Fakultät Informatik/Mathematik

HOCHSCHULE FÜR
TECHNIK UND WIRTSCHAFT
DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Lehrstuhl für Programmierung, Compilerbau/Interpreter, Mikrocontroller

Vorlesungsmitschrift

COMPILER/INTERPRETER

Mitschrift von

Falk-Jonatan Strube

Vorlesung von

Prof. Dr.-Ing. Arnold Beck

10. Februar 2018

INHALTSVERZEICHNIS

1 Einführung **7**

1.1 Motivation . . . 7
1.1.1 Der Compiler . . . 7
1.1.1.1 Aufbau eines Compilers . . . 8
1.1.1.2 Beispiel . . . 8
1.1.2 Dreiadressmaschine vs Stackmaschine . . . 9
1.1.3 Syntaxgesteuerter Einpasscompiler . . . 10
1.2 Grundlagen . . . 10
1.2.1 Sprache . . . 10
1.2.2 Alphabet . . . 11
1.2.3 Zeichenkette . . . 11
1.2.4 Freie Halbgruppe (A*) . . . 12
1.2.5 Syntaxregeln . . . 12
1.2.6 Grammatik und Parser . . . 12
1.2.7 BNF . . . 13
1.2.8 EBNF . . . 13
1.2.9 Syntaxgraph . . . 14
1.2.10 Grammatik von PL/0 . . . 15
1.3 Grammatik nach Chomsky . . . 16
1.3.1 Ableitung . . . 17
1.3.2 Chomsky Typ 0 . . . 18
1.3.3 Chomsky Typ 1 . . . 18
1.3.4 Chomsky Typ 2 . . . 18
1.3.5 Chomsky Typ 3 . . . 18
1.3.6 Chomsky Typ 4 . . . 19
1.3.7 Beispiel . . . 19
1.3.8 Analysestrategien . . . 20
1.4 Morpheme, Token, Atome . . . 20
1.4.1 Morpheme einfacher Ausdrücke . . . 20
1.4.2 Ein einfacher Lexer . . . 21
1.5 Grammatik 1: Einfache Ausdrücke . . . 22
1.5.1 Rekursiver Abstieg . . . 22
1.5.2 Anforderungen . . . 23
1.6 LL(1) Grammatiken . . . 23
1.7 Grammatik 2: Einfache Ausdrücke ohne Linksrekursion . . . 24
1.7.1 Linksfaktorisierung . . . 24
1.7.2 First/Follow . . . 24
1.7.3 Expression . . . 24
1.7.4 Term . . . 25
1.7.5 Faktor . . . 25
1.7.6 Main . . . 25
1.7.7 Ausführen . . . 26
1.8 Umformung der Grammatik . . . 26
1.8.1 Grammatik 3: Beseitigung der Linksrekursion durch Iteration . . . 26
1.8.2 EBNF . . . 27

1.8.3 Expression als C-Code . . . 27
1.8.4 Beseitigung der Linksrekursion bei klassischer BNF . . . 27
1.9 Grammatik für einfache Ausdrücke . . . 27
1.10 First/Follow . . . 28
1.10.1 Implementation . . . 28
1.11 Wiederholung . . . 29

2 Lexer **30**
2.1 Lexikalische Analyse . . . 30
2.1.1 Einordnung in den Compiler . . . 30
2.1.2 Funktionsweise . . . 31
2.2 Grammatik der PL/0 Token . . . 31
2.2.1 Regeln der Grammatik der Morpheme . . . 31
2.3 Endlicher deterministischer Automat . . . 32
2.3.1 Automatentabelle . . . 33
2.3.1.1 Zeichenklassenvektor . . . 33
2.3.1.2 Aktionen . . . 35
2.3.1.3 Fazit . . . 35
2.4 Implementierung des Lexers . . . 36
2.4.1 Datentypen . . . 36
2.4.2 Initialisierung . . . 38
2.4.3 Algorithmus . . . 38
2.4.4 Funktionen . . . 39
2.4.4.1 Beenden . . . 39
2.5 Schlüsselworterkennung . . . 40
2.5.1 Binäre Suche . . . 41
2.5.1.1 Tabelle . . . 43
2.5.1.2 Zeichenklassen für Buchstaben der Schlüsselwörter . . . 43
2.5.1.3 Extra Zeichenklasse für 1. Buchstaben von Schlüsselwörtern . . . 44
2.5.2 Automatentabelle . . . 44
2.5.3 Schlüsselworttabelle mit einfacher Hashtechnik . . . 44

3 Parser **46**
3.1 Grundlagen . . . 46
3.1.1 Kellerautomat . . . 46
3.2 LL(1)-Grammatik . . . 47
3.2.1 First/Follow . . . 47
3.2.2 Als Graph . . . 48
3.2.2.1 programm . . . 48
3.2.2.2 block . . . 48
3.2.2.3 statement . . . 48
3.2.2.4 expr . . . 49
3.2.2.5 term . . . 49
3.2.2.6 factor . . . 49
3.2.2.7 condition . . . 50
3.2.2.8 Bewertung . . . 50
3.3 Implementation von Graphen . . . 51
3.3.1 Struktur eines Bogens . . . 52
3.3.2 Erstellen eines Graphen . . . 52
3.3.3 Implementation von Bögen in Graphen . . . 53
3.3.4 Parsebaum (optional) . . . 54
3.3.5 Erweiterte Graphen mit Backtracking . . . 57

4 Namensliste **60**
4.1 Grundlagen . . . 60
4.2 Namenslisteneintrag . . . 60
4.3 Variablenbeschreibung . . . 61
4.4 Konstantenbeschreibung . . . 61
4.5 Prozedurbeschreibung . . . 62
4.6 Beispiel Namensliste . . . 62
4.7 Funktionen zur Namensliste . . . 64
4.8 Einordnung in Parser . . . 64

5 Virtuelle Maschine **66**
5.1 Grundlagen . . . 66
5.1.1 Dreiadressmaschine . . . 66
5.1.2 Zweiadressmaschine . . . 66
5.1.3 Einadressmaschine . . . 67
5.1.4 Nulladressmaschine - Stackmaschine . . . 67
5.2 Befehlssatz . . . 67
5.2.1 Arbeitsweise der Stackmaschine . . . 68
5.3 Datenstrukturen der VM . . . 68
5.4 Register der VM . . . 69
5.5 Prozedurtabelle . . . 69
5.6 Initialisierung . . . 69
5.7 Steuerschleife . . . 70
5.8 Prozeduraufruf . . . 71
5.9 Rekursion . . . 73
5.10 Prozedur mit Parametern . . . 73

6 Codegenerierung **74**
6.1 Grundlagen . . . 74
6.2 Funktionen zur Codegenerierung . . . 74
6.3 Variablen zur Codegenerierung . . . 75
6.3.1 EntryProc . . . 76
6.3.2 programm . . . 76
6.3.3 block . . . 76
6.3.4 statement . . . 78
6.3.5 expression . . . 78
6.3.6 term . . . 79
6.3.7 factor . . . 79
6.3.8 condition . . . 80
6.3.9 Auszuführende Funktionen in statement . . . 80
6.3.9.1 Zuweisungen . . . 80
6.3.9.2 conditional statement . . . 81
6.3.9.3 loop statement . . . 82
6.3.9.4 procedure call . . . 82
6.3.9.5 Eingabe/Ausgabe . . . 82

7 Beispielcompilierung **83**
7.1 Anzulegende Speicherbereiche . . . 83
7.2 Abarbeitung . . . 83

8 lex/flex **84**
8.1 Grundlagen . . . 84
8.1.1 Verwendung . . . 84
8.1.2 Reguläre Ausdrücke . . . 85
8.1.3 Besondere Zeichen . . . 85
8.1.4 Wiederholungen . . . 85
8.1.5 Reihenfolge der Pattern . . . 86
8.2 Aufbau einer flex-Quelldatei . . . 86
8.2.1 Rules division . . . 86
8.2.2 Vordefinierte Symbole . . . 87
8.3 Beispiele . . . 87
8.3.1 Beispiel Leerzeichen . . . 87
8.3.2 Zeichen/Zeilen zählen . . . 88
8.3.3 Wörter zählen . . . 88
8.3.4 Beispiel Regeln . . . 89
8.4 Lexer für PL/0 Compiler . . . 89

9 yacc **92**
9.1 Grundlagen . . . 92
9.1.1 Aufbau einer yacc-Datei . . . 92
9.1.2 Zusammenspiel lex – yacc . . . 92
9.2 yacc-Teile . . . 93
9.2.1 Definitionsteil . . . 93
9.2.2 Regelteil . . . 93
9.2.3 Beispiel 1 . . . 93
9.2.4 Beispiel 2: expr, term, factor . . . 94
9.3 Präzendenzregeln . . . 96
9.3.1 Ausnahme . . . 96
9.3.2 Beispiel . . . 96
9.4 pl/0 mit yacc und lex . . . 97
9.4.1 pl0.y . . . 98
9.4.1.1 Vereinbarungsteil . . . 98
9.4.1.2 Funktionenteil . . . 98
9.4.1.3 Regelteil . . . 98
9.4.2 Lexer . . . 99
9.4.2.1 Funktionenteil . . . 100
9.4.3 Regelteil . . . 100
9.4.4 Die Funktionen . . . 101

10 Tabellengesteuerte Verfahren **103**
10.1 Grundlagen . . . 103
10.2 Schrittweise Aufbau eines top-down Parsers für Ausdrücke . . . 103
10.2.1 Umgeformte Regeln . . . 104
10.2.2 Bestimmung der terminalen Anfänge . . . 104
10.2.3 Aufbau der Tabelle . . . 104
10.2.4 Der Algorithmus . . . 104
10.2.5 Beispiel . . . 105
10.3 bottom-up Analyse . . . 106
10.3.1 shift/reduce . . . 106
10.3.2 Automatentabelle . . . 106
10.3.3 Beispiel . . . 106

HINWEISE

Zugelassene Hilfsmittel Klausur: Spickzettel A-4 Blatt, doppelseitig
Wenn PL/0 compiler zum Ende des Semsters fertig: 10% der Klausurpunkte Bonus.
Mögliche Klausurthemen:

- Vorgegebener Lexer soll zusätzliche Elemente erkennen. Tabellen vorgegeben.
- PL/0 entsprechend erweitern. Bspw. mit Array, else-Zweig, do-while, repeat-until, Parameterliste (im Stack von unten aufsteigend; darüber Rücksprungadresse, ProcIndex und Varanfang; gefolgt von Relativadressen der Variablen [entsprechend Namensliste: 4 Byte]; Zugriff der Parameter dann relativ ab Adresse -16).
  Dafür: Graphen malen, Bögen mit Semantikroutinen.
- GGf. rekursiver Abstieg o.ä. Ggf. EBNF (linksrekursive Regel aufstellen usw.).

Gegeben von Prof. Beck: ASCII-Code Tabelle, Zwischencode (code.h)

LEHRINHALTE

- Einführung in die Begriffswelt der Theorie der formalen Sprachen
- Strategien der Analyse
- Aufbau von Compilern/Interpretern
- Arbeitsweise höherer Programmiersprachen
- Arbeiten mit Automaten und Graphen und deren programmiertechnische Umsetzung

1 EINFÜHRUNG

1.1 MOTIVATION

Implementation programmiersprachlicher Elemente in konventionellen Programmen, beispielsweise Ausdrucksberechner

Fördert das Verständnis von Programmiersprachen.

Methoden, Techniken und Algorithmen des Compilerbaus sind bei der Lösung auch anderer Probleme oft hilfreich.(Rekursion, Bäume, Tabellen, Listen, Graphen)

1.1.1 DER COMPILER

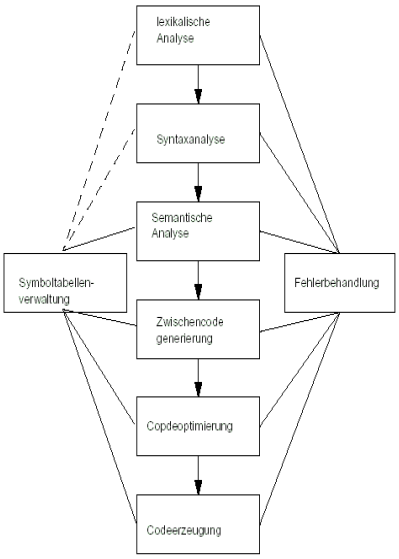
Klassische Compiler haben die Aufgabe, ein Programm einer Sprache in ein äquivalentes Programm einer anderen Sprache zu überführen.

Höhere Programmiersprache -> Maschinencode
Höhere Programmiersprache -> Assemblersprache
Höhere Programmiersprache -> Zwischencode
Höhere Programmiersprache -> Höhere Sprache

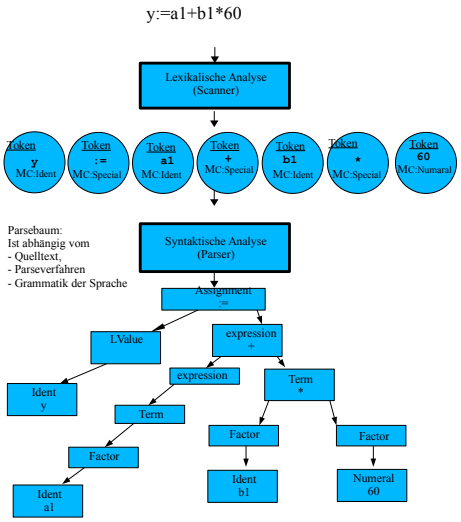
Auch zur Konvertierung oder Interpretation von Daten kommen Verfahren der Übersetzertechnik zum Einsatz, beispielsweise bei XML-Parsern.

Einsatz in intelligenten Editoren

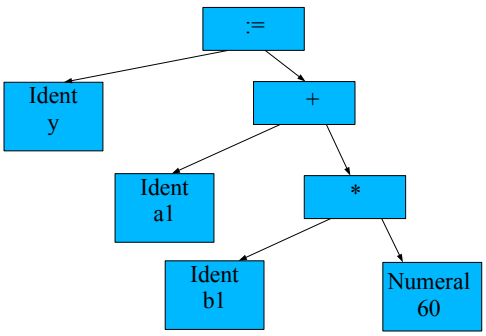
1.1.1.1 AUFBAU EINES COMPILERS

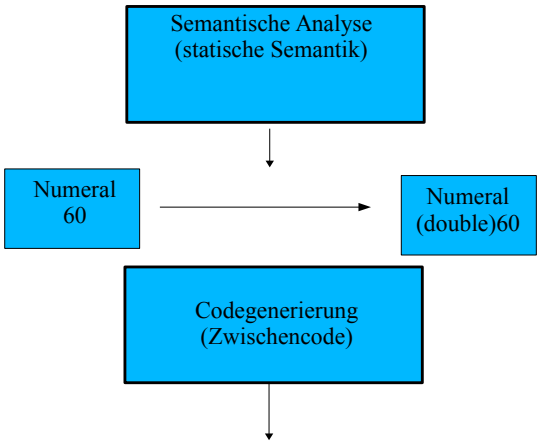


1.1.1.2 BEISPIEL



ABSTRAKTER SYNTAXBAUM





1.1.2 DREIADRESSMASCHINE VS STACKMASCHINE

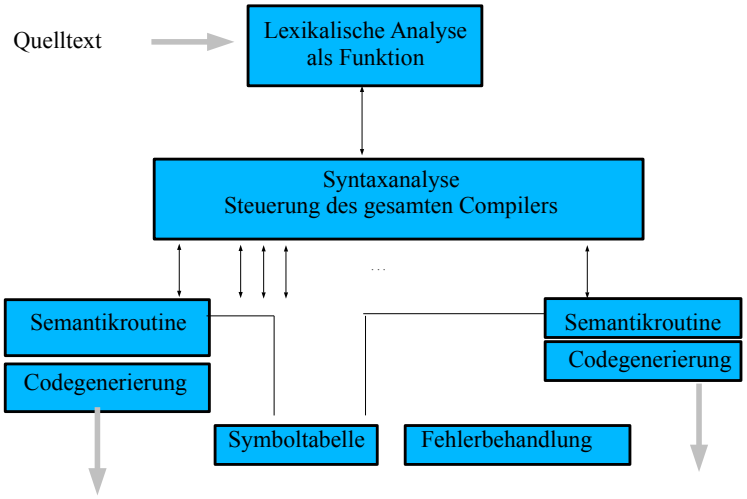
Dreiadressmaschine	Stackmaschine
temp1:=inttoreal(60)	pushAdr y
temp2:=var3*temp1	pushVal a1
temp3:=var2+temp2	pushVal b1
var1 := temp3	pushConst 60.0
	mul
	add
	store

Codeoptimierung
(Zwischencode)

Dreiadressmaschine	Stackmaschine
temp1:=var3*inttoreal(60)	push 60.0
Var1 :=var2+temp1	mul b1
	add a1
	Store y

Maschinencodegenerierung
(Assemblercode)

1.1.3 SYNTAXGESTEUERTER EINPASSCOMPILER



1.2 GRUNDLAGEN

Sprache	Menge aller Sätze
Alphabet	terminales / nicht terminales
terminales A.	Zeichen aus denen die Sätze der Sprache bestehen
nicht terminales A.	Hilfszeichen zum Bilden von Regeln
Zeichenkette	Aneinanderreihung von Zeichen aus A
#s	Länge der Zeichenkette
Ableitung	**`s=>t`**, wenn **`s->s1->s2-> ... sn->t`**
direkte Abl.	s= s1 s2 s3, t= s1 t2 s3, S2 -> t2
Grammatik	G = (T,N,s,P)

1.2.1 SPRACHE

- Sprache dient der Kommunikation, Programmiersprachen dienen der Kommunikation zwischen Mensch und Maschine, der Kommunikation zwischen Menschen (Metasprachen) und auch zwischen Maschinen zB. Postscript, XML, ...
- Es gibt natürliche Sprachen und künstliche Sprachen.
- **Sprache ist die Menge von Sätzen, die sich aus einer definierten Menge von Zeichen unter Beachtung der Syntaxregeln bilden lassen.**

1.2.2 ALPHABET

- **Die Menge von Zeichen aus denen die Sätze, Wörter oder Satzformen einer Sprache bestehen, bildet das Alphabet (A oder V).**
- Ein Zeichen kann nicht nur Buchstabe oder Ziffer sein, auch ein Wort im umgangssprachlichen Sinn wird als Zeichen aufgefasst (Wortsymbole, wie if, while, ... oder Bezeichner und Zahlen oder zusammengesetzte Symbole wie <=).
- **Terminales Alphabet (T)**:
  Die Menge der Zeichen, aus denen die Sätze der Spache gebildet werden, man bezeichnet diese auch als Atome, Morpheme, Token.
- **Nichtterminales Alphabet (N)**:
  Die Menge der Metasymbole, die zur Bildung syntaktischer Regeln benötigt werden.

