Compilerbau

Skriptum zur Vorlesung

Studiengang Informatik
Prof. Dr. Winfried Bantel

Wintersemester 2020 / 2021
Stand 8. Oktober 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

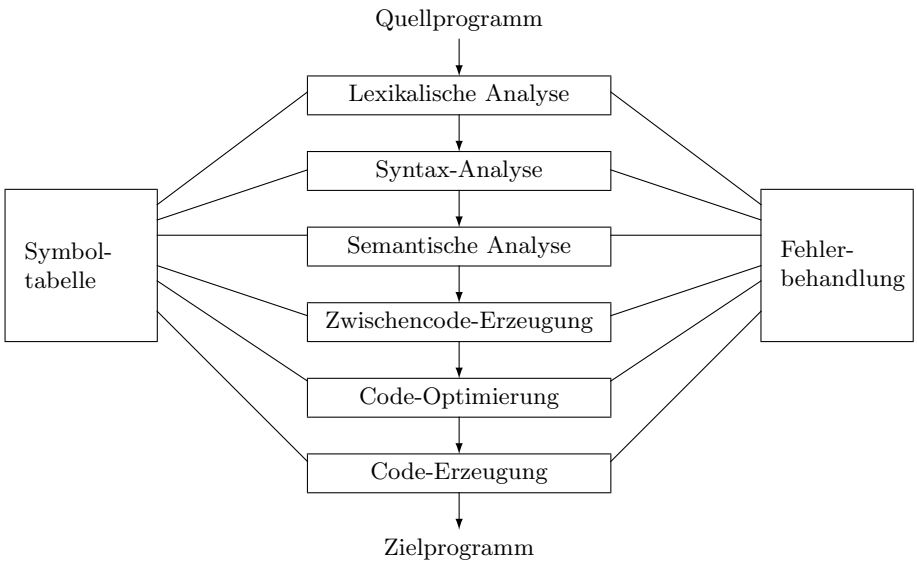
Listings

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
- Rechenprogramme (auch Tabellenkalkulationen)
- Syntax-Highlightning von Editoren
- Verarbeitung von ini-Dateien
- SQL-Interpreter
- Verarbeitung von Eingabedaten
- Web-Browser
- XML-Verarbeitung
- ...

1.3 Begriffsbildung

Definition 1.2 (Parser). *Ein Parser überptüft eine in einer formalen Sprache geschriebenes Eingabe anhand einer Grammatik.*

Definition 1.3 (Compiler). *Ein Compiler übersetzt ein in einer formalen Sprache geschriebene Eingabe in eine andere formale Sprache.*

Definition 1.4 (Interpreter). *Ein Interpreter führt ein in einer formalen Sprache geschriebene Eingabe 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 ")"
```

- Identifier
- Unäres Minus, siehe auch Grammatiken ...
- Scanner-Modifikationen
- ; Bei Verbundanweisung

.

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 fak;
4  BEGIN
5          IF n > 0 THEN
6          BEGIN
7                  DEBUG;
8                  f := f * n;
9                  n := n - 1;
10                 CALL fak;
11                 n := n + 1;
12         END;
13 END;
14 BEGIN
15     ? n;
16     f := 1;
17         DEBUG;
18     CALL fak;
19         DEBUG;
20     ! f;
21 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;
```

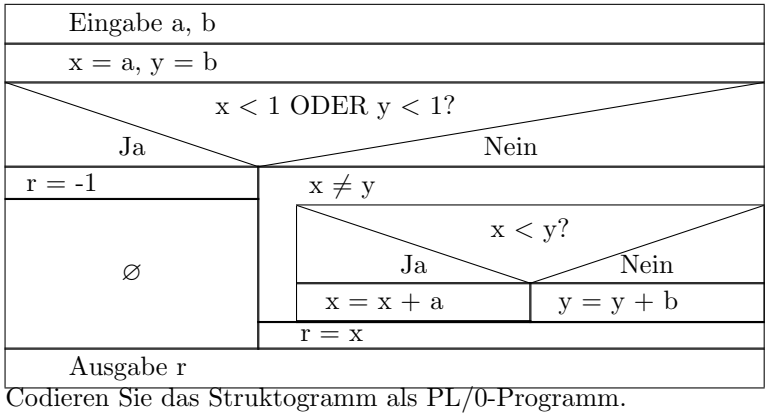
```
8           IF b > a THEN b := b - a;
9       END;
10      g := a;
11  END;
12  BEGIN (* Hauptprogramm *)
13      ? a;
14      ? b;
15      CALL ggt;
16      ! g;
17  END.
```

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 17), 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 2 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:

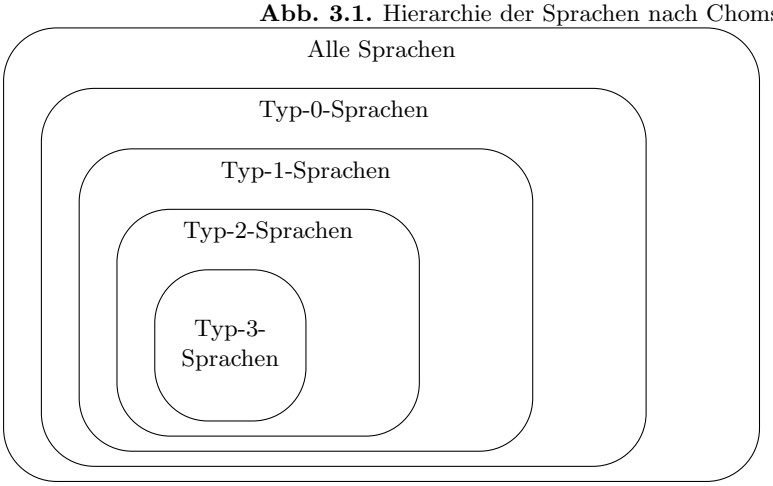
$$T_3 \subset T_2 \subset T_1 \subset T_0 \subset \text{Menge aller Sprachen}$$

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.

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

Abb. 3.1. Hierarchie der Sprachen nach Chomsky



3.1.2 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.*

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

3.1.3 Endliche Automaten und Formale Sprachen

Satz 4 *Zu jedem Keller-Automaten kann eine kontextfreie Grammatik entwickelt werden, die die von diesem Automat akzeptierte Sprache beschreibt.*

Es ist eine Reihe von Konventionen zur Benennung von Symbolen im Zusammenhang mit kfGs gebräuchlich. Wir werden uns an die folgenden Konventionen halten:

1. Kleinbuchstaben vom Anfang des Alphabets (a, b usw.) stehen für terminale Sym- bole. Wir werden zudem annehmen, dass Ziffern und andere Zeichen wie + und Klammern terminale Symbole sind.
2. Großbuchstaben vom Anfang des Alphabets (A, B usw.) repräsentieren Variable.
3. Kleinbuchstaben vom Ende des Alphabets (z. B. w oder z) bezeichnen Zeichenreihen aus terminalen Symbolen. Diese Konvention soll daran erinnern, dass die termina- len Symbole den Eingabesymbolen von Automaten entsprechen.
4. Großbuchstaben vom Ende des Alphabets (wie X oder Y) stehen für terminale Sym- bole oder Variable.
5. Griechische Kleinbuchstaben (z. B. α und β) repräsentieren Zeichenreihen, die aus terminalen Symbolen und/oder Variablen bestehen. Es gibt keine spezielle Notation für Zeichenreihen, die ausschließlich aus Variablen bestehen, da dieses Konzept nicht weiter wichtig ist. Es ist allerdings möglich, dass eine mit einem griechischen Kleinbuchstaben wie α benannte Zeichenreihe ausschließlich Variable enthält.

Siehe [HMU02] Seite 186

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.

$A = 'A','B','C',\ldots,'X','Y','Z'$	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“, „NP“, „VP“. Formell ist dies Grammatik in Γ_2 dargestellt. **Grammatik Γ_2: Einfach-Deutsch**

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

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.

Abb. 3.2. Syntaxbaum “Die Katze schnurrt“

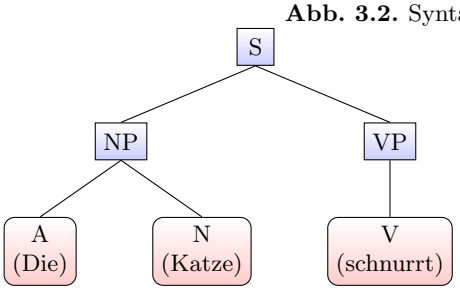
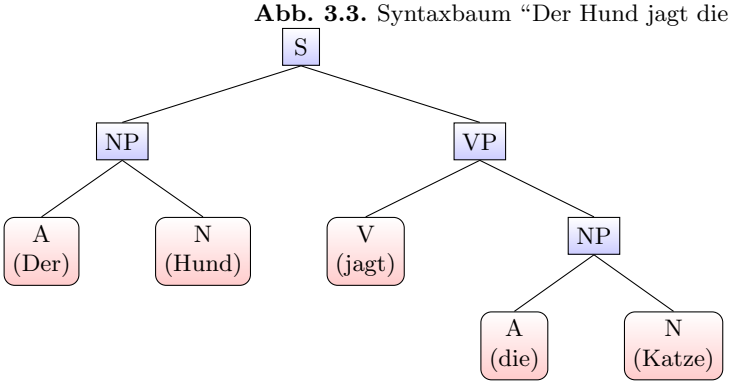


Abb. 3.3. Syntaxbaum “Der Hund jagt die Katze“



3.1.4 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>
```

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

```
<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:

- 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:

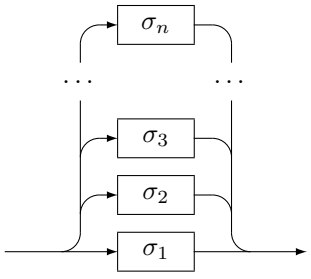
Syntaxdiagramm 1: Nicht-Terminal A

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

Syntaxdiagramm 2: Terminal-Symbol σ

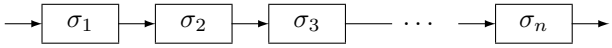
Alternative: Produktionsregeln der Form $A \to \sigma_1|\sigma_2|\sigma_3|\ldots|\sigma_n$

Syntaxdiagramm 3: Alternative



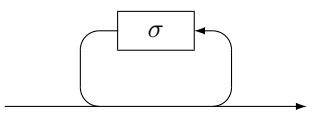
Sequenz: Produktionsregeln der Form $A \to \sigma_1\sigma_2\sigma_3\ldots\sigma_n$

Syntaxdiagramm 4: Sequenz



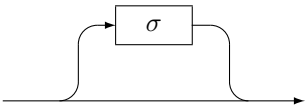
Wiederholung: Produktionsregeln der Form $A \to \{\sigma\}$

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.

3.1.5 Notation in diesem Script

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

- Das Produktionszeichen ist $\to$.
- 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 $\to$ **NP VP**
NP $\to$ **['a'] 'n'**
VP $\to$ **'v' [NP]**

3.1.6 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	→	**VK [NK]**
VK	→	**INT**
NK	→	**’.’ INT**
INT	→	**DIGIT DIGIT**
DIGIT	→	**’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	→	**INT** | **’(’ SATZ ’)’**
INT	→	**DIGIT {DIGIT}**
DIGIT	→	**’0’** | **’1’** | **’2’** | **’3’** | **’4’** | **’5’** | **’6’** | **’7’** | **’8’** | **’9’**

3.1.7 Operatorgrammatiken

Eine Operatorgrammatik liegt vor, wenn

1. keine ϵ-Produktion existiert,
2. keine rechte Seite einer Produktion direkt benachbarte Nichtterminale hat,

- Arithmetischer Term
- Ausdrücke in Programmiersprachen und Tabellenkalkulationen
- “where“-Bedingung in SQL
- PL/0

- Einfach zu parsen
- Verhindert Mehrdeutigkeiten

Beispiel: C-if ohne Klammer um Bedingung:

```
if a > b + c = d
```

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.

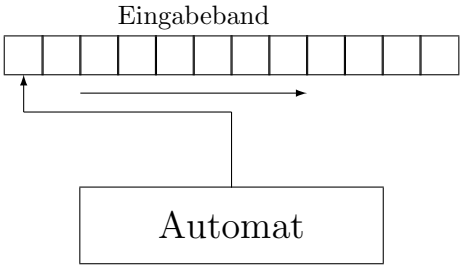
E ist das Eingabealphabet des Automaten (die Menge der Eingabesymbole).

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

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.

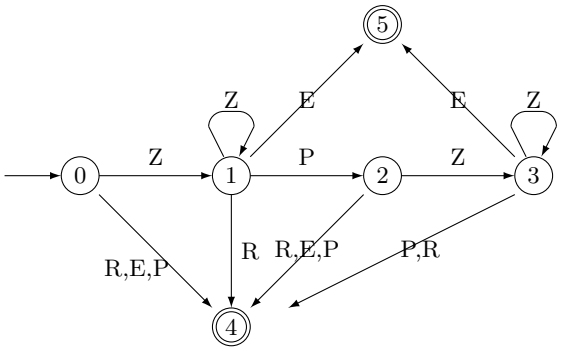
Von Endezuständen führen keine Pfeile weg. Der Startzustand kann durch einen Initialpfeil markiert werden.

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 30). 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

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

3.2.5 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 5 *Zu jeder kontextfreien Sprache (Chomsky-Typ-2) kann ein äquivalenter Kellerautomat konstruiert werden, der die Syntax dieser Sprache überprüft.*

Satz 6 *Zu jedem Kellerautomaten kann eine kontextfreien Grammatik (Chomsky-Typ-2) entwickelt werden, die die von diesem Automat akzeptierte Sprache beschreibt.*

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.

Ein Kellerautomat A ist damit ein Siebentupel A = (E, K, Z, δ, k_0, z_0, F).

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

K ist das Kelleralphabet des Automaten (die Menge der Kellersymbole).

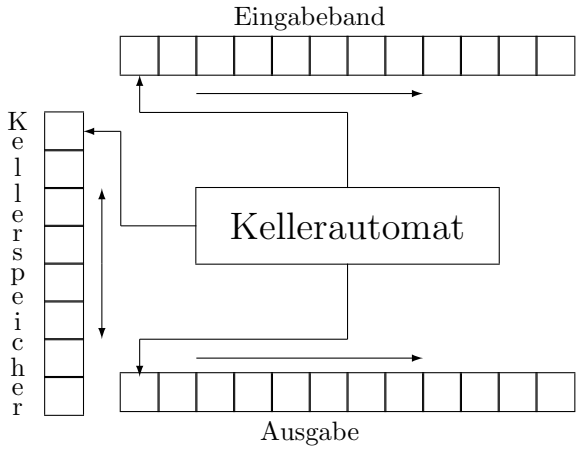
Z ist die Menge der Zustände in denen sich der Automat A befinden kann.

δ ist die Übergangsfunktion, die jeder Kombination aus Zustand und Eingabesymbol und Top-of-Stack eine Aktion und einen Folgezustand zuordnet. $Z \otimes E \otimes K \rightarrow Z \otimes K$.

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

k_0 ist das Kelleranfangssymbol, welches sich am Anfang im Keller befindet.

F ist die Menge der Endezustände ($F \subseteq Z$).



$L = \{a^n b^n | n \in \mathbb{N}\}$

$L = \{a^n b^n | n \in \mathbb{N}\}$

z e k	**Aktion z$_n$**
0 a \$	push a 0
0 a a	push a 0
0 b \$	err
0 b a	pop 1
1 a \$	err
1 a a	err
1 b \$	err
1 b a	pop 1

Keller	**z**	**Eingabe**	**Aktion**	$\mathbf{z}_n$
\$	0	a b \$	push a	0
\$ a	0	b \$	pop	1
\$	1	\$	accept	

Keller	**z**	**Eingabe**	**Aktion**	$\mathbf{z}_n$
\$	0	a a b b \$	push a	0
\$ a	0	a b b \$	push a	0
\$ a a	0	b b \$	pop	1
\$ a	0	b \$	pop	1
\$	1	\$	accept	

Es gibt Sprachen die mit einem Kellerautomat nicht geparst werden können.!

Beispiel $L = \{a^n b^n c^n | n \in \mathbb{N}\}$

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

UPN – Die Umgekehrte Polnische Notation

Die umgekehrte polnische Notation (UPN) oder reverse polnische Notation (englisch reverse Polish notation, kurz RPN), auch Postfixnotation genannt, ist eine von der polnischen Notation abgeleitete Schreibweise bzw. Eingabelogik für die Anwendung von Operationen. Bei der umgekehrten polnischen Notation werden zunächst die Operanden niedergeschrieben bzw. eingegeben und danach der darauf anzuwendende Operator.
(Wikipedia)

Statt „1 + 2“ schreibe „1 2 +“

Mathematische Formeln können anstatt in der üblichen Schreibweise auch in umgekehrter polnischer Notation geschrieben werden. Der Vorteil dieser Schreibweise ist dass man weder Rechenregeln (Punkt-vor-Strich) noch Klammern zur Berechnung benötigt. Stattdessen wird ein Kellerspeicher benötigt, auf dem Teilergebnisse zwischengespeichert werden.

Konventionell	**UPN**
$1 + 2$	`1 2 +`
$1 + 2 * 3$	`1 2 3 * +`
$(1 + 2) * 3$	`1 2 + 3 *`
$\frac{2-1}{3+4}$	`2 1 - 3 4 + /`
$\frac{2*3-4/-2}{3+4}$	`2 3 * 4 2 CHS / - 3 4 + /`
$sin(1/4 * \pi)$	`1 4 / π * sin`

4.1 Grammatik eines UPN-Ausdrucks

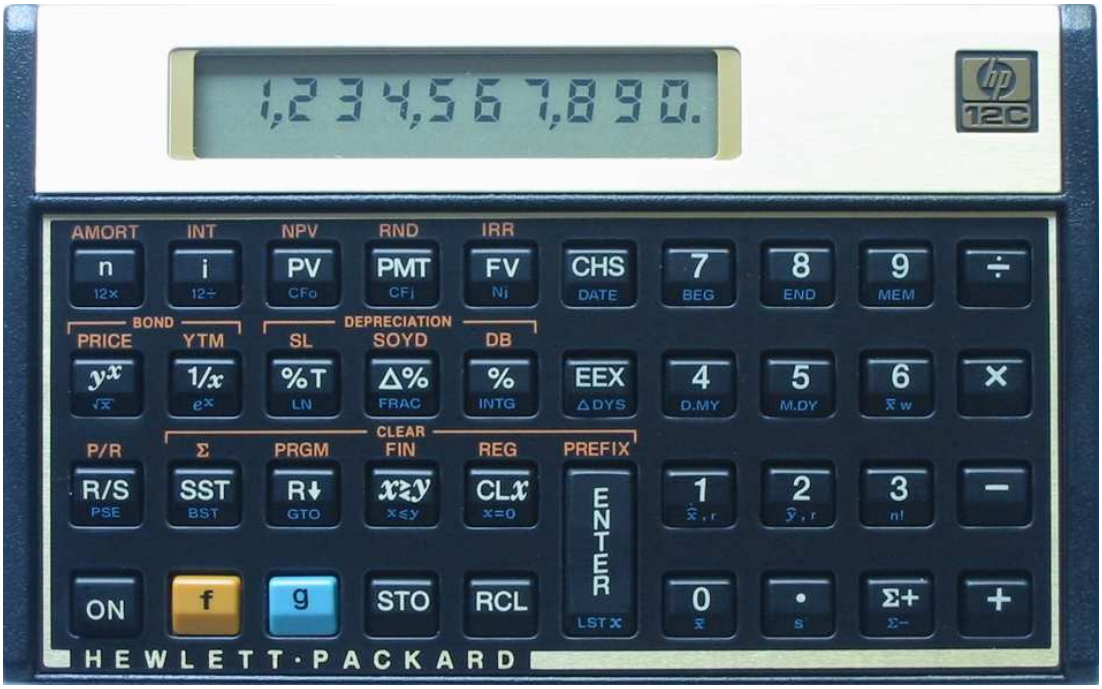
Grammatik Γ_{13}: UPN-Grammatik
upn → **'zahl'**
upn → **upn upn bin-op**
upn → **upn un-op**
bin-op → **'+' | '-' | '*' | '/'**
un-op → **'CHS'**

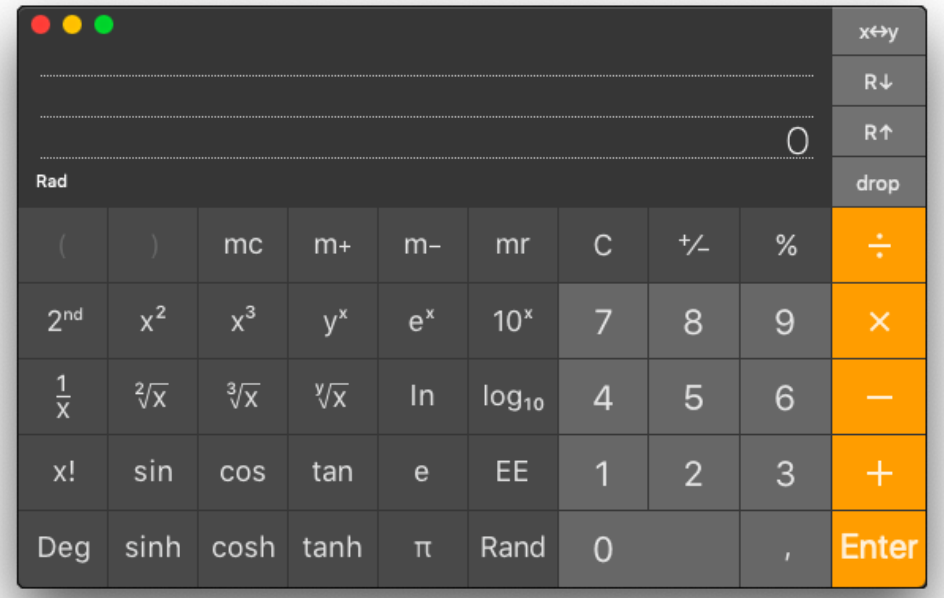
4.2 Berechnung eines UPN-Ausdrucks

UPN-Ausdrücke werden folgendermaßen ausgewertet:

- Zahlen werden auf den Stack gelegt
- Bei binären Operatoren (+, −, ∗ und /) werden zwei Zahlen vom Stack geholt, mit dem Operator verknüpft und das Ergebnis wieder auf dem Stack gespeichert

- Bei unären Operatoren (CHS) wirr eine Zahlen vom Stack geholt, mit dem Operator verknüpft und das Ergebnis wieder auf dem Stack gespeichert
- Bei Funktion (sin(), ln() etc., eine unabhängige Variable) wird eine Zahl als Funktionsargument vom Stack geholt, die Funktion von dieser Zahl berechnet und das Ergebnis wieder auf dem Stack gespeichert.
- Am Ende aller Operationen muss die Stackhöhe eins sein und das eine Element ist das Endergebnis des Ausdrucks



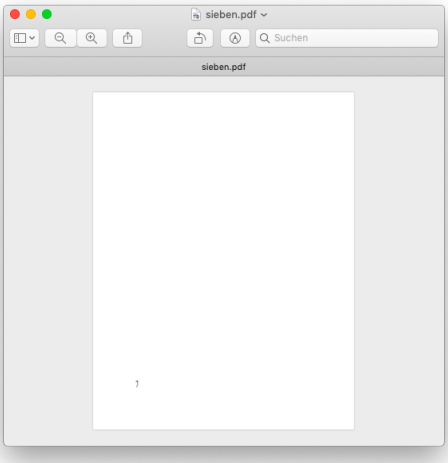


- Kein Kellerautomat notwendig
- Keine Klammerstrukturen → einfachere Auswertung
- Keine Operatorpriorität → einfachere Auswertung
- Keine Operatorassoziativität → einfachere Auswertung

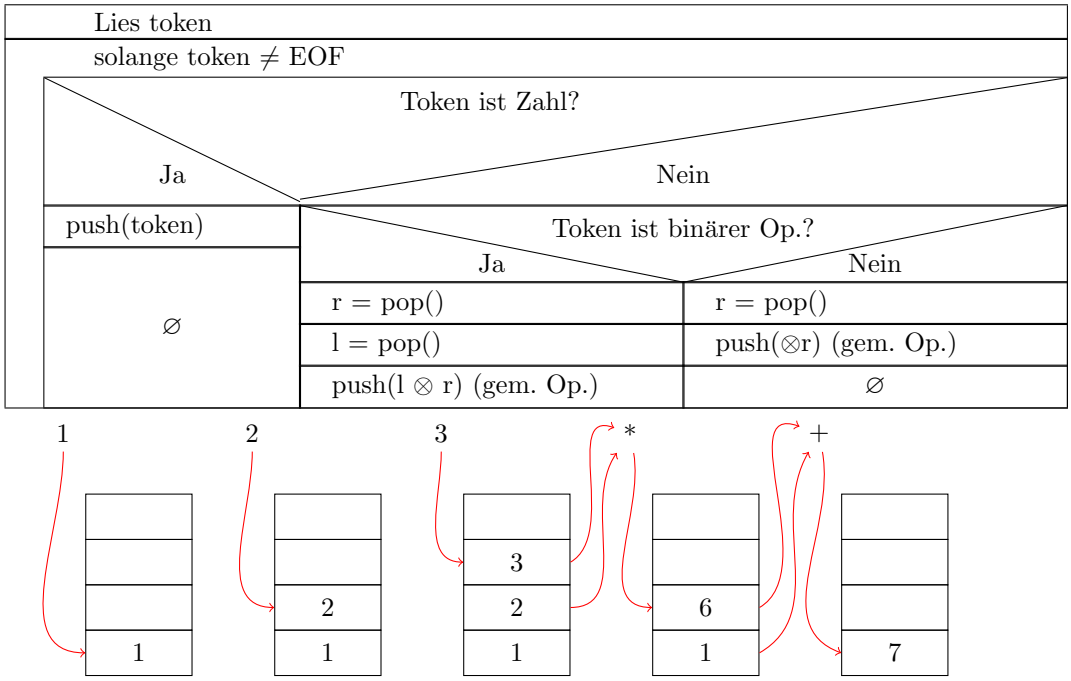
- Postscript
- Python-Bytecode
- Java-Bytecode (?)
- dc
- Hardware-Prozessoren

Listing 4.1. Postscript-Programm zur Becechnung 1+2*3 (upn/sieben.ps)

```
1 %!
2 /Courier findfont 12 scalefont [1 0 0 2 0 0] makefont setfont
3 100 100 moveto
4 1 2 3 mul add 10 string cvs show
5
6 showpage
```




```
==> cat sieben.dc
1 2 3 * +
p
==> dc sieben.dc
7
==>
```

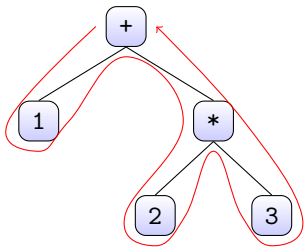


Listing 4.2. UPN-Verarbeitung (upn/upn-rechner.c)

```
1 #include <stdio.h>
2 #include <string.h>
3 #include "stack.h"
4 #include <stdlib.h>
5
6 int main () {
7 // char text[] = "  1   2 3 * +";
```

```
8     char text[256];
9     fgets(text, 256, stdin);
10    int z1, z2;
11    char * token = strtok(text, " \n");
12    while (token != NULL) {
13        if (token[0] >= '0' && token[0] <= '9') // Zahl!
14            push(atoi(token));
15        else if (token[0] == '*')
16            pop(&z2), pop(&z1), push(z1 * z2);
17        else if (token[0] == '+')
18            pop(&z2), pop(&z1), push(z1 + z2);
19        token = strtok(NULL, " \n");
20    }
21    pop(&z1), printf("Ergebnis: %d\n", z1);
22     return 0;
23 }
```

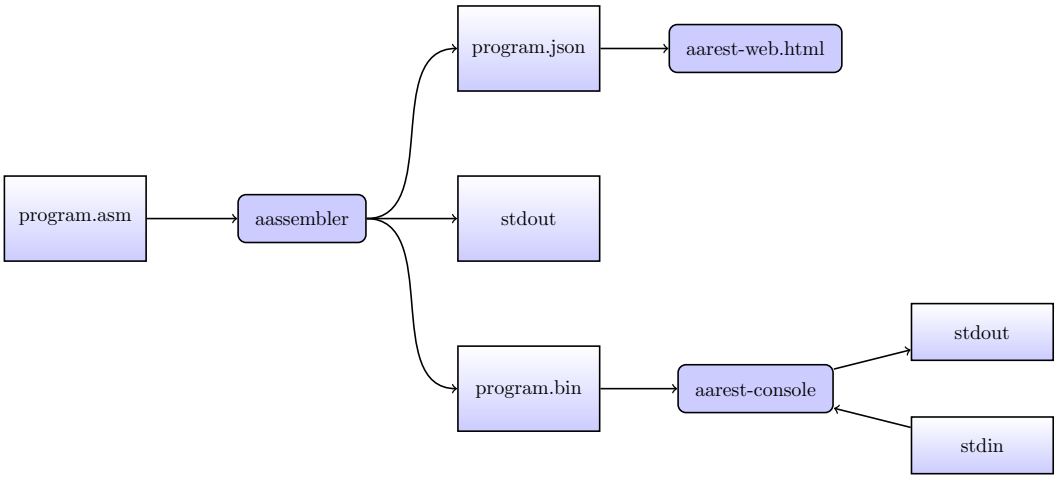
Algorithmus (Struktogramm)



1. Erzeuge den Operatorbaum
2. Gib den Operatorbaum in Postorder-Reihenfolge aus

5

Die virtuelle Maschine



- (Fast) Analog zu VM
- Aber symbolische Sprungziele:
  - Label ist beliebige int
  - Exakte Zeilennummer muss zur Codierung nicht bekannt sein
  - Unterscheidliche Befehle für io, call, ...
  - Kommentare mit `#`

Befehl	**Bedeutung**	**Wertigkeit**
add	Addition	binär
sub	Subtraktion	binär
mult	Multiplikation	binär
div	Division (ganzzahlig)	binär
mod	Modulus (ganzzahlig)	binär
chs	Vorzeichenwechsel	unär
cmpeq	Test auf Gleichheit	binär
cmpne	Test auf Ungleichheit	binär
cmple	Test auf $\leq$	binär
cmplt	Test auf $<$	binär
cmpge	Test auf $\geq$	binär
cmpgt	Test auf $>$	binär

Befehl	**Bedeutung**	**Argument**
jump	Unbedingter Sprung	Label
jumpz	Bedingter Sprung falls 0	Label
jumpnz	Bedingter Sprung falls nicht 0	Label
call	Funktionsaufruf	Label
return	Funktionsrücksprung	-

Befehl	**Bedeutung**	**Argument**
loadc	Konstante auf Stack	Konstante
loadr	Register auf Stack	Adresse
storer	Stack in Register	Adresse
loads	Register auf Stack (Adresse vom Stack)	-
stores	Stack in Register (Adresse vom Stack)	-
inc	TOS incrementieren	Konstante
dec	TOS decrementieren	Konstante
dup	TOS duplizieren	-
drop	TOS löschen	-
swap	Oberste zwei Stackelemente tauschen	-

Befehl	**Bedeutung**	**Argument**
read	Eingabe auf Stack	-
write	Ausgabe vom Stack	-

Befehl	**Bedeutung**	**Argument**
nop	„No Operation“	-

```
        loadc   1
        loadc   2
        loadc   3
        mult
        add
        write

# (5 * -(1 + 2 * 3)) + ( 3 + (1 + 2 * 3))
        loadc   1
        loadc   2
        loadc   3
        mult
        add                     # (1 + 2 * 3) fertig
        dup
        chs
        loadc   5
        mult            # Erster Summand fertig
        swap
        loadc   3
        add                     # Zweiter Summand fertig
        add
        write

        read
        storer  0
        loadr   0
        loadc   0
        cmplt
        jumpz   2
        loadr   0
```

```
        chs
        storer  0
2       loadr   0
        write

# Gib die Zahlen 0...4 aus
        loadc   0       # Initialisiere mit 0
        storer  0
1       loadr   0       # Schleifenbedingung
        loadc   5
        cmplt
        jumpz   2       # Feierabend?
        loadr   0
        write
        loadr   0       # Incementieren
        loadc   1
        add
        storer  0
        jump    1       # Und wieder nach oben...
2                               # Programmende
```

- Parameter werden vor Aufruf auf Stack gelegt
- Ergebnis wird am Funktionsende auf Stack gelegt

```
# Betragsberechnung ueber Funktion
                read
                call betrag
                write
                read
                call betrag
                write
                return

betrag  dup
                loadc   0
                cmpge
                jumpnz  betrag2
                chs
betrag2 return
```

```
#!/bin/bash
hexdump -e '2/4 "%8d " "\n"' $1 |
        awk '{printf("%4d%s\n", NR-1,$0)}'
```

```
==>  ./bindump.sh test.bin
   0       2        0
   1      12        2
   2       1       12
==>
```

Jeder Bytesatz besteht aus zwei Werten:

1. Befehl

2. Argument

Befehl	**Argument**	**Bedeutung**
op	Rechenoperation	Operator
loadc	Konstante auf Stack	Konstante
loadr	RAM → Stack	Adresse
storer	Stack → RAM	Adresse
jump	Unbedingter Sprung	Programmcounter
jumpz	Bedingter Sprung wenn 0	Programcounter
jumpnz	Bedingter Sprung wenn nicht 0	Programcounter
io	Ein- / Ausgabe	Richtung
nop	Keine Aktion	-

Operator	**Bedeutung**	**Wertigkeit**
add	Addition	binär
sub	Subtraktion	binär
mult	Multiplikation	binär
div	Division (ganzzahlig)	binär
mod	Modulus (ganzzahlig)	binär
chs	Vorzeichenwechsel	unär
cmpeq	Test auf Gleichheit	binär
cmpne	Test auf Ungleichheit	binär
cmple	Test auf $\leq$	binär
cmplt	Test auf $<$	binär
cmpge	Test auf $\geq$	binär
cmpgt	Test auf $>$	binär

Argument	**Bedeutung**
out	Ausgabe
in	Eingabe

6

Grundlagen

6.1 Scanner und Parser

Grammatik Γ_{14} ist eine Modifikation von Grammatik Γ_{12} (Seite 34). Dabei wurde das terminale Symbol `zahl` durch ein weiteres Non-Terminal ersetzt:

Grammatik Γ_{14}: 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.

