Compilerbau

Skriptum zur Vorlesung

Studiengang Informatik
Prof. Dr. Winfried Bantel

Sommersemester 2013
Stand 21. März 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

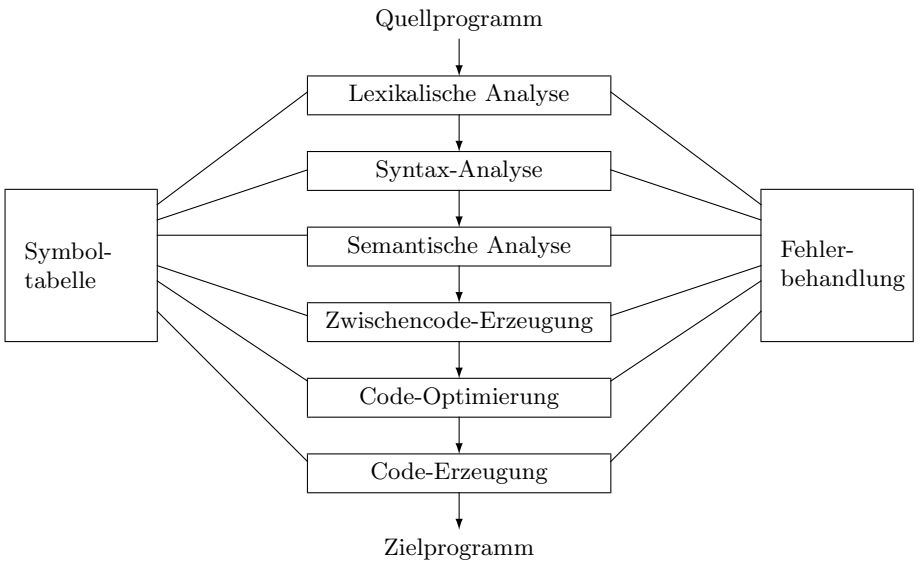
Listings

1

Einführung

Abbildung 1.1 zeigt die verschiedenen Phasen eines Compilers nach [ASU88].

Abb. 1.1. Phasen eines Compilers



Compilerbau ist das schwierigste Fachgebiet in der Informatik, da extrem viel Theorie verstanden werden muss!

Compilerbau ist das einfachste Gebiet der Informatik, da nur hier Programme entwickelt werden können, die zu einen Problem eine Lösung automatisch programmieren!

1.1 Der Komplex "Formale Sprachen - Automatentheorie - Compilerbau"

Definition 1.1. *Abgrenzung der drei Teilgebiete*

Formale Sprachen ist die Wissenschaft, Sprachen zu beschreiben.

Automatentheorie ist die Wissenschaft, die Syntax von Texten, die in einer formalen Sprache geschrieben sind, zu überprüfen.

Compilerbau ist die Wissenschaft, Automaten zu codieren und um ein Übersetzungsmodul zu erweitern.

1.2 Beispiele für Compilerbau

- Internet-Benutzereingaben
- Rechen-Programme
- Syntax-Highlightning von Editoren
- Verarbeitung von ini-Dateien
- SQL-Interpreter
- Verarbeitung von Eingabedaten
- Web-Browser
- XML-Verarbeitung

1.3 Begriffsbildung

Definition 1.2 (Compiler). *Ein Compiler übersetzt ein in einer formalen Sprache geschriebenes Programm in eine andere formale Sprache.*

Definition 1.3 (Interpreter). *Ein Interpreter führt ein in einer formalen Sprache geschriebenes Programm aus.*

Mischformen sind möglich, populärstes Beispiel ist Java.

2

Die Programmiersprache PL/0

Als Programiersprache, welche sich konsequent durch die Beispiele zieht, verwenden wir PL/0. Hierbei handelt es sich um ein Mini-Mini-Pascal, welches Wirth in [Wir86] vorschlägt.

2.1 Die Syntax

Grammatik Γ_1 zeigt die Grammatik der Programmiersprache PL/0, wie Wirth sie in [Wir86] einführt.

Grammatik Γ_1: PL/0

```
program  ::= block "."

block  ::= [ "CONST" ident "=" number {"," ident "=" number} ";"]
                  [ "VAR" ident {"," ident} ";"]
                  { "PROCEDURE" ident ";" block ";" } statement

statement  ::= [ ident ":=" expression
    | "CALL" ident
    | "?" ident
    | "!" expression
    | "BEGIN" statement {";" statement } "END"
    | "IF" condition "THEN" statement
    | "WHILE" condition "DO" statement ]

condition  ::= "ODD" expression
    | expression ("="|"#"|"<"|"<="|">"|">=") expression

expression  ::= [ "+"|"-"] term { ("+"|"-") term}

term    ::=     factor {("*"|"/") factor}

factor  ::= ident
    | number
    | "(" expression ")"
```

Da die Ausdrücke keine beliebigen Vorzeichen zulassen behalten wir uns vor, die Ausdrücke gemäß unserer Term-Grammatik zu implementieren, diese ist abwärtskompatibel zu PL/0-Ausdrücken.

PL/0 bietet wichtige Grundlagen von Programmiersprachen, jedoch fehlen auch wichtige Dinge:

- Datentypen, PL/0 bietet nur int-Daten
- Strings
- Parameter und Funktionswerte für Funktionen
- Felder
- Strukturen
- Zeiger
- Dynamische Hauptspeicherallokierung

Am Ende der Vorlesung werden wir einen kompletten PL/0-Compiler bzw. -Interpreter entwickelt haben.

2.2 Programmbeispiele

Listing 2.1. Rekursive Fakultät in PL/0 (pl-0/programme/fakultaet.pl0)

```
1  (* Rekursive Berechnung der Fakultät *)
2  VAR n, f;
3  PROCEDURE fakultaet;
4  BEGIN
5          ! n;
6          IF n > 0 THEN
7          BEGIN
8                  DEBUG;
9                  f := f * n;
10                 n := n - 1;
11                 CALL fakultaet;
12                 n := n + 1;
13                 DEBUG;
14         END;
15 END;
16 BEGIN
17     ? n;
18     f := 1;
19         DEBUG;
20     CALL fakultaet;
21         DEBUG;
22     ! f;
23 END.
```

Listing 2.2. Größter gemeinsamer Teiler (pl-0/programme/ggt.pl0)

```
1  (* Berechnet GGT von zwei Zahlen *)
2  VAR a, b, g;
3  PROCEDURE ggt;
4  BEGIN
5      WHILE a # b DO
6      BEGIN
7          IF a > b THEN a := a - b;
```

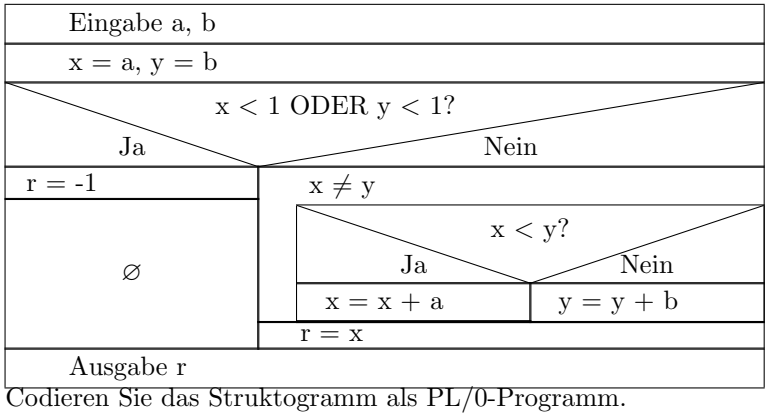
```
 8            IF b > a THEN b := b - a;
 9        END;
10        g := a;
11    END;
12    BEGIN (* Hauptprogramm *)
13        ? a;
14        ? b;
15        CALL ggt;
16        ! g;
17    END.
```

2.3 Übungen

Übung 2.1. Kleinstes gemeinsames Vielfaches

Algorithmus 1 berechnet das kleinste gemeinsame Vielfache (KGV) zweier ganzer Zahlen:

Algorithmus 1: Kleinstes gemeinsames Vielfaches



Codieren Sie das Struktogramm als PL/0-Programm.

Übung 2.2. Teilersuche

Entwickeln Sie ein PL/0-Programm, das alle Teiler einer einzugebenden Zahl errechnet und ausgibt.

3

Wiederholung “Automatentheorie und Formale Sprachen“

Betrachtet man Abbildung 1.1 (Seite 13), so findet man weit vorne die Phasen der lexikalischen sowie der syntaktischen Analyse. Für diese zwei Phasen existieren Theorien, nämlich die der Formalen Sprachen und der Automaten. Aus diesem Grund soll hier nochmals kurz eine Wiederholung durchgeführt werden.

3.1 Formale Sprachen

3.1.1 Einführung

Die erste wissenschaftlich ernsthafte Auseinandersetzung mit Grammatik und Sprachen stammt von dem Amerikaner Noam Chomsky. Er analysierte Grammatiken und teilte diese in die sog. Chomsky-Klassen ein.

Die Theorie der Formalen Sprachen wie sie heute betrieben wird (und damit auch Automatentheorie) basiert auf den Chomsky-Klassen.

Typ 3 Beim Chomsky-Typ 3 handelt es sich um die sog. regulären Sprachen

Typ 2 Beim Chomsky-Typ 1 handelt es sich um die sog. kontextfreien Sprachen.

Typ 1 Kontextsensitive Sprachen (für Informatik nur von untergeordneter Bedeutung)

Typ 0 Allgemeine Sprachen.

Eine Sprache eines Bestimmten Typs n ist immer auch eine Sprache des Typs (n+1), d.h. die Sprachen eines Typs n stellen eine Teilmenge der Sprachen eines Typs (n+1) dar:

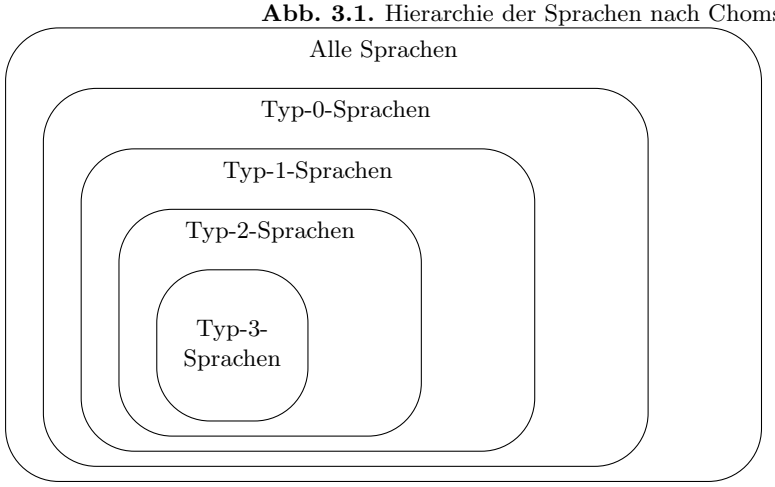
$$T_3 \subset T_2 \subset T_1 \subset T_0 \subset \text{Menge aller Sprachen}$$

Diese Mengenbeziehung ist in Abbildung 3.1 dargestellt.

Für die Theoretische Informatik sind die Chomsky-Typen 2 und 3 interessant, da für diese zwei Klassen effiziente Algorithmen entwickelt wurden, die die Sprache analysieren. Im Gegensatz dazu sind für die Klassen Chomsky-0 und Chomsky-1 keine effizienten Algorithmen bekannt. Die gängigen Programmiersprachen wie C Pascal, Delphi, C++ aber auch arithmetische Ausdrücke in der Mathematik sind Sprachen des Chomsky-Typs 2.

Eine Sprache besteht aus einem Alphabet, aus dem sich die Sätze der Sprache bilden lassen. Zusätzlich bestehen sog. Produktionsregeln, die die Bildung der Sätze zeigen.

Abb. 3.1. Hierarchie der Sprachen nach Chomsky



$A = 'A', 'B', 'C', \ldots, 'X', 'Y', 'Z'$	Alphabet für engl. Sprache
$B = '0', '1'$	Alphabet für Binärzahlen
$C = '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'$	Alphabet für ganze Zahlen
$D = '+', '-', '0', '1', '2', \ldots, '9'$	Alphabet für ganze Zahlen mit Vorzeichen
$E = '+', '-', '.', '0', '1', '2', \ldots, '9'$	Alphabet für Fixkommazahlen
$F = '+', '-', '.', 'E', '0', '1', '2', \ldots, '9'$	Alphabet für Fließkommazahlen
$G = 'if', 'else', 'for', 'while', 'do', 'int', 'float', \ldots$	Alphabet für einen C-Compiler
$H = 'zahl', '+', '-', '*', '/', '(', ')'$	Alphabet für arithmetische Ausdrücke

Formal besteht eine Sprache aus einem 4-Tupel

Definition 3.1. *T Die Menge der terminalen Symbole (engl. “terminals“)*

N Die Menge der nicht-terminalen Symbole (engl. “nonterminals“)

P Die Menge der Produktionsregeln

S Dem Startsymbol $\in N$

- Terminale Symbole sind die Symbole, die nicht mehr weiter zerlegt werden.
- Nicht-terminale Symbole sind die Symbole, die durch eine Produktionsregel zerlegt werden.

Als erstes Beispiel verwenden wir ein sehr einfaches Deutsch, dessen Sätze folgendermaßen gebildet werden:

- Ein Satz besteht aus einer Nominalphrase und einer Verbalphrase
- Eine Nominalphrase besteht aus einem Nomen, vor dem evtl. ein Artikel steht.
- Eine Verbalphrase besteht aus einem Verb, evtl. gefolgt von einer Nominalphrase.

Die terminalen Symbole sind jetzt “V“ (für Verb), “N“ (für Nomen) und “A“ (für Artikel). Die nicht-terminalen Symbole sind “S“ Formell ist dies Grammatik in Γ_2 dargestellt.

Grammatik Γ_2: Einfach-Deutsch

T :	$\{V, N, A\}$
N :	$\{S, NP, VP\}$
P :	$S \to NP \quad VP$
	$NP \to N$
	$NP \to A \quad N$
	$VP \to V$
	$VP \to V \quad NP$
S :	S

Bei der terminalen Symbolen steht N steht für “Nomen“, A für “Artikel“, V für “Verb“. Sätze wie “Die Katze schnurrt“ (Terminal-Folge A-N-V-A-n) oder “Der Hund jagt die Katze“ (Terminal-Folge A-N-V) lassen sich nach diesen Grammatikregeln bilden.

Eine übersichtliche Darstellung für Sätze ist der Syntaxbaum. Ausgehend vom Startsymbol ganz oben werden alle angewendeten Produktionsregeln durch Verzweigungen nach unten dargestellt. Abbildungen 3.2 und 3.3 zeigen die Syntaxbäume der obigen Beispielsätze.

Abb. 3.2. Syntaxbaum “Die Katze schnurrt“

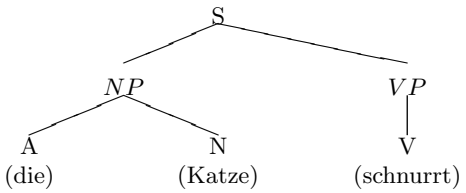
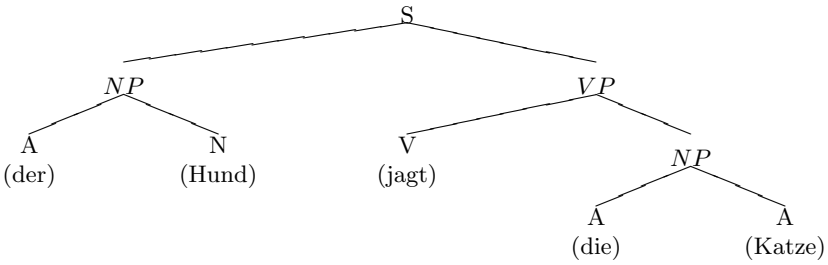


Abb. 3.3. Syntaxbaum “Der Hund jagt die Katze“



Sätze wie “Der Hund jagt die Katze auf einen Baum“ sind nach Grammatik Γ_2 nicht korrekt.

Ziel der Theorie der formalen Sprachen und des Compilerbaus ist es, vorhandene Sätze anhand der Grammatikregeln zu analysieren und den Syntaxbaum aufzustellen.

3.1.2 Notationsformen von Grammatiken

Die von Chomsky vorgeschlagene Notationsform für Grammatiken war für die Informatik nur bedingt geeignet. Im Zuge der Entwicklung der Programmiersprachen wurden geschicktere Notationsformen entwickelt, die teilweise auch maschinell verarbeitbar waren.

Backus-Naur-Form (BNF)

Die Backus-Naur-Form ([1] und [2]) ist eine der wichtigsten Beschreibungsformen für Programmiersprachen. Sie basiert auf Chomskys Notationsform, verwendet aber unterschiedliche Notation für terminale und nicht-terminale Symbole. Des weiteren werden zur Darstellung nur Zeichen des ASCII-Zeichensatzes benötigt.

- Linke und rechte Seite der Produktionen werden durch ’::=’ getrennt
- Nonterminals stehen in spitzen Klammern (’<’,’>’)
- Alternativen werden durch ’|’ getrennt
- Rekursion ist zulässig

Als Beispiel soll die IF-ELSE-Anweisung in Pascal in EBNF beschrieben werden, dargestellt in Grammatik in Γ_3.

Grammatik Γ_3: IF-ELSE in Pascal

```
<Statement>     ::= <Ifstatement> | <Assignment>
<Assigment>     ::= <Variable> := <Expression>
<Ifstatement>   ::= IF <Expression> THEN <Statement> <Elsepart>
<Elsepart>      ::= | ELSE <Statement>
```

Man findet in der Literatur häufig Varianten der “Ur“-EBNF:

- Als Produktionszeichen wird ’=’ statt ’::=’ verwendet.
- Die spitzen Klammern fehlen, dafür stehen Terminals in Anführungszeichen.
- Ein Punkt kennzeichnet das Ende einer Regel.

Die einfache Deutsch-Grammatik Γ_2 wird in BNF als Γ_4 dargestellt.

Grammatik Γ_4: Einfach-Deutsch in EBNF

```
<S>     ::= <NP> <VP>
<NP>    ::= N | A N
<VP>    ::= V | V <NP>
```

Die Backus-Naur-Form kann auch in sich selbst dargestellt werden, wie in Grammatik Γ_5 gezeigt.

Grammatik Γ_5: Die Backus-Naur-Form

```
<S>     ::= <NP> <VP>
<NP>    ::= N | A N
<VP>    ::= V | V <NP>
```

Erweiterte Backus-Naur-Form (EBNF)

In der erweiterten BNF (EBNF) In der erweiterten BNF (EBNF) gibt es einige zusätzliche Möglichkeiten zur Beschreibung einer Sprache:

[1] John Backus, amerikanischer Informatiker, *3.12.1924
[2] Peter Naur, dänischer Informatiker,*25. Oktober 1928

- Optionale Teile stehen in eckigen Klammern:
  `[<Elsepart>]`
- Geschweifte Klammern umschließen beliebige Wiederholungen (auch Null!):
  `<Var> {,<Var>}`
- Setzen von Prioritäten durch runde Klammern:
  `(<A>|<B>) <C>`

Die Grammatik der EBNF-Darstellung kann jetzt in EBNF angegeben werden: Grammatik Γ_6

Grammatik Γ_6: EBNF

Start = *syntax*
syntax = {*produktion*} .
produktion = *bezeichner* “=“ *ausdruck*.
ausdruck = *term* {“|“ *term*} .
term = *faktor* {*faktor*} .
faktor = *bezeichner* | *string* | “(“*ausdruck*“)“ | “[“*ausdruck*“]“ | “**{**“*ausdruck*“**}**“.
bezeichner = *buchstabe* {*buchstabe*|*ziffer*} .
string = “‘“ {*buchstabe*} “‘“.
buchstabe = “**A**“|“**B**“|...|“**Z**“.
ziffer = “**0**“|“**1**“|...|“**9**“.

Die einfache Deutsch-Grammatik Γ_2 wird in der erweiterten Backus-Naur-Form als Γ_7 dargestellt.

Grammatik Γ_7: Einfach-Deutsch in EBNF

```
<S>     ::= <NP> <VP>
<NP>    ::= [A] N
<VP>    ::= V [<NP>]
```

Syntaxdiagramme

In Syntaxdiagrammen werden Grammatiken grafisch dargestellt. Terminalsymbole werden durch einen Kreis oder ein Oval (oder ein Rechteck mit abgerundeten Ecken) dargestellt, Nichtterminalsymbole durch Rechtecke. Für die Möglichkeiten, eine Produktion einer in EBNF notierten Grammatik darzustellen ergeben sich die folgenden Graphen:

Nicht-Terminal: Produktionsregeln (Nicht-Terminale) werden durch Rechtecke dargestellt:

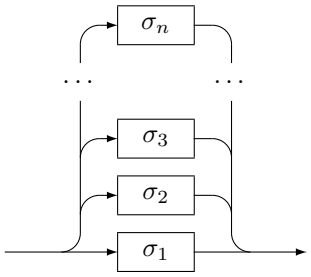
Syntaxdiagramm 1: Nicht-Terminal A

Terminal: Terminale Symbole werden durch Ovale (abgerundete Rechtecke / Ellipsen / Kreise) dargestellt:

Syntaxdiagramm 2: Terminal-Symbol σ

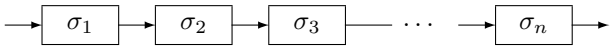
Alternative: Produktionsregeln der Form $A \to \sigma_1|\sigma_2|\sigma_3|\ldots|\sigma_n$

Syntaxdiagramm 3: Alternative



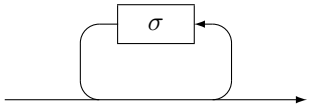
Sequenz: Produktionsregeln der Form $A \to \sigma_1\sigma_2\sigma_3\ldots\sigma_n$

Syntaxdiagramm 4: Sequenz



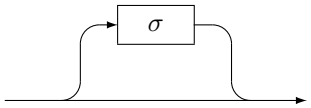
Wiederholung: Produktionsregeln der Form $A \to \{\sigma\}$

Syntaxdiagramm 5: Wiederholung



Option: Produktionsregeln der Form $A \to [\sigma]$

Syntaxdiagramm 6: Option



Reguläre Ausdrücke

Für Sprachen des Chomsky-Typs 3 - den regulären Sprachen - hat sich in der Informatik eine weitere Notationsform eingebürgert, nämlich die regulären Ausdrücke. Diese sind jedoch nicht in irgendeinem Sinne genormt, sondern von Programm zu Programm unterschiedlich. Sie bestehen aus Zeichen und sog. Meta-Zeichen, die eine spezielle bedeutung haben. So stellt das Zeichen ’*’ etwa eine Wiederholung dar, was in der EBN der ’’-Klammerung entspricht, das Zeichen ’?’ stellt die Option dar. Die Meta-Zeichen des Scanner-Generators Lex sind in Tabelle **??** (Seite **??**) dargestellt.

Reguläre Ausdrücke können keine Typ-3-Sprachen beschreiben, also keine Klammerstrukturen.

Die einfache Deutsch-Grammatik Γ_2 wird als regulärere Ausdruck als Γ_8 dargestellt.

Grammatik Γ_8: Einfach-Deutsch in EBNF

```
A?NV(A?N)?
```

Reguläre Ausdrücke bestehen nur aus einer Produktionsregel, das nichtterminale Symbol welches durch diese Produktionsregel beschrieben wird ist immer das Startsymbol und kann daher weggelassen werden. Auf der rechten Seite dieser einen Regel stehen nur terminale Symbole.

3.1.3 Notation in diesem Script

in diesem Script wird der besseren Lesbarkeit wegen fast durchgängig eine modifizierte EBNF verwendet:

- Das Produktionszeichen ist $\rightarrow$.
- Terminals stehen in "-Zeichen und sind klein geschrieben.
- Terminals sind groß geschrieben.

Die einfache Deutsch-Grammatik Γ_2 wird in der Script-Notation als Γ_9 dargestellt.

Grammatik Γ_9: Einfach-Deutsch in Script-Notation

S	$\rightarrow$	**NP VP**
NP	$\rightarrow$	**['a'] 'n'**
VP	$\rightarrow$	**'v' [NP]**

3.1.4 Grammatiken und Automaten

Es können zu einer Sprache mehrere Grammatiken angegeben werden, oder andersherum, mehrere Grammatiken können ein- und dieselbe Sprache beschreiben. Daraus resultiert, daß Grammatiken ungeformt werden können, worauf hier jedoch nicht näher eingegangen werden soll.

Definition 3.2 (Kontextfreie Grammatik). *Eine Grammatik heiss kontextfrei (Chomsky-Typ 2), wenn auf der linken Seite aller Produktionsregeln nur ein nicht-terminales Symbol steht.*

Definition 3.3 (Reguläre Grammatik). *Eine Grammatik heisst regulär (Chomsky-Typ 3), wenn die Grammatik auf eine Produktionsregel reduziert werden kann und das Startsymbol nicht auf der rechten Seite dieser Produktionsregel steht.*

Ein Beispiel für eine reguläre Grammatik ist die Deutsch-Grammatik der Einleitung oder die folgende Grammatik Γ_{10} die eine Fixkommazahl beschreibt.

Grammatik Γ_{10}: Fixkommazahl

ZAHL	$\rightarrow$	**VK [NK]**
VK	$\rightarrow$	**INT**
NK	$\rightarrow$	**'.' INT**
INT	$\rightarrow$	**DIGIT DIGIT**
DIGIT	$\rightarrow$	**'0'** | **'1'** | **'2'** | **'3'** | **'4'** | **'5'** | **'6'** | **'7'** | **'8'** | **'9'**

Ein Beispiel für eine kontextfreie Grammatik ist Γ_{11} die eine geklammerte ganze Zahlen beschreibt.

Grammatik Γ_{11}: Geklammerte Zahl

SATZ	$\rightarrow$	**INT** | **'(' SATZ ')'**
INT	$\rightarrow$	**DIGIT {DIGIT}**
DIGIT	$\rightarrow$	**'0'** | **'1'** | **'2'** | **'3'** | **'4'** | **'5'** | **'6'** | **'7'** | **'8'** | **'9'**

3.2 Endliche Automaten

3.2.1 Grundbegriffe

Ein endlicher Automat ist ein System, das interne Zustände abhängig von einer Eingabe annimmt.

Ein endlicher Automat A ist damit ein Fünftupel A = (Z, E, δ, z_0, F).

Z ist die Menge der Zustände in denen sich der Automat A befinden kann.

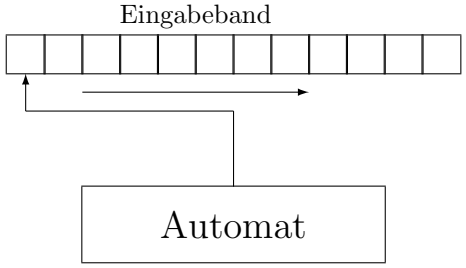
E ist das Eingabealphabet des Automaten (die Menge der Eingabesymbole).

δ ist die Übergangsfunktion, die jeder Kombination aus Zustand und Eingabesymbol einen Folgezustand zuordnet. $Z \otimes E \rightarrow Z$.

z_0 ist der Startzustand, in dem sich der Automat am Anfang befindet.

F ist die Menge der Endezustände ($F \subseteq Z$). Kommt ein Automat in einen Endezustand so hört er auf zu arbeiten.

3.2.2 Modell eines erkennenden Automaten



3.2.3 Darstellungsformen

Tabellarische Darstellung

Die Übergangsfunktion δ kann als Tabelle dargestellt werden. Dabei werden die Zustände meist als Zeilen und das Eingabealphabet als Spalten dargestellt.

Für Endezustände gibt es keine Zeilen (da aus diesen Zuständen nicht mehr gewechselt wird).

Die tabellarische Darstellung ist für uns Menschen weniger übersichtlich, lässt sich jedoch einfacher in ein Programm codieren.

Graphische Darstellung

Die Übergangsfunktion δ kann als Übergangsgraph (Diagramm) dargestellt werden. Dabei werden die Zustände als Kreise (doppelte Linien für Endezustände) dargestellt. Die Übergangsfunktion δ wird durch Pfeile dargestellt, die Pfeile werden mit einem Zeichen aus dem Eingabealphabet beschriftet.

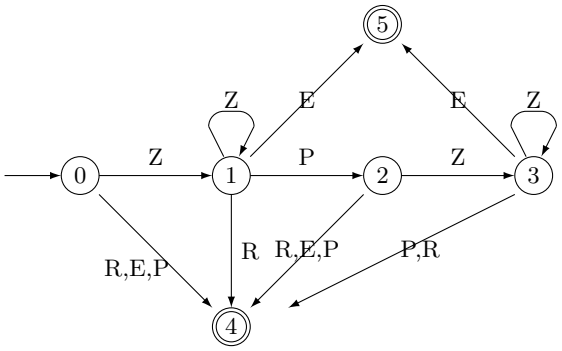
Von Endezuständen führen keine Pfeile weg. Der Startzustand kann durch einen Initialpfeil markiert werden.

Die graphische Darstellung ist für uns Menschen sehr übersichtlich, lässt sich jedoch nicht so einfach in ein Programm codieren.

3.2.4 Ein einführendes Beispiel

Ein Automat soll erkennen, ob es sich bei seiner Eingabe um eine korrekte Fixkommazahl handelt. Die reguläre Grammatik, die überprüft werden soll, war Γ_{10} (Seite 25). Das Eingabealphabet des Automaten besteht aus den Elementen “Z“ (Ziffer), “P“ (Dezimalpunkt, “E“ (Ende) und “R“ (Rest):

Abb. 3.4. Automat zur Zahlenerkennung



$Z = \{0, 1, 2, 3, 4, 5\}$
$E = \{E, Z, P, R\}$

$\delta =$

δ	E	Z	P	R
0	4	1	4	4
1	5	1	2	4
2	4	3	4	4
3	5	3	4	4

$z_0 = 0$
$F = \{4, 5\}$

Satz 1. Zu jeder regulären Sprache (Chomsky-Typ-3) kann ein äquivalenter endlicher Automat konstruiert werden, der die Syntax dieser Sprache überprüft.

3.2.5 Endliche Automaten und Formale Sprachen

Satz 2. zu jedem endlichen Automaten kann eine reguläre Grammatik entwickelt werden, die die von diesem Automat akzeptierte Sprache beschreibt.