BEISPIEL

Alphabet zur Bildung gebrochener Dezimalzahlen

```
T = {'0', '1', '2', '3', '4', '5',
     '6', '7', '8', '9', '-', '.' }
```

```
N = {ZAHL,
     GANZZAHL_TEIL,
     GEBROCHENZAHL_TEIL }
```

1.2.3 ZEICHENKETTE

Eine Aneinanderreihung von Zeichen eines Alphabetes wird **Zeichenkette** oder **Wort** genannt. Sie kann sowohl Elemente des terminalen, als auch des nichtterminalen Alphabetes enthalten und ist von einer Zeichenkette oder String im Sinne von C oder Pascal zu unterscheiden.

Bsp.: 12345.GEBROCHENZAHL_TEIL
GANZZAHL_TEIL.GEBROCHENZAHL_TEIL
GANZZAHL_TEIL.77
12345.77

1.2.4 FREIE HALBGRUPPE (A*)

- Die Menge aller Zeichenketten, die aus den Zeichen des Alphabetes bildbar sind, wird freie Halbgruppe genannt, sie enthält auch die leere Zeichenkette.
- Eine Sprache ist eine Untermenge der freien Halbgruppe, es existieren Regeln, die festlegen, welche Zeichenketten gültige Sätze der Sprache sind.

1.2.5 SYNTAXREGELN

- Die Menge aller Zeichenketten, die aus den Zeichen des Alphabetes bildbar sind, wird freie Halbgruppe genannt, sie enthält auch die leere Zeichenkette.
- Eine Sprache ist eine Untermenge der freien Halbgruppe, es existieren Regeln, die festlegen, welche Zeichenketten gültige Sätze der Sprache sind.

1.2.6 GRAMMATIK UND PARSER

LL(k)-Grammatik bildet die Grundlage für einen LL(k)-Parser, der die Eingabezeichen von links liest und immer das am weitesten links stehende Metasymbol ersetzt (Linksableitung). LL(k)-Parser arbeiten nach der Strategie top down

LR(k)-Grammatik bildet die Grundlage für einen LR(k)-Parser, der die Eingabezeichen von links liest und immer die am weitesten rechts stehenden Metasymbole ersetzt (Rechtsreduktion). LR(k)-Parser arbeiten nach der Strategie bottom up.

(Subjekt Praedikat Artikel Substantiv → Subjekt Praedikat Objekt)

Es wird immer die Vorausschau um k Zeichen benötigt, um die richtige alternative Regel zu finden.

1.2.7 BNF

Backus Naur Form oder Backus Normalform

Entstand im Rahmen der Entwicklung von Algol 60

Einführung der Metasymbole (Nicht Terminale Symbole) als linke Seiten der Regeln

Metasymbole werden häufig in '<' '>' gesetzt.

Linke und rechte Seite der Regeln werden durch das Ableitungssymbol '::=' getrennt

Regeln mit gleichen linken Seiten werden zusammengefasst, die Alternativen werden durch das Symbol '|' getrennt

Es gibt eine Reihe von Modifikationen, wie das Weglassen der spitzen Klammern, Anfügen eines Punktes am Regelende oder andere Definitionssymbole als '::='.

- Eignet sich zur Darstellung von Symbolfolgen und Alternativen
- Listen werden durch Rekursion formuliert
- BNF bildet die Grundlage für viele Parsergeneratoren

BEISPIEL

```
<const_def_list>   ::=  'const' <const_list> ';'
<const_list>       ::=  <const_def>
                   |    <const_list>','<const_def>
<const_def>        ::=  <const_name>=<const_symbol>
<const_name>       ::=  name
<const_symbol>     ::=  name | zahl | zeichen
```

Quelltextausschnitt aus pl0.y (yacc)

```
ConstDecl: T_CONST constList ';'
   |        ;
ConstList: constList ',' T_Ident '=' T_Num {AcreateConst($5,$3);};
   |       T_Ident '=' T_Num               {AcreateConst($3,$1);};
```

```
Const X=0, eins=1;
```

1.2.8 EBNF

- Eingeführt durch Niklaus Wirth im Rahmen der Definition von Pascal
- Darstellung optionaler Elemente durch Einführung eckiger Klammern '[' ']'
- Darstellung von Wiederholungen durch Einführung geschweifter Klammern '{' '}', die geklammerte Zeichenkette kann 0 mal, 1 mal oder beliebig oft auftreten
- Deutlich besser lesbar, durch top-down-Parser auch sehr elegant umsetzbar.

EBNF:

compound_stmt ::= "BEGIN" statement {";" statement} "END".

BNF (yacc)

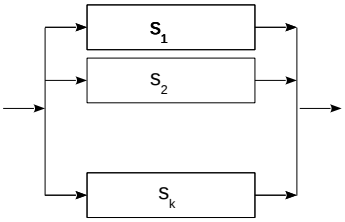
```
statementList:    statementList ';' statement ;
                |   statement ;
                |   ;
compound_stmnt:   T_BEGIN statementList T_END;
```

1.2.9 SYNTAXGRAPH

Darstellung eines Nicht Terminals:

Darstellung eines Terminals:

Darstellung von Alternativen:

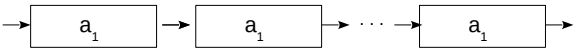


$A \rightarrow s_1 \mid s_2 \mid .... s_k$

Dabei sind s_i Teilgraphen, bestehen aus Termen, wie nachfolgend

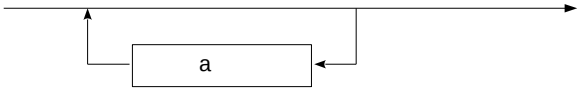
Sequenz

$S = a_1\ a_2\ ...a_k$



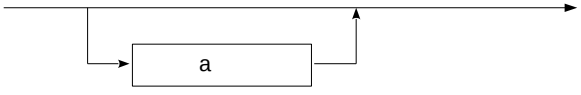
Iteration

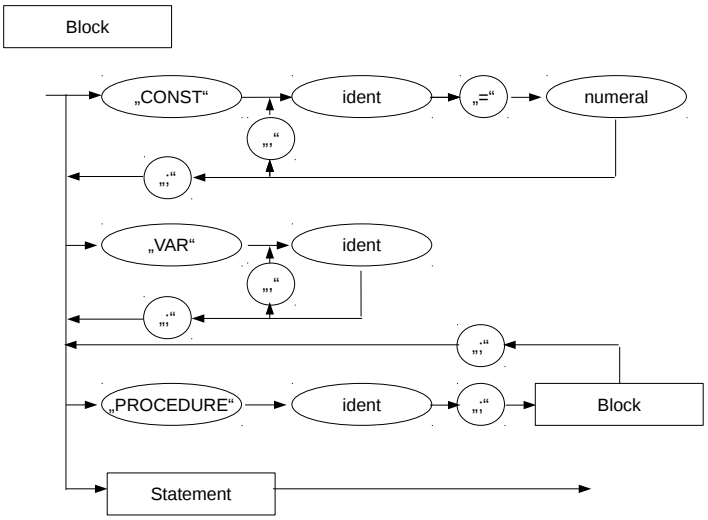
S = {a}



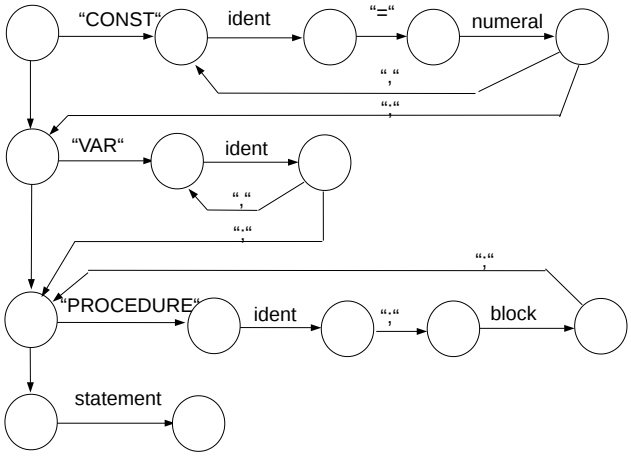
Optional

S = [a]





block:



1.2.10 GRAMMATIK VON PL/0

T	+ | - | * | / | := | , | . | ; | (|) | ? | ! | # | = | < | > | >= | <= |
	BEGIN | CALL | CONST | DO | If | ODD | PROCEDURE | VAR | WHILE |
	numeral| ident
N	program, block, statement, condition, expression, term, factor
s	Programm
P	→ next Page

```
programm    = block ".".
block       = ["CONST" ident=num{"," ident=num}";"]
              ["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        =faktor { ("*"|"/") faktor }.

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

BEISPIELE

```
var U,P,I,R;

procedure ProcP;
P:=7*U*U/R/10;

procedure ProcI;
I:=U/R;

begin
  ?U;
  ?R;
  if R>0 then
  begin
  call ProcI;
  call ProcP;
  ! I;
  ! P
  end
end.
```

```
const c=0,d=1;
var a,
    x,
    y,
    b;

begin
  ?a;
  ?b;
  x:=0;
  while x<=a do
  begin
  y:=0;
    while y<=b do
    begin
      !x*y;
      y:=y+1
    end;
    x:=x+1
  end
end.
```

```
var f,x;

procedure fakult;
var xl;
begin
  xl:=x;
  x:=x-1;
  if x>0 then
  begin
    call fakult;
    f:=f*xl
  end
end;

begin
  ?x;
  f:=1;
  call fakult;
  !f
end.
```

1.3 GRAMMATIK NACH CHOMSKY

G = (T,N,s,P) mit
T: terminales Alphabet
N: nichtterminales Alphabet
s: Startsymbol, Axiom
P: Menge der Regeln

BEISPIEL

```
N= { SATZ, SUBJEKT, PRAEDIKAT, OBJEKT, ARTIKEL, VERB,
     SUBSTANTIV}
T= {der, den, hund, briefträger, beisst}

R= {
   SATZ         -> SUBJEKT PRAEDIKAT OBJEKT(1)
   SUBJEKT      -> ARTIKEL SUSTANTIV       (2)
   OBJEKT       -> ARTIKEL SUSTANTIV       (3)
   PRÄDIKAT     -> VERB                    (4)
   SUBSTANTIV   -> hund                    (5)
   SUBSTANTIV   -> briefträger             (6)
   VERB         -> beisst                  (7)
   ARTIKEL      -> der                     (8)
   ARTIKEL      -> den                     (9)
   }

Startsymbol: Satz
```

1.3.1 ABLEITUNG

- Ableitung von li nach re wird auch Linksableitung, andersherum Rechtsableitung genannt.
- Linksableitung: Es wird das am weitesten links stehende NichtTerminal untersucht
- Rechtsableitung: es wird das am weitesten rechts stehende NichtTerminal untersucht
- Ein String t heißt ableitbar aus dem String s, wenn es eine Folge direkter Ableitungen in folgender Form gibt:
  s=>t, wenn **s->s1->s2-> ... sn->t**

  SATZ => der hund beißt den briefträger

DIREKTE ABLEITUNG

Ein String **t** heißt direkt ableitbar aus **s (s->t),** wenn t durch Anwendung einer einzigen Regel aus **s** ableitbar ist.

s= s1 s2 s3
t= s1 t2 s3
Es muß eine Regel ***s2 -> t2*** geben.

Satzform: Eine Satzform zu einer Grammatik G ist ein String, der aus einem Startsymbol (Axiom) ableitbar ist, sie kann terminale und nichtterminale Symbole enthalten.

Satz: Ein Satz ist eine Satzform, die nur terminale Symbole enthält.

Sprache: Die Menge der durch eine Grammatik G erzeugbaren Sätze heißt die durch G erzeugte Sprache.

1.3.2 CHOMSKY TYP 0

Die Produktionen **s -> t** mit **s,t ϵ A*** unterliegen keinerlei Einschränkungen. Produktionen der Form **s -> ε** sind erlaubt.
Typ 0-Grammatiken haben für praktische Belange kaum Bedeutung.

1.3.3 CHOMSKY TYP 1

Die Produktionen sind von der Form **s –> t** mit **s,t ∈ A*** und **#s<=#t** (# steht für Länge der Zeichenkette). Die bei der Ableitung entstehenden Satzformen stellen eine monoton länger werdende Folge dar.
Diese Grammatiken heißen nichtkontrahierend, oder kontextsensitiv. Ihre Produktionen können die Form

sAt–>sat mit **A∈N** und **a∈A*** und **#a>0** annehmen.

1.3.4 CHOMSKY TYP 2

Die Produktionen haben die Form **A–>s** mit **A∈N** und **s∈A***. Bei der Ableitung werden einzelne Nichtterminale durch eine Zeichenkette aus **A*** ersetzt. Grammatiken vom Typ 2 heißen kontextfreie Grammatiken, da sie nichtterminale Zeichen in einem String ohne ihren Kontext zu beachten, ersetzen. Die Syntax von Programmiersprachen wird typischer Weise durch kontextfreie Grammatiken beschrieben. Zur Analyse kontextfreier Sprachen benutzt man einen Kellerautomaten.

1.3.5 CHOMSKY TYP 3

Die Produktionen haben die Form

A–>a|Ba oder **A–>a|aB** mit **A,B∈N** und **a∈T**.

Diese Grammatiken heißen reguläre oder einseitig lineare Grammatiken, sie erlauben nicht die Beschreibung von Klammerungen. Sie werden benötigt, um die Morpheme der Programmiersprachen zu beschreiben. Zur Analyse regulärer Sprachen benutzt man den endlichen Automaten.

1.3.6 CHOMSKY TYP 4

Diese Grammatiken bilden endliche Sprachen, sie enthalten keine Rekursionen.
Endliche Sprachen sind für den Compilerbau von untergeordneter Bedeutung.

1.3.7 BEISPIEL

```
N={SATZ,SUBJEKT,PRÄDIKAT,OBJEKT,ARTIKEL,VERB
   ,SUBSTANTIV}
T= {der, den, hund, briefträger, beißt}
s= SATZ
R= {
   SATZ       -> SUBJEKT PRÄDIKAT OBJEKT (1)
   SUBJEKT    -> ARTIKEL SUBSTANTIV      (2)
   OBJEKT     -> ARTIKEL SUBSTANTIV      (3)
   PRÄDIKAT   -> VERB                    (4)
   SUBSTANTIV -> hund                    (5)
   SUBSTANTIV -> briefträger             (6)
   VERB       -> beißt                   (7)
   ARTIKEL    -> der                     (8)
   ARTIKEL    -> den                     (9)
   }
```

ANALYSEN

Satzform	Regel

Satz	
SUBJEKT PRÄDIKAT OBJEKT	1
ARTIKEL SUBSTANTIV PRÄDIKAT OBJEKT	2
der SUBSTANTIV PRÄDIKAT OBJEKT	8
der hund PRÄDIKAT OBJEKT	5
der hund VERB OBJEKT	4
der hund beißt OBJEKT	7
der hund beißt ARTIKEL SUBSTANTIV	3
der hund beißt den SUBSTANTIV	9
der hund beißt den briefträger	6

Satzform	Regel

der hund beißt den briefträger	
ARTIKEL hund beißt den briefträger	8
ARTIKEL SUBSTANTIV beißt den briefträger	5
SUBJEKT beißt den briefträger	2
SUBJEKT VERB den briefträger	7
SUBJEKT PRÄDIKAT den briefträger	4
SUBJEKT PRÄDIKAT ARTIKEL briefträger	9
SUBJEKT PRÄDIKAT ARTIKEL SUBSTANTIV	6
SUBJEKT PRÄDIKAT OBJEKT	2
SATZ	1

1.3.8 ANALYSESTRATEGIEN

Top down

Ausgehend vom Startsymbol werden Metasymbole durch Anwendung von Regeln ersetzt. Dabei geht man von links nach rechts vor. Es wird immer das am weitesten links stehende Metasymbol ersetzt. (Linksableitung)

Bottom up

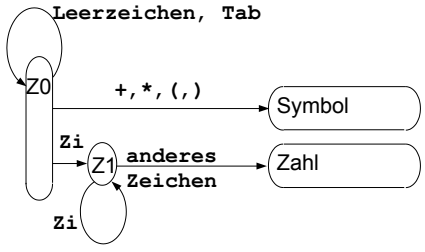
Ausgehend vom zu analysierenden Satz wird durch Reduktion versucht, das Startsymbol herzuleiten. Es wird immer versucht, ausgehend vom am weitesten rechts stehenden Metasymbol an Hand der Regeln soviel wie möglich zu reduzieren. (Rechtsreduktion)

1.4 MORPHEME, TOKEN, ATOME

Sind die kleinsten Bedeutung tragenden Bestandteile einer Programmiersprache (Zahlen, Symbole, wie +, - .., aber auch Wortsymbole und Bezeichener)
Eigenschaften sind:
•Morphemtyp
•Morpheminhalt
•Morphemstatus (bereits verarbeitet?)
•Morphemlänge, Morphemposition ...

1.4.1 MORPHEME EINFACHER AUSDRÜCKE

T : 0,1,2,3,4,5,6,7,8,9,+,*,(,)
N : Morphem, Zahl, Sy, Zi
s : Morphem
R : Morphem ::= Zahl | Sy
Zahl ::= Zi | Zi Zahl
Zi ::= 0,1,2,3,4,5,6,7,8,9
Sy ::= +,*,(,)



BEISPIEL 3+5*12

Zahl, 3
Symbol, '+'
Zahl, 5
Symbol, '*'
Zahl, 12

Prgramm (-teil), das einen Quelltext in seine Morpheme zerlegt, nennt man Lexer oder Scanner

1.4.2 EIN EINFACHER LEXER

```
typedef struct morph{
  int mc;
  double dval;
  char cval;
}MORPHEM;

enum mcodes{
  mempty,
  mop,
  mdbl
};

static MORPHEM m;

void lex(char * pX){
  static char * px;
  /* Initialisierung*/
  if (pX!=NULL){
    m.mc=mempty;
    px=pX;
  }
  /* lexiaklische Analyse */
  if (*px=='\0'){
    m.mc=mempty;
  } else {
    for (;*px==' '||*px=='\t';px++);
    if (isdigit(*px) || *px=='.'){
      m.dval=strtod(px,&px);
      m.mc =mdbl;
    } else
      switch (*px){
        case '+':
        case '':
        case '*':
        case '/':
        case '(':
        case ')':
```