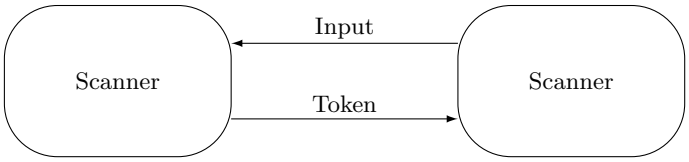
Die lexikalische Analyse übernimmt der sog. Scanner, die syntaktische Analyse der Parser.

Definition 6.1 (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 6.2 (Parser). *Parser überprüfen eine Folge von Zeichen ihres Eingabealphabets auf korrekte Reihenfolge. Diesen Vorgang nennt man auch "Syntaktische Analyse" oder "Grammatikalische Analyse".*

Definition 6.3. *Die Elemente einer Sprache, die ein Scanner findet, nennt man Token.*

Abb. 6.1. Zusammenspiel zwischen Scanner und Parser



6.1.1 Zusammenspiel von 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 6.1 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.

6.2 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 32), 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 6.1 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 6.1. Ein zeichenbasierter Scanner (grundl-scanner-fixkommazahl.c)

```
1  /*********************************************************
2  Scanner fuer 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.

6.3 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 6.2 demonstriert, wie einfach ein wortbasierter Scanner mittels der strtok-Funktion zu implementieren ist.

Listing 6.2. 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 }
```

6.3.1 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:

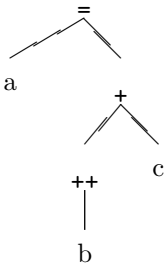
Definition 6.4. *Ein Scanner wählt immer das Token, das den längsten Eingabetext abbildet. Dies nennt man das "Maximum-Match-Prinzip".*

Als Beispiel für Maximum-Match dient der Ausdruck in Zeile 4 Listing 6.3.

Listing 6.3. 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 6.5. *Arbeitet ein Scanner vorausschauend, so nennt man diese einen Look-Ahead.*

6.3.2 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.

6.3.3 Fehlerbehandlung bei Scannern

6.4 Bootstrapping - Das Henne-Ei-Problem

6.5 Reguläre Ausdrücke

6.6 Reguläre Ausdrücke in C

Listing 6.4. Demo zur Benutzung von Regulären Ausdrücken in C (regex.c)

```
1  /************************************************************
2  Demonstrier Regulaere Ausdruecke 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 }
```

6.7 Übungen

Übung 6.6. 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 6.7. 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 6.8. 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.

7

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.

7.1 Grundbegriffe

Ein endlicher Automat ist ein System, das interne Zustände abhängig von einer Eingabe annimmt. Abbildung 7.1 zeigt das Modell eines Automaten.

Definition 7.1 (Endlicher Automat). *Ein endlicher Automat A ist ein Fünftupel $A = (Z, E, \delta, z_0, F)$.*

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

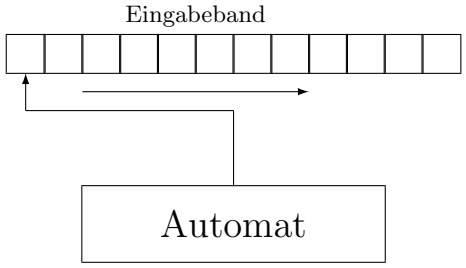
E ist das Eingabealphabet des Automaten (die Menge der Eingabesymbole).

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

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

F ist die Menge der Endezustände ($F \subseteq Z$).

Abb. 7.1. Modell eines endlichen Automaten



7.2 Darstellungsformen

Abbildung 3.4 auf Seite 32 zeigte bereits einen endlichen Automaten sowohl in grafischer als auch in tabellarischer Darstellungsform.

7.2.1 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

7.2.2 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.

7.3 Algorithmus eines endlichen Automaten

Endliche Automaten werden auf zwei verschiedene Arten codiert, die in den Abbildungen 7.2 und 7.3 dargestellt sind. Je nach Aufgabe - Erkennen, Suchen, Scannen, ... - ist mal die eine und mal die andere Variante zu bevorzugen.

7.4 Ein einführendes Beispiel

Der Automat zur Erkennung von Fixkommazahlen (Abbildung 3.4 Seite 32) soll gemäß Abbildung 7.2 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. 7.2. Algorithmus eines endlichen Automaten - Stop bei Endezustand

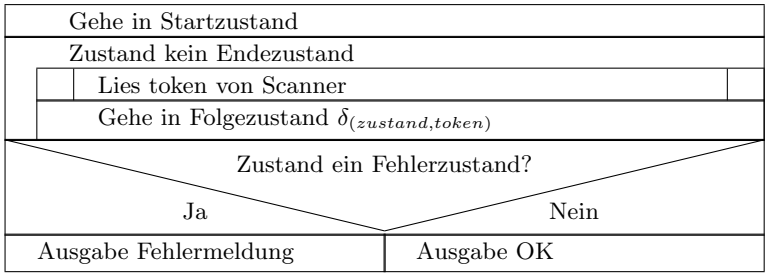
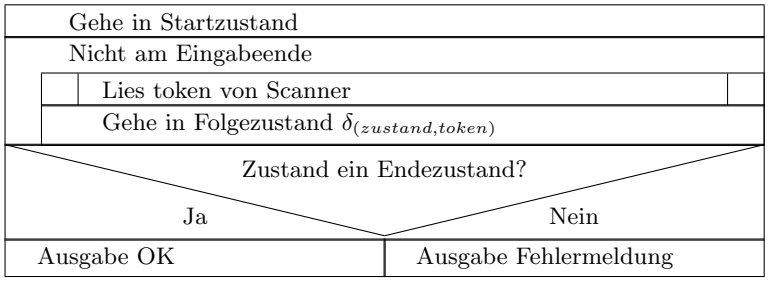


Abb. 7.3. Algorithmus eines endlichen Automaten - Stop bei Eingabeende



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 7.1. 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  }
```

7.4.1 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.

7.4.2 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.

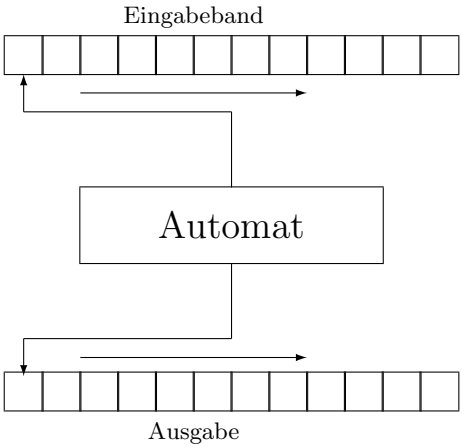
7.5 Endliche Automaten mit Ausgabe

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 bspw. Text gescannt werden, so muss der gescannte Text zusätzlich ausgegeben werden. Damit dies funktioniert, muss der Automat um eine Ausgabe erweitert werden. Aber auch die einfache Entscheidung, ob ein erkennender Automat nach Algorithmus 7.2 (Stop bei Endezustand) „richtig“ oder „falsch“ erkennt lässt sich gut über einen Automat mit Ausgabe realisieren.

Automaten mit Ausgabe werden typischerweise ebenfalls tabellengesteuert programmiert. Je nachdem wie kompex die Ausgabefunktion ist (lediglich abhängig vom Zustand oder aber

abhängig von Zustand und Eingabezeichen) werden die Automaten nach Ihren Entwicklern Moore- [1] und Mealey- [2] Automaten genannt. Als Eselsbrücke dient der Satz „Moore kan nur...“, da er die einfachere Ausgabefunktion hat (siehe Definitionen 7.2 und 7.3).

Abb. 7.4. Automat mit Ausgabe



7.5.1 Moore-Automaten

Ein Moore-Automat A ist ein Siebentupel A = (Z, E, δ, z_0, F, *Omega*, λ). Außer dem Ausgabealphabet Ω und der Ausgabefunktion λ ist der Moore-Automat identisch mit einem erkennenden Automaten.

Definition 7.2 (Moore-Automat). *Ein Moore-Automat ist ein Siebentupel (Z, E, δ, z_0, F, A, λ).*

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.

Ω ist das Ausgabealphabet des Automaten (die Menge der Ausgabesymbole).

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

7.5.2 Mealey-Automaten

Ein Mealey-Automat A ist ein Siebentupel A = (Z, E, δ, z_0, F, Ω, λ). Außer dem Ausgabealphabet Ω und der Ausgabefunktion λ ist der Moore-Automat identisch mit einem erkennenden Automaten.

[1] Edward F. Moore (1925 - 2003), Professor für Mathematik und Informatik in Wisconsin.
[2] George H. Mealy (1927 - 2010), US-amerikanischer Mathematiker, Harvard -Professor

Definition 7.3 (Mealey-Automat). *Ein Mealey-Automat ist ein Siebentupel (Z, E, δ, z_0, F, A, λ).*

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. $Z \otimes E \to Z$.

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.

Ω ist das Ausgabealphabet des Automaten (die Menge der Ausgabesymbole).

λ ist die Ausgabefunktion die jedem Zustand eine Ausgabe zuordnet. $Z \otimes E \to \Omega$.

7.5.3 Effiziente Codierung von Moore- und Mealey

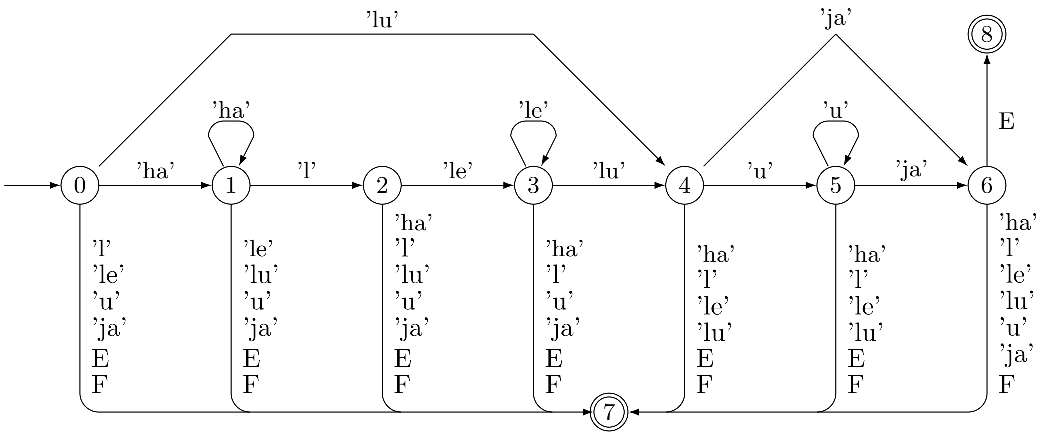
- Ziel: Reusable-Code
- δ-Tabelle:
  - Unbedingt zweidimensionales Array
  - Vermeiden Sie geschachtelte switch-case-Anweisungen
- λ-Tabelle:
  - Wenn möglich zweidimendsionales Array(s) oder Array aus Strukturen
  - Objektorientiert: λ-Funktion per Vererbung erben und modifizieren
  - Nicht-OO: Array aus Funktionszeigern

7.6 Anwendungen von endlichen Automaten

7.6.1 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 "hahahallelelelelеluuuuuuuuuuuja". Falsches Frohlocken wäre etwa "hallllleluja" oder "haluja".

Listing 7.2 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 7.2. 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 7.3. 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

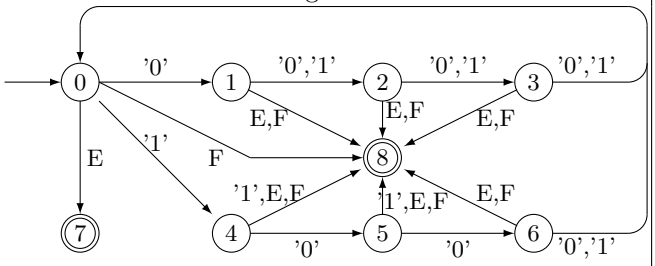
Ein Programm soll erkennen, ob ein eingegebener Text bestehend aus 0 und 1 eine korrekte BCD-Folge darstellt:

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 7.4. 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:

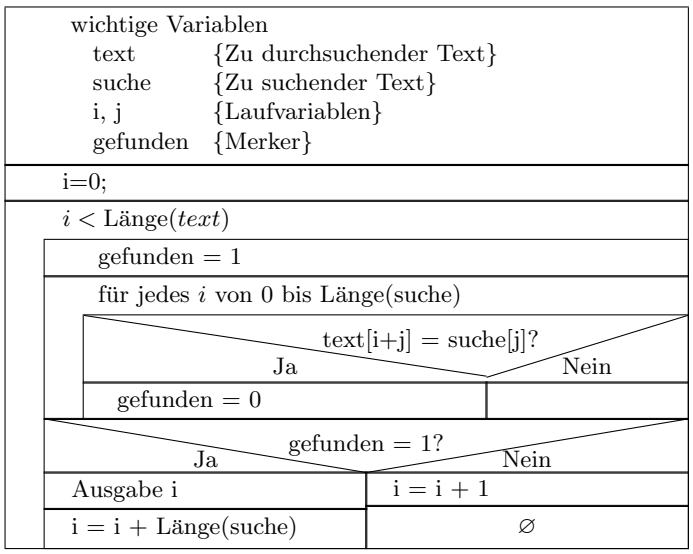
7.6.2 Suchen

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 7.5 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.

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.

Abb. 7.5. Einfache Textsuche



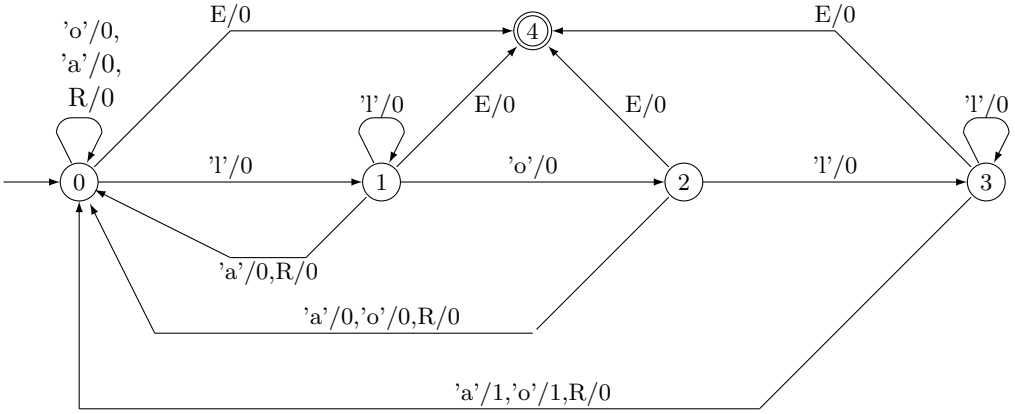
Listing 7.5. Textsuche - Einfachstversion (automaten-textsuche.c)

```
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. Fundstellen in einem Text wie Äls lola die Laolawelle sah sang sie olala lollorossosalat“ sind schon für Menschen schwer zu entdecken. 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 7.6. 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:

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

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

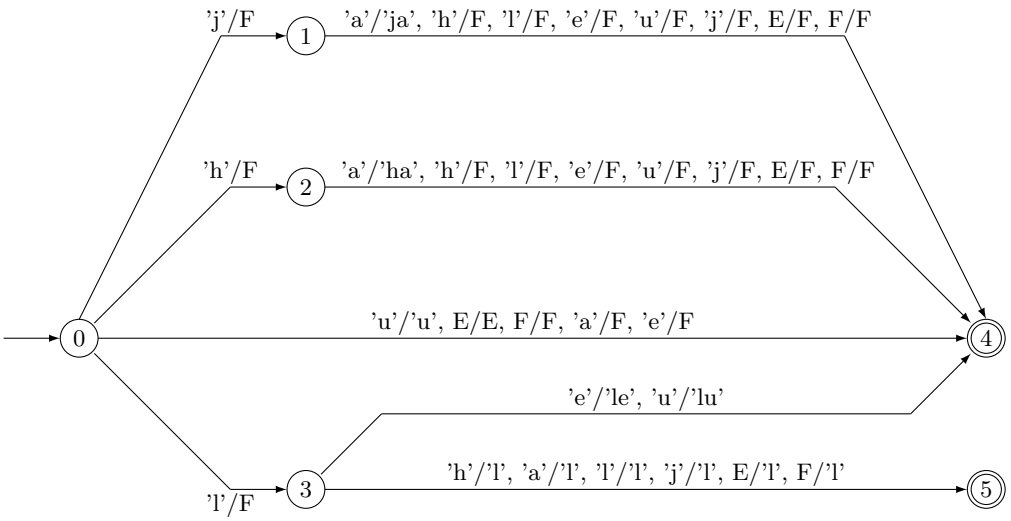
Bildschirmausgabe:

7.6.3 Scannen

Ein Frohlocken-Scanner

Der primitive Scanner des Aloisius-Automaten (Listing 7.2 Seite 60) 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. 7.6. 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 7.7. 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"      : 4
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

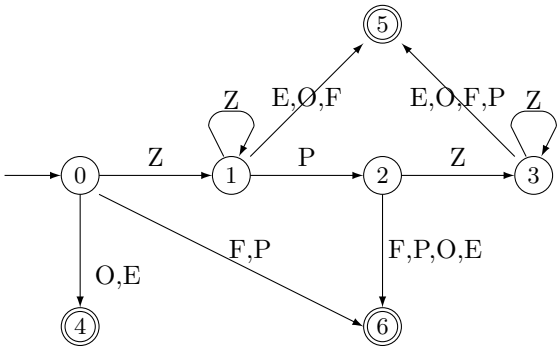
O Operator (“+“, “-“, “*“, “/“, “(“ und “)“)

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 7.8. 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 4,5,5,5,5,5,5,5,1,1,1,1,5,1,2,1,3,3,3,3,3,3,3,3,3,3,5,5,5,5,5,5,
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 7.9. 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 7.10. 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 8 vorgestellt werden.

7.7 Programmgesteuerte Automaten

7.8 Übungen

Übung 7.4. 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 7.5. 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 7.6. 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 7.7. 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 7.8. Aloisius 3
Ändern Sie den erkennenden Aloisius-Automaten in einen Moore-Automaten mit nur einem Endezustand ab.

Übung 7.9. 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 7.10. Aloisius 1
Erstellen Sie die Typ-3-Grammatik zum Frohlocken in EBNF

Übung 7.11. Aloisius 2
Erstellen Sie die Typ-3-Grammatik zum Frohlocken als Syntax-Graphen

Übung 7.12. 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 7.13. BCD 1
Erstellen Sie die Grammatik für BCD-Zahlen in EBNF.

Übung 7.14. BCD 2
Erstellen Sie die Grammatik für BCD-Zahlen als Syntax-Graph.

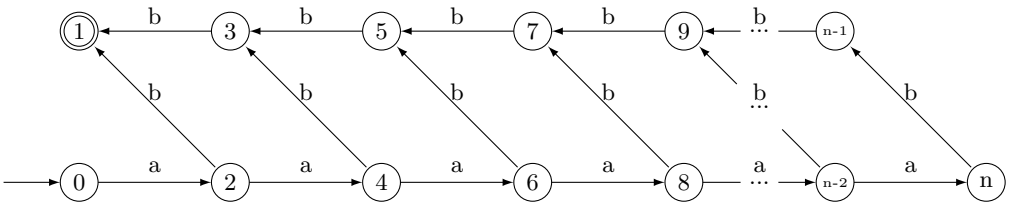
8

Keller-Automaten

8.1 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. 8.1. 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 8.2 zeigt das Modell eines Kellerautomaten.

Ein Kellerautomat kennt zum Speichern die drei Aktionen

push Legt ein Element auf oben auf den Stack

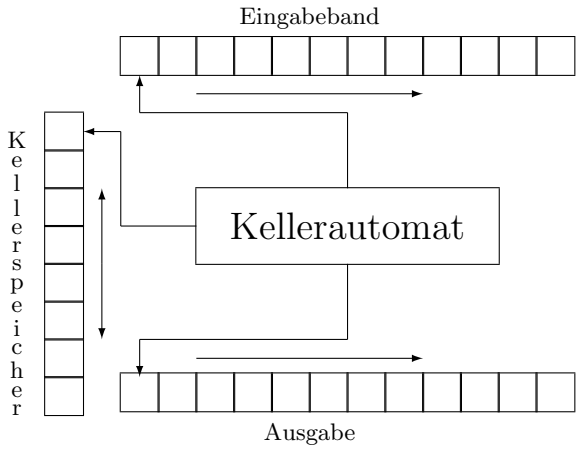
pop Holt ein Element löschend oben vom Stack

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()

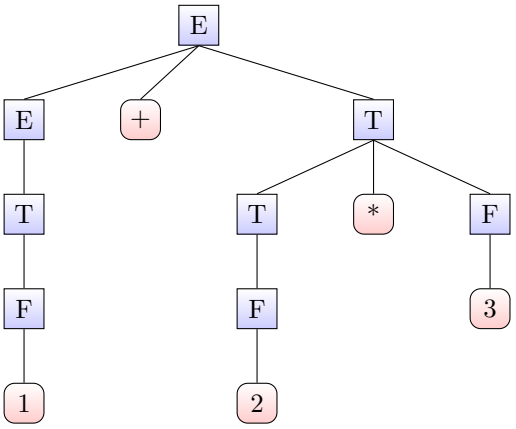
Abb. 8.2. Modell eines Kellerautomaten



8.1.1 Grammatiken

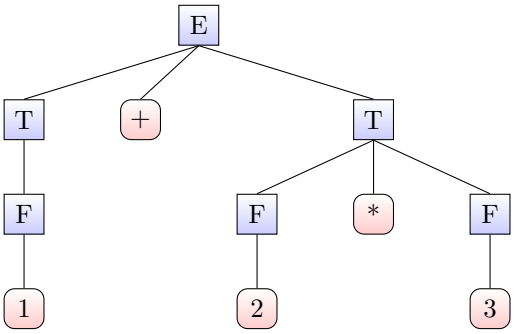
Grammatik Γ_{15}: Einfache Term-Grammatik (1)
E → T | E '+' T
T → F | T '*' F
F → '(' E ')' | '-' F | 'zahl'

- Nicht LL(1) !!! (Warum?)



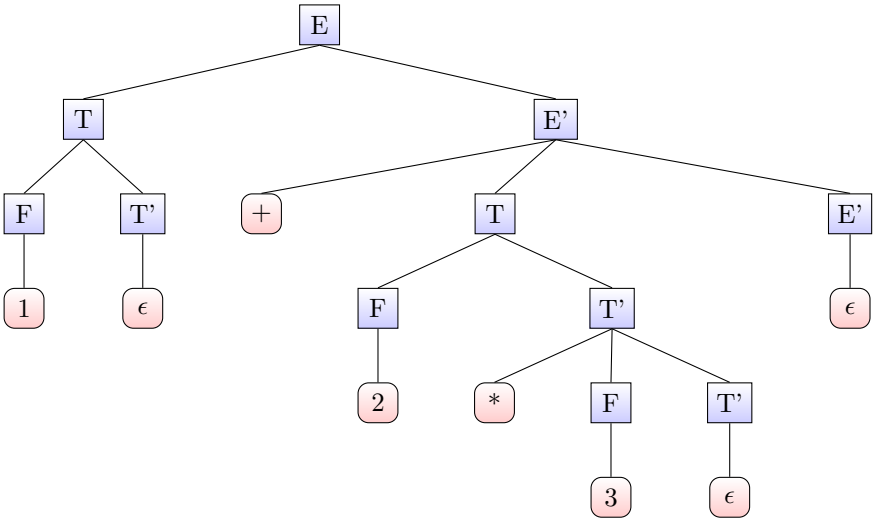
Grammatik Γ_{16}: Einfache Term-Grammatik (2)
E → T { '+' T}
T → F { '*' F}
F → '(' E ')' | '-' F | 'zahl'

- LL(1) (schwierig zu berechnen, warum?)
- Strich-Regel
- Achtung Assoziativität



Grammatik Γ_{17}: Einfache Term-Grammatik (3)
E → **T E'**
E' → **'+' T E'** | ϵ
T → **F T'**
T' → **'*' F T'** | ϵ
F → **'(' E ')'** | **'-' F** | **'zahl'**

- Keine Linksrekursionen



8.2 Native Konstruktion

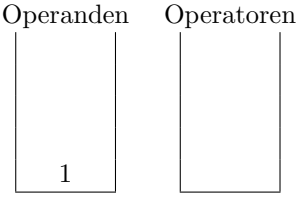
Es soll der folgende Ausdruck ausgewertet werden:

```
1+2*3
```

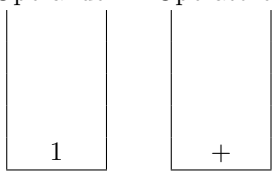
Die Token-Folge des Scanners lautet damit `ZAHL`, `PLUS`, `ZAHL`, `MAL`, `ZAHL`, `END`.

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

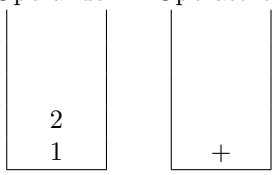
1. Als erstes wird die Zahl 1 eingelesen und auf dem Operandenkeller gespeichert:



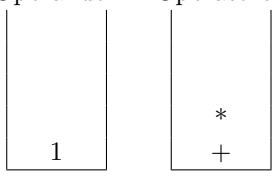
2. Jetzt wird der Operator '+' eingelesen und auf dem Operatorenkeller gespeichert:



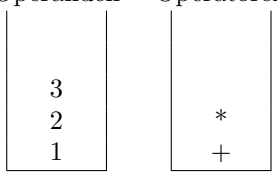
3. Jetzt wird die Zahl 2 eingelesen und auf dem Operandenkeller gespeichert:



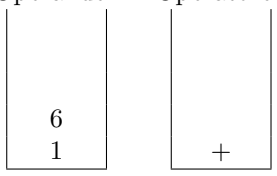
4. Jetzt wird der Operator '*' eingelesen und auf dem Operatorenkeller gespeichert:

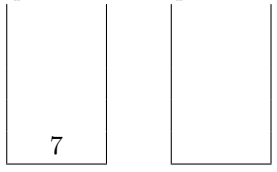


5. Jetzt wird die Zahl 3 eingelesen und auf dem Operandenkeller gespeichert:



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

X Exit, fertig
K Operator abkellern
G Kelleroperation durchführen, neuen Operator abkellern
D Kelleroperation durchführen (evtl. wiederholt)
L Ein Kellerelement löschen
F Fehler

X Exit, fertig
K Operator abkellern
G Kelleroperation durchführen, neuen Operator abkellern
D Kelleroperation durchführen (evtl. wiederholt)
L Ein Kellerelement löschen
F Fehler

Listing 8.1. 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;   // Operandenhoehe, Zahlenhoehe
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 loeschen
61                      oh--;
62                      break;
63                  case 'F':
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      int rc = 0;
78      switch (operator) {
79      case PLUS:  *z3 = z1 + z2; break;
80      case MINUS: *z3 = z1 - z2; break;
81      case MAL:   *z3 = z1 * z2; break;
82      case DIV:   if (z2 == 0)
83                      rc = 1;
```

```
84                  else
85                      *z3 = z1 / z2;
86              break;
87      }
88      return rc;
89  }
90
91  int main() {
92      char t[256];
93      int retcode;
94      double z;
95      while (printf("Rechnung: "),fgets(t, 255, stdin) != NULL)
96          if ((retcode = kellerautomat(t, &z)) == 0)
97              printf("Ergebnis: %lf\n\n", z);
98          else
99              printf("Fehler %d\n\n", retcode);
100     printf("\n");
101     return 0;
102 }
```

8.3 LL-Parsing

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

8.3.1 Einführung

Betrachten wir Grammatik Γ_{18}:

Grammatik Γ_{18}: 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“ vom Wurzelknoten ausgehend 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".

Etwas problematisch sind ϵ-Ableitungen in Regeln, wie Grammatik Γ_{21} zeigt.: **Grammatik Γ_{19}: LL(1)-Grammatik**
S → [**A**] **B**
A → **x y**
B → **x z**
Grammatik Γ_{20}: LL(1)-Grammatik
S → **A B** | **B**
A → **x y**
B → **x z**
Grammatik Γ_{21}: LL(1)-Grammatik
S → **A B**
A → **x y** | ϵ
B → **x z**

Definition 8.1 (LL(1)-Sprache). *Eine Sprache ist LL(1), wenn*

1. *bei allen Alternativen in der Grammatik die FIRST-Mengen der Alternativen disjunkt sind*
   und
2. *bei allen Regeln, die nach ϵ ableitbar sind, die FIRST- und die FOLLOW-Menge der Regel disjunkt sind.*

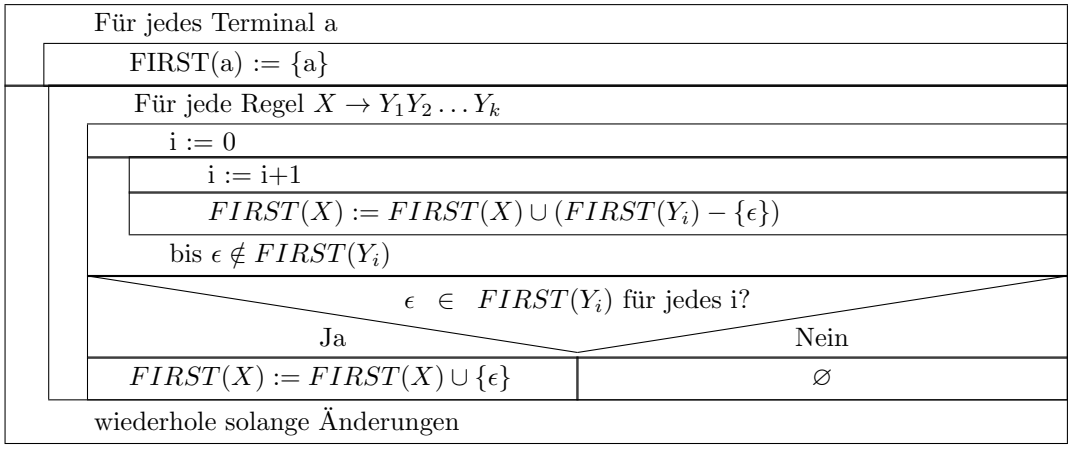
8.3.2 FIRST und FOLLOW

Definition 8.2 (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 8.3 (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.*

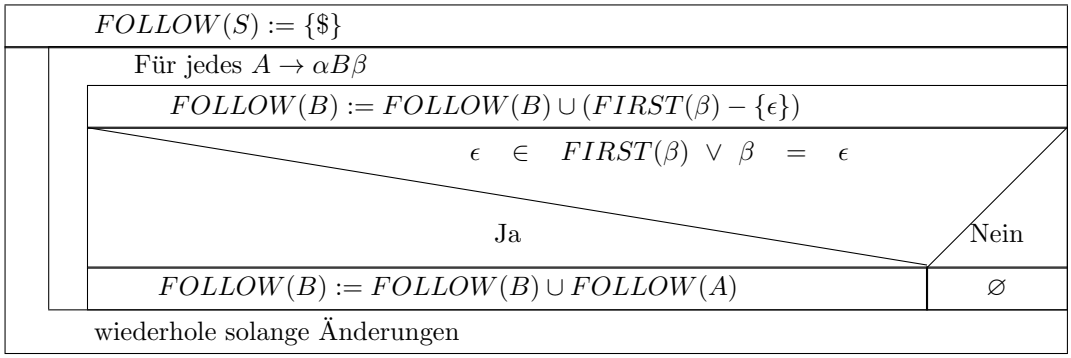
Algorithmus 4: Berechnung der FIRST-Mengen

Berechnung der FIRST-Mengen



Algorithmus 5: Berechnung der FOLLOW-Mengen

Berechnung der FOLLOW-Mengen



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

8.3.3 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.

- Bei RD-Parsern findet sich häufig das folgende Konstrukt:

```
// ...
if (token == t_XYZ)
        token = scanner(); // Alles OK, weiterlesen
else
        error();
```

- Daher kürzer über Unterfunktion void match(int);:

```
void match(int t)
        if (t == token)
        token = scanner(); // Alles OK, weiterlesen
else
        error();
```

- Damit die häufig ähnlichen Programmstellen statt oben:

```
// ...
match(t_XYZ);
```

Nun lässt sich ein Recursive-Descent-Parser zu Grammatik 16 entwickeln.

