit der Eingabe w erreicht.

Algorithmus:

Die Zustandsmenge von A' wird mit $\mathcal{S}$ bezeichnet. Jeder Zustand $z \in \mathcal{S}$ repräsentiert eine Klasse äquivalenter Zustände aus der Zustandsmenge S des Ausgangsautomaten A.

Die Äquivalenzklasseneinteilung wird sukzessive durch Verfeinerung berechnet:

1. (Initialisierung der Äquivalenzklassen) $\mathcal{S} := \{F, S \setminus F\}$

   (Für w=ε sind alle Endzustände und alle Nicht-Endzustände unterscheidbar)

2. (Verfeinerung der Äquivalenzklassen)

   Berechne $\mathcal{S}_{NEU}$ aus $\mathcal{S}$ (siehe unten)

3. (Solange neue Klasseneinteilung feiner, wiederholte Verfeinerung)

   if $(\mathcal{S}_{NEU} \neq \mathcal{S})$ **then**
   $(\mathcal{S} := \mathcal{S}_{NEU})$
   goto 2;
   end if

4. (Zustandstransformation eines Repräsentanten übernehmen)

   Wähle aus jeder Äquivalenzklasse einen Repräsentanten. Die Zustandsübergänge sind durch die Übergange der Repräsentanten definiert:

   Sei $next(s1, x) = s2$, *s1* und *s2* Repräsentanten der Äquivalenzklassen von *s1* bzw. *s2*, so sei definiert:

   $next(s1, x) = s2$

5. Entferne tote und nicht erreichbare Zustände

Verfeinerung der Äquivalenzklassen:

für jeden Zustand $z \in \mathcal{S}$: (z repräsentiert eine Klasse von Zuständen aus S)

- Verteile die Zustände in der Klasse z so auf Teilklassen, daß zwei Zustände des Ausgangsautomaten $s_1, s_2 \in S$ genau dann in der gleichen Teilklasse z' sind, wenn für alle Eingabesymbole $x \in \Sigma$ gilt: $next(s_1, x)$ und $next(s_2, x)$ sind in der gleichen Klasse von $\mathcal{S}$.
- Ersetze z in $\mathcal{S}$ durch die so gebildeten Teilklassen.

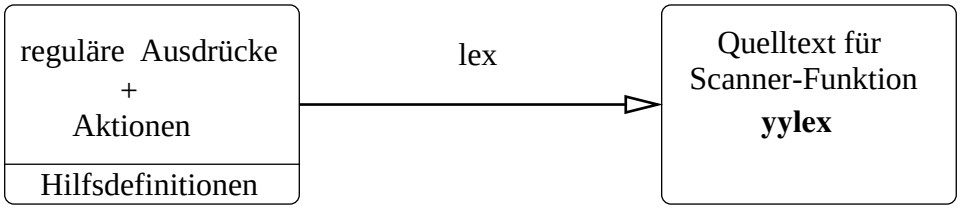
3.5 Scanner-Generatoren

Scanner-Generatoren erzeugen aus einer formalen Beschreibung der lexikalischen Syntax (reguläre Ausdrücke) einen Scanner. Die dazu notwendigen Schritte sind in 3.4, S. 31 beschrieben.

Es lohnt sich nicht, im Verlauf eines Compilerprojekts eigens einen Scanner-Generator zu entwickeln, um dann den Scanner generieren zu können: Der Aufwand ist unverhältnismäßig groß. (Bei der Parser-Entwicklung ist ein entsprechendes Vorgehen durchaus überlegenswert !)

Es lohnt sich dagegen durchaus, über den Einsatz eines Scannergenerators nachzudenken, denn gerade *lex*, *flex* und Konsorten sind auf vielen Rechnerplattformen verfügbar. Diese Generatoren erzeugen den Scanner in Form eines C-Quelltexts, so daß man an dem generierten Scanner noch Modifikationen auf der C-Ebene vornehmen kann. Auch für C++ und Pascal sind Generator-Varianten verfügbar.

3.5.1 Verwendung von *lex*



Die Verwendung von *lex* ist einfach:

- In der Eingabedatei gibt man für jedes Token an
  - die Syntax in Form eines regulären Ausdrucks
  - eine *Aktion* (C-Anweisung), die bei Erkennung des Tokens vom Scanner ausgeführt werden soll
- Der Generator erzeugt aus der Eingabedatei den Quelltext des Scanners, der aus der Definition der Scanner-Funktion *yylex* und aller benötigten Datenstrukturen und Hilfsfunktionen besteht.
- Der generierte Scanner wird compiliert und das erzeugte Objektmodul zusammen mit dem Parser und den anderen Modulen zum lauffähigen Compiler gebunden.

Die Schnittstelle zum einem von *yacc* generierten Parser betehen im wesentlichen aus folgenden Konventionen:

- Der Rückgabewert der *yylex*-Funktion identifiziert das erkannte Token in Integer-Codierung.
- Die globale Variable *yytext* enthält die Zeichen, aus denen das Token besteht.
- Die globale Variable *yyleng* gibt die Länge von *yytext* an.

In den Aktionen, die man den Tokens zuordnet, muß die gewünschte Parser-Schnittstelle explizit programmiert werden. Dadurch läßt sich ein *lex*-generierter Scanner mit einem beliebigen Parser leicht kombinieren. Das Kopieren des Tokens nach *yytext* geschieht allerdings automatisch, auch *yyleng* wird immer entsprechend angepasst.

Eingabedatei-Format

Eine typische *lex*-Eingabedatei hat folgenden Aufbau:

```
%{

    hier stehen C-Deklarationen, die in den generierten
    Scanner ohne Modifikation kopiert werden, dazu gehören
    alle Deklarationen, die in den Aktionen benötigt werden.

%}

   hier können reguläre (Hilfs-)Ausdrücke
   definiert werden, entweder, weil sie mehrfach benötigt
   werden, oder zur Strukturierung besonders komplizierter
   Ausdrücke

   Format:       Name  Ausdruck
   Verwendung:   {Name}
%%

   hier steht eine beliebige Liste von regulären Ausdrücken und
   zugehörigen Aktionen. Eine Aktion ist eine beliebige C-Anweisung.

%%
```

```
hier stehen Definitionen von Datenstrukturen und Funktionen, die
in den Scanner kopiert werden.
```

Bei der Verwendung benannter regulärer Ausdrücke ist darauf zu achten, daß *lex* eine reine Textersetzung durchführt, ohne den einzusetzenden Text zusätzlich mit Klammern zu versehen. Folgendes Beispiel zeigt die böse Falle auf:

```
BUCHSTABE  [a-z]|[A-Z]
%%
{BUCHSTABE}+     puts("Wort gefunden");
```

„`{BUCHSTABE}`“ wird ersetzt durch „`[a-z]|[A-Z]`“.
„`{BUCHSTABE}+`“ wird ersetzt durch „`[a-z]|[A-Z]+`“, nicht aber, wie gewünscht durch „`([a-z]|[A-Z])+`“. Sicherheitshalber sollte man also komplexe Ausdrücke in Definitionen mit Klammern versehen !

Der generierte Scanner verhält sich wie folgt:

- Falls ein Anfangsstück der Eingabedatei eindeutig zu einem Ausdruck zugeordnet werden kann, wird die dem Ausdruck zugeordnete Aktion ausgeführt. Sofern diese die Scannerfunktion nicht beendet (*return* oder *exit* in der Aktion) wird mit der Resteingabe genauso verfahren.
- Falls ein Anfangsstück der Eingabe einem Ausdruck A entspricht, ein längeres Anfangsstück aber einem Ausdruck B, wird die Aktion zu B ausgeführt. Der Scanner nimmt also immer das längstmögliche Token.
- Falls dasselbe Anfangsstück der Eingabe zu zwei verschiedenen AUsdrücken paßt, entscheidet die Reihenfolge: Die Aktion zum ersten Ausdruck wird ausgeführt.
- Falls ein Stück Eingabetext zu keinem der Ausdrücke paßt, wird es per Default einfach auf die Standardausgabe kopiert. Will man das nicht, sollte man dem regulären Ausdruck „.“, der ja zu jedem Eingabezeichen paßt eine andere Aktion (oder keine Aktion) zuordnen. Dieser Ausdruck muß als letzter angegeben werden.

Beispiel

- Folgende Behandlung von Integer- und Float-Literalen sei programmiert:

```
-?[0-9]+                              {return INT_NUMBER;}
-?[0-9]+(\.[0-9]+)?([eE]-?[0-9]+)?  {return FLOAT_NUMBER;}
```

Findet der Scanner eine Float-Konstante z.B. „-3.12“, wird korrekt die zweite Regel ausgeführt, da sich hierbei das längere Token (-3.12 statt -3) ergibt.

Findet der Scanner eine Integer-Konstante, passen beide Ausdrücke. Beim zweiten ist nämlich sowohl der Nachkomma- als auch der Exponenten-Anteil optional !.

Hier ist die Reihenfolge wesentlich: Die Integer-Aktion steht zuerst und wird korrekterweise asugeführt.

- Für die Erkennung von Schlüsselwörtern und die Abgrenzung zu Bezeichnern ist auch die Reihenfolge maßgeblich:

```
"while"                 {return WHILE;}
"do"                    {return DO;}
     ...
[_a-zA-Z][_a-zA-Z0-9]*  {return IDENTIFIER;}
```

Obwohl hier die Bezeichner-Syntax auch auf jedes beliebige Schlüsselwort paßt, werden die Schlüsselwörter korrekt erkannt, da sie zuerst spezifiziert werden !

Ein einfaches Beispiel ist ein Programm, das einen Text von der Standardeingabe kopiert und dabei alle Kleinbuchstaben in Großbuchstaben umwandelt. Der Generator erhält folgende Eingabedatei:

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%%
[a-z]    { putchar(toupper(*yytext)); }
%%
main(){
    yylex();
}
```

Ein weiteres Beispiel ist dem *flex*-Manual entnommen:

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+    {
    printf( "An integer: %s (%d)\n", yytext,
    atoi( yytext ) );
    }

{DIGIT}+"."{DIGIT}*        {
    printf( "A float: %s (%g)\n", yytext,
    atof( yytext ) );
    }

if|then|begin|end|procedure|function        {
    printf( "A keyword: %s\n", yytext );
    }

{ID}        printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"   printf( "An operator: %s\n", yytext );

"{"[^}\n]*"}"     /* eat up one-line comments */

[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );

%%

main( argc, argv )
int argc;
char **argv;
    {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
    yyin = fopen( argv[0], "r" );
    else
    yyin = stdin;

    yylex();
```

```
}
```

4 Syntaxanalyse

Die Syntaxanalyse prüft, ob Eingabetexte (z.B. Programm-Quelltexte) bezüglich einer zugrundeliegenden Grammatik syntaktisch korrekt sind. Ein Eingabetext ist genau dann syntaktisch korrekt, wenn ein Syntaxbaum, bzw. eine Ableitung gemäß den Regeln der Grammatik konstruierbar ist.

Während der Syntaxanalyse werden i.d.R. weitere Verarbeitungsschritte vorgenommen, z.B. semantische Prüfung, Aufbau einer internen Zwischen-Repräsentation für das Programm (Symboltabelle + Abstrakter Syntaxbaum) oder Zwischncode-Generierung. Diese Aspekte werden erst in den nachfolgenden Kapiteln behandelt. In diesem Kapitel geht es zunächst einmal nur um die Berechnung der Ableitung. Dann wird auf die Fehlerbehandlung eingegangen.

4.1 Strategien zur Berechnung des Ableitungsbaums

Ziel der Syntaxanalyse ist die (explizite oder implizite) Konstruktion des Ableitungsbaums zu einer Eingabe auf der Grundlage einer kontextfreien Grammatik.

Bei den **Top-Down-Verfahren** wird der Baum von der Wurzel aus konstruiert, d.h. die Ableitung entsprechend der zugrundeliegenden Grammatik wird vom Startsymbol ausgehend berechnet.

Bei den **Bottom-Up-Verfahren** wird der Baum von den Blättern aus konstruiert, die Ableitung wird rückwärts berechnet.

4.2 Mehrdeutige Grammatiken, Präzedenz und Assoziativität

Bevor konkrete Verfahren zur Bestimmung des Ableitungsbaums behandelt werden, soll das Problem der Mehrdeutigkeit behandelt werden.

Definition

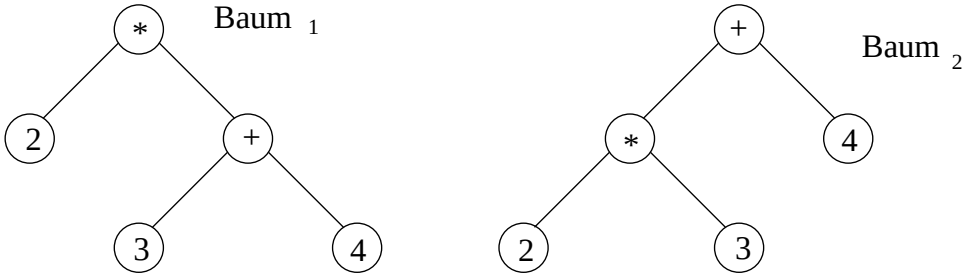
Eine Grammatik G heißt mehrdeutig, falls zu einem Wort w aus L(G) mehrere unterschiedliche Ableitungsbäume existieren.

Beispiel:

Expr →	Expr Operator Expr
	| '(' Expr ')'
	| KONSTANTE
	| BEZEICHNER
Operator →	'+' | '-' | '*' | '/'

Die Syntax von KONSTANTE und BEZEICHNER sei auf der lexikalischen Ebene definiert.

Mehrdeutigkeit zeigt sich etwa bei `2*3+4` :



4.2.1 Präzedenzen und Assoziativitätsregeln in Programmiersprachen

Bei ungeklammerten komplexen Ausdrücken ist die eindeutige Zuordnung der Operanden zu den Operatoren durch Präzedenz- und Assoziativitätsregeln definiert. (statt Präzedenz wird auch der Begriff Priorität gebraucht)

Präzedenz

Unter den Operatoren wird eine Präzedenzfolge festgelegt, ein Operator höherer Präzedenz bindet stärker.

Beispiel

Der Pascal-Ausdruck `3+4*5` wird aufgrund der höheren Präzedenz von `*` gegenüber `+` als `3+(4*5)` interpretiert.

Assoziativität

Falls Gruppen von Operatoren gleicher Präzedenz existieren, wird innerhalb

jeder Gruppe einheitlich linksassoziativ

$$a \bullet b \circ c = (a \bullet b) \circ c$$

bzw. rechtassoziativ

$$a \bullet b \circ c = a \bullet (b \circ c)$$

zugeordnet.