3.2.6 Grundalgorithmus eines Endlichen Automaten

Algorithmus 2: Grundalgorithmus 1 eines endlichen deterministischen Automaten

$zustand = z_0$
solange $zustand \notin F$
lies $Eingabezeichen$
$zustand = \delta(zustand, Eingabezeichen)$

Algorithmus 3: Grundalgorithmus 2 eines endlichen deterministischen Automaten

$zustand = z_0$
solange nicht am Eingabeende
lies $Eingabezeichen$
$zustand = \delta(zustand, Eingabezeichen)$

3.3 Kellerautomaten

Satz 3. Zu jeder kontextfreien Sprache (Chomsky-Typ-2) kann ein äquivalenter Kellerautomat konstruiert werden, der die Syntax dieser Sprache überprüft.

Satz 4. Zu jedem Kellerautomaten kann eine kontextfreien Grammatik (Chomsky-Typ-2) entwickelt werden, die die von diesem Automat akzeptierte Sprache beschreibt.

3.4 Übungen

Gegeben ist Grammatik Γ_{12}:

Grammatik Γ_{12}: Mathematischer Term
E → **T {('+' | '-') T}**
T → **F {('*' | '/') F}**
F → **zahl** | **'(' E ')'** | **'-' F** | **'+' F**
Dabei sind E, T und F Non-Terminals, zahl und die Zeichen '+', '-', '*', '/', '(' und ') sind Terminals. Γ_{12} ist eine Grammatik zur Beschreibung arithmetischer Terme.

Übung 3.4. Grammatik-Art
Von welchem Chomsky-Typ ist Γ_{12}?

Übung 3.5. Syntaxbaum
Erstellen Sie den Syntaxbaum für den Term $-(-1 * -2)$.

Übung 3.6. BNF-Umformung
Formen Sie Γ_{12} in BNF um.

Übung 3.7. Syntaxdiagramme
Zeichnen Sie die Syntaxdiagramme für Γ_{12}.

4

Grundlagen

4.1 UPN

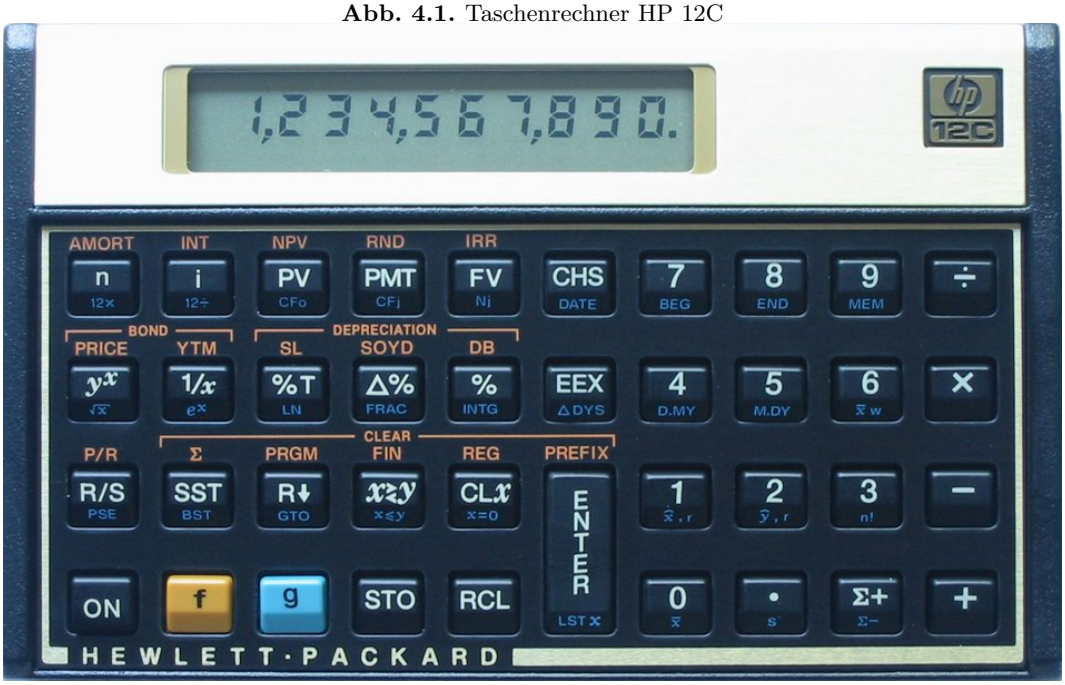
UPN steht für “Umgekehrte Polnische Notation“, im englischen Sprachgebrauch entsprechend RPN genannt. ein weiters Synonym ist “Postfix-Notation“.

UPN hat für maschinelle Verarbeitung entscheidende Vorteile:

- Es gibt Operatoren keinerlei Prioritätsregeln wie “Punkt vor Strich“.
- Es gibt keine Prioritätsklammern.

Aus diesem Grund wurden und werden Prozessoren gerne als Register-Stack-Maschinen entwickelt. Die Firma Hewlett-Packard baute lange Zeit Taschenrechner, die in UPN bedient wurden, Abbildung 4.1 zeigt exemplarisch das Modell HP 12C. Wie zu erkennen ist, hat der Taschenrechner weder eine Gleichheits- noch Klammertasten, dafür aber die Enter-Taste die zum Trennen der Eingabe dient. Übrigens verwenden die klassischen Taschenrechner keine

Abb. 4.1. Taschenrechner HP 12C



reine Infix-Notation, sondern bei Nutzung von Funktionen ebenfalls die Postfix-Notation. Zur Berechnung des Terms $\sqrt{3^2 + 4^2}$ ist die Wurzeltaste zuletzt zu drücken.

UPN wird folgendermaßen abgearbeitet:

- Zahlen (Operanden) werden auf den Stack gelegt.
- Bei Operatoren wird je nach Wertigkeit des Operators (unär oder binär) die entsprechende Zahl von Operanden vom Stack geholt, entsprechend dem Operator miteinander verknüpft und das Ergebnis wieder auf den Stack gelegt.
- Bei Funktionen wird je nach Anzahl der Funktionsparameter die entsprechende Zahl von Operanden vom Stack geholt, entsprechend der Funktion miteinander verknüpft und das Ergebnis wieder auf den Stack gelegt.

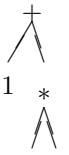
Ausdrücke müssen für Register-Stack-Maschinen in Reverse-Notation (UPN) umgewandelt werden:

Ausdruck	Maschinencode

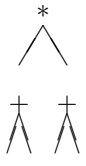
1 + 2 * 3	`LOADC 1`
	`LOADC 2`
	`LOADC 3`
	`MULT`
	`ADD`

(1 + 2) * (3 + 4)	`LOADC 1`
	`LOADC 2`
	`ADD`
	`LOADC 3`
	`LOADC 4`
	`ADD`
	`MULT`

Die Umsetzung kann durch den Syntaxbaum erfolgen.



Wird dieser Baum im Postorder-Verfahren Durchlaufen, so ergibt sich die Abfolge `1 2 3 * +`, was dem obigen Maschinencode entspricht.



Wird dieser Baum im Postorder-Verfahren Durchlaufen, so ergibt sich die Abfolge `1 2 + 3 4 + *`, was dem obigen Maschinencode entspricht.

4.2 Scanner und Parser

Grammatik Γ_{13} ist eine Modifikation von Grammatik Γ_{12} (Seite 28). Dabei wurde das terminale Symbol `zahl` durch ein weiteres Non-Terminal ersetzt:

Grammatik Γ_{13}: Term-Grammatik 1
E → **T {('+'—'-') T}**
T → **F {('*'—'/') F}**
F → **Z — '(' E ')' — '-' F**
Z → **('0'—'1'—...—'9'){'0'—'1'—...—'9'}**
Nun könnte ein Kellerautomat entwickelt werden, der die Syntax eines Eingabetextes überprüft, wobei der Eingabetext zeichenweise verarbeitet werden kann. in der Praxis ist dieses Vorgehen aber aus verschiedenen Gründen unpraktisch:

- Die Anzahl der Non-Terminals ist unnötig hoch.
- In Programmiersprachen entsteht ein Konflikt zwischen Bezeichnern und Schlüsselwörtern (siehe etwa die PL/0-Grammatik Γ_1.
- Sog. whitespace-Zeichen müssen überlesen werden und würden eine Grammatik unnötig komplizierter machen.

Daher wird man diejenigen Regeln einer Grammatik, die für sich genommen vom Chomsky-Typ 2 sind, aus der Grammatik herausnehmen und gesondert behandeln:

- Die als Typ 2 darstellbaren Regeln werden über die sog. lexikalische Analyse erkannt.
- Die restlichen Regeln werden von der syntaktischen Analyse erkannt.

Die lexikalische Analyse übernimmt der sog. Scanner, die syntaktische Analyse der Parser.

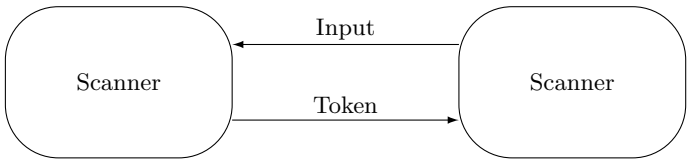
Definition 4.1 (Scanner). *Scanner zerlegen eine Eingabe in ihre Bestandteile. Es erfolgt keine Überprüfung ob die gefundenen Teile Sinn ergeben. Diesen Vorgang nennt man auch "Lexikalische Analyse".*

Definition 4.2 (Parser). *Parser überprüfen eine Folge von Zeichen ihres Eingabealphabets auf korrekte Reihenfolge. Diesen Vorgang nennt man auch "Syntaktische Analyse" oder "Grammatikalische Analyse".*

Definition 4.3. *Die Elemente einer Sprache, die ein Scanner findet, nennt man Token.*

4.2.1 Zusammenspiel von Scanner und Parser

Abb. 4.2. Zusammenspiel zwischen Scanner und Parser



Scanner und Parser sollten zwecks logischer Trennung in zwei unterschiedlichen Funktionen implementiert werden. Der Rückgabewert von Scanner zum Parser (das sog. “Token“) wird von Datentyp int angelegt.

Der Scanner wird vom Parser mehrfach aufgerufen und liefert das jeweils nächjste Token zurück. Vom Modell her liest er einfach aus der Eingabe. Ist die Eingabe eine Datei (ein FILE-Zeiger), so übernimmt das Betriebssystem die Verwaltung des Lesezeigers. Wird dagegen aus einem Speicherbereich gelesen, so muss über scanner-interne statische Variablen die aktuelle Leseposition gespeichert werden.

Als Beispiel soll die Token-Folge für den C-Text **if** (a > 2) betrachtet werden:

- Token “if“
- Token “Klammer auf“
- Token “Bezeichner“, Wert = “a“
- Token “Operator >“
- Token “Konstante“, Wert = 2
- Token “Klammer zu“

Bei vielen Grammatiken ist jedoch das Token als Rückgabe ungenügend. Unter 4.2 wurde diskutiert, die Grammatik einer formalen Sprache nicht bis ins letzte Zeichen zu formulieren, sondern einzelne Regeln, die für sich genommen eine Typ-2-Grammatik darstellen, durch den Scanner zu verarbeiten. daraus folgt aber, dass Scanner einer typischen Programmiersprache wuie PL/0 oder C alle Bezeichner durch ein- und dasselbe Token an den Parser melden und dieser damit keine Möglichkeit hat, den eigentlichen Namen des Bezeichners zu erfahren. Genauso verhält es sich mit Zahlen.

Aus diesem Grund geben Scanner in der Praxis nicht nur einen Tokenwert sondern zusätzlich den gescannten Text oder Zahlenwerte oder ähnliches zurück.

4.3 Primitive Scanner - zeichenbasiert

Häufig machen Scanner nichts anderes, als verschiedene Zeichen zu einer Gruppe oder Klasse zusammenzufassen, d.h. der Scanner liest immer genau ein Zeichen aus dem Eingabestrom. Diese Scanner sind sehr einfach aufgebaut, da sie immer nur einen Vergleich eines einzelnen Zeichens machen müssen. Betrachten wir noch einmal den Automat in Abbildung 3.4 (Seite 27), so erkennen wir, dass im gesamten Automaten die Ziffern 0 bis 9 immer gleich behandelt werden. Würde man nun alle Ziffern im Eingabealphabet E angeben, so wäre die *delta*-Tabelle 13 Spalten breit. Setzt man aber einen zeichenbasierten Scanner ein, der alle Ziffern zu einem Token $T \in E$ zusammenfasst, kommt die ß*delta*-Tabelle mit vier Spalten aus. Ein Scanner für diesen Automat sollte also eines der Tokens Z (Ziffer), P (Punkt), E (Ende) und R (Rest) zurückliefern. Der Scanner selbst ist in Listing 4.1 codiert, wobei das Hauptprogramm keinen Automaten darstellt, sondern lediglich den Aufruf des Scanners demonstriert. In Zeile 28 des Programms wird lediglich ein Zurücksetzen des Scanners durchgeführt, in Zeile 31 erfolgt der eigentliche Aufruf des Scanners.

Listing 4.1. Ein zeichenbasierter Scanner (grundl-scanner-fixkommazahl.c)

```
1  /*******************************************************
2  Scanner für Fixkommazahl
3  *******************************************************/
4  #include <stdio.h>
5
6  typedef enum {t_ende, t_ziffer, t_punkt, t_rest} token;
```

```
7
8  token fixkomma_scanner(char * input, int reset) {
9      static char * p;
10     token t;
11     if (reset) {
12         p = input - 1;
13         return t_ende; // Keine Verarbeitung!
14     }
15     if (*(++p) >= '0' && *p <= '9') t = t_ziffer;
16     else if (*p == '.' || *p == ',') t =  t_punkt;
17     else if (*p == '\0' || *p == '\n') t = t_ende;
18     else t = t_rest;
19
20     return t;
21 }
22
23 int main() {
24     token t;
25     char input[255];
26
27     while(printf("Brauche Input: "), fgets(input, 254, stdin) != NULL) {
28         fixkomma_scanner(input, 1);
29         printf("Token-Folge:");
30         do
31             printf(" %d", t = (int) fixkomma_scanner(input, 0));
32         while (t != t_ende);
33         printf("\n\n");
34     }
35     return 0;
36 }
```

Zeichenbasierte Scanner lassen sich sehr effizient tabellengesteuert implementieren.

4.4 Primitive Scanner - wortbasiert

Hin und wieder müssen Scanner entwickelt werden, die einen Text einfach in Worte zerlegen müssen, wobei diese Worte durch spezielle Zeichen getrennt sind. Die C-Standard-Bibliothek `string.h` stellt hierzu die strtok-Funktion (abgekürzt für string-token) zur Verfügung:

```
char * strtok(char * input, char * delimiter);
```

- Um das erste Wort zu scannen wird der zu scannende String als Parameter übergeben. Zurückgegeben wird ein Zeiger auf das Wort. Dies ist der Reset-Vorgang des Scanners.
- In den folgenden Scanner-Aufrufen wird für den Eingabestring-Zeiger der NULL-Zeiger übergeben. Zurückgegeben wird wieder ein Zeiger auf das gefundene Wort.
- Kann nichts mehr gescannt werden, so wird der Nullzeiger zurückgegeben.
- Zu beachten ist, dass der Eingabestring verändert wird.

Listing 4.2 demonstriert, wie einfach ein wortbasierter Scanner mittels der strtok-Funktion zu implementieren ist.

Listing 4.2. Scannen durch strtok-Funktion (grundl-strtok.c)

```
1  /*********************************************************
2  Demonstriert Scannen eines Strings durch die Bibliotheks-
3  Funktion strtok
4  *********************************************************/
5  #include <stdio.h>
6  #include <string.h>
7
8  int main() {
9      char to_scan[] = " \tDas ist\n ein Test\t zum scannen ";
10     char delimiter[] = " \t\n"; // Leer, Tab und Zeilenumbruch
11     char * word;
12     int nr = 0;
13
14     printf("Zu scannender text: '%s'\n", to_scan);
15     word = strtok(to_scan, delimiter); // Init und erstes Wort
16     while (word != NULL) {
17         printf("Wort %d: '%s'\n", ++nr, word);
18         word = strtok(NULL, delimiter); // folgende Woerter
19     }
20     return 0;
21 }
```

4.4.1 Prinzipien eines Scanners

Die Einfach-Scanner "wortweise" und "zeichenweise" kommen in der Praxis recht selten vor. Es wäre eine Zumutung, in C alle Sprachelemente durch Whitespace-Zeichen zu trennen. Statt "`if(a>2)`" müsste "`if␣(␣a␣>␣2␣)`" codiert werden.

Durch das Weglassen der trennenden Whitespace-Zeichen kann es aber zu Mehrdeutigkeiten beim Scannen führen. So kann der C-Text "`if_a`" vom Scanner entweder in die Token-Folge "Schlüsselworzt if - Bezeichner _a" oder aber in die Token-Folge "Bezeichner if_a" zerlegt werden. Das Verhalten eines Scanners in einem solchen Fall ist Definitionssache:

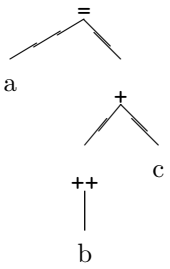
Definition 4.4. *Ein Scanner wählt immer das Token, das den längsten Eingabetext abbildet. Dies nennt man das "Maximum-Match-Prinzip".*

Als Beispiel für Maximum-Match dient der Ausdruck in Zeile 4 Listing 4.3.

Listing 4.3. Beispiel für Maximum-Match-Methode (grundl-maximum-match.c)

```
1  #include <stdio.h>
2  int main() {
3      int a, b = 1, c = 2;
4      a=b+++c; // Achtung!!!!
5      printf("a = %d, b = %d, c = %d\n", a, b, c);
6      return 0;
7  }
```

Der C-Scanner zerlegt den Ausdruck in Zeile 4 in die Token-Folge "Bezeichner a - Operator = - Bezeichner b - Operator ++ - Operator + - Bezeichner c". Aus dieser Token-Folge produziert der Parser dann den folgenden Syntaxbaum:



Das Maximum-Match-Problem träte auch in Zeile 6 auf, wenn das Leerzeichen nicht eingegeben worden wäre. In Diesem Fall würde der Scanner statt der Token-Folge “Schlüsselwort return - Int-Konstante 0“ die Tokenfolge “Bezeichner return0“ liefern.

Scanner müssen - wenn gefordert wird, möglichst keine Leerzeichen einzugeben - oft vorausschauend agieren. Beispielsweise wird ein Scanner im C-Ausdruck `1.2E3+4` bis zum Zeichen `+` davon ausgehen, eine Fließkomma-Zahl einzulesen. Das `+`-Zeichen produziert aber keinen Syntax-Fehler, sondern beendet das Scannen der Fließkommazahl und wird - für den nächsten Scan-Vorgang - in die Eingabe zurückgestellt.

Definition 4.5. *Arbeitet ein Scanner vorausschauend, so nennt man diese einen Look-Ahead.*

4.4.2 Implementierung von Scannern

Scanner bestehen meist aus einer Menge von regulären Ausdrücken und werden damit am besten als endliche Automaten codiert. Diese Technik sichert schnelle Scan-Vorgänge, da die Laufzeit proportional zur Eingabestromlänge ist. Schlechte Scanner arbeiten mit einfachen String-Vergleichen.

Da die δ-Tabellen der Automaten meist sehr groß sind, werden zur Codierung von Scannern gerne Tools wie Lex eingesetzt.

Desweiteren überlesen Scanner Whitespace-Zeichen, weshalb diese nicht in der Grammatik des Parsers vorkommen.

4.4.3 Fehlerbehandlung bei Scannern

4.5 Bootstrapping - Das Henne-Ei-Problem

4.6 Reguläre Ausdrücke

4.7 Reguläre Ausdrücke in C

Listing 4.4. Demo zur Benutzung von Regulären Ausdrücken in C (regex.c)

```
1  /************************************************************
2  Demonstrier Reguläre Ausdrücke mit C
3  ************************************************************/
4  #include <stdio.h>
5  #include <regex.h>
6  #include <string.h>
7
8  int main(int argc, char *argv[])
```

```
9  {
10     char input[255], rp[255]="^[0-9]+(\\.[0-9]+)?$", errortext[255];
11     regex_t regexpr;
12     int rc;
13
14     printf("Brauche Input: ");
15     fgets(input, 254, stdin);
16     input[strlen(input) - 1] = '\0';
17     printf("Input: %s\nRegExp: %s\n", input, rp);
18     if ((rc = regcomp(&regexpr, rp, REG_EXTENDED | REG_NOSUB)) != 0) {
19         regerror(rc, &regexpr, errortext, 254);
20         printf("Problem beim Ausdruck %s: %s\n", rp, errortext);
21     }
22     else {
23         rc = regexec(&regexpr, input, 0, NULL, 0);
24         regerror(rc, &regexpr, errortext, 254);
25         printf("Antwort: %d - %s\n", rc, errortext);
26     }
27     regfree(&regexpr);
28     return 0;
29 }
```

4.8 Übungen

Übung 4.6. Syntaxbäume
Setzen Sie die folgenden Mathematischen Terme in Syntaxbäume und UPN um, arbeiten Sie anschließend den UPN-Code ab.

$$a = \frac{1}{\frac{1}{b} + \frac{1}{c}}$$

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Übung 4.7. Scanner
"Spielen" Sie für folgenden C-Code Scanner:

```
int main() {
    int a = 125, b = 250;
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    printf("%d", a);
    return 0;
}
```

Übung 4.8. Grammatik
Gegeben ist die folgende Sprache:

```
S ::= E {O E}
```

```
E ::= F {(A | N) F}
F ::= W | '(' E ')'
O ::= ('O' | 'o') ('R' | 'r')
N ::= ('N' | 'n') ('O' | 'o') ('T' | 't')
A ::= ('A' | 'a') ('N' | 'n') ('D' | 'd')
W ::= B {B}
B ::= 'A' | 'a' | 'B' | 'b' | ... | 'Z' | 'z'
```

Die Grammatik enthält die Schlüsselwörter 'AND', 'OR' und 'NOT', wobei Groß-Kleinschreibung unbedeutend ist, bei der Regel W sind die drei Schlüsselwörter ausgenommen. Definieren Sie, welche der Regeln von einem Scanner und welche von einem Parser verarbeitet werden sollten.

5

Endliche Automaten

Wie eingangs erwähnt, sind formale Sprachen und Automaten eng miteinander verbunden. Zu jeder regulären Grammatik kann ein endlicher Automat konstruiert werden, der diese Grammatik erkennt. Anders herum kann zu jedem endlichen Automaten eine Grammatik angegeben werden.

Wie man zu regulären Sprachen Automaten konstruiert und wie man zu Automaten reguläre Sprachen entwickelt soll hier nicht Gegenstand sondern Voraussetzung sein.

5.1 Grundbegriffe

Ein endlicher Automat ist ein System, das interne Zustände abhängig von einer Eingabe annimmt. Abbildung 5.1 zeigt das Modell eines Automaten.

Ein endlicher Automat A ist damit ein Fünftupel A = (Z, E, δ, z_0, F).

Z ist die Menge der Zustände in denen sich der Automat A befinden kann.

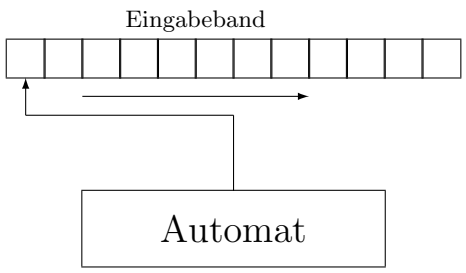
E ist das Eingabealphabet des Automaten (die Menge der Eingabesymbole).

δ ist die Übergangsfunktion, die jeder Kombination aus Zustand und Eingabesymbol einen Folgezustand zuordnet. $Z \otimes E \rightarrow Z$.

z_0 ist der Startzustand, in dem sich der Automat am Anfang befindet.

F ist die Menge der Endezustände ($F \subseteq Z$). Kommt ein Automat in einen Endezustand so hört er auf zu arbeiten.

Abb. 5.1. Modell eines endlichen Automaten



5.2 Darstellungsformen

Abbildung 3.4 auf Seite 27 zeigte bereits einen endlichen Automaten sowohl in grafischer als auch in tabellarischer Darstellungsform.

5.2.1 Tabellarische Darstellung

Die Übergangsfunktion δ kann als Tabelle dargestellt werden. Dabei werden die Zustände meist als Zeilen und das Eingabealphabet als Spalten dargestellt.

Für Endezustände gibt es keine Zeilen (da aus diesen Zuständen nicht mehr gewechselt wird).

Die tabellarische Darstellung ist für uns Menschen weniger übersichtlich, lässt sich jedoch einfacher in ein Programm codieren.

- Tabellarische Darstellung
- Graphische Darstellung

5.2.2 Graphische Darstellung

Die Übergangsfunktion δ kann als Übergangsgraph (Diagramm) dargestellt werden. Dabei werden die Zustände als Kreise (doppelte Linien für Endezustände) dargestellt. Die Übergangsfunktion δ wird durch Pfeile dargestellt, die Pfeile werden mit einem Zeichen aus dem Eingabealphabet beschriftet.

Von Endezuständen führen keine Pfeile weg. Der Startzustand kann durch einen Initialpfeil markiert werden.

Die graphische Darstellung ist für uns Menschen sehr übersichtlich, lässt sich jedoch nicht so einfach in ein Programm codieren.

5.3 Algorithmus eines endlichen Automaten

Endliche Automaten werden auf zwei verschiedene Arten codiert, die in den Abbildungen 5.2 und 5.3 dargestellt sind. Je nach Aufgabe - Erkennen, Suchen, Scannen, ... - ist mal die eine und mal die andere Variante zu bevorzugen.

5.4 Ein einführendes Beispiel

Der Automat zur Erkennung von Fixkommazahlen (Abbildung 3.4 Seite 27) soll gemäß Abbildung 5.2 codiert werden. Der reguläre Ausdruck für diesen Automaten lautet `[0-9]+("."[0-9]+)?`.

Das folgende Programm hat - um die Unterschiede zu verdeutlichen - den endlichen Automaten zweimal implementiert:

Tabellengesteuert In der Funktion automat_t wird der Automat tabellengesteuert implementiert. Die δ-Funktion des Automaten wird in einem zweidimensionalen Feld gespeichert. Die Zeilen entsprechen den Zuständen, die Spalten den Tokens der lexikalischen Analyse.

Abb. 5.2. Algorithmus eines endlichen Automaten - Stop bei Endezustand

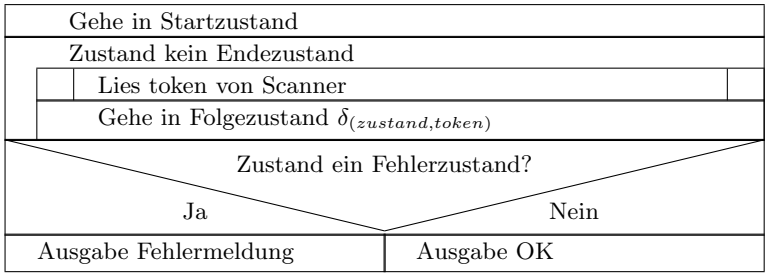
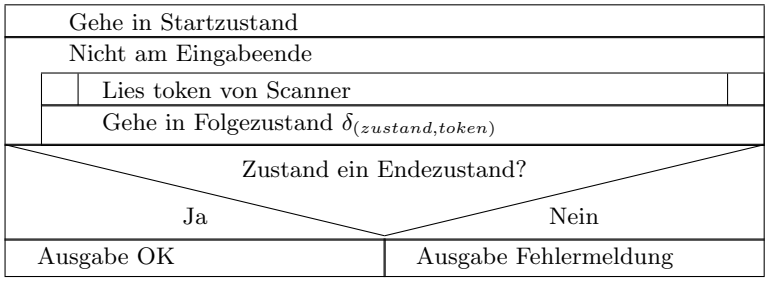


Abb. 5.3. Algorithmus eines endlichen Automaten - Stop bei Eingabeende



Programmgesteuert In der Funktion automat_p wird der Automat programmgesteuert implementiert. Dazu werden Mehrfachverzweigungen in zwei Ebenen ineinander geschachtelt. Die äußere Mehrfachverzweigung entspricht dem Zustand, die innere jeweils dem Token der lexikalischen Analyse.

Listing 5.1. Zahlenerkennung (automaten-zahlenerkennung.c)

```
1  /*********************************************************
2  Erkennung von Fixkommazahlen
3  Regulärer Ausdruck:
4  [0-9]+("."[0-9]+)?
5  *********************************************************/
6  #include <stdio.h>
7
8  int scanner(char *, int); // Prototypen
9  int automat_t(char *);
10 int automat_p(char *);
11
12 enum {E, Z, P, R};  // Token, Eingabealphabet
13
14 int main() {
15     char input[255];
16
17     while (printf("\nInput:"),scanf("%s", input) != EOF) {
```

```
18          printf("\nT:%s OK!", (!automat_t(input))? "":" nicht");
19          printf("\nP:%s OK!", (!automat_p(input))? "":" nicht");
20      }
21      printf("\n");
22      return 0;
23  }
24
25  int automat_t(char *in) { // Tabellengesteuerter Automat
26  // Rückgabe 0: eingabe OK; Rückgabe 1: Eingabe NICHT OK
27      int zustand = 0;
28      int delta[][4] = {/* 0 1 2 3  Token*/
29          /*Zustand 0*/   {4,1,4,4},
30          /*Zustand 1*/   {5,1,2,4},
31          /*Zustand 2*/   {4,3,4,4},
32          /*Zustand 3*/   {5,3,4,4}};
33
34      scanner(in, 1);
35      while (zustand <4)
36          zustand = delta[zustand][scanner(in, 0)];
37      return (zustand == 4);
38  }
39
40  int automat_p(char *in) { // Programmgesteuerter Automat
41  // Rückgabe 0: eingabe OK; Rückgabe 1: Eingabe NICHT OK
42      int zustand = 0, token;
43
44      scanner(in, 1); // Scanner-Reset
45      while (zustand <4) {
46          token = scanner(in, 0);
47          switch (zustand) {
48          case /*Zustand*/ 0: switch (token) {
49                  case 0: zustand = 4; break;
50                  case 1: zustand = 1; break;
51                  case 2: zustand = 4; break;
52                  case 3: zustand = 4; break;
53                  }
54                  break;
55          case /*Zustand*/ 1: switch (token) {
56                  case 0: zustand = 5; break;
57                  case 1: zustand = 1; break;
58                  case 2: zustand = 2; break;
59                  case 3: zustand = 4; break;
60                  }
61                  break;
62          case /*Zustand*/ 2: switch (token) {
63                  case 0: zustand = 4; break;
64                  case 1: zustand = 3; break;
65                  case 2: zustand = 4; break;
66                  case 3: zustand = 4; break;
67                  }
68                  break;
69          case /*Zustand*/ 3: switch (token) {
70                  case 0: zustand = 5; break;
71                  case 1: zustand = 3; break;
```

```
72                  case 2: zustand = 4; break;
73                  case 3: zustand = 4; break;
74                  }
75                  break;
76          }
77      }
78      return (zustand == 4);
79  }
80
81  int scanner(char *in, int reset) {
82      static char *p;
83      char c;
84      if (reset) { // p auf in setzen
85          p = in;
86          return E;
87      }
88      c = *(p++);
89      if (c >= '0' && c <= '9') return Z; // Ziffer
90      if (c == '\0') return E;            // Ende
91      if (c == '.') return P;             // Dezimalpunkt
92      return R;                           // Rest
93  }
```

5.4.1 Deterministische endliche Automaten

Ein deterministischer endlicher Automat besitzt für jede Kombination aus Zustand und Eingabealphabet genau einen Folgezustand. In diesem Skript werden bei den endlichen Automaten nur deterministische Automaten behandelt.