```
int main(void) {
  error = 0, token = yylex();
  f_expression();
  printf("Erg.: %d\n",
    error || token != 0);
  return 0;
}


void f_expression(void) {
  f_term();
  while(token == t_plus) {
    token = yylex();
    f_term();

  }

}

void f_term(void) {
```

```
int main(void) {
  error = 0, token = yylex();
  f_expression();
  printf("Erg.: %d\n",
       error || token != 0);
  return 0;
}


void f_expression(void) {
    f_term();
  while(token == t_plus) {
    token = yylex();
    f_term();
    |printf("\tadd\n");|
  }

}

void f_term(void) {
  f_factor();
  while(token == t_mal) {
    token = yylex();
    f_factor();
    |printf("\tmult\n");|
  }

}

void f_factor (void) {

  if (token == t_minus) {
    token = yylex();
    f_factor();
    |printf("\tchs\n");|
  }
  else if (token == t_zahl) {
    |printf("\tloadc\t%s\n", yylval.txt);|
    token = yylex();
  }
  else if (token == t_kla_auf) {
    token = yylex();
    f_expression();
    if (token == t_kla_zu)
      token = yylex();
    else
      error = 1;
  }
  else error=2;

}



int main(void) {
  error = 0, token = yylex();
  |int erg = |f_expression();
  printf("Erg.: %d Num. %d\n",
   error || token != 0, erg);
  return 0;
}


int f_expression(void) {
  |int erg =  f_term();|
  while(token == t_plus) {
    token = yylex();
    |erg += f_term();|

  }
  return erg;
}

int f_term(void) {
  |int erg =  f_factor();|
  while(token == t_mal) {
    token = yylex();
    |erg *= f_factor();|

  }
  return erg;
}
```

8.3.4 LL-Parsing mit Stackverwaltung

LL-Parsing mit Stackverwaltung

Gegeben ist linksrekursionsfreie Grammatik $\Gamma_{??}$.

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

<table>
<tr><td colspan="4">Gegeben: PTABLE[TOS][token]</td></tr>
<tr><td colspan="4">Lege Ende-Token auf Stack</td></tr>
<tr><td colspan="4">Lege Startsymbol auf Stack</td></tr>
<tr><td colspan="4">Lies token</td></tr>
<tr><td colspan="4">error = 0</td></tr>
<tr><td colspan="4">Eingabe nicht leer AND error = 0</td></tr>
<tr><td colspan="4">TOS ist Terminal
Ja / Nein</td></tr>
<tr><td colspan="2">TOS = token?
Ja / Nein</td><td colspan="2">PTABLE[TOS][token] gesetzt?
Ja / Nein</td></tr>
<tr><td>lies token</td><td>error = 1</td><td>lösche TOS</td><td>error = 1</td></tr>
<tr><td colspan="2">∅</td><td>Lege Regel PTABLE[TOS][token] rückwärts auf Stack</td><td>∅</td></tr>
</table>

Voraussetzung für den Algorithmus ist eine linksrekursionsfreie Grammatik.

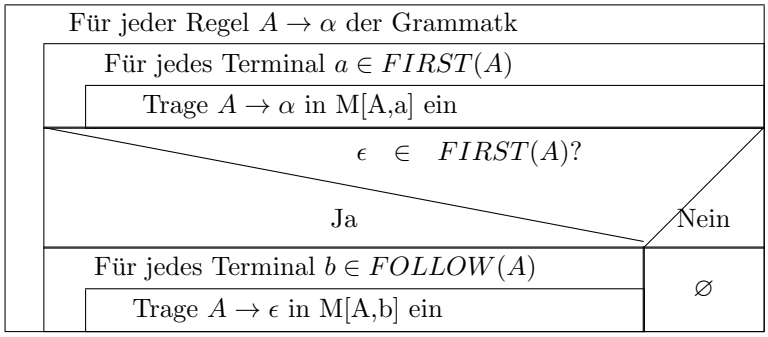
Anwendung von Algorithmus 4 zur Berechnung der FIRST-Mengen bzw. Algorithmus 5 zur Berechnung der FOLLOW-Mengen auf Grammatik $\Gamma_{??}$ ergibt

Tabelle 8.1. Parsing-Vorgang

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

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

Algorithmus 7: LL(1)-Tabelle erstellen



Siehe [ASU88], Seite 231ff.

Beispiel Grammatik $\Gamma_{??}$ in Tabelle 8.2

Tabelle 8.2. LL(1)-Parsetabelle

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

Listing 8.2 implementiert den hier erläuterten Parser.

Listing 8.2. LL(1)-Parser mit Stackverwaltung (ll1-stack.c)

```
1 /*******************************************************
2 Regel 1: E  ::= T E'
```

```
3  Regel 2: E' ::= + T E'
4  Regel 3: T  ::= F T'
5  Regel 4: T' ::= * F T'
6  Regel 5: F  ::= ( E )
7  Regel 6: F  ::= zahl
8  Regel 7: F  ::= - F
9  ************************************************************/
10 #include <stdio.h>
11 #include "term-scanner.h"
12 #include "term-scanner.c"
13 #include "genprog-stack.h"
14 #include "genprog-stack.c"
15
16 enum {nt_E , nt_Es, nt_T, nt_Ts, nt_F}; // Non-Terminals
17 typedef enum {type_t, type_nt} element_type;
18 struc stack_element {
19     element_type t;
20     int val;
21 }
22 map<int, map<int, int> ptab;
23 ptab[nt_E][t_kla_auf] = 0;
24
25 int g[11][4] = { // Grammatik
26 /*0*/ {  nt_E, END,  -1    },
27 /*1*/ {  nt_T, nt_E2,   -1    },
28 /*2*/ {  PLUS, nt_T, nt_E2,   -1 },
29 /*3*/ {  MINUS,   nt_T, nt_E2,   -1 },
30 /*4*/ {-1             },
31 /*5*/ {nt_F,   nt_T2,   -1    },
32 /*6*/ {MAL, nt_F, nt_T2,   -1 },
33 /*7*/ {DIV, nt_F, nt_T2,   -1 },
34 /*8*/ {-1             },
35 /*9*/ {KLA_AUF,   nt_E, KLA_ZU,  -1 },
36 /*10*/{ZAHL,   -1       },
37 };
38
39 stack s;
40
41 void regel_auf_stack(int r) {
42     int i;
43     i = -1;
44     printf("regel %d\n", r);
45     while (i++, g[r][i] >=0);
46     for (i--; i >= 0; i--)
47         stack_push(&s, &(g[r][i]));
48 }
49
50 int main() {
51      char input[] = " (1 * 2) - 2 + 3";
52      char text[255];
53      int token, error = 0, r, tos;
54
55      stack_init(&s, sizeof(int), 100, int_print);
56      term_scanner(input, NULL); // Scanner-Init
```

```
57      token = term_scanner(NULL, text);
58
59      regel_auf_stack(0); // Startregel auf Stack
60      while (!error && stack_height(&s)) {
61          if (stack_pop(&s, &tos), tos < nt_offset) // Terminal!
62              if (tos == token)
63                  token = term_scanner(NULL, text);
64              else
65                  error = 1;
66          else // Non-Terminal
67              if ((r = parse_table[tos - nt_offset ][token]) >= 0)
68           regel_auf_stack(r);
69              else
70                  error = 1;
71      }
72      printf("Parse-ergebnis: %d\n", error);
73     stack_del(&s);
74      return 0;
75  }
```

8.3.5 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.

8.3.6 Tabellen-gesteuertes LL-Parsing

Siehe auch [Wir96], Seite 22ff.

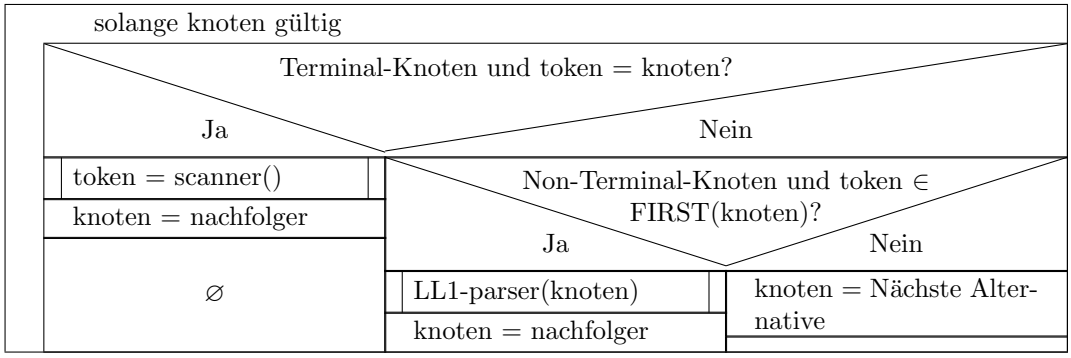
Am einfachsten basierend auf Syntaxgraphen der Regeln. Für jede Pfeilspitze wird ein Viertupel angelegt:

- 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

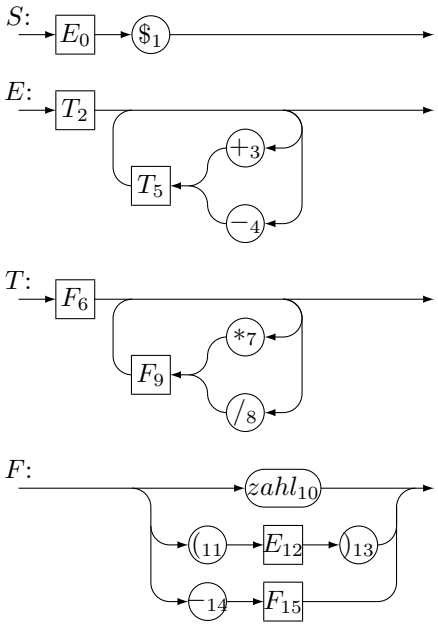
LL1-parser(knoten)



Als Nebeneffekt der beschriebenen Datenstruktur für die Syntaxdiagramme erscheint die Tatsache, dass die FIRST-Mengen einfach berechnet werden können. Listing 8.3 implementiert einen tabellengesteuerten LL(1)-Parser, der komplett unabhängig von der Grammatik codiert ist.

Als Beispiel soll für die Term-Grammatik $\Gamma_{??}$ (Seite **??**) ein Parser entwickelt werden. Dazu wird Syntax-Diagramm 7 entwickelt.

Syntaxdiagramm 7: Term-Grammatik



Listing 8.3. 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 8.3. 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  }
```

8.4 LR-Parsing

Bottom-Up-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.

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.
- Letzten Endes ist dieses Verfahren ein String-Matching, was ineffizient ist
- 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:

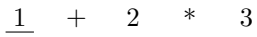
Reduziere das Handle am Stackende anhand einer Regel

Schiebe das nächste Eingabeelement auf den Stack

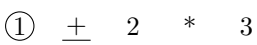
Akzeptiere die Eingabe

Fehler in der Eingabe

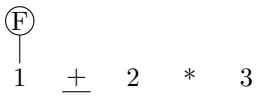
Zuerst soll der Syntaxbaum für den Ausdruck $1 + 2 * 3$ erstellt werden. In den folgenden Grafiken sind die eingekreisten Symbole “auf dem Stack“, das unterstrichene Symbol das nächste der Eingabe, also das Look-ahead-token.



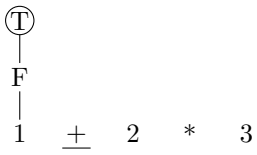
Da der Stack leer ist kann nur geschoben werden. Die 1 kommt auf den Stack, in der Eingabe steht nun das + an:



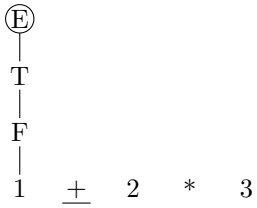
Es findet sich kein Handle, das mi “zahl +“ begint, daher kann nicht sinnvoll geschoben werden. Allerdings findet sich ein Handle “zahl“, welches nun zu “F“ reduziert werden kann:



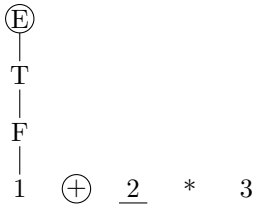
Auch “F +“ ist wird kein sinnvolles handle, aber “F“ kann zu “T“ reduziert werden:



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

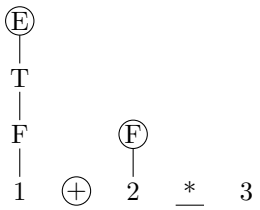


Nu kann “E“ nicht weiter reduziert werden, aber mit “E +“ beginnt ein Handle, daher wird geschoben:

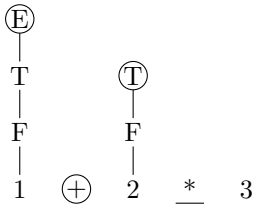


Es findet sich kein Handle, welches reduziert werden kann, daher kann nur geschoben werden:

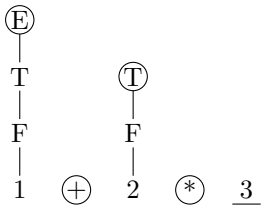
Als einzige sinnvolle Aktion findet sich nun die Reduktion von “zahl“ zu “F“:



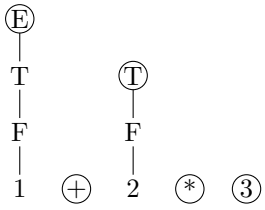
Nun kann lediglich “F“ zu “T“ reduziert werden:



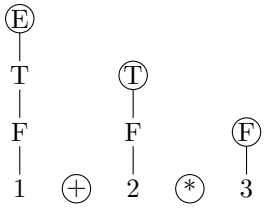
Jetzt entsteht ein Schiebe/Reduziere-Konflikt. Man könnte “E + T“ reduzieren oder aber schieben, da mit “T *“ auch ein Handle beginnt. Ohne zu begründen - wir schieben!



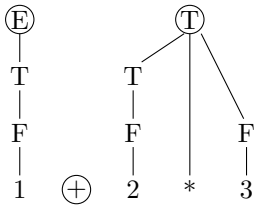
Nun kann wieder nur geschoben werden, die Eingabe ist leer:



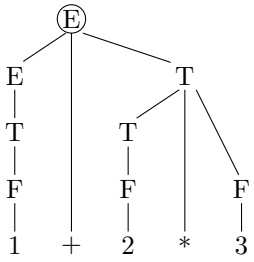
Wir reduzieren “zahl“ zu “F“:



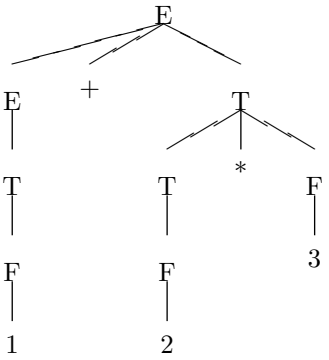
Nun kann “T * F“ zu “T“ reduziert werden:



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 8.4 dargestellt werden.

Tabelle 8.4. 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 Compiler-

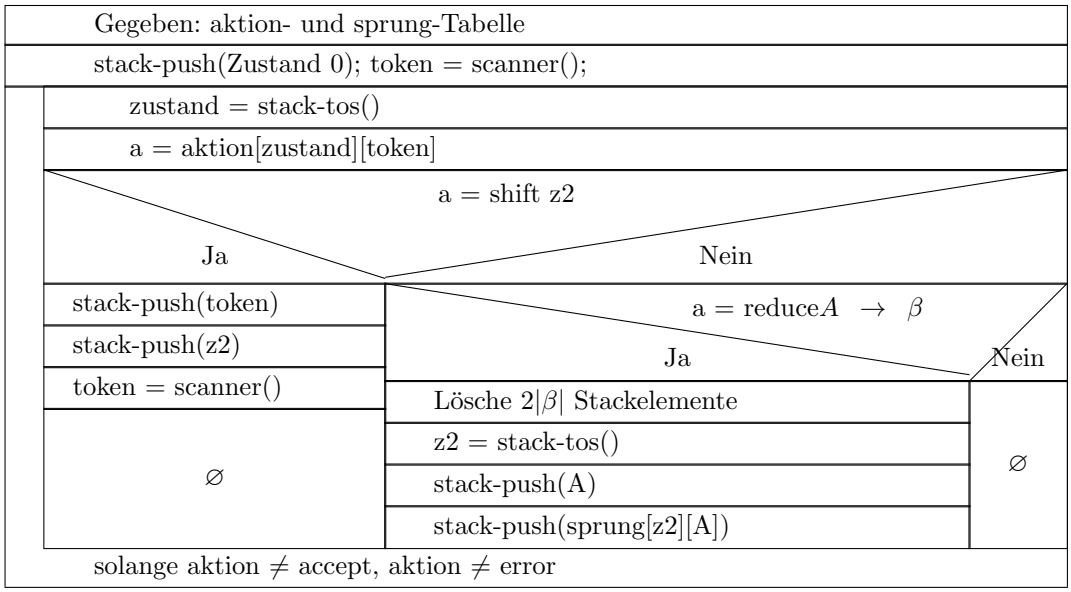
generatoren erstellt. Für ein Verständnis der Compilergeneratoren ist aber ein prinzipielles Verständnis dieses Verfahren notwendig.

Tabelle 8.5. 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 8.4 und die Eingabe $1+2*3$ an so ergibt sich der in Tabelle 8.6 dargestellte Parse-Vorgang:

Listing 8.4. 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] = {
9          /* $    +     -     *     /     (     )    zahl  err
```

Tabelle 8.6. 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

```
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 10.4 ab Seite 125. Der PL/0-Parser des Modellcompilers besteht aus einer Tabelle mit 99 Zuständen!

Als ein Beispiel für die Generierung von Tabellen betrachten wir Listing 8.5, einer Implementierung unserer Term-Grammatik in Yacc.

Listing 8.5. 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 8.5 ist diese Information in Listing **??** beschrieben: 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			

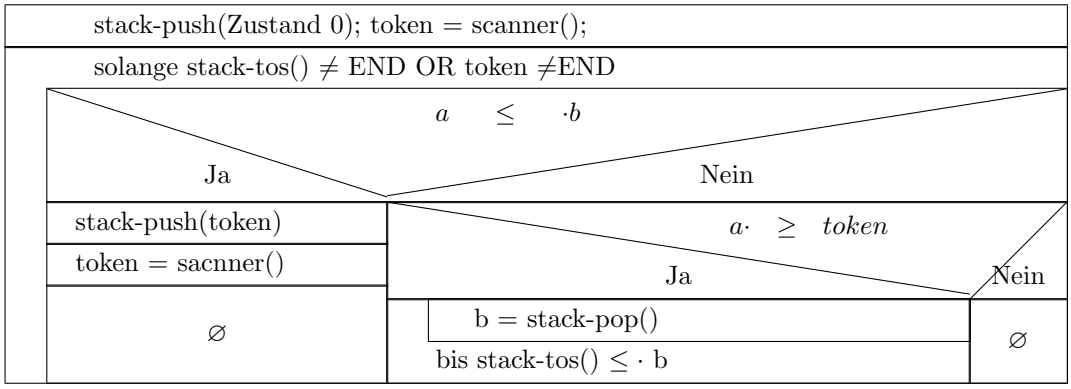
8.4.1 Probleme beim Bottom-Up-Parsen

8.5 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 8.6. 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 }
```

9

Verarbeitung besonderer Sprachen

9.1 Mehrdeutige Grammatiken

9.1.1 Das Dangling-Else-Problem

Siehe [KR88]

[KR88] Seite 54:

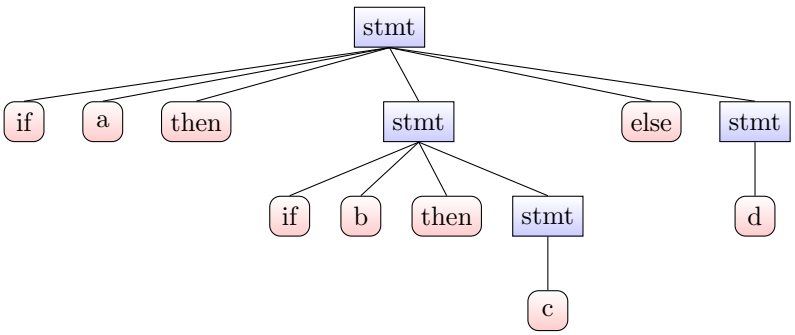
3.2 If-Else

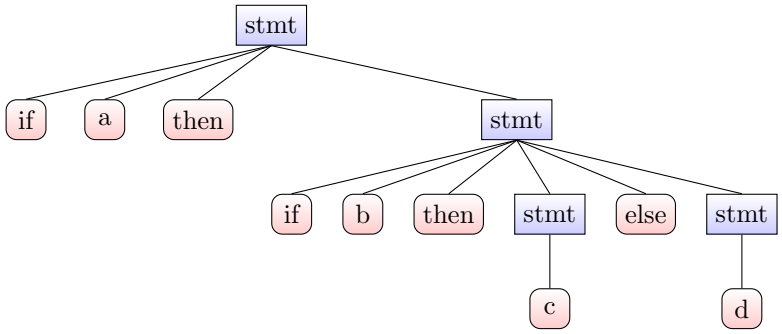
The if-else statement is used to express decisions. Formally, the syntax is

```
if (expression)
    statement1
else
    statement2
```

Because the else part of an if-else is optional, there is an ambiguity when an else is omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if.

Grammatik Γ_{22}: if-else-Grammatik
1) stmt → 'id'
2) stmt → 'if' 'id' 'then' stmt
3) stmt → 'if' 'id' 'then' stmt 'else' stmt





Das Dangling-Else-Problem im Bottom-Up-Parsen

Tabelle 9.1. LR-Parse-Vorgang „if a then if b then c else d“ (1)

Stack	Input	Aktion
	`if a then if b then c else d $`	s
`if`	`a then if b then c else d $`	s
`if id`	`then if b then c else d $`	s
`if id then`	`if b then c else d $`	s
`if id then if`	`b then c else d $`	s
`if id then if id`	`then c else d $`	s
`if id then if id then`	`c else d $`	s
`if id then if id then id`	`else d $`	r1
`if id then if id then stmt`	`else d $`	r2 ???
`if id then stmt`	`else d $`	s
`if id then stmt else`	`d $`	s
`if id then stmt else id`	`$`	r1
`if id then stmt else stmt`	`$`	r3
`stmt`	`$`	a

Tabelle 9.2. LR-Parse-Vorgang „if a then if b then c else d“ (2)

Stack	Input	Aktion
	`if a then if ... $`	s
`if`	`a then if ... $`	s
`if id`	`then if ... $`	s
`if id then`	`if b then c else d $`	s
`if id then if`	`b then c else d $`	s
`if id then if id`	`then c else d $`	s
`if id then if id then`	`c else d $`	s
`if id then if id then id`	`else d $`	r1
`if id then if id then stmt`	`else d $`	s ???
`if id then if id then stmt else`	`d $`	s
`if id then if id then stmt else id`	`$`	r1
`if id then if id then stmt else stmt`	`$`	r3
`if id then stmt`	`$`	r2
`stmt`	`$`	a

Das Dangling-Else-Problem im Top-Down-Parsen

Grammatik Γ_{23}: if-else-Grammatik
1) stmt $\rightarrow$ **'id' | 'if' 'id' 'then' stmt stmt_else**
2) stmt_else $\rightarrow$ **'else' stmt |** ϵ

```
if a then if b then c else d
```

$$FIRST(stmt) = \{id, if\}$$
$$FIRST(stmt\_else) = \{else, \epsilon\}$$

$$FOLLOW(stmt) = \{\$, else\}$$
$$FOLLOW(stmt\_else) = \{\$, else\}$$

```
void f_if_else { // t_if liegt sicher an!
        match(t_if);
        match(t_id);
        match(t_then);
        f_stmt();
        if (token == t_else) {
                match(t_else);
                f_stmt();
        }
}
```

Kein Problem wäre

```
if a then
        if b then
                c
else
        d
```

Großes Problem wäre

```
while a do
        if b then
                c
        else
                d
```

9.1.2 Packrat-Parser

9.2 Einrückungs-Sprachen

- PUG
- Python
- Markdown

- Einrückungs-Sprachen sind keine Typ-2-Sprachen
- Trick:
  - Scanner verwaltet Einrückung und liefert bei Änderung Meta-Token
    - token_indent (Einrückung)
    - token_dedent (Ausrückung)
  - Parser bekommt Tokenfolge für kontextfreie Sprache

```
if a < 0:
   a = -a
   b = b + 1
print(a)
```

Tokenfolge:

```
t_if t_id(a) t_lt t_const(0) t_colon t_eol
t_block_open
t_id(a) t_assign t_minus t_id(a) t_eol
t_id(b) t_assign t_id(b) t_plus t_const(1) t_eol
t_block_close
t_id(print) t_bra_open t_id(a) t_bra_close t_eol
```

Listing 9.1. PUG-Tokenizer (indent-languages/pug-ausgabe.l)

```
%x INDENT
%%
<<EOF>>                 BEGIN 0; ind_soll = 0; ind(); return 0;
<INDENT>^[ \t]*/[^ \t]  BEGIN 0; ind_soll = yyleng; ind();
\n                      BEGIN INDENT;
.+                      printf(" '%s '\n", yytext);
%%
void ind() {
    while (ind_ist != ind_soll) {
        printf("%c\n", (ind_soll > ind_ist) ? '{' : '}');
    ind_ist += (ind_soll > ind_ist) - (ind_soll < ind_ist);
    }
}
int main() {
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

Eingabe:

Listing 9.2. PUG-Code (indent-languages/1.pug)

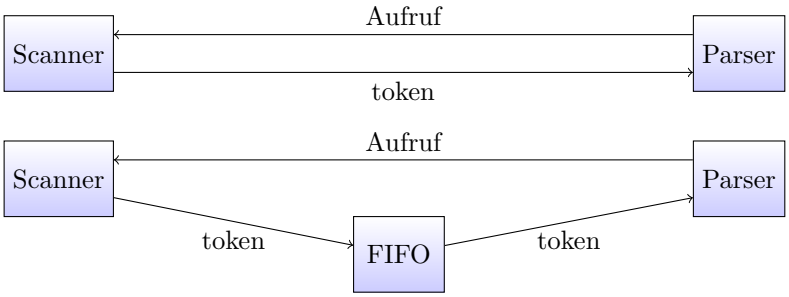
```
html
    head
        title
    body
```

```
5       div
6       div
```

Ausgabe:

```
'html' { 'head' { 'title' } 'body' { 'div' 'div' } }
```

- Typischer Scanner:
  - `token = scanner()`
- Indent-Scanner:
  - `token = scanner()`
  - FIFO o.ä. notwendig



Grammatik Γ_{24}: PUG-Grammatik
S → F { F }
F → 'tag' ['{' S '}']

Listing 9.3. Test (indent-languages/pug-parser.c)

```
1  int main() {
2      BEGIN INDENT;
3      rc = 0, token = yylex();
4      f_S(0);
5      if (token != t_end)
6          error(4);
7      return rc;
8  }
9
10 void f_S (int level) {
11     f_F(level);
12     while (token == t_tag) {
13         f_F(level);
14     }
15 }
```

Listing 9.4. Test (indent-languages/pug-parser.c)

```
1  void f_F(int level) {
2      char txt[100];
3      static char tabs[] = "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t",
4          *tab = tabs + sizeof(tabs) -1;
5      strcpy(txt, yytext);
6      if (token == t_tag) {
7          token = yylex();
```

```
8         if (token == t_indent) {
9            printf("%s<%s>\n", tab - level, txt);
10           token = yylex();
11           f_S(level+1);
12           match(t_dedent,1);
13           printf("%s</%s>\n", tab - level, txt);
14        }
15        else
16           printf("%s<%s/>\n", tab - level, txt);
17     }
18     else
19        error(2);
20  }
```

Listing 9.5. Ein Scanner für PUG (indent-languages/pug-einfach.l)

```
1  %{
2     int rc;
3     #include "lex-indent.h"
4  %}
5  %x DEDENT INDENT
6  %%
7  <INDENT>^[ \t]*/[^ \t]   if ((rc = lex_indent(yyleng))) return rc;
8  <DEDENT>.                if ((rc = lex_indent(-1))) return rc;
9  <<EOF>>                  return lex_indent(0);
10 \n                    BEGIN INDENT;
11
12 .+     return t_tag;
13 %%
14 #include "lex-indent.c"
15
16 int yywrap() {
17    return 1;
18 }
```

Listing 9.6. Ein Scanner für PUG (indent-languages/lex-indent.c)

```
1  int lex_indent(int s) {
2     int rc = 0;
3     static int ind_ist = 0, ind_soll = 0;
4     if (s >= 0)
5        ind_soll = s;//yyleng;
6     //printf("ded %d\n", s);
7     if (ind_ist == ind_soll -1) {
8        ind_ist++, rc = t_indent;
9        BEGIN 0;
10    }
11    else if (ind_ist > ind_soll) {
12       BEGIN DEDENT;
13       yyless(0);
14       ind_ist--, rc = t_dedent;
15    }
16    else if (ind_ist == ind_soll)
17       BEGIN 0;
18    return rc;
```

```
19 }
20
21 /*
22 int lex_eofdent() {
23     return (ind_ist--) ? t_dedent : t_end;
24 }
25
26 int __lex_dedent__() {
27     int rc = 0;
28     yyless(0);
29     if (ind_ist > ind_soll)
30         ind_ist--,rc = t_dedent;
31     else
32         BEGIN 0;
33     return rc;
34 }
35 */
```

Listing 9.7. PUG-Scannertest (indent-languages/pug-wrapper.l)

```
1  %{
2      int ind_ist = 0, ind_soll = 0;
3      enum {t_end, t_indent, t_dedent, t_tag};
4     #define YY_DECL int yylex2 (void)
5  %}
6  %x INDENT
7  %%
8  <<EOF>>                 BEGIN 0; ind_soll = 0; return 0;
9  <INDENT>^[ \t]*/[^ \t]  BEGIN 0; ind_soll = yyleng; if (ind_ist != ind_soll) return -
10 \n                      BEGIN INDENT;
11 .+      return t_tag;
12 %%
13 int yylex() {
14     int rc;
15     if (ind_ist == ind_soll)
16         rc = yylex2();
17     if (ind_ist > ind_soll)
18         ind_ist--, rc = t_dedent;
19     else if (ind_ist < ind_soll)
20         ind_ist++, rc = t_indent;
21     return rc;
22 }
23
24 /*
25 int yylex() {
26     int rc;
27     if (ind_ist > ind_soll)
28         ind_ist--, rc = t_dedent;
29     else {
30         rc = yylex2();
31         if (rc == t_indent)
32             ind_ist++;
33         else if (rc == t_dedent)
34             ind_ist--;
35     }
```

```
36     return rc;
37 }*/
38 int main() {
39     int token;
40     while ((token = yylex()) != 0) {
41         printf("%d '%s'\n", token, yytext);
42     }
43 }
44 int yywrap() {
45     return 1;
46 }
```

Listing 9.8. Einrückungs-Term (1) (indent-languages/ind1.txt)

```
1 +
2     1
3     *
4         2
5         3
6     4
```

Listing 9.9. Ein Scanner für Einrückungs-Terme (indent-languages/indent-term-scanner.l)

```
1  %{
2  int rc;
3  #include "lex-indent.h"
4  %}
5  %x DEDENT INDENT
6  %%
7  <INDENT>^[ \t]*/[^ \t]  if ((rc = lex_indent())) return rc;
8  <DEDENT>.               if ((rc = lex_indent())) return rc;
9  <<EOF>>                 return lex_eofdent();
10 \n                   BEGIN INDENT;
11
12 "+"$      return t_plus;
13 "-"$      return t_minus;
14 "*"$      return t_mal;
15 "/"$      return t_div;
16 [0-9]+$      return t_zahl;
17 .         printf("??? = %s\n", yytext); return 0;
18 %%
19 #include "lex-indent.c"
20
21 int yywrap() {
22     return 1;
23 }
```

Listing 9.10. Ein Scanner für Einrückungs-Terme (indent-languages/indent-term-scanner-main.c)

```
1 enum {t_end, t_plus, t_minus, t_mal, t_div, t_zahl,
2     t_indent, t_dedent, t_error};
3 #include "lex.yy.c"
4 int main() {
5     int token;
6     while ((token = yylex()) != t_end) {
```

```
7         switch (token) {
8             case t_indent:
9                 printf("{");
10                break;
11            case t_dedent:
12                printf("}");
13                break;
14            default:
15                printf(" %s ", yytext);
16        }
17    }
18    printf("\n");
19    return 0;
20 }
```

Grammatik Γ_{25}: Indent-Term-Grammatik (1)
S $\rightarrow$ **OP '{' S S { S } '}' | 'zahl'**
OP $\rightarrow$ **'+' | '-' | '*' | '/'**

Grammatik Γ_{26}: Indent-Term-Grammatik (2)
S $\rightarrow$ **OP '{' S S WDH '}' | 'zahl'**
WDH $\rightarrow$ **S WDH** $\mid \epsilon$
OP $\rightarrow$ **'+' | '-' | '*' | '/'**

$$
\begin{array}{llll}
FIRST(S) & = \{+, -, *, /, zahl\} & FOLLOW(S) & = \{\$, +, -, *, /, zahl, '\}'\} \\
FIRST(WDH) & = \{+, -, *, /, zahl, \epsilon, \} & FOLLOW(WDH) & = \{'\}'\} \\
FIRST(OP) & = \{(, -, zahl\} & FOLLOW(OP) & = \{+, \$,)\}
\end{array}
$$

Listing 9.11. Einfache HTML-Seite (PUG) (indent-languages/1.pug)

```
1 html
2     head
3         title
4     body
5         div
6         div
```

↓

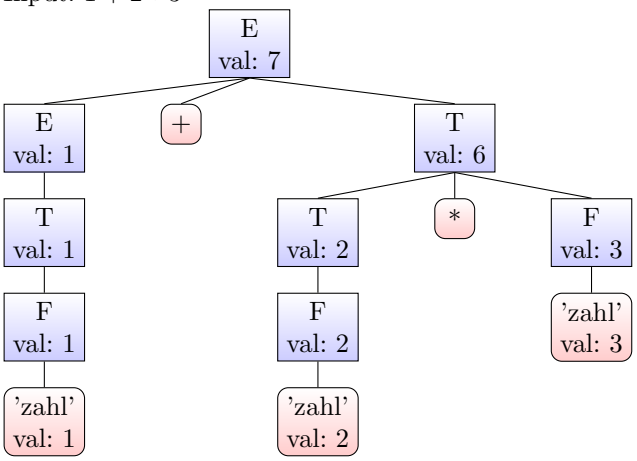
Listing 9.12. Einfache HTML-Seite (PUG) (indent-languages/1.html)

```
1 <html>
2     <head>
3         <title/>
4     </head>
5     <body>
6         <div/>
7         <div/>
8     </body>
9 </html>
```

9.3 Attributierte Grammatiken

- Idee: Knuth 1966
- Syntaxbäume werden um Attribute erweitert
- Grammatik wird um Regeln zur Berechnung der Attribute erweitert
- Grammatik wird um Bedingungen zur Semantischen Analyse erweitert

Grammatik: Wie üblich...
Input: $1 + 2 * 3$



Eine Attributierte Grammatik ist (nach Knuth) ein Viertupel $AG = (G, A, R, B)$:

$$G = (N, T, P, Z) \quad \text{ist eine kontextfreie Grammatik}$$
$$A = \bigcup_{X \in (T \cup N)} A(X) \quad \text{ist eine endliche Menge von Attributen}$$
$$R = \bigcup_{p \in P} R(p) \quad \text{ist eine endliche Menge von Attributionsregeln}$$
$$B = \bigcup_{p \in P} B(p) \quad \text{ist eine endliche Menge von Bedingungen}$$

- Verarbeitung kontextsensitiver Sprachen als kontextfreie Sprachen bspw.
  - XML und andere Klammersprachen
  - Programmiersprachen mit Deklarationspflicht
- Semantische Analyse bspw.
  - Ganzzahlige Typen bei Array-Indizes
  - LValues

Attributarten:

Synthetisiert : rechte Seite einer Produktion bestimmt linke Seite (bottomup)

Ererbt : linke Seite bestimmt rechte Seite (topdown)

$X \rightarrow X_1\ X_2\ X_3\ \ldots\ X_{n-1}\ X_n$

Definition 9.1 (S-Attributgrammatik:). *S-Attributgrammatiken sind Attributgrammatiken, die nur auf synthetischen Attributen arbeiten.*

Definition 9.2 (L-Attributgrammatik:). *Eine AG ist L-Attributiert wenn jedes ererbte Attribut eines jeden Symbols X_j in einer Regel*
$A \to X_1 X_2 X_3 \ldots X_{n-1} X_n$
dieses Attribut nur abhängig ist von

- *Attribute der Symbole $X_1 \ldots X_{j-1}$*
- *A*

Anmerkung: Jede S-AG ist per definitionem implizit auch L-AG!