Wir bezeichnen einen Operator $\circ$ als *nicht-assoziativ*, wenn ein Ausdruck der Form $a \circ b \circ c$ nicht erlaubt ist.

Beispiel: In Pascal wird der Ausdruck `a-b+c` wegen der Links-Assoziativität der Additions-Operatorengruppe als `(a-b)+c` interpretiert.

Pascal-Operatoren in absteigender Präzedenzfolge:

not, - (einstellig)	nicht-assoziativ
*, /, **div**, **mod**, **and**	linksassoziativ
+, −, **or**	linksassoziativ
=, <=, >=, <>, <, >	nicht-assoziativ

C-Operatoren in absteigender Präzedenzfolge:

Bezeichnung	Operatoren	Assoziativität
Primäroperatoren	`() [] . ->`	links
einstellige Operatoren	`++ -- - ! ~ & * sizeof`	rechts
Multiplikationsoperatoren	`* / %`	links
Additionsoperatoren	`+ -`	links
Shift-Operatoren	`<< >>`	links
relationale Operatoren	`< > <= >=`	links
Gleichheitstest	`== !=`	links
Bit-and	`&`	links
Bit-exor	`^`	links
Bit-or	`|`	links
and	`&&`	links
or	`||`	links
conditional	`?:`	rechts
Zuweisungsoperatoren	`= += -= *= /= %= <<= >>= &= |= ^=`	rechts
Komma-Operator	`,`	links

Man beachte die unterschiedliche Einordnung der logischen Operatoren. Während in C Vergleichs- oder Zuweisungsoperatoren hintereinander auf-

treten dürfen, ist dies in Pascal nicht erlaubt. Daher ist auch keine Assoziativitätsregelung für diese Pascal-Operatoren erforderlich.

4.2.2 Präzedenz und Assoziativität in der Grammatik

Die Mehrdeutigkeiten bei ungeklammerten Ausdrücken mit mehreren Operatoren lassen sich durch Berücksichtigung von Präzendenzen und Assoziativitäten in der Grammatik auflösen.

Man beachte, daß es bei Parser-Generatoren (z.B. yacc) oft die komfortable Möglichkeit gibt, zu einer mehrdeutigen Grammatik die Präzedenzen und Assoziativitäten der Operatoren direkt anzugeben. Dies ist in aller Regel günstiger, da die Grammatik dadurch wesentlich kompakter gehalten werden kann.

Beispiele für einfache Ausdrucks-Grammatiken:

1. mehrdeutig keine Assoziativität für `+` festgelegt

   Expr → Expr '+' Expr
   Expr → '(' Expr ')'
   Expr → KONSTANTE

2. eindeutig, `+` ist linksassoziativ

   Expr → Expr '+' Term
   Expr → Term
   Term → '(' Expr ')'
   Term → KONSTANTE

3. eindeutig, `+` ist rechtsassoziativ

   Expr → Term '+' Expr
   Expr → Term
   Term → '(' Expr ')'
   Term → KONSTANTE

4. eindeutig, `+` und `*` linksassoziativ

   Expr → Expr '+' Term (*niedrigste Präzedenz*)
   Expr → Term
   Term → Term '*' Faktor (*mittlere Präzedenz*)
   Term → Faktor
   Faktor → '(' Expr ')' (*höchste Präzedenz*)
   Faktor → KONSTANTE

Schema zum Einbringen von Präzedenz und Assoziativität

Eine Ableitungsregel der Form $X \to Xw$ heißt *linksrekursiv*. Eine Ableitungsregel der Form $X \to wX$ heißt *rechtsrekursiv*.

Präzedenzgruppen und Assoziativitätsregeln für zweistellige Operatoren lassen sich in einer Grammatik wie folgt umsetzen:

- Jeder Präzedenzstufe i wird in der Grammatik ein eigenes Nonterminalsymbol $Expr_i$ zugeordnet.
- Ein linksassoziativer Operator Op_i der Stufe i wird durch die linksrekursive Regel
$$Expr_i \rightarrow Expr_i Op_i Expr_{i+1}$$
beschrieben. Dadurch ist sichergestellt, daß der rechte Operand weder den Operator selbst, noch Operatoren gleicher oder geringerer Präzedenz (z.B. Stufe i-1) in ungeklammerter Form enthält.
- Entsprechend wird ein rechtsassoziativer Operator durch die rechtsrekursive Regel
$$Expr_i \rightarrow Expr_{i+1} Op_i Expr_i$$
beschrieben.
- Für einen nichtassoziativen Operator Op der Stufe i lautet die Regel
$$Expr_i \rightarrow Expr_{i+1} Op_i Expr_{i+1}$$

4.2.3 Grammatik-Transformationen für Top-Down-Analyse

Entfernung linksrekursiver Regeln

Transformation bei direkter Linksrekursion:

Ersetze linksrekursive Ableitung

$$X \rightarrow X\alpha \mid \beta$$

durch

$$\begin{aligned} X &\rightarrow \beta X' \\ X' &\rightarrow \varepsilon \mid \alpha X' \end{aligned}$$

Der Algorithmus zur Beseitigung indirekter Linksrekursionen steht im „Drachenbuch“.

Linksfaktorisierung

Falls zwei alternative rechte Regelseiten für ein Nonterminal X mit einem gemeinsamen Präfix u der Länge k beginnen, ist die eindeutige Regelauswahl mit einer Vorausschau der Länge k nicht möglich:

$$\begin{aligned} X &\rightarrow uv_1 \\ X &\rightarrow uv_2 \end{aligned}$$

Durch Linksfaktorisierung läßt sich das Problem lösen:

- Erzeuge eine Regel für den gemeinsamen Präfix der rechten Regelseiten
- Repräsentiere den Rest durch eine neue Variable
- Generiere die unterschiedlichen Suffix-Alternativen aus der neuen Variablen

$$\begin{aligned} X &\rightarrow uX' \\ X' &\rightarrow v_1 \\ X' &\rightarrow v_2 \end{aligned}$$

4.3 Top-Down-Verfahren

4.3.1 Nichtdeterministische Spezifikation einer Top-Down-Analyse:

Eingabe: Grammatik $G = (N, T, S, P)$, Eingabewort $w \in T^*$

Ausgabe: Syntaxbaum zu w

Algorithmus:

Beginne die Konstruktion des Ableitungsbaums mit der Wurzel, die mit dem Startsymbol der Grammatik, S, markiert wird.

while (mit Nonterminalsymbol markierter Blattknoten vorhanden)
do

Bestimme einen mit einem Nonterminalsymbol A markierten Blattknoten zum neuen aktuellen Knoten k;

Wähle zu A eine Regel

$$A \to a_1 \ldots a_n$$

Erzeuge für jedes Symbol der rechten Seite einen mit diesem Symbol markierten Baumknoten als Nachfolger von k

end while

if (Blattmarkierungen stimmen mit w überein) **then**

konstruierter Baum ist Syntaxbaum zu w;

die in der Schleife ausgewählten Regeln bilden eine Ableitung für w

end if

Probleme

1. **Welcher Knoten ist auszuwählen, falls mehrere Blätter mit Nonterminalen markiert sind ?**

   Für eindeutige Grammatiken ist die Auswahlreihenfolge zunächst einmal ohne Bedeutung, denn den Syntaxbaum ist eindeutig bestimmt.

   Bei einer Linksableitung wird jeweils das äußert links vorkommende Nonterminal gewählt. (Analog: Rechtsableitung)

2. **Bei der Auswahl ungeeigneter Regeln führt die Baumkonstruktion in eine Sackgasse.**

   Lösungsmöglichkeit: „trial and error“
   Revidiere bei Sackgassen jeweils die zuletzt getroffene Auswahlentscheidung und fahre mit nächsten passenden Regel fort. (Backtracking) Falls alle Alternativen in eine Sackgasse führen, ist das Wort nicht ableitbar.

   Aber:

   Analyse mit Backtracking ist i.a. zu ineffizient (n Tokens - $O(n^3)$ Ableitungsschritte — für ein Programm mit 5000 Quelltextzeilen a 10 Tokens ist $n^3 = 1,25 * 10^{14}$!)

   Lösungsmöglichkeit:

   Anstatt irgendeine Regel auszuwählen, versucht man anhand der Eingabe die nächste Regel eindeutig zu bestimmen (Vorausschau, „lookahead“)

   Um den Abgleich mit der Eingabe durchzuführen, muß der Top-Down-Parser eine Linksableitung berechnen !

Falls eine Grammatik durch Vorausschau auf die nächsten k Tokens in jedem Fall eine eindeutige Regelauswahl ermöglicht, heißt sie LL(k)-Grammatik.

3. **Bei linksrekursiven Regeln sind Endlosschleifen möglich.**

   Lösungsmöglichkeit:

   Beseitigung der Linksrekursion durch Grammatik-Transformation.

4.3.2 Top-Down-Analyse durch rekursiven Abstieg

Voraussetzung: LL(1)-Grammatik $G = (N, T, P, S)$

Parser-Aufbau:

- Zu jedem Nonterminalsymbol $A \in N$ wird ein Unterprogramm $PARSE_A$ konstruiert, daß alle aus A ableitbaren Wörter analysiert
- $PARSE_A$ wählt anhand des lookahead-Symbols die wegen der LL(1)-Eigenschaft der Grammatik eindeutig bestimmte Ableitungsregel

$$A \to a_1 \ldots a_n$$

- Die rechte Seite $a_1 \ldots a_n$ wird von links nach rechts wie folgt verarbeitet:

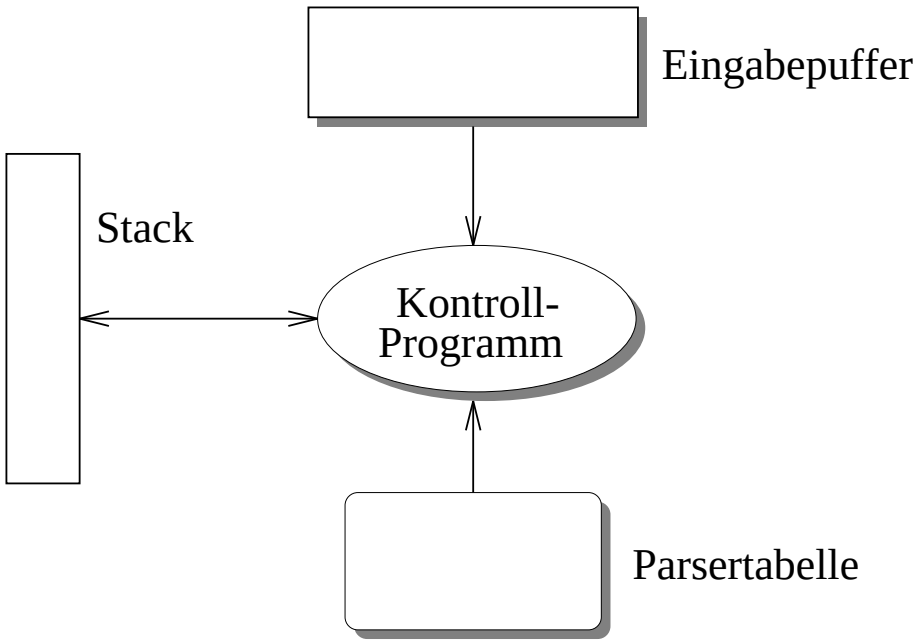
$a_i \in T \Longrightarrow$	**if** (a_i=lookahead-Symbol) **then**
	bestimme nächstes lookahead-Symbol
	else Fehler
	end if
$a_i \in N \Longrightarrow$	$PARSE_{a_i}$ aufrufen

Anmerkung

Falls die Grammatik keine eindeutige Regelauswahl erlaubt, kann das gleiche Schema verwendet werden. allerdings müssen bei mehreren Ableitungsalternativen mittels Backtracking alle Möglichkeiten durchprobiert werden. (Endlosrekursion vermeiden !)

4.3.3 Top-Down-Analyse mit tabellengesteuertem LL(1)-Parser

Modell eines tabellengesteuerten LL(k)-Parsers



Funktionsweise:

- Wir verwenden im folgenden $ als Endmarkierung der Eingabe
- Der Algorithmus gibt die Ableitung des Eingabeworts Regel für Regel aus

Der Algorithmus des Kontrollprogramms:

- Die Grammatik sei G=(T, N, P, S).
- In der Tabelle TAB steht zu dem gerade bearbeiteten Nonterminal $X \in N$ und dem aktuellen Eingabesymbol $a \in T$ die zu verfolgende Ableitungsregel aus P:

  $TAB[X, a] = X \rightarrow r$

  (oder, falls es diese nicht gibt, eine Fehlermarkierung *error*)
- X ist immer das oberste Stack-Element

```
push(S) /* Startsymbol S auf Stack */
a:=nexttoken /* Eingabesymbol := erstes Symbol */
repeat
    if Stack leer and a=$
    then stop /* erfolgreich beendet*/
    elseif X = a then /* "Match"eines Terminalsymbols */
        pop
        a:=nexttoken
    elseif X ∈ N and TAB[X, a] = (X → Y1Y2 ...Yk)
    then
        print(X → Y1Y2 ...Yk) /* Ausgabe des nächsten Ableitungs-
schritts*/
        pop
        push Yk, ..., push Y2, push Y1 /*Y1 wird als nächstes bearbeitet
*/
    else
        Fehlerbehandlung
    end if
end repeat
```

Konstruktion der LL(k)-Parsertabelle zur Grammatik G=(T, N, P, S):

1. Bestimme zu jedem Grammatik-Symbol $X \in (T \cup N)$ die Menge FIRST(X) (Menge aller Terminalzeichen, mit denen aus X abgeleitete Wörter beginnen können; enthält ε, falls ε aus X ableitbar)

   (a) Falls X ∈ T, FIRST(X)=X

   (b) Falls $(X \to \varepsilon) \in P$, übernehme ε in FIRST(X)

   (c) Für jede Regel $(X \to Y_1 \ldots Y_k) \in P$ verfahre wie folgt:
   Falls $\varepsilon \in FIRST(Y_1) \wedge \ldots \wedge \varepsilon \in FIRST(Y_{i-1})$ für ein i, $1 \leq i <\leq n$, übernehme alle Terminalzeichen aus FIRST(Y_i) in FIRST(X)
   Falls $\varepsilon \in FIRST(Y_1) \wedge \ldots \wedge \varepsilon \in FIRST(Y_k)$ übernehme ε in FIRST(X)

   (Man beachte, daß b und c iteriert werden müssen, bis sich an den FIRST-Mengen nichts mehr ändert, eine geschickte Implementierung berücksichtigt die Abhängigkeiten der Nonterminale zur Berechnung einer optimalen Reihenfolge → topologische Sortierung)

   Hat man zu jedem Symbol die FIRST-Menge berechnet, so läßt sich daraus zu jeder beliebigen Satzform $X_1 \ldots X_k$ die Menge der Terminale berechnen, mit denen aus der Satzform abgeleitete Wörter beginnen können (FIRST($X_1 \ldots X_k$)). 1 k

2. Bestimme zu jedem Nonterminal $X \in T$ die Menge FOLLOW(X) (Menge aller Terminalzeichen, die in einer Satzform direkt hinter X auftreten können; enthält $, falls X in einer Satzform ganz rechts auftreten kann)

   (a) Übernehme $ in FOLLOW(S)

   (b) Für jede Regel $(A \to uXw) \in P$ übernehme alle Terminalzeichen aus FIRST(w) in FOLLOW(X).