```
38            m.cval=*px++;
39            m.mc=mop;
40            break;
41          default :
42            printf("wrong ...: %c\n-- canceling --\n",*px);
43            exit (1);
44        }
45    }
46 }
```

1.5 GRAMMATIK 1: EINFACHE AUSDRÜCKE

N : Faktor, Term, Expression
T : Zahl,'+','*','(',')'
s : Expression
R : Expression ::= Expression '+' Term | Term
Term ::= Term '*' Faktor | Factor
Faktor ::= Zahl | '(' Expression ')'

1.5.1 REKURSIVER ABSTIEG

- Eine Funktion für jedes Nichtterminal
- Ein Nichtterminal auf der rechten Seite einer Regel bewirkt den Aufruf der zugehörigen Funktion
- Ein Terminalsymbol wird mit dem lookahead verglichen und bei Übereinstimmung verarbeitet (aus dem Eingabestrom entfernt – eat, Aufruf des Lexers um das nächste Token bereitzustellen)

ALTERNATIVE REGELN

```
1  if (lookahead == First(A1)){
2    A1();
3  } else if (lookahead == First(A2)) {
4    A2();
5  }
6  /* ... */
7  else if (lookahead == First(An)){
8    An();
9  } else {
10   error();
11 }
```

First ist das jeweils erste terminale Symbol einer alternativen Regel

1.5.2 ANFORDERUNGEN

- Keine Linksrekursion, weil eine solche bei diesem Verfahren zur Endlosrekursion führen würde.
- An Hand des gerade aktuellen Tokens muss bei alternativen Regeln entschieden werden können, welche Regel auszuwählen ist.
- Die geforderten Bedingungen werden von LL(1) Grammatiken erfüllt

1.6 LL(1) GRAMMATIKEN

- Sonderform der LL(k) Grammatiken
- Eingabe wird von Links gelesen, Regeln werden von Links bearbeitet (Regeln sind Linkslinear/Rechtsrekursiv)
- k steht für k Zeichen Lookahaed um eindeutig bestimmen zu können, welche Regel als nächstes anzuwenden ist.
- Um dies zu prüfen, werden die Mengen **first** und **follow** zu jedem Nichtterminal bestimmt.

- First: Die Menge der terminalen Anfänge aller Zeichenketten, die aus dem Nichtterminalen abgeleitet werden können. Die Firstmengen alternativer Regeln müssen zueinander disjunkt sein.

  $A \rightarrow a|b$

  $First(a) \cap First(b) = 0$

- Kann aus einem Nichtterminal die leere Menge abgeleitet werden, so müssen die Followmengen zusätzlich bestimmt werden.
- Follow: Die Menge der terminalen Anfänge, die auf das betrachtete Nichtterminal folgen können. Die Firstmengen alternativer Regeln und die Followmenge müssen zueinander disjunkt sein.

  $Fist(a) \cap First(b) \cap Follow(A) = 0$

1.7 GRAMMATIK 2: EINFACHE AUSDRÜCKE OHNE LINKSREKURSION

N : Faktor, Term, Expression
T : Zahl,'+','*','(',')'
s : Expression
R : Expression ::= Term | Term '+' Expression
Term ::= Faktor | Faktor '*' Term
Faktor ::= Zahl | '(' Expression ')'

Oder nach **Linksfaktorisierung**

1.7.1 LINKSFAKTORISIERUNG

StatementList → statement ';' statementList
| statement
StatementList → statement statementList2
StatementList2 → ';' statementList
| ε

AN GRAMMATIK

R : Expression ::= Term | Term '+' Expression Term ::= Faktor | Faktor '*' Term Faktor ::= Zahl | '(' Expression ')'

R' : Expression ::= Term (ε | '+' Expression) Term ::= Faktor (ε | '*' Term) Faktor ::= Zahl | '(' Expression ')'

R'': Expression ::= Term Expression' Expression' ::= + Expression | ε Term ::= Faktor Term' Term' ::= * Term | ε Faktor ::= Zahl | '(' Expression ')'

1.7.2 FIRST/FOLLOW

	First	Follow
Expression	Zahl, (	$,)
Expression'	+, ε	$,)
Term	Zahl, (	$,), +
Term'	*, ε	$,), +
Factor	Zahl, (	$,), +, *

1.7.3 EXPRESSION

```
double expr(void){
  double tmp=term();
  if (m.mc==mop && m.cval=='+'){
    lex(NULL);
    tmp+=expr();
  }
  return tmp;
}
```

1.7.4 TERM

```
double term(void){
  double tmp=fac();
  if (m.mc==mop && m.cval=='*'){
    lex(NULL);
    tmp*=term();
  }
  return tmp;
}
```

1.7.5 FAKTOR

```
double fac(){
  double tmp;
  if (m.mc==mop)  {
    if (m.cval=='(')  {
      lex(NULL);
      tmp=expr();
      if (m.mc != mop || m.cval != ')')
        exit (1);
      lex(NULL);
    } else
      exit (1);
  } else
    if (m.mc==mdbl){
      tmp=m.dval;
      lex(NULL);
    } else
      exit (1);
  return tmp;
}
```

1.7.6 MAIN

```
int main(int argc, char*argv[]){
  char *pBuf=argv[1];
  printf("%s\n",pBuf);
```


```
4     lex(pBuf);
5     printf("%10.4f\n",expr());
6     free(pBuf);
7     return 0;
8   }
```

1.7.7 AUSFÜHREN

```
1 beck@linux:~/COMPILER> ./tPM 5+2*3
2 5+2*3
3 11.0000
4 aber:
5 beck@linux:~/COMPILER>./tPM 12/2*3
```

→ Grammatik rechnet nicht korrekt!

1.8 UMFORMUNG DER GRAMMATIK

Vorlesung
25.10.2017

Die Umformung der Grammatik G1->G2 liefert eine Grammatik, die die Syntax korrekt erkennt, aber die Semantik verletzt.

Durch Einführung der Rechtsrekursion werden die Ausdrücke nun auch von rechts nach links bewertet und ausgerechnet.

12/2*3

1. 2*3 → 6

   12/6 → 2

Die Umformung der Grammatik ist also auf diese Weise unzulässig

1.8.1 GRAMMATIK 3: BESEITIGUNG DER LINKSREKURSION DURCH ITERATION

N : Faktor, Term, Expression
T : Zahl,'+','*','(',')'
s : Expression
R : Expression ::= Term {'+' Term}
Term ::= Faktor {'*' Factor}
Faktor ::= Zahl | '(' Expression ')'

1.8.2 EBNF

- Erweiterung der BNF durch nachfolgende Konstrukte:
- {…} Die in geschweiften Klammern angegebene Zeichenkette kommt 0 mal, einmal, oder mehrfach vor.
- […] Die in eckigen Klammern angegebene Zeichenkette kommt optional einmal vor

1.8.3 EXPRESSION ALS C-CODE

Expression ::= Term {'+' Term}

```
double expression(void)
{
 double tmp;
 tmp=term();

 while (m.mc==mop && (m.cval=='+')
 {
  if (m.cval=='+' ) {lex(NULL); tmp +=term();}
 }
 return tmp;
}
```

1.8.4 BESEITIGUNG DER LINKSREKURSION BEI KLASSISCHER BNF

A → A a | b	Expr → Expr '+' Term | Term	A: Expr
		a: '+' Term
		b: Term
A → b A'	Expr → Term Rexpr	A': Rexpr
A' → a A'|ε	Rexpr → '+' Term Rexpr | ε	

1.9 GRAMMATIK FÜR EINFACHE AUSDRÜCKE

N : Faktor, Term, Expression
T : Zahl,'+','*','(',')'
s : Expression
R : Expression ::= Term Rexpr
Rexpr ::= '+' Term Rexpr
| ε
Term ::= Faktor Rterm
Rterm ::= '*' Factor Rterm
| ε
Faktor ::= Zahl
| '(' Expression ')'

1.10 FIRST/FOLLOW

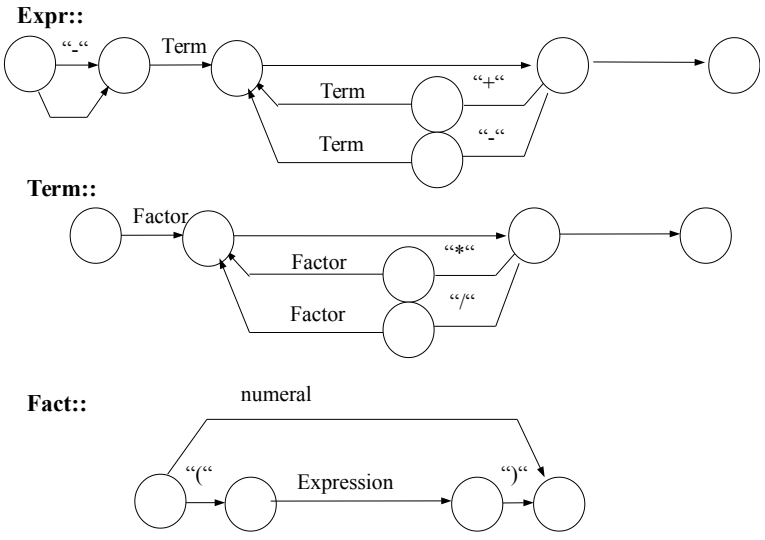
	First	Follow
Expression	Zahl, (	$,)
Rexpr	+, ε	$,)
Term	Zahl, (	$,), +
Rterm	*, ε	$,), +
Factor	Zahl, (	$,), +, *

1.10.1 IMPLEMENTATION

```
double expr(void)
{
  double tmp=term();
  return Rexpr(tmp);
}

double Rexpr(double tmp)
{
   if (m.cval=='-' && m.mc==mop )
            {lex(NULL); tmp -=term();tmp=Rexpr(tmp);} else
   if (m.cval=='+' && m.mc==mop )
             {lex(NULL); tmp+=term();tmp=Rexpr(tmp);}
  return tmp;
}
```

Hier übergeben wir den vorher in term berechneten Wert

SYNTAXGRAPH



1.11 WIEDERHOLUNG

Sprache	Menge aller gültigen Sätze zu einem Alphabet
Alphabet	terminales / nicht terminales
terminales A.	Zeichen aus denen die Sätze der Sprache bestehen
nicht terminales A.	Hilfszeichen zum Bilden von Regeln
Zeichenkette	Aneinanderreihung von Zeichen aus A
#s	Länge der Zeichenkette
Ableitung	**s=>t**, wenn **s->s1->s2-> ... sn->t**
direkte Abl.	s= s1 s2 s3, t= s1 t2 s3, S2 -> t2
Grammatik	G = (T,N,s,P)

2 LEXER

2.1 LEXIKALISCHE ANALYSE

Aufgaben:

Lesen des Eingabetextes und Zerlegung in die einzelnen Zeichen im Sinne der Grammatik der Programmiersprache(Morpheme oder auch Atome oder Lexeme).

Morpheme können aufeinander folgen (a+b*77) oder durch Trennzeichen voneinander getrennt sein (call f1)

Eliminierung von irrelevanten Textteilen (Kommentare, Leerzeichen...)

Konvertierungen

Erkennung von Wortsymbolen und Ersetzung durch besser handhabbare Symbolcodes

Die Aufgabenteilung von Lexer und Parser ist verschieblich. So kann die Erkennung zusammengesetzter Symbole durch den Scanner, aber auch von der Syntaxanalyse realisiert werden. Auch der Aufbau von Symboltabellen wird mitunter schon von der lexikalischen Analyse vorbereitet.[1]

2.1.1 EINORDNUNG IN DEN COMPILER

Die lexikalische Analyse kann als

- selbständiger Compilerpass,
- als paralleler Prozess oder als
- Unterprogramm der Syntaxanalyse

oranisiert sein. Bei Einpasscompilern, die im Rahmen der Vorlesung vorrangig diskutiert werden sollen, soll die lexikalische Analyse ein Unterprogramm sein, das jeweils das nächste Morphem liefert.

2.1.2 FUNKTIONSWEISE

Der Lexer selbst arbeitet in zwei Schritten.

Schritt 1:

Aus dem Eingabestrom werden die Zeichen gelesen und in einem Puffer alle zum Morphem gehörenden Zeichen gesammelt. Hierzu wird ein Automat verwendet.

Schritt 2:

Nachbereitung in Abhängigkeit des letzten Zustandes des Automaten.

Schlüsselworterkennung, falls Bezeichner

Konvertierung, falls Zahl

Eintragen aller relevanten Informationen (Morphemcode, Morphemwert, Zeilen-/Spaltenposition) in eine Struktur Morphem.

2.2 GRAMMATIK DER PL/0 TOKEN

Terminales Alphabet(T)

Ziffern (0..9)

Buchstaben (A..Z, a..z)

Sonderzeichen

(+ | - | * | / | , | . | ; | (|) | ? | ! | # | = | < | > | :)

Nicht terminales Alphabet(N)

<Morphem>, <Sonderezeichen>, <Zahl>, <Zi>, <Bezeichner>, <Buzi>, <Bu>

Startsymbol(s)

<Morphem>

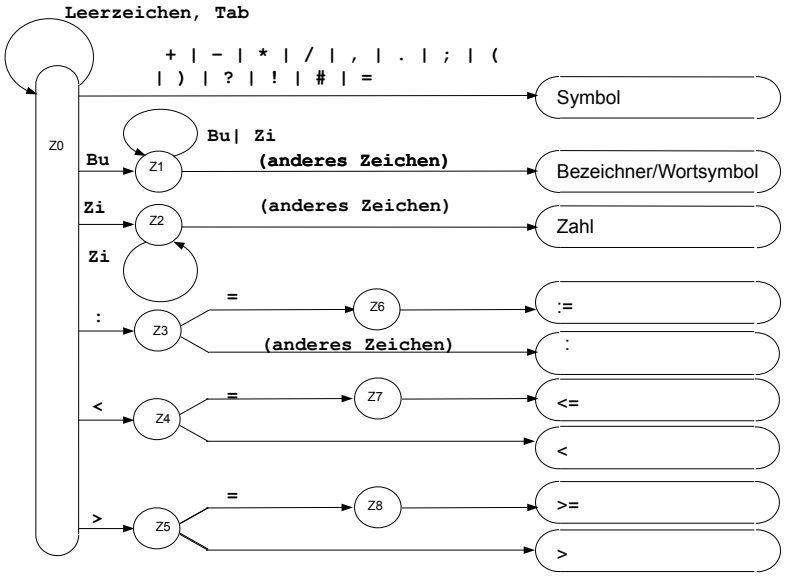
Regeln(R)

Buzi: Buchstabe/Ziffer; Bu: Buchstabe

2.2.1 REGELN DER GRAMMATIK DER MORPHEME

```
<Morphem>       ::=<Sonderzeichen>|<Zahl>|<Bezeichner>

<Sonderzeichen> ::=    + | - | * | /  |  ,
                     |   . | ; | ( | )  | ?
                     |   ! | # | = | <  | <=
                     |   > | >=| :=

<Zahl>          ::=  Zi {<Zi>}

<Zi>            ::=  0 | 1 | 2 . . . 7 | 8 | 9

<Bezeichner>    ::=  Bu {BuZi}

<BuZi>          ::=  Bu | Zi

<Bu>            ::=  A | B | ... | Z | a | b | ... | z
```

SYNTAXGRAPH



2.3 ENDLICHER DETERMINISTISCHER AUTOMAT

$A=(X,Y,Z,f,g,F,z_0)$

X: Eingabealphabet
Y: Resultatalphabet
Z: Menge der Zustände
f: Schalt-/Übergangsfunktion
g: Ausgabefunktion
F: Menge der Finalzustände
z_0: Anfangszustand

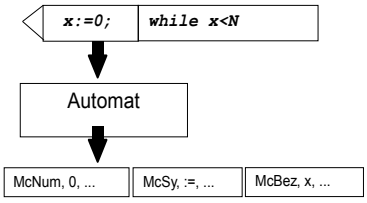


Bild 1: Automat erzeugt Morpheme

Eingabealphabet X:

Für einen PL/0 Scanner umfasst das Eingabealphabet
die Buchstaben (A-Z, a-z),
die Ziffern (0-9) sowie
eine Reihe von Sonderzeichen (+ - * / ( ) , . ; # ? ! < > : =).

Menge der Finalzustände F:

Ein Finalzustand ist dann erreicht, wenn ein Token vollständig erkannt wurde. Die zu erkennenden Token umfassen dabei sinnvollerweise:

- Sonderzeichen (Z0, Z3, Z4, Z5)
- Bezeichner / Wortsymbol (Z1)
- Zahl (Z2)
- := (Z6)
- <= (Z7)
- >= (Z8)

Resultatalphabet

Token Sonderzeichen

Token Bezeichner

Token Zahl

Token :=, :, <=, <, >=, >

Schaltfunktion

Jeweils der Folgezustand in Abhängigkeit von Eingabezeichen und Zustand

Ausgabefunktion

Übernahme des Eingabezeichens in den Puffer

2.3.1 AUTOMATENTABELLE

Tabelle bestehend aus Zeilen und Spalten, gut als zweidimensionales Array implementierbar

Zeilen bilden die Zustände des Automaten (Z0..Z5)

Spalten werden durch die Eingabezeichen gebildet. Es ergibt sich eine sehr große Automatentabelle, auf Grund sehr vieler Spalten.

Verringerung der Spalten durch Klassifizierung
(Indizierung durch Ascii-Code → 128 Spalten)
Beschränkung auf die tatsächlich verwendeten Eingabezeichen (A..Z, a..z, 0..9, + - * / ( ) , . ; # ? ! < > : =) → Indizierungsfunktion notwendig
Einführung von Zeichenklassen (Buchstaben, Ziffern, ...)

2.3.1.1 ZEICHENKLASSENVEKTOR

Mittels Zeichenklassenvektor kann die Zeichenklassifizierung sehr effizient vorgenommen werden.

Der Zeichenklassenvektor hat die Länge 128, bzw. 256, wenn man den Ascii-code zugrunde legt

Er enthält zu jedem Ascii-code an dessen Stelle einen Wert, der die Zeichenklasse repräsentiert.