5.4.2 Nichtdeterministische endliche Automaten

Ein nichtdeterministischer endlicher Automat kann für eine Kombination aus Zustand und Eingabealphabet mehrere Folgezustände besitzen. Für die Implementierung von nichtdeterministischen Automaten sind daher Backtracking-Algorithmen notwendig, was die Implementierung nicht gerade vereinfacht.

Für jeden nichtdeterministischen Automaten kann ein äquivalenter deterministischer Automat konstruiert werden.

5.5 Endliche Automaten mit Ausgabe

Die bisher behandelten Automaten verarbeiten eine Eingabe und anschließend wird durch Auswertung der Endezustände eine Entscheidung getroffen. Dieses Verhalten genügt meistens, wenn eine Eingabe gemäß einer Typ-3-Grammatik geprüft werden soll. Sollen aber Text gescannt werden, so muss der gescannte Text zusätzlich ausgegeben werden. Damit dies funktioniert, muss der Automat um eine Ausgabe erweitert werden.

5.5.1 Moore-Automaten

Ein Moore-Automat A ist ein Sechstupel A = (Z, E, δ, λ, z_0, F). Außer der Ausgabefunktion λ ist der Moore-Automat identisch mit einem erkennenden Automaten.

λ ist die Ausgabefunktion die jedem Zustand eine Ausgabe zuordnet. $Z \rightarrow Ausgabealphabet$.

5.5.2 Mealey-Automaten

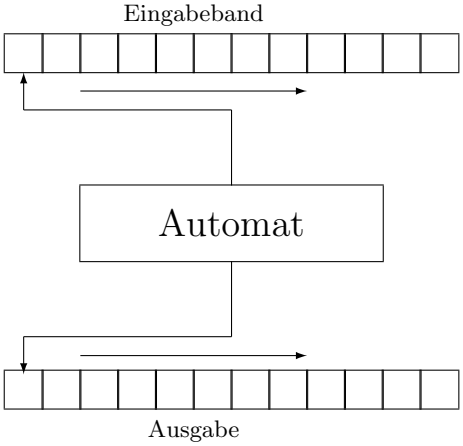
Ein Mealey-Automat A ist ein Sechstupel A = (Z, E, δ, λ, z_0, F). Außer der Ausgabefunktion λ ist der Mealey-Automat identisch mit einem erkennenden Automaten.

λ ist die Ausgabefunktion, die jeder Kombination aus Zustand und Eingabezeichen eine Ausgabe zuordnet. $Z \otimes E \rightarrow Ausgabealphabet$.

Modell eines Moore- oder Mealey-Automaten

Abbildung 5.4 zeigt das Modell eines Moore- bzw Mealey- Automaten.

Abb. 5.4. Modell Moore- und Mealey-Automat

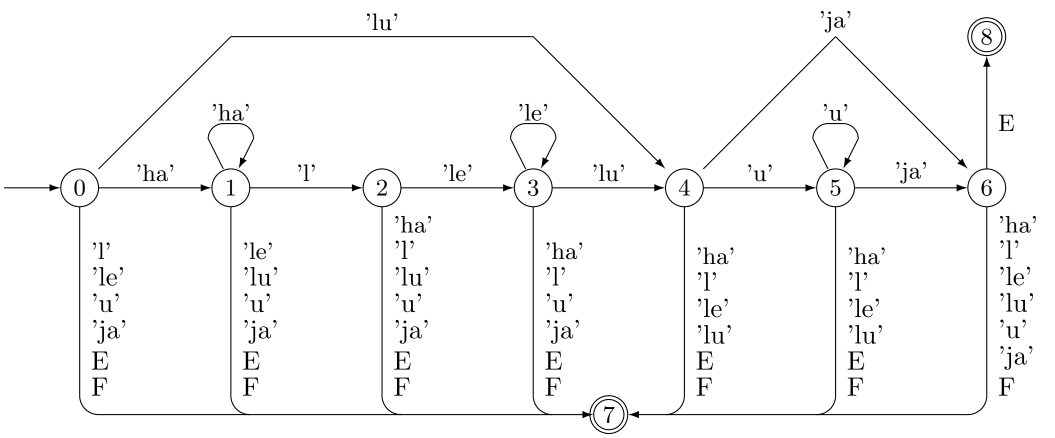


5.6 Anwendungen von endlichen automaten

5.6.1 Erkennen

Ein Münchner im Himmel (1)

Als der Münchener Dienstmann Alois in den Himmel kam, war er so betrunken, dass er nicht einmal mehr frohlockend “Halleluja“ rufen konnte. Daher wurde ihm das “Halleluja“ in Form eines erkennenden Automaten erklärt:



Parsing-Tabelle:

δ	0 E	1 'ha'	2 'l'	3 'le'	4 'lu'	5 'u'	6 'ja'	7 F
0	7	1	7	7	4	7	7	7
1	7	1	2	7	7	7	7	7
2	7	7	7	3	7	7	7	7
3	7	7	7	3	4	7	7	7
4	7	7	7	7	7	5	6	7
5	7	7	7	7	7	5	6	7
6	8	7	7	7	7	7	7	7

Beispiele für korrektes Frohlocken sind etwa “halleluja“, “halleleluja“, “luja“ oder aber auch “hahahalleleleleleluuuuuuuuuuuja“. Falsches Frohlocken wäre etwa “hallllleluja“ oder “haluja“.

Listing 5.2 implementiert einen Scanner für den Aloisius-Automaten. Er ist primitiv und nicht laufzeitoptimiert mit Stringvergleichen aufgebaut. Wegen dem Maximum-Match-Prinzip ist die Reihenfolge der Zeilen 14-16 wichtig, damit beim Auffinden der Zeichenfolge “lu“ das Token lu und nicht das einfache l zurückgegeben wird.

Listing 5.2. Aloisius-Scanner 1 (primitiv) (automaten-aloisius-scanner1.c)

```
1 /********************************************************
2 Scanner 1 für Aloisius-Automat
3 ********************************************************/
4 #include <string.h>
5
6 int aloisius_scanner(char *in, int reset) {
7     static char *p;
```

```
8       enum {token_end, token_ha, token_l, token_le,token_lu,
9             token_u, token_ja, token_fehler} token;
10      if (reset) {
11          p = in;
12          return 0;
13      }
14      if (!strncmp(p, "le", 2)) p+=2, token = token_le;
15      else if (!strncmp(p, "lu", 2)) p+=2, token = token_lu;
16      else if (!strncmp(p, "l", 1)) p+=1, token = token_l;
17      else if (!strncmp(p, "ha", 2)) p+=2, token = token_ha;
18      else if (!strncmp(p, "u", 1)) p+=1, token = token_u;
19      else if (!strncmp(p, "ja", 2)) p+=2, token = token_ja;
20      else if (!strncmp(p, "\0", 1)) token = token_end;
21      else if (!strncmp(p, "\n", 1)) token = token_end;
22      else return token_fehler;
23      return token;
24  }
```

Listing 5.3. Aloisius-Automat (automaten-aloisius.c)

```
1   /*******************************************************
2   Erkennender Automat für das Frohlocken des Münchner
3   Dienstmanns Alois (Engel Aloisius) im Himmel
4   *******************************************************/
5   #include <stdio.h>
6   #include <string.h>
7   #include "automaten-aloisius-scanner1.c"
8   //#include "automaten-aloisius-scanner2.c"
9
10  int aloisius_parser(char *in) {
11      static int ptable[7][8]={/*0*/ {8,1,7,7,4,7,7,7},
12                               /*1*/ {7,1,2,7,7,7,7,7},
13                               /*2*/ {7,7,7,3,7,7,7,7},
14                               /*3*/ {7,7,7,3,4,7,7,7},
15                               /*4*/ {7,7,7,7,7,5,6,7},
16                               /*5*/ {7,7,7,7,7,5,6,7},
17                               /*6*/ {8,7,7,7,7,7,7,7}};
18      int zustand = 0, token;
19      aloisius_scanner(in, 1); // Reset
20
21      while (zustand < 7) {
22          token=aloisius_scanner(in,0);
23  //          printf("Zustand %d Token %d", zustand, token);
24          zustand = ptable[zustand][token];
25  //          printf(" --> %d\n", zustand);
26      }
27      return zustand == 7; // 7 ist Fehlerzustand
28  }
29
30  int main() {
31      char input[255];
32
33      while(printf("Frohlocken: "),fgets(input, 255, stdin)!= NULL)
34          printf("%sOK!\n", (aloisius_parser(input))?"Nicht ":"");
35
```

```
36      printf("\n");
37      return 0;
38  }
```

Bildschirmausgabe:

BCD-Erkenner

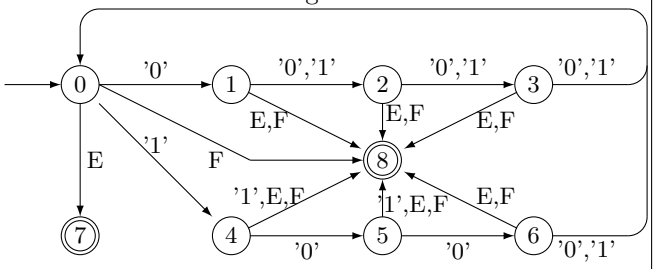
Ein Programm soll erkennen, ob ein eingegebener Text bestehend aus 0 und 1 eine korrekte BCD-Folge darstellt:

BCD	Dez.
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Mögliche Fehler für eine Eingabe sind dabei falsche Zeichen (bspw. “0120“), falsche BCD-Codierung (bsp. “1011“) und falsche Länge (bspw. “10010“).

Ein erkennender Automat könnte folgendermaßen konstruiert werden:

Diagramm



Tabelle

δ	0 E	1 '0'	2 '1'	3 F
0	7	1	4	8
1	8	2	2	8
2	8	3	3	8
3	8	0	0	8
4	8	5	8	8
5	8	6	8	8
6	8	0	0	8

Das Eingabealphabet des Automaten ist “Ende“, “0“, “1“ und “Fehler“ (falsche Zeichen).

Listing 5.4. BCD-Erkenner (automaten-bcd.c)

```
1  /*******************************************************
2  BCD-Erkenner via Automat
3  *******************************************************/
4  #include <stdio.h>
5
6  int bcd_scanner(char *, int);
7  int bcd_parser(char *);
8
```

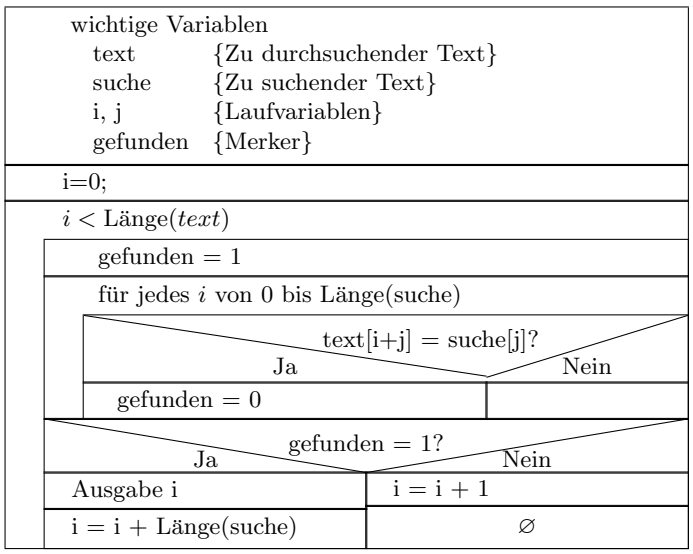
```
 9  int main() {
10      char input[255];
11
12      while(printf("BCD-Input: "),fgets(input, 255, stdin)!= NULL)
13          printf("%sOK!\n", (bcd_parser(input))?"Nicht ":"");
14
15      printf("\n");
16      return 0;
17  }
18
19  int bcd_parser(char *in) {
20      int zustand = 0;
21      static int ptable[7][4] = { /*0*/ {7,1,4,8},
22                                  /*1*/ {8,2,2,8},
23                                  /*2*/ {8,3,3,8},
24                                  /*3*/ {8,0,0,8},
25                                  /*4*/ {8,5,8,8},
26                                  /*5*/ {8,6,8,8},
27                                  /*6*/ {8,0,0,8}};
28      bcd_scanner(in,1); // Scanner-Reset
29      while (zustand < 7)
30          zustand = ptable[zustand][bcd_scanner(in, 0)];
31      return zustand == 8;
32  }
33
34
35  int bcd_scanner(char *in, int reset) {
36      static char *p;
37      enum {token_end, token_0, token_1, token_fehler};
38      if (reset) {
39          p = in;
40          return 0;
41      }
42      switch (*p++) {
43      case '0':   return token_0;
44      case '1':   return token_1;
45      case '\0':
46      case '\n':  return token_end;
47      default:    return token_fehler;
48      }
49  }
```

Bildschirmausgabe:

5.6.2 Suchen

Wird in einem langen Text (z.B. einer Datei auf der Festplatte) nach einer Zeichenkette gesucht, so ist die einfachste Suche der zeichenweise Vergleich des zu durchsuchenden Textes mit dem Suchtext. Dies ist in Abbildung 5.5 dargestellt. Hier muss ein wichtiges Prinzip der Suche angesprochen werden, nämlich die überlappenden Treffer. Sucht man im Text “rororo“ nach der Zeichenfolge “roro“ so ergibt sich nur ein Treffer in den Stellen 1 bis 4. Durch diesen Treffer darf erst ab Stelle 5 weitergesucht werden und der theoretisch denkbare Treffer in den Stellen 3 bis 6 darf nicht erkannt werden.

Abb. 5.5. Einfache Textsuche



An den zwei ineinander geschachtelten Schleifen erkennt man, dass das Laufzeitverhalten des Algorithmus etwa proportional zu Länge(*text*) ∗ Länge(*suche*) ist.

Sollen mehrere Begriffe gleichzeitig gesucht werden, so wird die Suche entsprechend noch langsamer. Wenn aber nach einem regulären Ausdruck gesucht wird (so dass es evtl. unendlich viele Suchbegriffe gibt) kann dieser Einfachalgorithmus keine Lösung sein.

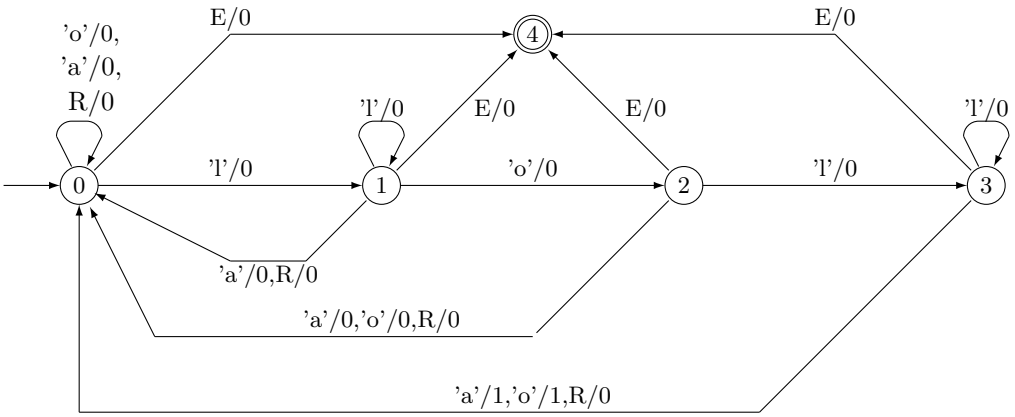
Listing 5.5. Textsuche - Einfachstversion (automaten-textsuche.c)

```
1  /*********************************************************
2  Suche in einem Text nach einem fixen Subtext
3  Brachialmethode
4  *********************************************************/
5  #include <stdio.h>
6
7  int main() {
8      char text[]=
9  "als lola die laolawelle sah sang sie olola, lollolarossosalat";
10     char suche[] = "lola";
11     int suchlang, textlang, i, j, gefunden, anz;
12
13     suchlang = 0;
14     while (suche[suchlang] != '\0')
15         suchlang++;
16
17     textlang = 0;
18     while (text[textlang] != '\0')
19         textlang++;
20
21     anz = textlang - suchlang + 1;
22     i = 0;
23     while (i < anz) {
24         gefunden = 1;
```

```
25          for (j = 0; j < suchlang; j++)
26              if (text[i + j] != suche[j])
27                  gefunden = 0;
28          if (gefunden) {
29              printf("Gefunden ab Stelle %d\n", i);
30              i += suchlang;
31          }
32          else
33              i++;
34      }
35  }
```

Durch Konstruktion endlicher Automaten wird die Laufzeit der Suche proportional zur Länge des zu durchsuchenden Textes unabhängig von Länge oder Komplexität des zu suchenden Textes (regulären Ausdrucks)!

In einem Text soll der reguläre Ausdruck `"lol""l"*("o"|"a")` gesucht werden. Dazu wird ein Mealey-Automat konstruiert, der immer nach Erkennen des regulären Ausdrucks eine Ausgabe macht:



Alternativ die beiden Tabellen:

δ	E 0	'l' 1	'o' 2	'a' 3	R 4
0	4	1	0	0	0
1	4	1	2	0	0
2	4	3	0	0	0
3	4	3	0	0	0

λ	E 0	'l' 1	'o' 2	'a' 3	R 4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	1	1	0

Verfolgen Sie den Automat anhand des Eingabetextes

```
als lola die laolawelle sah sang sie olala, lollorossosalat
```

Zeichen		a	l	s		l	o	l	a		d	i	e	...
Zustand	0	0	1	0	0	1	2	3	0	0	0	0	0	...
Ausgabe									1					...

Listing 5.6. Lola-Automat (automaten-lola.c)

```
1  /*******************************************************
2  Automat sucht in einem Text den regulären Ausdruck
```

```
3  "lol""l"*("o"|"a")
4  Bsp. lola, lollla, lollo
5  ************************************************************/
6  #include <stdio.h>
7  int lola_parser(char * in) {
8      int ctyp[256] = {
9  0,4,4,4,4,4,4,4,4,4,0,4,4,0,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
10 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
11 4,3,4,4,4,4,4,4,4,4,4,4,4,1,4,4,2,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
12 4,3,4,4,4,4,4,4,4,4,4,4,4,1,4,4,2,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
13 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
14 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
15 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
16 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4};
17 /*  ctyp['\0']=ctyp['\n']=ctyp['\r']=0
18     ctyp['L']=ctyp['l']=1
19     ctyp['O']=ctyp['o']=2
20     ctyp['A']=ctyp['a']=3
21     alle anderen Indizes x: ctyp[x] = 4 */
22     int delta[4][5] = {/*  E  l  o  a  R */
23                        /*0*/ {4, 1, 0, 0, 0},
24                        /*1*/ {4, 1, 2, 0, 0},
25                        /*2*/ {4, 3, 0, 0, 0},
26                        /*3*/ {4, 3, 0, 0, 0}};
27     int lambda[4][5] = {/* E  l  o  a  R */
28                        /*0*/ {0, 0, 0, 0, 0},
29                        /*1*/ {0, 0, 0, 0, 0},
30                        /*2*/ {0, 0, 0, 0, 0},
31                        /*3*/ {0, 0, 1, 1, 0}};
32     int zustand = 0, pos = -1, token;
33     while (zustand != 4) {
34         token = ctyp[*(in + ++pos)];
35 //        printf("Zustand %d, Token %d ", zustand, token);
36         if (lambda[zustand][token])
37             printf("Treffer, Ende an Stelle %d\n", pos);
38         zustand = delta[zustand][token];
39 //        printf("--> wechsle in Zustand %d\n", zustand);
40     }
41 }
42
43 int main() {
44     char text[256] =
45 "als Lola die Laolawelle sah sang sie olala, Lllollorossosalat";
46
47 /*    printf("Geben Sie einen Text ein:\n");
48     fgets(text, 255, stdin); */
49     printf("Durchsuche\n\"%s\"\n", text);
50     printf("nach \"lol\"\"l\"*(\"a\"|\"o\"):\n\n");
51     lola_parser(text);
52     return 1;
53 }
```

Innerhalb des Parsers wird eine Tabelle `ctyp` benutzt, die allen ASCII-Zeichen von 0...255 einen der Werte 0...4 (Eingabealphabet des Parsers) zuordnet:

$$ctyp[x] = \begin{cases} 0 & \text{für } x \in \{\texttt{'\textbackslash 0'}, \texttt{'\textbackslash n'}, \texttt{'\textbackslash r'}\} \\ 1 & \text{für } x \in \{\texttt{'L'}, \texttt{'l'}\} \\ 2 & \text{für } x \in \{\texttt{'O'}, \texttt{'o'}\} \\ 3 & \text{für } x \in \{\texttt{'A'}, \texttt{'a'}\} \\ 4 & \text{sonst} \end{cases}$$

Beim Erstellen solcher Tabellen ist die ASCII-Tabelle (S. 139) hilfreich!

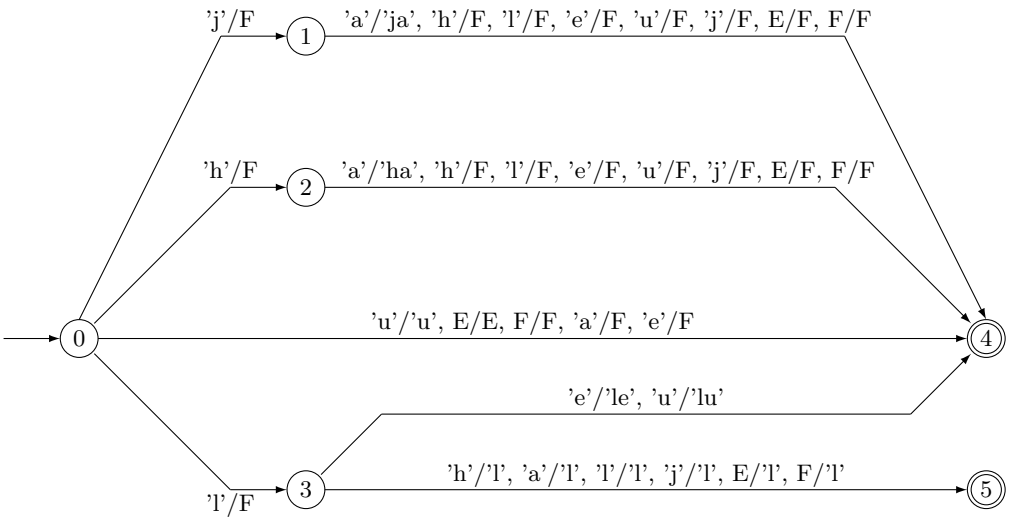
Bildschirmausgabe:

5.6.3 Scannen

Ein Frohlocken-Scanner

Der primitive Scanner des Aloisius-Automaten (Listing 5.2 Seite 47) vergleicht die Eingabe mithilfe des C-Stringvergleichs. Für längere Texte oder kompliziertere Textsuchen wäre dies sehr ineffizient. Es soll daher ein Mealey-Automat vorgestellt werden, der mit linearem Laufzeitverhalten die Eingabe scannt.

Abb. 5.6. Aloisius-Scanner als Mealey-Automat



Im Endezustand 5 muss ein Look-Ahead rückgängig gemacht werden, im Endezustand 4 dagegen nicht! Die Ausgabe wird jeweils gemerkt und nach Erreichen eines Endezustands wird die letzte gemerkte Ausgabe als Returnwert zurückgegeben.

δ	0 E	1 'h'	2 'a'	3 'l'	4 'e'	5 'u'	6 'j'	7 F
0	4	2	4	3	4	4	1	4
1	4	4	4	4	4	4	4	4
2	4	4	4	4	4	4	4	4
3	5	5	5	5	4	4	5	5

λ	0 E	1 'h'	2 'a'	3 'l'	4 'e'	5 'u'	6 'j'	7 F
0	E		F		F	'u'		F
1	F	F	'ja'	F	F	F	F	F
2	F	F	'ha'	F	F	F	F	F
3	'l'	'l'	'l'	'l'	'le'	'lu'	'l'	'l'

Text	Code
Ende	0
“ha“	1
“l“	2
“le“	3
“lu“	4
“u“	5
“ja“	6
Fehler	7

Listing 5.7. Aloisius-Scanner 2 (schnell) (automaten-aloisius-scanner2.c)

```
1  /*********************************************************
2  Scanner 1 für Aloisius-Automat
3  Ausgabealphabet:
4  Ende      : 0
5  "ha"      : 1
6  "l"       : 2
7  "le"      : 3
8  "lu"      : 4
9  "u"       : 5
10 "ja"      : 6
11 Fehler    : 7
12 *********************************************************/
13 #include <string.h>
14
15 int aloisius_scanner(char *in, int reset) {
16     static char *p;
17     static int delta[4][8]={/*    E h a l e u j F  */
18                             /*0*/ {4,2,4,3,4,4,1,4},
19                             /*1*/ {4,4,4,4,4,4,4,4},
20                             /*2*/ {4,4,4,4,4,4,4,4},
21                             /*3*/ {5,5,5,5,4,4,5,5}};
22     static int lambda[4][8]={/*  E h a l e u j F  */
23                             /*0*/ {0,7,7,7,7,5,7,7},
24                             /*1*/ {7,7,6,7,7,7,7,7},
25                             /*2*/ {7,7,1,7,7,7,7,7},
26                             /*3*/ {2,2,2,2,3,4,1,1}};
27     int chartyp[256] = {
28 0,7,7,7,7,7,7,7,7,7,0,7,7,0,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
29 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
30 7,2,7,7,7,4,7,7,1,7,6,7,3,7,7,2,7,7,7,7,7,5,7,7,7,7,7,7,7,7,7,7,
31 7,2,7,7,7,4,7,7,1,7,6,7,3,7,7,2,7,7,7,7,7,5,7,7,7,7,7,7,7,7,7,7,
32 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
33 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
34 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
35 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7};
36     int zustand = 0, token, returnval;
37     if (reset) {
38         p = in;
39         return 0;
```

```
40      }
41      while (zustand < 4) {
42          token=chartyp[*p++];
43          returnval = lambda[zustand][token];
44          zustand = delta[zustand][token];
45      }
46      if (zustand == 5) // Lookahead rückgängig machen
47          p--;
48      return returnval;
49  }
```

Internet-Suchmaschine

Syntaktische Analyse von mathematischen Ausdrücken

Zur Analyse eines mathematischen Ausdrucks benötigen wir eine lexikalische Analyse, welche uns folgendes Alphabet ausgibt:

Ende

\+ (Addition oder positives Vorzeichen)

\- (Subtraktion oder negatives Vorzeichen)

* (Multiplikation)

/ (Division)

((Öffnende Klammer)

) (Schließende Klammer)

Zahl

Fehler

Der zuständige Automat muss dabei vorausschauend operieren (Look-Ahead). Er liest solange ein, bis die Eingabe kein gültiges Wort mehr ist. Danach muss das letzte gelesene Zeichen in die Eingabe zurückgestellt werden.

Als Endezeichen sollten die Zeichen `\r`, `\n`, und `\0` genutzt werden.

Um für die lexikalische Analyse - die als DFA implementiert wird - nicht noch eine eigene lexikalische Analyse zu schreiben, werden hier (ausnahmsweise) Scanner und Parser als eine Einheit implementiert.

Eingabealphabet des Parsers:

E Ende

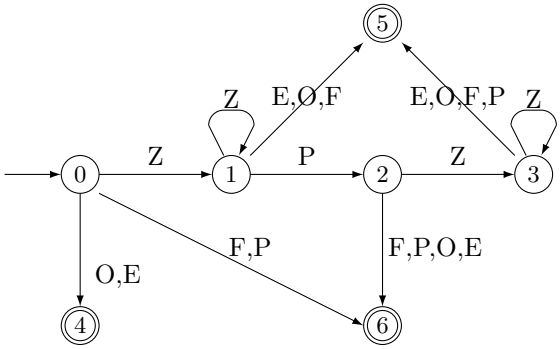
O Operator (“+“, “-“, “*“, “/“, “(“ und “)“)

Z Ziffer (0-9)

P Dezimalpunkt (.)

F Fehler (Falsche Zeichen)

Diagramm



Tabelle

δ	E	O	P	Z	F
0	4	4	6	1	6
1	5	5	2	1	5
2	6	6	6	3	6
3	5	5	5	3	5

Endezustand 4 bedeutet dass ein Operator oder das Ende gefunden wurde.

Endezustand 5 bedeutet dass eine Fixkommazahl gefunden wurde. In diesem Fall wurde aber ein Zeichen zuviel gelesen (Look-Ahead), welches quasi in die Eingabe zurückgegeben werden muss.

Endezustand 6 bedeutet dass ein Fehler aufgetreten ist.