   (c) Für jede Regel $(A \to uB) \in P$ sowie jede Regel $(A \to uBv) \in P$ mit $\varepsilon \in FIRST(v)$, übernehme alle Terminalzeichen aus FOLLOW(A) in FOLLOW(B).

   (auch hier sind b und c zu iterieren)

3. Konstruiere die Parsertabelle durch Ausführung der folgenden Schritte a und b für jede Ableitungsregel A→a :

   (a) Für jedes Terminalsymbol $a \in FIRST(a)$ übernehme die Regel in $TAB[A, a]$

(b) Falls $\epsilon \in FIRST(a)$, übernehme die Regel in $TAB[A, b]$ für jedes $b \in FOLLOW(A)$
Falls $\epsilon \in FIRST(a)$ und \$ $\in FOLLOW(A)$, übernehme die Regel in TAB[A, \$].

Alle leeren Tabelleneinträge werden mit *error* markiert.

Eine Grammatik hat die LL(1)-Eigenschaft, g.d.w. kein Tabelleneintrag $TAB[X, a]$ mehrere Regeln enthält.

4.3.4 LL(k) - Verfahren bei mehrdeutigen Grammatiken

Mehrdeutigkeit führt bei der Syntaxanalyse zu Situationen, in denen die Auswahl einer Ableitungsregel nicht möglich ist. Da die Mehrdeutigkeit natürlich auch nicht durch Vorausschau aufgelöst werden kann, hat eine mehrdeutige Grammatik niemals die LL(k)-Eigenschaft für irgendein k.

Trotz allem werden mehrdeutige Grammatiken in Verbindung mit LL(k)-Verfahren zur Syntaxanalyse eingesetzt, denn sie sind oft wesentlich einfacher als entsprechende eindeutige Grammatiken.

Vorgehensweise bei mehrdeutigen Grammatiken:

- man bestimme die LL(k)-Parsertabelle wie gewohnt.
- die Tabelle enthält an mindestens einer Stelle mehrere Ableitungsregeln
- man mache die Tabelle durch Entfernen der „unerwünschten“ Regeln eindeutig

4.4 Bottom-Up-Syntaxanalyse (LR-Verfahren)

Funktionsprinzip: Rückwärts-Konstruktion einer **Rechtsableitung**

Aktionen des Parsers:

shift	nächstes Token auf den Stack legen
reduce(i)	oben auf dem Stack wurde ein „handle“, d.h. die rechte Seite der nächsten Ableitungsregel (i) erkannt; ersetze das handle durch die linke Seite der Regel
accept	Eingabe gelesen, Startsymbol auf Stack $\Rightarrow$ akzeptiere
error	Syntaxfehler

Beispiel

Expr	$\rightarrow$	Expr '+' Term
Expr	$\rightarrow$	Term
Term	$\rightarrow$	Term '*' Factor
Term	$\rightarrow$	Factor
Factor	$\rightarrow$	'(' Expr ')'
Factor	$\rightarrow$	KONSTANTE

Parser-Konfigurationen

Nr.	Stack (handles fett)	Aktion	Resteingabe vor Aktion
1		shift	3 * 5 + 6 * 7
2	**KONSTANTE**	reduce(6)	* 5 + 6 * 7
3	**Factor**	reduce(4)	* 5 + 6 * 7
4	Term	shift	* 5 + 6 * 7
5	Term *	shift	+ 6 * 7
6	Term * **KONSTANTE**	reduce(6)	+ 6 * 7
7	**Term * Factor**	reduce(3)	+ 6 * 7
8	**Term**	reduce(2)	+ 6 * 7
9	Expr	shift	+ 6 * 7
10	Expr +	shift	6 * 7
11	Expr + **KONSTANTE**	reduce(6)	* 7
12	Expr + **Factor**	reduce(4)	* 7
13	Expr + Term	shift	* 7
14	Expr + Term *	shift	7
15	Expr + Term * **KONSTANTE**	reduce(6)	
16	Expr + **Term * Factor**	reduce(3)	
17	**Expr + Term**	reduce(1)	
18	Expr	accept	

Grobspezifikation der Arbeitsweise der LR-Parser:

while not Eingabeende **do**
  while not nächstes *handle* oben auf Stack **do**
    shift
  end while
  reduce
end while

Zwei Konfliktsituationen sind möglich:

1. reduce/reduce-Konflikt

   Der aktuelle Stack läßt unterschiedliche Reduktionen zu
   Welche Regel ist zu wählen ?

2. shift/reduce-Konflikt

   Der aktuelle Stack läßt (mindestens) eine Reduktion zu. Soll reduziert werden oder sollen weitere Eingabesymbole gelesen werden (shift), um ein anderes „handle“ zu erhalten ?

Konfliktsituationen im Beispiel:

Konfiguration 4: shift/reduce-Konflikt

Statt shift wäre auch reduce(2) ($Expr \rightarrow Term$) möglich gewesen.

Durch eine Vorausschau auf das nächste Symbol kann der Parser erkennen, daß der Stack-Inhalt kein „handle“, sondern nur ein Teil eines „handle“ der Form *Term * Factor* sein muß.

Konfiguration 7: reduce/reduce-Konflikt

Statt reduce(3) wäre *Factor* auch ein „handle“-Kandidat für reduce(4) ($Term \rightarrow Factor$) gewesen (führt in Sackgasse !).

Aufgrund der „Vorgeschichte“, genaugenommen aufgrund der Tatsache, daß $Term * Factor$ oben auf dem Stack steht, kann die richtige Reduktion bestimmt werden.

Die Lösung der Konfliktsituationen ist sowohl vom Stack-Inhalt als auch von der Vorausschau abhängig.

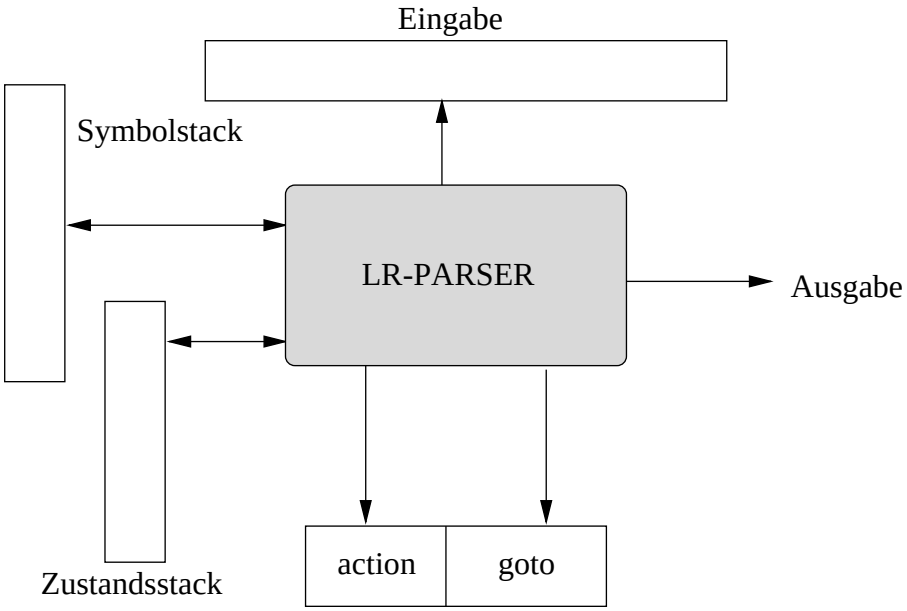
Ergebnis der LR-Parser-Theorie: Die Menge der möglichen Stack-Inhalte bildet eine reguläre Sprache

Konsequenz: Statt zur Konfliktlösung den Stack zu analysieren (ineffizient !), kann die notwendige Information in Form eines DEA-Zustands berechnet werden. Der aktuelle Zustand wird auf dem Stack abgelegt; er enthält die zur Konfliktlösung notwendige Information über den darunterliegenden Stack-Inhalt.

Die Bestimmung von Zustandsmenge und Parser-Tabelle, in der jedem Zustand abhängig von der Vorausschau ein Folgezustand und eine Aktion zugeordnet ist, erfolgt für die einzelnen Verfahren (LR, SLR, LALR) unterschiedlich.

4.4.1 LR-Parser-Schema

Das nachfolgende Schema verdeutlicht die Arbeitsweise eines LR-Parsers. SLR- und LALR-Parser arbeiten auf gleiche Weise, nur die verwendeten Tabellen sind unterschiedlich.



Der Parser ist anfangs im Ausgangszustand *s0*. Dieser Zustand wird auf den Zustandsstack geschoben, der Symbolstack ist leer.

Der nächste Schritt des Parsers bestimmt sich

- aus dem lookahead-Symbol l
- dem aktuellen Zustand s

durch den Aktionstabelleneintrag **action**(l, s)

Fall 1: action(s, l)=*shift s1*

Parser schiebt l in einer SHIFT-Aktion auf den Symbolstack, geht über in Zustand *s1* und schiebt *s1* auf den Zustandsstack.

Fall 2: action(s, l)=*reduce* $A \rightarrow B$

Sei n die Anzahl der Symbole auf der rechten Regelseite B, d.h. die Länge des auf dem Symbolstack liegenden „Handles“.

Sowohl vom Zustandsstack als auch vom Symbolstack werden n Elemente entfernt. A wird auf den Symbolstack geschoben. Der neue auf den Zustandsstack zu schiebende Zustand ergibt sich aus dem jetzt oben liegenden Zustand *s1* aus der *goto*-Tabelle: **goto**(*s1*, A)

4.5 Fehlerbehandlung

Unproblematisch ist, bei erkanntem Fehler eine aussagefähige Fehlermeldung ausgeben.

Problematisch dagegen ist die Frage nach der Fortsetzung der Syntaxanalyse eines fehlerhaften Programms.

Heuristische Vorgehensweise beim Wiederaufsetzen

Jeder Fehlersituation ist eine **Hypothese über die Art des Fehlers** zuzuordnen, z.B.

- Terminalsymbol wurde vergessen
- komplette Sub-Struktur wurde vergessen
- Schreibfehler bei Operator oder Bezeichner (Scanner liefert spezielles Fehler-Token)
- Schreibfehler in einem Schlüsselwort (Scanner liefert undefinierten Bezeichner)

Je nach Hypothese werden **fehlende Terminalsymbole ergänzt** oder auch **fehlerhafte Strukturen ignoriert**.

Als **Wiederaufsetzpunkte** für die Analyse einer fehlerhaften Struktur S bieten sich die Symbole in *FOLLOW(S)* an. Insbesondere sind Schlüsselwörter geeignet, da diese selten vergessen oder falsch geschrieben werden.

Daraus ergibt sich für einen „recursive descent"-Parser die Forderung, daß jede Parser-Prozedur die am Ort Ihres Aufrufs gültige FOLLOW-Menge kennt (diese ist ggf. als zusätzlicher Parameter zu übergeben).

Zielloses Überspringen von Text ist allerdings zu vermeiden, denn gerade das als Wiederaufsetzpunkt gesuchte Folgesymbol könnte ja auch fehlen! Hier können eindeutige **Anfangssymbole wichtiger Satzformen** als „Notbremse" dienen.

Man beachte insbesondere **Erfahrungswerte**, wie etwa das regelmäßige Vergessen von Interpunktationszeichen (z.B. Semikolon am Zeilenende). Hier kann der Parser intern von einer Grammatik ausgehen, bei der das Interpunktationszeichen optional ist. Natürlich erfolgt beim Fehlen eine entsprechende Meldung, ansonsten wird die Analyse „normal" fortgesetzt.

Solche Grammatik-Erweiterungen durch **Fehler-Produktionen** sind insbesondere auch bei automatischer Parser-Generierung sinnvoll.

4.6 Wertung der Analyseverfahren

Forschungsboom vor 25-30 Jahren erbrachte umfangreiche Parsing-Theorien und -Verfahren, z.B.

Parser-Klasse	**Grammatik-Klasse**
Top-Down	
Rekursiver Abstieg	ohne Linksrekursion
LL(k)	ohne Linksrekursion, Regelauswahl durch Vorausschau auf k Tokens möglich
Bottom-Up	
Operator-Präzedenz-Verfahren	Konfliktlösung durch Präzedenztabelle, besonders für Ausdrücke
SLR-Verfahren	einfachste LR-Methode
LALR-Verfahren	mächtiger, aber komplexer als SLR
kanonisches LR(k)-Verfahren	mächtiger, aber komplexer als LALR — Konfliktlösung durch Vorausschau auf k Tokens möglich
Early-Verfahren	beliebige kontextfreie Grammatik
Cocke-Kasami-Younger-Verfahren	beliebige kontextfreie Grammatik

- **Kompromiß bei Auswahl des Verfahrens erforderlich:** Je mehr spezielle Grammatik-Eigenschaften vorausgesetzt werden, desto einfacher und effizienter sind die verwendbaren Verfahren
- Top-Down-Verfahren sind (ohne Compilerbau-Werkzeuge) einfacher zu implementieren als Bottom-Up-Verfahren
- Bottom-Up-Verfahren sind insofern allgemeiner als Top-Down-Verfahren, als für viele reale Programmiersprachen LR(k)-Grammatiken existieren, LL(k)-Grammatiken dagegen nicht.

  (LL(k)-Grammatiken können allerdings – mit entsprechendem Transformations-Aufwand – für die meisten Sprachen erstellt werden).
- Jede LL(k)-Sprache ist eine LR(k)-Sprache. Die Umkehrung gilt nicht. (L ist LL(k)- (LR(k)-) Sprache, falls eine LL(k)- (bzw. LR(k)-) Grammatik für L existiert)

4.7 Der Parser-Generator *yacc*

Yacc steht für *yet another compiler-compiler*. Der Parser-Generator ist auf unterschiedlichen Systemplattformen verfügbar und weit verbreitet. Der kompatible Generator *bison* ist unter den GNU-Linzenzbedingungen als

Quelltext frei verfügbar. Das nachfolgenden Erläuterungen zu *yacc* gelten auch für *bison*, soweit nicht explizit auf Unterschiede hingewiesen wird.