- Terminal-Attribute werden vom Scanner geliefert
  Beispiele:
  - Zahlen- und String-Konstanten
  - Identifier
- Werden nur zur Synthese verwendet
- Werden nicht ererbt

- YACC / Bison unterstützen nur S-Attribute
- RD-Parser nur einfach mit S-Attributen

Listing 9.13. XML-Namespaces (symboltable/xml-namespaces.xml)

```
1  <html xmlns="http://www.w3.org/1999/xhtml"
2      xmlns:svg="http://www.w3.org/2000/svg">
3      <head>
4          <title>Beispiel-Datei mit mehreren Namensraeumen</title>
5      </head>
6      <body>
7          <math xmlns="http://www.w3.org/1998/Math/MathML">
8              <mi>x</mi><mo>=</mo><mn>2</mn>
9          </math>
10         <svg:svg>
11             <svg:rect x="0" y="0" width="10" height="10" />
12             <svg:text>
13                 <svg:tspan>Eine Formel in der Grafik:</svg:tspan>
14                 <svg:tspan>
15                     <math xmlns="http://www.w3.org/1998/Math/MathML">
16                         <mi>y</mi><mo>=</mo><mn>1</mn>
17                     </math>
18                 </svg:tspan>
19             </svg:text>
20         </svg:svg>
21     </body>
22 </html>
```

Beispiele:

- Term-Interpreter
- XML-Namespace

Grammatik Γ_{27}: S $\{x^n y^n z^n | \ n > 0\}$
Sprache S: $\{x^n y^n z^n | \ n > 0\}$

Produktion	Regel	Bedingung
S → X Y Z		**n(X) = n(Y) = n(z)**
X → 'x'	**n(X) = 1**	
X → X_1 'x'	**n(X) = n(X_1) + 1**	
Y → 'y'	**n(Y) = 1**	
Y → Y_1 'y'	**n(Y) = n(Y_1) + 1**	
Z → 'z'	**n(Z) = 1**	
Z → Z_1 'x'	**n(Z) = n(Z_1) + 1**	

Listing 9.14. Nicht-KFG (attribute-grammar/x-n-y-n-z-n.y)

```
1  %union {int n;}
2  %type<n> X Y Z
3  %%
4  S: X Y Z {
5      if ($1 != $2 || $2 != $3) YYERROR;
6  }
7  X: 'x' {$$ = 1;}
8      | X 'x' {$$ = 1 + $1;}
9      ;
10 Y: 'y' {$$ = 1;}
11     | Y 'y' {$$ = 1 + $1;}
12     ;
13 Z: 'z' {$$ = 1;}
14     | Z 'z' {$$ = 1 + $1;}
15     ;
16 %%
```

Grammatik Γ_{28}: XML als Attribut-Grammatik

Produktion	Regel	Bedingung
XML → TAG		
TAG →TG_O CONTENT TG_C **TAG → TAG_S**		**n(TG_O) = n(TG_C)**
CONTENT → { 'text' | TAG }		
TG_O → '<' 'id' '>'	**n(TG_O) = n(id)**	
TG_C → '</' 'id' '>'	**n(TG_C) = n(id)**	
TG_S → '<' 'id' '/>'		

Listing 9.15. XML-Parser mit Attributiertewr Grammatik (attribute-grammar/xml-parser-y.y)

```
1  %union {char *txt;}
2  %token<txt> t_tag_o t_tag_c t_tag_s t_text t_wht
3  %%
4  xml:  tag | tag t_wht
5      ;
6  tag   :  t_tag_s
7      |  t_tag_o content t_tag_c {
8         if (strcmp($1,$3)) { // Semantic attribute
9            sprintf(txt, "<%s> ==> </%s>", $1, $3);
10           yyerror(txt);
11           YYERROR;
12        }
13     }
```

```
14      ;
15  content: /* epsilon */
16      | content tag
17      | content t_text
18      | content t_wht
19      ;
20  %%
```

Grammatik Γ_{29}: Term als Attribut-Grammatik

Produktion	Regel	Bedingung
$\mathbf{E \to T_1}$	$\mathbf{val(E) = val(T_1)}$	
$\mathbf{E \to E_1}$ **'+'** $\mathbf{T_3}$	$\mathbf{val(E) = val(E_1) + val(T_3)}$	
$\mathbf{T \to F}$	$\mathbf{val(T) = val(F_1)}$	
$\mathbf{T \to T}$ **'*'** $\mathbf{F}$	$\mathbf{val(T) = val(T_1) * val(F_3)}$	
F → 'zahl'	**val(F) = val(zahl)**	
$\mathbf{F \to}$ **'('** $\mathbf{E}$ **')'**	$\mathbf{val(F) = val(E_2)}$	
$\mathbf{F \to}$ **'-'** $\mathbf{F}$	$\mathbf{val(F) = -val(F_2)}$	

Grammatik Γ_{30}: C-Division als Attribut-Grammatik

Produktion	Regel	Bedingung
$\mathbf{T \to F}$	$\mathbf{val(T) = val(F_1)}$ $\mathbf{typ(T) = typ(F_1)}$	
$\mathbf{T \to T}$ **'/'** $\mathbf{F}$	**Falls** $\mathbf{typ(T_1) = typ(F_3) =}$ **int:** $\mathbf{val(T) = val(T_1) / val(F_3)}$ **(ganzzahl.)** $\mathbf{typ(T) =}$ **int** **sonst:** $\mathbf{val(T) = val(T_1) / val(F_3)}$ **(kommazahl.)** $\mathbf{typ(T) =}$ **double**	
$\mathbf{T \to T}$ **'%'** $\mathbf{F}$	$\mathbf{val(E) = val(T_1) \% val(F_3)}$ $\mathbf{typ(E) =}$ **int**	$\mathbf{typ(F_1) = typ(F_3) =}$ **int**

9.4 Übungen

- Erweitern Sie die Term-Grammatik so, dass + und - auch unär verwendet können (genau eine Einrückungszeile)
- Erweitern Sie die Grammatik (oder den Scanner) um Konstanten wie π, e
- Erweitern Sie die Grammatik um Funktionen (mit genau einem Argument) wie $\sin(x)$ etc.
- Erweitern Sie den Parser zu einem Compiler
- Erweitern Sie den Parser zu einem Interpreter
- Korrigieren Sie die PUG-Grammatik dass nur ein Wurzelelement erlaubt ist.
- Erweitern Sie den PUG-Parser um HTML- / XHTML- / HTML5-Ausgabe.
- Erweitern Sie die PUG-Grammatik um css-Klassen, IDs und Attribute

```
div#my-id.my-class
    a(href="www.hs-aalen.de",target="hsaa")
```

10

Generator-Tools

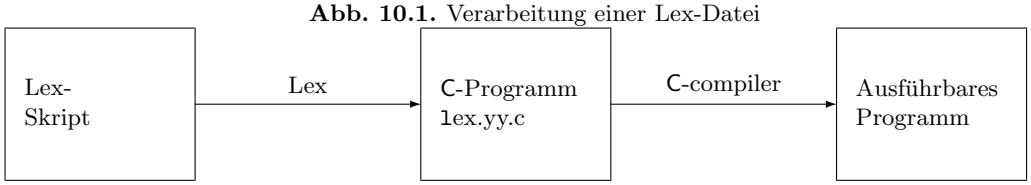
10.1 Einleitung

- YACC Mitte der 1970er Jahre an den Bell Labs durch Stephen C. Johnson
- Lex als Scanner-Generator
  - Erstmalig Mike Lesk in C Mitte 1970er
  - Recodierung durch Eric Schmidt

10.2 Generator-Tools

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

Abb. 10.1. Verarbeitung einer Lex-Datei



10.3 Lex

10.3.1 Allgemeines

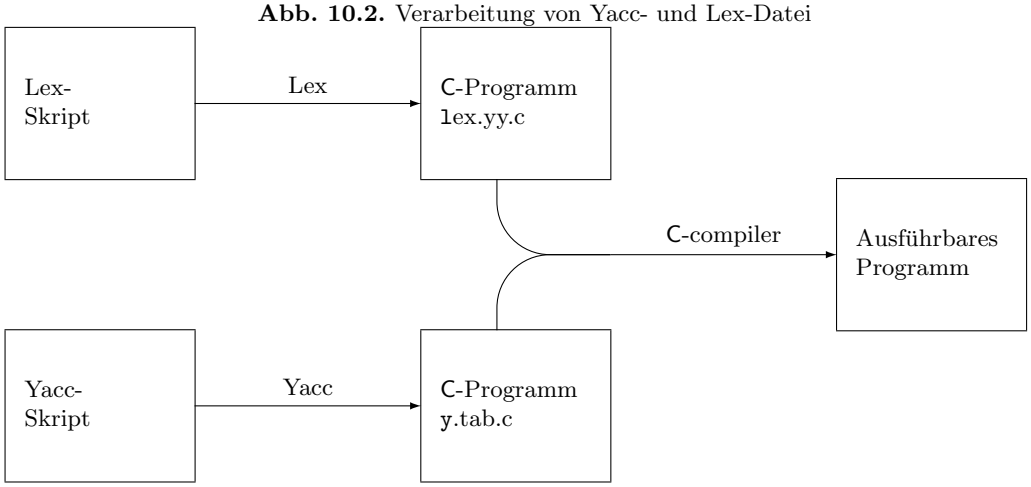
- Entwickelt in den 1970ern von Lesk / Schmidt bei Bell [LS75]
- Häufig Varianten im Einsatz, etwa GNU-Lex oder JLex

10.3.2 Grundaufbau einer Lex-Datei

Listing 10.1. Grundaufbau einer Lex-Datei (lexyacc/grundaufbau.txt)

```
1 <Definitionen, C–Code>
2 %%
```

Abb. 10.2. Verarbeitung von Yacc- und Lex-Datei




```
3 <RegExp-Aktions-Paare>
4 %%
5 <Zusatz-Code>
```

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

10.3.3 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

10.3.4 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.

10.3.5 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- tor	`Tor/ für Deutschland`
`""`	String-Operator	`"+"`
`<>`	Start-Bedingung	`<COMMENT>`

10.3.6 main-Funktion bei Lex-Programmen

- Codiert im dritten Teil
- Separiert mit Include der C-Datei
- Separiert mit getrennter Übersetzung

Listing 10.2. main im Lex-File (lexyacc/int-scanner-1.l)

```
1  %%
2  [0-9]+   printf("Treffer %s in Zeile %d\n", yytext, yylineno);
3  .|\n     ;
4  %%
5  int main() {
6      yylex();
7      return 0;
8  }
9  int yywrap() {
10     return 1;
11 }
```

Listing 10.3. main im Lex-File (lexyacc/int-scanner-2.l)

```
%%
[0-9]+   printf("Treffer %s in Zeile %d\n", yytext, yylineno);
.|\n     ;
%%

int yywrap() {
    return 1;
}
```

Listing 10.4. main im C-File (lexyacc/int-scanner-2.c)

```
#include <stdio.h>
#include "lex.yy.c"
int main() {
    yylex();
    return 0;
}
```

Listing 10.5. main im Lex-File (lexyacc/int-scanner-3/int-scanner-3.l)

```
%%
[0-9]+   printf("Treffer %s in Zeile %d\n", yytext, yylineno);
.|\n     ;
%%

int yywrap() {
    return 1;
}
```

Listing 10.6. main im C-File (lexyacc/int-scanner-3/int-scanner-3.c)

```
#include <stdio.h>
#include "lex.yy.h"
int main() {
    yylex();
    return 0;
}
```

```
>>> lex --header-file=lex.yy.h int-scanner-3.l
>>> clang -c lex.yy.c
>>> clang -c int-scanner-3.c
>>> clang lex.yy.o  int-scanner-3.o

>>> lex --header-file=lex.yy.h int-scanner-3.l
>>> clang -c lex.yy.c
>>> clang -c int-scanner-3.c
>>> clang lex.yy.o  int-scanner-3.o
```

Listing 10.7. Makefile (lexyacc/int-scanner-3/makefile)

```
#
# Makefile zum int-scanner-3
```

```
3  #
4  int-scanner-3: int-scanner-3.o lex.yy.o
5      clang int-scanner-3.o lex.yy.o -o int-scanner-3
6
7  lex.yy.c:    int-scanner-3.l
8      lex --header-file=lex.yy.h int-scanner-3.l
9
10 lex.yy.o:    lex.yy.c
11     clang -c lex.yy.c
12
13 int-scanner-3.o:   int-scanner-3.c
14     clang -c int-scanner-3.c
15
16 clean:
17     rm lex.yy.* *.o
```

10.3.7 Lex-Variablen

`yytext` Gescannter Text

`yyleng` Länge des gescannten Textes

`yylineno` Länge des gescannten Textes

`yyin` Eingabedatei

`yyout` Ausgabedatei

10.3.8 Lex-Funktionen und -Makros

`yywrap` Eingabeende-Funktion

`input` Input-Funktion

`unput` unput-Funktion

`yyles` unput-Funktion

`yymode` Weiterlese-Funktion

ESC-Sequenzen

10.3.9 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: int-Zähler:

Listing 10.8. int-Zähler (lexyacc/int-zaehler.l)

```
1  %{
```

```
2       int anz = 0;
3   %}
4   %%
5   [0-9]+   anz++;
6   .|\n     ;
7   %%
8   int main() {
9       yylex();
10      printf("Anz = %d\n", anz);
11  }
12  int yywrap() {
13      return 1;
14  }
```

```
> lex.yy.c a.out
ls: a.out: No such file or directory
ls: lex.yy.c: No such file or directory
> lex int-zaehler.l
> ll lex.yy.c a.out
ls: a.out: No such file or directory
-rw-r--r--  1 wbantel  staff  43953 20 Okt 07:52 lex.yy.c
> clang lex.yy.c
> ll lex.yy.c a.out
-rwxr-xr-x  1 wbantel  staff  19828 20 Okt 07:53 a.out
-rw-r--r--  1 wbantel  staff  43953 20 Okt 07:52 lex.yy.c
>echo "123 test321"|./a.out
Anz = 2
>
```

Beispiel: Inline-Kommentar in C:

Listing 10.9. 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 10.10. PL/0-Zuweisungen (lexyacc/pl0-var-assignment.l)

```
%{
/* Sucht Zuweisungen an Variablen in PL/0-Programmen */
int lnr = 1, nr = 0;
%}
ID      [A-Za-z_][A-Za-z_0-9]*
WHT     [ \t\n\f]
%%
{ID}/{WHT}*:=   printf("Zeile %d: %s\n", lnr, yytext);
:=      nr++;
\n      lnr++;
.       ;
%%
int yywrap() {
    printf("%d Zuweisungen gefunden!\n", nr);
    return 1;
}
```

10.3.10 Beispiele

UPN-Interpreter

Listing 10.11. UPN-Interpreter (Lex-File) (lexyacc/upninterpreter/upninterpreter.l)

```
%{ // UPN-Interpreter ohne Fehlerverarbeitung
#include "upninterpreter.h"
    int op1, op2;
%}
%%
[0-9]+  push(atoi(yytext));
"+"     op1 = pop(), op2 = pop(), push(op2 + op1);
"*"     op1 = pop(), op2 = pop(), push(op2 * op1);
"-"     op1 = pop(), op2 = pop(), push(op2 - op1);
"/"     op1 = pop(), op2 = pop(), push(op2 / op1);
C|c     push(-pop());
" "|\n  ;
.       printf("Error");
%%
int yywrap() {
    return 1;
}
```

Listing 10.12. UPN-Interpreter (Hauptprogramm) (lexyacc/upninterpreter/upninterpreter.c)

```
#include <stdio.h>
struct{int h; int s[10];} stack = {0};
#include "upninterpreter.h"
#include "lex.yy.h"

int main() {
    yylex();
    printf("Ergebnis: %d\n", pop());
```

```
}

void push(int z) {
    stack.s[stack.h++] = z;
}

int pop() {
    return stack.s[--stack.h];
}
```

Listing 10.13. UPN-Interpreter (Hauptprogramm) (lexyacc/upninterpreter/upninterpreter.h)

```
void push(int z);
int pop();
```

Listing 10.14. UPN-Interpreter (Hauptprogramm) (lexyacc/upninterpreter/makefile)

```
#
# Makefile zum int-scanner-3
#
upninterpreter:   lex.yy.o upninterpreter.o
   clang upninterpreter.o lex.yy.o -o upninterpreter

lex.yy.c:   upninterpreter.l
   lex --header-file=lex.yy.h upninterpreter.l

lex.yy.o:   lex.yy.c
   clang -c lex.yy.c

upninterpreter.o: upninterpreter.c
   clang -c upninterpreter.c

clean:
   rm lex.yy.* *.o upninterpreter
```

Ein Term-Scanner

Listing 10.15. Term-Scanner mit Lex (lexyacc/term-scanner.l)

```
%{
#include <string.h>
#include "term-scanner.h"
%}
%%
"+"                 return t_plus;
"-"                 return t_minus;
"*"                 return t_mal;
"/"                 return t_div;
"("                 return t_kla_auf;
")"                 return t_kla_zu;
[0-9]+("."[0-9]+)?  return t_zahl;
[ \t\n]               /* do nothing */;
";"                 return t_semikolon;
```

```
15 .                    return t_fehler;
16 %%
17 int yywrap() { return 1; }
18
19
20
21
22
23 int term_scanner(char *input,char *text) {
24 // Adaptiert Schnittstelle für Automaten-Beispiele
25     int rc;
26     if (input != NULL) { // Reset
27         yy_scan_string(input);
28         return 0;
29     }
30     rc = yylex();
31     strcpy(text,yytext);
32     return rc;
33 }
34
35
36
37 // int main() {
38 //     int token;
39 //
40 //     while ((token = yylex()) != 0)
41 //       printf("%d '%s'\n", token, yytext);
42 //     return 0;
43 // }
```

10.4 Yacc

10.4.1 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

10.4.2 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.

10.4.3 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, welches im Parser zu verarbeiten ist. Da oft verschiedene Datentypen über Schnittstellen übergeben 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
```

- Wenn der Scanner zu einem Token zusätzliche Information liefern muss so kann in Kombination mit der union-Direktive ein Datentyp angegeben werden. Das Ergebnis wird in der entsprechenden union-Komponente der globalen Variablen yylval erwartet. Ausserdem können Regeln mit Ergebnistypen festgelegt werden.

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

10.4.4 Yacc-Grammatikteil

Im Grammatikteil kann eine LR-Grammatik in einer modifizierten BNF angegeben werden:

- Das Produktionszeichen ist `:`.
- Regeln dürfen / sollen umgebrochen werden
- Regeln werden mit einem `;` abgeschlossen

```
%token  ZAHL KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
%%
```

```
expression: term
          | expression PLUS term
          ;
term:       factor
          | term MAL factor
          ;
factor:     ZAHL
          | KLA_AUF expression KLA_ZU
          ;
%%
```

10.4.5 Zusammenspiel von Lex und YACC

- YACC erwartet einen Scanner mit der Schnittstelle

  ```
  int yylex();
  ```

- Zusätzlich existiert eine globale Variable yylval, die eine union gemäß der `%union`-Direktive ist.
- Das Ende-Token hat den Wert 0.

10.4.6 Yacc-Variablen

yylval

10.4.7 Code in YACC

Siehe Listing 10.17

10.4.8 Code in YACC

Siehe Listing **??**

10.5 Konflikte in Yacc-Grammatiken

shift-reduce-Konflikte

reduce-reduce-Konflikte

10.5.1 Beispiele

Ein Term-Parser

Listing 10.16. Ein Term-Parser (lexyacc/term-parser.y)

```
1 %{
2  int yyerror(char *s);
3  int yylex();
4 %}
```

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

Ein Term-Compiler

Listing 10.17. Ein Term-Parser (lexyacc/term-compiler.y)

```
1  %{ /* Compiler fuer arithmetische Ausdruecke: Term -> UPN */
2  #include <stdio.h>
3  int yylex(); int yyerror(char *s);
4  %}
5  %union {char t[100];}
6  %token       KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
7  %token<t>    ZAHL
8  %%
9  expression: term
10           | expression PLUS term          {printf("+\n");}
11           | expression MINUS term         {printf("-\n");}
12           ;
13 term:       factor
14           | term MAL factor               {printf("*\n");}
```

```
15              | term DIV factor                { printf("/\n");}
16              ;
17  factor:       ZAHL                           { printf("%s\n", $1);}
18              | KLA_AUF expression KLA_ZU
19              | PLUS factor
20              | MINUS factor                   { printf("CHS\n");}
21              ;
22  %%
23  #include "lex.yy.c"
24
25  int yyerror(char *s) {
26      printf("%s\n", s);
27      return 0;
28  }
29
30  int main() {
31      int rc;
32      rc = yyparse();
33      if (rc == 0)
34          printf("Syntax OK\n");
35      else
36          printf("Syntax nicht OK\n");
37      return rc;
38  }
```

Ein Term-Rechner

Listing 10.18. Ein Term-Parser (lexyacc/term-rechner.y)

```
1   %{ /* Interpreter fuer arithmetische Ausdruecke */
2   #include <stdio.h>
3   #include <stdlib.h>
4   int yylex();
5   int yyerror(char *s);
6   double erg;
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  expression: term                           {$$ = erg = $1;}
14              | expression PLUS term         {$$ = erg = $1 + $3;}
15              | expression MINUS term        {$$ = erg = $1 - $3;}
16              ;
17  term:         factor                       {$$ = $1;}
18              | term MAL factor              {$$ = $1 * $3;}
19              | term DIV factor              {$$ = $1 / $3;}
20              ;
21  factor:       ZAHL                         {$$ = atof($1);}
22              | KLA_AUF expression KLA_ZU    {$$ =  $2;}
23              | PLUS factor                  {$$ =  $2;}
24              | MINUS factor                 {$$ = -$2;}
```

```
25              ;
26 %%
27 #include "lex.yy.c"
28
29 int yyerror(char *s) {
30     printf("%s\n", s);
31     return 0;
32 }
33
34 int main() {
35     int rc;
36     rc = yyparse();
37     if (rc == 0)
38         printf("erg = %lf\n", erg);
39     else
40         printf("Syntax nicht OK\n");
41     return rc;
42 }
```

Ein Term-AST-Generator

Listing 10.19. Ein Term-Parser (lexyacc/term-ast-gen.y)

```
1  %{
2  /*********************************************************
3  Konstruiert AST für arithmetische Ausdrücke (Operator-Baum)
4  *********************************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include "term-ast.h"
8  #include "term-ast.c"
9  struct node * tree;
10 %}
11 %union {char t[100]; struct node *p;}
12 %token      KLA_AUF KLA_ZU PLUS MINUS MAL DIV END FEHLER
13 %token<t>   ZAHL
14 %type<p> factor term expression
15 %%
16 input:      expression              {tree = $1;}
17           ;
18
19 expression: term                {$$ = $1;}
20           | expression PLUS term        {$$ = new_node("+", $1, $3);}
21           | expression MINUS term    {$$ = new_node("-", $1, $3);}
22           ;
23
24 term:       factor                {$$ = $1;}
25           | term MAL factor             {$$ = new_node("*", $1, $3);}
26           | term DIV factor             {$$ = new_node("/", $1, $3);}
27           ;
28
29 factor:     ZAHL                {$$ = new_node($1, NULL, NULL);}
30           | KLA_AUF expression KLA_ZU  {$$ = $2;}
```

```
31              | PLUS factor             {$$ = $2;}
32              | MINUS factor            {$$ = new_node("CHS", $2, NULL);}
33              ;
34 %%
35 #include "lex.yy.c"
36
37 int yyerror(char *s) {
38     printf("%s\n", s);
39
40 }
41
42 int main() {
43     int rc;
44     rc = yyparse();
45     if (rc == 0) {
46         printf("Syntax OK\n");
47    printf("Baum:\n"); tree_output(tree,0);
48    printf("UPN:\n"); tree_code(tree);
49    printf("Ergebnis: %lf\n", tree_result(tree));
50    }
51     else
52         printf("Syntax nicht OK\n");
53     tree_free(tree);
54     return rc;
55 }
```

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

11

Semantische Analyse

12

Symboltabellen

- Problem: Verwendung von Bezeichnern nicht über Grammatik kontrollierbar.
- Programmiersprachen sind eigentlich keine Typ-2-Sprachen!
- Beispiel: ! a.
- Typ-2-Grammatik, die Bezeichnernamen beinhaltet, wäre zu komplex
- Abhilfe: Symboltabelle und „So-tun-als-ob Typ-2“
  - Ersetzt alle Bezeichner durch int-Werte
  - Für jeden Bezeichner ist ein Tripel zu verwalten:
    - Level-Delta im statischen Code
    - (Offset, laufende Nr im jeweiligen Level)
    - Typ des Bezeichners
    - Weitere typspezifische info

Ersetzen in Programm 16.1 (Seite 194) alle Bezeichner durch Typ/Level/Offset

Wichtig bei der Diskussion der Symboltabelle sind Begriffe wie

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

12.1 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

12.2 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. 12.1. Aufbau einer Symboltabelle

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

In Zeile 2 von Listing 16.1 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).

12.3 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

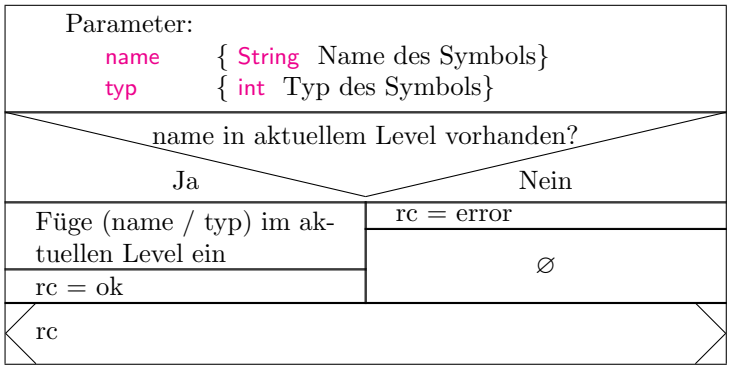
12.4 Namens-Suche

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.

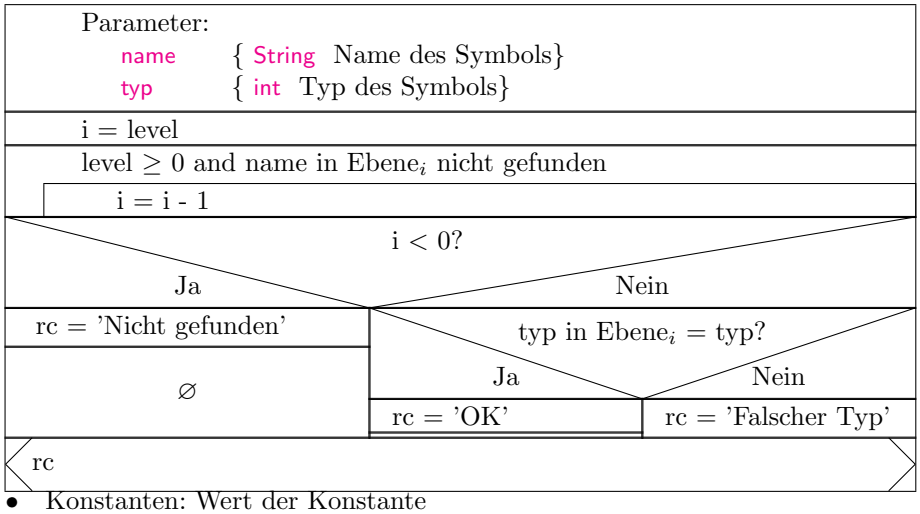
Möglichkeiten zur Beschleunigung wären

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

int insert(name,typ)



int insert(name,typ)



- Konstanten: Wert der Konstante

```
const a = 1;
```

- Variablen: Laufende Nummer der Variablen in Prozedur
- Prozeduren: Global laufende Nummer

12.5 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 12.1. Symboltabelle - Klasse symtabentry (pl-0/pl0-symtab.hpp)

```
1  enum {st_var = 1, st_const = 2, st_proc = 4};
2
3  class symtab_entry{
4     public:
5        symtab_entry(const int = -1, const int = -1);
6        int type;    // Art des Eintrags
7        int val; // Wert (z.B. procnr)
8  };
```

Listing 12.2. Symboltabelle - Klasse symtab (pl-0/pl0-symtab.hpp)

```
1     public:
2        symtab ();
3        void level_up();
4        void level_down();
5        int insert(const string name, const int typ,
6           const int value);
7        int lookup(string name, int type, int & l,
8           int & value);
9        void print();
10    private:
11       vector< map<string,symtab_entry> > content;
12 };
```

Listing 12.3. Symboltabelle - Konstruktoren (pl-0/pl0-symtab.cpp)