Listing 5.8. Scanner für arithmetische Ausdrücke (term-scanner.c)

```
1  /*******************************************************
2  Lexikalische Analyse für mathematische Ausdrücke die aus
3  Zahlen (Fixkomma), Klammern und den Operatoren + - * /
4  bestehen.
5
6  Zum Reset des Scanners einen String als ersten Parameter
7  übergeben.
8  Zum Scannen einen NULL-Zeiger übergeben.
9  *******************************************************/
10 #include <stdlib.h>
11 #include <string.h>
12 #include <malloc.h>
13 #include "term-scanner.h"
14
15 char yytext[100];
16 int yylex() {
17     return term_scanner(NULL, yytext);
18 }
19
20 int term_scanner(char *input,char *text) {
21
22     static char *myinput = NULL;
23     static int delta[][6] = {    /*      E O . Z W R  */
24                                  /*0*/ {4,4,6,1,0,6},
25                                  /*1*/ {5,5,2,1,5,5},
26                                  /*2*/ {6,6,6,3,6,6},
27                                  /*3*/ {5,5,5,3,5,5}};
28     static int lambda[][6] = {   /*      E O . Z W R  */
```

```
29                                      /*0*/ {1,1,1,1,0,1},
30                                      /*1*/ {0,0,1,1,0,0},
31                                      /*2*/ {0,1,1,1,1,1},
32                                      /*3*/ {0,0,0,1,0,0}};
33      int zustand = 0, rc;
34      static int pos, token, klassentab[256] = {
35  0,5,5,5,5,5,5,5,5,5,0,5,5,0,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
36  4,5,5,5,5,5,5,5,1,1,1,1,5,1,2,1,3,3,3,3,3,3,3,3,3,5,5,5,5,5,5,5,
37  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
38  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
39  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
40  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
41  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
42  5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5};
43      if (input != NULL) { // Reset
44          if (myinput != NULL)
45              free(myinput);
46          myinput = (char *) malloc(sizeof(char) * (strlen(input) + 1));
47          if (myinput == NULL)
48              return -1; // Kein Memory mehr
49          strcpy(myinput, input);
50          pos = 0;
51          return 0;
52      }
53      if (myinput == NULL) // Kein Reset oder kein Hauptspeicjer
54          return -2;
55      do {
56          token = klassentab[myinput[pos]];
57  //rintf("token=%d\n", token);
58          if (lambda[zustand][token])
59              *text++=myinput[pos];
60          zustand = delta[zustand][token];
61          pos++;
62      } while (zustand < 4);
63      *text = 0;  // Strinende schreiben
64      if (zustand == 5) { // Zahl gefunden
65          pos--;              // Look-ahead rückgaängig machen
66          rc = ZAHL;
67      }
68      else if (zustand == 4) { // Operator oder Ende
69          switch (*(text-1)) {
70              case '+':   rc = PLUS; break;
71              case '-':   rc = MINUS;; break;
72              case '*':   rc = MAL;; break;
73              case '/':   rc = DIV;; break;
74              case '(':   rc = KLA_AUF;; break;
75              case ')':   rc = KLA_ZU;; break;
76              case '\n':
77              case '\0':  rc = END;; break;
78          }
79      }
80      else
81          rc = FEHLER;
82      return rc;
```

```
83 }
```

Listing 5.9. Header-Datei zum Term-Scanner (term-scanner.h)

```
1 #ifndef __TERM_SCANNER__
2 #define __TERM_SCANNER__ 1
3 //     0     1     2     3     4     5          6        7      8
4 enum {END, PLUS, MINUS, MAL, DIV, KLA_AUF, KLA_ZU, ZAHL, FEHLER};
5
6 int term_scanner(char *input,char *text);
7
8 #endif
```

Zum Test der lexikalischen Analyse dient folgendes Testprogramm:

Listing 5.10. Testprogramm für Scanner (term-scanner-test.c)

```
1  /*******************************************************
2  Testprogramm für Mathe-Scanner
3  *******************************************************/
4  #include <stdio.h>
5  #include "term-scanner.h"
6
7  int main() {
8      char t[255], z[100];
9      int token;
10
11     while (printf("Input: "),fgets(t, 255, stdin) != NULL) {
12         term_scanner(t, NULL);
13         while ((token = term_scanner(NULL, z)) != END)
14             printf("%d '%s'\n", token, z);
15     }
16     printf("\n");
17     return 0;
18 }
```

Eine Syntaktische Analyse sowie ein Rechenprogramm, das diesen Scanner nutzt, wird im Kapitel 6 vorgestellt werden.

5.7 Übungen

Übung 5.1. Fixkommazahl-Erkennung
Erstellen Sie einen Automaten, der Fixkommazahlen mit Vorzeichen und optionalem Vorkommateil akzeptiert (bspw. Akzeptieren von “-2“, “+.5“, “2.9“, Ablehnen von “2.“, “1.2+“).

Übung 5.2. Fließkommazahl-Erkennung
Erstellen Sie einen Automaten, der Fließkommazahlen mit Vorzeichen und optionalem Vorkommateil und optionalem 10er-Exponenten (ganzzahlig) akzeptiert (bspw. Akzeptieren von “-2E-4“, “+.5E1“, “2.9“, Ablehnen von “E-2“, “1.2E2.5“).

Übung 5.3. EBNF-Darstellung mit Scanner
Unter Nutzung eines Scanners kann die Grammatik der EBNF-Form einfacher geschrieben werden:

Start = *syntax*
syntax = {*produktion*} .
produktion = *bezeichner* “=“ *ausdruck*.
ausdruck = *term* {“|“ *term*} .
term = *faktor* {*faktor*} .
faktor = “*bezeichner*“ | “*string*“ | “(“*ausdruck*“)“ | “[“*ausdruck*“]“ | “{“*ausdruck*“}“.

Erstellen Sie einen Scanner, der die Terminalsymbole

$$T = \{\text{“(“, “)“, “[“, “]“, “{“, “}“, “=“, “.“, “|“, “bezeichner“, “string“}}$$

dieser Grammatik scannt. Testen Sie den Scanner mit der Grammatik

```
ausdruck    = term {("+"|"-") term}.
term        = faktor {("*"|"/") faktor}.
faktor      = zahl | "(" ausdruck ")".
zahl        = ziffer {ziffer}.
ziffer      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Übung 5.4. Sprachsteuerung
Das Kino “Informatikon“ hat zwei Kinos. Im Kino A gibt es Plätze zu 10 Euro, 14 Euro und 20 Euro. Im Kino B gibt es Plätze zu 10 Euro und 16 Euro.

Der Film “Von Bits und Bytes“ läuft im Kino A um 14:00 Uhr, 18:00 Uhr und 22:00 Uhr sowie im Kino B um 15:00 Uhr. Der Film “Digitale Virologie“ läuft im Kino A um 20:00 Uhr und im Kino B um 18:00 Uhr.

Entwickeln Sie für ein telefonisches Spracherkennungssystem - das als Ausgabe die Ziffern 0-9 sowie “E“ (Ende) und “F“ (Fehler) hat - einen Automaten, der die Buchung vornimmt. Die Eingabe des Automaten soll über Tastatur simuliert werden.

Übung 5.5. Aloisius 3
Ändern Sie den erkennenden Aloisius-Automaten in einen Moore-Automaten mit nur einem Endezustand ab.

Übung 5.6. Aloisius 4
Ändern Sie Grammatik / Automat so ab, dass mehrere “Halleluja“ mit ein oder mehreren Maß Bier zwischen den einzelnen “Halleluja“s möglich sind.

Übung 5.7. Aloisius 1
Erstellen Sie die Typ-3-Grammatik zum Frohlocken in EBNF

Übung 5.8. Aloisius 2
Erstellen Sie die Typ-3-Grammatik zum Frohlocken als Syntax-Graphen

Übung 5.9. Aloisius 5
Erstellen Sie einen erkennenden Automaten als Scanner für den Aloisius-Parser. Dieser Automat hat für jedes Zeichen des Parser-Eingabealphabets einen Endezustand. Durch geschickte Wahl der Zustandsnummern kann aus dem Endezustand direkt das Zeichen des Ausgabealphabets abgeleitet werden.

Übung 5.10. BCD 1
Erstellen Sie die Grammatik für BCD-Zahlen in EBNF.

Übung 5.11. BCD 2
Erstellen Sie die Grammatik für BCD-Zahlen als Syntax-Graph.

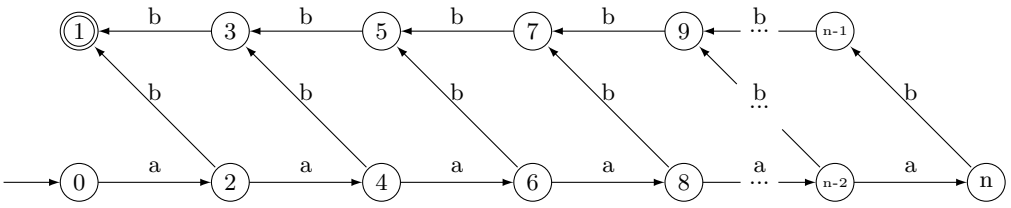
6

Keller-Automaten

6.1 Einführung

Die Sprache $L = a^n b^n$ produziert Sätze $\epsilon, ab, aabb, aaabbb, \ldots$. Soll diese Sprache durch einen endlichen Automaten erkannt werden, so müsste dieser Automat unendlich viele Zustände haben, wie Abbildung fig endlicher automat an bn zeigt.

Abb. 6.1. Endlicher Automat für $L = a^n b^n$



Kellerautomaten haben im Gegensatz zu den einfachen Automaten einen Speicher in Form eines Kellers (Stacks). Auf dem Stack wird der Zustand des Automaten gespeichert. Dadurch kann der Automat - wenn er in einen anderen Zustand geht - sich seine alten Zustände "merken". Abbildung 6.2 zeigt das Modell eines Kellerautomaten.

Ein Kellerautomat kennt zum Speichern die drei Aktionen

push Legt ein Element auf oben auf den Stack

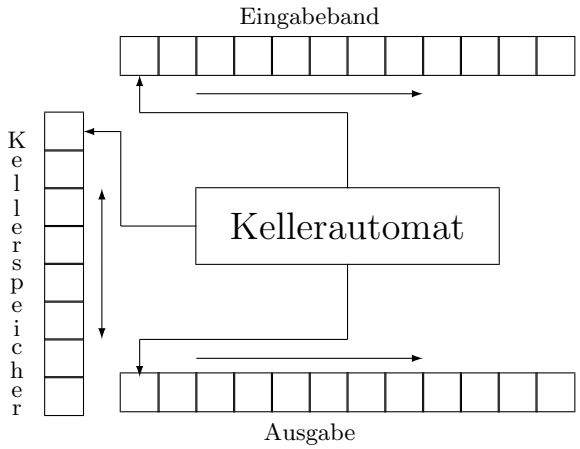
pop Holt ein Element löschend oben vom Stack

tos Holt ein Element nichtlöschend oben vom Stack

Abhängig vom anliegenden Eingabezeichen und dem obersten Stackelement kann nun tabellarisch das Verhalten des Automaten bestimmt werden.

	END	a	b
END	ACCEPT	push(a);read()	ERROR
a	ERROR	push(a);read()	pop();read()

Abb. 6.2. Modell eines Kellerautomaten



6.2 Native Konstruktion

Es soll der folgende Ausdruck ausgewertet werden:

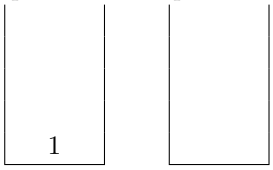
```
1+2*3
```

Die Token-Folge des Scanners lautet damit `ZAHL`, `PLUS`, `ZAHL`, `MAL`, `ZAHL`, `END`.

Es werden zwei Kellerspeicher angelegt, einer für die Zahlen (Operanden), einer für die Operatoren.

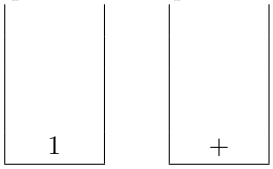
1. Als erstes wird die Zahl 1 eingelesen und auf dem Operandenkeller gespeichert:

   Operanden Operatoren

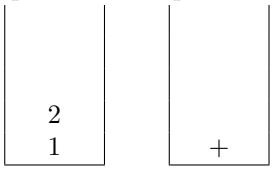
2. Jetzt wird der Operator '+' eingelesen und auf dem Operatorenkeller gespeichert:

   Operanden Operatoren

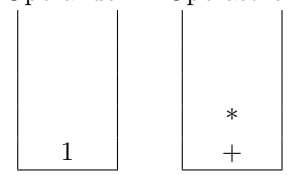
   

3. Jetzt wird die Zahl 2 eingelesen und auf dem Operandenkeller gespeichert:

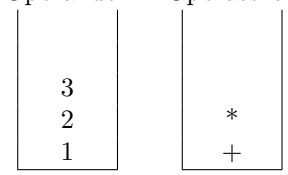
   Operanden Operatoren

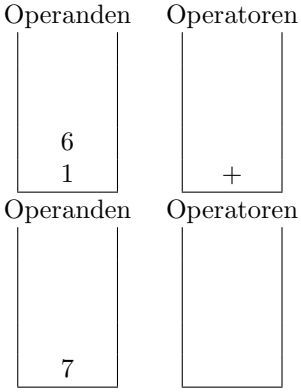
4. Jetzt wird der Operator '*' eingelesen und auf dem Operatorenkeller gespeichert:



5. Jetzt wird die Zahl 3 eingelesen und auf dem Operandenkeller gespeichert:



6. Jetzt wird das Ende-Token eingelesen. Der Keller wird abgearbeitet indem jeweils die zwei obersten Operanden gelesen werden, mit dem obersten Operator verknüpft werden und das Ergebnis auf dem Operandenkeller gespeichert wird:



7. Am Ende ist der Operatorkeller leer und der Operandenkeller enthält eine Zahl - das Ergebnis 7 des Ausdrucks `1+2*3`!

Wird also eine Zahl gefunden so wird sie (immer) auf dem Operatorkeller gespeichert. Wird dagegen ein Operand gefunden so wird eine Aktion durchgeführt die abhängig von dem gefundenen Operator und dem obersten im Operandenkeller gespeicherten Operator ist:

	Token						
	E	+	-	*	/	(	)
E	X	K	K	K	K	K	F
+	D	G	G	K	K	K	D
-	D	G	G	K	K	K	D
*	D	D	D	G	G	K	D
/	D	D	D	G	G	K	D
(	F	K	K	K	K	K	L

Dabei bedutet

X Exit, fertig

K Operator abkellern

G Kelleroperation durchführen, neuen Operator abkellern

D Kelleroperation durchführen (evtl. wiederholt)

L Ein Kellerelement löschen

F Fehler

Listing 6.1. Kellerautomat zur Berechnung arithmetischer Ausdrücke (keller-arith1.c)

```
1  /*********************************************************
2  Rechenprogramm für arithmetische Ausdrücke
3  Kann Grundrechenarten mit Punkt-Vor-Strich und Klammern
4  Kann KEINE Vorzeichen!!!!!
5  *********************************************************/
6  #include <stdio.h>
7  #include "automaten-mathe-scanner.c"
8
9  #define STACK_HEIGHT 10
10
11 int kellerautomat(char *, double *);
12 int keller_r(double, double, int, double *);
13
14 int kellerautomat(char *input, double *z) {
15     double zk[STACK_HEIGHT];          // Zahlenkeller
16     int ok[STACK_HEIGHT];             // Operandenkeller
17     int oh = -1, zh = -1;   // Operandenhöhe, Zahlenhöhe
18     double z1, z2, z3;
19     int token, error = 0;
20     char aktion, ptable[6][7]={
21 /*                 END  +   -   *   /   (   )   */
22 /* END      */  {'X', 'K', 'K', 'K', 'K', 'K', 'F'},
23 /* +        */  {'D', 'G', 'G', 'K', 'K', 'K', 'D'},
24 /* -        */  {'D', 'G', 'G', 'K', 'K', 'K', 'D'},
25 /* *        */  {'D', 'D', 'D', 'G', 'G', 'K', 'D'},
26 /* /        */  {'D', 'D', 'D', 'G', 'G', 'K', 'D'},
27 /* (        */  {'F', 'K', 'K', 'K', 'K', 'K', 'L'}};
28
29     mathescanner(input, NULL); // Scanner-Reset
30     ok[++oh] = END;
31     do {
32         token = mathescanner(input, &z1);
33         if (token == ZAHL) // Zahl abkellern
34             zk[++zh] = z1;
35         else if (token == FEHLER)
36             error = 3;
37         else // Operand
38             do {
39                 aktion = ptable[ok[oh]][token];
40                 switch (aktion) {
41                 case 'K':   // Operator kellern
42                     ok[++oh] = token;
43                     break;
44                 case 'G':   // Gleichrangige Operation
45                     z2 = zk[zh--];
46                     z1 = zk[zh--];
47                     error = keller_r(z1, z2, ok[oh--], &z3);
48                     if (!error) {
49                         zk[++zh] = z3;     // Ergebnis auf Stack
50                         ok[++oh] = token; // Neuer Operand
51                     }
52                     break;
53                 case 'D':   // Keller abarbeiten
```

```
54                      z2 = zk[zh--];
55                      z1 = zk[zh--];
56                      error = keller_r(z1, z2, ok[oh--], &z3);
57                      if (!error)
58                          zk[++zh] = z3;     // Ergebnis auf Stack
59                      break;
60                  case 'L':   // Gekellerte Klammer löschen
61                      oh--;
62                      break;
63                  case 'F':   // Gekellerte Klammer löschen
64                      error = 1;
65                      break;
66                  }
67              } while (!error && aktion == 'D');
68      } while (!error && token != END);
69      if (zh != 0)
70          error = 2;
71      if (!error)
72          *z = zk[0];
73      return error;
74  }
75
76  int keller_r(double z1, double z2, int operator, double *z3) {
77      switch (operator) {
78      case PLUS:  *z3 = z1 + z2; return 0;
79      case MINUS: *z3 = z1 - z2; return 0;
80      case MAL:   *z3 = z1 * z2; return 0;
81      case DIV:    if (z2 == 0)   return 1;
82                  *z3 = z1 / z2; return 0;
83      }
84  }
85
86  int main() {
87      char t[256];
88      int retcode;
89      double z;
90      while (printf("Rechnung: "),fgets(t, 255, stdin) != NULL)
91          if ((retcode = kellerautomat(t, &z)) == 0)
92              printf("Ergebnis: %lf\n\n", z);
93          else
94              printf("Fehler %d\n\n", retcode);
95      printf("\n");
96      return 0;
97  }
```

6.3 LL-Parsing

Im folgenden werden Parsing-Algorithmen entwickelt, die einen Syntaxbaum beginnend von der Wurzel des Baumes entwickeln. Dies nennt man Top-Down-Parsing.

6.3.1 Einführung

Betrachten wir Grammatik Γ_{14}: **Grammatik Γ_{14}: LL(1)-Grammatik**

S → **A** | **B** | **C**
A → **[D] E**
B → **{F} G**
C → **p** | **q**
D → **r s**
E → **{t | u} v**
F → **[w] x**
G → **y z**

Nun wollen wir den Syntaxbaum für den Satz "rstuuuv" entwickeln. Warum geht das so einfach?

$$\begin{array}{lcl}
FIRST(S) = FIRST(A) \cup FIRST(B) \cup FIRST(C) & = & \{p,q,r,t,u,v,w,x,y\} \\
FIRST(A) = FIRST(D) \cup FIRST(E) & = & \{r,t,u,v\} \\
FIRST(B) = FIRST(F) \cup FIRST(G) & = & \{w,x,y\} \\
FIRST(C) & = & \{p,q\} \\
FIRST(D) & = & \{r\} \\
FIRST(E) & = & \{t,u,v\} \\
FIRST(F) & = & \{w,x\} \\
FIRST(G) & = & \{y\} \\
FIRST(G) & = & \{y\}
\end{array}$$

$$\begin{array}{lcl}
FIRST(a) & = & \{a\} \quad a \text{ ist Terminal!} \\
FIRST(AB) & = & FIRST(A) \quad \text{falls A nicht nach } \epsilon \text{ ableitbar} \\
FIRST(AB) & = & FIRST(A) \cup FIRST(B) \quad \text{falls A nach } \epsilon \text{ ableitbar} \\
FIRST(\{A\}B) & = & FIRST(A) \cup FIRST(B) \\
FIRST([A]B) & = & FIRST(A) \cup FIRST(B) \\
FIRST(A|B) & = & FIRST(A) \cup FIRST(B)
\end{array}$$

$FIRST(S) = FIRST(A) \cup FIRST(B) \cup FIRST(C) = \{p,q,r,t,u,v,w,x,y\}$
$FIRST(A) = FIRST(D) \cup FIRST(E) = \{r,t,u,v\}$
$FIRST(B) = FIRST(F) \cup FIRST(G) = \{w,x,y\}$
$FIRST(C) = \{p,q\}$
$FIRST(D) = \{r\}$
$FIRST(E) = \{t,u,v\}$
$FIRST(F) = \{w,x\}$
$FIRST(G) = \{y\}$

Es folgt die Überprüfung der LL(1)-Bedingungen:

Regel 0: $FIRST(A) \cap FIRST(B) = \{\} \wedge FIRST(A) \cap FIRST(C) = \{\} \wedge FIRST(B) \cap FIRST(C) = \{\}$
Regel 1: $FIRST(D) \cap FIRST(E) = \{\}$
Regel 2: $FIRST(F) \cap FIRST(G) = \{\}$

Eine kontextfreie Grammatik heißt LL(1)-Grammatik, wenn zusätzlich zur Kontextfreiheit die folgenden Bedingungen erfüllt sind:

- Für alle Produktionen mit Alternativen $(A \to \sigma_1|\sigma_2|\ldots|\sigma_n)$ muß gelten

$$FIRST(\sigma_i) \cap FIRST(\sigma_j) = \{\} \text{ für alle } i,j \text{ mit } i \neq j$$

- Für alle Produktionen die sich auf den Leerstring ableiten lassen $(A \to \epsilon)$ muß gelten

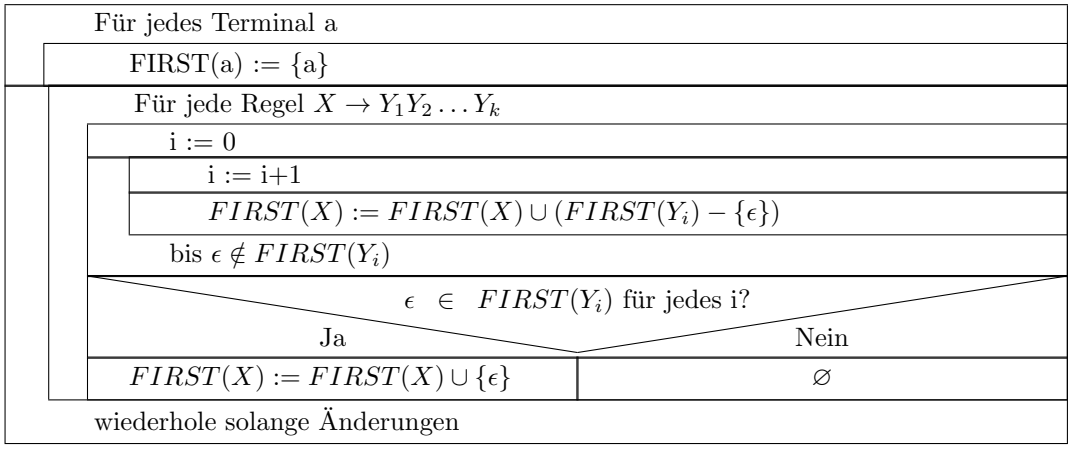
$$FIRST(A) \cap FOLLOW(A) = \{\}$$

Der Name LL(1) bedeutet “Left-to-right-scanning, leftmost derivation, look-ahead 1 token“.

6.3.2 FIRST und FOLLOW

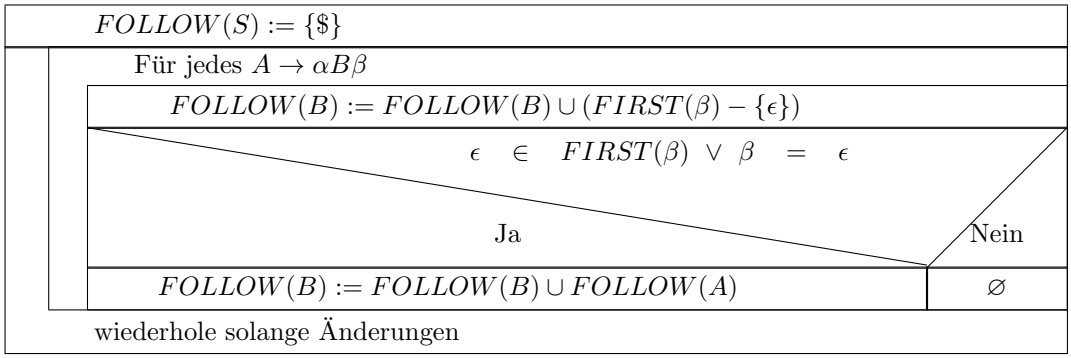
Algorithmus 4: Berechnung der FIRST-Mengen

Berechnung der FIRST-Mengen



Algorithmus 5: Berechnung der FOLLOW-Mengen

Berechnung der FOLLOW-Mengen



Definition 6.1 (FIRST). *Unter der Menge $FIRST(A)$eines Nicht-Terminals A versteht man die Menge aller Terminal-Symbole σ, die zu Beginn eines Textes stehen können, der aus A produziert wird.*

Definition 6.2 (FOLLOW). *Unter der Menge $FOLLOW(A)$ eines Nicht-Terminals A versteht man die Menge aller Terminal-Symbole σ, die unmittelbar nach einem Text stehen können, der aus A produziert wird.*

6.3.3 Programm-gesteuertes LL(1)-Parsing

Die einfachste möglichkeit, einen LL(1)-Parser zu entwickeln, ist die Methode des rekursiven Abstiegs (engl. “recursive descent“). Siehe auch [Wir86], Seite 24ff., [Wir96], Seite 16ff, [Sed92], S. 361ff.

Voraussetzung für einen Recursive-descent-Parser ist eine LL(1)-Grammatik. Für eine solche Grammatik kann ein Parser folgendermaßen konstruiert werden:

1. Eine globale Variable token dient als Look-Ahead-Variable
2. Eine allgemeine Fehlerfunktion void error() wird im Fehlerfall aufgerufen.
3. Es existiert eine Funktion void get_token() die das jeweils nächste Token liest (Scanner-Aufruf) und in der globalen Variablen token speichert.
4. für jeder Produktionsregel der Form $A \rightarrow \beta$ wird eine Funktion void f_A() codiert.
5. Nichtterminale Symbole X werden dadurch verarbeitet, indem ihre Funktion void f_X() aufgerufen wird.
6. Terminale Symbole werden dadurch verarbeitet, indem das aktuelle Token mit dem erwarteteten Token verglichen wird. Im Gleichheitsfall wird get_token() aufgerufen, andernfalls error();
7. Sequenzen werden durch sequentielle Codierung der Symbole abgearbeitet.
8. Alternativen werden durch mehrfache if-else-Verzweigungen oder durch switch-case-Statements codiert. Eventuell wird error() aufgerufen.
9. Optionen werden durch ein einfaches if ohne else codiert.
10. Wiederholungen werden durch kopfgesteuerte Schleifen codiert.

Betrachten wir Grammatik Γ_{15}:

Grammatik Γ_{15}: Term-Grammatik in EBNF
E → T {('+'|'-') T}
T → F {('*'|'/') F}
F → '-' F | '(' E ')' | id

Listing 6.2. Programmgesteuerter LL(1)-Parser (ll1-programm.c)

```
1  /*********************************************************
2  *********************************************************/
3  #include <stdio.h>
4  #include "mathe-scanner.h"
5
6  void f_error();
7  void f_next_token();
8  void f_calculation();
9  void f_factor();
10 void f_expression();
11 void f_term();
12
13 int token, error;
14 char text[255];
15
16 void f_error() {
17     error = 1;
18 }
19
20 void f_next_token() {
21     token = mathescanner(NULL, text);
22 }
23
24 void f_factor () {
25     if (token == ZAHL)
26         f_next_token();
27     else if (token == KLA_AUF) {
28         f_next_token();
```