Yacc erhält als Eingabe eine Grammatik, die die korrekte syntaktische Struktur der vom Parser zu analysierenden Eingabetexte beschreibt. Der Generator erzeugt daraus einen tabellengesteuerten Bottom-Up-Parser in Form eines C- (oder C++ -)Quelltexts. Der wesentliche Bestandteil des generierten Parsers ist die Definition der Analysefunktion

```
int yyparse(void)
```

Diese Analysefunktion prüft zunächst nur die Eingaben auf syntaktische Korrektheit gemäß der Grammatik. Für praktische Anwendungen sind aber darüber hinaus gehende Verarbeitungsschritte, je nach der gewählten Phasenaufteilung des Compilers z.B. Typprüfungen, Erzeugung eines Syntaxbaums und einer Symboltabelle, oder Zwischencode-Erzeugung notwendig. Diese sollen schon während der Syntaxanalyse durchgeführt werden, müssen also in die Analysefunktion *yyparse* integriert werden. Dazu unterstützt der Generator die Möglichkeit, semantische Aktionen in Form von C-Anweisungen in die Regeln der Grammatik einzubetten.

Im nächsten Kapitel wird darauf näher eingegangen.

4.7.1 Das Eingabe-Dateiformat

Die Eingabedatei für den Generator hat (typischerweise) folgende Bestandteile:

```
%{
  C-Deklarationen
%}
  yacc-Deklarationen
%%
  Grammatik + Aktionen
%%
  C-Definitionen
```

Beispiele für yacc-Deklarationen:

```
/* Startsymbol */
%start Module

/* Operatoren, geringste Praezedenz zuerst */
%nonassoc '=' LEQ '<' GEQ '>' '#'
%left '+' '-' OR
%left DIV MOD '&' '*'
%right '~'
%right UMINUS
%right '.' '['

/* sonstige Tokens */
%token ASSIGN IDENT INTEGER MODULE TYPE CONST ...
```

Das Format für die Grammatik-Regeln entnehmen Sie bitte den folgenden Beispielen:

```
FieldList : IdentList ':' Type
          | /* EMPTY */
;

FieldDecls : FieldList ';' FieldDecls
           | FieldList
;

RecordType : RECORD FieldDecls END ;

Type : IDENT | ArrayType | RecordType ;

WhileStatement : WHILE Expression DO StatementSequence END ;

Statement : Assignment | ProcedureCall | IfStatement | WhileStatement ;
```

In die rechten Seiten der Regeln lassen sich, wie oben schon gesagt, *semantische Aktionen*, einstreuen, deren Stellung innerhalb der Regel den Zeitpunkt der Ausführung durch den generierten Parser bestimmt.

Beispiel:

```
X : X1 { Aktion_1 } X2 { Aktion_2 } X3 ;
```

Der generierte Parser wird die drei Bestandteile von X analysieren, nach der Analyse von X1 die erste und nahc der Analyse von X2 die zweite Aktion ausführen.

Die C-Deklarationen und -Definitionen werden vom Generator in den generierten Parser-Quelltext kopiert. Benötigt werden in jedem Fall die Scanner-Fuktion *yylex* und eine Fehlermeldungsausgabe *yyerror*:

```
int yylex(void) /* Scanner */

int yyerror(char*mesg){
  /* Fehlerausgabe */
  fputs(mesg,stderr);
}

main(){ ... }
```

4.7.2 „Debuggen“ einer Grammatik

Rufen Sie den Generator (*yacc* oder *bison*) auf, um Ihre Grammatik auf **Konfliktfreiheit** zu testen:

```
bison -dtv parser.y
```

oder

```
yacc -b parser -dtv parser.y
```

Aufrufoptionen:

-d	erzeugt die C-Definitionen für die Tokenklassen in einem separaten Header-File (bison: `parser.tab.h`, yacc: `y.tab.h`).
-t	erzeugt Code zum debuggen des generierten Parsers
-v	erzeugt eine lesbare Spezifikation des generierten Parsers mit allen Konfliktmeldungen
-b *prefix*	weist *yacc* an, als Präfix für die Ausgabedateien den String *prefix* zu verwenden. Ansonsten wird *y* genommen.

Ausgabedateien:

`parser.tab.h`	Scanner-Schnittstelle: Token-Definitionen u.a.
`parser.tab.c`	Parser-Quelltext: *yyparse* u.a.
`parser.output`	Parser-Beschreibungsdatei zum Debuggen der Grammatik

4.7.3 „Debuggen“ des generierten Parsers

Wird der Generator mit der Option `-t` aufgerufen, erzeugt er im generierten Programm Anweisungen zum Verfolgen der Analyse. Der Parser gibt seine Aktionen dann auf *stderr* detailliert aus.

Dieser Code wird vom C-Compiler nur mitübersetzt, wenn `YYDEBUG` definiert ist. Im C-Deklarationsteil der yacc-Eingabedatei sollte dann am Anfang stehen:

```
#define YYDEBUG 1
```

Der Code zum Verfolgen der Analyse kann dann zur Laufzeit des Parsers nach Belieben aktiviert oder deaktiviert werden:

```
extern int yydebug;

...

main(){
   yydebug=1; /* Debugging einschalten */
   yyparse();
   return 0;
}
```

Zur partiellen Verfolgung bietet es sich an, das Ein- und Ausschalten des Protokolls als semantische Aktion in die Regeln der Grammatik aufzunehmen. Beispiel:

```
/* Verfolgen der Analyse der rechten Seite einer Wertzuweisung */
Assignment : IDENT Selector ASSIGN
             {yydebug=1;} Expression {yydebug=0;} ;
```

Eine yacc-Datei zum Eingewöhnen

Machen Sie sich mit *bison* / *yacc* zunächst anhand der nachfolgenden Eingabedatei `grammar.y` vertraut.

Mit

```
make YACC=bison YFLAGS=-vdt grammar
```

sollten Generator, C-Compiler und Linker daraus einen lauffähigen Parser namens *grammar* erzeugen, den Sie über die Standardeingabedatei mit der Testeingabe `abbc` versorgen sollten.

```
/* grammar.y - yacc Test-Eingabedatei */

%{
#include <stdio.h>
```

```
#include <ctype.h>
#define YYDEBUG 1
extern int yydebug;

%}

%%
S: A B C ;
A: 'a' | 'b' ;
B: 'b' B | /* empty */ ;
C: 'c' ;
%%
int yyerror(char*mesg){
  fputs(mesg,stderr);
}

int yylex(){
  int c;

  if (feof(stdin))
    return(-1);

  while(isspace(c=getchar()))
       ;

  return c;
}

main(){
  yydebug=1;
  yyparse();
  exit(0);
}
```

4.8 Fehlerbehandlung mit yacc/bison

Momentan (Zeitmangel !) muss an dieser Stelle der Verweis auf die Bison-Dokumentation genügen:

```
Error Recovery
**************

   It is not usually acceptable to have a program terminate on a parse
error.  For example, a compiler should recover sufficiently to parse the
```

rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to allow ‘yyparse’ to return 1 on error and have the caller ignore the rest of the input line when that happens (and then call ‘yyparse’ again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token ‘error’. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Bison parser generates an ‘error’ token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue.

For example:

```
stmnts:  /* empty string */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any ‘stmnts’.

What happens if a syntax error occurs in the middle of an ‘exp’? The error recovery rule, interpreted strictly, applies to the precise sequence of a ‘stmnts’, an ‘error’ and a newline. If an error occurs in the middle of an ‘exp’, there will probably be some additional tokens and subexpressions on the stack after the last ‘stmnts’, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the ‘error’ token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete ‘stmnts’.) At this point the ‘error’ token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Bison reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies

for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
stmnt: error ';'  /* on error, skip until ';' is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary:  '(' expr ')'
        | '(' error ')'
        ...
        ;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one ‘stmnt’. Suppose that instead a spurious semicolon is inserted in the middle of a valid ‘stmnt’. After the error recovery rule recovers from the first error, another syntax error will be found straightaway, since the text following the spurious semicolon is also an invalid ‘stmnt’.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Note that rules which accept the ‘error’ token may have actions, just as any other rules can.

You can make error messages resume immediately by using the macro ‘yyerrok’ in an action. If you do this in the error rule’s action, no error messages will be suppressed. This macro requires no arguments; ‘yyerrok;’ is a valid C statement.

The previous look-ahead token is reanalyzed immediately after an error. If this is unacceptable, then the macro ‘yyclearin’ may be used to clear this token. Write the statement ‘yyclearin;’ in the error rule’s action.

For example, suppose that on a parse error, an error handling routine is called that advances the input stream to some point where parsing should once again commence. The next symbol returned by the lexical scanner is probably correct. The previous look-ahead token ought to be discarded with ‘yyclearin;’.

```
   The macro ‘YYRECOVERING’ stands for an expression that has the value
1 when the parser is recovering from a syntax error, and 0 the rest of
the time.  A value of 1 indicates that error messages are currently
suppressed for new syntax errors.
```

5 Semantische Analyse

5.1 Programmiersprachliche Grundlagen

5.1.1 Typen

Die meisten modernen Programmiersprachen sind **typisiert**. Bei der Deklaration von Variablen und Unterprogrammen muss der Programmierer i.d.R. deren Typinformation genau angeben. Der Compiler kann mit diesen Angaben jedem Ausdruck eindeutig ein Typ zuordnen.

Vorteile der Typisierung:

- Der Compiler kann bei allen Operatoren, Funktionen und Variablen überprüfen, ob die Verwendung Typkonform ist. Dabei wird ein großer Teil der Codierfehler aufgedeckt, die sonst durch mühsames Debuggen zur Laufzeit ermittelt werden müssten. Die Programmentwicklung wird also deutlich sicherer. Vor allem bei grossen Programmsystemen, die aus vielen getrennt übersetzten Modulen bestehen, ist die (Modulübergreifende) Typprüfung ein außerordentlich wertvolles Hilfsmittel für die Entwickler.
- Mit der Typinformation sind Laufzeitoptimierungen möglich, die Programme werden schneller.

Nachteile der Typisierung

- In den meisten heute verwendeten Sprachen ist das Typsystem für gehobene Ansprüche nicht flexibel genug. Insbesondere parametrisierbare Typen sind gar nicht oder nur eingeschränkt benutzbar.

  Dies ist kein Nachteil der Typisierung an sich, sondern zeugt schlichtweg von einer gewissen technischen Rückständigkeit der gebräuchlichen Sprachen.
- Durch die meist erforderlichen Deklarationen wird der Programmumfang etwas umfangreicher. Dies ist i.d.R. vernachlässigbar.

Ein Typ legt eine Wertemenge zusammen zugehörigen Operationen fest. Damit ist auch die interne Repräsentation eindeutig festgelegt.

Wir unterscheiden

- vordefinierte Typen

  Dies sind meist einfache Basistypen wie *integer* oder *boolean*)

- benutzerdefininierte Typen

  mit Typkonstruktoren kann der Programmierer beliebige neue Typen definieren, um die für die Applikation benötigten Daten zu modellieren.

  Dazu gehören z.B. Aufzählungstypen (C: *enum*), Feldtypen, Verbundtypen (C: *struct*), Typvarianten (C: *union*), Zeigertypen oder Funktionstypen.

  Die Konstruktoren lassen nahezu beliebig kombinieren.

Die Compiler-interne Darstellung von Typen muss die Verschachtelbarkeit der Konstruktoren berücksichtigen: Typen sind durch komplexe, hierarchisch aufgebaute Datenstrukturen zu repräsentieren. Naheliegend ist ein abstrakter Syntaxbaum pro Typ.

5.1.2 Deklarationen

Eine Deklaration führt einen neuen Bezeichner ein. Der Compiler erzeugt aus der Deklaration einen Symboltabelleneintrag mit den für die Übersetzung benötigten Attributen.

5.1.3 Sichtbarkeit von Bezeichnern

Für die Erstellung komplexer, aus vielen separaten Modulen bestehender Softwaresysteme, an der i.d.R. viele Entwickler beteiligt sind, wäre es sehr hinderlich, wenn alle Bezeichner systemweit eindeutig sein müssten.

Stattdessen führt man das Konzept der Lokalität von Bezeichner ein. So kann beispielsweise der Programmierer innerhalb einer Funktionsdefinition lokale Hilfsvariablen definieren und verwenden, ohne sich darum zu kümmern, ob in anderen Funktionen deren Bezeichner schon einmal verwendet wurden.

Gemäß dem Geheimnisprinzip sind die lokalen Bezeichner ausserhalb der Funktionsdefinition nicht sichtbar, d.h. nicht benutzbar.

Der Bereich innerhalb eines Programm-Quelltextes, in dem ein Bezeichner benutzbar ist, heisst *Gültigkeitsbereich* oder *Sichtbarkeitsbereich* (*Scope*).

Im Compiler wird ein Gültigkeitsbereich durch eine Symboltabelle repräsentiert (ggf. durch eine Untertabelle).

5.1.4 Polymorphismus, Überladungen, Konversionen

Polymorphismus liegt vor, wenn in einer typisierten Sprache die Zuordnung einer Definition zu einer Verwendungsstelle eines Bezeichners erst zur Programmlaufzeit möglich ist. Der Compiler kann also den Typ des Bezeichners nicht (oder nicht vollständig) bestimmen.

Beispiel: Polymorphismus in einer objektorientierten Sprache

Eine Klasse *2D-Objekt* hat zwei speziellere Subklassen: *Rechteck* und *Kreis*. Sowohl für *Rechteck*-Objekte als auch für *Kreis*-Objekte sei eine spezifische Methode *zoom* definiert, die die Darstellung des Objekts auf dem Bildschirm in der Grösse verändert.

Sei *zwei_D_o* ein Objekt der Klasse *2D-Objekt*, *r* ein *Rechteck*-Objekt und *k* ein *Kreis*-Objekt.