```
/* Zeichenklassenvector */
static char vZKl[128]=
/*      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    */
/*-------------------------------------------------------*/
 /* 0*/{7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,/* 0*/
 /*10*/ 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,/*10*/
 /*20*/ 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,/*20*/
 /*30*/ 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 0, 5, 4, 6, 0,/*30*/
 /*40*/ 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,/*40*/
 /*50*/ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0,/*50*/
 /*60*/ 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,/*60*/
 /*70*/ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0}/*70*/;
```

0: Sonderzeichen (alle oder nur erlaubte)

1 Ziffern

2 Buchstaben

3 :

4 =

5<

6>

7 Sonstige Steuerzeichen

→Jedem ASCII-Zeichen wird eine Spalte („Handlung“) in der Automatentabelle zugewiesen (mit Hilfe des Syntaxgraphs in Abschnitt 2.2.1):

Folgezustand bei Beenden unerheblich, da jeder neue Aufruf des Lexers in Zustand 0 beginnt

Der Zustand, in dem der Lexer beendet wird, entspricht dem Finalzustand

Aktion (lesen, schreiben, schreiben als Großbuchstabe, beenden)								
	SoZei	Ziffer	Buchstabe	:	=	<	>	Sonst
Z0	0 s+l+b	2 s+l	1 sg+l	3 s+l	s+l+b	4 s+l	5 s+l	b
Z1	b	1 s+l	1 sg+l	b	b	b	b	b
Z2	b	2 s+l	b	b	b	b	b	b
Z3	b	b	b	b	6 s+l	b	b	b
Z4	b	b	b	b	7 s+l	b	b	b
Z5	b	b	b	b	8 s+l	b	b	b
Z6	b	b	b	b	b	b	b	b
Z7	b	b	b	b	b	b	b	b
Z8	b	b	b	b	b	b	b	b

s: Schreibe des betrachteten Symbols in die Ausgabe
g/sg: Schreibe Großbuchstabe des betrachteten Buchstaben
l: Lese nächstes Symbol
b: Beende das Erkennen dieses Tokens

2.3.1.2 AKTIONEN

Schreiben, lesen, beenden	`void fslb(void);`
Schreiben, lesen	`void fsl(void);`
Schreiben Großbuchstabe, lesen	`void fgl(void);`
Beenden	`void fb(void);`
Lesen	`void fl(void);`

- Die Aktionen bestehen wiederum aus den elementaren Aktionen
  - Lesen
  - Schreiben
  - Beenden
  - Schreiben als Großbuchstabe

2.3.1.3 FAZIT

- Jede Zelle der Automatentabelle enthält nun
  - den jeweiligen Folgezustand
  - Index auf eine auszuführende Aktion
- Implementationsmöglichkeiten sind
  - Objekte mit 2 Elementen
    - Folgezustand
    - Funktionspointer
  - Array oder Struktur mit
    - Folgezustand
    - Funktionsindex
    - Array der Funktionen ist erforderlich

VARIABLEN BEISPIELLEXER

Variable	Verwendung	Wert bei Aufruf
char x;	aktuelles Eingabezeichen	Akt. Eingabezeichen
char z;	Zustand des Automaten	0
char zx;	Folgezustand des Automaten	0
char end;	Flag über das Beenden	0
tMorph Morph;	Aktuelles Morphem	0
tMorph MorphInit;	Leere Struktur Morphem zur Initialisierung	0
char vBuf[1024];	Puffer zur Morphembildung	leerer String
char * pBuf;	Zeiger in den Puffer vBuf	vBuf
int line, col;	Morphemoposition in Zeile und Spalte	
	ZeichenklassenVektor	
	Automatentabelle	
	Funktionenarray	

2.4 IMPLEMENTIERUNG DES LEXERS

```
int initLex(char* fileName);
tMorph* lex();

Oder:
int lexInit(char* fileName);
tMorph* lexGetMorph();   // akt. Morphem
tMorph* lexNextMorph(); // synonym zu lex
```

2.4.1 DATENTYPEN

- Jeder Eintrag der Automatentabelle besteht aus einem Byte.
- Das niederwertige Halbbyte (4bit) enthält den Folgezustand
- Das höherwertige Halbbyte (4bit) enthält einen Funktionsindex, multipliziert mit 16, oder um 4 bit nach links verschoben


```
/* Zeichenklassenvector */
static char vZKl[128]=
/*      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F     */
/*-------------------------------------------------------*/
 /* 0*/{7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,/* 0*/
 /*10*/ 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,/*10*/
 /*20*/ 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,/*20*/
 /*30*/ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 0, 5, 4, 6, 0,/*30*/
 /*40*/ 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,/*40*/
 /*50*/ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0,/*50*/
 /*60*/ 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,/*60*/
 /*70*/ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0}/*70*/;
```

```
/* Schalt und Ausgabefunktionen des Automaten */
static void fl (void);
static void fb (void);
static void fgl (void);
static void fsl (void);
static void fslb(void);
typedef void (*FX)(void);
static FX vfx[]={fl,fb,fgl,fsl,fslb};

/* Funktionsindex *0x10, bzw. *16 */
typedef enum T_Fx{
  ifl=0x0,
  ifb=0x10,
  ifgl=0x20,
  ifsl=0x30,
  ifslb=0x40
}tFx;

/* Morphemcodes */
typedef enum T_MC{
```

```
  mcEmpty, mcSymb, mcNum, mcIdent
}tMC;
typedef enum T_ZS{
  zNIL,
  zERG=128,zLE,zGE,
  zBGN,zCLL,zCST,zDO,zEND,zIF,zODD,zPRC,zTHN,zVAR,zWHL
}tZS;

typedef struct{
  tMC MC; /* Morphemcode */
  int PosLine; /* Zeile */
  int PosCol; /* Spalte */
  union VAL {
    long Num;
    char*pStr;
    int Symb;
  }Val;
  int MLen; /* Morphemlnge*/
}tMorph;

int initLex(char* fname);
tMorph* Lex(void);
```

```
static char vSMatrix[][8]=
/*        So      Zi      Bu      ':'      '='    '<'     '>'   Space
/*-------------------------------------------------------------------
*/
/* 0 */{0+ifslb,1+ifsl ,2+ifgl ,3+ifsl ,0+ifslb,4+ifsl ,5+ifsl ,0+ifl ,
/* 1 */ 0+ifb  ,1+ifsl ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb ,
/* 2 */ 0+ifb  ,2+ifsl ,2+ifgl ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb ,
/* 3 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,6+ifsl ,0+ifb  ,0+ifb  ,0+ifb ,
/* 4 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,7+ifsl ,0+ifb  ,0+ifb  ,0+ifb ,
/* 5 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,8+ifsl ,0+ifb  ,0+ifb  ,0+ifb ,
/* 6 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb ,
/* 7 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb ,
/* 8 */ 0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb  ,0+ifb
};
```

Finalzustand festhalten:

```
static char vSMatrix[][8]=
/*        So      Zi      Bu      ':'      '='    '<'     '>'   Space
/*-------------------------------------------------------------------
*/
/* 0 */{0+ifslb,1+ifsl ,2+ifgl ,3+ifsl ,0+ifslb,4+ifsl ,5+ifsl ,0+ifl ,
/* 1 */ 1+ifb  ,1+ifsl ,1+ifb  ,1+ifb  ,1+ifb  ,1+ifb  ,1+ifb  ,1+ifb ,
/* 2 */ 2+ifb  ,2+ifsl ,2+ifgl ,2+ifb  ,2+ifb  ,2+ifb  ,2+ifb  ,2+ifb ,
. . .
/* 8 */ 8+ifb  ,8+ifb  ,8+ifb  ,8+ifb  ,8+ifb  ,8+ifb  ,8+ifb  ,8+ifb
};
```

Durch Einführen eines weiteren Zustandes (9) kann das Beenden des Automaten über die Automatentabelle gesteuert werden. Dabei wird keine neue Zeile für diesen Zustand benötigt, solange dieser Zustand den höchsten Index zugeordnet bekommt.

```
static char vSMatrix[][8]=
/*        So      Zi      Bu      ':'      '='    '<'     '>'   Space
/*-------------------------------------------------------------------*/
/* 0 */{9+ifslb,1+ifsl ,2+ifgl ,3+ifsl ,9+ifslb,4+ifsl ,5+ifsl ,0+ifl ,
/* 1 */ 9+ifb  ,1+ifsl ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb ,
/* 2 */ 9+ifb  ,2+ifsl ,2+ifgl ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb ,
/* 3 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,6+ifsl ,9+ifb  ,9+ifb  ,9+ifb ,
/* 4 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,7+ifsl ,9+ifb  ,9+ifb  ,9+ifb ,
/* 5 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,8+ifsl ,9+ifb  ,9+ifb  ,9+ifb ,
/* 6 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb ,
/* 7 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb ,
/* 8 */ 9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb  ,9+ifb
};
```

2.4.2 INITIALISIERUNG

```
static FILE *pIF;
static tMorph MorphInit;
extern tMorph Morph;
static int X;
static int Z;
static char vBuf[1024+1];
static char* pBuf;
static int line,col;
static int Ende; // Entfällt, wenn Zustand 9 -> Ende
/* Initialisierung der lexiaklischen Analyse */
int initLex(char* fname){
  char vName[128+1];
  strcpy(vName,fname);
  if (strstr(vName,".pl0")==NULL)
    strcat(vName,".pl0");
  pIF=fopen(vName,"r+t");
  if (pIF!=NULL){
    X=fgetc(pIF); /* Lesen des ersten Zeichens */
    return OK;
  }
  return FAIL;
}
```

2.4.3 ALGORITHMUS

```
tMorph* Lex(void){
  Z=0; // Anfangszustand, wenn nicht aus Tabelle
  int zx;
  Morph=MorphInit;
  Morph.PosLine=line;
  Morph.PosCol =col;
  pBuf=vBuf;
```

```
  End=0;
  do{
    zx= vSMatrix[Z][vZKl[X&0x7f]]&0xF;
    vfx[(vSMatrix[Z][vZKl[X&0x7f]])>>4]();
    Z=zx;
  }while (End==0); // (Z!=zEnd) // zEnd:9
  return &Morph;
}
```

Hinweis Praktikum: Klassifizierung soll über Zeichenklassenvektor passieren!

2.4.4 FUNKTIONEN

Vorlesung
01.11.2017

```
/* lesen */
static void fl (void){
  X=fgetc(pIF);
  if (X=='\n')
    line++,
    col=0;
  else
    col++;
}
/* schreiben als Großbuchstabe, lesen */
static void fgl (void){
  *pBuf=(char)toupper(X); // oder *Buf=(char)X&0xdf;
  *(++pBuf)=0;
  fl();
}
/* schreiben, lesen */
static void fsl (void){
  *pBuf=(char)X;
  *(++pBuf)=0;
  fl();
}
/*schreiben, lesen, beenden */
static void fslb(void){
  fsl();
  fb();
}
```

2.4.4.1 BEENDEN

```
static void fb (){
  int i,j;
  switch (Z){
    /* Symbol */
    case 3: // :
    case 4: // <
    case 5: // >
    case 0:
```


```
9         Morph.Val.Symb=vBuf[0];
10        Morph.MC =mcSymb;
11        break;
12      /* Zahl */
13      case 1:
14        Morph.Val.Num=atol(vBuf);
15        Morph.MC =mcNum;
16        break;
17      /* Ergibtzeichen */
18      case 6:
19        Morph.Val.Symb=(long)zErg;
20        Morph.MC =mcSymb;
21        break;
22      /* KleinerGleich */
23      case 7:
24        Morph.Val.Symb=(long)zle;
25        Morph.MC =mcSymb;
26        break;
27      /* GroesserGleich */
28      case 8:
29        Morph.Val.Symb=(long)zge;
30        Morph.MC =mcSymb;
31        break;
32    }
33    Ende=1; // entfällt bei Variante mit Zustand zEnd
34  }
```

Die zusammengesetzten Sonderzeichen erhalten als Morpheminhalt einen Code, der im Headerfile wie folgt vereinbart ist:

```
typedef enum T_ZS
{
zNIL,
zErg=128,zle,zge,
. . .
}
```

2.5 SCHLÜSSELWORTERKENNUNG

Wird für alle erkannten Zeichenfolgen, die mit einem Buchstaben beginnen, durchlaufen (sehr oft)

Intelligente Lösungen bestimmen hier das Laufzeitverhalten des Compilers wesentlich mit.

1. Das Einstiegsmodell:

Schlüsselwörter, in einem Vektor gespeichert, durchsuchen.

2. Schlüsselworte liegen sortiert vor, Abbruch, wenn gesuchtes Wort kleiner als zu vergleichendes Schlüsselwort ist.
3. Binäre Suche
4. Hashtechniken

Einbeziehung der Automatentabelle

4. Könnte auch bereits durch die Automatentabelle gelöst werden, jedoch hat dann die Zeichenklassifizierung keinen Sinn mehr. Für jedes Schlüsselwort werden so viele Zustände benötigt, wie es Zeichen hat, für jedes Zeichen, das in einem Schlüsselwort vorkommt, wird eine Spalte benötigt.
5. Finalzustand „Potenzielles Schlüsselwort“ wird gebildet, wenn ein Token nur aus Zeichen, die in Schlüsselworten vorkommen, besteht (Neue Zeichenklasse erforderlich).
6. Um die Schlüsselworterkennung nur aufzurufen, wenn sie wirklich benötigt wird, könnte man weitere Zustände einführen. Ein Schlüsselwort ist mindestens 2 Zeichen lang und enthält keine Ziffern (keine neue Zeichenklasse erforderlich)
7. Neue Zeichenklasse für alle Buchstaben, mit denen ein Schlüsselwort beginnt.

IMPLEMENTATIONSVARIANTEN

- Binäre Suche

- Arbeit mir der Automatentabelle

- Hashtechnik

2.5.1 BINÄRE SUCHE

```c
int main(int argc, char*argv[]){
  /* sortierter Vektor der Keywords: */
  const char* Keyw[]= {"BEGIN","CALL","CONST","DO","END","IF","ODD"
     ,"PROCEDURE","THEN","VAR","WHILE"};
  int n=sizeof Keyw/sizeof(char*);
  int i;
  printf("Anz:%d\n",n);
  i=binary_search(Keyw,n,argv[1]);
  if (i>=0)
    printf("Ergebnis: %d %s\n",i, Keyw[i]);
  else
    printf("Ergebnis: not found\n");
  return 0;
}
```

NICHT-REKURSIV

```c
int binary_search(
  const char** M, // stock
  int n,          // number
  const char* X){ // needle
  unsigned mitte;
  unsigned links = 0; /* man beginne beim kleinsten Index */
  unsigned rechts = n 1;
```

```
  /* Arrays sind 0basiert
  */
  int ret=1;
  int bool;
  do{
    if (n<=0)
      break;
    mitte = links + ((rechts links)/ 2); // Bereich halbieren
    if (rechts < links)
      break; // alles wurde durchsucht, X nicht gefunden
    bool=strcmp(M[mitte],X);
    if (bool==0) // Element X gefunden?
      ret=mitte;
    else if (bool >0) // im linken Abschnitt weitersuchen
      rechts = mitte;
    else  // im rechten Abschnitt weitersuchen
      links = mitte + 1;
    n=(n)/2;
  } while (bool!=0);
  return ret;
}
```

REKURSIV

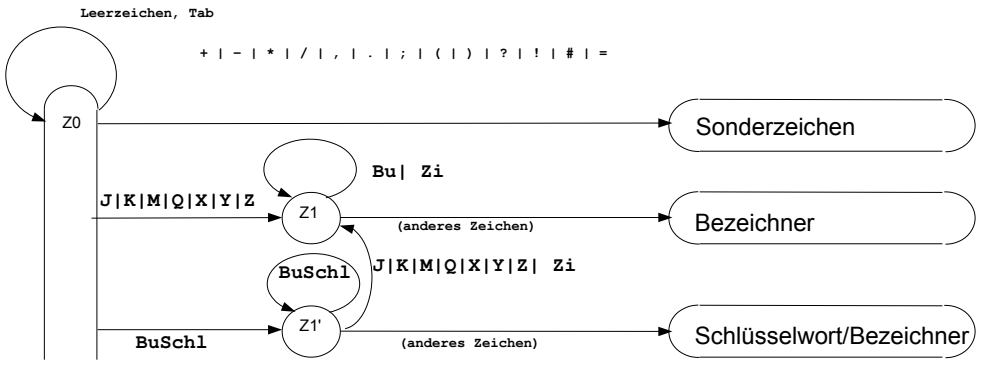
```
int count=0;
int binary_search(
  const char** M, // stock
  int n,          // number
  const char* X,  // needle
  int offset){    // offset fuer return
  count ++;
  unsigned mitte, index, ret=1;
  unsigned links = 0; // man beginne beim kleinsten Index
  unsigned rechts = n 1;
  // Arrays sind 0basiert
  if (M == NULL || n <= 0) // Bereichsüberprüfung
    printf("out of Range n:%d M[0]:%s m[%d]:%s\n",n, M[0], n1, n>0?
        M[n1]:"OutOfRange");
  else{
    int bool;
    mitte = links + ((rechts links) / 2); // Bereich halbieren
    if (rechts >= links){
      bool=strcmp(M[mitte],X);
      if (bool==0) // gefunden
        ret= mitte+offset;
      else if (bool >0) // links weiter
        ret= binary_search(M, n/2, X,offset);
      else // rechts weiter
        ret= binary_search(M+mitte+1, n/2, X, offset+=mitte+1);
    }
```