```
29          f_expression();
30          if (token == KLA_ZU)
31              f_next_token();
32          else
33              f_error();
34      }
35      else f_error();
36  }
37
38  void f_term() {
39      f_factor();
40      while(token == MAL || token == DIV) {
41          f_next_token();
42          f_factor();
43      }
44  }
45
46  void f_expression() {
47      f_term();
48      while(token == PLUS || token == MINUS) {
49          f_next_token();
50          f_term();
51      }
52  }
53
54  void f_calculation() {
55      f_expression();
56      if (token == END)
57          f_next_token();
58      else
59          f_error();
60  }
61
62  int main() {
63      char input[255];
64      while(printf("Input:"), fgets(input,255,stdin)!= NULL) {
65          mathescanner(input, NULL);          // Scanner-Reset
66          token = mathescanner(NULL, text);
67          error = 0;
68          f_calculation();
69          if (!error)
70              printf("Syntax OK!\n");
71          else
72              printf("Syntax nicht OK!\a\n");
73      }
74      return 0;
75  }
```

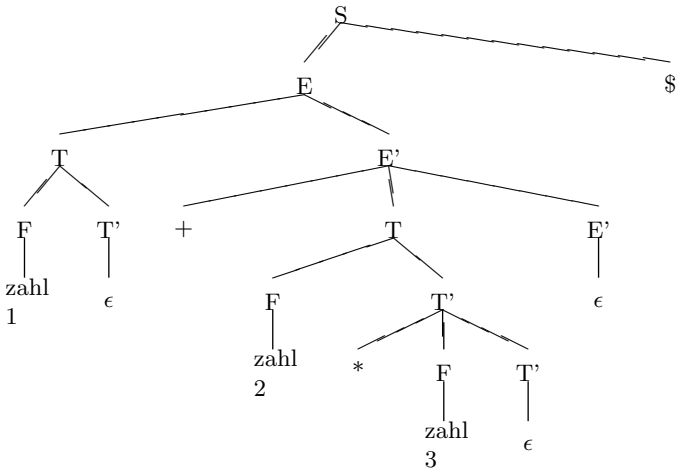
6.3.4 LL-Parsing mit Stackverwaltung

Gegeben ist linksrekursionsfreie Grammatik Γ_{16}.

Grammatik Γ_{16}: Linksrekurionsfreie Term-Grammatik in BNF

S → **E $**
E → **T E'**
E' → **+ T E'** | ϵ
T → **F T'**
T' → *** F T'** | ϵ
F → **(E)** | **zahl**

Abb. 6.3. Syntaxbaum bei linksrekursionsfreier Grammatik



Der Parser verhält sich nun folgendermaßen:

- Wenn auf TOS ein Terminal ist:
  - Wenn TOS = token dann lösche TOS und lies weiter,
  - andernfalls FEHLER!
- Andernfalls (TOS ist Non-Terminal)
  - Wenn in Parser-Tabelle(TOS,token) Regel eingetragen dann lösche TOS und lege Regel rückwärts auf Stack
  - andernfalls FEHLER!
- Wiederhole die ersten zwei Punkte bis Stack und Eingabe leer sind.

Algorithmus 6: LL-1-Algorithmus mit Stackverwaltung

Gegeben: PTABLE[TOS][token]	
Lege Ende-Token auf Stack	
Lege Startregel rückwärts auf Stack	
Lies token	
error = 0	
Eingabe nicht leer AND error = 0	
TOS ist Terminal	
Ja	Nein
TOS = token?	PTABLE[TOS][token] gesetzt?
Ja / Nein	Ja / Nein
lies token / error = 1	lösche TOS / error = 1
∅	Lege Regel PTABLE[TOS][token] rückwärts auf Stack / ∅

Voraussetzung für den Algorithmus ist eine linksrekursionsfreie Grammatik.

Anwendung von Algorithmus 4 zur Berechnung der FIRST-Mengen bzw. Algorithmus 5 zur Berechnung der FOLLOW-Mengen auf Grammatik Γ_{16} ergibt

$$\begin{aligned}
FIRST(E) &= \{(, id\} \\
FIRST(E') &= \{+, \epsilon\} \\
FIRST(T) &= \{(, id\} \\
FIRST(T') &= \{*, \epsilon\} \\
FIRST(F) &= \{(, id\} \\
FOLLOW(E) &= \{\$,)\} \\
FOLLOW(E') &= \{\$,)\} \\
FOLLOW(T) &= \{+, \$,)\} \\
FOLLOW(T') &= \{+, \$,)\} \\
FOLLOW(F) &= \{*, +, \$,)\}
\end{aligned}$$

Tabelle 6.2 zeigt, wie tabellengesteuert Code ausgegeben werden kann. Terminale Symbole werden beim Weiterlesen ausgegeben, zu nicht-terminalen Symbolen kann in der Grammatik ein Ausgabetext hinterlegt werden, welcher dann ausgegeben wird, wenn das Symbol auf dem Stack gelöscht wird (Algorithmus 6). Die Grammatik mit Ausgabe ist in Γ_{17} dargestellt.

Grammatik Γ_{17}: Linksrekurionsfreie Term-Grammatik in BNF mit Ausgabe

S → **E $**
E → **T E'**
E' → **+ T E'**$_{ADD}$ | ϵ
T → **F T'**
T' → *** F T'**$_{MULT}$ | ϵ
F → **(E)** | **zahl**

Tabelle 6.1. Parsing-Vorgang

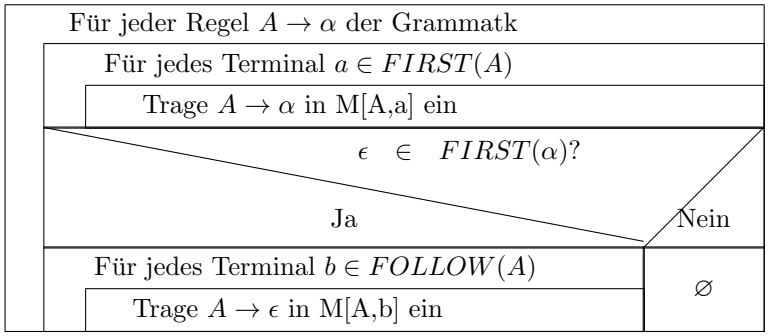
Stapel	Eingabe	Aktion
$\$E$	1 + 2 * 3 $	$E \to TE'$
$\$E'T$	1 + 2 * 3 $	$T \to FT'$
$\$E'T'F$	1 + 2 * 3 $	$F \to zahl$
$\$E'T'zahl$	1 + 2 * 3 $	
$\$E'T'$	+ 2 * 3 $	$T' \to \epsilon$
$\$E'$	+ 2 * 3 $	$E' \to +TE'$
$\$E'T+$	+ 2 * 3 $	
$\$E'T$	2 * 3 $	$T \to FT'$
$\$E'T'F$	2 * 3 $	$T \to zahl$
$\$E'T'zahl$	2 * 3 $	
$\$E'T'$	* 3 $	$T' \to *FT'$
$\$E'T'F*$	* 3 $	
$\$E'T'F$	3 $	$F \to zahl$
$\$E'T'zahl$	3 $	
$\$E'T'$	$	$T' \to \epsilon$
$\$E'$	$	$E' \to \epsilon$
$	$	

Tabelle 6.2. Parsing-Vorgang mit Code-Erzeugung

Stapel	Eingabe	Aktion	Ausgabe
$\$E$	1 + 2 * 3 $	$E \to TE'$	
$\$E'T$	1 + 2 * 3 $	$T \to FT'$	
$\$E'T'F$	1 + 2 * 3 $	$F \to zahl$	
$\$E'T'zahl$	1 + 2 * 3 $		1
$\$E'T'$	+ 2 * 3 $	$T' \to \epsilon$	
$\$E'$	+ 2 * 3 $	$E' \to +TE'$	
$\$E'_{ADD}T+$	+ 2 * 3 $		
$\$E'_{ADD}T$	2 * 3 $	$T \to FT'$	
$\$E'_{ADD}T'F$	2 * 3 $	$T \to zahl$	
$\$E'_{ADD}T'zahl$	2 * 3 $		2
$\$E'_{ADD}T'$	* 3 $	$T' \to *FT'$	
$\$E'_{ADD}T'_{MULT}F*$	* 3 $		
$\$E'_{ADD}T'_{MULT}F$	3 $	$F \to zahl$	
$\$E'_{ADD}T'_{MULT}zahl$	3 $		3
$\$E'_{ADD}T'_{MULT}$	$	$T' \to \epsilon$	MULT
$\$E'_{ADD}$	$	$E' \to \epsilon$	ADD
$	$		

Die Konstruktion der Parse-Tabelle nach [ASU88], Seite 231ff ist in Algorithmus 7 dargestellt.

Algorithmus 7: LL(1)-Tabelle erstellen



Beispiel Grammatik Γ_{18} in Tabelle 6.3

Grammatik Γ_{18}: Mathematischer Term linksrekursionsfrei

E → **T E'**
E' → **+ T E'**
E' → **- T E'**
E' → ϵ
T → **F T'**
T' → *** F T'**
T' → **/ F T'**
T' → ϵ
F → **(E)**
F → **id**

Tabelle 6.3. LL(1)-Parsetabelle

Top of Stack	Token								
	END	+	-	*	/	(	)	ZAHL	ERR
E						T E'		T E'	
E'	ϵ	+ t e'	- t e'				ϵ		
T						F T'		F T'	
T'	ϵ	ϵ	ϵ	* F T'	/ F T'		ϵ		
F						(E)		zahl	

Listing 6.3 implementiert den hier erläuterten Parser.

Listing 6.3. LL(1)-Parser mit Stackverwaltung (ll1-stack.c)

```
1  /*********************************************************
2  Tabellengesteuerter rekursionsfreier LL(1)-Parser
3  Ähnlich Grammatik 4.11 aus Drache 1, Seite 214:
4  Regel 0: S  ::= E $
5  Regel 1: E  ::= T E'
6  Regel 2: E' ::= + T E'
7  Regel 3: E' ::= - T E'
8  Regel 4: E' ::= eps
9  Regel 5: T  ::= F T'
10 Regel 6: T' ::= * F T'
11 Regel 7: T' ::= / F T'
12 Regel 8: T' ::= eps
13 Regel 9: F  ::= ( E )
14 Regel 10:F  ::= zahl
15 *********************************************************/
16 #include <stdio.h>
17 #include "term-scanner.h"
18 #include "term-scanner.c"
19 #include "genprog-stack.h"
20 #include "genprog-stack.c"
21
22 enum {nt_E = 256, nt_E2, nt_T, nt_T2, nt_F}; // Non-Terminals
23 int nt_offset = 256;
24
25 int parse_table[5][9] = {
26 /*          END  +   -   *   /   (   )  zahl  err */
```

```
27 /*E  */ {   -1, -1, -1, -1, -1,  1, -1,  1, -1},
28 /*E' */ {    4,  2,  3, -1, -1, -1,  4, -1, -1},
29 /*T  */ {   -1, -1, -1, -1, -1,  5, -1,  5, -1},
30 /*T' */ {    8,  8,  8,  6,  7, -1,  8, -1, -1},
31 /*F  */ {   -1, -1, -1, -1, -1,  9, -1, 10, -1},
32 };
33
34 int g[11][4] = { // Grammatik
35 /*0*/ {  nt_E, END,  -1    },
36 /*1*/ {  nt_T, nt_E2,  -1    },
37 /*2*/ {  PLUS, nt_T, nt_E2,  -1 },
38 /*3*/ {  MINUS,  nt_T, nt_E2,  -1 },
39 /*4*/ {-1            },
40 /*5*/ {nt_F,  nt_T2,  -1    },
41 /*6*/ {MAL, nt_F, nt_T2,  -1 },
42 /*7*/ {DIV, nt_F, nt_T2,  -1 },
43 /*8*/ {-1            },
44 /*9*/ {KLA_AUF,  nt_E, KLA_ZU,  -1 },
45 /*10*/{ZAHL,  -1       },
46 };
47
48 stack s;
49
50 void regel_auf_stack(int r) {
51    int i;
52    i = -1;
53    printf("regel %d\n", r);
54    while (i++, g[r][i] >=0);
55    for (i--; i >= 0; i--)
56       stack_push(&s, &(g[r][i]));
57 }
58
59 void int_print(int *i) { // für stack_print-Funktion
60    printf("%4d", *i);
61 }
62
63 int main() {
64     char input[] = " (1 * 2) - 2 + 3";
65     char text[255];
66     int token, error = 0, r, tos;
67
68     stack_init(&s, sizeof(int), 100, int_print);
69     term_scanner(input, NULL); // Scanner-Init
70     token = term_scanner(NULL, text);
71
72     regel_auf_stack(0); // Startregel auf Stack
73     while (!error && stack_height(&s)) {
74         if (stack_pop(&s, &tos), tos < nt_offset) // Terminal!
75             if (tos == token)
76                 token = term_scanner(NULL, text);
77             else
78                 error = 1;
79         else // Non-Terminal
80             if ((r = parse_table[tos - nt_offset ][token]) >= 0)
```

```
81              regel_auf_stack(r);
82                  else
83                      error = 1;
84          }
85          printf("Parse-ergebnis: %d\n", error);
86          stack_del(&s);
87          return 0;
88      }
```

6.3.5 Eliminierung der Links-Rekursion

Algorithmus 8: Eliminierung der Links-Rekursion

Gegeben: $A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$
(Kein β_i beginnt mit A)

Verfahren : Ersetze
$A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$
durch
$A \to \beta_1 A'|\beta_2 A'|\ldots|\beta_n A'$
und füge hinzu
$A' \to \alpha_1 A'|\alpha_2 A'|\ldots|\alpha_m A'|\epsilon$

Siehe [ASU88], Seite 214.

6.3.6 Tabellen-gesteuertes LL-Parsing

Siehe auch [Wir96], Seite 22ff.

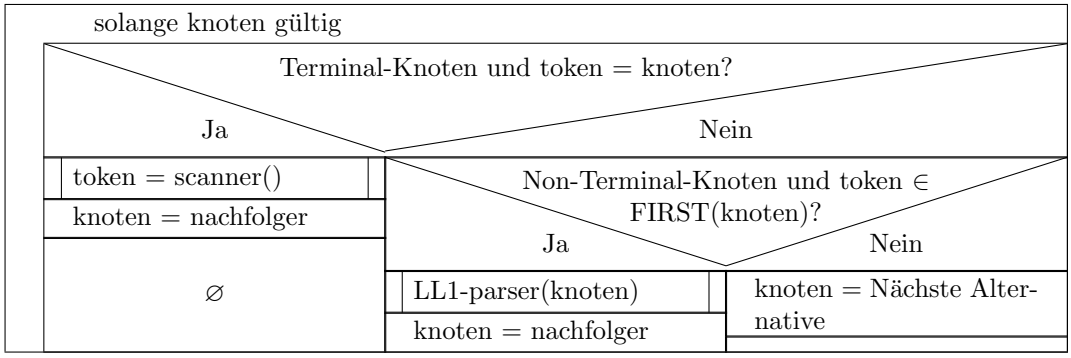
Am einfachsten basierend auf Syntaxgraphen der Regeln. Für jede Pfeilspitze wird ein Viertupel angelegt:

- Art des Knotens (Terminal oder Non-Terminal),
- Nr des entsprechenden Eintrags,
- Alternative, falls kein Match,
- Fortsetzung, falls Match.

Damit kann der Recursive-Descent-Parser unabhängig von der konkreten Grammatik implementiert werden:

Algorithmus 9: Tabellen-gesteuertes LL-Parsing

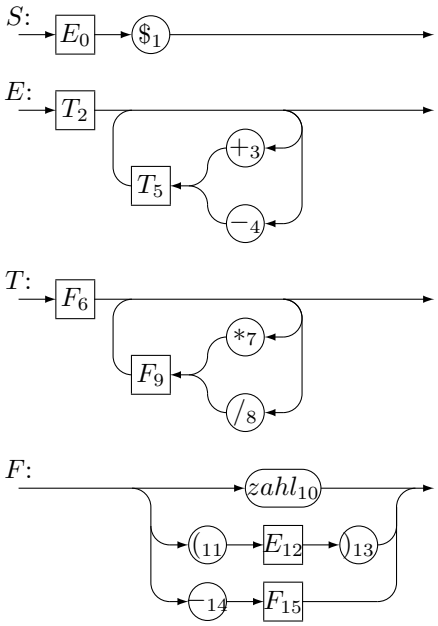
LL1-parser(knoten)



Als Nebeneffekt der beschriebenen Datenstruktur für die Syntaxdiagramme erscheint die Tatsache, dass die FIRST-Mengen einfach berechnet werden können. Listing 6.4 implementiert einen tabellengesteuerten LL(1)-Parser, der komplett unabhängig von der Grammatik codiert ist.

Als Beispiel soll für die Term-Grammatik Γ_{15} (Seite 68) ein Parser entwickelt werden. Dazu wird Syntax-Diagramm 7 entwickelt.

Syntaxdiagramm 7: Term-Grammatik



Listing 6.4. Tabellengesteuerter LL(1)-Parser (ll1-tabellarisch.c)

```
1  /*********************************************************
2   Tabellarisch gesteuerter LL(1)-Parser
3   ToDo: First-Mengen aus Tabelle konstruieren!
4   *********************************************************/
5  #include "term-scanner.c"
6  #include <stdio.h>
7
```

Tabelle 6.4. Parstabelle zu tabellengesteuertem LL(1)-Parsen mit Rekursion

Regel	Knoten	Typ	Wert	Alt.	Nachf.	FIRST
S	0	NT	2	e	1	{zahl,(,+,-}
	1	T	$	e	x	
E	2	NT	6	e	3	{zahl,(,+,-}
	3	T	+	4	5	
	4	T	-	x	5	
	5	NT	6	e	3	
T	6	NT	10	e	7	{zahl,(,+,-}
	7	T	*	8	9	
	8	T	/	x	9	
	9	NT	10	e	7	
F	10	T	zahl	11	x	{zahl,(,+,-}
	11	T	(	14	12	
	12	NT	2	e	13	
	13	T	)	e	x	
	14	T	-	e	15	
	15	NT	10	e	x	

```
 8  int LL1_first(node * table, int act_node, int token) {
 9      int first_set[256] = {0}, nt[256] = {0}, i, ok = 0;
10      do {
11          nt[act_node] = 1;
12          while (act_node >= 0) {
13              if (table[act_node].type == terminal)
14                  first_set[table[act_node].nr] = 1;
15              else if (table[act_node].type == nonterminal)
16                  if (nt[table[act_node].nr] == 0)
17                      nt[table[act_node].nr] = 2; // Noch abarbeiten!
18              act_node = table[act_node].alt;
19          }
20          act_node = -1; // Jetz noch unbearbeitet NT-Regeln suchen
21          while (++act_node < 256 && nt[act_node] != 2);
22      } while (act_node < 256);
23      if (token >= 0) {
24          for (i = 0; i < 256; i++)
25              if (first_set[i] && i == token)
26                  ok = 1;
27      }
28      else
29          for (i = 0; i < 256; i++)
30              if (first_set[i])
31                  printf(" %d", i);
32      return ok;
33  }
34
35  int LL1_parser(node * table, int act_node, int (*scanner)()) {
36      int error = 0;
37      static int token;
38      static char text[100];
39      if (table == NULL) {
40          token = scanner(text);
41          return 0;
42      }
```

```
43      while (act_node >= 0 && !error) {
44          //printf("act_node = %d, token = %d\n", act_node, token);
45          if (table[act_node].type == terminal
46              && table[act_node].nr == token) {
47              act_node = table[act_node].next;
48              token = scanner(text);
49          }
50          else if (table[act_node].type == nonterminal
51                  && (1|| LL1_first(table, act_node, token))) {
52              error = LL1_parser(table, table[act_node].nr, scanner);
53              act_node = table[act_node].next;
54          }
55          else {
56              act_node = table[act_node].alt;
57          }
58      }
59      return error || (act_node == -1);
60  }
```

6.4 LR-Parsing

Im Gegensatz zum Top-Down-Parsing wird beim Bottom-Up-Parsing versucht, ausgehend von den Blättern eines Syntaxbaums durch Rückwärts-Anwendung der Produktionsregeln der Grammatik (durch Reduzieren), einen Eingabetext auf das Startsymbol zu reduzieren.

Handles sind dabei rechte Seiten einer Regel. Ähnlich wie beim tabellengesteuerten LL(1)-Parsing müssen die Produktionsregeln daher einfach aufgebaut sein.

Betrachten wir Grammatik Γ_{19} (Beispiel aus [ASU88] Seite 267:
Grammatik Γ_{19}: Einfache Term-Grammatik in BNF
1) E → E '+' T
2) E → T
3) T → T * F
4) T → F
5) F → '(' E ')'
6) F → zahl

Ein sog. LR-Parser besteht nun aus einem Stack und einer Eingabe. Je nach internem Zustand des Automaten wird entweder das nächste Zeichen der Eingabe auf den Stack geschoben oder der Stack modifiziert. Im letzteren Fall wir - wenn ein handle am oberen Ende des Stacks gefunden wird, dieses zur linken Seite der Regel reduziert. Daraus resultieren verschiedene Probleme:

- Der Stack ist kein Stack im eigentlichen Sinn, da nicht nur das oberste Element gelesen wird, sondern auch Elemente unterhalb des obersten Elements.
- Prinzipiell kann der Parse-Vorgang mehrdeutig sein, da evtl mehrere verschiedene Handle gefunden werden. auch ist manchmal nicht eindeutig, ob reduziert werden muss oder oder geschoben. Diese Probleme werden Reduziere/Reduziere-Konflikte und Schiebe/Reduziere-Konflikte genannt.

Ein LR-Kellerautomat kennt vier Aktionen:

Reduziere das Handle am Stackende anhand einer Regel

Schiebe das nächste Eingabeelement auf den Stack

Akzeptiere die Eingabe

Fehler in der Eingabe

Zuerst soll der Syntaxbaum für den Ausdruck $1 + 2 * 3$ erstellt werden. In den folgenden Grafiken sind die eingekreisten Symbole “auf dem Stack“, das unterstrichene Symbol das nächste der Eingabe, also das Look-ahead-token.

<u>1</u> + 2 * 3

Da der Stack leer ist kann nur geschoben werden. Die 1 kommt auf den Stack, in der Eingabe steht nun das + an:

(1) <u>+</u> 2 * 3

Es findet sich kein Handle, das mi “zahl +“ begint, daher kann nicht sinnvoll geschoben werden. Allerdings findet sich ein Handle “zahl“, welches nun zu “F“ reduziert werden kann:

(F)
|
1 <u>+</u> 2 * 3

Auch “F +“ ist wird kein sinnvolles handle, aber “F“ kann zu “T“ reduziert werden:

(T)
|
F
|
1 <u>+</u> 2 * 3

Auch “T +“ ist wird kein sinnvolles handle, aber “T“ kann zu “E“ reduziert werden:

(E)
|
T
|
F
|
1 <u>+</u> 2 * 3

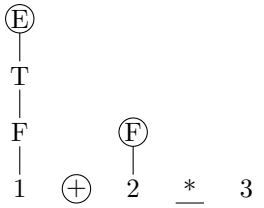
Nu kann “E“ nicht weiter reduziert werden, aber mit “E +“ beginnt ein Handle, daher wird geschoben:

(E)
|
T
|
F
|
1 ⊕ <u>2</u> * 3

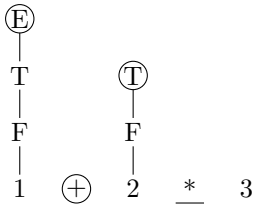
Es findet sich kein Handle, welches reduziert werden kann, daher kann nur geschoben werden:

(E)
|
T
|
F
|
1 ⊕ (2) <u>*</u> 3

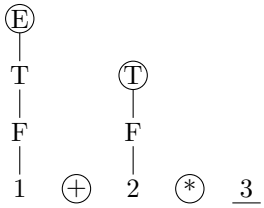
Als einzige sinnvolle Aktion findet sich nun die Reduktion von “zahl“ zu “F“:



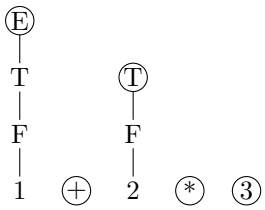
Nun kann lediglich “F“ zu “T“ reduziert werden:



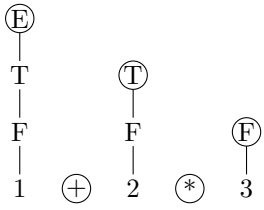
Jetzt entsteht ein Schiebe/Reduziere-Konflikt. Man könnte “E + T“ reduzieren oder aber schieben, da mit “T *“ auch ein Handle beginnt. Ohne zu begründen - wir schieben!



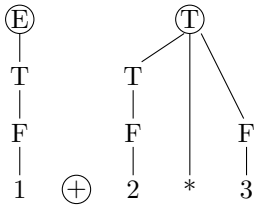
Nun kann wieder nur geschoben werden, die Eingabe ist leer:



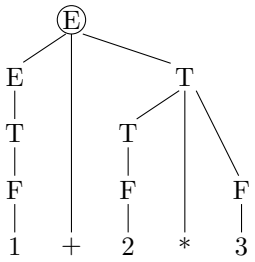
Wir reduzieren “zahl“ zu “F“:



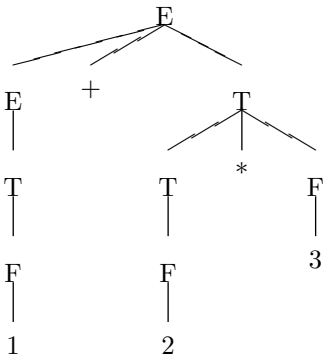
Nun kann “T * F“ zu “T“ reduziert werden:



Als letzter Schritt wird “E + T“ zu “E“ reduziert:



Dies sieht - als Baum mit korrekt dargestellten Höhen der Knoten - folgendermaßen aus:



Während des Parsens ist es wieder nicht notwendig, den gesamten Baum im Hauptspeicher zu halten. Stattdessen kann mit einem einfachen Stack gearbeitet werden. Der Parse-Vorgang der in den vorhergehenden Grafiken dargestellt wurde kann also kompakt als Tabelle 6.5 dargestellt werden.

Tabelle 6.5. LR-Parsevorgang 1+2*3

Stack	Input	Aktion
	1 + 2 * 3 $	s
1	+ 2 * 3 $	r6
F	+ 2 * 3 $	r4
T	+ 2 * 3 $	r2
E	+ 2 * 3 $	s
E +	2 * 3 $	s
E + 2	* 3 $	r6
E + F	* 3 $	r4
E + T	* 3 $	s (!!!)
E + T *	3 $	s
E + T * 3	$	r6
E + T * F	$	r3
E + T	$	r1
E	$	a

Interessant ist der mit !!! markierte Zustand. Man hätte hier zwar das Handle T über Regel 2 zu E reduzieren können, da aber ein *-Zeichen als Eingabe anliegt und das Handle aus Regel 5 mit “T*“ beginnt macht es sinn, auf dieses Handle zu setzen.

Die Entscheidung, welche Operation durchgeführt wird, hängt also einerseits von mehreren Elementen am Ende des Stacks und andererseits von der Eingabe ab.

Da nun die Suche nach dem optimalen Handle zeitaufwendig ist, werden Parse-Tabellen entwickelt, die lediglich anhand des letzten Stack-Elements operieren können. Das Erstellen dieser Tabellen, die für typische Programmiersprachen wie Pascal oder C mehrere hundert

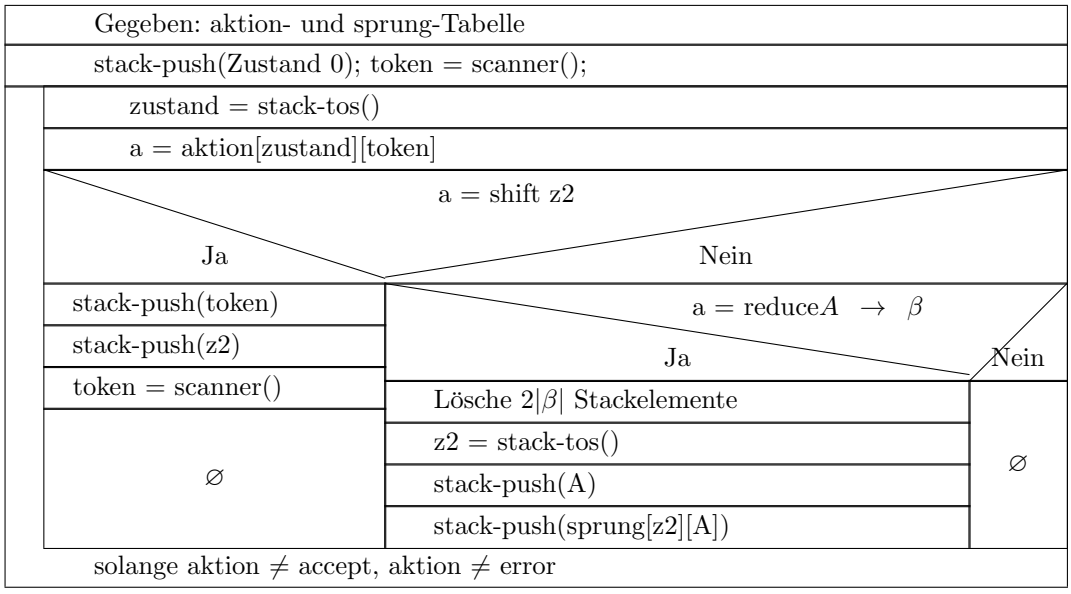
Zeilen enthalten, ist in der Praxis zu aufwändig. Daher werden diese Tabellen von Compilergeneratoren erstellt. Für ein Verständnis der Compilergeneratoren ist aber ein prinzipielles Verständnis dieses Verfahren notwendig.

Tabelle 6.6. LR-Tabelle für Ausdrücke

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

Algorithmus 10: LR-Algorithmus

LR-Algorithmus



Wendet man nun Algorithmus 10 auf Tabelle 6.4 und die Eingabe $1+2*3$ an so ergibt sich der in Tabelle 6.7 dargestellte Parse-Vorgang:

Listing 6.5. LR-Parser (keller-lr-parser.c)

```
1 /*******************************************************
2 LR-Parser
3 Zu Drache S 267
4 *******************************************************/
5 #include "term-scanner.c"
6 #include "genprog-stack.c"
7
8 char aktion[12][9][4] = {
```

Tabelle 6.7. LR-Parse-Vorgang 1+2*3

Stack	Input	Aktion
0	`1 + 2 * 3 $`	s5
0 z5	`+ 2 * 3 $`	r6
0 F3	`+ 2 * 3 $`	r4
0 T2	`+ 2 * 3 $`	r2
0 E1	`+ 2 * 3 $`	s6
0 E1 +6	`2 * 3 $`	s5
0 E1 +6 z5	`* 3 $`	r6
0 E1 +6 F3	`* 3 $`	r4
0 E1 +6 T9	`* 3 $`	s7
0 E1 +6 T9 *7	`3 $`	s5
0 E1 +6 T9 *7 z5	`$`	r6
0 E1 +6 T9 *7 F10	`$`	r3
0 E1 +6 T9	`$`	r1
0 E1	`$`	a

```
9           /* $    +    -    *    /    (    )   zahl  err
10  /* 0*/ {"er","er","er","er","er","s4","er","s5","er"},
11  /* 1*/ {"ac","s6","s6","er","er","er","er","er","er"},
12  /* 2*/ {"r2","r2","r2","s7","s7","er","r2","er","er"},
13  /* 3*/ {"r4","r4","r4","r4","r4","er","r4","er","er"},
14  /* 4*/ {"er","er","er","er","er","s4","er","s5","er"},
15  /* 5*/ {"r6","r6","r6","r6","r6","er","r6","er","er"},
16  /* 6*/ {"er","er","er","er","er","s4","er","s5","er"},
17  /* 7*/ {"er","er","er","er","er","s4","er","s5","er"},
18  /* 8*/ {"er","s6","s6","er","er","er","s11","er","er"},
19  /* 9*/ {"r1","r1","r1","s7","s7","er","r1","er","er"},
20  /*10*/ {"r3","r3","r3","r3","r3","er","r3","er","er"},
21  /*11*/ {"r5","r5","r5","r5","r5","er","r5","er","er"}};
22
23  int sprung[12][3] = {   /* 0*/ { 1, 2, 3},
24            /* 1*/ {-1,-1,-1},
25            /* 2*/ {-1,-1,-1},
26            /* 3*/ {-1,-1,-1},
27            /* 4*/ { 8, 2, 3},
28            /* 5*/ {-1,-1,-1},
29            /* 6*/ {-1, 9, 3},
30            /* 7*/ {-1,-1,10},
31            /* 8*/ {-1,-1,-1},
32            /* 9*/ {-1,-1,-1},
33            /*10*/ {-1,-1,-1},
34            /*11*/ {-1,-1,-1}};
35
36  enum {nt_E, nt_T, nt_F};
37  struct {int nt; int l;} g[7] = {{-1, -1}, // nicht verwendet
38            /*R1*/   {nt_E, 3},
39            /*R2*/   {nt_E, 1},
40            /*R3*/   {nt_T, 3},
41            /*R4*/   {nt_T, 1},
42            /*R5*/   {nt_F, 3},
43            /*R6*/   {nt_F, 1}};
44
45  void int_print(void *p) { // Für Stack-Programme
```

```
46      printf("%d", *(int *) p);
47  }
48
49  int main() {
50      int token, i, r, tos, a, dummy, z2;
51      char to_scan[] = "1 * 2 + 3", text[10];
52      stack s;
53
54      stack_init(&s, sizeof(int), 100, int_print);
55      dummy = 0, stack_push(&s, &dummy);
56
57      term_scanner(to_scan, NULL);
58      token = term_scanner(NULL, text);
59
60      do {
61          stack_tos(&s, &tos);
62          printf("tos = %2d  token = %d  aktion = %s\n",
63              tos, token, aktion[tos][token]);
64          a = aktion[tos][token][0];
65          r = atoi(aktion[tos][token]+1);
66          if (a == 's') { // Schiebe
67              stack_push(&s, &token);
68              stack_push(&s, &r);
69              token = term_scanner(NULL, text);
70          }
71          else if (a == 'r') { // Reduziere
72              for (i = 0; i < 2 * g[r].l; i++)
73                  stack_pop(&s, &dummy);
74              stack_tos(&s, &z2);
75              stack_push(&s, &(g[r].nt));
76              stack_push(&s, &(sprung[z2][g[r].nt]));
77          }
78
79      } while (a != 'e' && a != 'a');
80      printf("Ergebnis: %c\n", a);
81      return 0;
82  }
```

Erzeugung von LR-Parser-Tabellen

LR-Parser-Tabellen sind - zumindest für die klassischen Programmiersprachen - zu aufwändig für eine händische Erstellung. Für eine detaillierte Lektüre empfiehlt sich [ASU88], Seiten 269ff.

In der Praxis werden diese Tabellen mittels eines Parser-Generators automatisch erstellt. Der sicherlich bekannteste Parser-Generator ist YACC bzw. sein GNU-Derivat Bison (siehe auch 12.4 ab Seite 131. Der PL/0-Parser des Modellcompilers besteht aus einer Tabelle mit 99 Zuständen!

Als ein Beispiel für die Generierung von Tabellen betrachten wir Listing 6.6, einer Implementierung unserer Term-Grammatik in Yacc.

Listing 6.6. Term-Grammatik in Yacc (lr-term.y)