```
1  symtab_entry::symtab_entry(const int type_,
2     const int val_) {
3     type = type_, val = val_;
4  }
5
6  symtab::symtab () {
7  }
8
9
10 void symtab::level_up() {
```

Listing 12.4. Symboltabelle - Level-Methoden (pl-0/pl0-symtab.cpp)

```
1     cout << "Symtab-level-up\n";
2     map<string,symtab_entry> tmp;
3     content.push_back(tmp);
4  }
5
6  void symtab::level_down() {
7     int level = content.size() - 1;
8     cout << "Symtab-level-down " << level << ":" << content.size() << "\n";
9     content[level].clear();
10    content.pop_back();
11 }
```

Listing 12.5. Symboltabelle - insert-Methode (pl-0/pl0-symtab.cpp)

```
1  int symtab::insert(const string name,
2     const int typ, const int value){
3     int level = content.size() - 1, r = -1;
4
5     if (content[level].find(name) == content[level].end()) {
6        content[level][name] = symtab_entry(typ, value);
7        cout << "Symtab-insert '" << name << "': " <<
8           content[level][name].type <<
9            "value: " << value << endl;
10       r = 0;
11    }
12    return r;
13 }
```

Listing 12.6. Symboltabelle - lookup-Methode (pl-0/pl0-symtab.cpp)

```
1  int symtab::lookup(string name, int type, int & l,
2     int & value) {
3     int level = content.size() - 1;
4     int i = level+1, rc = 0;
5     cout << "Symtab-lookup '" << name <<
6        "' (Typ " << type << ")" << endl;
7     l = -1;
8     while (--i >= 0 && content[i].find(name) == content[i].end());
9     if (i >= 0)
10       if (content[i][name].type & type)
11          l = level - i,
12             value = content[i][name].val;
13       else
14          rc = -1; // Falscher Typ
15    else
16       rc = -2; // Nicht gefunden
17    return rc;
18 }
```

Listing 12.7. Symboltabelle - print-Methode (pl-0/pl0-symtab.cpp)

```
1  void symtab::print(){
2     int level = content.size() - 1;
3     map<string,symtab_entry>::iterator pos;
4     cout << "Akt. Level: "<<  level << endl;
```

```
5     for (int i = 0; i <= level; i++) {
6        cout << "Level " << i << ": Hoehe " <<
7           content[i].size() << endl;
8        pos = content[i].begin();
9        for(pos = content[i].begin();
10          pos != content[i].end();
11          ++pos){
12          cout << "Key: " << (*pos).first <<
13             (*pos).second.type << endl;
14       }
15    }
16 }
```

```
program:          block t_punkt { ast = $1;}
                ;

block:            {st.level_up();}
                  const_decl var_decl proc_list statement
                  {st.level_down();}
                ;

constdec:    t_id t_eq t_number  {
          st.insert($$->name, st_const, $3);
       }
       ;
var_id:   t_ident  {
          st.insert($$->name, st_var, local_var_nr++);
       }
       ;

procedure:  t_proc
          t_ident   {
             st.insert($2, st_proc, ++global_proc_nr);
          }
          t_semik
          block     {...}
       ;

assignement:     t_ident t_assign expression
                 {int stl, sto;
                  st.lookup($1, st_var, &stl, &varnr);
                 }
                 ;
proc_call:       t_call t_ident
                 {int stl, sto;
                  st.lookup($2, st_proc, &stl, &procnr);
                 }
                 ;
read:            t_read t_ident
                 {int stl, sto;
                  st.lookup($2, st_var, &stl, &varnr);
                 }
                 ;
```

```
factor:         t_ident
                {int stl, sto;
                 st.lookup($1, st_var|st_const, &stl, &varnr);
                }
              | t_number
              | t_bra_o expression t_bra_c
              | t_plus factor
              | t_minus factor
              ;
```

13

Zwischencode

- Begründung
- Erzeugung und Arten
- Vorstellung Binärbaum für Ausdrücke mit Erzeugung und Backends
- Erzeugung im RD
- Erzeugung im YACC
- Vorstellung Syntaxbaum mit Erzeugung und Backends
- Syntaxbaum: RD und YACC
- Syntaxbaum: Tabellengesteuertes LL(1)-Parsen
- ST 2 AST für RD
- ST 2 AST für YACC
- ST 2 AST für Tab-LL(1)
- ASTs für Programmiersprachen
- ST für PL0 (alt)
- ST für PL0 (neu) aus RD
- AST für PL0

Betrachten wir wieder einmal Abbildung 1.1 auf Seite 17. 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 13.2 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 **??** dargestellt.
- Änderungen in der Vorgehensweise des Compilers ziehen viele Veränderungen 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 34) oder Grammatik **??** (Seite **??**).

Abb. 13.1. Compiler-Collection ohne Schnittstelle

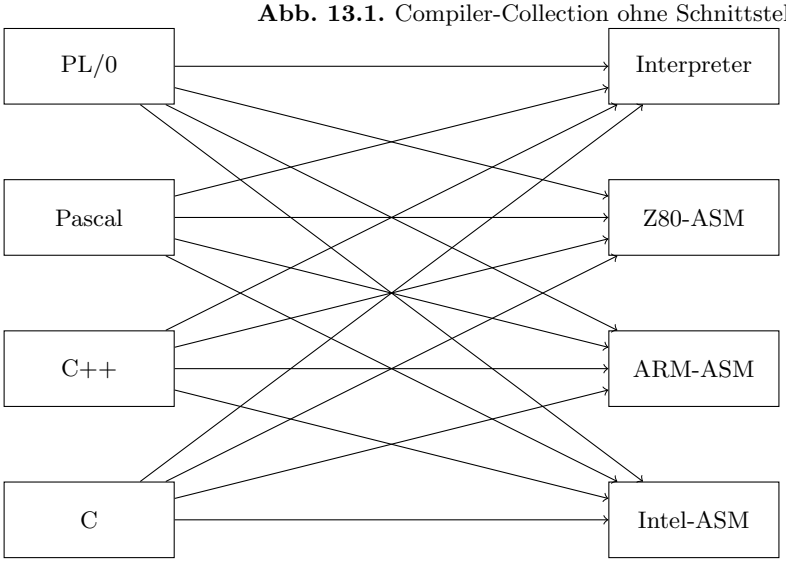
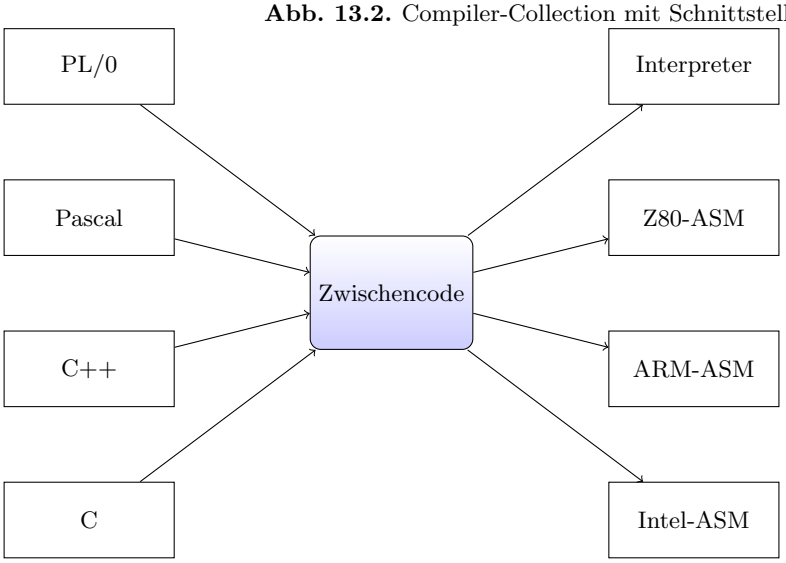


Abb. 13.2. Compiler-Collection mit Schnittstelle



In der Praxis ist Zwischencode häufig

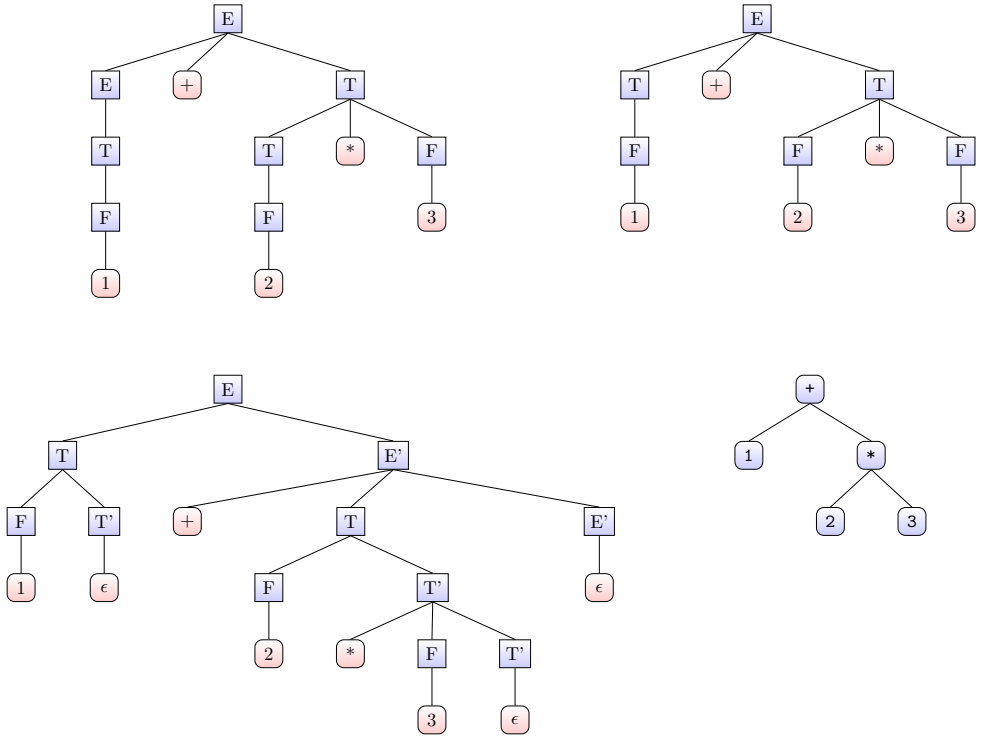
- ein Zielsystem-unabhängiger Pseudo-Assembler-Code
  - für Register-Stack-Maschinen
  - für Zwei-Adress-Maschinen
  - für Drei-Adress-Maschinen
- ein Syntaxbaum

- ein abstrakter 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“.

Zwischencode ist korrekt!!!

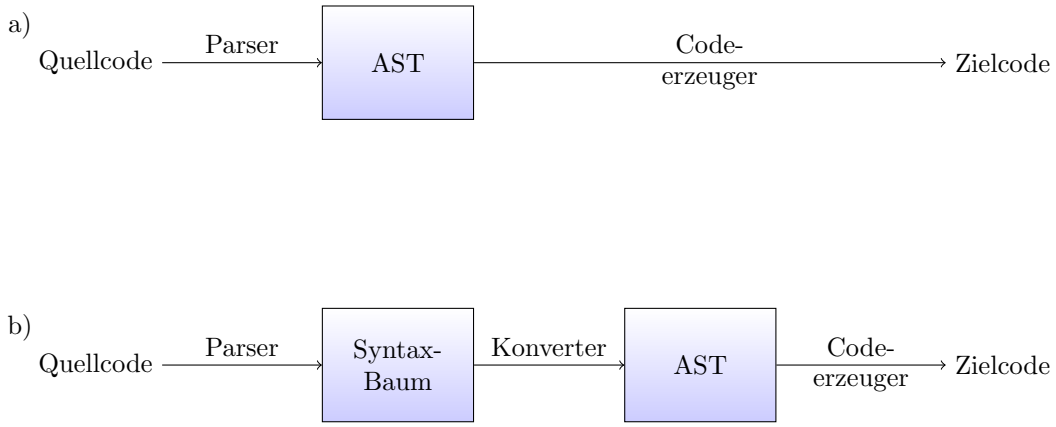
- Bei Dateien keine syntaktischen Fehler
- Bei dynamischen Datenstrukturen keine falsch gesetzten Zeiger
- ...



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

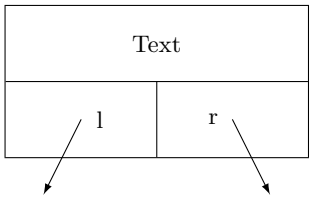
13.1 Einführung

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 **??** wird klar, dass diese Schnittstelle sowohl C- als auch Fortran-, Pascal- und weitere Programme abbilden können muss.



13.2 Ein Binärbaum als AST für Ausdrücke

Abb. 13.3. Datenstruktur für binären Operator-Baum




```
typedef struct s_ast_node ast_node;
struct s_ast_node {
        char * txt;//txt[64];
        ast_node * l;
        ast_node * r;
};

ast_node * ast_new_node(char * txt, ast_node * l, ast_node * r);
void ast_tikz(ast_node * p, int n, int path);
void ast_tikz_simple(ast_node * p, int level);
void ast_tikz_path(ast_node * p, int level);
void ast_explorer(ast_node * p, int level);
void ast_2_aassembler(ast_node * p);
void ast_deltree(ast_node * p);
int ast_eval(ast_node * p);

int ast_1_address(ast_node * p, int level);
int ast_2_address(ast_node * p, int level);
int ast_3_address(ast_node * p, int level);
```

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.

Ein Compiler, der in einem ersten Schritt den Operator-Baum erstellt und diesen anschließend mehrfach verarbeiten lässt findet sich in Listing 13.1.

- 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.

13.3 Syntaxbaum-Generierung

13.3.1 Recursive-Descent

13.3.2 YACC

13.4 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 **??** dargestellt.

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.

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

13.4.1 Erzeugung des AST

Grammatik Γ_{31}: Term als Attribut-Grammatik

Produktion	Regel	Bedingung
$\mathbf{E \rightarrow T_1}$	$\mathbf{tree(E) = T_1}$	
$\mathbf{E \rightarrow E_1}$ **'+'** $\mathbf{T_3}$	$\mathbf{tree(E) = tree('+', E_1, T_3)}$	
$\mathbf{T \rightarrow F}$	$\mathbf{tree(T) = F_1}$	
$\mathbf{T \rightarrow T}$ **'*'** $\mathbf{F}$	$\mathbf{tree(T) = tree('+', T_1, F_3)}$	
$\mathbf{F \rightarrow}$ **'zahl'**	$\mathbf{tree(F) =}$ **'ZAHL'**	
$\mathbf{F \rightarrow}$ **'('** $\mathbf{E}$ **')'**	$\mathbf{tree(F) = tree(\ E_2)}$	
$\mathbf{F \rightarrow}$ **'-'** $\mathbf{F}$	$\mathbf{tree(F) = tree('CHS', F_2)}$	

```
int main(void) {
  error = 0, token = yylex();
  |ast_node * root = |f_expression();
  if (!error && token == 0)
    ast_tikz(root, 0, 1);
  else
    printf("Fehler %d\n", error);
  return 0;
}

ast_node * f_expression(void) {
  |ast_node *| p = f_term();
  while(token == t_plus) {
    |p = new_ast_node(yytext, p, NULL);|
    token = yylex();
    |p->r =| f_term();
  }
  return p;
}

ast_node * f_term(void) {
  |ast_node * p = |f_factor();
  while(token == t_mal) {
    |p = new_ast_node(yytext, p, NULL);|
    token = yylex();
    |p->r =| f_factor();
  }
  return p;
}

ast_node * f_factor (void) {
  |ast_node * p;|
  if (token == t_minus) {
    token = yylex();
    |p = ast_new_node("-", f_factor(), NULL);|

  }
  else if (token == t_zahl) {
    |p = ast_new_node(yytext, NULL, NULL);|
    token = yylex();
  }
  else if (token == t_kla_auf) {
    token = yylex();
    |p = f_expression();|
    if (token == t_kla_zu)
      token = yylex();
    else
      error = 1;
  }
  else error=2;
  return p;
}
```

Listing 13.1. AST-Erzeugung mit YACC-Parser (ast/term-ast-gen.y)

```
1 %union {char t[100]; ast p;}
2 %token  t_plus t_minus t_mal t_div t_kla_auf t_kla_zu t_fehler
3 %token<t>   t_zahl
4 %type<p> factor term expression
```

```
5  %%
6  input:      expression              {tree = $1;}
7            ;
8  expression: term                 {$$ = $1;}
9            | expression t_plus term  {$$ = new_node("+", $1, $3);}
10           | expression t_minus term {$$ = new_node("-", $1, $3);}
11           ;
12 term:       factor                {$$ = $1;}
13           | term t_mal factor      {$$ = new_node("*", $1, $3);}
14           | term t_div factor        {$$ = new_node("/", $1, $3);}
15           ;
16 factor:     t_zahl                {$$ = new_node($1, NULL, NULL);}
17           | t_kla_auf expression t_kla_zu {$$ = $2;}
18           | t_plus factor         {$$ = $2;}
19           | t_minus factor        {$$ = new_node("CHS", $2, NULL);}
20           ;
21 %%
```

???

Warum: L-Attributierte Grammatik!!!
Abhilfe: Echter Syntaxbaum und anschließende Konvertierung

Listing 13.2. Die Syntaxbaum-Datenstruktur (include/syntaxtree.h)

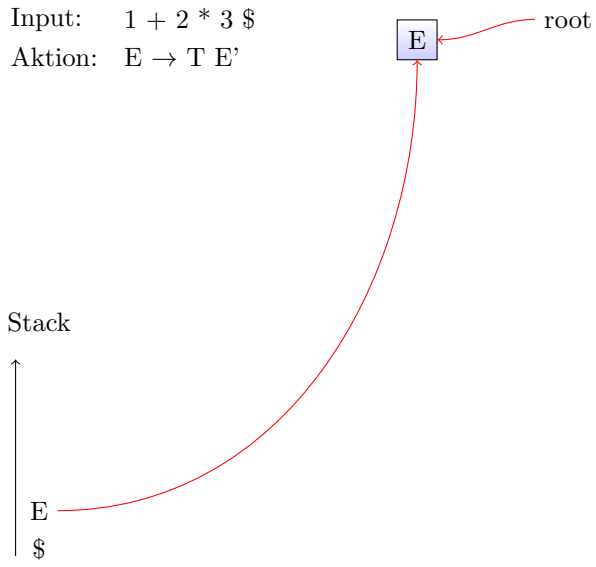
```
1  #pragma once
2  typedef struct s_node node;
3  struct s_node {
4     //char txt[10];
5     char * txt;
6     node * child[30];
7  };
8  node * new_node(char * txt);
9  void tree_tikz(node * p, int n);
10 void tree_explorer(node * p, int level);
```

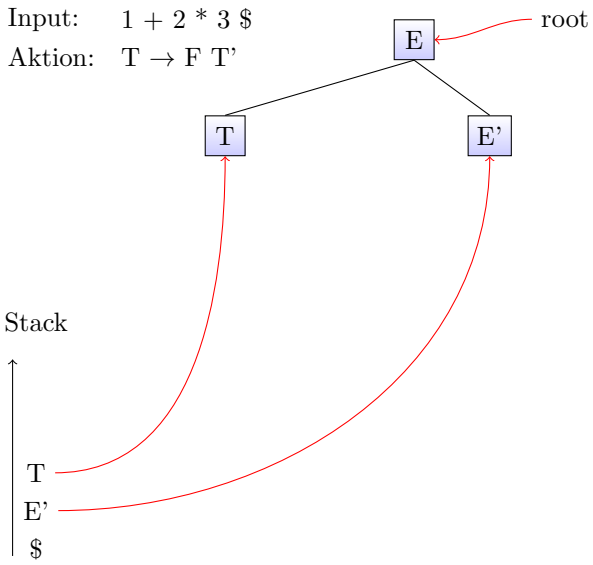
```
int main(void) {
  node * root;
  error = 0, token = yylex();
  |root = f_expression();|
  printf("Ertebnis: %d\n",
    !error && !token);
  return 0;
}

node * f_expression(void) {
  |node * p = new_node("E");|
  int i = 0;
  p->child[0] = f_term();
  while(token == t_plus) {
    |p->child[++i] = new_node(yytext);|
    token = yylex();
    |p->child[++i] = f_term();|
  }
  return p;
}
node * f_term(void) {
  |node * p = new_node("T");|
  int i = 0;
  |p->child[0] = f_factor();|
  while(token == t_mal) {
    |p->child[++i] = new_node(yytext);|
    token = yylex();
    |p->child[++i] = f_factor();|
  }
  return p;
}

node * f_factor (void) {
```

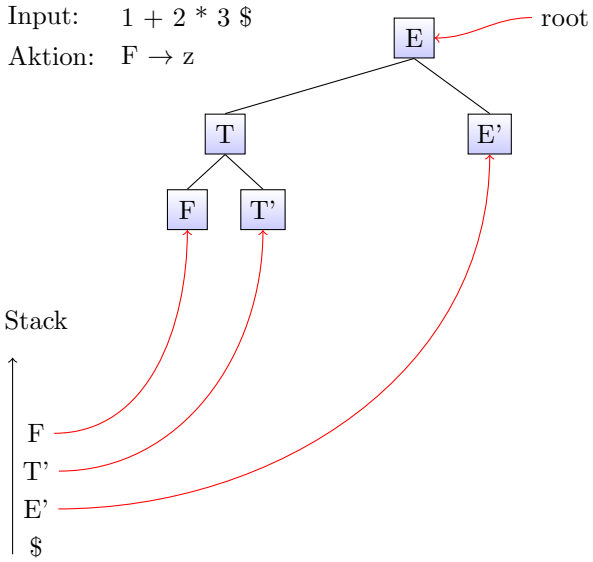
Listing 13.3. ST-Erzeugung mit YACC-Parser (kellerautomaten/yacc-syntaxtree.y)

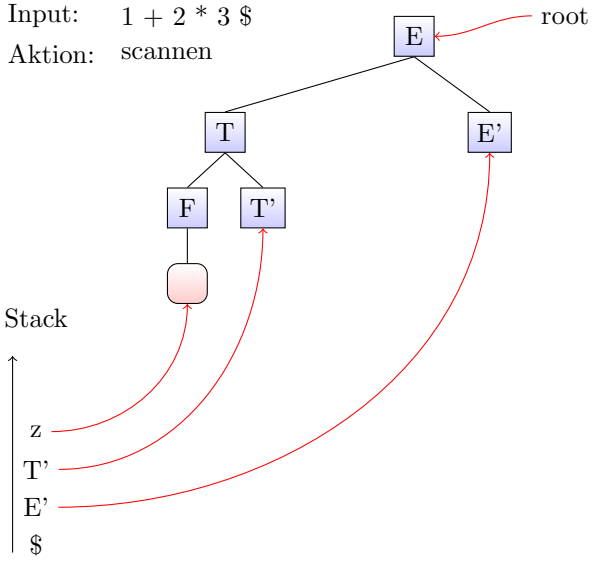
```
1  %%
2  start:    expression {root = $1;}
3  expression: term {$$ = new_node("E"); $$->child[0] = $1;}
4          | expression t_plus term    {
5      $$ = new_node("E"); $$->child[0] = $1;
6      $$->child[1] = new_node("+"); $$->child[2] = $3;}
7              ;
8
9  term: factor {$$ = new_node("T"); $$->child[0] = $1;}
10         | term t_mal factor   {
11     $$ = new_node("T"); $$->child[0] = $1;
12     $$->child[1] = new_node("*"); $$->child[2] = $3;}
13             ;
14
15 factor:   t_zahl {$$ = new_node("F"); $$->child[0] = new_node($1);}
16         | t_kla_auf expression t_kla_zu {
17     $$ = new_node("F"); $$->child[0] = new_node("(");
18     $$->child[1] = $2; $$->child[2] = new_node(")");}
19         | t_minus factor {
20     $$ = new_node("F");
21     $$->child[0] = new_node("-");$$->child[0] = $2;}
22             ;
23 %%
```

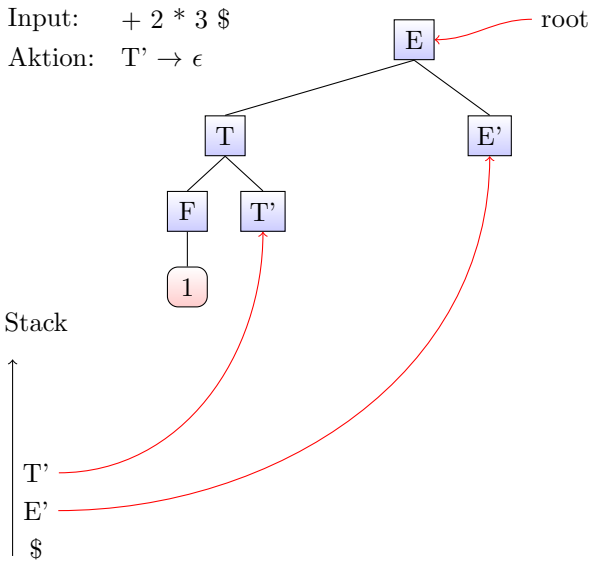
- Auf dem Stack werden Bitupel abgelegt:
  - Symbol
  - Zeiger auf Knoten
- Anfänglich Startsymbol und Root-Knotem

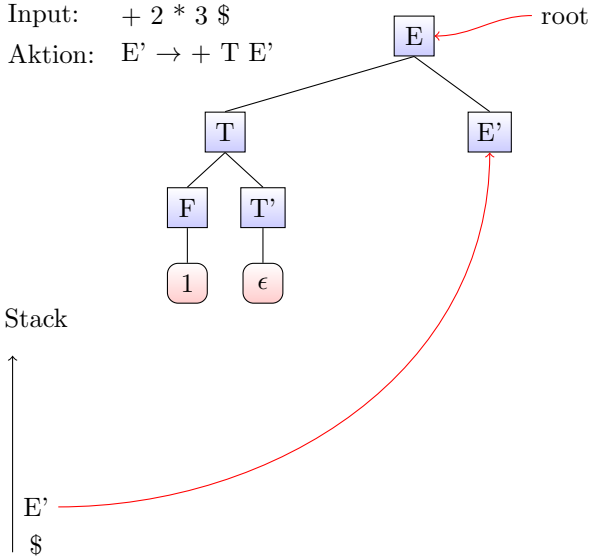


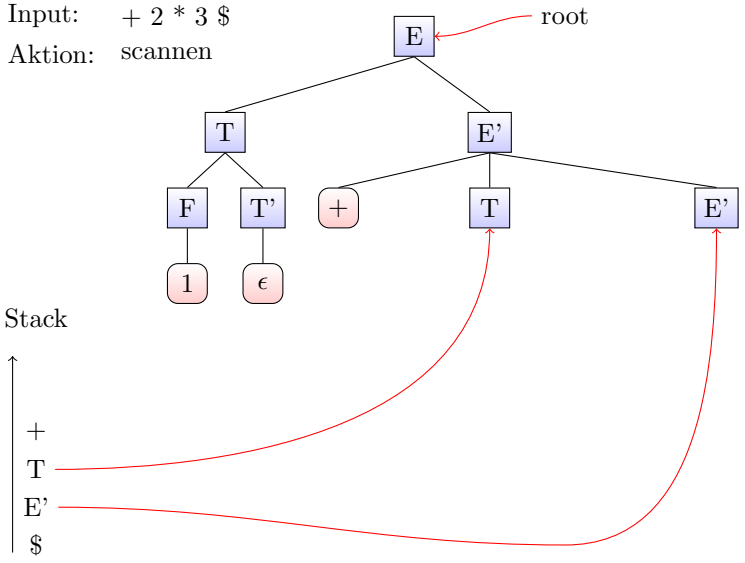

Input: 1 + 2 * 3 $
Aktion: T → F T'
E
root
T
E'
Stack
T
E'
$

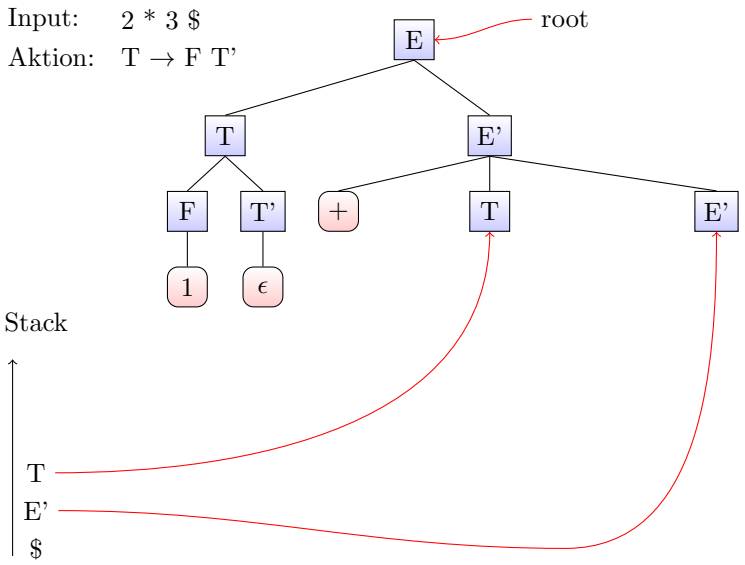

Input: 1 + 2 * 3 $
Aktion: F → z
E
root
T
E'
F
T'
Stack
F
T'
E'
$

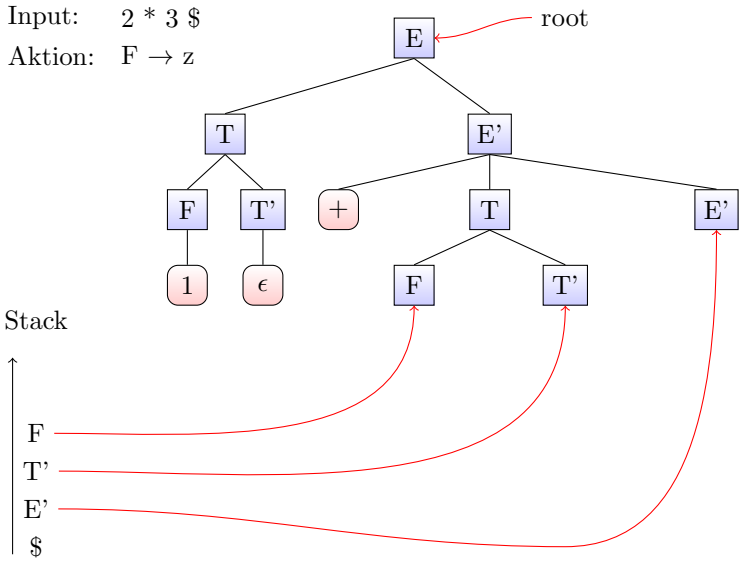

Input: 1 + 2 * 3 $
Aktion: scannen
E
root
T
E'
F
T'
Stack
z
T'
E'
$

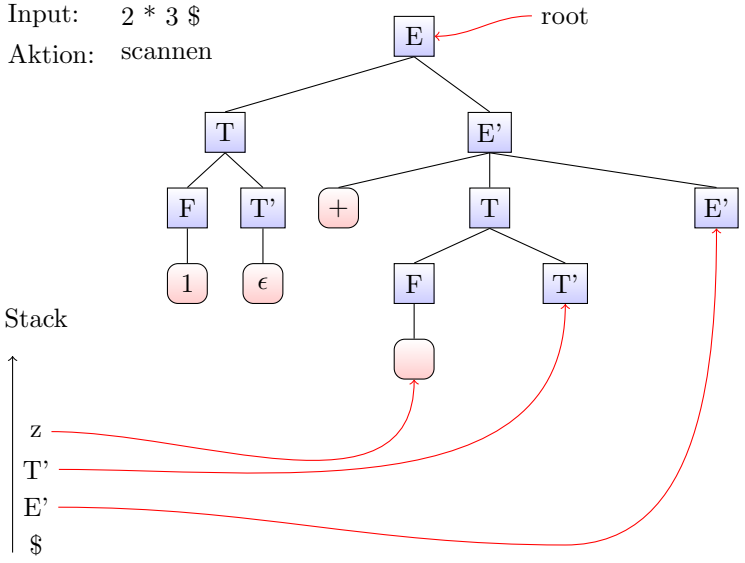

Input: + 2 * 3 $
Aktion: T' → ϵ
E
root
T
E'
F
T'
1
Stack
T'
E'
$

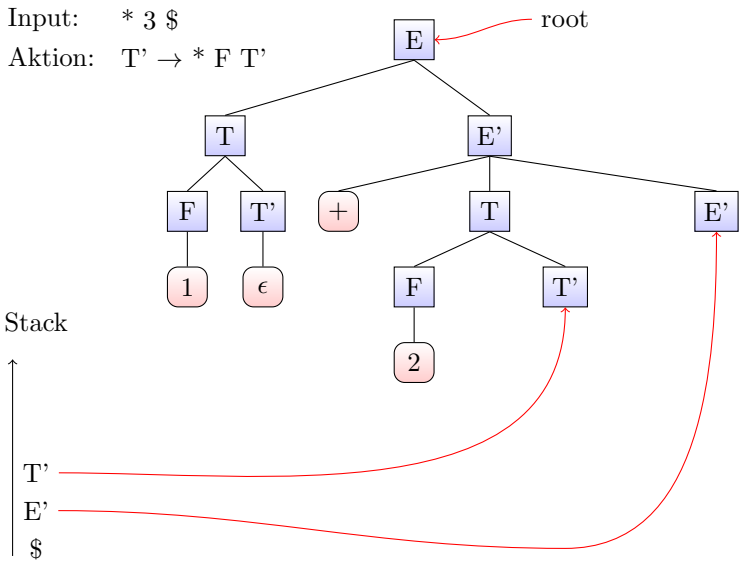

Input: + 2 * 3 $
Aktion: E' → + T E'
E
root
T
E'
F
T'
1
ϵ
Stack
E'
$

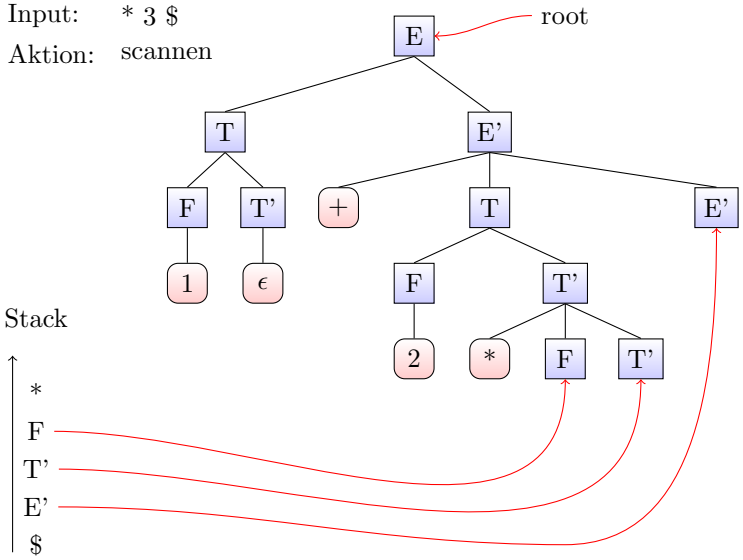

Input: + 2 * 3 $
Aktion: scannen
E
root
T
E'
F
T'
+
T
E'
1
ϵ
Stack
+
T
E'
$

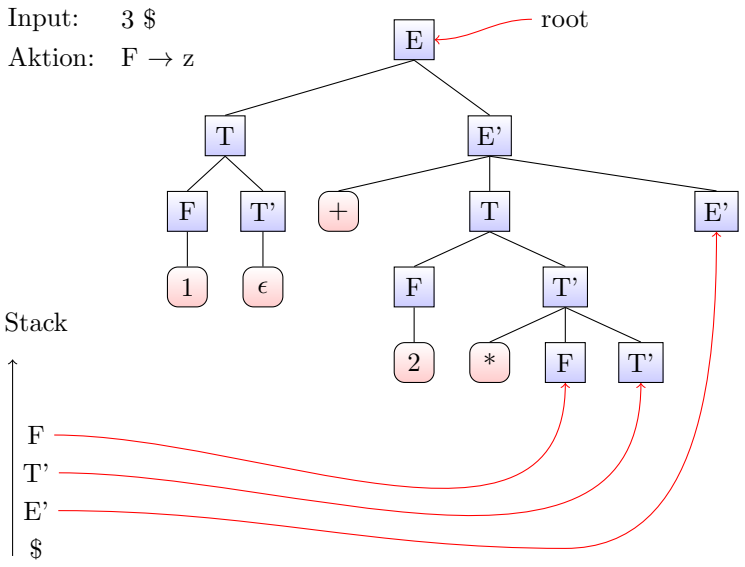

Input: 2 * 3 $
Aktion: T → F T'
E
root
T
E'
F
T'
+
T
E'
1
ϵ
Stack
T
E'
$

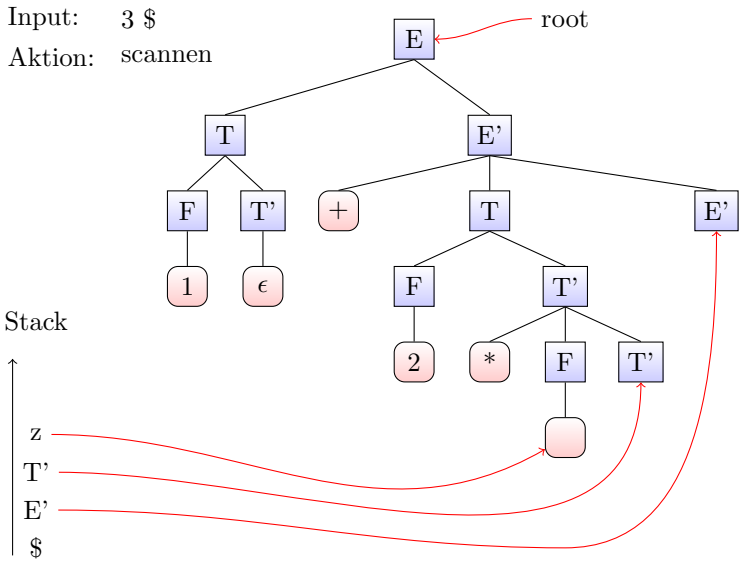

Input: 2 * 3 $
Aktion: F → z
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
Stack
F
T'
E'
$

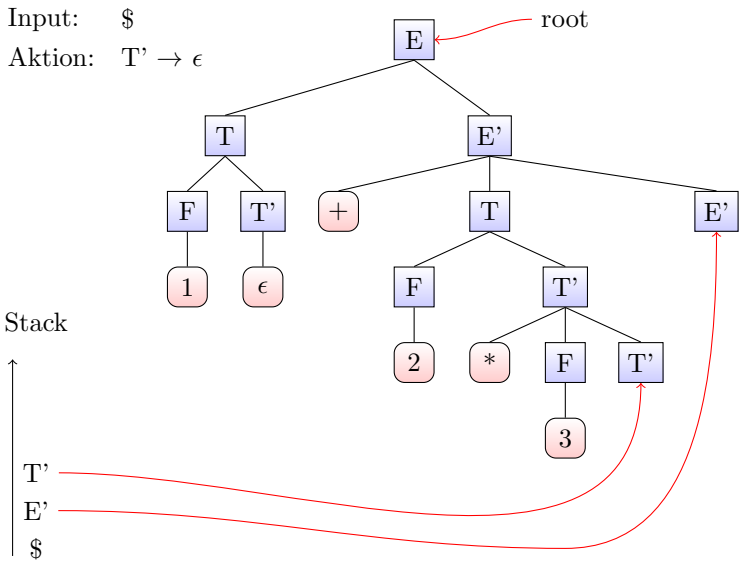

Input: 2 * 3 $
Aktion: scannen
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
Stack
z
T'
E'
$

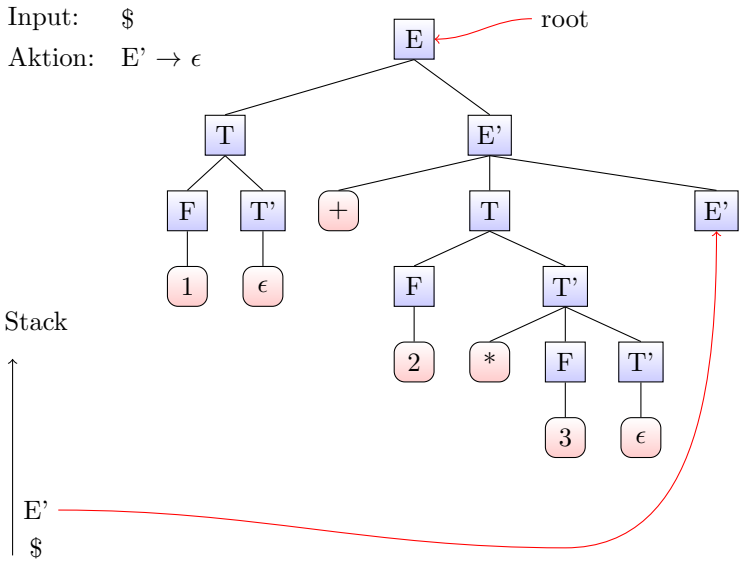

Input: * 3 $
Aktion: T' → * F T'
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
Stack
T'
E'
$

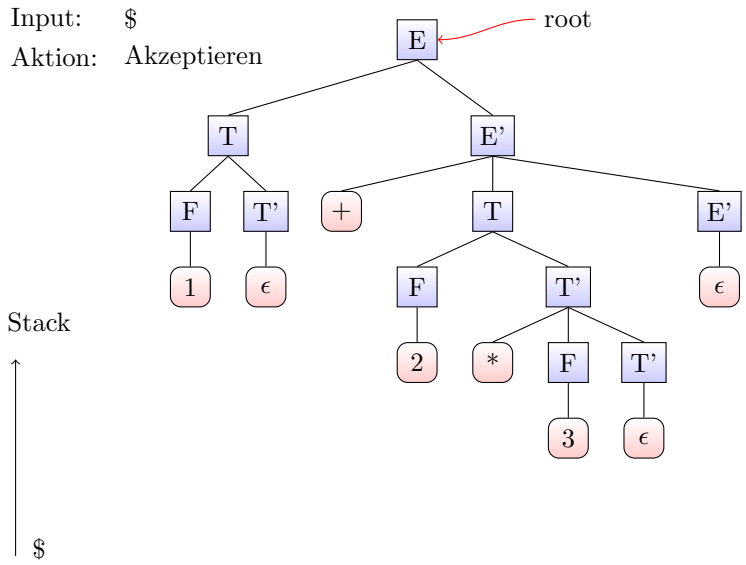

Input: * 3 $
Aktion: scannen
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
*
F
T'
Stack
*
F
T'
E'
$


Input: 3 $
Aktion: F → z
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
*
F
T'
Stack
F
T'
E'
$


Input: 3 $
Aktion: scannen
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
*
F
T'
Stack
z
T'
E'
$


Input: $
Aktion: T' → ϵ
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
*
F
T'
3
Stack
T'
E'
$


Input: $
Aktion: E' → ϵ
E
root
T
E'
F
T'
+
T
E'
1
ϵ
F
T'
2
*
F
T'
3
ϵ
Stack
E'
$



Listing 13.4. ST-Erzeugung mit tabellengesteuertem LL(1)-Parser (kellerautomaten/stackverwaltung/ll1-stack-syntaxtree.c)