```
26    }
27    return ret;
28 }
```

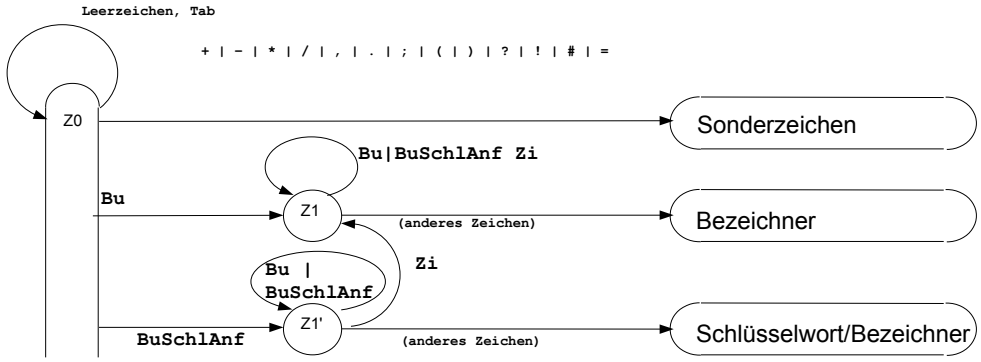
2.5.1.1 TABELLE

	kommt vor	Beginnt mit	begin	call	const	Do	end	if	odd	procedure	then	var	while
A	X			x								x	
B	X	X	x										
C	X	X		x	x					x			
D	X	X				x	x		x	x			
E	X	X	x				x			x	x		x
F	X							x					
G	X		x										
H	X										x		x
I	X	X	x					x					x
J													
K													
L	X			x									x
M													
N	X		x		x		x				x		
O	X	X			x	x			x	x			
P	X	X								x			
Q													
R	X									x		x	
S	X				x								
T	X	X			x						x		
U	X									x			
V	X											x	
W	X	X											x
X													
Y													
Z													

2.5.1.2 ZEICHENKLASSEN FÜR BUCHSTABEN DER SCHLÜSSELWÖRTER



2.5.1.3 EXTRA ZEICHENKLASSE FÜR 1. BUCHSTABEN VON SCHLÜSSELWÖRTERN



2.5.2 AUTOMATENTABELLE

mit Schlüsselwörtern `BEGIN`, `CALL`, `CONST`

		So	Zi	Bu	:	=	<	>	sonst	A	B	C	D	E	F	G	H	I	L	N	O	P	R	S	T	U	V	W
sozei	0	0slb	1sl	2gl	3sl	0slb	4sl	5sl	0l	2gl	9gl	14gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
Zahl	1	1b	1sl	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b	1b
ident	2	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
:	3	3b	3b	3b	3b	6sl	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b	3b
<	4	4b	4b	4b	4b	7sl	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b	4b
>	5	5b	5b	5b	5b	8sl	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b	5b
:=	6	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b	6b
<=	7	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b	7b
>=	8	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b	8b
	9	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	10gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	10	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	11gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	11	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	12gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	12	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	13gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
BEGIN	13	13b	2sl	2gl	13b	13b	13b	13b	13b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	14	2b	2sl	2gl	2b	2b	2b	2b	2b	15gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	18gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	15	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	16gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	16	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	17gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
CALL	17	17b	2sl	2gl	17b	17b	17b	17b	17b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	18	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	19gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl
	19	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	20gl	2gl	2gl	2gl
	20	2b	2sl	2gl	2b	2b	2b	2b	2b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	21gl	2gl	2gl
CONST	21	21b	2sl	2gl	21b	21b	21b	21b	21b	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl	2gl

2.5.3 SCHLÜSSELWORTTABELLE MIT EINFACHER HASHTECHNIK

Die Hashfunktion berechnet aus Abfangsbuchstaben und Länge -2 die Stelle in der Tabelle
+ Es gibt keine Dopplungen – sehr schnell
- Matrix ist sehr dünn belegt

```
tKeyWordTab mSclW['Z'-'A'+1][8]=
{
/*Len:     2           3           4              5              6           7           8           9
*/
/* A */ {{0L ,zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* B */ {{0L ,zNIL},{0L,   zNIL},{0L,    zNIL},{"EGIN",zBGN},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* C */ {{0L ,zNIL},{0L,   zNIL},{"ALL",zCLL},{"ONST",zCST},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* D */ {{"O",zDO },{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* E */ {{0L ,zNIL},{"ND",zEND},{"LSE",zELS},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* F */ {{0L ,zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* G */ {{0L ,zNIL},{"ET",zGET},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* H */ {{0L ,zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* I */ {{"F",zIF },{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* J */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* K */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* L */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* M */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* N */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* O */ {{0L, zNIL},{"DD",zODD},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* P */ {{0L, zNIL},{"UT",zPUT},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{"ROCEDURE",zPRC}},
/* Q */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* R */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* S */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* T */ {{0L, zNIL},{0L,   zNIL},{"HEN",zTHN},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* U */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* V */ {{0L, zNIL},{"AR",zVAR},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* W */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{"HILE",zWHL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* X */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* Y */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}},
/* Z */ {{0L, zNIL},{0L,   zNIL},{0L,    zNIL},{ 0L,    zNIL},{0L,zNIL},{0L,zNIL},{0L,zNIL},{0L,         zNIL}}};
```

```
/* Bezeichner/Wortsymbol */
case 2:
  i=vBuf[0]-'A';
  j=strlen(vBuf)-2;
  if (j>=0)
    if (mSclW[i][j].pKeyWord)
      if (strcmp(vBuf+1,mSclW[i][j].pKeyWord)==0){
        Morph.MC =mcSymb;
        Morph.Val.Symb=mSclW[i][j].KWCode;
        break;
      }
  Morph.Val.pStr=vBuf;
  Morph.MC =mcIdent;
  break;
```

```
typedef enum T_ZS{
  zNIL,
  zErg=128,zle,zge,
  zBGN,zCLL,zCST,zDO,zEND,zIF,zODD,zPRC,zTHN,zVAR,zWHL
}tZS;
```


3 PARSER

3.1 GRUNDLAGEN

Der Teil eines Compilers, der prüft, ob die Folge der Token einen gültigen Satz der Sprache bildet

Unterscheidung nach der Strategie in TopDown oder ButtomUp Parser

Unterscheidung nach der Arbeitsweise

LL- oder LR-Parser

LL: von links lesend, das am weitesten links stehende Nichtterminal ersetzend

LR: von links lesend, das am weitesten rechts stehende Nichtterminal ersetzend

3.1.1 KELLERAUTOMAT

- Parser basieren auf Kellerautomaten.
- Vorstellbar als eine Menge von Automaten, die sich gegenseitig aufrufen, wie Funktionen. Der Zustand des aufrufenden Automaten wird gewissermaßen im Stack konserviert.
- Kellerautomaten schalten in Abhängigkeit der nächsten k Eingabesymbole und dem obersten Stackelement und ihrem aktuellen Zustand.
- Der Stack ergibt sich bei vielen Verfahren implizit aus dem call-stack, zB. Beim Verfahren des rekursiven Abstiegs

Ausgangspunkt ist immer die Grammatik der zugrunde liegenden Sprache

Grammatik und Parser hängen eng zusammen, LL(1) Parser verarbeiten LL(1) Grammatiken

Zur Analyse wird k Zeichen vorausgeschaut, um eine Alternative sicher zu erkennen, Bei LL(1) Grammatiken ist k=1.

LL Grammatiken müssen linksrekursionsfrei sein

Durch geeignete Umformungen können Linksrekursionen beseitigt werden.

Je nach Verfahren werden weitere Umformungen nötig (BNF->EBNF, EBNF->BNF, (E)BNF->Graphen)

3.2 LL(1)-GRAMMATIK

Eine Klasse von Grammatiken

Jeder Ableitungsschritt wird eindeutig durch das nächste Eingabesymbol bestimmt, Lookahead=1

Linksrekursionen sind nicht erlaubt

First: die terminalen Anfänge aller Zeichenketten, die aus einem NT hergeleitet werden können müssen verschieden sein.

Kann auch ε hergeleitet werden, so müssen zusätzlich alle auf das NT folgen könnenden terminalen Anfänge betrachtet werden und müssen ebenfalls unterschiedlich sein.

```
programm    = block '.'
block       = ["CONST" ident=num{"," ident=num}";"]
              ['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        =faktor { ('*'|'/') faktor }

factor      =ident | number | '(' expression ')'
```

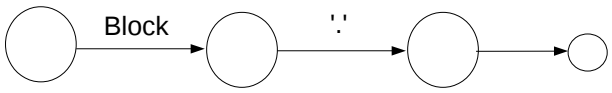
Hinweis: In PL0 darf das `statement` auch leer sein (da eckige Klammern).

3.2.1 FIRST/FOLLOW

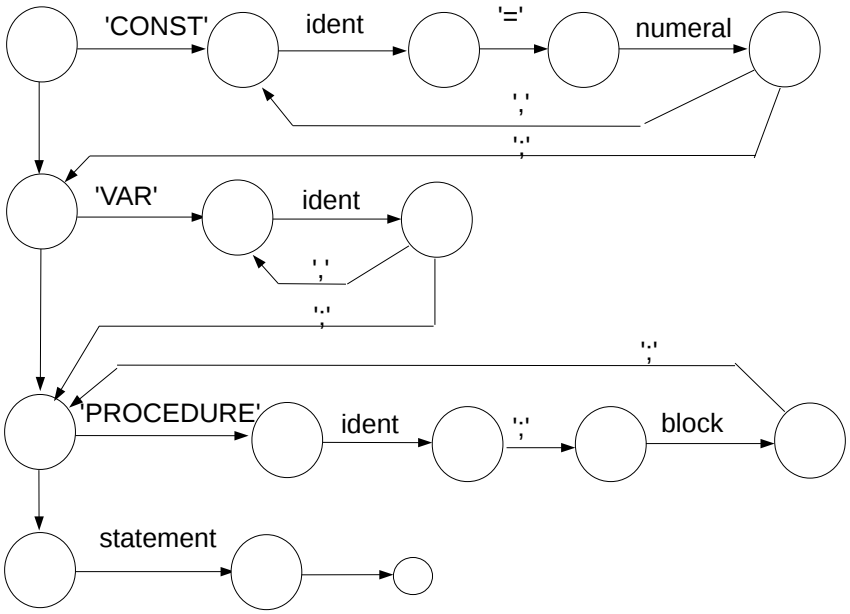
NichtTerminal X	First (X)	Follow (X)
block	CONST VAR PROCEDURE ident CALL BEGIN IF WHILE ? ! ε	. ;
statement	ident CALL BEGIN IF WHILE ? !	. ; end
condition	ODD + - ident Numeral (	THEN DO
expression	(ident numeral + -	) . ; = # < <= > >= THEN DO END
term	(ident numeral	) . ; + - = # < <= > >= THEN DO END
factor	(ident numeral	) . ; + - * / = # < <= > >= THEN DO END

3.2.2 ALS GRAPH

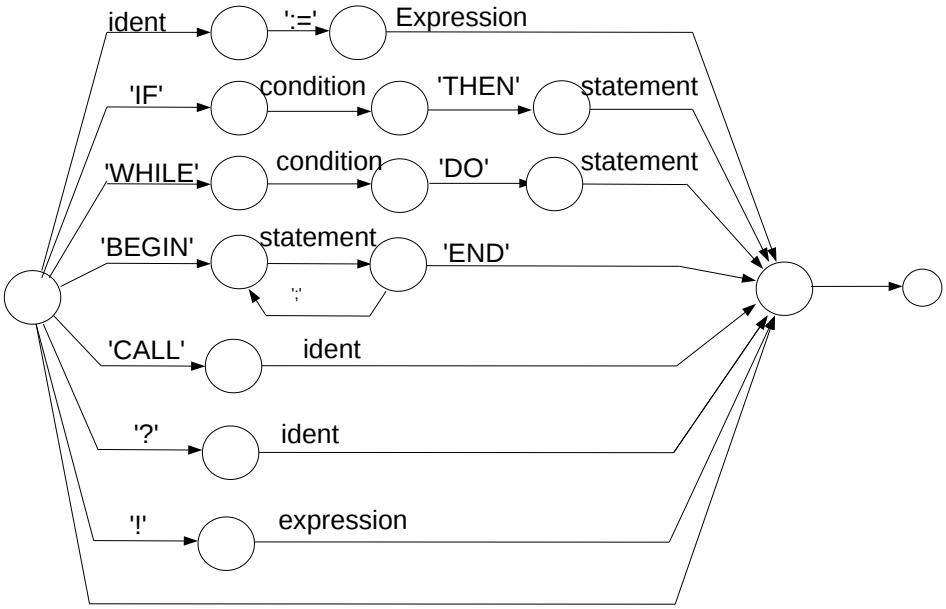
3.2.2.1 PROGRAMM



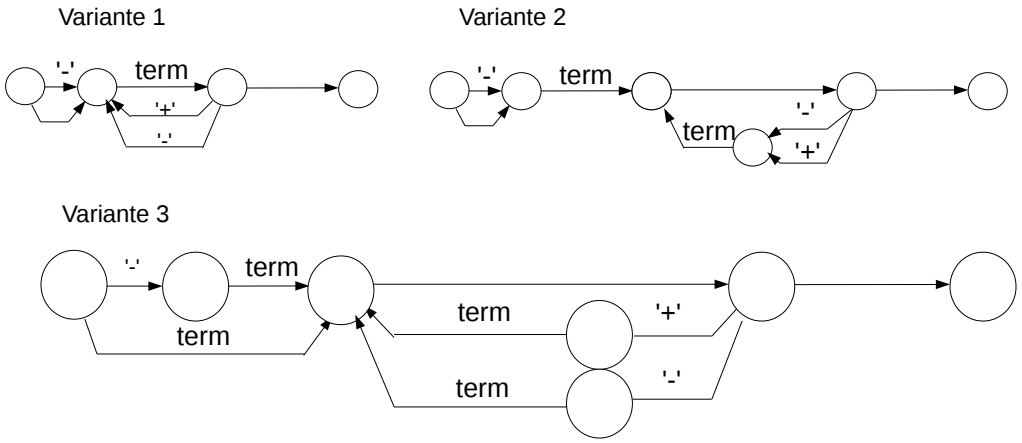
3.2.2.2 BLOCK



3.2.2.3 STATEMENT

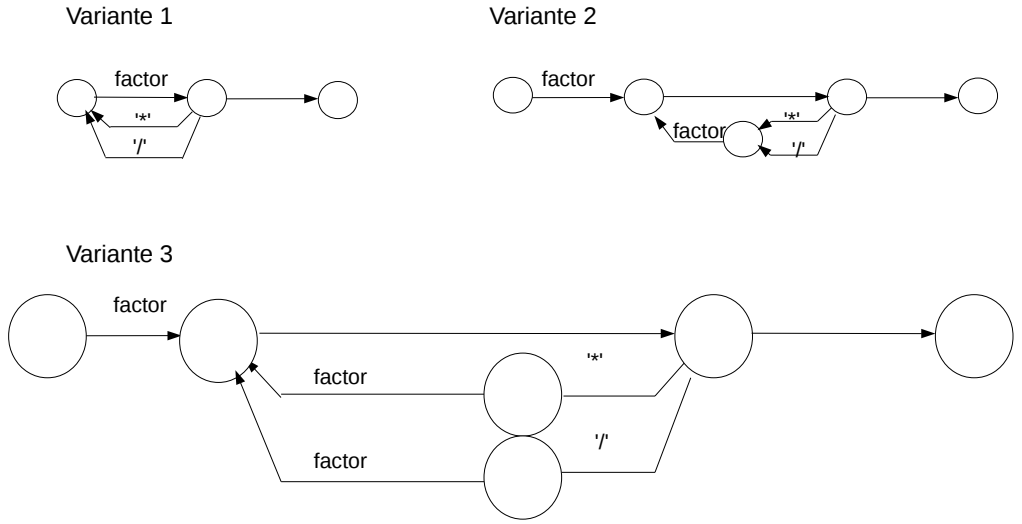


3.2.2.4 EXPR

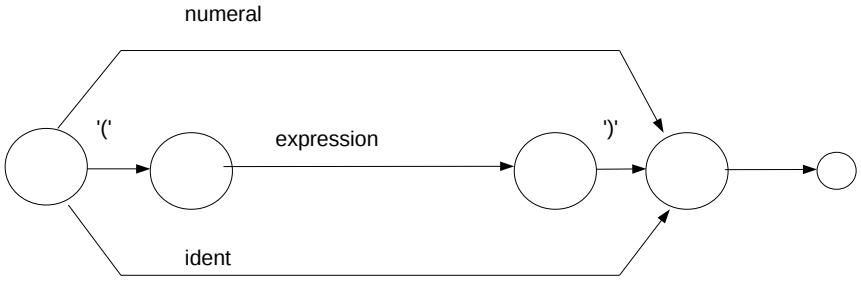


1. Variante reicht für Parser, 3. Variante ist sinnvoller, wenn das Ergebnis im Compiler weiter genutzt werden soll.

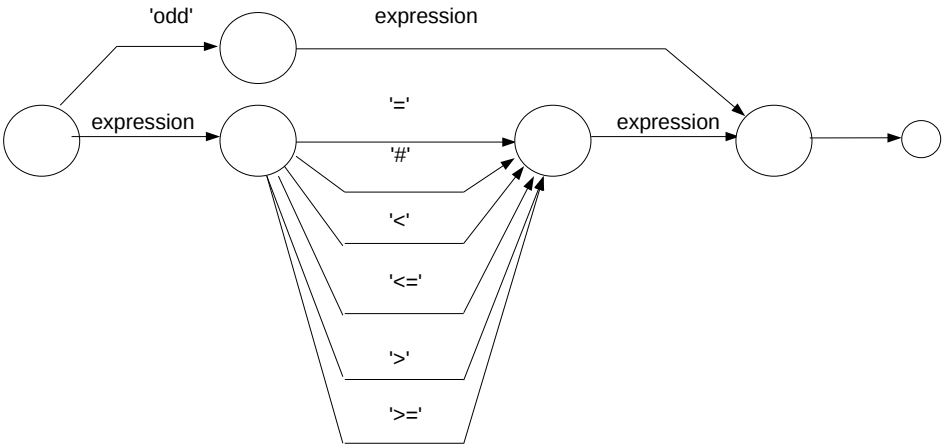
3.2.2.5 TERM



3.2.2.6 FACTOR



3.2.2.7 CONDITION



3.2.2.8 BEWERTUNG

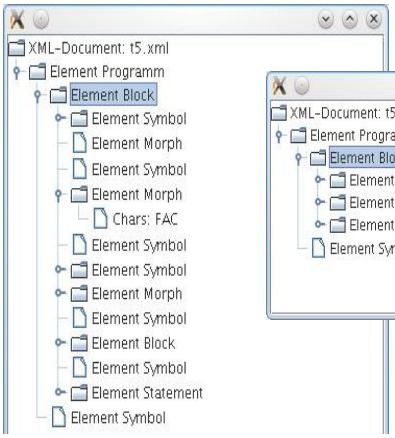
Graphen sind ein direktes Abbild der Regeln der EBNF

Alternativen werden sofort am Beginn des Grafen sackgassenfrei erkannt.

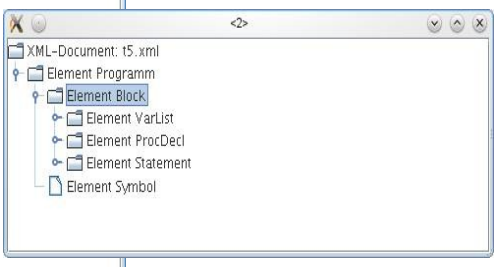
Protokolliert man die akzeptierten Bögen, so ergibt sich der Syntaxbaum.

Der entstehende Syntaxbaum hat keine schöne Struktur.

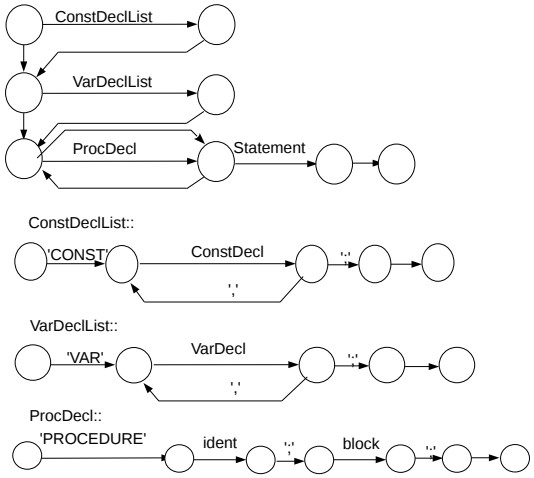
Variante 1



Variante 2



ALTERNATIVE



Es entsteht ein sehr gut lesbarer Syntaxbaum

Alternativen werden nicht sofort erkannt

Es entstehen Sackgassen

Wird eine Sackgassse erkannt, bevor ein Token verarbeitet wurde, ist ein Backtracking bis zur nächsten Alternative möglich. Gelangt die Analyse in eine Sackgasse nachdem ein Token in dem aktuellen Graphen verarbeitet worden ist, so liegt ein Syntaxfehler vor.

3.3 IMPLEMENTATION VON GRAPHEN

Ein Graph kann als Array von Bogenbeschreibungen implementiert werden.

Jede Bogenbeschreibung hat dabei ihren Index im Array und beinhaltet den Index des Folgebogens, wenn der Bogen akzeptiert wird und den Index eines Alternativbogens, falls vorhanden, wenn der Bogen nicht akzeptiert wird.

Die Akzeptanz eines Bogens ist abhängig von der Bogenbewertung und dem aktuellen Morphem.

3.3.1 STRUKTUR EINES BOGENS

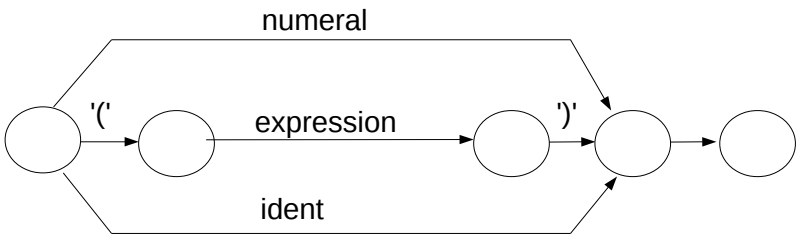
Bogentyp
Bogenbeschreibung
  Symbol
  Morphemcode
  Adresse/Index Graph
Aktion
Index Folgebogen
Index Alternativbogen