```
1  %token zahl
```

```
2  %%
3  E: E '+' T {printf("+\n");}
4      | T
5      ;
6
7  T: T '*' F
8      | F
9      ;
10
11 F: '(' E ')'
12     | zahl
13     ;
14 %%
15 int yylex() {
16     int c = getchar();
17     return (c != EOF) ? c : 0;
18 }
```

Man man Yacc so aufrufen, dass Zusatzinformationen wie Grammatik-Konflikte aber auch der produzierte Parser in einer gesonderten Datei beschrieben werden. Für Listing 6.6 ist diese Information in Listing 6.7 beschrieben:

Listing 6.7. Term-Grammatik in Yacc - Zusatzinformationen (lr-term.output)

```
1  Grammatik
2
3      0 $accept: E $end
4
5      1 E: E '+' T
6      2  | T
7
8      3 T: T '*' F
9      4  | F
10
11     5 F: '(' E ')'
12     6  | zahl
13
14
15 Terminale und die Regeln, in denen sie verwendet werden
16
17 $end (0) 0
18 '(' (40) 5
19 ')' (41) 5
20 '*' (42) 3
21 '+' (43) 1
22 error (256)
23 zahl (258) 6
24
25
26 Nicht-Terminal und die Regeln, in denen sie verwendet werden
27
28 $accept (8)
29     auf der linken Seite: 0
30 E (9)
31     auf der linken Seite: 1 2, auf der rechten Seite: 0 1 5
```

```
T (10)
    auf der linken Seite: 3 4, auf der rechten Seite: 1 2 3
F (11)
    auf der linken Seite: 5 6, auf der rechten Seite: 3 4


Zustand 0

    0 $accept: . E $end

    zahl  schiebe und gehe zu Zustand 1 Ã¼ber
    '('   schiebe und gehe zu Zustand 2 Ã¼ber

    E  gehe zu Zustand 3 Ã¼ber
    T  gehe zu Zustand 4 Ã¼ber
    F  gehe zu Zustand 5 Ã¼ber


Zustand 1

    6 F: zahl .

    $default  reduziere mit Regel 6 (F)


Zustand 2

    5 F: '(' . E ')'

    zahl  schiebe und gehe zu Zustand 1 Ã¼ber
    '('   schiebe und gehe zu Zustand 2 Ã¼ber

    E  gehe zu Zustand 6 Ã¼ber
    T  gehe zu Zustand 4 Ã¼ber
    F  gehe zu Zustand 5 Ã¼ber


Zustand 3

    0 $accept: E . $end
    1 E: E . '+' T

    $end  schiebe und gehe zu Zustand 7 Ã¼ber
    '+'   schiebe und gehe zu Zustand 8 Ã¼ber


Zustand 4

    2 E: T .
    3 T: T . '*' F

    '*'  schiebe und gehe zu Zustand 9 Ã¼ber

```

```
85      $default  reduziere mit Regel 2 (E)
86
87
88  Zustand 5
89
90      4 T: F .
91
92      $default  reduziere mit Regel 4 (T)
93
94
95  Zustand 6
96
97      1 E: E . '+' T
98      5 F: '(' E . ')'
99
100     '+'  schiebe und gehe zu Zustand 8 Ã¼ber
101     ')'  schiebe und gehe zu Zustand 10 Ã¼ber
102
103
104 Zustand 7
105
106     0 $accept: E $end .
107
108     $default  annehmen
109
110
111 Zustand 8
112
113     1 E: E '+' . T
114
115     zahl  schiebe und gehe zu Zustand 1 Ã¼ber
116     '('   schiebe und gehe zu Zustand 2 Ã¼ber
117
118     T  gehe zu Zustand 11 Ã¼ber
119     F  gehe zu Zustand 5 Ã¼ber
120
121
122 Zustand 9
123
124     3 T: T '*' . F
125
126     zahl  schiebe und gehe zu Zustand 1 Ã¼ber
127     '('   schiebe und gehe zu Zustand 2 Ã¼ber
128
129     F  gehe zu Zustand 12 Ã¼ber
130
131
132 Zustand 10
133
134     5 F: '(' E ')' .
135
136     $default  reduziere mit Regel 5 (F)
137
138
```

```
139 Zustand 11
140
141     1 E: E '+' T .
142     3 T: T . '*' F
143
144     '*'  schiebe und gehe zu Zustand 9 Ã¼ber
145
146     $default  reduziere mit Regel 1 (E)
147
148
149 Zustand 12
150
151     3 T: T '*' F .
152
153     $default  reduziere mit Regel 3 (T)
```

Analysiert man die Informationen in der Datei, so ergibt sich die folgende Parser-Tabelle:

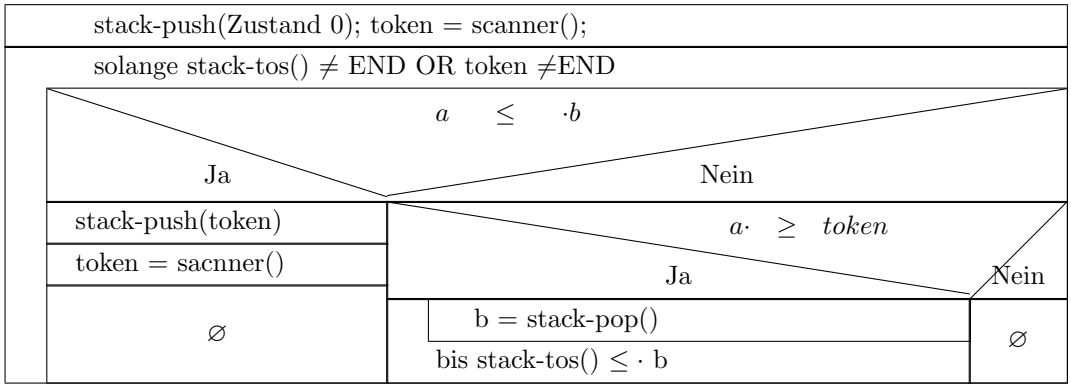
Zustand	Aktion						Sprung		
	z	+	*	(	)	$	E	T	F
0	s1			s2			3	4	5
1	r6	r6	r6	r6	r6	r6			
2	s1			s2			6	4	5
3		s8				s7			
4	r2	r2	s9	r2	r2	r2	2		
5	r4	r4	r4	r4	r4	r4			
6		s8			s10				
7	a	a	a	a	a	a			
8	s1			s2				11	5
9	s1			s2		1			12
10	r5	r5	r5	r5	r5	r5			
11	r1	r1	s9	r1	r1	r1			
12	r3	r3	r3	r3	r3	r3			

6.5 Operator-Prioritäts-Analyse

$a \leq \cdot b$ “Priorität von a ist kleiner als Priorität von b4
$a \doteq b$ “Priorität von a ist gleich Priorität von b4
$a \cdot \geq b$ “Priorität von a ist kleiner als Priorität von b4

Algorithmus 11: Operator-Präzendenz-Algorithmus

Operator-Präzendenz-Algorithmus



Listing 6.8. Operator-Präzedenz-Analyse (operator-precedence.c)

```
1  /****************************************
2  Operator-Precedence
3  ****************************************/
4  #include "term-scanner.h"
5  #include "genprog-stack.h"
6  #include <stdio.h>
7  int priority [9][9] = {
8
9  //END, PLUS, MINUS, MAL, DIV, KLA_AUF, KLA_ZU, ZAHL, FEHLER};
10
11 /* END     */    {-2, -1, -1, -1, -1, -1, -2, -1, -2},
12 /* PLUS    */    {+1, +1, +1, -1, -1, -1, +1, -1, -2},
13 /* MINUS*/   {+1, +1, +1, -1, -1, -1, +1, -1, -2},
14 /* MAL     */    {+1, +1, +1, +1, +1, -1, +1, -1, -2},
15 /* DIV     */    {+1, +1, +1, +1, +1, -1, +1, -1, -2},
16 /* KLA_AUF */   {-2, -1, -1, -1, -1, -1,  0, -1, -2},
17 /* KLA_ZU */    {+1, +1, +1, +1, +1, -2, +1, -2, -2},
18 /* ZAHL */   {+1, +1, +1, +1, +1, -2, +1, -2, -2},
19 /* FEHLER */    {-2, -2, -2, -2, -2, -2, -2, -2, -2}  };
20
21 typedef struct {
22     int token;
23     char text[20];
24 } stack_elem;
25
```

```
26 void token_print(stack_elem *p) {
27    printf("%d (\"%s\")", p->token, p->text);
28 }
29
30
31 int main() {
32    char input[] = "(1 + 2) * 3 ", text[100];
33    stack s;
34    stack_elem next, tos, elem;
35    int error;
36
37    stack_init(&s, sizeof(stack_elem), 15, token_print);
38
39    term_scanner(input, NULL);
40    next.token = term_scanner(NULL, next.text);
41
42    error = 0;
43    tos.token = 0, tos.text[0] = '\0', stack_push(&s, &tos);
44 // stack_print(&s);
45    while (!error && ((stack_tos(&s, &tos), tos.token != END)) || next.token != END ) {
46       printf("TOS: %d (\"%s\")", tos.token, tos.text);
47       printf(" input: %d (\"%s\")\n", next.token, next.text);
48       if (priority[tos.token][next.token] == -1 || priority[tos.token][next.token] ==
49          stack_push(&s, &next);
50          next.token = term_scanner(NULL, next.text);
51          printf("\tpush\n");
52 //       stack_print(&s);
53       }
54       else if (priority[tos.token][next.token] == +1) {
55          do {
56             stack_pop(&s, &elem);
57             printf("\tpop %d (\"%s\")\n", elem.token, elem.text);
58          } while (stack_tos(&s, &tos), priority[tos.token][elem.token] != -1);
59
60       }
61       else
62          error = 1;
63       fflush(stdout);
64
65    }
66
67    return 0;
68 }
```

7

Symboltabellen

- Problem: Verwendung von Bezeichnern nicht über Grammatik kontrollierbar.
- Beispiel: ! a.
- Grammatik, die Bezeichnernamen beinhaltet, wäre zu komplex
- Abhilfe: Symboltabelle
  - Ersetzt alle Bezeichner durch int-Werte
  - Wegen Rekursion zur Compilezeit aber noch keine Errechnung eines HS-PLatzes möglich
  - Für jeden Bezeichner ist ein Tripel zu verwalten:
    - Level-Delta im statischen Code
    - Offset, laufende Nr im jeweiligen Level
    - Typ des Bezeichners

Ersetzen in Programm 9.1 (Seite 106) alle Bezeichner durch Typ/Level/Offset

7.1 Einführung

Wichtig bei der Diskussion der Symboltabelle sind Begriffe wie

- Namensbereiche von Bezeichnern (CT)
- Lokale und globale Bezeichner
- Lebenszeit eines Bezeichners (RT)

7.2 Methoden der Symboltabelle

Eine Symboltabelle hat prinzipiell die folgenden vier Methoden:

- Einfügen von neuen Bezeichnern: insert
- Suchen von Bezeichnern: lookup
- Betreten eines neuen Namensbereichs: level-up
- Verlassen eines Namensbereichs: level-down

7.3 Aufbau eines Symboltabelle

Vorstellung: Die Symboltabelle ist eine zweidimensionale Tabelle, die in der einen Dimension die Symbole und in der anderen Dimension die Namensbereiche verwaltet.

Praktisch wird man eine Symboltabelle so nicht codieren, da für alle Namensbereiche eine feste feste maximale Zahl von Bezeichnern.

Die Tabelle beinhaltet Name und Typ eines Bezeichners. Die wichtigen Werte Level und Offset ergeben sich aus der Position eines Bezeichners in der Symboltabelle.

Abb. 7.1. Aufbau einer Symboltabelle

4				
3				
2				
1				
0				
Level	*** Ebene 0**	**Ebene 1**	**Ebene 2**	**Ebene 3**

In Zeile 2 von Listing 9.1 werden die Variablen a und d lokal im Hauptprogramm deklariert, in Zeile 4 kommt die Prozedur f dazu. Die Symboltabelle hat nun folgendes Aussehen:

4				
3				
2	f proc			
1	d var			
0	a var			
Level	*** Ebene 0**	**Ebene 1**	**Ebene 2**	**Ebene 3**

In Zeile 4 wird ein neuer Namensbereich betreten, in Zeile 5 die Variablen b und d deklariert und in Zeile 7 die Prozedur g lokal zu f deklariert:

4				
3				
2	f proc	g proc		
2	d var	d var		
0	a var	b var		
Level	**Ebene 0**	*** Ebene 1**	**Ebene 2**	**Ebene 3**

In Zeile 7 wird dann ein neuer Namensraum betreten und lokal die Variablen c und d deklariert:

4				
3				
2	f proc	g proc		
1	d var	d var	d var	
0	a var	b var	c var	
Level	**Ebene 0**	**Ebene 1**	*** Ebene 2**	**Ebene 3**

Bei den Variablenzugriffen in den Zeilen 10 bis 14 ergibt sich nun für a das Bitupel (Level 2, Offset 0), für c das Bitupel (Level 0, Offset 0) und für d das Bitupel (Level 0, Offset 1).

Wird bei der Compilierung Zeile 17 erreicht, so siht die Symboltabelle wieder aus wie nach Erreichen des Namensraums von Prozedur f. Für a ergibt sich jetzt das Bitupel (Level 1, Offset 0), für b das Bitupel (Level 0 Offset 0),für d ergibt sich (Level 0, Offset 1) und für g (was ja eine Prozedur ist!) (Level 0, Offset 2).

7.4 Fehlermöglichkeiten

Insert: • Kein Speicherplatz mehr vorhanden
- Bezeichnernamen im aktuellen Level bereits eingetragen

Lookup: • Bezeichner nicht vorhanden
- Bezeichner falscher Typ

Level-Up: • Kein Speicherplatz mehr

Level-Down: • Bereits in Level 0

7.5 Namens-Suche

Bei der Symboltabelle ist die schnelle Suche nach Strings nötig. Bei kleinen Programmen, die übersetzt werden, ist ein einfacher String-Vergleich problemlos. Kritisch wird es aber, wenn etwa mit einem C-Compiler ein ganzes Betriebssystem mit mehreren zehntausend Zeilen Quellcode compiliert werden muss. In einem solchen Fall muss der Text-Suche in der Symboltabelle besondere Bedeutung zugemessen werden.

Möglichkeiten zur Beschleunigung wären

- Hash-Verfahren
- Sortierte Index-Felder für binäre Suche
- Dynamische Generierung endlicher Automaten zur Indexierung

7.6 Eine Symboltabelle auf Basis der C++-STL-Map

Die Symboltabelle wird aus einem Array aus Maps (siehe C++-STL) aufgebaut. Das Array bildet die verschiedenen Level der Symboltabelle ab. Die einzelnen Maps - die aus Key-Value-Paaren aufgebaut sind - haben als Zugriffs-Key den Namen des Symbols und als Value das Tupel {Eintragsart; Laufende Nr.}. Die jeweilige aktuelle Höhe der Spalten wird in einem Array height gespeichert. Über die Maps - die intern über B-Bäume realisiert sind - sind nun effiziente String-Suchen möglich.

Listing **??** zeigt die Klassendeklaration, Listing **??** den Code der Klasse.

8

Zwischencode

Betrachten wir wieder einmal Abbildung 1.1 auf Seite 13. Im Frontend des Compilers befinden sich Scanner und Parser, im Backend die (noch zu behandelnde) Codeerzeugung.

Bei kleinen Compilern wird die Code-Generierung gerne direkt in den Parser integriert, dies spart Arbeit und macht den gesamten Code des Compilers kompakt. Es gibt aber einige gute Gründe, warum der Parse-Vorgang (das Frontend) von der Code-Generierung (dem Backend) getrennt werden sollte:

- Soll nicht nur ein einzelner, isolierter Compiler erstellt werden, sondern eine ganze Compiler-Collection wie etwa die GCC, so müssten für n Quellsprachen und m Zielsysteme ($n*m$) verschiedene Compiler erstellt werden. Abbildung 8.1 zeigt diesen Sachverhalt.

  Wird statt der direkten Integration von Front- und Backend aber zwischen diesen Modulen eine sauber definierte und vor allem für alle Quellsprachen und Zielsysteme einheitliche Schnittstelle verwendet, so reduziert sich der Aufwand auf die Erstellung von n Frontends und m Backends. Dies wird in Abbildung 8.2 dargestellt.
- Änderungen in der Vorgehensweise des Compilers ziehen viele Vreänderungen nach sich. So ist es in der Regel nicht einfach möglich, einen Recursive-Descent-Parser durch einen Bottom-Up-Parser zu ersetzen. Dies resultiert aus der Tatsache, dass die Grammatiken meist für bestimmte Parse-Techniken angepasst werden müssen. So haben wir bereits verschiedene Grammtiken für arithmetische Terme kennengelernt, etwa Grammatik 12 (Seite 28) oder Grammatik 18 (Seite 73).

Aus diesen Gründen wird die Codeerzeugung vom Parser getrennt und ein sog. Zwischencode als Schnittstelle zwischen Front- und Backend eingesetzt.

8.1 Einführung

Leider existiert für den Zwischencode nicht eine so schöne Theorie wie die der Formalen Sprachen oder der Automatentheorie, diese Theorien hatten die lexikalische und die syntaktische Analyse einfach werden lassen. Stattdessen muss eine allgemeine Schnittstelle gefunden werden. Beim Betrachten von Abbildung 8.2 wird klar, dass diese Schnittstelle sowohl C- als auch Fortran-, Pascal- und weitere Programme abbilden können muss.

In der Praxis ist Zwischencode häufig

- ein Zielsystem-unabhängiger Pseudo-Assembler-Code
  - für Register-Stack-Maschinen
  - für Zwei-Adress-Maschinen

Abb. 8.1. Compiler-Collection ohne Schnittstelle

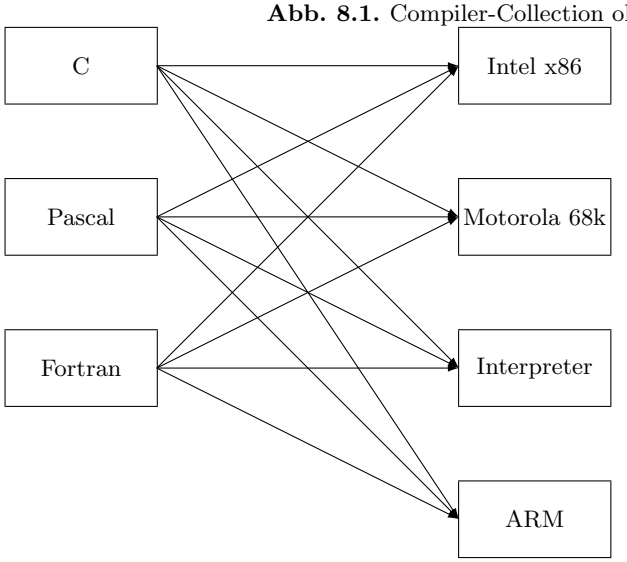
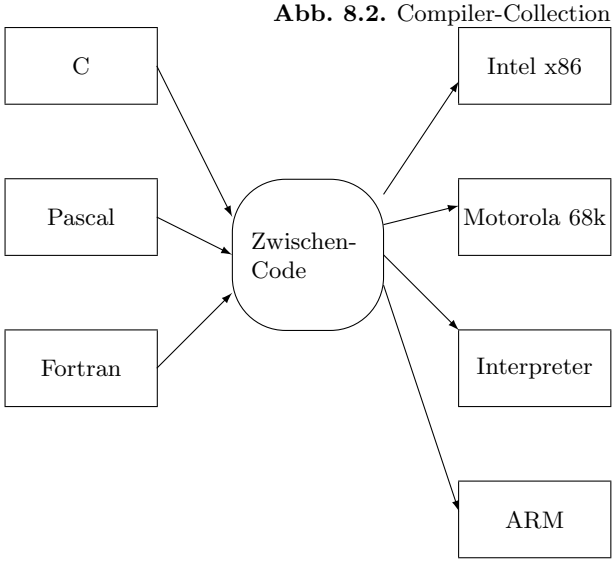


Abb. 8.2. Compiler-Collection mit Schnittstelle



  - für Drei-Adress-Maschinen
- ein Syntaxbaum

Syntax-Bäume sind mit Sicherheit die beste, aber auch die schwierigste Variante von Zwischencodes. Bei Syntax-Bäumen ist es sehr wichtig, dass sie die Sprache abbilden und nicht die Grammatik. Andernfalls wäre ein Austausch des Parsers wie oben diskutiert nicht möglich. Daraus resultiert die Namensgebung “Abstrakter Syntaxbaum“.

8.2 AST für Terme

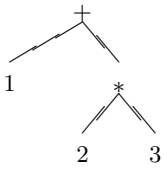
Gewählt wird nicht der Syntaxbaum der Grammatik, sondern der Rechenbaum (Operatorbaum).

Die Regeln zum Erstellen eines Syntaxbaums sind die folgenden:

- Es ist bzgl. Operatorpriorität und -assoziativität der schwächste Operator zu suchen und oben im Baum einzutragen. Für jeden Operanden wird ein AST nach unten[1] gezeichnet.
- Für jeden Operanden wird
  - dieser am Ende des Asts eingetragen, falls der Operand eine Zahl ist,
  - ein Teilbaum gezeichnet, falls der Operand ein Teilausdruck ist.

Als Beispiel soll der Syntaxbaum für den Ausdruck $1+2*3$ erstellt werden: Der schwächste Operator ist der +-Operator. Der linke Operand ist die Zahl 1, der rechte Operand der Teilausdruck $2*3$. Da dieser nur noch einen Operator enthält ist die Erstellung des entsprechenden Teilbaums trivial. Der gesamte Syntaxbaum ist in Abbildung 8.3 dargestellt.

Abb. 8.3. Syntaxbaum für Term $1+2*3$



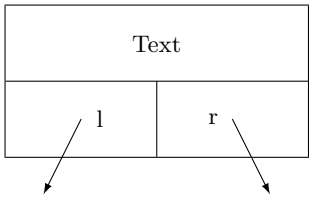
In diesen Syntaxbäumen sind keine Klammern mehr vorhanden. Diese werden über die Form des Syntaxbaums abgebildet.

Hat ein Compiler einen Syntaxbaum erstellt, so kann in einen zweiten Schritt dieser abgearbeitet werden.

- Durchläuft man den Baum im Inorder-Verfahren unter Berücksichtigung der verschiedenen Operatoren so kann das Ergebnis des Terms errechnet werden.
- Analog kann durch das Inorder-Verfahren auch Drei-Address-Code erzeugt werden.
- Durchläuft man den Baum im Postorder-Verfahren so kann die UPN-Syntax ausgegeben werden.

Da Operatoren zwei Operanden haben (mit der Ausnahme der unären Vorzeichen, die nur einen Operanden haben), bietet sich eine dynamische Datenstruktur für die Abbildung des Baums an.

Abb. 8.4. Datenstruktur für binären Operator-Baum




```
typedef struct s_node * ast;
typedef struct s_node ast_node;
struct s_node {
    char text[10];
    ast l;
    ast r;
};
```

[1] In der Informatik stehen Bäume immer auf dem Kopf!

Im Text wird entweder der Zahlenwert oder aber der Operator als ASCII-Text gespeichert, wobei für das unäre Minus der Text "CHS" ("Change-sign") eingesetzt wird. In den Blattknoten sind l- und r-Zeiger auf NULL gesetzt, beim unären Minus ist nur der l-Zeiger gesetzt.

Diese Datenstruktur ist als struct node in Listing 8.1 definiert. Der Code findet sich in Listing 8.2.

Listing 8.1. Definitionsdatei zum Operator-Baum (ast/term-ast.h)

```
1  /*******************************************************
2  Header-Datei zum Syntaxbaum
3  *******************************************************/
4  #ifndef __TERM_TREE_H__
5  #define __TERM_TREE_H__ 1
6  typedef struct s_node * ast;
7  typedef struct s_node ast_node;
8  struct s_node {
9      char text[10];
10     ast l;
11     ast r;
12 };
13 // Prototypen
14 void tree_output(ast , int);
15 void tree_free(ast );
16 void tree_code(ast );
17 double tree_result(ast );
18 ast new_node(char *, ast, ast);
19
20 #endif
```

Listing 8.2. Programm-Code zum Operator-Baum (ast/term-ast.c)

```
1  /*******************************************************
2  Syntaxbaum - Bibliothek
3  *******************************************************/
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include "term-ast.h"
8
9  void tree_output(ast p, int n) {
10 // Baum-Traversierung First-Order
11   if (p != NULL) {
12     printf("%s", "                "+14-2*n);
13     printf("%s\n", p->text);
14     tree_output(p->l, n+1);
15     tree_output(p->r, n+1);
16   }
17 }
18
19 void tree_code(ast p) {
20 // Baum-Traversierung Post-Order in UPN
21   if (p != NULL) {
22     tree_code(p->l);
23     tree_code(p->r);
24     printf("%s\n", p->text);
```

```
25    }
26  }
27
28  double tree_result(ast p) {
29      double erg;
30      switch (p->text[0]) {
31      case '+': erg = tree_result(p->l) + tree_result(p->r); break;
32      case '-': erg = tree_result(p->l) - tree_result(p->r); break;
33      case '*': erg = tree_result(p->l) * tree_result(p->r); break;
34      case '/': erg = tree_result(p->l) / tree_result(p->r); break;
35      case 'C': erg = -tree_result(p->l); break;  // Change-Sign
36      default:  erg = atof(p->text);              // Zahl
37      }
38      return erg;
39  }
40
41  ast new_node(char *t, ast l, ast r) {
42      ast p = (ast) malloc(sizeof(ast_node));
43      if (p != NULL) {
44          p->l = l, p->r = r;
45          strcpy(p->text, t);
46      }
47      return p;
48  }
49
50  void tree_free(ast p) { // Baum löschen
51      if (p != NULL) {
52          tree_free(p->l);
53          tree_free(p->r);
54          free(p);
55      }
56  }
```

Ein Compiler, der in einem ersten Schritt den Operator-Baum erstellt und diesen anschließend mehrfach verarbeiten lässt findet sich in Listing **??**.

Listing 8.3. Compiler zur Operator-Baum-Erstellung (ast/term-treegen.c)

```
1  /*******************************************************
2  RD-Parser für Terme
3  Baut binären Operator-Syntaxbaum aus Term-Ausdruck
4  *******************************************************/
5  #include <stdio.h>
6  #include "term-ast.h"
7  #include "term-scanner.h"
8
9  // Prototypen
10 void scanner();
11 ast f_start();
12 ast f_factor();
13 ast f_expression();
14 ast f_term();
15
16 // Globale Daten
17 char zahl[20];
```

```
18 int token, error;
19
20 // Hauptprogramm
21 int main() {
22     char text[100];
23     ast s;
24
25     while(printf("Input:"), fgets(text,255,stdin)!= NULL) {
26         term_scanner(text, NULL); // Scanner-Reset
27         s = f_start();
28         if (!error) {
29             printf("Syntaxbaum:\n");
30             tree_output(s, 0);
31             printf("UPN-Codeausgabe\n");
32             tree_code(s);
33             printf("Ergebnis: %lf\n", tree_result(s));
34         }
35         else
36             printf("Syntax nicht OK, error=%d\a\n", error);
37         tree_free(s);
38     }
39     return 0;
40 }
41
42 // Funktionen
43
44 void scanner() {
45     token = term_scanner(NULL, zahl);
46 }
47
48 ast f_factor () {
49     ast p;
50     if (token == PLUS) {
51             scanner();
52        p = f_factor();
53     }
54     else if (token == MINUS) {
55             scanner();
56             p = new_node("CHS", f_factor(), NULL);
57     }
58     else if (token == ZAHL) {
59         p = new_node(zahl, NULL, NULL);
60         scanner();
61     }
62     else if (token == KLA_AUF) {
63         scanner();
64         p = f_expression();
65         if (token == KLA_ZU)
66             scanner();
67         else
68             error = 1;
69     }
70     else error=2;
71     return p;
```

```
72 }
73
74 ast f_term() {
75     ast p, p2;
76     char * txt;
77     p = f_factor();
78     while(token == MAL || token == DIV) {
79         txt = (token == MAL) ? "*" : "/";
80         scanner();
81         p = new_node(txt, p, f_factor());
82     }
83     return p;
84 }
85
86 ast f_expression() {
87     ast p, p2;
88     int token2;
89     p = f_term();
90     while(token == PLUS || token == MINUS) {
91         token2 = token;
92         scanner();
93         p = new_node((token2 == PLUS) ? "+" : "-", p, f_term());
94     }
95     return p;
96 }
97
98 ast f_start() {
99     scanner(), error = 0;
100    ast p = f_expression();
101    if (token != END)
102        error = 3;
103    return p;
104 }
```

8.3 AST für PL/0

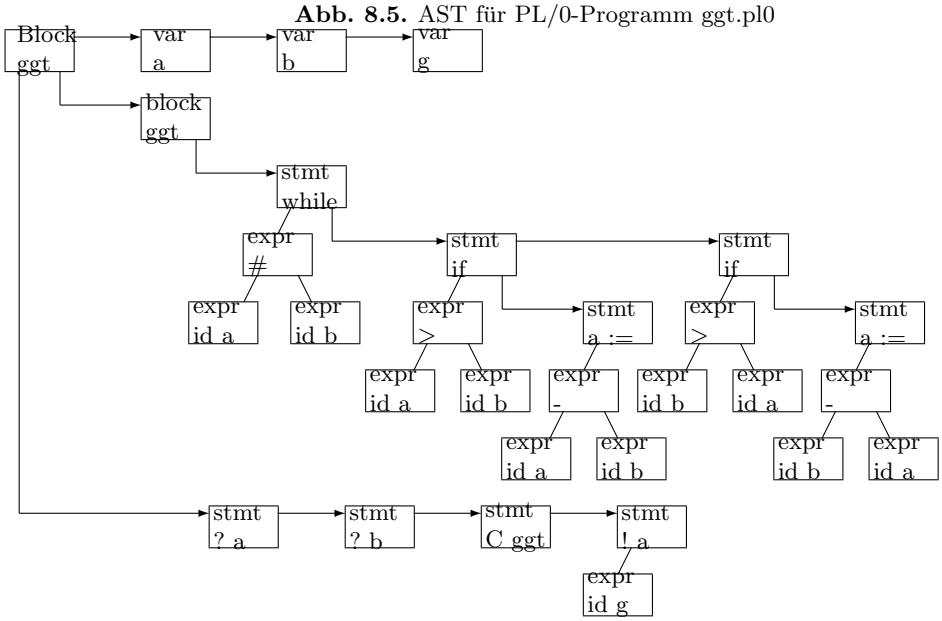
Mehrere Knoten-Arten:

- Block-Knoten
  - Zweiger auf Var-Knoten
- Var-Knoten
  - Zweiger auf Var-Knoten
- statement-Knoten
  - Zweiger auf Statement-Knoten
- Expression-Knoten anaolg zu arithmetischen Termen
  - Zweiger auf Expression-Knoten

Abbildung 8.5 zeigt den Syntaxbaum für das GGT-Programm (Listing 2.2), Seite 16).

Der Syntaxbaum sollte bereits die Level-Offset-Werte der Symboltabelle enthalten, dann wird die spätere Weiterverarbeitung einfacher.

Abb. 8.5. AST für PL/0-Programm ggt.pl0