```
typedef struct {
    int value;
    node * p;
} stack_element;

int parse_table[5][9] = {
    /*            END  +   -   *   /   (   )  zahl  err */
    /*E  */ {    -1, -1,  0, -1, -1,  0, -1,  0,  -1},
    /*E'*/ {    2,  1, -1, -1, -1, -1,  2, -1,  -1},
    /*T  */ {    -1, -1,  3, -1, -1,  3, -1,  3,  -1},
    /*T' */ {     5,  5, -1,  4, -1, -1,  5, -1,  -1},
    /*F  */ {    -1, -1,  7, -1, -1,  6, -1,  8,  -1},
};
```

Listing 13.5. ST-Erzeugung mit tabellengesteuertem LL(1)-Parser (kellerautomaten/stackverwaltung/ll1-stack-syntaxtree.c)

```
void regel_auf_stack(int r, stack * stack, node * p) {
    int i = -1;
    stack_element e;
    while (i++, g[r][i] >=0) {};
    if (i > 0) {
        for (i--; i >= 0; i--) {
            e.value = (g[r][i]);
            e.p = p->child[i] = new_node(
                (g[r][i] < nt_offset)
                    ? yytext
                    : nttext[g[r][i] - nt_offset]
            );
            stack_push(stack, &e);
        }
    }
    else // Epsilon-Produktion
        p->child[0] = new_node("");
```

```
18 }
19
20 void int_print(int *i) { // fuer stack_print-Funktion
```

- Vorteil: Einunddasselbe Programm für alle Grammatiken!
- Konsequenz: AST-Erzeugung als Syntaxbaum
- Nachteile:
  - Nur linksrekursionsfreie Grammatikdn BNF-Form möglich
  - Syntaxbäume „gewöhnungsbedürftig“
  - Keine S-Attributgrammatiken möglich

```
ast_node * st2ast_E(node * p);
ast_node * st2ast_Es(node * p, ast_node *);
ast_node * st2ast_T(node * p);
ast_node * st2ast_Ts(node * p, ast_node *);
ast_node * st2ast_F(node * p);

ast_node * st2ast_E(node * p) {
    return st2ast_Es(p->child[1], st2ast_T(p->child[0]));
}

ast_node * st2ast_Es(node * p, ast_node * l) {
    return (p->child[1] != NULL) ? st2ast_Es(p->child[2],
        ast_new_node("+", l, st2ast_T(p->child[1]))) : l;
}

ast_node * st2ast_T(node * p) {
    return st2ast_Ts(p->child[1], st2ast_F(p->child[0]));
}

ast_node * st2ast_Ts(node * p, ast_node * l) {
    return (p->child[1] != NULL) ? st2ast_Ts(p->child[2],
        ast_new_node("*", l, st2ast_F(p->child[1]))) : l;
}

ast_node * st2ast_F(node * p) {
    ast_node * rc;
    if (p->child[0]->txt[0] == '-')
        rc = ast_new_node("CHS", st2ast_F(p->child[1]),
            NULL);
    else if (p->child[0]->txt[0] == '(')
        rc = st2ast_E(p->child[1]);
    else
        rc = ast_new_node(p->child[0]->txt, NULL, NULL);
    return rc;
}
```

- Datentypen
- Variablen:
  - Namen?

- Deklarations-Nr.
- Deklarations-Level
- Datentyp

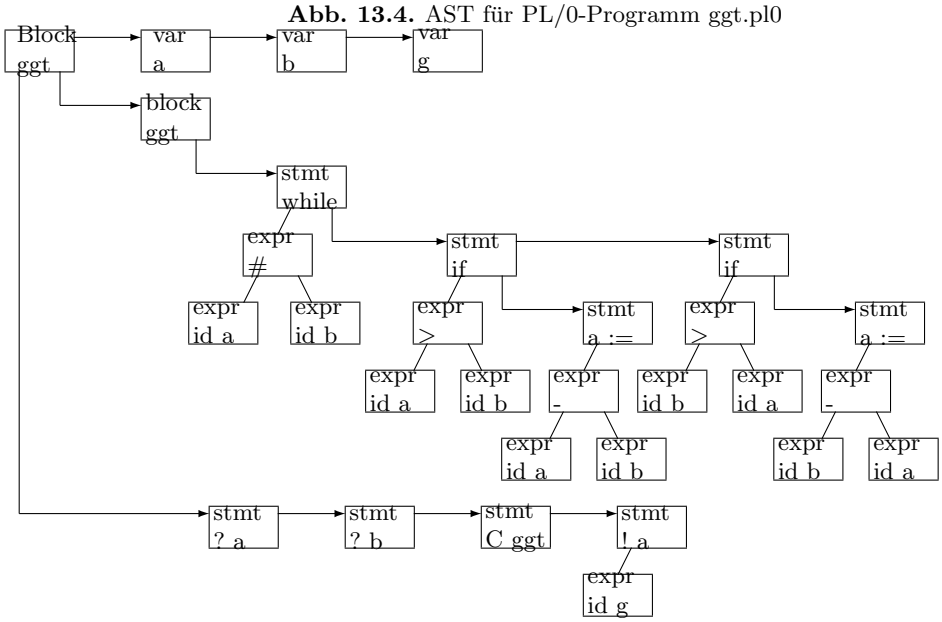
13.5 AST für PL/0

Mehrere Knoten-Arten:

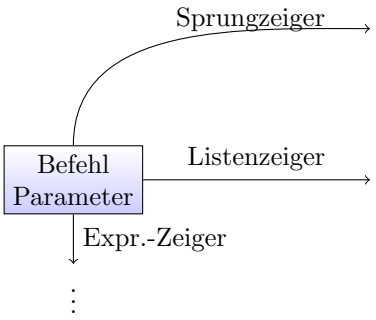
- Block-Knoten
  - Zweiger auf Var-Knoten
- Var-Knoten
  - Zweiger auf Var-Knoten
- statement-Knoten
  - Zweiger auf Statement-Knoten
- Expression-Knoten anaolg zu arithmetischen Termen
  - Zweiger auf Expression-Knoten

Abbildung 13.4 zeigt den Syntaxbaum für das GGT-Programm (Listing 2.2), Seite 20).

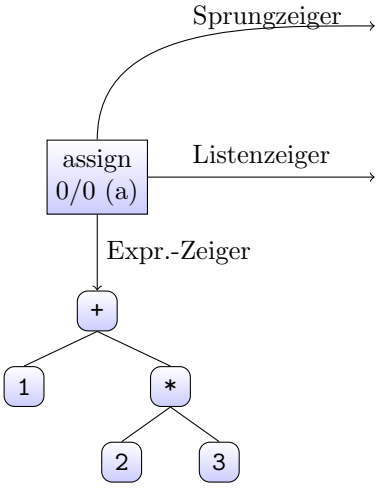
Abb. 13.4. AST für PL/0-Programm ggt.pl0



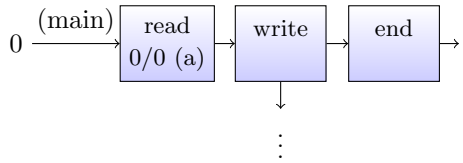
Der Syntaxbaum sollte bereits die Level-Offset-Werte der Symboltabelle enthalten, dann wird die spätere Weiterverarbeitung einfacher.



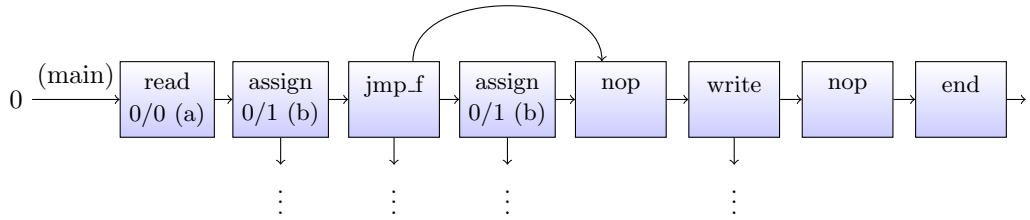

```
enum {stmt_end, stmt_assign, stmt_call, stmt_read, stmt_write,
      stmt_debug, stmt_jump_false, stmt_nop, stmt_jump};

struct s_ast_stmt {
      int type;                 // Art des Knotens
      char id[10];     // Name des Bezeichners, DEBUG
      int stl;                  // Symtab-Level
      int val;                  // Value für procs etc.
      struct s_ast_expr * expr;
      struct s_ast_stmt * next;
      struct s_ast_stmt * jump;
};
typedef struct {
      ast_stmt *start;
      char name[20];
      int n_var;
} ast_entry ;

ast_entry ast[10];
ast_stmt * new_ast_stmt(int, struct s_ast_expr * = NULL, char * = NULL, int val = -1,
```




```
var a;
begin
   ? a;
   ! 2*a
end .
```

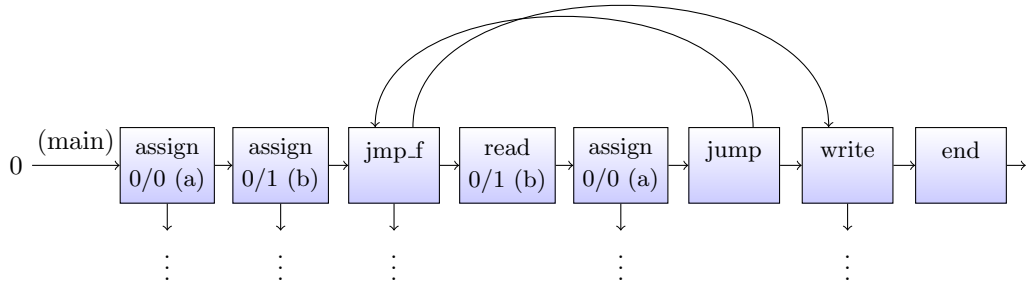



```
var a, b;
begin
    ? a;
    b := a;
    if a < 0 then
        b := -a;
    ! b;
end.
```




```
var a, b;
begin
    a :=0;
    b := -1;
    while b < 0 do
    begin
        ? b;
        a := a + 1;
    end;
    ! a;
end.
```

AST (bereits nop-optimiert):




```
var a, b;
begin
    ? a;
    if a >= 0 then
        b := a;
    else
        b := -a;
    ! b;
```

```
end.
```

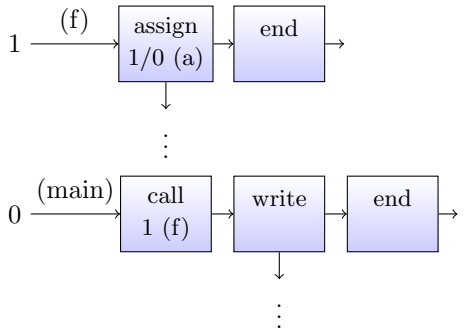
```
var a, b;
begin
    a :=0;
    repeat
        ? b;
        a := a + 1;
    until b > 0;
    ! a;
end.
```

Anmerkung: Die fußgesteuerte Schleife in Pascal impliziert eine Verbundanweisung, daher kein BEGIN-END notwendig!

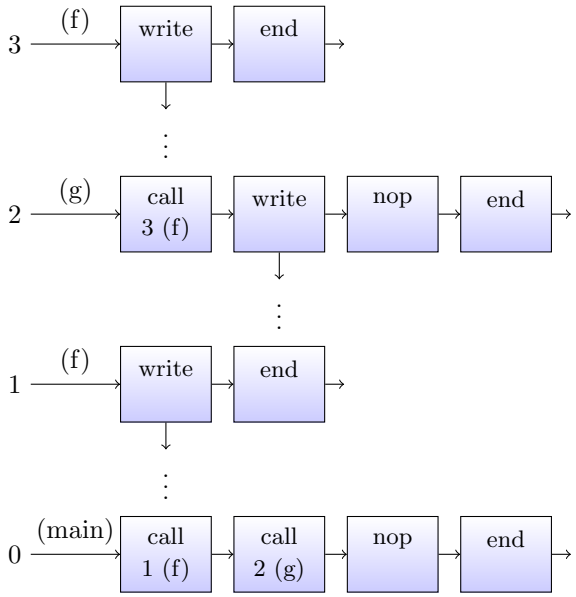
```
var i;
begin
    for i:= 1 * 2 to 1 + 2 * 3 do
        ! i
end.
```

Anmerkung: Die zählergesteuerte Schleife in Pascal arbeitet letzten Endes kopfgesteuert!

```
var a;
procedure f;
    a := 1;

begin
    call f;
    ! a
end.
```




```
procedure f;
    ! 1;

procedure g;
    procedure f;
    ! 2;

    begin
        call f;
        ! 3;
    end;

begin
    call f;
    call g;
end.
```



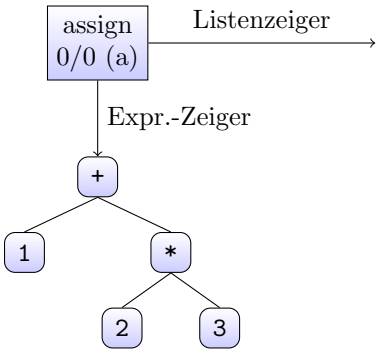
13.5.1 Die Knotenarten

Befehlsknoten

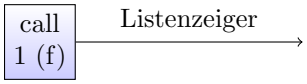
- Ende
- Zuweisung
- Funktionsaufruf
- Eingabe
- Ausgabe
- Bedingter Sprung (false)
- Unbedingter Sprung
- Leere Knoten

Keine Einträge

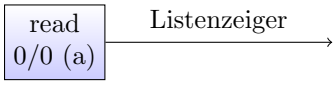
- Ziel der Zuweisung (Deklaration, zwei Int)
- Wert der Zuweisung EXPR-Zeiger
- Ggf. Name für Debugging
- next-Zeiger



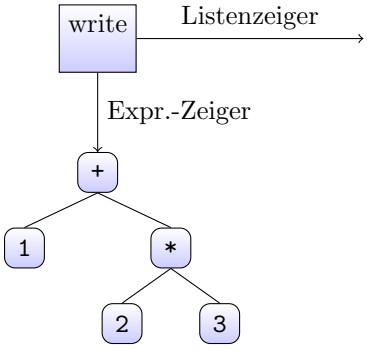
- Globale Funktions-Nummer (Int)
- Ggf. Name für Debugging
- next-Zeiger



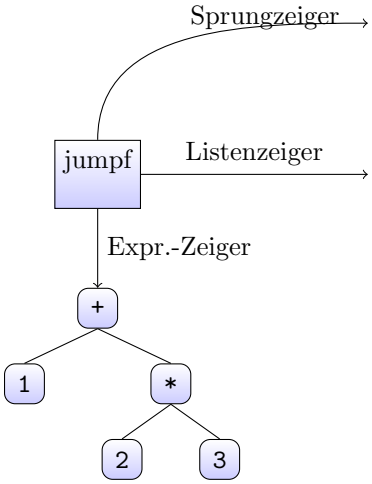
- Ziel der Eingabe (Deklaration, zwei Int)
- Ggf. Name für Debugging
- next-Zeiger



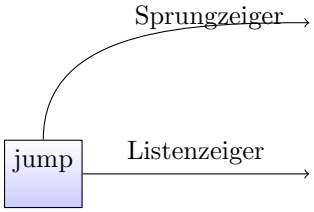
- Wert der Zuweisung EXPR-Zeiger
- next-Zeiger



- Wert der Bedingung
  EXPR-Zeiger
- next-Zeiger
- jump-Zeiger



- next-Zeiger
- jump-Zeiger



Keine Einträge 

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

- Einfache Befehle geben eine Liste mit einem Element zurück:
  - write
  - read
  - call
  - assign
- Komplexere Befehle geben eine Liste mit mehr als einem Element zurück:
  - Befehle mit impliziten Sprüngen
    - Verzweigungen
    - Schleifen
  - Verbundanweisung

AST-Erzeugung

```
s_ast_stmt * f_statement() {
    ast_stmt_list * p;
    if (token == t_write)
        p = f_stmt_write();
    else if (token == t_if)
        p = f_stmt_if();
```

```
    else if ...
        ...
    ...
    else if (token == t_begin)
        p = f_stmt_compound();
    else
        p = new_ast_stmt(stmt_nop);
    return p;
}

s_ast_stmt * f_stmt write() {
    ast_stmt_list * p = new_ast_stmt(stmt_write);
    token = scanner();
    p->expr = f_expression();
    return p;
}

s_ast_stmt *  end_of_list(s_ast_stmt * start);
// Liefert Zeiger auf letztes Element der Liste

s_ast_stmt * f_stmt_if() {
    ast_stmt_list * start = new_ast_stmt(stmt_jump_false),
        * end =  new_ast_stmt(stmt_nop);
    start->jump = end;
    token = scanner();
    start->expr = f_expression();
    start->next = f_statement();
    end_of_list(start)->next = end; // ToDo!!!
    return start;
}

s_ast_stmt * f_stmt_begin_end() {
    ast_stmt_list * start, * end;
    match(t_begin);
    start = f_statement();
    end = end_of_list(start);
    while (token == t_semikolon) {
        match(t_semikolon);
        end->next = f_statement();
        end = end_of_list(end->next);
    }
    match(t_end);
    return start;
}
```

Alternativ: Kein end-Zeiger:

```
    end_of_list(start)->next = f_statement();
```

- Datenstruktur mit zwei Zeigern
  - Schnittstellen frei von Zeigern
  - Zugriff auf Listenende einfach (konstante Laufzeit)

```
struct s_ast_stmt_list {
    s_ast_stmt *start, *end;
};
```

(Datenstruktur einer Queue)

14

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.

14.1 Optimierung der Kontrollstrukturen

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

14.1.1 Invarianter Code in Schleifen

Listing 14.1. Invariante Stringlänge (optimierung/string-durchlauf-1.c)

```
1  #include <stdio.h>
2  #include <string.h>
3  const int l_max = 1000000;
4
5  int main() {
6      char text[l_max];
7      int i, anz = 0;
8      for (i = 0; i < l_max; i++)
9          text[i] = (i % 7 == 0) ? 'a' : ' ';
10     text[l_max - 1] = '\0';
11     printf("Los geht's!\n");
12     for (i = 0; i < strlen(text); i++)
13         anz += (text[i] == ' ');
14     printf("%d Leerzeichen\n", anz);
15     return 0;
16 }
```

In der Sprache C kann der Compiler die Länge von Strings – wegen der internen Darstellung als Character-Array mit explizitem Stringende-Zeichen – nur bestimmen indem er den gesamten String durchsucht.

Listing 14.2. Invariante Stringlänge (optimierung/string-durchlauf-2.c)

```
1  #include <stdio.h>
2  #include <string.h>
3  const int l_max = 1000000;
4
5  int main() {
6      char text[l_max];
7      int i, anz = 0, l;
8      for (i = 0; i < l_max; i++)
9          text[i] = (i % 7 != 0) ? 'a' : ' ';
10     text[l_max - 1] = '\0';
11     l = strlen(text);
12     for (i = 0; i < l; i++)
13         anz += (text[i] == ' ');
14     printf("%d Leerzeichen\n", anz);
15     return 0;
16 }
```

14.1.2 If-else-if Schachtelung statistisch wählen

Listing 14.3. If-else-if Schachtelung statistisch wählen (optimierung/if-else-statistisch-1.c)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main() {
6      int z, i;
7      srand(time(NULL));
8      for (i = 0; i < 1000000000; i++) {
9          z = rand() % 10;
10         if (z == 0)
11             1;//printf("Z ist 0\n");
12         else if (z == 1)
13             2;//printf("Z ist 1\n");
14         else if (z == 2)
15             3;//printf("Z ist 2\n");
16         else if (z == 3)
17             4;//printf("Z ist 3\n");
18         else
19             5;//printf("4 <= z <= 10\n");
20     }
21     return 0;
22 }
```

Listing 14.4. If-else-if Schachtelung statistisch wählen (optimierung/if-else-statistisch-2.c)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main() {
6      int z, i;
```