```
typedef struct BOGEN
{
  tBg BgD;
  union BGX
  {
    unsigned long X;
    int           S;
    tMC           M;
    tIdxGr        G;
  }   BgX;
  int (*fx)(void);
  int iNext;
  int iAlt;
}tBog;
```

```
typedef enum BOGEN_DESC
          {BgNl= 0,  // NIL
           BgSy= 1,  // Symbol
           BgMo= 2,  // Morphem
           BgGr= 4,  // Graph
           BgEn= 8,  // Graphen e
           }tBg;
```

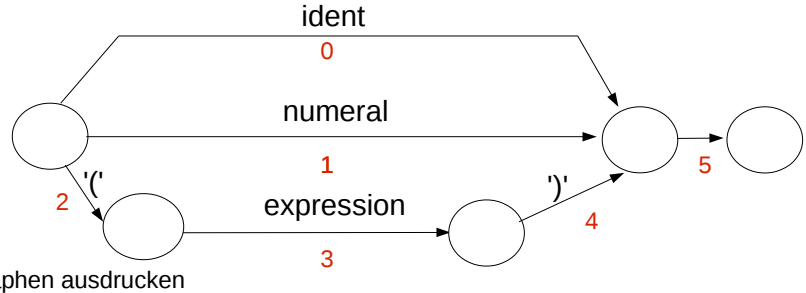
BEISPIEL FACTOR

```
typedef unsigned long ul;
tBog gFact[] = {
/* 0*/ {BgMo,{(ul)mcIdent }, NULL, 5, 1}, /*(0)---ident--->(E)*/
/* 1*/ {BgMo,{(ul)mcNumb  }, NULL, 5, 2}, /* +----number-->(E)*/
/* 2*/ {BgSy,{(ul)'('     }, NULL, 3, 0}, /*(+)----'('---->(3)*/
/* 3*/ {BgGr,{(ul)gExpr   }, NULL, 4, 0}, /*(1)--express-->(4)*/
/* 4*/ {BgSy,{(ul)')'     }, NULL, 5, 0}, /*(0)----')'---->(E)*/
/* 5*/ {BgEn,{(ul)0       }, NULL, 0, 0}  /*(E)---------(ENDE)*/
};
```



Hinweis: 0 in `iAlt` bedeutet keine Alternative (vorher festgelegt, könnte auch -1 sein).

3.3.2 ERSTELLEN EINES GRAPHEN



Graphen ausdrucken
Bögen nummerieren
Bogenbeschreibungen zusammenstellen
Bogen 1 ist Altenativbogen zu Bogen1 und Bogen 2 ist Altenativebogen zu Bogen 1

```
tBog gFact[]={
/* 0*/ {BgMo,{(ul)mcIdent }, NULL, 5, 1},
/* 1*/ {BgMo,{(ul)mcNumb  }, NULL, 5, 2},
/* 2*/ {BgSy,{(ul)'('     }, NULL, 3, 0},
/* 3*/ {BgGr,{(ul)gExpr   }, NULL, 4, 0},
/* 4*/ {BgSy,{(ul)')'     }, NULL, 5, 0},
/* 5*/ {BgEn,{(ul)0       }, NULL, 0, 0}
};
```

HINWEISE ZUR UMSETZUNG

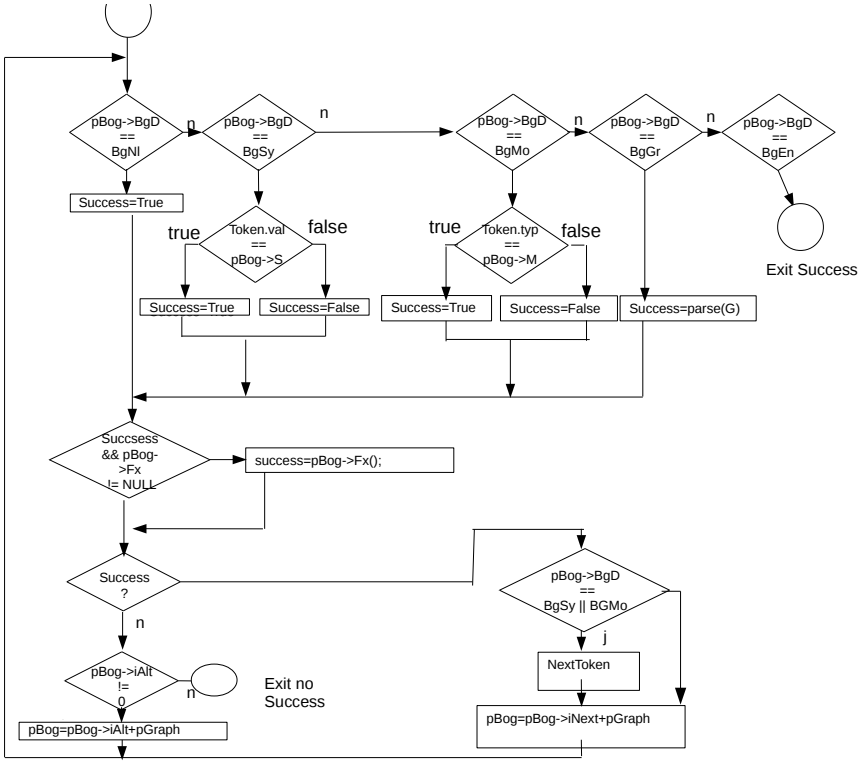
Bei alternativen Bögen ist die Reihenfolge unerheblich, außer bei Vorhandensein von nil-Bögen, diese müssen immer als letzte Alternative angegeben werden.

Index als Kommentar mitführen

Wenn der Bogen mit dem Index 0 nirgends als Alternativbogen auftaucht, kann die Angabe von 0 als Index auf Alternativbogen als Flag „keine Alternative vorhanden“ genutzt werden.

Andere Möglichkeiten das Vorhandensein von Alternativen zu steuern bestehen in dem Indexwert -1 für kein Alternativbogen, oder im Definieren eines Bit im Beschreibungsbyte des Bogens (Bogentyp)

3.3.3 IMPLEMENTATION VON BÖGEN IN GRAPHEN



```
int pars(tBog* pGraph){
  tBog* pBog=pGraph;
  int succ=0;
  if (Morph.MC==mcEmpty)Lex();
  while(1){
    switch(pBog->BgD){
      case BgNl:
        succ=1;
        break;
      case BgSy:
        succ=(Morph.Val.Symb==pBog->BgX.S);
        break;
      case BgMo:
        succ=(Morph.MC==(tMC)pBog->BgX.M);
        break;
      case BgGr:
        succ=pars(pBog->BgX.G);
        break;
      case BgEn:
        return 1; /* Ende erreichet Erfolg */
    }
    if (succ && pBog->fx!=NULL)
      succ=pBog->fx();
    if (!succ)/* Alternativbogen probieren */
      if (pBog->iAlt != 0)
        pBog=pGraph+pBog->iAlt;
      else
        return FAIL;
    else{/* Morphem formal akzeptiert (eaten) */
      if (pBog->BgD & BgSy || pBog->BgD & BgMo)
        Lex();
      pBog=pGraph+pBog->iNext;
    }
  }/* while */
}
```

3.3.4 PARSEBAUM (OPTIONAL)

- Baum ergibt sich als Logbuch des Parsens
- Baum besteht aus Listen
- Listenelement beschreibt einen akzeptiertenBogen
- Enthielt der Bogen ein Metasymbol (Graph), wird eine neue Liste angelegt
- Bei erfolgreicher Kompilierung erfolgt die Ausgabe das Baumes in XML

```
typedef struct{
  int      line;
  char     Type[10];
```


```
  tList*  pList;
  char *  Descr;
}TreeItem;

TreeItem* crItem( tList* pList, // neue Liste od. NULL
                  char* Descr, // Description
                  char* Type, // Bogentyp
                  int line){ // Quellzeile
  TreeItem *ptmp=malloc(sizeof(TreeItem));
  strncpy(ptmp->Type,Type,9);
  ptmp->pList=pList;
  ptmp->Descr=Descr;
  ptmp->line=line;
  return ptmp;
}
```

Aufruf:

```
InsertHead(pTree,crItem((pl=CreateList()),"Prog","MetaSymb",0));
if (pars(gProg,pl)==1)
/* ... */

int pars(tBog* pGraph,tList* pList){
  int verarbMorph=0;
  tBog* pBog=pGraph;
  int succ;
  tList* pl=NULL;
  TreeItem *pItem=NULL;
  if (Morph.MC==mcEmpty)Lex();
  while(1){
    switch(pBog->BgD & (BgNl+BgSy+BgMo+BgGr+BgEn)){
      case BgNl:
        succ=1;
        break;
      case BgSy:
        succ=(Morph.Val.Symb==pBog->BgX.S);
        if (succ)
          InsertTail(pList,crItem(NULL,(crStr(Morph.vBuf)),"Symbol"
             ,Morph.PosLine));
        break;
      case BgMo:
        succ=(Morph.MC==(tMC)pBog->BgX.M);
        if (succ)
          InsertTail(pList,crItem(NULL,(crStr(Morph.vBuf)),"Morph",
             Morph.PosLine));
        break;
      case BgGr:
        pl=CreateList();
        pItem=crItem((pl),crStr(StrGr[pBog->BgX.G]),"MetaSymb",
           Morph.PosLine);
        InsertTail(pList,pItem);
        succ=pars(vGr[pBog->BgX.G],pl);
```

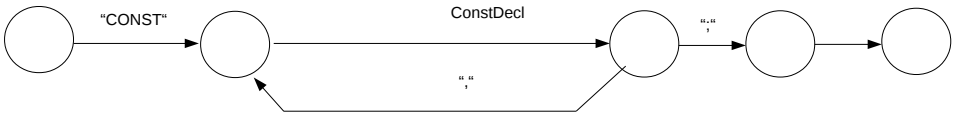

```
32          if (succ!=OK){
33            DeleteList(pl);
34            free(pItem->Descr);
35            free(pItem);
36            GetLast(pList);
37            RemoveItem(pList);
38          }
39          break;
40        case BgEn:
41          return 1; /* Ende erreichet - Erfolg */
42      }
43
44      if (succ && pBog->fx!=NULL)
45        succ=pBog->fx();
46      if (!succ){/* Alternativbogen probieren */
47        if (pBog->iAlt != 0)
48          pBog=pGraph+pBog->iAlt;
49        /* Graph nicht erfolgreich verlassen */
50        else if (verarbMorph)
51          return ERROR; /* Syntaxfehler */
52        else{
53          //printf("go back\n");
54          while(pItem=GetFirst(pList)){
55            free(pItem->Descr);
56            free(pItem);
57            RemoveItem(pList);
58          }
59          return FAIL; /* vielleicht gibt es noch Alternative */
60        }
61      } else { /* Morphem formal akzeptiert */
62        if (pBog->BgD & BgSy || pBog->BgD & BgMo){
63          verarbMorph=1;
64          Lex();
65        }
66        pBog=pGraph+pBog->iNext;
67      }
68    }
69  }
```

```
tBog* vGr[]={gFact,
             gTerm,
             gExpr,
             gCond,
             gStmnt,
             gBlock,
             gProg,
             gAssign,
             gCall,
             gBegin,
             gIf,
             gWhile,
             gInput,
             gOutput,
             gConstList,
             gConstDecl,
             gVarList,
             gVarDecl,
             gProcDecl};
```

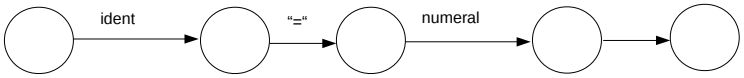
```
char* StrGr[]={"Factor",
               "Term",
               "Expression",
               "Condition",
               "Statement",
               "Block",
               "Program",
               "AssignmentStatement",
               "PrCall",
               "CompoundStatement",
               "ConditionalStatement",
               "LoopStatement",
               "InputStatement",
               "OutputStatement",
               "ConstList",
               "ConstDecl",
               "VarList",
               "VarDecl",
               "ProcDecl"};
```


3.3.5 ERWEITERTE GRAPHEN MIT BACKTRACKING

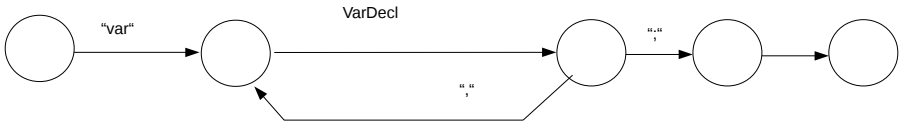
ConstDeclList::



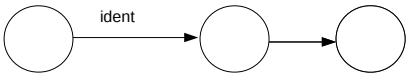
ConstDecl::



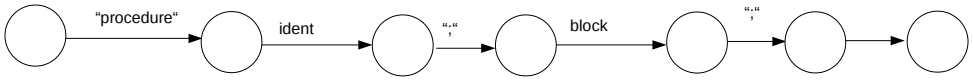
VarDeclList::



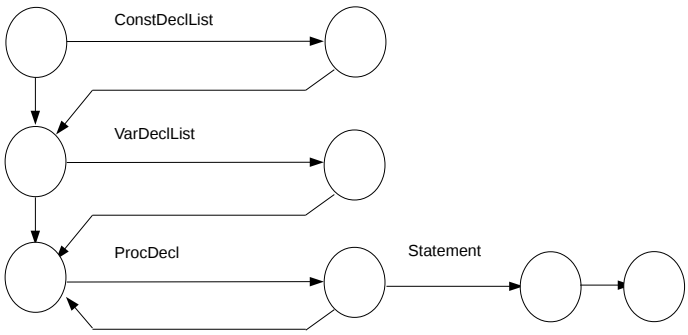
VarDecl::



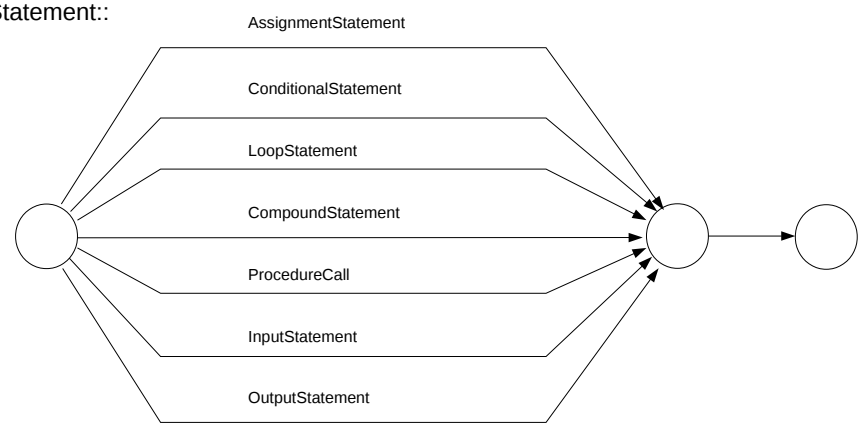
ProcDecl::



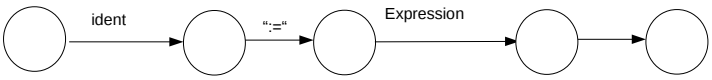
Block::



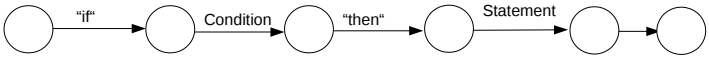
Statement::



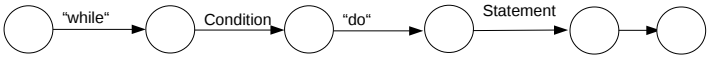
AssignmentStmt::



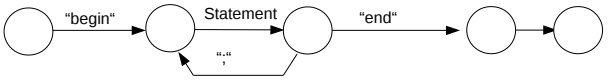
ConditionalStmnt::



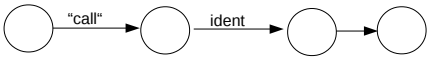
LoopStmnt::



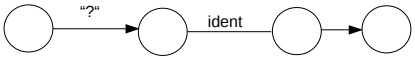
CompoundStmnt::



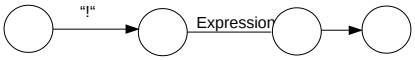
ProcedureCall::



InputStmnt::



OutputStmnt::