```
if (...)
   zwei_D_o = r;
else
   zwei_D_o = k;

zwei_D_o.zoom(...);
```

Bei dynamischer Bindung wird erst zur Laufzeit des Programms bestimmt, ob der o.g. zoom-Aufruf die *Rechteck*-Methode *zoom* oder die gleichlautende *Kreis*-Methode aktiviert. Die Auswahl ist abhängig vom aktuellen Subtyp des Objekts.

Insofern ist also der Typ von *zwei_D_o* nicht vollständig vom Compiler bestimmbar, also polymorph.

Von *Überladungen* spricht man dagegen, wenn mehrere Deklarationen desselben Bezeichners (oder Operators) nebeneinander gültig sind, die eindeutige Zuordnung einer Deklaration aber durch den Compiler erfolgen muss.

In vielen Sprachen ist beispielsweise der Additionsoperator "+" überladen: Integeraddition oder Gleitkomma-Addition. Der Compiler stellt aber bei jeder Verwendung eindeutig fest, um welche Operation es sich handelt. Bietet die Programmiersprache dem Benutzer individuelle Überladungen an, so könnte auch die benutzerdefinierte Vektoraddition durch den "+"-Operator bezeichnet werden.

Konversionen, genauer gesagt *Typkonversionen*, erfolgen explizit durch Konversionsoperationen oder implizit gemäß sprachspezifischen Konversionsregeln.

Automatische Typanpassungen ersparen dem Programmierer etwas Schreibaufwand, erschweren andererseits aber auch die Fehlererkennung durch den

Compiler.

5.1.5 Typprüfung

In einer typisierten Sprache sind für jeden Operator und jede Funktion Argumenttypen und Resultatstypen bekannt. Die Typprüfung stellt bei jeder Operator-Verwendung bzw. bei jedem Funktionsaufruf fest, ob die erwarteten Argumenttypen kompatibel sind zu den Typen der aktuellen Parameter.

Die Typen der aktuellen Parameter müssen dazu natürlich erst berechnet werden. Damit ergibt sich die Notwendigkeit, für jeden Ausdruck dessen Typ zu bestimmen.

Die Typisierung erfolgt „bottom-up“. Durch Konversionen, Überladungen und Polymorhismus wird die Typberechnung deutlich komplizierter.

5.2 Syntaxorientierte Spezifikation – Syntaxorientierte Übersetzung

Ein Übersetzungsprozess erfordert unterschiedliche Berechnungen z.B.

- Überprüfung von Typkompatibilitäten
- Erzeugung interner Programmrepräsentation
- Erzeugung Zielmaschinen-Code

syntaxorientierte Übersetzung heißt:

Algorithmen für Berechnungen werden eindeutig den Syntaxregeln der Sprache zugeordnet

Vorteile:

- flexibler Spezifikationsmechanismus
- automatische Generierung eines Übersetzers (Compiler-Compiler) möglich

Varianten syntaxorientierter Übersetzung:

- Übersetzungsschemata

  beliebige Algorithmen (semantische Aktionen) werden in die Syntaxregeln eingefügt
- Attributierte Grammatiken

- jedem Grammatik-Symbol können nach Bedarf Attribute als Informationsträger zugeordnet werden
- jeder Syntaxregel werden die Algorithmen für die Attribut-Berechnung zugeordnet

5.3 Attributierte Grammatiken

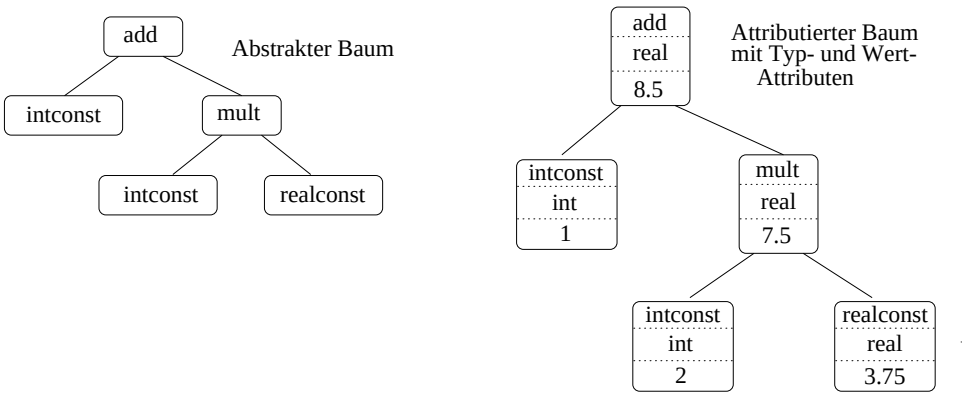
Ein Attribut ist eine Information, die an einen Knotentyp im (abstrakten) Syntaxbaum geknüpft ist.

Alle für die Übersetzung benötigten Informationen können als Attribute der Baumknoten definiert werden.

Beispiel: In einer typisierten Sprache ist ein Berechnung der Typen für alle Ausdrücke notwendig. Jedem Ausdrucks-Knoten im Baum wird das Attribut „Typ“ zugeordnet

Ein Syntaxbaum mit berechneten Attributwerten heißt *attributierter* oder *dekorierter Baum*.

Beispiel: Pascal-Ausdruck `1 + 2 * 3.75`



5.3.1 Vererbte und synthetische Attribute

Ein Attributwert, der einem Baumknoten vom Vaterknoten übergeben wird, heißt **vererbt** (top-down-Berechnung).

Ein Attributwert, der an den Vaterknoten übergeben wird, heißt **synthetisch** (bottom-up-Berechnung).

Eine **attributierte Grammatik** ist eine Grammatik $G = (T, N, S, P)$ mit folgenden Erweiterungen:

- jedem Symbol $X \in \{T \cup N\}$ ist eine endliche Menge von Attributen $\{X.A_1, \ldots, X.A_k\}$ zugeordnet.
- Zu jeder Ableitungsregel $X \rightarrow w$ wird angegeben, wie die synthetischen Attribute von X zu berechnen sind.
- Zu jeder Ableitungsregel $X \rightarrow w$ wird angegeben, wie die vererbten Attribute aller in w vorkommenden Nonterminalzeichen zu berechnen sind.

Notation: Durch die spezielle Schreibweise $X. \uparrow A_i$ wird ein Attribut A_i von X als synthetisches Attribut kenntlich gemacht, während $X. \downarrow A_i$ ein vererbtes Attribut ist.

Natürlich muß noch festgelegt werden, in welcher Form die Algorithmen zur Attributwertberechnung angegeben werden sollen. Wenn man Tools verwendet, die Attributierte Grammatiken verarbeiten, ist für jedes System eine spezielle Sprache dafür definiert. Für unsere Beispiele verwenden wir Pseudo-Code, der an C oder Pascal angelehnt ist oder die unten beschriebene *yacc*-Notation.

Falls ein Nonterminalsymbol X in einer Regel mehrfach vorkommt, so verwenden wir die Indizes 1, 2, . . . von links nach rechts zur eindeutigen Zuordnung der Attribute, z.B.

$$Expr \rightarrow Expr + Expr \; \{Expr_1. \uparrow Wert := Expr_2. \uparrow Wert + Expr_3. \uparrow Wert\}$$

5.3.2 yacc-Notation für Attribute

Der Parser-Generator *yacc* unterstützt synthetische Attribute.

Jedem Grammatik-Symbol kann man nur **ein** Attribut zuweisen. Alle Attribute haben den gleichen Typ YYSTYPE.

Dies sieht auf den ersten Blick wie eine Einschränkung aus, denn man möchte schließlich zu jedem Grammatik-Symbol einen eigenen Satz von spezifischen Attributen verwenden.

Da jedoch YYSTYPE ein in C beliebig definierbarer Typ ist, kann man dem dadurch Rechnung tragen, daß man YYSTYPE als *union*-Typ deklariert und für jedes Grammatik-Symbol eine spezifische Variante definiert.

Auch die Einschränkung auf ein einziges Attribut pro Grammatik-Symbol ist keine echte Einschränkung, denn die zugehörige Typ-Variante kann ein strukturierter Typ mit beliebigen Komponenten sein.

Die Attributwertberechnung wird durch *Aktionen*, das sind C-Verbundanweisungen, spezifiziert, in denen kann auf die Attribute mit $$ (Attribut der linken Regelseite), $1, $2 usw. (Attribute des 1., 2., ...Symbols auf der rechten Regelseite) zugegriffen werden.

Die Deklaration von YYSTYPE erfolgt mit einer speziellen `%union`-Anweisung, die Zuordnung einer Variante von YYSTYPE zu einem Symbol der Grammatik durch die speziellen Deklarationen

`%token` < Variante > *Token* ...
`%left` < Variante > *Operator* ...
`%right` < Variante > *Operator* ...
`%nonassoc` < Variante > *Operator* ...
`%type` < Variante > *Noterminal* ...

Beispiele für *yacc*-Spezifikationen finden sich in der Aufgabenstellung der 2. Hausübung.

5.3.3 Beispiel für eine attributierte Grammatik: Roboter

Ein Roboter kann sich auf einen der Befehle N, O, S, W schrittweise in eine der vier Himmelsrichtungen bewegen.

Ausgehend von den Koordinaten $x = 0$ und $y = 0$ soll zu jeder Befehlsfolge die erreichte Endposition berechnet werden.

z.B. NNNOO x=2, y=3 oder SSW x=-1, y=-2

Grammatik:

Folge	$\rightarrow$	Folge Schritt
Folge	$\rightarrow$	ε
Schritt	$\rightarrow$	N
Schritt	$\rightarrow$	O
Schritt	$\rightarrow$	S
Schritt	$\rightarrow$	W

Attribute:

Folge.↑ex	x-Koordinate der Endposition
Folge.↑ey	y-Koordinate der Endposition
Schritt.↑deltax	relative Veränderung der x-Koordinate
Schritt.↑deltay	relative Veränderung der y-Koordinate

attributierte Grammatik:

Syntaxregeln			**Attributberechnungsregeln**
Folge	→	Folge Schritt	Folge_1.↑ex:=Folge_2.↑ex+Schritt.↑deltax; Folge_1.↑ey:=Folge_2.↑ey+Schritt.↑deltay;
Folge	→	ε	Folge.↑ex:=0; Folge.↑ey:=0;
Schritt	→	N	Schritt.↑deltax:= 0; Schritt.↑deltay:= 1;
Schritt	→	O	Schritt.↑deltax:= 1; Schritt.↑deltay:= 0;
Schritt	→	S	Schritt.↑deltax:= 0; Schritt.↑deltay:= -1;
Schritt	→	W	Schritt.↑deltax:= -1; Schritt.↑deltay:= 0;

5.4 Semantische Aktionen und Übersetzungschemata

Ein **Übersetzungsschema** ist eine Grammatik, in deren Regeln auf der rechten Seite zwischen den Grammatik-Symbolen auch **semantische Aktionen** eingefügt sind, die beliebige Berechnungen spezifizieren.

Die Reihenfolge der Berechnungen ist durch die Stellung der semantischen Aktionen in der Regel eindeutig bestimmt. Dadurch ist eine automatische Generierung eines Parsers, der die Berechnungen durchführt, ohne weiteres möglich.

5.4.1 Ausführung der Aktionen durch Top-Down-Parser

Ein Top-Down-Parser selektiert zunächst eine Ableitungsregel

$$A \to a_1 \dots a_k$$

und bearbeitet dann die Bestandteile der rechten Regelseite $a_1 \dots a_k$ von links nach rechts.

Wenn wir nun neben Terminal- und Nonterminalzeichen Aktionen als reguläre Bestandteile der rechten Regelseite zulassen, heißt *Bearbeitung* einfach: Ausführen der Aktion.

Beim rekursiven Abstieg ist nach der Selektion der Regel die rechte Seite $a_1 \ldots a_n$ von links nach rechts wie folgt zu verarbeiten:

$a_i \in T \Longrightarrow$	**if** (a_i=lookahead-Symbol) **then**
	bestimme nächstes lookahead-Symbol
	else Fehler
	end if

$a_i \in N \Longrightarrow$	$PARSE_{a_i}$ aufrufen
a_i *Aktion* $\Longrightarrow$	a_i ausführen

Beim Tabellen-gesteuerten LL(1)-Parser sind die Aktionen nach der Regelauswahl zunächst wie Terminal- und Nonterminalzeichen auf dem Stack abzulegen. Die Ausführung der Aktion erfolgt, wenn die Aktion als oberstes Stack-Symbol bearbeitet werden muß.

5.4.2 Ein Übersetzer für Ausdrücke vom Infix- in das Postfixformat

Als Beispiel für die Verwendung semantischer Aktionen, soll mit *yacc* ein Ausdrucksübersetzer erzeugt werden. Die zu übersetzenden Ausdrücke sind Integer-Konstanten oder mit zweistelligen Operatoren (+,-,*,/) und Klammern gebildete komplexe Ausdrücke.

Folgende Regeln spezifizieren den Übersetzer:

Postfix(`INTCONST`)	=	`INTCONST`
Postfix(`A+B`)	=	Postfix(`A`) Postfix(`B`) `'+'`
Postfix(`A-B`)	=	Postfix(`A`) Postfix(`B`) `'-'`
Postfix(`A*B`)	=	Postfix(`A`) Postfix(B) `'*'`
Postfix(`A/B`)	=	Postfix(`A`) Postfix(`B`) `'/'`
Postfix(`(A)`)	=	Postfix(A)

Beispiel:

```
  Postfix( (3-4) * 5 )
= Postfix( (3-4) ) Postfix(5) *
= Postfix(  3-4  )  5    *
= Postfix(3) Postfix(4) - 5 *
= 3 4 - 5 *
```

Die yacc-Spezifikation für den Übersetzer zeigt Listing 5.4.2, S. 81. Man beachte, dass bei den Operatoren-Regeln nur jeweils die Ausgabe des Operators selbst, nicht aber der Argumente angegeben ist, z.B.