```
7       srand(time(NULL));
8       for (i = 0; i < 1000000000; i++) {
9           z = rand() % 10;
10          if (z >= 4)
11              5;//printf("4 <= z <= 10\n");
12          else if (z == 0)
13              1;//printf("Z ist 0\n");
14          else if (z == 1)
15              2;//printf("Z ist 1\n");
16          else if (z == 2)
17              3;//printf("Z ist 2\n");
18          else if (z == 3)
19              4;//printf("Z ist 2\n");
20      }
21      return 0;
22  }
```

14.1.3 Binäre Schachtelung statt linearer Schachtelung

$$NL(lat) = floor\left(\frac{2\cdot\pi}{\arccos\left(1-\frac{1-\cos\left(\frac{\pi}{2\cdot NZ}\right)}{\cos^2\left(\frac{\pi}{180°}\cdot|lat|\right)}\right)}\right)$$

Table A-21. Look-Up Table for Number of Longitude Zones, NL.

Condition	Transition Latitude		Number of Longitude Zones, NL	
	Degrees (decimal)	32-bit AWB (hexadecimal)		
If lat| <	10.47047130	07 72 17 54	**Then NL(lat) =**	59
Else if lat| <	14.82817437	0A 8B 63 03	**Then NL(lat) =**	58
Else if lat| <	18.18626357	0C EE B5 50	**Then NL(lat) =**	57
Else if lat| <	21.02939493	0E F4 4B D6	**Then NL(lat) =**	56
Else if lat| <	23.54504487	10 BE 3E 9F	**Then NL(lat) =**	55
Else if lat| <	25.82924707	12 5E 12 29	**Then NL(lat) =**	54
Else if lat| <	27.93898710	13 DE 23 2C	**Then NL(lat) =**	53
Else if lat| <	29.91135686	15 45 32 43	**Then NL(lat) =**	52
Else if lat| <	31.77209708	16 97 EF 0B	**Then NL(lat) =**	51
Else if lat| <	33.53993436	17 D9 C2 3B	**Then NL(lat) =**	50
Else if lat| <	35.22899598	19 0D 3E 35	**Then NL(lat) =**	49
Else if lat| <	36.85025108	1A 34 62 2C	**Then NL(lat) =**	48
Else if lat| <	38.41241892	1B 50 C4 78	**Then NL(lat) =**	47
Else if lat| <	39.92256684	1C 63 AE 77	**Then NL(lat) =**	46
Else if lat| <	41.38651832	1D 6E 2F BC	**Then NL(lat) =**	45
Else if lat| <	42.80914012	1E 71 2A B8	**Then NL(lat) =**	44
Else if lat| <	44.19454951	1F 6D 5F 49	**Then NL(lat) =**	43
Else if lat| <	45.54626723	20 63 71 E6	**Then NL(lat) =**	42
Else if lat| <	46.86733252	21 53 F0 01	**Then NL(lat) =**	41
Else if lat| <	48.16039128	22 3F 54 E9	**Then NL(lat) =**	40
Else if lat| <	49.42776439	23 26 0C C7	**Then NL(lat) =**	39
Else if lat| <	50.67150166	24 08 77 22	**Then NL(lat) =**	38
Else if lat| <	51.89342469	24 E6 E8 E0	**Then NL(lat) =**	37
Else if lat| <	53.09516153	25 C1 AD DF	**Then NL(lat) =**	36
Else if lat| <	54.27817472	26 99 0A 48	**Then NL(lat) =**	35
Else if lat| <	55.44378444	27 6D 3B A2	**Then NL(lat) =**	34
Else if lat| <	56.59318756	28 3E 79 B3	**Then NL(lat) =**	33
Else if lat|	57.72747354	29 0C F7 42	**Then NL(lat) =**	31
Else if lat| <	58.84763776	29 D8 E2 B2	**Then NL(lat) =**	30
Else if lat| <	59.95459277	2A A2 66 B9	**Then NL(lat) =**	30
Else if lat| <	61.04917774	2B 69 A9 E5	**Then NL(lat) =**	29
Else if lat| <	62.13216659	2C 2E D0 D5	**Then NL(lat) =**	28
Else if lat| <	63.20427479	2C F1 FC B2	**Then NL(lat) =**	27

Condition	Transition Latitude		Number of Longitude Zones, NL	
	Degrees (decimal)	32-bit AWB (hexadecimal)		
Else if lat| <	64.26616523	2D B3 4C 60	**Then NL(lat) =**	26
Else if lat| <	65.31845310	2E 72 DC BC	**Then NL(lat) =**	25
Else if lat| <	66.36171008	2F 30 C7 D8	**Then NL(lat) =**	24
Else if lat| <	67.39646774	2F ED 27 0C	**Then NL(lat) =**	23
Else if lat| <	68.42322022	30 A8 11 2E	**Then NL(lat) =**	22
Else if lat| <	69.44242631	31 61 9B A1	**Then NL(lat) =**	21
Else if lat| <	70.45451075	32 19 DA 2E	**Then NL(lat) =**	20
Else if lat| <	71.45986473	32 D0 DF 12	**Then NL(lat) =**	19
Else if lat| <	72.45884545	33 B6 BA F3	**Then NL(lat) =**	18
Else if lat| <	73.45177442	34 3B 7C CB	**Then NL(lat) =**	17
Else if lat| <	74.43893416	34 EF 31 C5	**Then NL(lat) =**	16
Else if lat| <	75.42056257	35 A1 E4 F8	**Then NL(lat) =**	15
Else if lat| <	76.39684391	36 53 9E FA	**Then NL(lat) =**	14
Else if lat| <	77.36789461	37 04 65 38	**Then NL(lat) =**	13
Else if lat| <	78.33374083	37 B4 38 EB	**Then NL(lat) =**	12
Else if lat| <	79.29428225	38 63 15 64	**Then NL(lat) =**	11
Else if lat| <	80.24923213	39 10 ED 48	**Then NL(lat) =**	10
Else if lat| <	81.19801349	39 BD A5 B3	**Then NL(lat) =**	9
Else if lat| <	82.13956981	3A 69 0D 67	**Then NL(lat) =**	8
Else if lat| <	83.07199445	3B 12 CB 8A	**Then NL(lat) =**	7
Else if lat| <	83.99173563	3B BA 3A 96	**Then NL(lat) =**	6
Else if lat| <	84.89166191	3C 5E 0E 31	**Then NL(lat) =**	5
Else if lat| <	85.75541621	3C FB 4C 0F	**Then NL(lat) =**	4
Else if lat| <	86.53536998	3D 89 48 BA	**Then NL(lat) =**	3
Else if lat| <	87.00000000	3D DD DD DE	**Then NL(lat) =**	2
Else			**NL(lat) =**	1

Listing 14.5. Zonenberechnung bei ADS-B mittels binärer Suche (optimierung/ads-b-binaere-suche.c)

```
1  int NL(double x) {
2      static double zones [] = {
3      0.000000000, 10.47047130, 14.82817437, 18.18626357, 21.02939493, 23.54504487,
4      25.82924707, 27.93898710, 29.91135686, 31.77209708, 33.53993436, 35.22899598,
5      36.85025108, 38.41241892, 39.92256684, 41.38651832, 42.80914012, 44.19454951,
6      45.54626723, 46.86733252, 48.16039128, 49.42776439, 50.67150166, 51.89342469,
7      53.09516153, 54.27817472, 55.44378444, 56.59318756, 57.72747354, 58.84763776,
8      59.95459277, 61.04917774, 62.13216659, 63.20427479, 64.26616523, 65.31845310,
9      66.36171008, 67.39646774, 68.42322022, 69.44242631, 70.45451075, 71.45986473,
10     72.45884545, 73.45177442, 74.43893416, 75.42056257, 76.39684391, 77.36789461,
11     78.33374083, 79.29428225, 80.24923213, 81.19801349, 82.13956981, 83.07199445,
12     83.99173563, 84.89166191, 85.75541621, 86.53536998, 87.00000000, 90.00000000};
13     static int n = sizeof(zones) / sizeof(double);
14      int ug = 0, og = n - 1, ok = 0, m;
15      do {
16          m = (ug + og) / 2;
17          if (x < zones[m])              og = m - 1;
18          else if (x >= zones[m + 1]) ug = m + 1;
19          else                           ok = 1;
20      } while (!ok);
21      return 59 - m;
22 }
```

14.1.4 Mehrfachverzeigung statt if-else-if bei Vergleichen mit Konstanten

- Mehrfachverzweigungen sind flexibel, bei vielen Verzeigungsmöglichkeiten aber langsam
- switch-case extrem schnell:

- Ganzzahliger Verzweigungsausdruck
- Konstante Sprungziele (zur Compiletime bekannt)

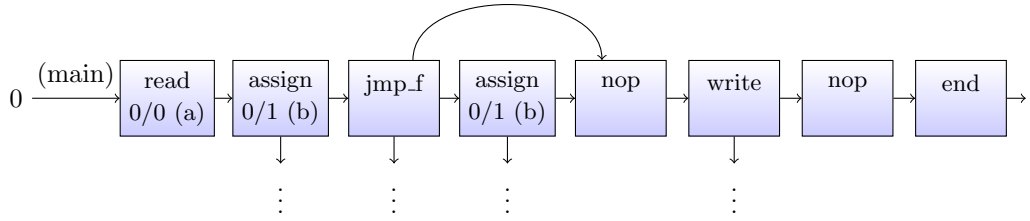
ermöglichen Sprünge über Lookup-Tabellen

14.2 Optimierung unnötiger Speicherarbeiten

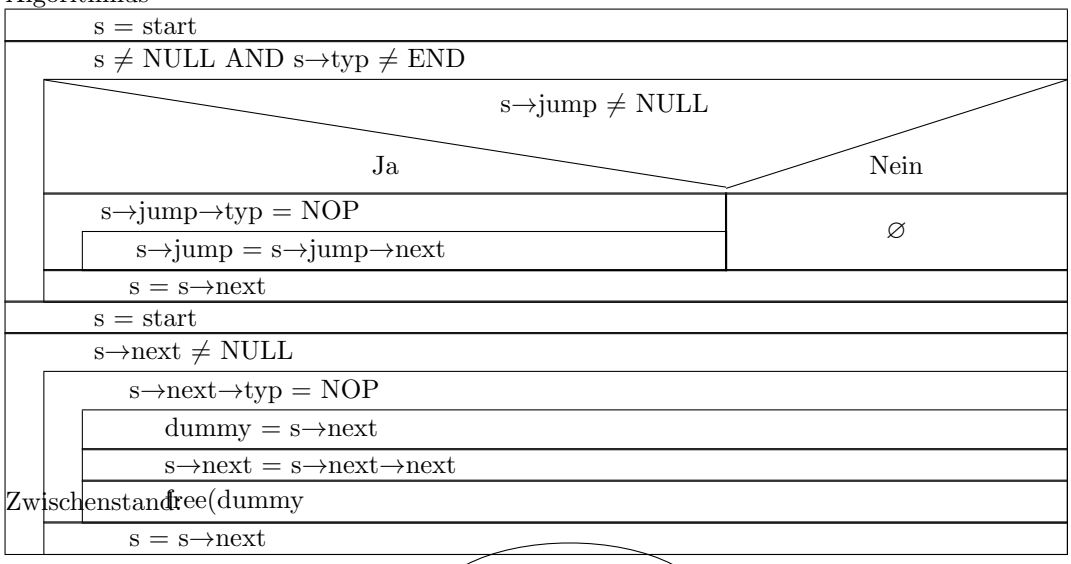
- Statische Variablen bei Initialisierung
- Call-by-Reference statt Call-by-Value
- Unnötige Zwischenvariablen vermeiden

14.2.1 Optimierung von NOPs

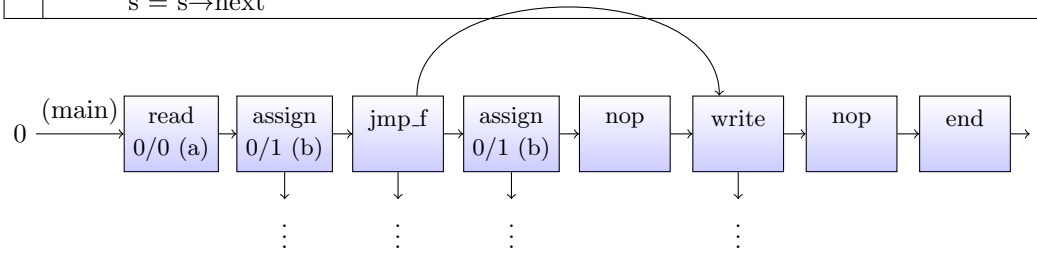
In vielen Ziel-Sprachen werden NNo-OperationStatements (kurz NOPs) als Sprungziele verwendet. Diese NOPs sollten im Zuge der Optimierung entfernt werden.



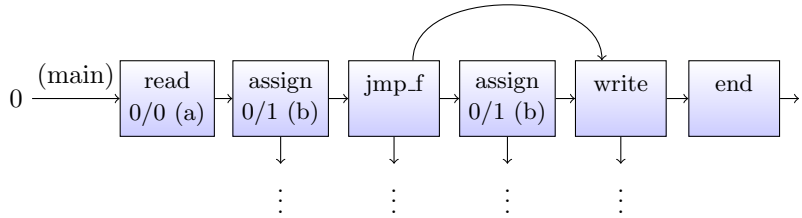
Algorithmus



Zwischenstand:

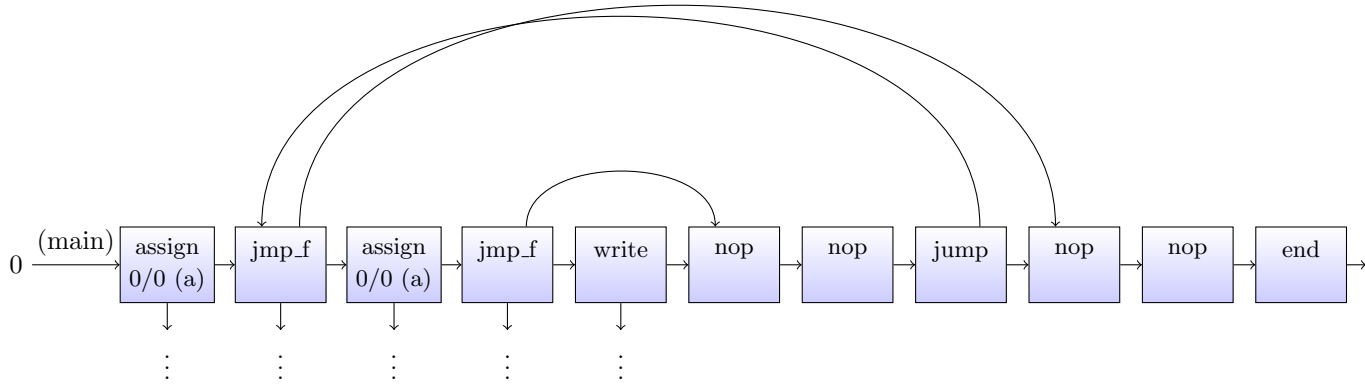


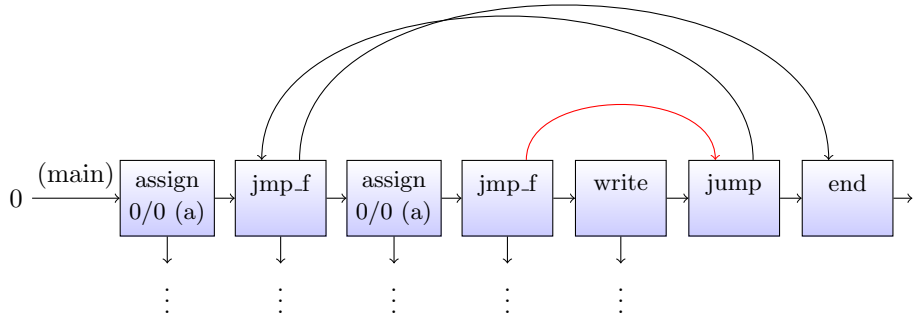
Endstand:

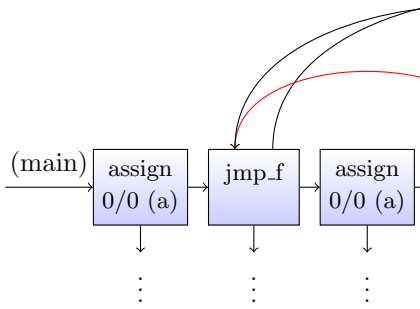


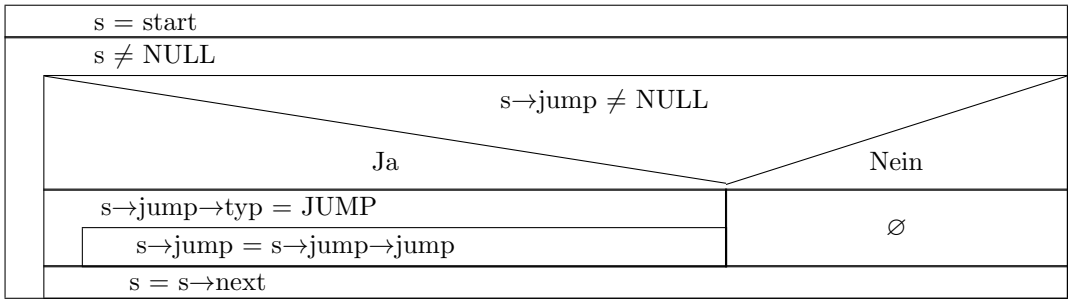
14.2.2 Eliminierung doppelter Sprünge

```
VAR a;
BEGIN
    a := 10;
    WHILE a > 0 DO
    BEGIN
        a := a - 1;
        IF a > 5 THEN
            ! a;
    END;
END.
```









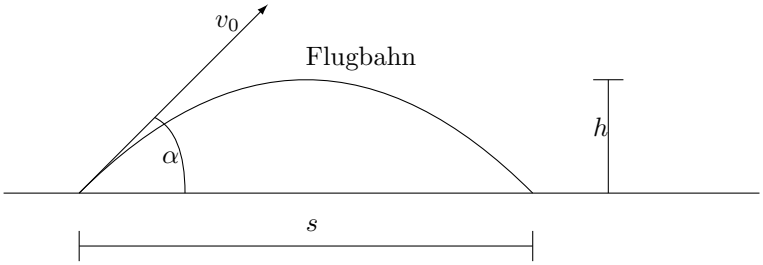
14.3 Optimierung von Ausdrücken

- Suche identischer Bäume
- Zwischenergebnisse abspeichern
- Geschickte arithmetische Umformungen:
  - Quadratbildung oder Multiplikation?
  - Vermeidung doppelter Rekursion
- Konstante Teile zur CT auswerten

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

Listing 14.6. Rekursives 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:

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

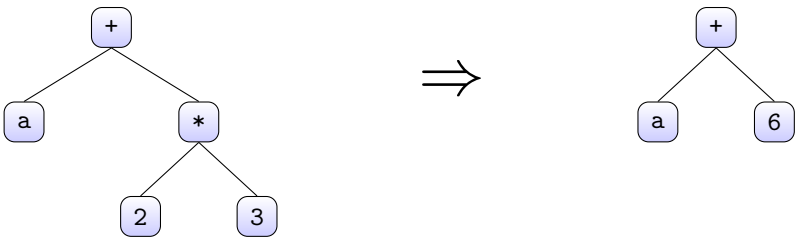
- Wird die Quadrierung über die Potenzierung abgebildet so wird über Logarithmen langsam gerechnet.
- Wird die Quadrierung über Multiplizierung abgebildet so müssen insgesamt drei Multiplikationen durchgeführt werden und $\sin(\alpha)$ muss zweimal berechnet werden.

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

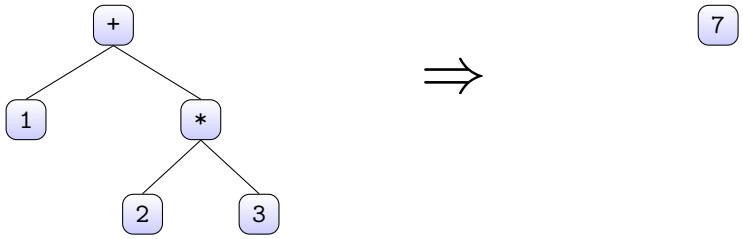
```
temp := v0 * sin(alpha);
h = temp * temp / (2 * g);
```

Auf diese Weise werden nur zwei Multiplikationen benötigt und $\sin(\alpha)$ muss nur einmal berechnet werden.

```
var a, b;
begin
        ? a;
        b := a + 2 * 3;
        ! b;
end.
```

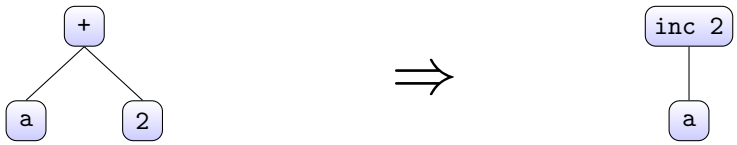



```
const a = 1;
var b;
begin
        b := a + 2 * 3;
        ! b;
end.
```



Wenn rechter Kindknoten eine Konstante ist verwende inc / dec

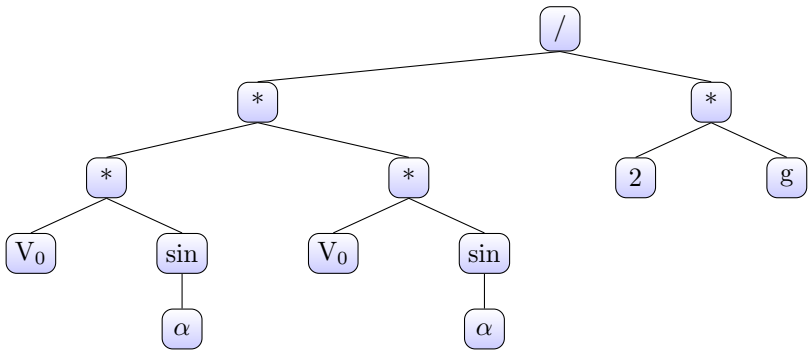
```
const a = 1;
var b;
begin
        b := a + 2;
        ! b;
end.
```

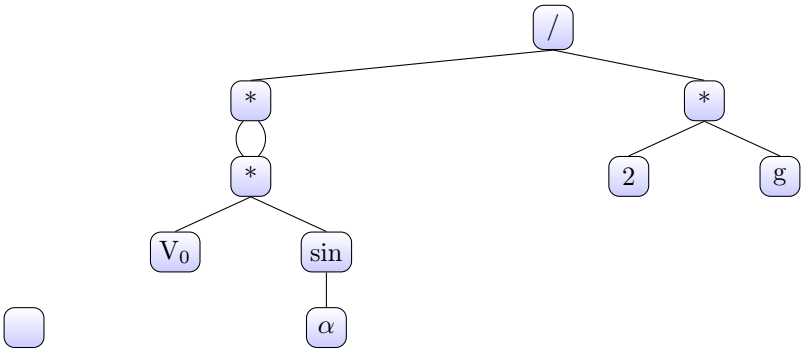


14.4 Eliminierung unnötigen Codes

- Schreibvorgänge ohne nachfolgende Lesevorgänge
- GAG statt Baum

```
var a, b;
begin
        a := 1;
        b := 2;
        ! b;
end.
```





14.5 Lookup-Tabellen

- Häufig effizientere Implementierung als programmgesteuerte Implementierung
- Beispiele:
  - Delta-Tabelle der Automaten
  - ADS-B Zonenberechnung
  - Zündzeitpunkt einer elektronischen Verbrennungsmotorsteuerung

Listing 14.7. Schulnotenausgabe (optimierung/ueb-schulnoten.c)

Listing 14.8. Schulnotenausgabe (optimierung/ueb-schulnoten-2.c)

Listing 14.9. Extensive Sinus-Nutzung (optimierung/sinus-1/sinus-1.ino)

```
1  void setup() {
2    // put your setup code here, to run once:
3
4  }
5
6  void loop() {
7    static double y;
8    static int i;
9    y = sin(i / 180.0 * PI);
10   // ...
11   i = i + 1 % 360;
12 }
```

Listing 14.10. Minimale Sinus-Nutzung (optimierung/sinus-2/sinus-2.ino)

```
1  double sinus_lookup[360];
2
3  void setup() {
4    for (int i = 0; i < 360; i++)
5    sinus_lookup[i] = sin(i / 180.0 * PI);
6  }
7
8  void loop() {
```

```
 9     static double y;
10     static int i;
11     y = sinus_lookup[i]
12     // ...
13     i = i + 1 % 360;
14 }
```

14.6 Optimierung des Speicherzugriffs

RAM / Register / HEAP / Stack

15

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:

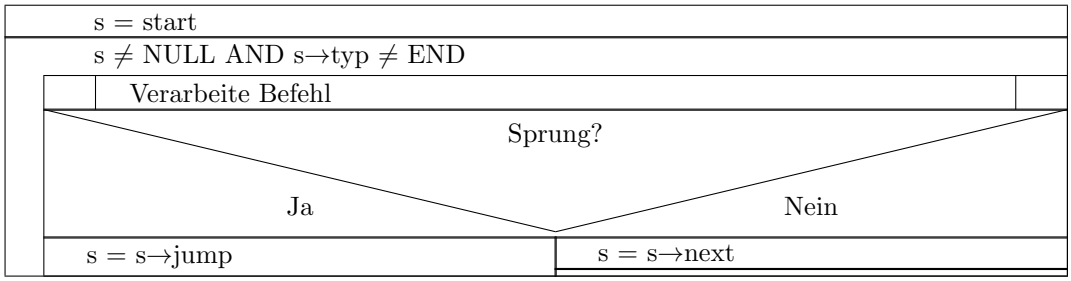
- Direkte Interpretierung des AST
- Erzeugung von Assembler-Code
- Erzeugung einer anderen Hochsprache

- Der Zwischencode ist syntaktisch richtig!

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

15.1 Interpretierung

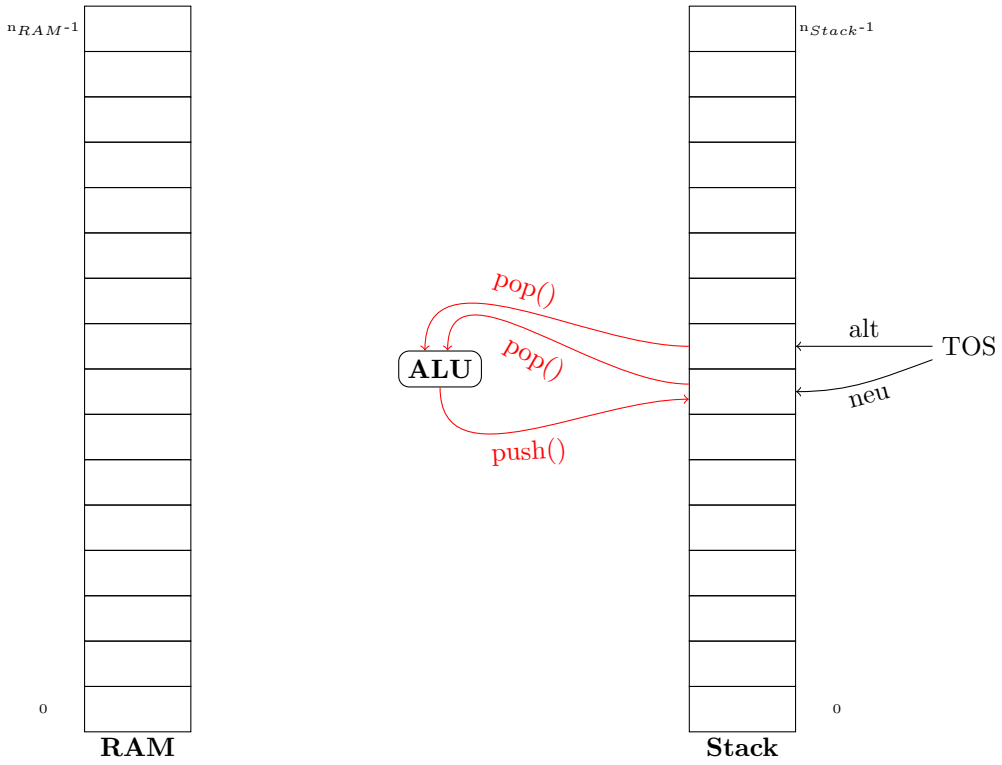
Sprachen die keinerlei Sprünge kennen, können direkt während des Parse-Vorgangs interpretiert werden. Entfernt man aus der PL/0-Grammatik (Grammatik Γ_1 Seite 19) aus der Block-Regel die Prozedur-Deklaration sowie die Befehle `IF` und `WHILE` so entsteht ein linear ablaufendes Programm.



15.2 Erzeugung von Assembler-Code

15.2.1 Maschinen

- Einfachste Maschine
- Einfach zu programmieren
- Besitzt einen Stack
- Alle Operationen beziehen sich auf Stack
  →Keine Adressrechnung notwendig

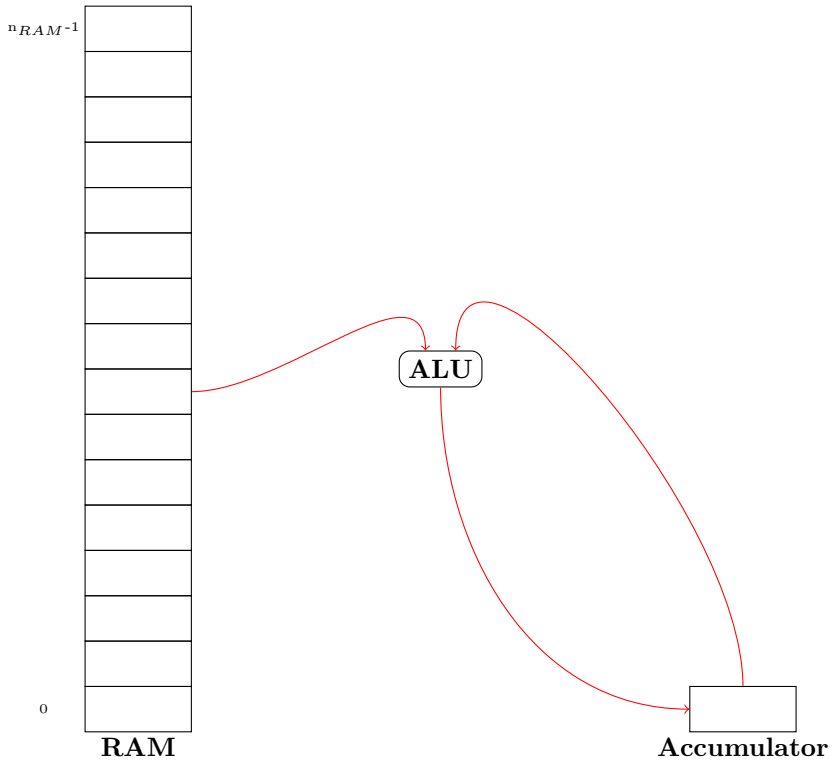


1*2+3*4

```
load  1
load  2
mult
load  3
load  4
mult
add
```

- Akkumulator
- Quasi Register-Stack-Maschine mit Höhe 1
- Typische Befehle:
  - loadc speichert Konstante in Akku
  - storer kopiert Akku in RAM
  - loadr kopiert RAM in AKKU
- Operatoren verknüpfen RAM mit Akkumulator
- C-Äquivalent(e) für c = a + b:

```
accu = a;
accu += b;
c := accu;
```




```
int ast_1_address(ast_node * p, int level) {
        static int tmpadr;
        int adr_1, adr_2, rc;
        if (!level)
                tmpadr = 999;
        if (p->l == NULL) { // Blatt
                printf("loadc\t%s\n", p->txt);
                printf("storer\t%d\n", rc = ++tmpadr);
        }
        else {
                adr_1 = ast_1_address(p->l, level + 1);
                adr_2 = ast_1_address(p->r, level + 1);
                printf("loadr\t%d\n", adr_1);
                printf("%s=\t%d\n", p->txt, adr_2);
                printf("storer\t%d\n", rc = ++tmpadr);
        }
        return rc;
}
```

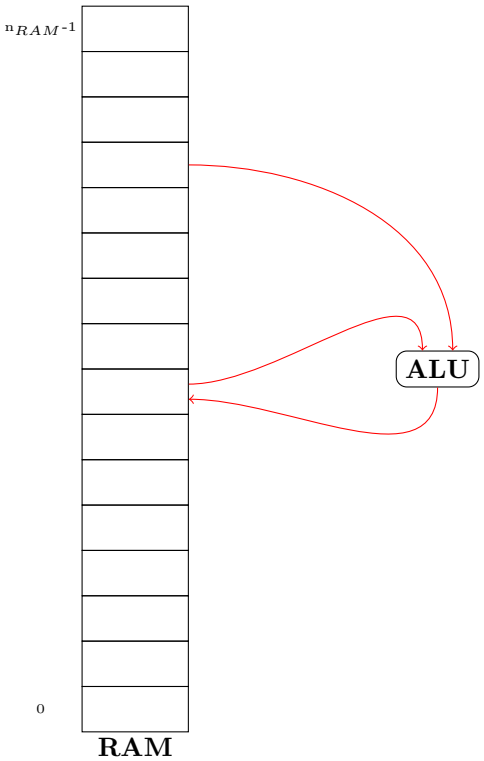
1*2+3*4

```
loadc 1
storer 1000
loadc 2
storer 1001
loadr 1000
*= 1001
storer 1002
loadc 3
storer 1003
```

```
loadc 4
storer 1004
loadr 1003
*= 1004
storer 1005
loadr 1002
+= 1005
storer 1006
```

- Code einfach zu optimieren
- Typische Befehle:
  - storer speichert Konstante in RAM
  - move kopiert RAM in RAM
- Jeder Rechenbefehl ist Tripel
  - Operator
  - Linker Operand = Ziel(-Operand)
  - Rechter Operand
- C-Äquivalent(e) für c = a + b:

```
c = a;
c += b;
```




```
int ast_2_address(ast_node * p, int level) {
        static int tmpadr;
        int adr_1, adr_2, rc;
        if (!level)
```

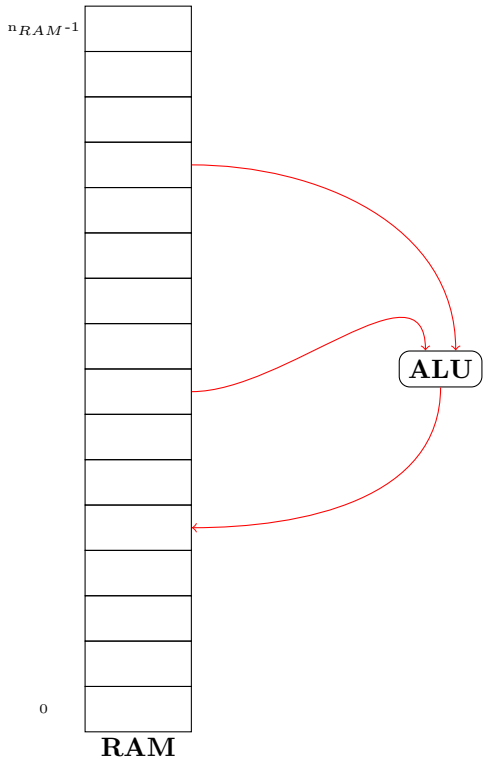
```
            tmpadr = 999;
        if (p->l == NULL) { // Blatt
            printf("storer\t%d %s\n", rc = ++tmpadr, p->txt);
        }
        else {
            adr_1 = ast_2_address(p->l, level + 1);
            adr_2 = ast_2_address(p->r, level + 1);
            printf("move %d => %d\n", adr_1, rc = ++tmpadr);
            printf("%d %s= %d\n", rc, p->txt, adr_2);
        }
        return rc;
}
```

1*2+3*4

```
storer 1000 1
storer 1001 2
move 1000 => 1002
1002 *= 1001
storer 1003 3
storer 1004 4
move 1003 => 1005
1005 *= 1004
move 1002 => 1006
1006 += 1005
```

- Wird gerne als Zwischencode eingesetzt
- Code einfach zu optimieren
- Typische Befehle:
  - storer speichert Konstante in RAM
  - move kopiert RAM in RAM (seltener notwendig als bei 2-Adress-Maschine)
- Jeder Rechenbefehl ist Quadrupel
  - Operator
  - Linker Operand
  - Rechter Operand
  - Ziel(-Operand)
- C-Äquivalent(e) für c = a + b:

```
c := a + b;
```



- Rekursiv:
  - Blätter geben Wert zurück
  - Knoten erzeugen temporären Namen
  - Code wird ausgegeben
  - Temporärer Name wird zurückgegeben

```
int ast_3_address(ast_node * p, int level) {
        static int tmpadr;
        int adr_1, adr_2, rc;
        if (!level)
        tmpadr = 999;
        if (p->l == NULL) { // Blatt
                printf("storer\t%d %s\n", rc = ++tmpadr, p->txt);
        }
        else {
                adr_1 = ast_3_address(p->l, level + 1);
                adr_2 = ast_3_address(p->r, level + 1);
                printf("%d = %d %s %d\n",
                        rc = ++tmpadr, adr_1, p->txt, adr_2);
        }
        return rc;
}
```

1*2+3*4

```
storer 1000 1
storer 1001 2
1002 = 1000 * 1001
```

```
storer 1003 3
storer 1004 4
1005 = 1003 * 1004
1006 = 1002 + 1005
```

15.2.2 Zielsprache-Erzeugung

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

Soll Assembler- oder Maschinencode erzeugt 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.

15.2.3 Der Emitter

```
int emit_n(int _nr, command_code cmd, int arg, char * comment);
// _nr: Zeilen-Nr in die ausgegeben wird,
// wenn _nr < 0 dann Ausgabe in neue Zeile am Ende
// code: Befehl
```

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.