```
enum {cmd_end, cmd_write, cmd_read, cmd_assign, cmd_if, cmd_while};

struct ast_node{
  int type;                  // Befehl
  int stl;                   // Level der Symboltabelle für assign und read
  int sto;                   // Offset der Symboltabelle für assign und read
  struct ast_node * next;    // Nächster Befehl
  struct ast_node * cond;    // Abhängige Kette für if und while
};
```

8.4 Neu -

8.4.1 Die Knotenarten

Befehlsknoten

- Eingabe
- Ausgabe
- Funktionsaufruf
- Bedingter Sprung
- Unbedingter Sprung
- Sprungziel

- Ziel der Zuweisung
  (Symboltabellen-Informationen, zwei Int)
- Wert der Zuweisung
  EXPR-Zeiger

- Nr der Funktion
  (Symboltabellen-Informationen, zwei Int)
- Name der Funktion
  String
- Ziel der Eingabe
  (Symboltabellen-Informationen, zwei Int)
- Wert der Zuweisung
  EXPR-Zeiger
- Sprungziel
  (Eineindeutige int oder Label=
- Wert der Bedingung
  EXPR-Zeiger
- Sprungziel
  (Eineindeutige int oder Label)
- Sprungziel
  (Eineindeutige int oder Label)

Die Srtuktur der Knoten kann entweder als C-Struktur (Vereinigungsmenge aller Statementknoten) oder als C++-Klasse (per Vererbung) realisiert werden.

Code-Erzeugung

```
void c_adr_var(int stl, int sto, char * name) {
    int i;
    emit("              set I31, P0[0]    # Adr Var " + string(name) + ": Sym "+ string(
    for (i = 0; i < stl; i++)
        emit("              set I31, P0[I31]");
    emit("              sub I31, " + string(itoa(sto+2)));
}
```

orthand

Ausdruck-Knoten

Ausdruck-Knoten enthalten eine der drei Möglichkeiten:

- Operator (innerer Knoten)
- Konstante (Blattknoten)
- Variable (Blattknoten)

Programmstruktur-Knoten

```
VAR a;

BEGIN
  ? a;
  IF a < 0 THEN
```

```
        a := - a;
   ! a;
END.
```

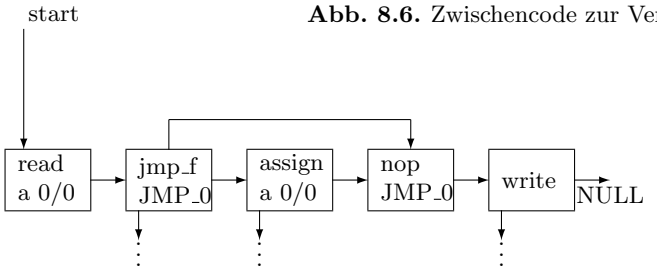


Abb. 8.6. Zwischencode zur Verzweigung

```
VAR a;

BEGIN
   a := 1;
   WHILE a <= 10 DO
   BEGIN
        ! a;
        a := a + 1;
   END;
END.
```

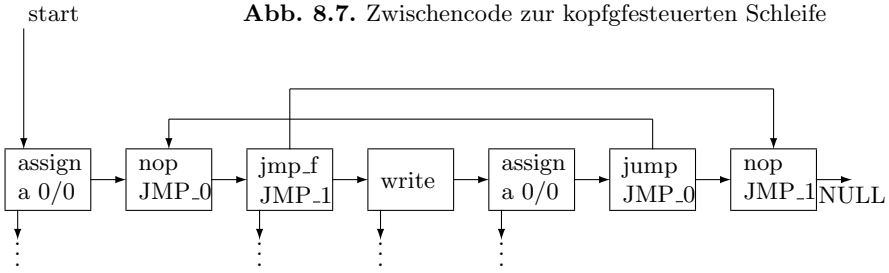


Abb. 8.7. Zwischencode zur kopfgfesteuerten Schleife

9

Speicherverwaltung

Bis jetzt ist unser Compiler in der Lage, ein Quellprogramm in einen AST abzubilden. Dieser AST enthält bereits die Informationen der Symboltabelle, in welchem Level und an welcher Stelle die Variable zu suchen ist. Bevor nun die Codeerzeugung erfolgt, muss noch geklärt werden, wie die Variablen des Quellprogramms auf den Speicher des Zielsystems abgebildet werden. Betrachten wir noch einmal Abbildung **??** (Seite **??**), so wird schnell klar, dass hier nur über eine eindeutige Adresse ein Zugriff auf den Speicher erfolgen kann.

9.1 Einführung

Es gibt zwei Ansätze in der Speicherverwaltung. Je nachdem, ob die Adresse einer Variablen zur Compiletime oder zur Runtime errechnet wird, spricht man von statischer und von dynamischer Speicherverwaltung.

Moderne Programmiersprachen bieten meist beide Arten an. Wird in C eine Variable ohne weitere Angabe deklariert (etwa `int a;`), so wird diese Variable dynamisch verwaltet. Teilweise erden diese Variablen auch "automatische" Variablen genannt [KR90]. Durch Angabe des Schlüsselworts `static` kann für die Variable aber auch eine statische Verwaltung gewählt werden (etwa `static int b;`).

Beide Varianten haben Vor- und Nachteile:

- Da die Adresse von statischen Variablen zur CT errechnet werden kann ist der Variablenzugriff schneller.
- Ein Rekursionsstack ist nur mit dynamischer Speicherverwaltung möglich. Im Falle von Rekursion kommt eine Variable des Quellprogramms mehrfach im Hauptspeicher vor.
- Da die Lebenszeit von Variablen zur RT unterschiedlicher Funktionen sich oft nicht überlappt, kommt dynamische Speicherverwaltung evtl. mit weniger Hauptspeicher aus.

Es muss also gut überlegt werden, ob man eine Sprache mit statischer, dynamischer oder beliebiger Speicherverwaltung entwickelt.

9.2 Gültigkeitsbereiche

Vereinbarungen:

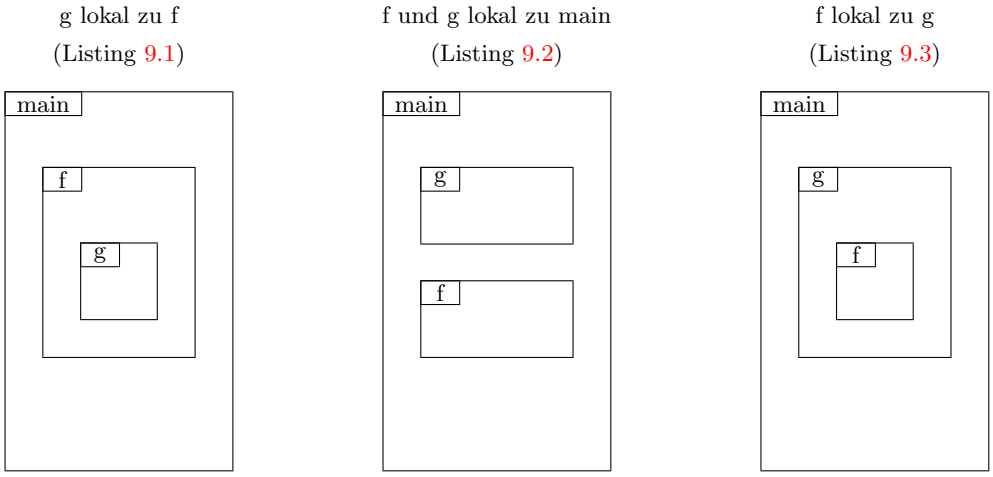
- f ruft g
- Die Level seien L_f und L_g

- Variablen in
  - Hauptprogramm: a, d
  - f: b,d;
  - g: c,d;

Problem: wegen Rekursion kann Speicheradresse zur Compilezeit nicht erstellt werden Lösung: Im Hauptspeicher muss Dynamische Kette (Dynamic Link, kurz DL) als Stack geführt werden. Zusätzlich muss Statische Kette (Static Link, kurz SL) geführt werden Neues Problem: wie müssen SL-Zeiger gesetzt werden?

Analyse:

Abb. 9.1. Schachtelungsmöglichkeiten von Funktionen



Listing 9.1. g lokal zu f (pl-0/programme/speicher-g-in-f.pl0)

```
1  (* g lokal zu f *)
2  VAR a, d;
3
4      PROCEDURE f;
5      VAR b, d;
6
7          PROCEDURE g;
8          VAR c, d;
9          BEGIN
10                             a := 12;
11                             b := 22;
12                             c := 32;
13             d := 42;
14                             DEBUG;     (* Hauptspeicherausgabe *)
15                             ! a;
16                             ! b;
17             ! c;
18             ! d;
19         END;
20
21     BEGIN
```

```
22          a := 11;
23                      b := 21;
24          d := 41;
25                      DEBUG;              (* Hauptspeicherausgabe *)
26          CALL g;
27                      ! a;
28                      ! b;
29                      ! d;
30      END;
31
32  BEGIN
33      a := 10;
34      d := 40;
35          DEBUG;                          (* Hauptspeicherausgabe *)
36      CALL f;
37      ! a;
38      ! d;
39  END.
```

Listing 9.2. f und g lokal zu main (pl-0/programme/speicher-f-g-in-main.pl0)

```
1   (* f und g global *)
2   VAR a, d;
3       PROCEDURE g;
4       VAR c, d;
5       BEGIN
6           a := 12;
7                       c := 32;
8                       d := 42;
9                       DEBUG;      (* Hauptspeicher *)
10          ! a;
11          ! c;
12          ! d;
13      END;
14
15      PROCEDURE f;
16      VAR b, d;
17      BEGIN
18                      a := 11;
19                      b := 21;
20          d := 41;
21                      DEBUG;      (* Hauptspeicher *)
22          CALL g;
23                      ! a;
24                      ! b;
25                      ! d;
26      END;
27
28  BEGIN
29      a := 10;
30      d := 40;
31          DEBUG;                  (* Hauptspeicher *)
32      CALL f;
33      ! a;
34      ! d;
```

```
35 END.
```

Listing 9.3. f lokal zu g (pl-0/programme/speicher-f-in-g.pl0)

```
1  (* f lokal zu g ruft g !!!! Indirekte Rekursion !!! *)
2  VAR a, d;
3
4      PROCEDURE g;
5      VAR c, d;
6
7          PROCEDURE f;
8          VAR b, d;
9
10
11         BEGIN
12                         a := 11;
13                         b := 21;
14                         c := 31;
15                         d := 41;
16                         DEBUG;          (* Hauptspeicher *)
17                         CALL g;
18         END;
19
20
21     BEGIN
22                 c := 32;
23                 d := 42;
24         IF a = 10 THEN
25                 BEGIN
26                     DEBUG;              (* Hauptspeicher *)
27                     CALL f;
28                 END;
29                 IF a = 11 THEN
30                 BEGIN
31                     a := 12;
32                     DEBUG;              (* Hauptspeicher *)
33                 END;
34
35
36     END;
37
38 BEGIN
39     a := 10;
40     d := 40;
41         DEBUG;                          (* Hauptspeicher *)
42     CALL g;
43     ! a;
44     ! d;
45 END.
```

9.3 Statische Speicherverwaltung

Eine Möglichkeit der Speicherzuweisung ist, die Adresse der Variablen bereits zur Compiletime zu errechnen. Man spricht hier von statischer Speicherverwaltung.

Für statische Speicherverwaltung sollte die Symboltabelle etwas modifiziert werden. Ihr grundsätzlicher Aufbau ist zwar immer noch wie inb Abbildung 7.1 (Seite 92) gezeigt, es muss aber zusätzlich für die Variablen eine laufende Nummer mitgespeichert werden. Im Zwischencode genügt es nun, für die Variablen anstelle von Level-Δ und Offset einfach diese laufende Nr zu speichern.

Abb. 9.2. Symboltabelle bei statischer Hauptspeicherverwaltung (Listing 9.1)

4							
3							
2	f	proc	g	proc			
1	d	var 1	d	var 3	d	var 5	
0	a	var 0	b	var 2	c	var 4	
Level	**Ebene 0**		**Ebene 1**		*** Ebene 2**		**Ebene 3**

Abb. 9.3. Statischer Hauptspeicher g lokal zu f (Listing 9.1)

Alle Zeilen

17		
16		
15		
14		
13		
12		
11		
10		
9		
8		
7		
6		
5	42	g d
4	32	g c
3	41	f d
2	22	f b
1	40	main d
0	12	main a

9.4 Dynamische Speicherverwaltung

Im Hauptspeicher muss für Rücksprünge etc. ein Stack aufgebaut werden. Der Gültigkeitsbereich der Variablen ist aber nicht identisch mit der Stack-Abfolge.

Daher muss im Hauptspeicher ein Zahlen-Tupel für jede Funktion gespeichert sein:

- Zeiger zur Stackelement der aufrufenden Funktion (Dynamic Link)
- Zeiger zur Stackelement der aus Sicht des namensbvereichs umgebenden Funktion (Static Link)

Dieses Tupel wird zusammen mit den lokalen Variablen und weiteren Daten - etwa der Rücksprungadresse RS - im sog. “Activation-Record“ (kurz AR) gespeichert. Je nach konkreter Anwendung - Haupspeicher eines auszuführenden Programms, Interpreter, Cross-Compiler etc. müssen im AR weitere Informationen gespeichert werden. Wenn z.B. ein Prozessor im Assembler lediglich eine “GOTO“-Befehl kennt, aber keinen “GOSUB“, so muss die Rücksprungadresse zusätzlich im AR gespeichert werden.

Wir organisieren unseren Stack aufsteigend, am oberen Ende eines jeden AR findet sich SL und DL.

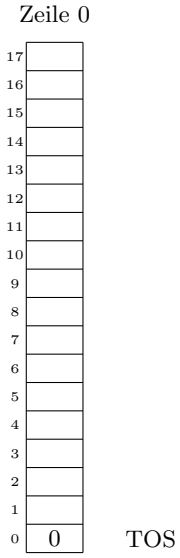
Abbildung 9.5 bezieht sich auf Listing 9.1.

Abbildung 9.6 bezieht sich auf Listing 9.2.

Abbildung 9.7 bezieht sich auf Listing 9.3.

Zum Programmstart ist der Hauptspeicher bei allen Programmen leer, wie in Abbildung 9.4 dargestellt. Beim Aufruf einer Funktion / Prozedur - und das gilt auch für das Hauptprogramm - wird zuallererst ein neues Segment im Hauptspeicher angelegt. In Adresse 0 im Hauptspeicher wird Top-of-Stack gespeichert, welches am Programmstart mit 0 initialisiert wird (Abbildung 9.4).

Abb. 9.4. Leerer Hauptspeicher



Anhand von Level-Δ und Offset aus der Symboltabelle kann nun über die SL-Kette und Offset der Speicherplatz eines jeden Bezeichners zur Laufzeit ermittelt werden.

Analyse der SL-Kette für den Zugriff auf die Variable a innerhalb der Funktion g:

$L_f = L_g - 1$ (g lokal zu f):

- Symboltabellen-$\Delta = 2$
- Zweimal entlang SL-Kette gehen

$L_f = L_g$ (f und g lokal zu main):

- Symboltabellen-$\Delta = 1$
- Einmal entlang SL-Kette gehen

$L_f = L_g + 1$ (f lokal zu g):

Abb. 9.5. Hauptspeicher g lokal zu f (Listing 9.1)

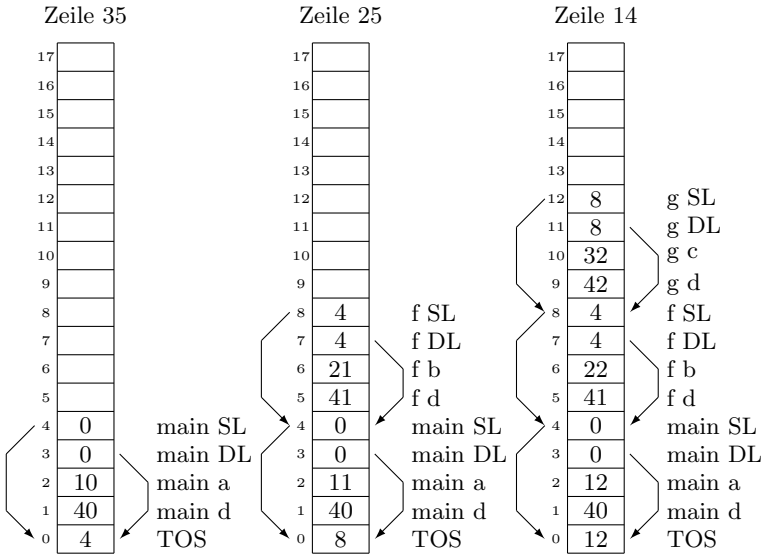
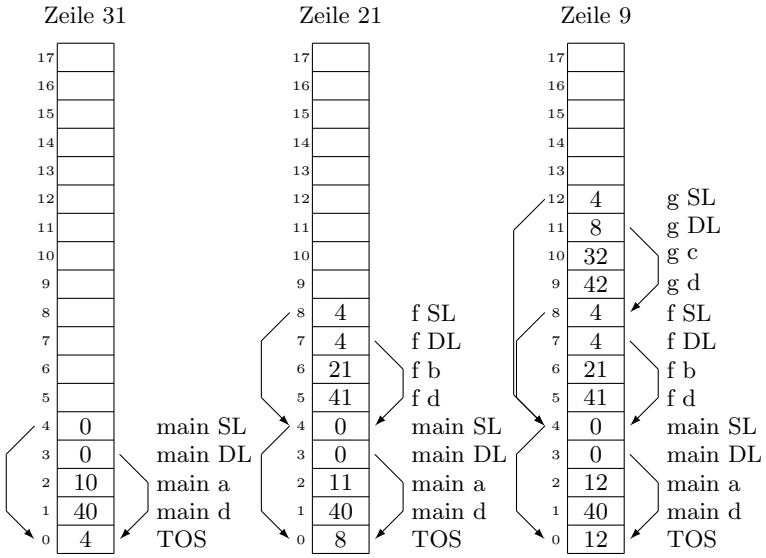


Abb. 9.6. Hauptspeicher g und f lokal zu main (Listing 9.2)



- Symboltabellen-$\Delta = 1$
- Einmal entlang SL-Kette gehen, unabhängig von Rekursionslevel

Analyse der SL-Kette für den Funktionsaufruf “f ruft g auf“:

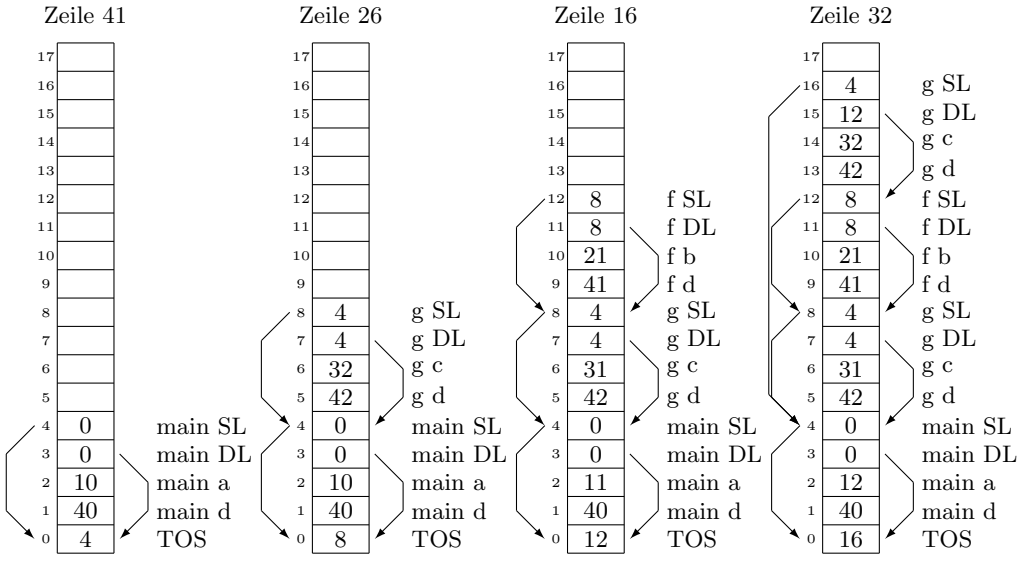
$L_f = L_g - 1$ (g lokal zu f):

- Symboltabellen-$\Delta = 0$
- Nullmal entlang SL-Kette gehen

$L_f = L_g$ (f und g lokal zu main):

- Symboltabellen-$\Delta = 1$
- Einmal entlang SL-Kette gehen

Abb. 9.7. Hauptspeicher f lokal zu g (Listing 9.3)



$L_f = L_g + 1$ (f lokal zu g):

- Symboltabellen-$\Delta = 2$
- Zwei entlang SL-Kette gehen, unabhängig von Rekursionslevel

Allgemein:

- Es kann beliebig nach außen gesprungen werden, aber nur einen Ebene nach innen
- $L_f = L_g + \Delta,\ -1 \leq \Delta < \infty$: $L_f - L_g + 1$ mal entlang SL-Kette gehen

9.4.1 Methoden im Hauptspeicher

Algorithmus 12: Berechnung der Adresse einer Variablen

int ram_var_adr(delta, nr)

```
Parameter
  delta, nr  {Informationen der Symboltabelle}
adr = RAM[0]
WDH delta mal
    adr = RAM[adr]
adr = adr - AR-Größe
adr = adr - nr
adr
```

Algorithmus 13: Anlegen eines neuen Hauptspeichersegments

void ram_neusegment(n, delta)

```
Parameter
  n       {Anzahl der Variablen}
  delta   {Information der Symboltabelle}
RAM[RAM[0] + n + AR-Größe -1] = RAM[0] (DL)
adr = RAM[0]
WDH delta mal
    adr = RAM[adr]
RAM[RAM[0] + n + AR-Größe] = adr (SL)
RAM[0] = RAM[0] + n + AR-Größe (TOS)
```

Algorithmus 14: Löschen eines Hauptspeichersegments

void ram_loeschsegment()

```
RAM[0] = RAM[RAM[0]-1] (TOS)
```

9.5 Statische vs. Dynamische Speicherverwaltung

Tabelle 9.1. Vor- und Nachteile von statischer und dynamischer Speicherverwaltung

	Pro	Contra
Statisch	• Einfach zu programmieren • Schnell zur Laufzeit	• Rekursion schwierig • Verschwendung von Speicherplatz
Dynamisch	• Effiziente Speicherplatznutzung • Rekursion einfach	• Schwierig zu programmieren • Langsam zur Laufzeit

10

Code-Optimierung

Die Optimierung der generierten Codes ist sehr stark von der Zielmaschine abhängig. Teilweise ist “Intelligenz“ zur Optimierung notwendig.

Die Code-Optimierung ist in Teielen bereits im Zwischencode möglich (z.B. Vereinfachung von Ausdrücken), in Teilen aber auch erst nach der Code-Erzeugung. Optimierung im Zwischencode ist damit von der Zielmaschine unabhängig.

10.1 Optimierung der Kontrollstrukturen

- Invarianter Code in Schleifen
- If-else-if Schachtelung statistisch wählen
- Mehrfachverzeigung statt if-else-id bei Vergleichen mit Konstanten

10.2 Optimierung von Sprüngen

10.2.1 Optimierung von NOPs

In vielen Ziel-Sprachen werden NNo-OperationStatements (kurz NOPs) als Sprungziele verwendet. Diese NOPs sollten im Zuge der Optimierung entfernt werden.

NOP vor Optimierung

```
# I0 enthält einen Wert, dessen
# Betrag berechnet werden soll.
     lt I0, 0, BOTTOM
     mul I0, -1
LBL: NOP   # Nicht in Parrot!!!
     print I0
```

NOP nach Optimierung

```
# I0 enthält einen Wert, dessen
# Betrag berechnet werden soll.
     lt I0, 0, BOTTOM
     mul I0, -1

LBL: print I0
```

10.2.2 Eliminierung unnötiger Sprünge

Verzweigungen Am Schleifenende haben unnötige Sprünge zur Folge.

GGT vor Optimierung

```
# GGT-Berechnung von I0 und I1
TOP:  eq I0, I1, END
      lt I0, I1, L1
      sub I0, I1
      branch L2
L1:   sub I1, I0
L2:   branch TOP

END: print I0
```

GGT nach Optimierung

```
# GGT-Berechnung von I0 und I1
TOP:  eq I0, I1, END
      lt I0, I1, L1
      sub I0, I1
      branch TOP
L1:   sub I1, I0
      branch TOP

END: print I0
```

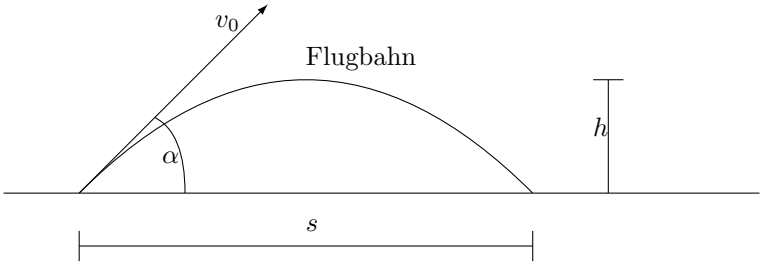
10.3 Optimierung von Ausdrücken

- Suche identischer Bäume
- Zwischenergebnisse abspeichern
- Geschickte arithmetische Umformungen:
  - Quadratbildung oder Multiplikation?
  - Vermeidung doppelter Rekursion

$$a^b = \begin{cases} 1 \text{ für } b = 0 \\ (a^{\frac{b}{2}})^2 \text{ für } b > 0, b \textit{ gerade} \\ a * (a^{\frac{b-1}{2}})^2 \text{ für } b > 0, b \textit{ ungerade} \end{cases}$$

Listing 10.1. Kekursives Potenzieren nach Lagrange (ueb-lagrange.c)

```
1  #include <stdio.h>
2
3  int lagrange_potenz(int a, int b) {
4      int p;
5      if (b == 0)
6          p = 1;
7      else if (b % 2 == 0) // b > 0 und gerade
8          p = lagrange_potenz(a, b / 2), p *= p;
9      else  // b > 0 und ungerade
10         p = lagrange_potenz(a, (b - 1) / 2), p *= (a * p);
11     return p;
12 }
13
14 int main() {
15   int basis, exponent;
16   printf("Gib Basis und Exponent ein: ");
17   scanf("%d%d", &basis, &exponent);
18   printf("%d ^ %d = %d\n", basis, exponent, lagrange_potenz(basis, exponent));
19   return 0;
20 }
```



Die physikalischen Formeln lauten:

$$h = \frac{(v_0 sin\ \alpha)^2}{2g} \qquad s = \frac{v_0^2 sin\ 2\alpha}{g}$$

- Wird die Quadrierung über die Potenzierung abgebildet so wird über Logarithmen langsam gerechnet.
- Wird die Quadrierung über Multiplizierung abgebildet so müssen insgesamt drei Multiplikationen durchgeführt werden und sin(α) muss zweimal berechnet werden.

Die Berechnung von h geht am schnellsten über eine temporäre Variable:

```
temp := v0 * sin(alpha);
h = temp * temp / (2 * g);
```

Auf diese Weise werden nur zwei Multiplpikationen benötigt und sin(α) muss nur einmal berechnet werden.

10.4 Optimierung des Speicherzugriffs

RAM / Register / HEAP / Stack

11

Code-Erzeugung

Am Ende einer Comilierungsphase muss der AST in irgendeiner Form in eine Ausgabe gebracht werden. Dies geschieht in der sog. Code-Erzeugung (siehe auch Abbildung 1.1 Seite fig phasen compiler). Wir werden drei grundsätzlich verschiedene Arten der Code-Erzeugung diskutieren:

- Direkte Interpretierung des AST
- Erzeugung von Assembler-Code
- Erzeugung einer anderen Hochsprache

Natürlich wären noch weitere Arten der Verarbeitung möglich, etwa Erzeugung von Cross-ReferenzTabellen o.ä.

11.1 Interpretierung

Sprachen die keinerlei Sprünge kennen, können direkt während des Parse-Vorgangs interpretiert werden. Entfernt man aus der PL/0-Grammatik (Grammatik Γ_1 Seite 15) aus der Block-Regel die Prozedur-Deklaration sowie die Befehle `IF` und `WHILE` so entsteht ein linear ablaufendes Programm.

11.2 Assembler-Erzeugung

Der zu erzeugende Assembler wurde bereits in Kapitel **??** vorgestellt.

Soll Assembler- oder Maschinencode erzeuigt werden, so muss zwischen zwei grundsätzlich verschiedenen Ziel-Sprachen unterschieden werden:

- Sprachen mit symbolischen Sprungziel

```
          LOADR  0
          JMPZ  LABEL_A
          READ
LABEL_A   NOP
          WRITE
```

- Sprachen mit numerischem Sprungziel