```
expr: expr '+' expr { printf("%s", " + "); }
```

Für die Übersetzung der Argumente wird nämlich während deren Analyse schon gesorgt. Die Ausgabe des Operators erfolgt erst nach der Analyse

beider Argumente und damit auch nach der Postfix-Code-Generierung für diese Argumente gemäss obiger Übersetzungsregel.

5.4.3 Ausführung der Aktionen durch Bottom-Up-Parser

Beim Bottom-Up-Parser unterscheiden wir zwischen Regelende-Aktionen („end-rule action“) und Regelmitte-Aktionen („mid-rule action“). Eine Regelende-Aktion ist letzter Bestandteil der rechten Regelseite einer Grammatik-Regel, Regelmitte-Aktionen stehen an anderer Stelle.

Behandlung von Regelende-Aktionen:

Eine Regelende-Aktion wird bei der Reduktion der rechten Regelseite ausgeführt.

Behandlung von Regelmitte-Aktionen:

Wenn auf der rechten Regelseite eine Aktion A als Regelmitte-Aktion auftritt:

$$X \to u\{A\}v$$

, will man damit ausdrücken, daß die Aktion nach dem Lesen von u und noch vor der Bearbeitung von v ausgeführt werden soll.

Allerdings berechnet der Bottom-Up-Parser erst unmittelbar vor einem Reduktionsschritt die korrekte Ableitungsregel, dann ist der v zugehörige Teil der Eingabe aber schon gelesen.

Eine Regelmitte-Aktion wird daher in eine Regelende-Aktion transformiert: Ein neues Nonterminalsymbol N_A repräsentiert die Aktion:

$$X \to uN_Av$$

für N_A wird eine neue Regel, nun aber mit Regelende-Aktion, eingeführt:

$$N_A \to \varepsilon\{A\}$$

5.5 Syntax-orientierte Spezifikationstechniken im Vergleich

Attributierte Grammatiken

- nichtdeterministisch, da Zeitpunkt der Attribut-Auswertung nicht angegeben wird
- Auflösung des Nichtdeterminismus, d.h. Bestimmung eines Attribut-Auswertungs-Algorithmus durch:
  1. Analyse der Abhängigkeiten und manuelle Umsetzung in semantische Aktionen und/oder Programm

Listing 5.1 Infix-Postfix-Übersetzer

```
%{
#include <stdio.h>
%}

%left '+' '-'
%left '*' '/'

%%
lines: line | line lines ;
line: expr '\n' { putchar('\n'); } ;
expr: intconst        { putchar(' '); }
      | expr '+' expr { printf("%s", " + "); }
      | expr '-' expr { printf("%s", " - "); }
      | expr '*' expr { printf("%s", " * "); }
      | expr '/' expr { printf("%s", " / "); }
      | '(' expr ')'
;

intconst: digit | digit intconst;
digit:  '0' { putchar('0'); }
      | '1' { putchar('1'); }
      | '2' { putchar('2'); }
      | '3' { putchar('3'); }
      | '4' { putchar('4'); }
      | '5' { putchar('5'); }
      | '6' { putchar('6'); }
      | '7' { putchar('7'); }
      | '8' { putchar('8'); }
      | '9' { putchar('9'); }
;

%%
yyerror(char *message) {
 fprintf(stderr, "parser error: %s", message);
 }
yylex(){int c=getchar(); if(!feof(stdin)) return(c); else exit(0); }
main(){ yyparse(); }
```

2. automatische Abhängigkeitsananalyse, automatische Umsetzung in Programm zur Auswertung (keine zyklischen Abhängigkeiten erlaubt)
3. automatische Umsetzung in Programm ohne Abhängigkeitsanalyse, Auswertungsschema fest durch Parser-Strategie bestimmt $\Longrightarrow$ Einschränkungen bei der Attributierung (häufig nur synthetische Attribute erlaubt, „S-Attributierung“, oder Auswertung bottom-up von links nach rechts, „L-Attributierung“)

Attributauswertung durch yacc

- nach Methode c
- Unterstützt eigentlich nur S-Attributierung. Durch freie Zugriffsmöglichkeit auf den Stack sind in sehr beschränktem Umfang auch L-attributierte Grammatiken verarbeitbar.
  - $$ ist der neue Top-of-Stack-Wert nach reduce-Aktion
  - $i ist der Wert des Stack-Eintrags zur i-ten *handle*-Komponente vor *reduce*-Aktion
  - $0, $-1, $-2, . . . sind die Attribute der darunterliegenden Stackelemente

Ein Beispiel für die Verwendung von L-attributierten Grammatiken:

Folgende Regeln beschreiben die Funktionsaufruf-Syntax einer imaginären Programmiersprache S:

```
FunctionCall :  FunctionIdentifier '(' Parameters ')' ;
Parameters :    Expression
                |  Parameters ',' Expression
                ;
```

Greifen wir uns die Regel

```
Parameters :   Parameters ',' Expression ;
```

heraus und betrachten die Typprüfung: Der Typ des aktuellen Parameters (*Expression*), der sich aus `$3` ergibt, muss mit dem korrespondierenden formalen Parametertyp kompatibel sein. Wie erhält man Informationen zum formalen Parametertyp ?.

Falls *Parameters* ausschliesslich innerhalb von Funktionsaufrufen auftreten, kann man sich behelfen: Bei der Analyse von *Expression* ist der Parserstack nämlich immer wie folgt aufgebaut:

...	FunctionIdentifier	'('	Parameters	','	Expression
...	`$-1`	`$0`	`$1`	`$2`	`$3`

5.5.1 Implementierung des Roboter-Beispiels mit yacc

Bei dem oben diskutierten Roboter-Beispiel ist die Umsetzung mittels yacc einfach, da nur synthetische Attribute verwendet werden.

Die yacc-Eingabedatei `roboter.y`:

```
%{
#include <stdio.h>
#include <unistd.h>

#define YYDEBUG 1
%}

%union {
  struct { int ex, ey; } endpunkt;
  struct { int dx, dy; } delta;
}

%type <endpunkt> kommandos;
%type <delta>    schritt;

%%
start:     kommandos { printf("X=%d, Y=%d\n", $1.ex, $1.ey); }
kommandos: kommandos schritt  { $$.ex=$1.ex+$2.dx;
                                $$.ey=$1.ey+$2.dy; }
         | /* empty */        { $$.ex=$$.ey=0; }
;

schritt:   'o'    { $$.dx= 1; $$.dy= 0; }
         | 's'    { $$.dx= 0; $$.dy=-1; }
         | 'w'    { $$.dx=-1; $$.dy= 0; }
         | 'n'    { $$.dx= 0; $$.dy= 1; }
;

%%
int yyerror(char *msg){fputs(msg,stderr);}
int yylex(){
  int c=getchar();
  if(c=='\n')
    return EOF;

  return c;
}

int main(){
  yydebug=1;
  yyparse();
}
```

Die von *bison* erzeugte Spezifikation des Parsers in `roboter.output` (*yacc*-Format etwas anders):

```
Grammar
rule 1    start -> kommandos
rule 2    kommandos -> kommandos schritt
rule 3    kommandos ->/* empty */
rule 4    schritt -> 'o'
rule 5    schritt -> 's'
rule 6    schritt -> 'w'
rule 7    schritt -> 'n'

Terminals, with rules where they appear

$ (-1)
'n' (110) 7
'o' (111) 4
's' (115) 5
'w' (119) 6
error (256)

Nonterminals, with rules where they appear

start (7)
    on left: 1
kommandos (8)
    on left: 2 3, on right: 1 2
schritt (9)
    on left: 4 5 6 7, on right: 2


state 0
    $default reduce using rule 3 (kommandos)
    start go to state 7
    kommandos go to state 1
state 1
    start  ->  kommandos .   (rule 1)
    kommandos  ->  kommandos . schritt   (rule 2)

    'o'  shift, and go to state 2
    's'  shift, and go to state 3
    'w'  shift, and go to state 4
    'n'  shift, and go to state 5
    $default reduce using rule 1 (start)

    schritt go to state 6
state 2
    schritt  ->  'o' .   (rule 4)
```

```
    $default reduce using rule 4 (schritt)
state 3
    schritt  ->  's' .   (rule 5)

    $default reduce using rule 5 (schritt)
state 4
    schritt  ->  'w' .   (rule 6)

    $default reduce using rule 6 (schritt)
state 5
    schritt  ->  'n' .   (rule 7)

    $default reduce using rule 7 (schritt)
state 6
    kommandos  ->  kommandos schritt .   (rule 2)

    $default reduce using rule 2 (kommandos)
state 7
    $    go to state 8
state 8
    $    go to state 9
state 9
    $default accept
```

Die Interpretation der Zustandsbeschreibungen ergibt sich aus dem in 4.4.1, S. 60 beschriebenen Aufbau eines LR-Parsers.

Wir betrachten dazu Zustand 1 als Beispiel:

Aus den beiden ersten Zeilen

```
start  ->  kommandos .   (rule 1)
kommandos  ->  kommandos . schritt   (rule 2)
```

kann man die „Handles“ entnehmen, die der Parser auf dem Symbolstack aufzubauen versucht, entweder die rechte Regelseite von Regel 1 oder die rechte Seite von Regel 2. Die Symbole links des Punktes (ggf. auch "_") sind schon auf dem Stack, hier also das Nonterminal *kommandos*.

Die Entscheidung für eine der beiden Regeln ist hier durch die Vorausschau auf das nächste Token eindeutig zu treffen:

- Kommt noch ein Schritt hinzu (Token 'o','s','w','n') ist das zweite Handle korrekt
- Ist die Schrittfolge beendet, kann auf *start* reduziert werden.

Die nächsten 5 Zeilen der Zustandsbeschreibung geben die durchzuführende Aktion an, `$default` steht dabei für alle Vorausschau-Symbole:

```
'o'       shift, and go to state 2
's'       shift, and go to state 3
'w'       shift, and go to state 4
'n'       shift, and go to state 5

$default  reduce using rule 1 (start)
```

Die letzte Angabe in der Zustandsbeschreibung enthält keine shift- oder reduce-Aktion, sondern spezifiziert einen vom Symbolstack abhängigen Zustandübergang. In der schematischen Darstellung des LR-Parsers sind diese Übergange als *goto-Tabelle* bezeichnet.

Diese Tabelle wird nach jeder Reduktion benötigt: Bei einer Reduktion einer rechten Regelseite der Länge n werden n Symbole vom Symbolstack und n Zustände vom Zustandsstack heruntergenommen. Oben auf dem Zustandsstack erscheint dadurch ein alter Zustand s.

Nehmen wir an, s repräsentiert $x.y$. Falls inzwischen y durch Reduktion auf dem Stack steht, wird der in der goto-Tabelle angegebene Folgezustand $xy.$ repräsentieren. In unserem Beispiel heißt das für Zustand 1:

```
schritt   go to state 6
```

Falls nach einer Reduktion Zustand 1 oben auf dem Zustandsstack steht, und gleichzeitig das Symbol *schritt* oben auf dem Symbolstack, geht der Parser über in Zustand 6 (`kommandos -> kommandos schritt .`).

Eine Trace-Ausgabe des generierten Parsers zeigt das Listing 5.5.1, S. 87.

5.6 Symboltabelle

Eine Symboltabelle enthält zu jedem Bezeichner einen Eintrag mit allen für die Übersetzung relevanten Attributen. Dieser Eintrag wird bei der Analyse der Definitionsstelle des Bezeichners erzeugt. Bei jeder Verwendungsstelle eines Bezeichners wird der Compiler den Symboltabelleneintrag lesen, um die korrekte Verwendung überprüfen oder den richtigen Code erzeugen zu können.

5.6.1 Symboltabellen-Einträge

Anhand der *Bezeichnerklasse* stellen wir fest, ob Bezeichner Variablen, Typen, Prozeduren oder sonstige benannte Objekte repräsentieren. Ein Bezeichnereintrag muss in jedem Fall den Namen und die Klasse enthalten. Welche weiteren Attribute für die Übersetzung benötigt werden, ist von der Klasse abhängig.

Listing 5.2 Ausgabe des generierten Parsers für die Eingabe "oon"