```
int emit_s(int _nr, command_code cmd, int arg, char * comment);
// _nr: Label-Nr, -1 wenn kein Label ausgegeben werden soll
// code: Befehl
```

15.2.4 Ausdrücke

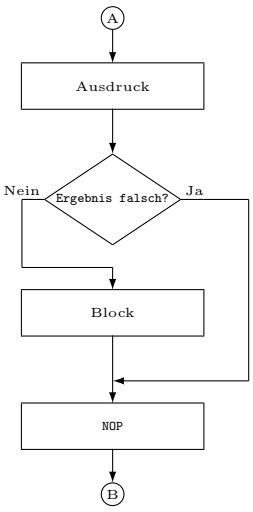
Ausdrücke werden wie schon behandelt (bei Register-Stack-Masachinen) in UPN umgesetzt.

15.2.5 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 15.1 als Ablaufplan und Assembler dargestellt.

Abb. 15.1. Ablaufplan und Assembler Einfache Verzweigung

Ablaufplan if




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

TODO
```

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.

```
1 void f_if() {
2         int lnr1, lnr2;
3         match(t_if, e_if_req);              // Lies weiter
4         f_condition();                      // Code-Ausgabe der Bedingung
5         match(t_then, e_then_req);          // Lies weiter
6         lnr1 = emit_n(-1, cmd_JMPZ, 0, ""); // Sprungziel falsch!!
```

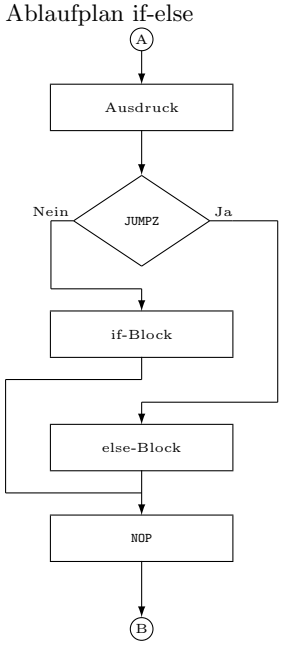
```
7           f_statement();
8           lnr2 = emit_n(-1, cmd_NOP, 0, "Sprungziel");
9           emit(lnr1, cmd_JMPZ, lnr2, "");     // Sprungziel richtig
10  }
```

```
1   int label_nr = 0;
2   void f_if() {
3           int l = label_nr ++;
4           match(t_if, e_if_req);              // Lies weiter
5           f_condition();                      // Code-Ausgabe der Bedingung
6           match(t_then, e_then_req);          // Lies weiter
7           emit_s(-1, cmd_JMPZ, l, "");
8           f_statement();
9           emit(l, cmd_NOP, 0, "Sprungziel");
10  }
```

15.2.6 if-else-Statement

Abb. 15.2. Ablaufplan Verzweigung mit Alternative

Ablaufplan if-else



15.2.7 Schleifen

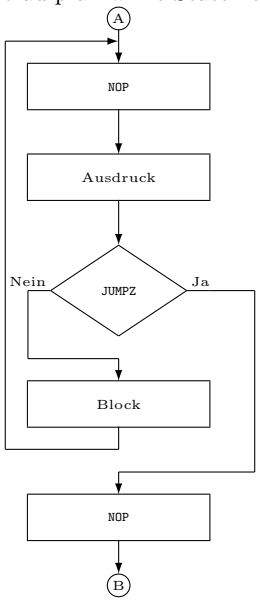
Der Code "`while '(' <Ausdruck> ')' <Block>`" muß folgendermaßen umgesetzt werden:

15.2.8 Variablenzugriff

Wird dynamische Speicherverwaltung eingesetzt so ist der Variablenzugriff relativ kompliziert, da die Adresse einer Variablen erst zur Runtime errechnet werden kann. Je nachdem

Abb. 15.3. Ablaufplan kopfgesteuerte schleife

Ablaufplan while-Statement



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.

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 15.1 zeigt den Zugriff.

Listing 15.1. 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 15.1 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.

15.3 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

15.3.1 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 15.1 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!!!

}
```

15.4 Code-Erzeugung aus AST

- Im AST sind Befehle bereits eindeutig!
  - Zeilen-Nummern bei zeilenbasiertem Zwischencode
  - Pointer-Werte bei dynamischen Datenstrukturen
- Diese Werte können als symbolische Labels benutzt werden

15.5 Erzeugung einer anderen Hochsprache

15.5.1 Erzeugung eines LaTeX-tikz-Tree

```
void ast_tikz_simple(ast_node * p, int level) {
        char tabs[] = "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t",
        * tl = tabs + sizeof(tabs) - 1;
        int nr, i;
        if (p != NULL) {
                if (!level)
                        printf("\\Tree");
                printf("%s", tl - level);
                if (p->l != NULL) // Inneren Knoten auszeichnen
                        printf("[.");
                printf("{%s}\n", p->txt);
                ast_tikz_simple(p->l, level + 1);
                ast_tikz_simple(p->r, level + 1);
                if (p->l != NULL) // Inneren Knoten auszeichnen
                        printf("%s]\n", tl - level);
        }
}
```

```
\Tree [.{+}
        {1}
        [.{*}
                {2}
                {3}
        ]
]
```

16

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.

16.1 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.

16.2 Gültigkeitsbereiche

Vereinbarungen:

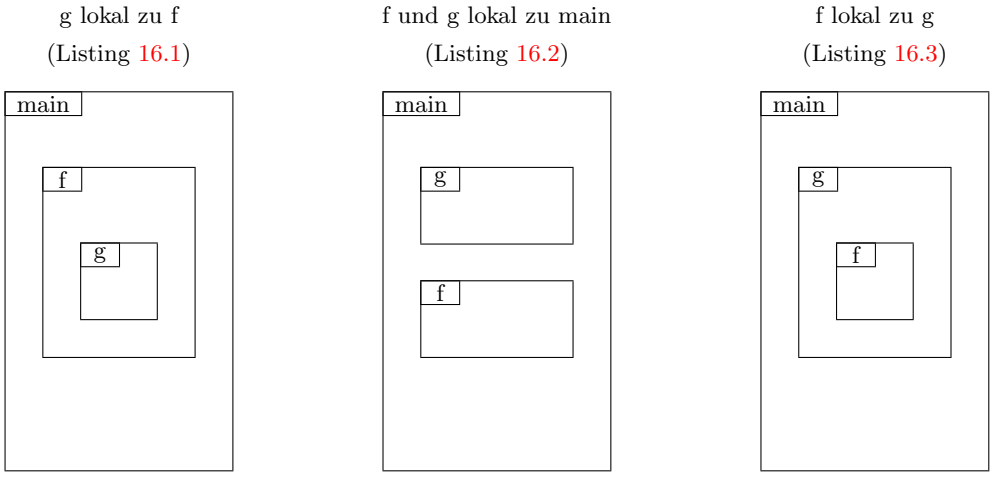
- f ruft g
- Die Level seien L_f und L_g

- 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. 16.1. Schachtelungsmöglichkeiten von Funktionen



Listing 16.1. 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 16.2. 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 16.3. 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.
```

16.3 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 12.1 (Seite 136) 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. 16.2. Symboltabelle bei statischer Hauptspeicherverwaltung (Listing 16.1)

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. 16.3. Statischer Hauptspeicher g lokal zu f (Listing 16.1)

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

16.4 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.

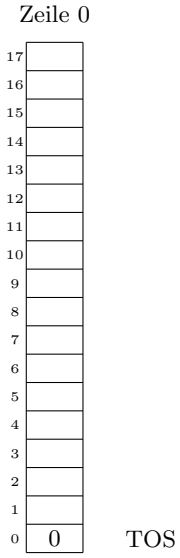
Abbildung 16.5 bezieht sich auf Listing 16.1.

Abbildung 16.6 bezieht sich auf Listing 16.2.

Abbildung 16.7 bezieht sich auf Listing 16.3.

Zum Programmstart ist der Hauptspeicher bei allen Programmen leer, wie in Abbildung 16.4 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 16.4).

Abb. 16.4. 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. 16.5. Hauptspeicher g lokal zu f (Listing 16.1)

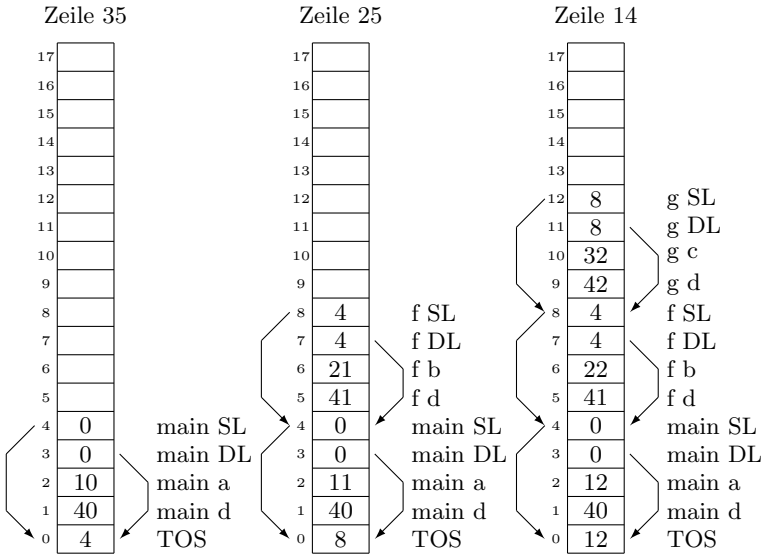
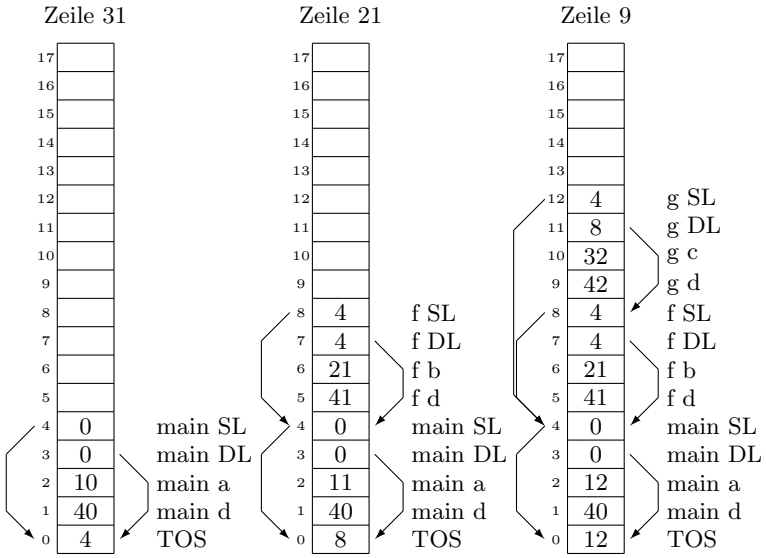


Abb. 16.6. Hauptspeicher g und f lokal zu main (Listing 16.2)



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

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

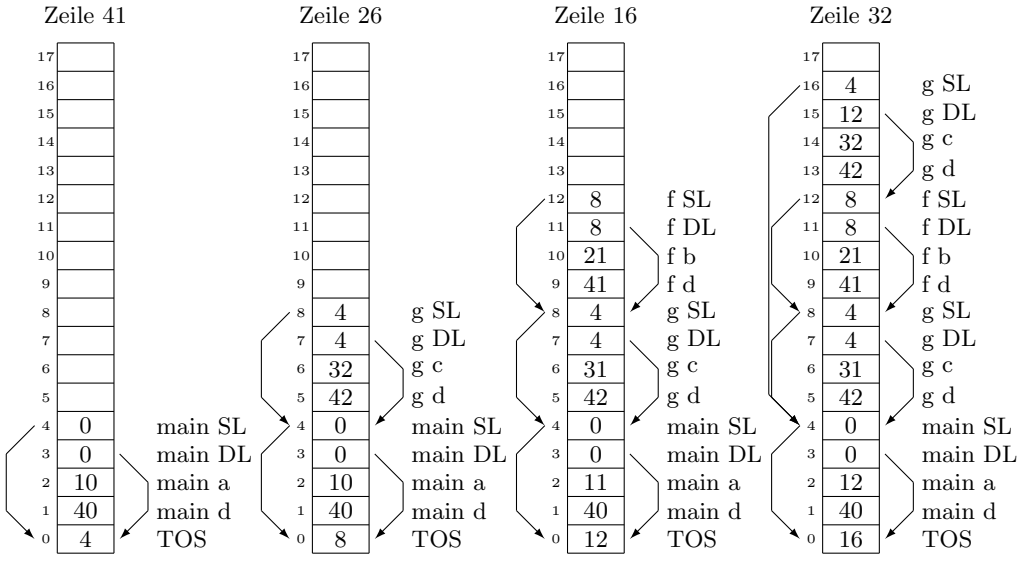
$L_f = L_g - 1$ (g lokal zu f):

- 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. 16.7. Hauptspeicher f lokal zu g (Listing 16.3)



$L_f = L_g + 1$ (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

16.4.1 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)
```

16.5 Statische vs. Dynamische Speicherverwaltung

Tabelle 16.1. 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

Label	Befehl	Argument	Kommentar	Stack
	loadr	0	# Alter TOS	4
	inc	2	# n_var	6
	inc	2	# AR-Größe	8
	storer	0		

Achtung: Alter TOS muss noch existieren!

Label	Befehl	Argument	Kommentar	Stack
	loadr	0	# Alter TOS	4
	inc	2	# n_var	6
	inc	2	# AR-Größe	8
	dec	1		7
	loadr	0	# Alter TOS	7 4
	swap			4 7
	stores			

Annahme: Neuer SL (4) auf Stack

Label	Befehl	Argument	Kommentar	Stack
	loadr	0	# Alter TOS	4 4
	inc	2	# n_var	4 6
	inc	2	# AR-Größe	4 8
	stores			

1. Neuen TOS berechnen
2. DL berechnen / setzen
3. neuen TOS setzen
4. SL setzen

Annahme: Neuer SL (4) auf Stack

Label	Befehl	Argument	Kommentar	Stack
1	loadr	0	# Alter TOS	4 4
	inc	2	# n_var	4 6
	inc	2	# AR-Größe	4 8
2	dup			4 8 8
	dec	1		4 8 7
	loadr	0	# Alter TOS	4 8 7 4
	swap			4 8 4 7
	stores		# DL gesetzt	4 8
3	dup			4 8 8
	storer	0	# TOS gesetzt	4 8
4	stores		# SL gesetzt	4 8

- main ruft f, $\Delta = 0$, $n_{var} = 2$
- Auf Stack:
  1. Neuer SL (hier 4)
  2. n_{var} (hier 2)

Label	Befehl	Argument	RAM 0 1 2 3 4 4 6 7 8	Stack
	CALL	RAM_UP	4 0 0	4 2
	...			
RAM_UP	loadr	0	4 0 0	4 2 4
	add		4 0 0	4 6
	inc	2	4 0 0	4 8
	dup	2	4 0 0	4 8 8
	dec	1	4 0 0	4 8 7
	loadr	0	4 0 0	4 8 7 4
	swap		4 0 0	4 8 4 7
	stores		4 0 0 4	4 8
	dup		4 0 0 4	4 8 8
	storer	0	8 0 0 4	4 8
	stores		8 0 0 4 4	
	return		8 0 0 4 4	

Label	Befehl	Argument	RAM 0 1 2 3 4 4 6 7 8	Stack
	...			
	CALL	RAM_DOWN	8 0 0 4 4	
	return			
RAM_DOWN	loadr	0	8 0 0 4 4	8
	dec	1	8 0 0 4 4	7
	loads		8 0 0 4 4	4
	storer	0	4 0 0	
	return			

16.6 Zeiger und referenzen

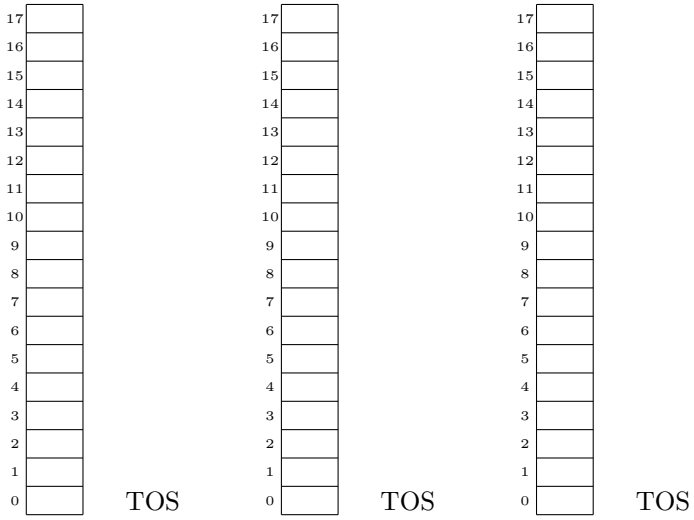
```
procedure-heading     =
"procedure" identifier [ formal-parameter-list ]
function-heading         =
"function" identifier [ formal-parameter-list ] ":" type-identifier
 formal-parameter-list =
  "(" formal-parameter-section { ";" formal-parameter-section } ")"
formal-parameter-section =
value-parameter-section | variable-parameter-section
value-parameter-section =
identifier-list ":" parameter-type
variable-parameter-section =
"var" identifier-list ":" parameter-type .
```

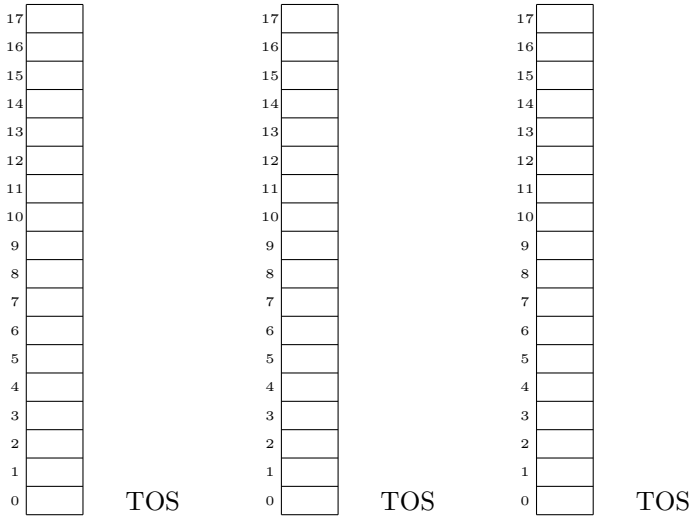
```
function ggt(a,b:integer):integer;
var r:integer;
begin
      repeat
            r := a mod b;
            a := b;
            b := r;
      until r=0;
      ggt := a;
end;
```

```
var a, b, g;

function ggt(a,b);
var r;
begin
     repeat
          r := a mod b;
          a := b;
          b := r;
     until r=0;
     ggt := a;
end;

begin
        ? a;
        ? b;
        g = ggt(a,b);
        ! g;
end.
```

```
procedure swap(var x, y: integer);
var temp: integer;
begin
   temp := x;
   x:= y;
   y := temp;
end;

var a, b;

procedure swap(var x, y);
var temp;
begin
   temp := x;
   x:= y;
   y := temp;
end;

begin
        ? a;
        ? b;
        swap(a,b);
        ! a;
        ! b;
end.
```

- Parameterwerte (auch Zeigerwerte) werden vor Aufruf auf Stack gelegt
- Beim Aufbau des Steckframe werden Parameter an entsprechende Stelle kopiert
- Vor Rücksprung wird Ergebnis auf Stack gelegt





1. Adressübergabe
2. Adresszugriff

```
procedure f;
          ! 1;

begin
          call f;
end.
```

Label	Befehl	Argument	Kommentar
	loadc	0	# RAM-INIT
	storer	0	
	loadc	0	# Pseudo-SL fuer main
	jump	FKT_0	
RAM_UP	...		
RAM_DOWN	...		
FKT_0	nop		# main
	loadc	0	# n_var
	call	RAM_UP	
	loadr	0	# SL auf Stack, ggf Kette
	call	FKT_1	# f
	call	RAM_DOWN	
	return		# Ende Funktion main
FKT_1	nop		# f
	loadc	0	# n_var
	call	RAM_UP	
	loadc	1	
	write		
	call	RAM_DOWN	
	return		# Ende Funktion f

Beispiel 1: Δ=0, lnr=1:

Label	Befehl	Argument	Kommentar
	loadr	0	# TOS
	dec	3	# lnr + AR-Size

Beispiel 2: Δ=2, lnr=3:

Label	Befehl	Argument	Kommentar
	loadr	0	# TOS
	loads		# SL-Kette 1.)
	loads		# SL-Kette 2.
	dec	5	# lnr + AR-Size

- Für Lesezugriff:
  - Anschließend loads
- Für Schreibzugriff:
  - Vorher zu schreibenen Wert auf Stack (read, expression)
  - Anschließend stores

17

Laufzeit

Siehe Semantik

Index wird Ausdruck statt int

18

Fehlerbehandlung

https://de.wikipedia.org/wiki/Laufzeitfehler

Lexikalische Fehler: Der Scanner findet keinen zugelassenen regulären Ausdruck

Syntaktische Fehler: Der Parser kann keinen Syntaxbaum aufbauen

Semantische Fehler: Die Eingabe ist grammatikalisch Korrekt, ergibt aber keinen Sinn

Abbruch: Der Compiler bricht die Verarbeitung ab.

Weiter-Übersetzung ohne Ausgabe: Der Compiler versuch ab einer gewissen Stelle mit der Grammatik wieder aufzusetzen

Ignorieren: Der Compiler versucht die Eingabe bestmöglich zu verarbeiten

18.1 Lexikalische Fehler

- Scanner hat mehrere Möglichkeiten
  - Verändern des Textes
    1,x → 1.0
  - Auslassen (Überlesen) des Textes
  - Fehlertoken (Weiterverarbeitung im Parser)

18.2 Semantische Fehler

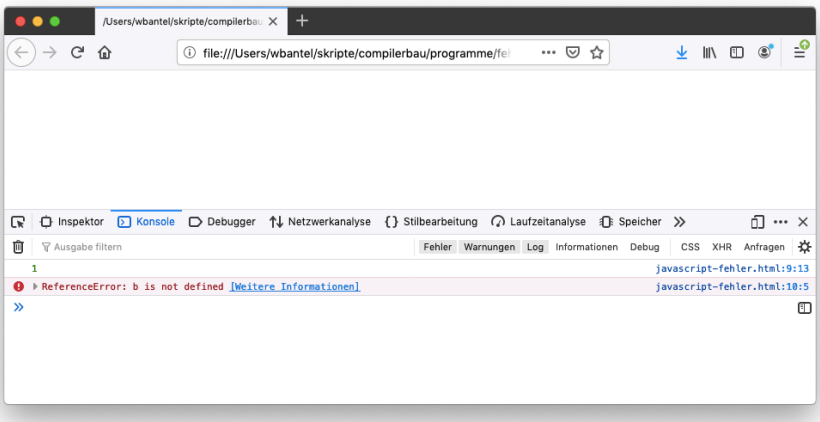
18.3 Fehlerbehandlung „Abbruch“

- JavaScript im Browser
- Browser (XHTML)
- Diverse Interpreter (bspw. SQL)
- Tabellenkalkulation

Listing 18.1. JavaScript - Abbruch bei Fehler (fehlerbehandlung/javascript-fehler.html)

```
1 <html>
2 <head>
3     <meta http-equiv="Content-Type"
```

```
4       content="text/html;charset=ISO-8859-1">
5   </head>
6   <body>
7       <script>
8           var a = 1, c = 2;
9           console.log(a);
10          console.log(b);
11          console.log(c);
12      </script>
13  </body>
14  </html>
```



- Programmiersprachen-Compiler

18.4 Fehlerbehandlung„Weiter-Übersetzung ohne Ausgabe“

Listing 18.2. C - Weiterübersetzung bei Fehler (fehlerbehandlung/c-fehler.c)

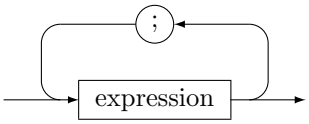
```
1  #include <stdio.h>
2  struct a {
3      int b;
4  }
5  int main() {
6      int a;
7      b = a + 1;
8      c = b + 2;
9      return;
10 }
```

Bildschirmausgabe:

```
c-fehler.c:4:2: error: expected ';' after struct
}
 ^
 ;
c-fehler.c:7:5: error: use of undeclared identifier 'b'
    b = a + 1;
    ^
c-fehler.c:8:5: error: use of undeclared identifier 'c'
    c = b + 2;
    ^
c-fehler.c:8:9: error: use of undeclared identifier 'b'
    c = b + 2;
        ^
c-fehler.c:9:5: error: non-void function 'main' should return a value [-Wreturn-type]
    return;
    ^
5 errors generated.
```

- „DAU-Compiler“
- Internet-Eingaben
- Browser (HTML)

Syntaxdiagramm 8: Wiederholung



Listing 18.3. Weiterübersetzen Recursive-Descent (fehlerbehandlung/weiteruebersetzen-rd.c)

```
1  void f_input() {
2      ast_node * p;
3      int oncemore;
4      do {
5          error = 0, p = f_expression();
6          if (!error)
7              printf("Ergebnis: %d\n", ast_eval(p));
8          else
9              printf("Syntaxfehler\n");
10         if ((oncemore = token == t_semikolon) != 0)
11             token = yylex();
12     } while (oncemore);
13 }
```

```
==>  cat rechnungen.txt
2+-1;
1+2*3+;
3;
==>  ./a.out < rechnungen.txt
Ergebnis: 1
Syntaxfehler
Ergebnis: 3
```

```
Syntaxfehler
Fehler 2
==>
```

- Wenn Fehler auftritt wird error-Token generiert
- Wenn error-Token in Grammatik vorkommt wird Regel abgearbeitet
- YACC versucht nach drei gülten Tokens wiederaufzusetzen
- yyerrok-Statement setzt explizit wieder auf

Listing 18.4. Weiterübersetzen YACC (fehlerbehandlung/weiteruebersetzen-yacc.y)

```
1  input : expr0
2        | input t_semikolon {++nr; } expr0
3        ;
4  expr0 : expr {printf("Ergebnis %d: %d\n", nr, ast_eval($1));}
5        | error {printf("Fehler %d\n", nr); yyerrok;}
6        ;
7  expr: term {$$ = $1;}
8        | expr t_plus term {$$ = ast_new_node("+", $1, $3);}
9        ;
10 term  : factor  {$$ = $1;}
11       | term t_mal factor {$$ = ast_new_node("*", $1, $3);}
12       ;
13 factor   : t_zahl {$$ = ast_new_node($1, NULL, NULL);}
14       | t_kla_auf expr t_kla_zu {$$ = $2;}
15       | t_minus factor {$$ = ast_new_node("CHS", $2, NULL);}
16       ;
```

18.5 Fehlerbehandlung „Tolerieren“

- Internetbasierte Systeme (ebay, google, ...)
- Browser (HTML)

```
int main(void) {
  error = 0, token = yylex();
  ast_node * root = f_expression();
  if (!error && token == 0)
    ast_tikz(root, 0, 1);
  else
    |printf("Fehler %d\n", error);|
  return 0;
}

ast_node * f_expression(void) {
  ast_node * p = f_term();
  while(token == t_plus) {
    p = new_ast_node(yytext, p, NULL);
    token = yylex();
    p->r = f_term();
  }
  return p;
}

ast_node * f_term(void) {
  ast_node * p = f_factor();
  while(token == t_mal) {
    p = new_ast_node(yytext, p, NULL);
    token = yylex();
    p->r = f_factor();
  }
  return p;
}

ast_node * f_factor (void) {
```

```
int main(void) {
  ast_node * root;
  token = yylex();
  root = f_expression();
  |while (token != t_end)|
    |root = ast_new_node("+",|
    |root, f_expression());|
  ast_explorer(root, 0);
  return 0;
}

ast_node * f_expression(void) {
  ast_node * p = f_term();
  while(token == t_plus) {
    p = ast_new_node(yytext, p, NULL);
    token = yylex();
    p->r = f_term();
  }
  return p;
}

ast_node * f_term(void) {
  ast_node * p = f_factor();
  while(token == t_mal) {
    p = ast_new_node(yytext, p, NULL);
    token = yylex();
    p->r = f_factor();
  }
  return p;
}


ast_node * f_factor (void) {
  ast_node * p;
  if (token == t_minus) {
    token = yylex();
    p = ast_new_node("C", f_factor(), NULL);
  }
  else if (token == t_zahl) {
    p = ast_new_node(yytext, NULL, NULL);
    token = yylex();
  }
  else if (token == t_kla_auf) {
    token = yylex();
    p = f_expression();
    if (token == t_kla_zu) {
      token = yylex();
    }  // Kein else-error !!!
  }
  else{ |// Knoten mit 0 !!!|
    p = ast_new_node("0", NULL, NULL);
    token = yylex();
  }
  return p;
}
```

A

Die Binärbaum-Library

Listing A.1. Die Binärbaum-Library (include/ast.h)

```
1  typedef struct s_ast_node ast_node;
2  struct s_ast_node {
3      char * txt;//txt[64];
4      ast_node * l;
5      ast_node * r;
6  };
7
8  ast_node * ast_new_node(char * txt, ast_node * l, ast_node * r);
9  void ast_tikz(ast_node * p, int n, int path);
10 void ast_tikz_simple(ast_node * p, int level);
11 void ast_tikz_path(ast_node * p, int level);
12 void ast_explorer(ast_node * p, int level);
13 void ast_2_aassembler(ast_node * p);
14 void ast_deltree(ast_node * p);
15 int ast_eval(ast_node * p);
16
17 int ast_1_address(ast_node * p, int level);
18 int ast_2_address(ast_node * p, int level);
19 int ast_3_address(ast_node * p, int level);
```

B

Die Syntaxbaum-Library

Listing B.1. Die Syntaxbaum-Library (include/syntaxtree.h)

```
1  #pragma once
2  typedef struct s_node node;
3  struct s_node {
4      //char txt[10];
5      char * txt;
6      node * child[30];
7  };
8  node * new_node(char * txt);
9  void tree_tikz(node * p, int n);
10 void tree_explorer(node * p, int level);
```

C

Die Stack-Library

Listing C.1. Die Stack-Library (include/stack.h)

```
1  #pragma once
2  #define stack_max_height 256
3  typedef struct {
4      int s[stack_max_height];
5      int h;
6  } stack;
7
8  int push(stack *, int);
9  int pop(stack *, int *);
10 int tos(stack *, int *);
```

Literaturverzeichnis

ASU88. Alfred Aho, Ravi Sehti, and Jeffrey D. Ullman. *Compilerbau - Band 1.* Addison-Wesley, Bonn, 1988. ISBN 3-89319-150-X. 17, 87, 89, 98

Her95. Helmut Herold. *Lex und Yacc: Lexikalische und syntaktische Analyse.* Addison-Wesley, Bonn, 2. edition, 1995. 117

HMU02. Hopcroft, Motwani, and Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson, 2. edition, 2002. ebook verfügbar. 24

KR88. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2. edition, 1988. 103

KR90. Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Hanser, München, 2. edition, 1990. ISBN 3-446-15497-3. 193

LS75. M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *Computing Science Technical Report No. 39*, October, 1975. 117

Sed92. Robert Sedgewick. *Algorithmen.* Addison-Wesley, 1992. 83

Wir86. Niklaus Wirth. *Compilerbau.* Teubner, Stuttgart, vierte edition, 1986. 19, 83

Wir96. Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, Bonn, 1996. 83, 89