```
0000      LOADR  0
0001      JMPZ  0003
```

```
0002    READ
0003    NOP
0004    WRITE
```

Das Problem entsteht bei den Vorwärts-Sprüngen des JMPZ-Befehls. Wenn die Zielsprache symbolische Sprungziele unterstützt sind Vorwärts-Sprünge einfach zu generieren. Die Code-Ausgabe erzeugt ein eindeutiges Sprungziel (ein sog. Label) - etwa durch Erhöhen eines globalen Zählers - und erzeugt die Code-Zeilen mit Sprung und Sprungziel.

Wesentlich schwieriger wird dies bei numerischen Sprungzielen, also der konkreten Zeilennummmer. Da die Zahl der Code-Zeilen, die übersprungen werden sollen, nicht bekannt ist, bleibt nur, die Sprunganweisung zweimal auszugeben. Zum ersten mal mit einem vorläufigen (falschen!!!!) Sprungziel. Erst wenn dann später das Sprungziel bekannt ist, kann das Sprungziel der ersten Ausgabe korrigiert werden.

Rückwärts-Sprünge sind einfacher zu generieren. Bei symbolischen Sprungzielen wird ebenfalls ein Label generiert und später beim JUMP referenziert. Müssen Sprungziele numerisch angegeben werden so hilft eine Merk-Variable.

11.2.1 Der Emitter

```
int emit(int _nr, command_code cmd, int arg, char * comment);
```

Ist die übergebene Zeilen-Nummer `_nr` negativ, so wird an das Ende der Ausgabe angehängt und die Nummer der ausgegebenen Zeile zurückgegeben. Im anderen Fall - bei einer Zeilen-Nummer ≥ 0 wird in genau diese Zeile ausgegeben. Damit kann bei numerischen Sprungzielen die spätere Korrektur eines Sprungs einfach erfolgen.

11.2.2 Ausdrücke

Ausdrücke werden wie schon behandelt (bei Register-Stack-Masachinen) in UPN umgesetzt.

11.2.3 if-Statement

Der Code "`if (<Ausdruck>) <Block>`" wird in Assembler dadurch umgesetzt, dass die Bedingung ausgewertet wird und der folgende (bedingte) Code übersprungen wird, falls die Bedingung nicht erfüllt ist. Während in der Hochsprache codiert und "gedacht" wird "Wenn Bedingung erfüllt dann führe den Code aus", ist das Verfahren im Assembler also genau anders herum: "Wenn die Bedingung nicht erfüllt ist dann überspringe den folgendne Code". Dies ist in Abbildung 11.1 als Ablaufplan und Assembler dargestellt.

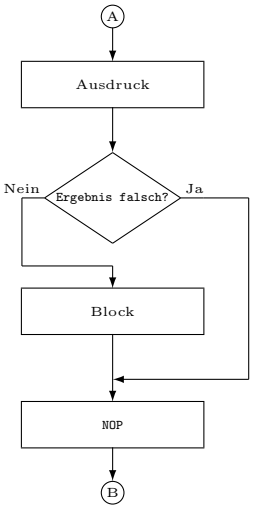
Analysiert man die Assembler-Zeilen 1 bis 7, so stellt man fest, dass nur die Zeilen 4 und 7 für die eigentliche if-Anweisung notwendig sind:

- Zeilen 1 bis 3 sind die Abarbeitung des Bedingungs-Ausdrucks
- Zeilen 5 und 6 sind die bedingte Anweisung

Betrachten wir einmal die Code-Erzeugung direkt im Parser, also ohne Verwendung eines Zwischencodes, da hier das erklären einfacher ist. Listing **??** zeigt eine Funktion, die in einem Recursive-Descent-Parser das Statement if verarbeitet. Durch Aufruf der Funktion `f_condition()` wird der Code für die Verzweigung erzeugt, durch Aufruf der Funktion `f_sattement()` der Code für die Anweisung. Es muss also nur noch der bedingte Sprung sowie das zugehörige Sprungziel erzeugt werden.

Abb. 11.1. Ablaufplan und Assembler Einfache Verzweigung

Ablaufplan if




```
if a > 0 then
   b := 1;

01 LOADV a
02 LOADC 0
03 CMPGT
04 JUMPZ 07
05 LOADC 1
06 STOREV 2
07 NOP
```

```
1 void f_if() {
2         match(t_if, e_if_req);              // Lies weiter
3         f_condition();                      // Code-Ausgabe der Bedingung
4         match(t_then, e_then_req);          // Lies weiter
5         lnr1 = emit(-1, cmd_JMPZ, 0, ""); // Sprungziel falsch!!
6         f_statement();
7         lnr2 = emit(-1, cmd_NOP, 0, "Sprungziel");
8         emit(lnr1, cmd_JMPZ, lnr, "");     // Sprungziel richtig
9 }
```

11.2.4 if-else-Statement

11.3 Schleifen

Der Code "`while '(' <Ausdruck> ')' <Block>`" muß folgendermaßen umgesetzt werden:

11.4 Variablenzugriff

Wird dynamische Speicherverwaltung eingesetzt so ist der Variablenzugriff relativ kompliziert, da die Adresse einer Variablen erst zur Runtime errechnet werden kann. Je nachdem wie kompliziert der Code ist und abhängig davon, wie einfach in der Zielsprache ein Unterprogrammaufruf ist bieten sich zwei Möglichkeiten an:

- Der Code kann über ein Unterprogramm in der Codeausgabe für jeden Variablenzugriff generiert werden
- Der Code kann über ein Unterprogramm in der Zielsprache einmalig generiert werden, dieses Unterprogramm wird nun über einen Funktionsaufruf in der Zielsprache aufgerufen.

Abb. 11.2. Ablaufplan Verzweigung mit Alternative

Ablaufplan if-else

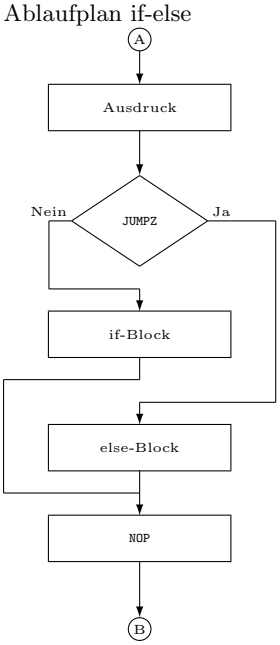
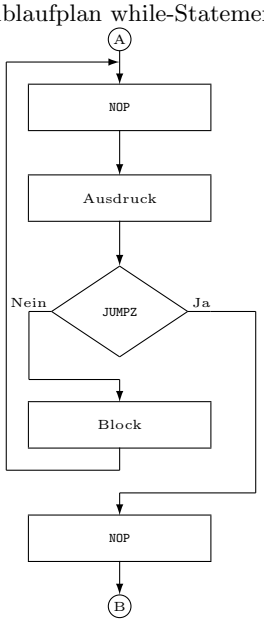


Abb. 11.3. Ablaufplan kopfgesteuerte schleife

Ablaufplan while-Statement



Im Beispiel-Assembler der Vorlesung ist der Funktionsaufruf mit zwei Parametern mindestens so kompliziert wie die direkte Code-Ausgabe, diese besteht minimal - abhängig von der Δ-Information der Symboltabelle - aus fünf Assemblerbefehlen. Listing 11.1 zeigt den Zugriff.

Listing 11.1. Codeerzeugung für Variablenzugriff

```
1  void c_adr_var(int stl, int sto, char * name) {
2      int i;
3      char comment[255];
4      sprintf(comment, "Adr Var %s: Sym %d/%d", name, stl, sto);
5      emit(-1, cmd_LOADR, 0, comment);
6      for (i = 0; i < stl; i++)
7          emit(-1, cmd_LOADS, 0, "");
8      emit(-1, cmd_LOADC, 2, "Offset wegen SL/DL/RS"); // Offset anpassen
9      emit(-1, cmd_SUB, 0, "");
10     emit(-1, cmd_LOADC, sto, "Symtab-Offset");
11     emit(-1, cmd_SUB, 0, "ADR VAR");
12 }
```

Nach Abarbeitung des von Listing 11.1 erzeugten Codes liegt die Adresse der Variablen auf dem Stack, danach kann mit `LOADS` der Werr der Variablen auf den Stack geladen werden bzw. mit `STORES` der (vorher errechnete und damit unter der Adresse auf dem Stack liegende) Wert gespeichert werden.

11.5 Funktionsaufruf

- Der Speicherbedarf des Stack-Segments muss ermittelt werden
- Neuer DL := Alter TOS
- Neuer SL := Alter TOS und Δ mal entlang der SL-Kette gehen
- Evtl Rücksprung-Adresse auf Stack legen
- TOS erhöhen

11.6 Sprünge

Bei der Codeerzeugung müssen Sprünge in die Ausgabe eingebaut werden. Nun muss man unterscheiden, ob die Sprungziele wie in einem üblichen Assemler symbolisch sein können oder ob bereits eine Zeilennummer als Sprungziel angegeben werden muss.

So könnte der Assemblercode für eine einfache Verzweigung

```
IF a < 0 THEN
  a := -a;
```

analog Abbildung 11.1 folgfendermaßen aussehen:

```
LOADV a
LOADC 0
CMTLT
JUMPZ LBL_1
LOADV a
```

```
        CHS
        STOREV a
LBL_1   NOP
```

Durch eine fortlaufende Numerierung der Labels (numerisch oder alphabetisch) ist die Codeerzeugung einfach zu bewerkstelligen.

```
void f_if() {
    int label = get_next_label();
    match(t_if);
    f_expression();
    printf("         JUMPZ   LBL_%d\n", label);
    match(t_then);
    f_statement();
    printf("LBL_%d   NOP\n");
}
```

Die Codeausgabe für eine Verzweigung besteht also nur aus einem bedingten Sprung und einem Sprungziel. Das Schachteln von Verzweigungen und Schleifen in der Quellsprache ist unkritisch, da die Variable **label** in obigem Codefragment lokal ist.

Anders sieht es aus, wenn keine symbolischen Sprungziele erlaubt sind, sondern bereits Zeilennummern als Sprungziel angegeben werden müssen. Der Assemblercode sieht in dann beispielhaft wie folgt aus:

```
0000    LOADV a
0001    LOADC 0
0002    CMTLT
0003    JUMPZ 0007
0004    LOADV a
0005    CHS
0006    STOREV a
0007    NOP
```

Hier entsteht nun das Problem, dass das Sprungziel des bedingten Sprungs zum Zeitpunkt der Ausgabe noch gar nicht bekannt sein kann. Daher muss der Sprung erst einmal mit einem falschen (da nicht bekannten) Sprungziel ausgegeben werden. Nachdem nun der Code des Statements ausgegeben ist kann erst das Sprungziel ausgegeben werden und anschließend muss der Bedinge Sprung noch einmal ausgegeben werden. Man benötigt daher eine spezielle Funktion für die Ausgabe, die sog. Emitter-Funktion die beispielhaft so aussehen könnte:

```
int emit(int _nr, char * txt, char * arg) {
    const int linewidth = 23;    // Fr Windows anpassen
    static int nr = -1;          // Nr der Ausgabezeile
    if (_nr < 0) // Neue Zeile
        _nr = ++nr;
    fseek(out_file, linewidth * _nr, SEEK_SET);
    fprintf(out_file, "%04d %-6s %10d\n", _nr, txt, arg);
    return nr;
}
```

Mit einer solchen Emitter-Funktion kann nun die Codeausgabe auch mit Zeilennummern programmiert werden:

```
void f_if() {
    int label het_next_label();
    match(t_if);
```

```
    f_expression();
    lnr_1 = emit(-1, "JUMPZ", itoa(0)); // Sprungziel noch falsch!!!
    match(t_then);
    f_statement();
    lnr_2 = emit(-1, "NOP", "");
    emit(lnr_1, "JUMPZ", itoa(lnr_2)); // Sprungziel korrigiert!!!

}
```

12

Generator-Tools

12.1 Einleitung

12.2 Generator-Tools

[Her95] Lex und Yacc, Flex, Bison, JLex, ...

Abb. 12.1. Verarbeitung einer Lex-Datei

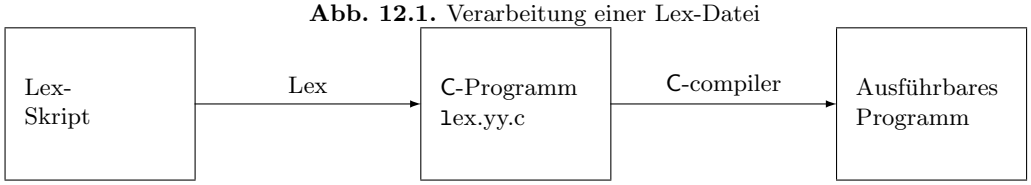
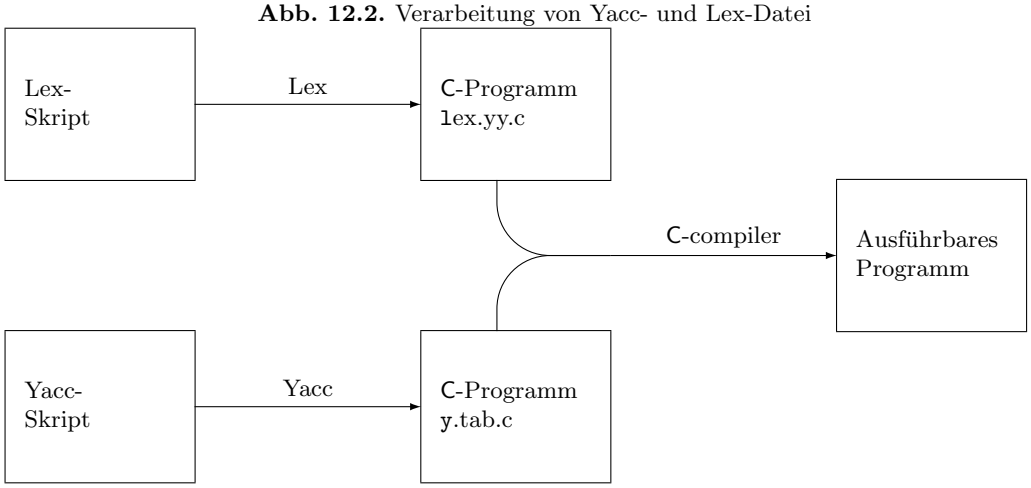


Abb. 12.2. Verarbeitung von Yacc- und Lex-Datei



12.3 Lex

12.3.1 Allgemeines

- Entwickelt in den 1970ern von Lesk / Schmidt bei Bell [LS75]
- Häufig Varianten im Einsatz, etwa GNU-Lex oder JLex

12.3.2 Grundaufbau einer Lex-Datei

```
<Definitionen, C-Code>
%%
<RegExp-Aktions-Paare>
%%
<Zusatz-Code>
```

Zentraler Punkt ist der Mittelteil, der aus Regulären Ausdrücken in Kombination mit Aktionen (C-Code) besteht.

12.3.3 Definitionsteil

- C-Code

  ```
  %{
  #include <stdio.h>
  int i;
  %}
  ```

- Reguläre Definitionen

  ```
  DEC_INT [0-9]+
  HEX_INT 0x[0-9A-F]
  ```

- Startbedingungen

  ```
  %start COMMENT
  ```

- Zeichensatz-Tabellen
- Tabellen-Größen

12.3.4 Regelteil

- Besteht aus Regulären Ausdrücken (Regeln) und Aktionen
- Trifft genau ein Regulärer Ausdruck, so wird die entsprechende Aktion ausgeführt.
- Treffen mehrere Regeln, so wird die Aktion der Regel mit dem längsten Match ausgeführt.
- Treffen mehrere Regeln mit gleichlangem Match , so wird die Aktion der oberstenn Regel ausgeführt.
- Trifft keine Regel, so wird die Eingabe einfach ausgegeben.

12.3.5 Reguläre Ausdrücke

- Bestehen aus Zeichen, die in der Eingabe gesucht werden und
- Meta-Zeichen mit speziellen Bedeutungen

Metazeichen

Zeichen	Bedeutung	Beispiel
`\`	ESC-Sequenz	`\n`
`\`	metazeichen-Ausschaltung	`\\`
`^`	Zeilenanfang	`^#include`
`$`	Zeilenende	`TEST$`
`.`	Beliebiges Zeichen	`T.`
`[]`	Zeichenklasse	`[0-9]`
`|`	Oder-Verknüpfung	`der|die`
`()`	Prioritäts-Klammer	`D(er|ie|as)`
`*`	n-fache Wiederholnung, $0 \leq n \leq \infty$	`Too*r`
`+`	n-fache Wiederholnung, $1 \leq n \leq \infty$	`To+r`
`?`	Option, n-fache Wiederholnung $0 \leq n \leq 1$	`Tore?`
`{}`	m-bisn-fache Wiederholnung	`To{1,10}r`
`{}`	Reguläre Definition	`{DEC_INT}`
`/`	Kontext-Operator	`Tor/ für Deutschland`
`""`	String-Operator	`"+"`
`<>`	Start-Bedingung	`<COMMENT>`

12.3.6 Lex-Variablen

`yytext` Gescannter Text

`yyin` Eingabedatei

`yyout` Ausgabedatei

12.3.7 Lex-Funktionen und -Makros

`yywrap` Eingabeende-Funktion

`input` Input-Funktion

`unput` unput-Funktion

`yyles` unput-Funktion

`yymode` Weiterlese-Funktion

ESC-Sequenzen

12.3.8 Startbedingungen

- Definition im Definitionsteil mit `%start`
- Umschalten der Bedingungen mit Makro `BEGIN`
- Regel mit Startbedingung ist nur aktiv, wenn mit `BEGIN` die jeweilige Startbedingung eingeschlaten wurde
- Regel ohne Startbedingung ist immer (!!!!!!) aktiv

- Startbedingungen werden mit `BEGIN 0` ausgeschalten

Beispiel: Inline-Kommentar in C:

Listing 12.1. Lex-Programm zur Extraktion von C-Inline-Kommentaren (lexyacc/inline-comment.l)

```
1  %start COMMENT
2  %{
3   int lnr = 1;
4  %}
5  %%
6  "//"        BEGIN COMMENT;
7  <COMMENT>\n BEGIN 0;
8  <COMMENT>[^\n]*    printf("Zeile %d: '%s'\n", lnr, yytext);
9  .          /* do nothing */;
10 \n         lnr++;
11 %%
```

Der Kontext-Operator

Sucht den Regulären Ausdruck mit zwei Besonderheiten:

- In yytext wird nur der Match bis vor dem Kontextoperator gespeichert
- Das Weiterlesen erfolgt hinter dem Kontextoperator

Listing 12.2. PL/0-Zusweingen (lexyacc/pl0-var-assignment.l)

```
1  %{
2  /* Sucht Zuweisungen an Variablen in PL/0-Programmen */
3   int lnr = 1, nr = 0;
4  %}
5  ID      [A-Za-z_][A-Za-z_0-9]*
6  WHT     [ \t\n\f]
7  %%
8  {ID}/{WHT}*:=   printf("Zeile %d: %s\n", lnr, yytext);
9  :=          nr++;
10 \n          lnr++;
11 .           ;
12 %%
13 int yywrap() {
14     printf("%d Zuweisungen gefunden!\n", nr);
15     return 1;
16 }
```

12.3.9 Beispiele

Ein Term-Scanner

Listing 12.3. Term-Scanner mit Lex (lexyacc/term-scanner.l)

```
1  %{
2  /*********************************************************
```

```
3  LEX-Programm für Berechnung arithmetischer Ausdrücke
4  Erkennt Fixkomma-Zahlen, Grundrechenarten und
5  runde Klammern
6  ******************************************************/
7  //#include "term-scanner.h"
8  #include <string.h>
9  //enum {PLUS = 1, MINUS, MAL, DIV, KLA_AUF, KLA_ZU, ZAHL, FEHLER};
10
11 %}
12 %%
13 "+"                  return PLUS;
14 "-"                  return MINUS;
15 "*"                  return MAL;
16 "/"                  return DIV;
17 "("                  return KLA_AUF;
18 ")"                  return KLA_ZU;
19 [0-9]+("."[0-9]+)?   {strcpy(yylval.t, yytext); return ZAHL;}
20 [ \t\n]               /* do nothing */;
21 .                    return FEHLER;
22 %%
23
24 int term_scanner(char *input,char *text) {
25 // Adaptiert Schnittstelle für Automaten-Beispiele
26     int rc;
27     if (input != NULL) { // Reset
28         yy_scan_string(input);
29         return 0;
30     }
31     rc = yylex();
32     strcpy(text,yytext);
33     return rc;
34 }
35
36 int yywrap() {
37     return 1;
38 }
39
40
41 // int main() {
42 //     int token;
43 //
44 //     while ((token = yylex()) != 0)
45 //       printf("%d '%s'\n", token, yytext);
46 //     return 0;
47 // }
```

Ein PL/0-Scanner

12.4 Yacc

12.4.1 Allgemeines

- Entwickelt in den 1970ern von Johnson bei Bell

- Yet another COmpiler-Compiler
- Häufig Varianten im Einsatz, etwa GNU-Bison oder JYacc
- Yacc produziert einen LALR-Parser
- Yacc Setzt eine Scanner-Funktoin mit dem Prototypen `int yylex();` voraus

12.4.2 Grundaufbau einer Yacc-Datei

```
<Definitionen, C-Code>
%%
Grammatik-Aktions-Tupel
%%
<Zusatz-Code>
```

Zentraler Punkt ist der Mittelteil, der aus einer Grammatik in einer BNF-ähnlichen Notation sowie zugehörigen Aktionen besteht.

12.4.3 Definitionsteil

- C-Code

  ```
  %{
  #include <stdio.h>
  int i;
  %}
  ```

- Token-Definitionen Yacc akteptiert prinzipiell char-Konstanten als Token. Andere Token werden mittels `%token` angelegt:

  ```
  %token t_plus t_mult
  ```

  Die hinter `%token` angegeben Tokens werden mit int-Werten ab 255 aufsteigend belegt. Das Ende-Token word von Yacc mit dem numerischen Wert 0 definiert.

- Datentypen: Häufig haben Scanner-Resultate oder Grammatik-Resultate ein Ergebnis. Da oft verschiedene Datentypen gehandelt werden müssen, bietet sich die union als Datentyp an. Yacc-Direktive ist `%union`.

  ```
  %union {
  int _int;
  double _double;
  char text[20];
  }
  ```

- Startregel `%start`
- Operator-Priorität und -Assoziativität, sofern nicht aus Grammatik ersichtlich

  ```
  %left t_plus
  %right t_power
  ```

- Datentypen der Token (Scanner-Resultate)

  ```
  %token<_int> t_int
  %token<_double> t_double
  %token<_text> t_identifier
  ```

12.4.4 Yacc-Variablen

yylval

12.5 Konflikte in Yacc-Grammatiken

shift-reduce-Konflikte

reduce-reduce-Konflikte

12.5.1 Beispiele

Ein Term-Parser

Listing 12.4. Ein Term-Parser (lexyacc/term-parser.y)

```
1  %{
2  /*******************************************************
3  Parser für arithmetische Ausdrücke
4  *******************************************************/
5  #include <stdio.h>
6  %}
7  %token  ZAHL KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
8  %union {char t[100];}
9  %%
10 expression: term
11           | expression PLUS term
12           | expression MINUS term
13           ;
14
15 term:       factor
16           | term MAL factor
17           | term DIV factor
18           ;
19
20 factor:     ZAHL
21           | KLA_AUF expression KLA_ZU
22           | PLUS factor
23           | MINUS factor
24           ;
25 %%
26 #include "lex.yy.c"
27
28 int yyerror(char *s) {
29     printf("%s\n", s);
30
31 }
32
33 int main() {
34     int rc;
35     rc = yyparse();
36     if (rc == 0)
```

```
37          printf("Syntax OK\n");
38      else
39          printf("Syntax nicht OK\n");
40      return rc;
41  }
```

Ein Term-Compiler

Listing 12.5. Ein Term-Parser (lexyacc/term-compiler.y)

```
1  %{
2  /*******************************************************
3  compiler für arithmetische Ausdrücke: Term -> UPN
4  *******************************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  %}
8  %union {char t[100];}
9  %token      KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
10 %token<t>   ZAHL
11 %%
12 expression: term
13           | expression PLUS term {printf("+\n");}
14           | expression MINUS term   {printf("-\n");}
15           ;
16
17 term:       factor
18           | term MAL factor       {printf("*\n");}
19           | term DIV factor       {printf("/\n");}
20           ;
21
22 factor:     ZAHL         {printf("%s\n", $1);}
23           | KLA_AUF expression KLA_ZU
24           | PLUS factor
25           | MINUS factor       {printf("CHS\n");}
26           ;
27 %%
28 #include "lex.yy.c"
29
30 int yyerror(char *s) {
31     printf("%s\n", s);
32
33 }
34
35 int main() {
36     int rc;
37     rc = yyparse();
38     if (rc == 0)
39         printf("Syntax OK\n");
40     else
41         printf("Syntax nicht OK\n");
42     return rc;
43 }
```

Ein Term-Rechner

Listing 12.6. Ein Term-Parser (lexyacc/term-rechner.y)

```
1  %{
2  /*******************************************************
3  Interpreter für arithmetische Ausdrücke
4  *******************************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  %}
8  %union {char t[100]; double x;}
9  %token      KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
10 %token<t>   ZAHL
11 %type<x> factor term expression
12 %%
13 input:       expression        {printf("Erg: %lf\n", $1);}
14           ;
15
16 expression: term          {$$ = $1;}
17           | expression PLUS term {$$ = $1 + $3;}
18           | expression MINUS term   {$$ = $1 - $3;}
19           ;
20
21 term:        factor          {$$ = $1;}
22           | term MAL factor       {$$ = $1 * $3;}
23           | term DIV factor       {$$ = $1 / $3;}
24           ;
25
26 factor:      ZAHL          {$$ = atof($1);}
27           | KLA_AUF expression KLA_ZU  {$$ = $2;}
28           | PLUS factor         {$$ = $2;}
29           | MINUS factor        {$$ = -$2;}
30           ;
31 %%
32 #include "lex.yy.c"
33
34 int yyerror(char *s) {
35     printf("%s\n", s);
36
37 }
38
39 int main() {
40     int rc;
41     rc = yyparse();
42     if (rc == 0)
43         printf("Syntax OK\n");
44     else
45         printf("Syntax nicht OK\n");
46     return rc;
47 }
```

Ein Term-AST-Generator

Listing 12.7. Ein Term-Parser (lexyacc/term-ast-gen.y)

```
%{
/*********************************************************
Konstruiert AST für arithmetische Ausdrücke (Operator-Baum)
*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "term-ast.h"
#include "term-ast.c"
struct node * tree;
%}
%union {char t[100]; struct node *p;}
%token      KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
%token<t>   ZAHL
%type<p> factor term expression
%%
input:      expression              {tree = $1;}
          ;

expression: term                {$$ = $1;}
          | expression PLUS term        {$$ = new_node("+", $1, $3);}
          | expression MINUS term    {$$ = new_node("-", $1, $3);}
          ;

term:       factor                {$$ = $1;}
          | term MAL factor             {$$ = new_node("*", $1, $3);}
          | term DIV factor             {$$ = new_node("/", $1, $3);}
          ;

factor:     ZAHL                {$$ = new_node($1, NULL, NULL);}
          | KLA_AUF expression KLA_ZU  {$$ = $2;}
          | PLUS factor             {$$ = $2;}
          | MINUS factor            {$$ = new_node("CHS", $2, NULL);}
          ;
%%
#include "lex.yy.c"

int yyerror(char *s) {
    printf("%s\n", s);

}

int main() {
    int rc;
    rc = yyparse();
    if (rc == 0) {
        printf("Syntax OK\n");
   printf("Baum:\n"); tree_output(tree,0);
   printf("UPN:\n"); tree_code(tree);
   printf("Ergebnis: %lf\n", tree_result(tree));
   }
    else
        printf("Syntax nicht OK\n");
    tree_free(tree);
```

```
54      return rc;
55  }
```

Die Programme benötigen zur Compilierung noch die Programme für den Term-AST, Listing **??** (Seite **??**) und **??** (Seite **??**).

A

ASCII-Tabelle

dez	hex	0	16	32	48	64	80	96	112
		0	10	20	30	40	50	60	70
0	0	\0		␣	0	@	P	‘	p
1	1			!	1	A	Q	a	q
2	2			"	2	B	R	b	r
3	3			#	3	C	S	c	s
4	4			$	4	D	T	d	t
5	5			%	5	E	U	e	u
6	6			&	6	F	V	f	v
7	7	\a		’	7	G	W	g	w
8	8	\b		(	8	H	X	h	x
9	9	\t		)	9	I	Y	i	y
10	A	\n		*	:	J	Z	j	z
11	B			+	;	K	[	k	{
12	C	\f		,	<	L	\	l	|
13	D	\r		-	=	M	]	m	}
14	E			.	>	N	^	n	~
15	F			/	?	O	_	o	del

Die ASCII-Codes 0 bis 31 (dez) stellen Steuerzeichen dar. In der Tabelle sind nur die Steuerzeichen eingetragen, für die es in C Escapesequenzen gibt. ␣ stellt das Leerzeichen dar.

Die Codes von 128-255 (dez) sind nicht standardisiert, und von Betriebssystem zu Betriebssystem unterschiedlich belegt. So können Programme, die deutsche Umlaute verwenden, auf einem anderen Bestriebssystem z.B. falsche Anzeigen verursachen. In Zeiten der 7-bit-Datenverarbeitung musste - da es keine Codes > 127 gab - im ASCII-Code einige Zeichen durch die deutschen Umlaute ersetzt werden. So gibt es eine deutsche Variante des 7-bit-ASCII-Codes.

Umlaut	Unix / Windows	DOS / WIN-Konsole	ASCII (D)
Ä	196/C4	142/8E	91/5B
Ö	214/D6	153/99	92/5C
Ü	220/DC	154/9A	93/5D
ä	228/E4	132/84	123/7B
ö	246/F6	148/94	124/7C
ü	252/FC	129/81	125/7D
ß	223/DF	225/E1	126/7E

B

Abkürzungen

CT Compile-time
RT Compile-time
SL Static Link
DL Dynamic Link
AR Activation Record

Literaturverzeichnis

ASU88. Alfred Aho, Ravi Sehti, and Jeffrey D. Ullman. *Compilerbau - Band 1.* Addison-Wesley, Bonn, 1988. ISBN 3-89319-150-X. 13, 72, 75, 78, 84

Her95. Helmut Herold. *Lex und Yacc: Lexikalische und syntaktische Analyse.* Addison-Wesley, Bonn, 2. edition, 1995. 127

KR90. Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Hanser, München, 2. edition, 1990. ISBN 3-446-15497-3. 105

LS75. M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *Computing Science Technical Report No. 39*, October, 1975. 128

Sed92. Robert Sedgewick. *Algorithmen.* Addison-Wesley, 1992. 67

Wir86. Niklaus Wirth. *Compilerbau.* Teubner, Stuttgart, vierte edition, 1986. 15, 67

Wir96. Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, Bonn, 1996. 67, 75