```
roboter: make
bison -dtv roboter.y
cc -o roboter -g roboter.tab.c
roboter: roboter
Starting parse
Entering state 0
Reducing via rule 3 (line 20),  -> kommandos
state stack now 0
Entering state 1
Reading a token: oon
Next token is 111 ('o')
Shifting token 111 ('o'), Entering state 2
Reducing via rule 4 (line 23), 'o'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Next token is 111 ('o')
Shifting token 111 ('o'), Entering state 2
Reducing via rule 4 (line 23), 'o'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Next token is 110 ('n')
Shifting token 110 ('n'), Entering state 5
Reducing via rule 7 (line 26), 'n'  -> schritt
state stack now 0 1
Entering state 6
Reducing via rule 2 (line 18), kommandos schritt  -> kommandos
state stack now 0
Entering state 1
Reading a token: Now at end of input.
Reducing via rule 1 (line 17), kommandos  -> start
X=2, Y=1
state stack now 0
Entering state 7
Now at end of input.
Shifting token 0 ($), Entering state 8
Now at end of input.
```

Beispiel:

Klasse	**Attribute**
Variable	Typ, Speicherdresse
Funktion	Resultatstyp, Speicheradresse, für jeden Parameter: Typ + ggf. Übergabeverfahren
Konstante	Typ, Wert

Welche Bezeichnerklassen existieren, hängt im wesentlichen von der Quellsprache ab. Die jeweils benötigten Attribute ergeben sich aus den vom Compiler für Typprüfung und Codeerzeugung benötigten Informationen. Man beachte, dass Adressen erst bei der Codeerzeugung vergeben werden, während die anderen Attribute aus der Definition entnommen werden.

5.6.2 Operationen

Der Datentyp Symboltabelle könnte etwa folgende Operationen besitzen:

- neue Tabelle erzeugen: S = Erzeuge_Symtab ()
- Tabelle löschen: Entferne_Symtab (S)
- neuen Eintrag erzeugen:
  Eintrag = Erzeuge_Eintrag(S, IDENT, Klasse)

  Man beachte dass beim Eintragen eines neuen Bezeichners dessen Klasse bekannt ist.
- Eintrag für Bezeichner IDENT in einer Tabelle S suchen:
  Eintrag = Lookup(IDENT, S)
- Für jedes benötigte Bezeichner-Attribut X:
  Setze_*Attribut_X* (Eintrag)
- Für jedes benötigte Bezeichner-Attribut X:
  Attribut = *Attribut_X* (Eintrag)

Nach der Compilierung wird die Symboltabelle i.d.R. nicht mehr benötigt. Ausnahme: Falls eine symbolischer Debugger eingesetzt werden soll, müssen die Bezeichnerinformationen zur Laufzeit weiter verfügbar sein.

5.6.3 Symboltabellen und Gültigkeitsbereiche

In höheren Programmiersprachen sind unterschiedliche Gültigkeitsbereichs-Konzepte verwirklicht. Typischerweise lassen sich globale und Unterprogramm-lokale Definitionen unterscheiden. Der Sprachdefinierer entscheidet, ob eine beliebige Hierarchie von Gültigkeitsbereichen möglich ist (z.B.

verschachtelte Unterprogramm-Definitionen in Pascal oder ADA), oder nur eine flache Struktur (global – lokal).

In jedem Fall muss die Bindung einer Definition an einen Gültigkeitsbereich in der Symboltabelle zum Ausdruck kommen. Konzeptuell ist für jeden Gültigkeitsbereich eine separate Symboltabelle vorzusehen.

Beispiel C:

- pro Modul eine globale Symboltabelle mit den Einträgen für globale Variablen, Typbezeichner und Funktionsbezeichner
- pro Funktionsdefinition ein Symboltabelle mit den lokalen Bezeichnern

Ob die einzelnen Symboltabellen dann auch durch separate Datenstrukturen implementiert werden, oder ob eine einzige Datenstruktur für alle Gültigkeitsbereiche verwendet wird, ist eine Implementierungsentscheidung.

5.6.4 Implementierung

Eine Symboltabelle läßt sich als verkettete Liste von Symboltabellen-Einträgen implementieren. Die Suche eines Bezeichnereintrags ist eine sehr häufige Operation, so dass sich die Implementierung eines effizienten Suchverfahrens anbietet (z.B. Hashverfahren oder sortierter Baum).

6 Laufzeitsysteme

Der Maschinen-Code eines Programms ist oft nicht ohne Unterstützung durch ein Laufzeitsystem möglich, das etwa die dynamische Speicherverwaltung (z.B. C, Pascal) oder Scheduling-Aufgaben in Sprachen mit Parallelisierungskonzepten (z.B. Multitasking in ADA) übernimmt.

Exemplarisch betrachten wir die Speicherverwaltung für Pascal.

6.1 Programmiersprachliche Grundlagen

Die Semantik eines Variablenbezeichners in Pascal ist mathematisch gesehen durch eine zweistufige Abbildung gekennzeichnet:

- Die erste Abbildung, das „Environment“ (Umgebung), ordnet dem Bezeichner einen Speicherplatz zu
- Die zweite Abbildung, der Zustand („state“) , ordnet dem Speicherplatz einen Wert zu

$$Variablenbezeichner \overset{Environment}{\longrightarrow} Speicherplatz \overset{State}{\longrightarrow} Wert$$

Eine Wertzuweisung `x:=5` verändert den Zustand, hat jedoch keinen Einfluß auf das Environment. Allerdings ist in Pascal wie bei vielen anderen Programmiersprachen auch das Environment eines Bezeichners dynamischen Änderungen unterworfen.

6.1.1 Das Environment in Pascal

Die Zuordnung eines Speicherplatzes zu einem Bezeichner in Pascal-Programmen hat sowohl einen statischen als auch einen dynamischen Aspekt.

- Für Pascal gilt das Prinzip der **statischen Bindung**:

  Ein Bezeichner wird einer Deklaration aufgrund der statischen Gültigkeitsbereichsregeln zugeordnet. Hier ist also die Verschachtelung der Unterprogramm-Deklarationen im Programmtext maßgebend.

- Der dynamische Aspekt des Environment zeigt sich bei Unterprogramm-Aufrufen:

  Jeder Unterprogramm-Aufruf definiert ein neues Environment für die lokalen Variablen des Unterprogramms.

  Die Variablendeklaration `i: integer` innerhalb eines rekursiven Unterprogramms führt zu einer neuen Speicherplatzzuordnung für jede Rekursions-Ebene.

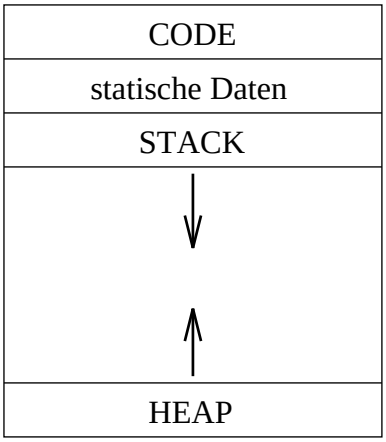
Somit kann die Bestimmung des Werts eines lokalen Variablenbezeichners X nach folgendem Schema erfolgen:

1. Bestimmung des Unterprogramms U, das die Deklaration von X enthält (statisch, durch Compiler)
2. Bestimmung des Speicherplatzes („Bindung“ der Variablen) gemäß der dynamischen Aufruffolge (aktuelle „Inkarnation“ von U bestimmen, durch Laufzeitsystem)
3. Bestimmung des Werts

6.2 Speicherverwaltung

6.2.1 Speicheraufteilungsschema

Typisch für Pascal und C ist folgendes Aufteilungsschema:



Der Stack enthält für jeden noch nicht abgeschlossenen Unterprogramm-Aufruf einen Eintrag, den „activation record“ (auch Aktivierungsrahmen oder „frame“) des Unterprogramm-Aufrufs.

Dieser enthält alle für den Aufruf benötigten Daten bzw. Speicherplätze z.B.

- Rückgabewert
- aktuelle Parameter
- Kontroll-Verweis (Verweis auf vorangehenden Rahmen)
- Zugriffs-Verweis (Verweis auf obersten Rahmen des statisch umfassenden Unterprogramms)
- „save area“ zwischengespeicherter Status der Maschine bei Aufruf

- Speicher für lokale Variablen
- Speicher für Zwischenergebnisse

6.2.2 Speicher-Layout für lokale Daten

Aus dem bekannten Speicherbedarf für die Werte der Basistypen kann der Übersetzer den Speicherbedarf für strukturierte Variablen (z.B. Felder, Records, Varianten) ermitteln.

Bei Bearbeitung eines Deklarationsteils wird jeder Variablen eine relative Adresse (Distanz, „offset“), bezogen auf den Beginn des Speicherbereichs für die lokalen Daten dieses Unterprogramms zugeordnet.

Dabei sind maschinenspezifische Ausrichtungs-Anforderungen („alignment“) zu beachten (z.B. Hauptspeichereinteilung in 4-Byte-Maschinenworte, Wort-Ausrichtung für *real* und *integer*-Werte).

Die Adresse einer lokalen Variablen zur Laufzeit kann dann aus Rahmen-Adresse und Distanz berechnet werden.

Für Strukturen deren Länge erst nach erfolgtem Sprung in das Unterprogramm ermittelt werden kann (z.B. arrays in ALGOL68), ist eine Sonderbehandlung erforderlich:

Anstatt im Rahmen Platz für die Strukturen vorzusehen, wird Platz für einen Verweis auf deren Anfangsadresse reserviert. Die Strukturen werden vom aufgerufenen Unterprogramm, am Ende des Rahmens abgelegt (beliebig viel Platz), der Zugriff erfolgt indirekt über den zugehörigen Verweis.

6.2.3 Aktionen beim Unterprogramm-Aufruf

P_1 ruft P_2 auf.

Aufruf:

- P_1 wertet die aktuellen Parameter aus und speichert sie im Rahmen von P_2 ab
- P_1 trägt die Rücksprung-Adresse in den Rahmen von P_2 ein
- P_1 trägt den alten *top of stack*-Wert in den Rahmen von P_2 ein
- P_1 erhöht den *top of stack*-Wert so, daß er in den Rahmen von P_2 hinter die Rücksprungadresse verweist
- P_2 sichert die Registerwerte in seinen Rahmen
- P_2 initialisiert lokale Daten und beginnt mit der Ausführung des Unterprogramm-Rumpfs

Rücksprung:

- P_2 speichert sein Resultat direkt am Anfang seines Rahmens ab
- P_2 setzt die Register auf die alten Werte zurück
- P_2 setzt den *top of stack*-Zeiger zurück
- P_2 springt zur übergebenen Rücksprung-Adresse
- P_1 findet das Resultat direkt hinter dem eigenen Rahmen

Man beachte, daß dieses Verfahren den Aufruf von Unterprogrammen mit einer variablen Parameter-Anzahl zuläßt.

6.3 Zugriff auf globale Adressen

Bei statischer Bindung ist ein besonderer Mechanismus für den Zugriff auf globale Adressen im Stack notwendig.

Sicherlich muß beim Zugriff auf eine solche globale Variable ein Unterprogramm-Rahmen existieren, in dem der Wert gespeichert ist. Bei rekursiven Unterprogrammen können beliebig viele solcher Rahmen auf dem Stack stehen, der letzte ist der richtige.

Beispiel:

```
program demo;                          Ebene 0
  procedure p1;                        Ebene 1
    var X: real;
    procedure p11;                     Ebene 2
      begin X:=5.0 end;
    procedure p12;                     Ebene 2
      var X: real;
      begin p11 end;
    begin
    ...if ...then p1 else p12; ...
    end;
  begin
    ...p1; ...
  end.
```

7 Code-Erzeugung

7.1 Einführung in die Maschinencode-Generierung und Optimierung

7.1.1 Abstrakte Maschinen und Zwischencode-Generierung

Ein Ein-Phasen-Compiler wird den Code direkt während der Analyse erzeugen. Bei einem Frontend-Backend-Compiler erzeugt das Frontend zunächst einen Syntaxbaum. Der Baum ist dann die Basis für die Code-Erzeugung durch das Backend.

Unabhängig von der Phaseneinteilung wird bei der Code-Erzeugung in vielen Fällen nicht gleich Maschinencode sondern zunächst ein Zwischencode erzeugt. Dieser basiert in der Regel auf den Datenstrukturen und dem Befehlssatz einer abstrakten virtuellen Maschine

Die Vorteile:

- Optimierungen auf hoher Abstraktionsebene möglich
- bei Portierung auf andere Maschine ist nur das maschinenanhängige „back end“ neu zu schreiben
- bei Implementierung ähnlicher Sprachen kann evtl. das gleiche „back end“ verwendet werden (z.B. für C und Pascal)
- bei einer Prototyp-Implementierung einer neuen Sprache kann man den Zwischencode zunächst interpretieren (bei logischen und funktionalen Sprachen wird oft auf Maschinencode-Generierung ganz verzichtet)

Bekannte Zwischenrepräsentationen sind:

- 3-Adreß-Code (prozedurale Sprachen)
- SECD-Maschine (funktionale Sprachen)
- „Warren“-Maschine (logische Sprachen)

7.1.2 Code-Erzeugung für Stack-Maschinen

Als Vorüberlegung betrachten wir das Prinzip der Code-Erzeugung für Stack-Maschinen anhand einiger Beispiele. Eine Stack-Maschine benutzt

für die Speicherung von Zwischenergebnissen einen Stack (anstelle von Registern). Wir gehen der Einfachheit halber von einem Wort-adressierten Hauptspeicher aus.

Wir verwenden für unsere Beispiele folgende Befehle:

Befehl	**Wirkung**	**Seiteneffekt**
PUSH w	Wert w wird auf dem Stack oben angefügt	`TOP++`
LOAD a	Lade Inhalt des Speicherworts a auf den Stack	`TOP++`
STORE a	Lege den oben auf dem Stack stehenden Wert in Speicherwort a ab	`TOP--`
ADD	Ersetze die beiden oben stehenden Werte durch ihre Summe	`TOP--`
SUB	Ersetze die beiden oben stehenden Werte durch ihre Differenz	`TOP--`
MUL	Ersetze die beiden oben stehenden Werte durch ihre Produkt	`TOP--`

Betrachten wir nun einige Code-Beispiele dazu:

Quelltext	**Stackmaschinencode**
`5`	`PUSH 5`

`5+3`	`PUSH 5`
	`PUSH 3`
	`ADD`

`j-2*i`	`LOAD Adr(j)`
	`PUSH 2`
	`LOAD Adr(i)`
	`MULT`
	`SUB`

`k:=2*i`	`PUSH 2`
	`LOAD Adr(i)`
	`MULT`
	`STORE Adr(k)`

Wir können die Codeerzeugung sehr einfach syntaxorientiert formulieren. Wir verwenden dabei ein Semikolon für das *Aneinanderhängen* von Maschinencodesequenzen. Die dabei verwendeten Attribute sollten selbsterklärend sein.

Beispiele:

CODE(`IntConst`)	=	`PUSH` *IntConst.Wert*
CODE(`VarIdent`)	=	`LOAD` *VarIdent.Adresse*
CODE(`Expr1+Expr2`)	=	CODE(`Expr1`) ; CODE(`Expr2`) ; `ADD`
CODE(`Expr1-Expr2`)	=	CODE(`Expr1`) ; CODE(`Expr2`) ; `SUB`
CODE(`VarIdent:=Expr`)	=	CODE(`Expr`) ; `STORE` *VarIdent.Adresse*

Die im nächsten Abschnitt beschriebene Rechnerarchitektur ist keine Stackmaschine, sondern ein virtueller RISC-Rechner mit 32 Mehrzweckregistern.

Die Codererzeugung benutzt dabei das oben erläuterte Prinzip. Dazu wird einfach der vorhandene Registersatz als Stack verwendet, der Stackpointer TOP verweist auf das letzte belegte Register.

Für den Fall, dass die Register nicht ausreichen, werden wir später eine geeignete Lösung suchen. Im Prinzip müssen bei Platzmangel die Registerinhalte vorübergehend in den Hauptspeicher kopiert werden.

7.1.3 Zielrechner-Architektur

Als Zielrechner betrachten wir die in [Wirth] definierte virtuelle RISC-Maschine. Dieser Rechner hat 32 Mehrzweck-Register (R0,...,R31) von 32 Bit Breite. R0 ist immer gleich 0. R31 wird von Unterprogramm-Einsprungbefehlen implizit mit der Rücksprungadresse geladen (Link-Register). Alle anderen Register sind frei verfügbar.

Das Instruktionsregister IR enthält den jeweils aktuellen Maschinenbefehl. Der Programmzähler PC enthält die Adresse des nächsten auszuführenden Befehls.

Der Hauptspeicher besteht aus 4-Byte-Wörtern und wird Byte-adressiert (Speicheradressen sind also immer Vielfache von 4).

Die Maschine hat die folgenden 3 Befehlsformate:

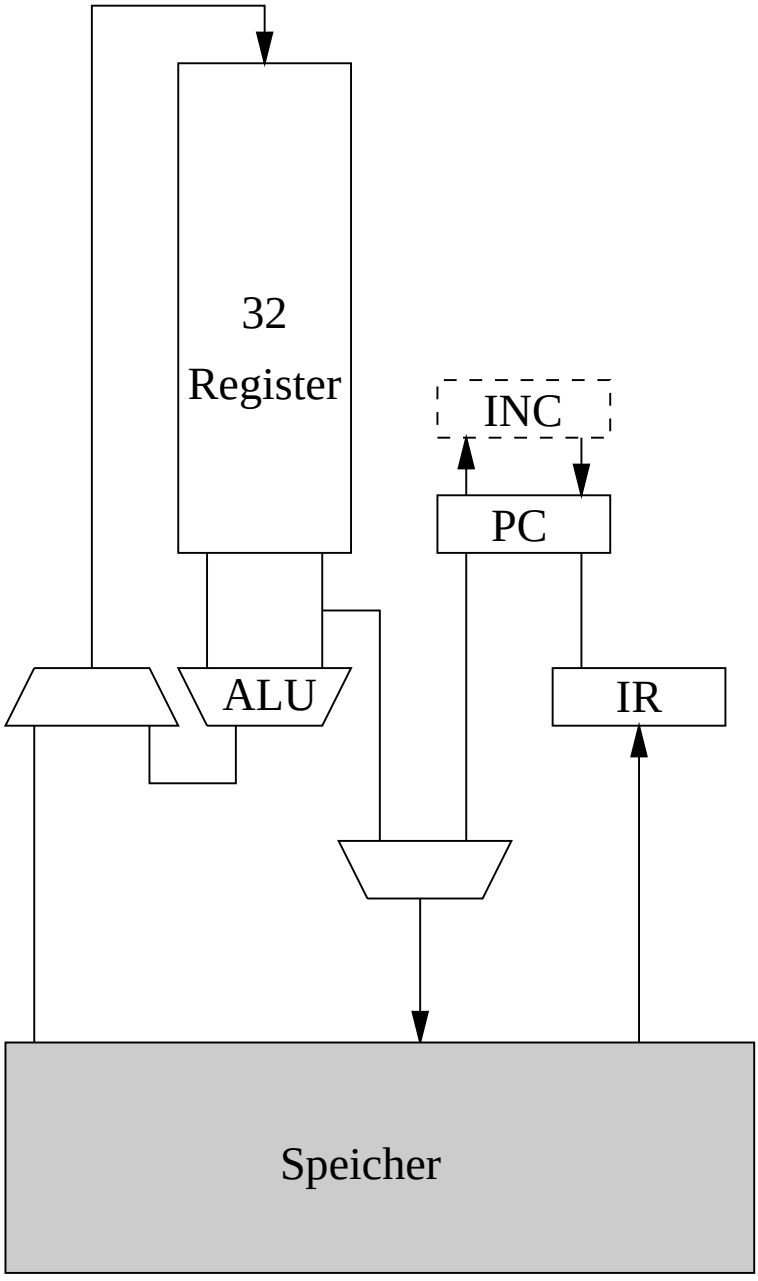
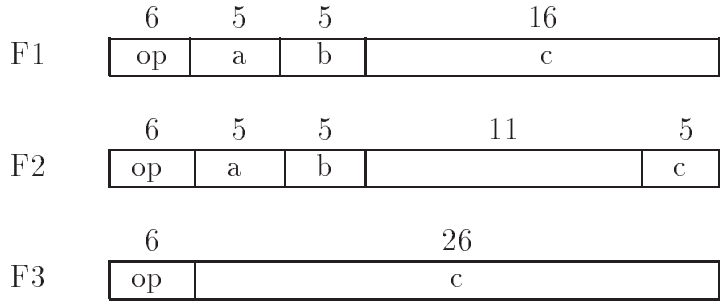


Abbildung 1: Architekturschema der virtuellen Maschine



Registerbefehle

Register + Register (Format F2)			**Register + Direktoperand** (Format F1)		
ADD	a,b,c	R.a:=R.b+R.c	ADDI	a,b,d	R.a:=R.b+c
SUB	a,b,c	R.a:=R.b-R.c	SUBI	a,b,d	R.a:=R.b-c
MUL	a,b,c	R.a:=R.b*R.c	MULI	a,b,d	R.a:=R.b*c
DIV	a,b,c	R.a:=R.b DIV R.c	DIVI	a,b,d	R.a:=R.b DIV c
MOD	a,b,c	R.a:=R.b MOD R.c	MODI	a,b,d	R.a:=R.b MOD c
CMP	a,b,c	R.a:=R.b-R.c	CMPI	a,b,d	R.a:=R.b-c
CHK	a,c	$0 \leq$ R.a $<$ R.c	CHKI	a,d	$0 \leq$ R.a $<$ c
AND	a,b,c	R.a:=R.b & R.c	ANDI	a,b,d	R.a:=R.b & c
BIC	a,b,c	R.a:=R.b & ~R.c	BICI	a,b,d	R.a:=R.b & ~c
OR	a,b,c	R.a:=R.b OR R.c	ORI	a,b,d	R.a:=R.b OR c
XOR	a,b,c	R.a:=R.b XOR R.c	XORI	a,b,d	R.a:=R.b XOR c
LSH	a,b,c	R.a:=LSH(R.b,R.c)	LSHI	a,b,d	R.a:=LSH(R.b,c)
ASH	a,b,c	R.a:=ASH(R.b,R.c)	ASHI	a,b,d	R.a:=ASH(R.b,c)

Anmerkung

- Der 16-Bit Direktoperand (Format F1) ist mit d bezeichnet, in der Semantikspezifikation steht stattdessen c. c ist die auf 32 Bit erweiterte Darstellung von d.
- CMP und SUB verhalten sich bei Überlauf unterschiedlich: SUB zeigt Überlauf an, CMP nicht.
- LSH steht für *Logical Shift*, ASH für *Arithmetic Shift*. Positive Werte für R.c (bei ASHI und LSHI c) bedeuten eine Verschiebung nach links, negative Werte eine Verschiebung nach rechts.

Speicherbefehle (F1-Format)

LDW	a,b,c	R.a := MEM[R.b+c]	load word
LDB	a,b,c	R.a := MEM[R.b+c]	load byte
STW	a,b,c	MEM[R.b+c] := R.a	store word
STB	a,b,c	MEM[R.b+c] := R.a	store byte
PSH	a,b,c	R.b := R.b-c; MEM[R.b]:=R.a	push stack
POP	a,b,c	R.a := MEM[R.b]; R.b:=R.b+c	pop stack

Sprungbefehle (F1-Format, Adresse PC-relativ)

BEQ	a,c	Branch to c, if R.a = 0
BNE	a,c	Branch to c, if R.a != 0
BLT	a,c	Branch to c, if R.a $<$ 0
BGE	a,c	Branch to c, if R.a $\geq$ 0
BGT	a,c	Branch to c, if R.a $>$ 0
BLE	a,c	Branch to c, if R.a $\leq$ 0

BSR	c	Save PC in R.31, branch to c (F1, Adresse PC-relativ)
JSR	c	Save PC in R.31, jump to c (F3, Adresse absolut)
RET	c	Jump to address in R.c (F2, Adresse absolut)

7.1.4 Grundlagen der Codeerzeugung für die virtuelle RISC-Maschine

Wie oben erläutert, wird das Stackmaschinen-Prinzip verwendet, wobei die Register R1 bis R30 den Stack bilden. Der Compiler muss sich nur das letzte belegte Register *r* merken.

Statt des Stackmaschinenbefehls `ADD`, wird jetzt der RISC-Maschinenbefehl

```
ADD r-1,r-1,r
```

verwendet:

- der zweite Operand der Addition steht ganz oben auf dem Stack. d.h. im Register *r*
- der zweite Operand der Addition steht auf dem Stack unmittelbar darunter, also im Register *r-1*
- beide Operanden werden vom Stack entfernt – Register *r* und *r-1* werden frei – die Summe auf dem Stack gespeichert, im Register *r-1*, dem ersten freien Register

Gegenüberstellung

Quelltext	Stackmaschinencode	RISC-Code	Seiteneffekt
			(nach Befehlsausgabe)
5	PUSH 5	ADDI r+1,0,5	r++
5+3	PUSH 5	ADDI r+1,0,5	r++
	PUSH 3	ADDI r+1,0,3	r++
	ADD	ADD r-1,r-1,r	r--
j-2*i	LOAD Adr(j)	LDW r+1,0, Adr(j)	r++
	PUSH 2	ADDI r+1,0,2	r++
	LOAD Adr(i)	LDW r+1,0, Adr(i)	r++
	MULT	MUL r-1,r-1,r	r--
	SUB	SUB r-1,r-1,r	r--
k:=2*i	PUSH 2	ADDI r+1,0,2	r++
	LOAD Adr(i)	LDW r+1,0, Adr(i)	r++
	MULT	MUL r-1,r-1,r	r--
	STORE Adr(k)	STW r,Adr(k)	r--

Falls R1 das erste freie Register ist, ergibt sich also zur Wertzuweisung

```
k := j-2*i
```

folgender Code:

Befehl	Wirkung	Stack
LDW R1,0,Adr(j)	R1 := j	j
ADDI R2,0,2	R2 := 2	j,2
LDW R3,0,Adr(i)	R3 := i	j,2,i
MUL R2,R2,R3	R2 := R2 * R3	j, 2*i
SUB R1,R1,R2	R1 := R1 - R2	j - 2*i
STW R1,Adr(k)	k := R1	--

7.1.5 Erste Optimierungen – Verzögerte Code-Erzeugung

Das oben verwendete Schema berücksichtigt bei den Operatoren nicht die Adressierung der Operanden. Schon deshalb ist der Code nicht sonderlich effizient. Der Ausdruck

```
2*i
```

wird übersetzt in

```
ADDI R2,0,2
```

```
LDW  R3,0,Adr(i)
MUL  R2,R2,R3
```

Effizienter wäre dagegen

```
LDW  R2,0,Adr(i)
MULI R2,R2,2
```

Dazu ist eine Fallunterscheidung notwendig:

- Operand ist Konstante
- Operand ist Variable
- Operand ist Zwischenergebnis

Im ersten Fall kann dann für die Operation ein *Immediate*-Befehl (Befehl mit Direktoperand) erzeugt werden. Falls beide Operanden Konstanten sind, kann der Compiler natürlich den Wert sofort bestimmen.

Wie wird die für die Fallunterscheidung notwendige Information berechnet ?

Wir benötigen für jeden Ausdruck ein Attribut *ort*, das bei der Code-Erzeugung Aufschluss über die Speicherung des Wert gibt:

- *ort.modus* gibt die Adressierungsart an:
  - direkt (Konstante)
  - Hauptspeicher (Variable)
  - Register (Zwischenwert)
- *ort.wert* enthält den Wert, falls Modus=direkt
- *ort.adresse* enthält die Speicheradresse, falls Modus=Hauptspeicher
- *ort.register* enthält die Registernummer, falls Modus=Register

Mit dieser Information kann man die Code-Erzeugung für Konstantenwerte verzögern bzw. ganz unterdrücken, wenn bessere Lösungen existieren. Die Verzögerung besteht hier darin, dass nicht sofort beim Auftreten einer Konstante k die Maschinenanweisung

```
ADDI r,0,k
```

erzeugt wird, sondern erst bei der Anwendung eines Operators, abhängig von den *ort*-Attributen der Operanden über den zu erzeugenden Code entschieden wird.

Beispiele:

Wir betrachten die Subtraktion A - B. Jede Kombination A.*ort.modus*, B.*ort.modus* wird gesondert betrachtet und möglichst effizient behandelt. Einige Kombinationen:

A.*ort*	B.*ort*	**Behandlung**	**Resultat**
direkt *w1*	direkt *w2*	keine Code-Erzeugung, Compiler berechnet Wert	direkt *w1+w2*
register X	register Y	SUB-Befehl erzeugen (SUB X,X,Y), Y freigeben	Register X
register X	direkt *w*	SUBI-Befehl erzeugen (SUBI(X,X,*w*))	Register X
register X	Speicher *a*	Register Y für B reservieren, B laden (LDW Y,0,*a*), SUB-Befehl (SUB X,X,Y), Y freigeben	Register X
Speicher *a*	register X	Register Y für A reservieren, A laden (LDW Y,0,*a*), SUB-Befehl (SUB Y,Y,X), X freigeben	Register Y
direkt *w*	register X	Register Y für A reservieren, A laden (ADDI Y,Y,w), SUB-Befehl (SUB Y,Y,X), X freigeben	Register Y

Man beachte insbesondere, dass die Buchführung über den Ort, an dem der Wert gespeichert ist, eine andere Registerverwaltung ermöglicht:

Statt die Register strikt nach dem Stackprinzip zu verwalten, kann bei Bedarf ein beliebiges freies Register reserviert werden.

8 Literatur

- Das Standardwerk zum Compilerbau („Drachenbuch“):

  Aho, Sethi, Ullmann:
  Compilers
  Principles, Techniques, and Tools
  Addison-Wesley, 1986

  (Deutsche Ausgabe in 2 Bänden: Compilerbau)

- Wirth:
  Grundlagen und Techniken des Compilerbaus
  Addison-Wesley, 1996

A Grammatik für Oberon-0 in *yacc*-Eingabeformat

```
%start Module

%token IDENT INTEGER MODULE TYPE CONST PROCEDURE BEGIN VAR \
       RECORD OF ARRAY DO WHILE ELSE ELSIF END THEN IF
%nonassoc '=' LEQ '<' GEQ '>' '#'
%left '+' '-' OR
%left DIV MOD '&' '*'
%right '~'
%right UMINUS

%%
Number : INTEGER ;

Expression : number
             | IDENT
             | Expression '[' Expression ']'
```

```
              | Expression '.'
              | '(' Expression ')'
              | '~' Expression
              | '-' Expression            %prec UMINUS
              | Expression '*' Expression
              | Expression DIV Expression
              | Expression MOD Expression
              | Expression '&' Expression
              | Expression '+' Expression
              | Expression '-' Expression
              | Expression OR  Expression
              | Expression '=' Expression
              | Expression LEQ Expression
              | Expression '<' Expression
              | Expression GEQ Expression
              | Expression '>' Expression
              | Expression '#' Expression
;

Assignment : Expression ':=' Expression ;

ActualParameters : '(' ExpressionList ')' ;

ExpressionList : Expression ExpressionList
                 | /* EMPTY */
;

ProcedureCall : Expression [ActualParameters] ;

IfStatement : IF Expression THEN StatementSequenc
              ElseifList
              Optelse
              END
;

ElseifList : ELSIF Expression THEN StatementSequence ElseifList
             | /* EMPTY */
;

OptElse : ELSE StatementSequence
          | /* EMPTY */
;

WhileStatement : WHILE Expression DO StatementSequence END ;

Statement : Assignment | ProcedureCall | IfStatement | WhileStatement ;

StatementSequence : Statement StatementSequence
                    | Statement
```

```
;

IdentList : IDENT
          | IDENT ',' IdentList
;

ArrayType : ARRAY Expression OF Type ;

FieldList : IdentList ':' type
          | /* EMPTY */
;

Fielddecls : FieldList ';' Fielddecls
           | FieldList
;

RecordType : RECORD FieldDecls END ;

Type : IDENT | ArrayType | RecordType ;

OptVAR : VAR
       | /* EMPTY */
;

FPSection : OptVAR IdentList ':' Type ;

OptFPDecls :  FPDecls |   /* EMPTY */ ;

FPDecls :     FPSection ';' FPDecls
          | FPSection
;

FormalParameters : '(' OptFPDecls ')'
                 |  /* EMPTY */
;

OptBody :  BEGIN StatementSequence
         |  /* EMPTY */
;

ProcedureDeclaration : PROCEDURE IDENT FormalParameters
                       declarations
                       OptBody
                       END IDENT
;

Constdecl :       IDENT '=' Expression ';'
Constdecls :      Constdecl Constdecls |
                  /* EMPTY */
```

```
;
Consts : CONST Constdecls

Typedecl :       IDENT '=' Type ';'
Typedecls :      Typedecl Typedecls |
                  /* EMPTY */
;
Types : TYPE Typedecls

Vardecl :       IdentList ':' Type ';'
Vardecls :      Vardecl Vardecls |
                  /* EMPTY */
;
Vars : VAR Vardecls

Procedures : ProcedureDeclaration ';' Procedures
             | /* EMPTY */
;

Declarations : Consts
               Types
               Vars
               Procedures
;

Module : MODULE IDENT ';'
         Declaration ;
         OptBody
         END IDENT
         '.'
;
```