```
tBog gStmnt[]= {
/* 0*/ {BgGr,{(unsigned long)iAssign  }, NULL, 7, 1},
/* 1*/ {BgGr,{(unsigned long)iCall    }, NULL, 7, 2},
/* 2*/ {BgGr,{(unsigned long)iBegin   }, NULL, 7, 3},
/* 3*/ {BgGr,{(unsigned long)iIf      }, NULL, 7, 4},
/* 4*/ {BgGr,{(unsigned long)iWhile   }, St4 , 7, 5},
/* 5*/ {BgGr,{(unsigned long)iInput   }, NULL, 7, 6},
/* 6*/ {BgGr,{(unsigned long)iOutput  }, NULL, 7, 0},
/* 7*/ {BgEn,{(unsigned long)0        }, NULL, 0, 0}
```

```
};
tBog gAssign[]={
/* 0*/ {BgMo,{(unsigned long)mcIdent  }, St0 , 1, 0},
/* 1*/ {BgSy,{(unsigned long)zErg     }, NULL, 2, 0},
/* 2*/ {BgGr,{(unsigned long)iExpr    }, St8 , 3, 0},
/* 3*/ {BgEn,{(unsigned long)0        }, NULL, 0, 0}
};
```


4 NAMENSLISTE

4.1 GRUNDLAGEN

- Teil der semantischen Aktionen
- Führt eine Liste über alle gültigen Bezeichner und die Eigenschaften der benannten Sprachkonstrukte
- Realisiert nebenbei die Gültigkeitsbereiche von Bezeichnern
- Operationen der Namensliste:
- einfügen eines Bezeichners in die aktuelle Namensliste
- lokale Suche eines Bezeichners in der aktuellen Namensliste
- globale Suche eines Bezeichners „von innen nach außen“

IMPLEMENTATIONSVARIANTEN

Namensliste für jede Prozedur

Jede Prozedur verfügt über ihre eigene Namensliste.

Über Eltern-Kind-Beziehungen ist die globale Suche von „innen nach außen“ möglich.

Pulsierender Keller

Alle Namen werden in einem Keller geführt.

Anfang einer jeden lokalen Namensliste muss mitgeführt werden.

4.2 NAMENSLISTENEINTRAG

```
typedef struct{
  tKz Kz;          // Kennzeichen (enum)
  short IdxProc;   // Prozedurnummer
  void* pObj;      // Poiner auf benanntes Objekt
  int Len;
  char* pName;     // Pointer auf Name
}tBez;

typedef enum tKZ{
  KzBez,
  KzPrc,
  KzConst,
  KzVar,
  KzLabl,
```

```
15   KzOptr,
16   // ...
17 }
```

4.3 VARIABLENBESCHREIBUNG

Variablen werden zur Laufzeit im Stack angelegt

In jeder neuen Prozedur beginnt die Adressierung der Variablen mit 0

Mit jeder neuen Variablen wird ein Variablenzähler (SpzzVar) um 4 erhöht

1. Variable: relAddr=0
2. Variable: relAddr=4
3. Variable: relAddr:=8 . . .

```
1 typedef struct tVAR{
2   tKz Kz;
3   int Dspl; // Displacement
4 }tVar;
```

4.4 KONSTANTENBESCHREIBUNG

Alle Konstanten werden in einem Konstantenblock gesammelt, der am Ende an den generierten Code angehängt wird.

Die Konstanten sind 4 Byte groß

Die Konstanten werden indiziert

1. Konstante: ConstBlock[0]
2. Konstante: ConstBlock[1]

```
1 typedef struct tCONST{
2   tKz Kz;
3   long Val; // Wert der Konstanten
4   int Idx;  // Index der Konstanten im ConstBlock
5 }tConst;
```


4.5 PROZEDURBESCHREIBUNG

Jede Prozedur und das Hauptprogramm erhalten eine Nummer, das Hauptprogramm hat die Nummer 0

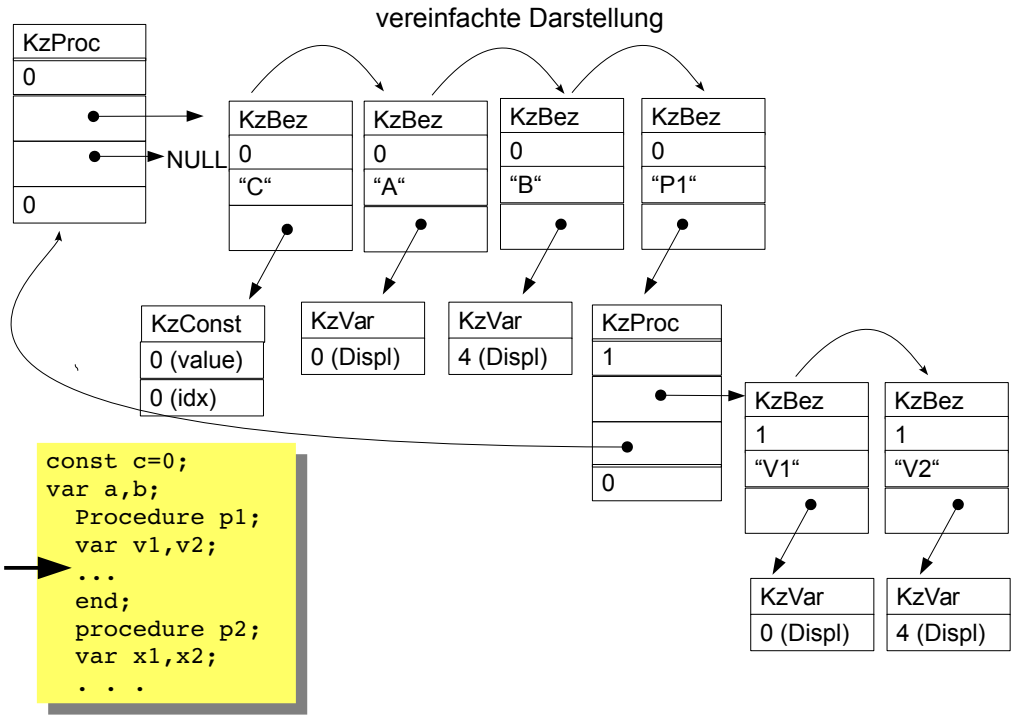
Die Nummern werden fortlaufend vergeben

Jede Prozedurbeschreibung enthält eine Namensliste mit den lokalen Namen

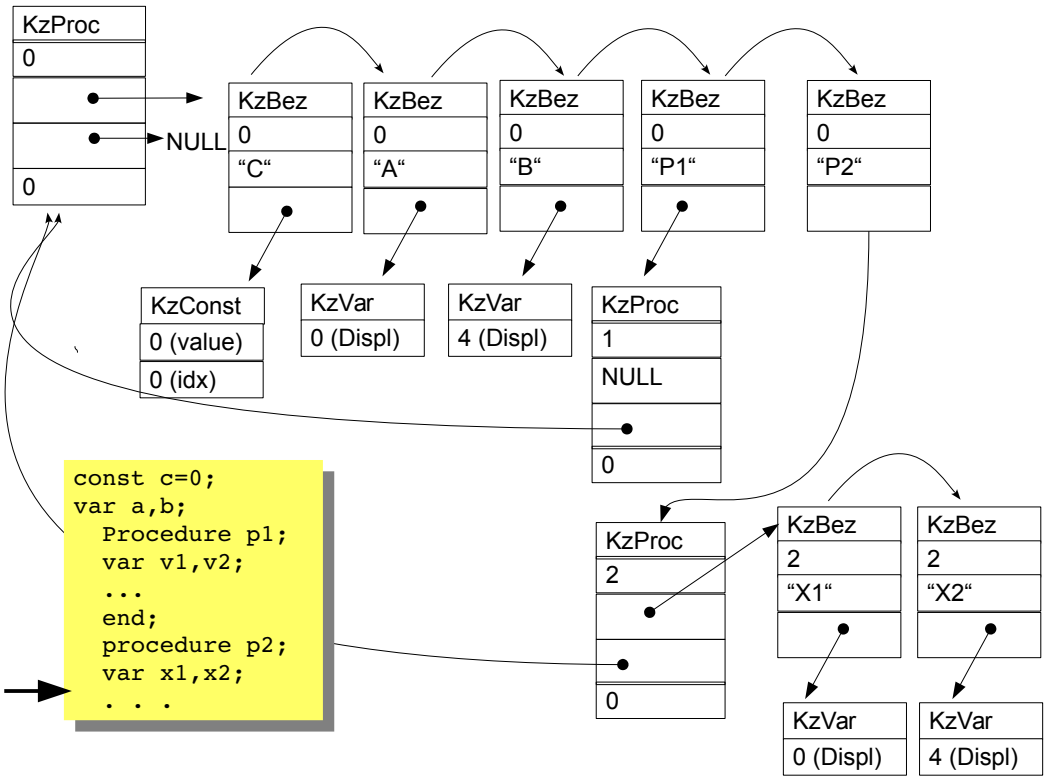
Der Name einer Prozedur steht dabei immer in der übergeordneten Namensliste

```
typedef struct tPROC{
  tKz Kz;
  short IdxProc;          // Prozedurnummer
  struct tPROC*pParent; // Pointer auf die Prozedurbeschreibung der
      Parentprozedur
  tList *pLBez;           // Lokale Namensliste der Prozedur
  int SpzzVar;            // Speicherplatzzuordnungszähler für
     Variable
}tProc;
```

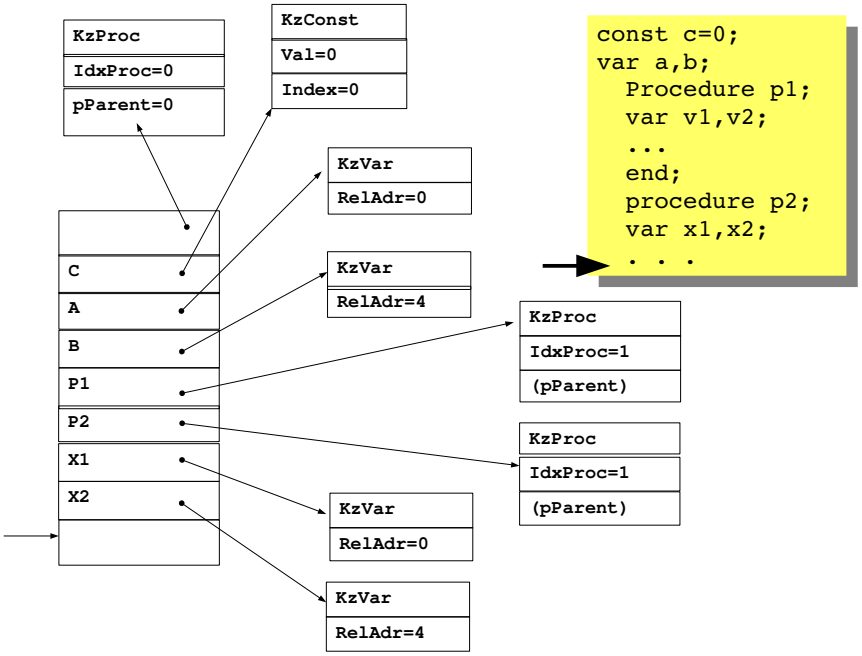
4.6 BEISPIEL NAMENSLISTE



↓



REALISIERUNG ALS STACK

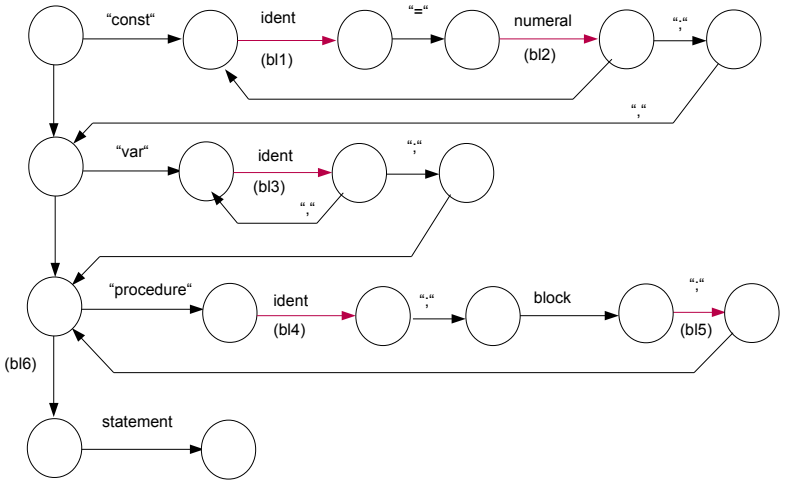


4.7 FUNKTIONEN ZUR NAMENSLISTE

```
tBez* createBez(char*pBez);
tConst* createConst(long Val);
tConst* searchConst(long Val);
int createVar(void);
tProc* createProc(tProc*pParent);
tBez* searchBez(tProc*pProc,char* pBez);
tBez* searchBezGlobal(char* pBez);
```

4.8 EINORDNUNG IN PARSER

bl* wird ausgeführt, wenn Bogen formal akzeptiert wurde.



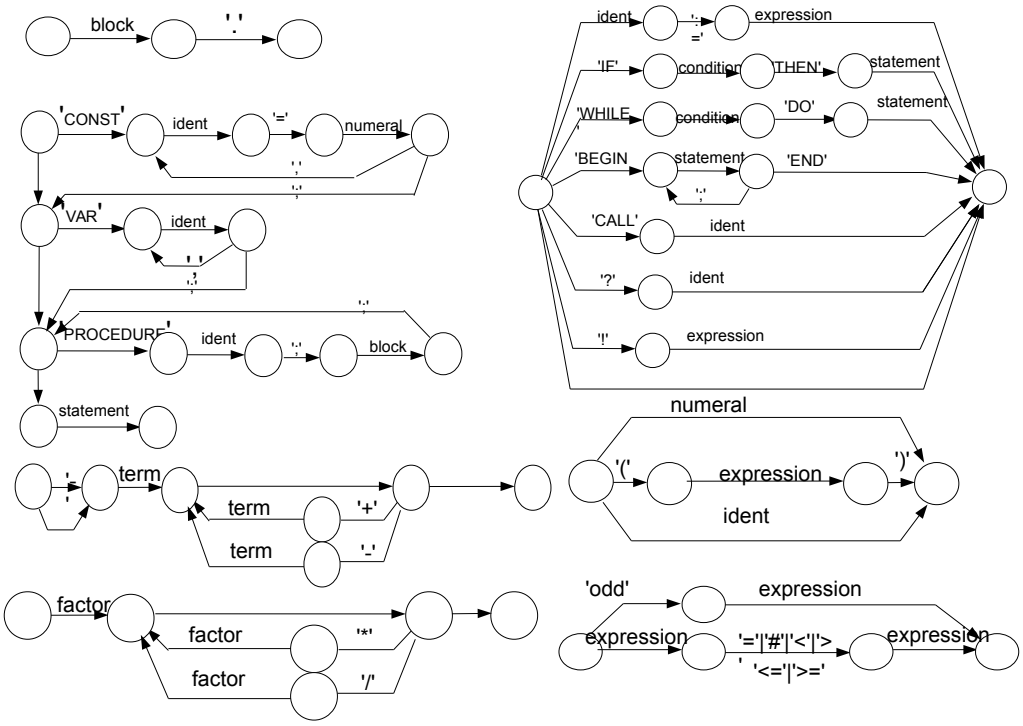
bl1(Konstantenbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen

bl2 (Konstantenwert):
- Konstantenbeschreibung anlegen
- Suche nach Konstante im Konstantenblock
  - gefunden -> Index der Konstanten eintragen in Konstantenbeschreibung
  - Konstante anlegen im Konstantenblock und Index der Konstanten eintragen in Konstantenbeschreibung
- In letzten Bezeichner Zeiger auf Konstante eintragen

bl3 (Variablenbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen
- Variablenbeschreibung anlegen und Pointer in Bezeichner eintragen
- Relativadresse ermitteln aus SpzzVar, SpzzVar um 4 erhöhen (Virtuelle Maschine arbeitet mit 4 Byte langen long-Werten)

bl4(Prozedurbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen
- Prozedurbeschreibung anlegen
- Pointer auf Parent-Prozedur eintragen
- Pointer auf Prozedurbeschreibung in letzten Bezeichner eintragen
- Neue Prozedur ist jetzt aktuelle Prozedur

bl5 (Ende der Prozedurverinbarung):
- Codegenerierung: `retProc`
- Codelänge in den Befehl `entryProc` als 1. Parameter nachtragen
- Code aus dem Codepuffer in die Ausgabedatei schreiben (anfügen)
- Namensliste mit allen Konstanten-, Variablen- und Prozedurbeschreibungen auflösen; die Prozedurbeschreibung selbst muss noch erhalten bleiben
- Die Parent-Prozedur wird die aktuelle Prozedur



5 VIRTUELLE MASCHINE

5.1 GRUNDLAGEN

Vorlesung
29.11.2017

- Eine virtuelle Maschine ist ein Rechner, der physisch nicht existiert. Seine Bestandteile werden durch ein Programm realisiert, das alle Bestandteile eines physischen Rechners nachbildet.

  - Register
  - Speicher
  - Steuerschleife
  - Befehlssatz

5.1.1 DREIADRESSMASCHINE

Eine beispielsweise arithmetische Operation wird durch einen Befehl mit 3 Parametern realisiert, den beiden Operanden und dem Ziel, auf dem das Ergebnis der Operation abgelegt wird. Zwischenergebnisse komplexer Ausdrücke werden in temporären Variablen abgelegt.

Erg ← Op1 + Op2

Erg ← Op1 - Op2

Erg ← Op1 * Op2

Erg ← Op1 / Op2

Erg ← Op1 & Op2

...

5.1.2 ZWEIADRESSMASCHINE

ähnlich der Dreiadressmaschine, jedoch überschreibt das Ergebnis einen der beiden Operanden, so dass der 3. Parameter entfällt.

Op1 ← Op1 + Op2

Op1 ← Op1 - Op2

Op1 ← Op1 * Op2

Op1 ← Op1 / Op2

Op1 ← Op1 & Op2

...

Beispiel ASM x86:

```
xor    eax,eax
add    esi,eax ; Summe
shl    al,3
```

5.1.3 EINADRESSMASCHINE

Alle Operationen werden mit einem Akkumulatorregister ausgeführt, das den jeweils 1. Operanden enthält

Load Operand

Akkumulator += Operand

Akkumulator -= Operand

Akkumulator &= Operand

….

Store Destination

5.1.4 NULLADRESSMASCHINE - STACKMASCHINE

Operanden und Ergebnisse werden auf einem Kellerspeicher abgelegt. Auch die Variablen der Prozeduren werden im Keller angelegt. Sie werden durch eine Relativadresse, die relativ zum Anfang des Variablenbereiches (Adresse der 1. Variablen) gebildet wird, adressiert.

Push op1

Push op2

Push op3

Add

Mul

….

(op3+op2)*op1

Die virtuelle Mascine der LV basiert auf einer Stackmaschine

5.2 BEFEHLSSATZ

- Befehle zum Datentransport (push, pop, store)
- Arithmetische Befehle (add, sub, cmp,...)
- Sprungbefehle(jmp, call, jnot, …)
- Befehle zur Programmorganisation (entryproc)
- I/O-befehle

```
typedef enum TCODE{
/*--- Kellerbefehle ---*/
puValVrLocl,/*00 (short Displ)  [Kellern Wert lokale Variable]               */
puValVrMain,/*01 (short Displ)  [Kellern Wert Main Variable]                 */
puValVrGlob,/*02 (short Displ,short Proc) [Kellern Wert globale Variable]    */
puAdrVrLocl,/*03 (short Displ)  [Kellern Adresse lokale Variable]            */
puAdrVrMain,/*04 (short Displ)  [Kellern Adresse Main Variable]              */
puAdrVrGlob,/*05 (short Displ,short Proc) [Kellern Adresse globale Variable] */
puConst,    /*06 (short Index)  [Kellern einer Konstanten]                   */
storeVal,   /*07 ()              [Speichern Wert ->Adresse, beides aus Keller]*/
putVal,     /*08 ()              [Ausgabe eines Wertes aus Keller nach stdout]*/
getVal,     /*09 ()     [Eingabe eines Wertes von stdin -> Addr. im Keller ] */
/*--- arithmetische Befehle ---*/
vzMinus,    /*0A ()              [Vorzeichen ]                                */
odd,        /*0B ()              [ungerade -> 0/1]
   */
```

```
16 /*--- binäre Operatoren kellern 2 Operanden aus und das Ergebnis ein ---*/
17 OpAdd,       /*0C ()              [Addition]                                  */
18 OpSub,       /*0D ()              [Subtraktion ]                              */
19 OpMult,      /*0E ()              [Multiplikation ]                           */
20 OpDiv,       /*0F ()              [Division ]                                 */
21 cmpEQ,       /*10 ()              [Vergleich =  -> 0/1]                       */
22 cmpNE,       /*11 ()              [Vergleich #  -> 0/1]                       */
23 cmpLT,       /*12 ()              [Vergleich <  -> 0/1]                       */
24 cmpGT,       /*13 ()              [Vergleich >  -> 0/1]                       */
25 cmpLE,       /*14 ()              [Vergleich <= -> 0/1]                       */
26 cmpGE,       /*15 ()              [Vergleich >= -> 0/1]                       */
27 /*--- Sprungbefehle ---*/
28 call,        /*16 (short ProzNr) [Prozeduraufruf]                            */
29 retProc,     /*17 ()              [Rücksprung]                                */
30 jmp,         /*18 (short RelAdr) [SPZZ innerhalb der Funktion]               */
31 jnot,        /*19 (short RelAdr) [SPZZ innerhalb der Funkt.,Beding.aus Keller] */
32 entryProc,   /*1A (short lenCode,short ProcIdx,short lenVar)                 */
33 /*--- Zusätzliche, (für uns) nicht benötigte Befehle ---*/
34 putStrg,     /*1B (char[])                                                   */
35 pop,         /*1C                                                          */
36 swap,        /*1D                 Austausch Adresse gegen Wert             */
37 EndOfCode    /*1E                                                          */
38 } tCode;
```

5.2.1 ARBEITSWEISE DER STACKMASCHINE

PL/0:

```
const c=3;
var   a,b;
begin
  ?b;
  a:=B*10+c;
  !a
end
.
```

```
Code:

Procedure
00000000: 1A EntryProc          001F,0000,0008
00000007: 04 PushAdrVarMain     0004   <- Adresse Variable B
0000000A: 09 GetVal

0000000B: 04 PushAdrVarMain     0000   <- Adresse Variable A
0000000E: 01 PushValVarMain     0004   <- Wert Variable B
00000011: 06 PushConst          0001
00000014: 0E Mul
00000015: 06 PushConst          0000
00000018: 0C Add
00000019: 07 StoreVal
0000001A: 01 PushValVarMain     0000

0000001D: 08 PutVal
0000001E: 17 ReturnProc
Const 0000:0003
Const 0001:0010
```

5.3 DATENSTRUKTUREN DER VM

- Speicherbereich für den Stack: Wird beim Start angelegt, während der Programmabarbeitung überwacht und bei Bedarf vergrößert.
- Speicherbereich für den Code und die Konstanten: Wird beim Laden des Zwischencodefiles angelegt.
- Prozedurtabelle: wird nach dem Laden des Zwischencodefiles angelegt. Sie enthält die Anfangsadressen der Prozeduren sowie ein zunächst leere Felder für die Anfangsadresse des Variablenbereiches (Stackframe der Prozedur), das durch ***entryProc*** beim Aufruf der Prozedur belegt wird.
- Die Einträge der Prozedurtabelle sind nach Prozedurnummer aufsteigend geordnet, so dass die Prozedurtabelle durch die Prozedurnummer indiziert werden kann.

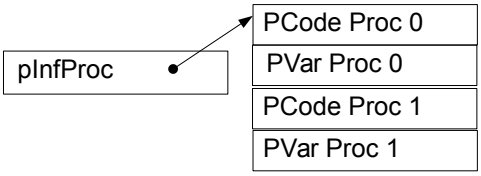
5.4 REGISTER DER VM

Register	*Variable der VM*
ProgrammCounter	***pC***
StackPointer	***pS***
KonstantenPointer	***pConst***
Aktuelle Prozedur	***iCProc***

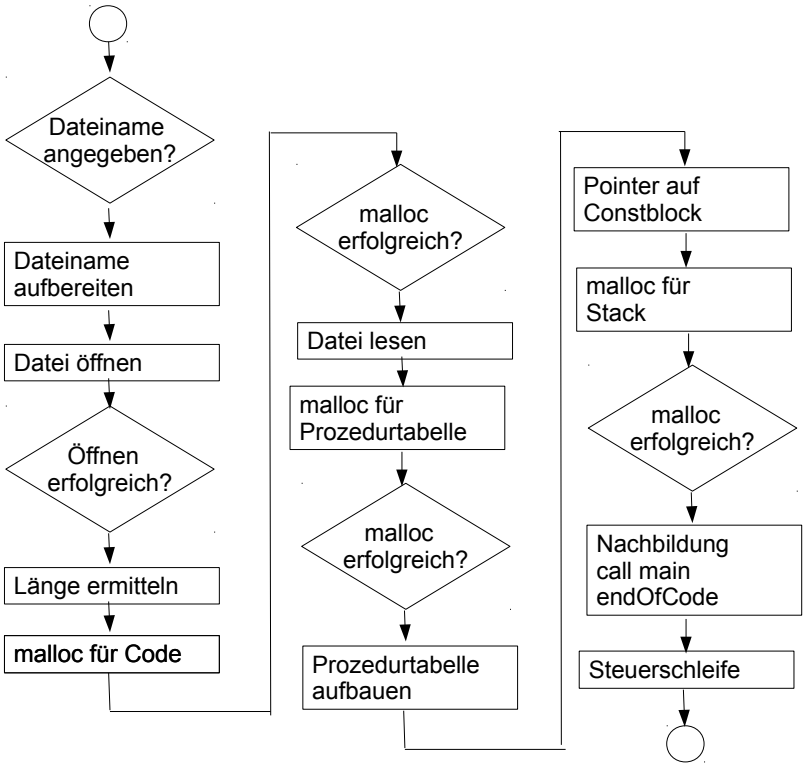
Pointer auf Prozedurtabe	***pInfProc***

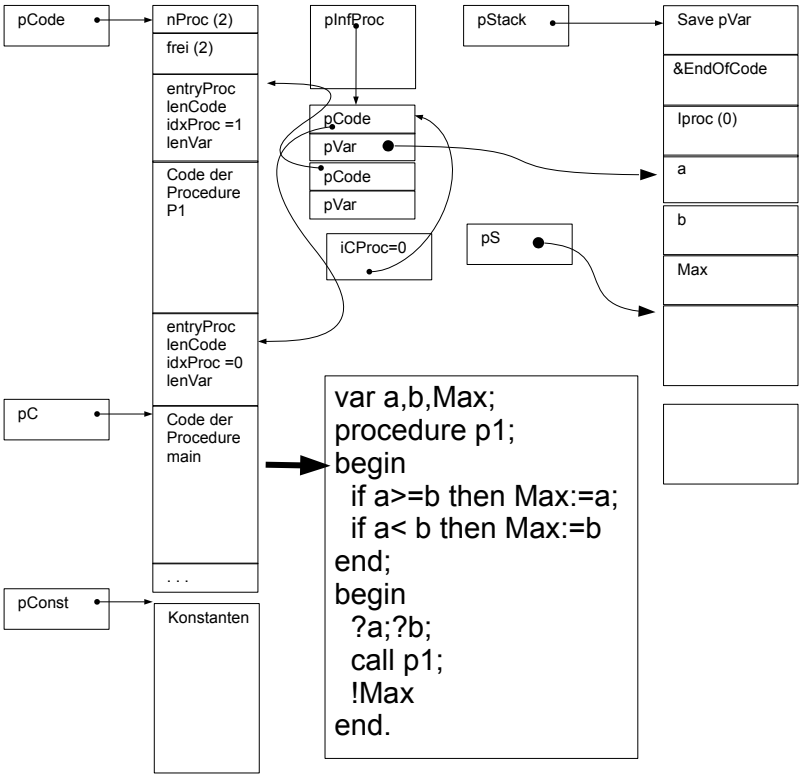
5.5 PROZEDURTABELLE

- Enthält für jede Prozedur 2 Einträge (Pointer)
  - Startadresse des Codes (Adresse von EntryProc der Prozedur), wird beim Programmladen eingetragen
  - Adresse des Variablenbereiches der Prozedur (Adresse der Variablen mit Offset 0), wird bei Prozeduraufruf zur Laufzeit eingetragen
- Tabelle wird durch die Prozedurnummer indiziert



5.6 INITIALISIERUNG





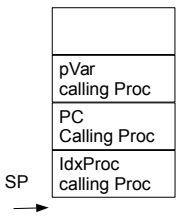
5.7 STEUERSCHLEIFE

```
typedef int (*fx)(void);
fx vx[]={
  FcpuValVrLocl, FcpuValVrMain,
  FcpuValVrGlob, FcpuAdrVrLocl,
  FcpuAdrVrMain, FcpuAdrVrGlob,
  FcpuConst, FcstoreVal,
  FcputVal, FcgetVal,
  FcvzMinus, Fcodd,
  FcOpAdd, FcOpSub,
  FcOpMult, FcOpDiv,
  FccmpEQ, FccmpNE,
  FccmpLT, FccmpGT,
  FccmpLE, FccmpGE,
  Fccall, FcRetProc,
  Fcjmp, Fcjnot,
  FcEntryProc, FcputStrg,
  Fpop, Fswap,
  FcEndOfCode, Fcput,
  Fcget
};

while (!Ende){
  vx[*pC++]();
}

```

```
while (!Ende){
  switch (*pC++){
    case puValVrLocl:FcpuValVrLocl(); break;
    case puValVrMain:FcpuValVrMain(); break;
    case puValVrGlob:FcpuValVrGlob(); break;
    case puAdrVrLocl:FcpuAdrVrLocl(); break;
    case puAdrVrMain:FcpuAdrVrMain(); break;
    case puAdrVrGlob:FcpuAdrVrGlob(); break;
    case puConst :FcpuConst(); break;
    case storeVal :FcstoreVal(); break;
    case putVal :FcputVal(); break;
    case getVal :FcgetVal(); break;
    case vzMinus :FcvzMinus(); break;
    case odd :Fcodd(); break;
    case OpAdd :FcOpAdd(); break;
    case OpSub :FcOpSub(); break;
    case OpMult :FcOpMult(); break;
    case OpDiv :FcOpDiv(); break;
    /* ... */
    case pop :Fpop(); break;
    case swap :Fswap(); break;
    case EndOfCode :FcEndOfCode(); break;
    case put :Fcput(); break;
    case get :Fcget(); break;
  }
}
```

5.8 PROZEDURAUFRUF

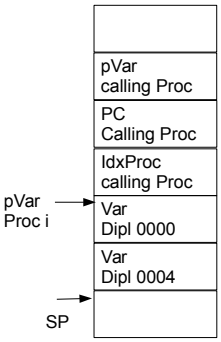
- Call PrNr
  - Kellern des Zeigers auf Variablenbereich der rufenden Prozedur aus der Prozedurtabelle.
  - Kellern des aktuellen Befehlszählers (zeigt jetzt hinter call)
  - Kellern der Prozedurnr. der rufenden Procedure
  - Ausführen des Sprunges durch Eintragen der Adresse des Befehls entyProc der aufzurufenden Prozedur




```
int Fccall(void){
  short ProcNr;
  ProcNr=gtSrtPar(pC);pC+=2;
  push4((int4_t)(pInfProc[iCProc].pVar));
  push4((int4_t)pC);
  pC=pInfProc[ProcNr].pCode;//Sprung zu entryProc
  push4(iCProc);
```

```
  return OK;
}
int FcRetProc(void){
  pS=pInfProc[iCProc].pVar;
  iCProc =pop4();
  pC =(char*)pop4();
  pInfProc[iCProc].pVar=(char*)pop4();
  return OK;
}
```

- EntryProc
  - Setzen der aktuellen Prozedur
  - Einrichten des Variablenbereiches im Stack
  - Setzen des Zeigers auf Variablenbereich in der Prozedurtabelle
- Die Verwaltung der Variablenbereiche über dieProzedurtabelle ermöglicht den rekursiven Prozeduraufruf, wobei immer der richtige Satz von Variablen benutzt wird




```
/* Entry Procedure */
int FcEntryProc(void){
  short lVar;
  /* Codelaenge ueberlesen */
  PC+=2;
  /* Prozedurnummer lesen und einstellen */
  iCProc=gtSrtPar(pC);pC+=2;
  /* Variablenlaenge lesen */
  lVar =gtSrtPar(pC);pC+=2;
  /* Zeiger auf Variablenbereich */
  pInfProc[iCProc].pVar=pS;
  /* Neuer Stackpointer */
  pS+=lVar;
  return OK;
}
```

5.9 REKURSION

Rekusion mit globaler Variable

```
var x;
  procedure r;
  var a;
   procedure p;
   !a;
  if x<5 then
  begin
   x:=x+1;
   a:=x;
   call p;
   call r;
   call p
  end;
begin
x:=0;
call r
end.
```

Berechnung der Fakultät

```
var a,fac;

Procedure p1;
 var b,c;
 begin
  b := a;
  a := a-1;
  c := a;
  !c;
  if c>1 then call p1;
  fac := fac*b;
  !fac
 end;

begin
  ?a;
  fac := 1;
  call p1;
  !fac
end.
```

5.10 PROZEDUR MIT PARAMETERN

Parameter wird vor Aufruf auf den Stack geschrieben. Dieser hat die relative Adresse -16 (die erste lokale Variable hat die relative Adresse 0, Die Adresse der Funktion sind 3 Byte, daher haben dann Parameter die Adresse ab -16).

```
var x,y;
  procedure facult(a);
  begin
    if a-1>1 then call facult(a-1);
    y:=y*a
  end;
begin
  ?x;
  y:=1;
  call facult(x);
  !y;
  !"\n"
end.
```

6 CODEGENERIERUNG

Vorlesung
06.12.2017

6.1 GRUNDLAGEN

Code wird nacheinander für jede Prozedur gesondert erzeugt.

Code wird innerhalb von Block für statement generiert.

Ist der Code für eine Prozedur generiert, kann er in die Ausgabedatei geschrieben werden.

Vor der eigentlichen Codegenerierung ist der Befehl **`entryProc(Codelen,ProcIdx,VarLen)`** zu generieren, dies erfolgt in dem Nil-Bogen vor statement. CodeLen bleibt dabei zunächst 0, VarLen ist dem SpzzVar zu entnehmen.

Die Codegenerierung erfolgt durch die Aktionen, wenn Bögen akzeptiert worden sind.

6.2 FUNKTIONEN ZUR CODEGENERIERUNG

int code(char OpCode, …);

Befehle haben 0, 1, 2 oder 3 Parameter.

Parameter sind alle vom Typ short (2 Byte).

Operationscodes sind 1 Byte groß.

Die Parameter werden in derr Byteorder little endian (intel) gespeichert.

Soll der Compiler portabel sein (und das sollte er), muss eine Funktion zum Schreiben der Parameter gebaut werden.

```
1  // Schreibt am aktuellen Programmcounter
2  void wr2ToCode(short x){
3    *pCode++=(unsigned char)(x & 0xff);
4    *pCode++=(unsigned char)(x >> 8);
5  }
6  // Schreibt an der übergebenen Stelle
7  void wr2ToCodeAtP(short x,char*pD){
8    * pD =(unsigned char)(x & 0xff);
9    *(pD+1)=(unsigned char)(x >> 8);
10 }
11 // Schreibt Befehl mit 0, 1, 2 oder 3 Parametern in den
     Codeausgabepuffer
12 int code(tCode Code,...){
```

```
13    va_list ap;
14    short sarg;
15    if (pCode-vCode+MAX_LEN_OF_CODE>=LenCode){
16      char* xCode=realloc(vCode,(LenCode+=1024));
17      if (xCode==NULL) Error(ENoMem);
18      pCode=xCode+(pCodevCode);
19      vCode=xCode;
20    }
21    *pCode++=(char)Code;
22    va_start(ap,Code);
23    switch (Code){
24      /* Befehle mit 3 Parametern */
25      case entryProc:
26        sarg=va_arg(ap,int);
27        wr2ToCode(sarg);
28      /* Befehle mit 2 Parametern */
29      case puValVrGlob:
30      case puAdrVrGlob:
31        sarg=va_arg(ap,int);
32        wr2ToCode(sarg);
33      /* Befehle mit 1 Parameter */
34      case puValVrMain:
35      case puAdrVrMain:
36      case puValVrLocl:
37      case puAdrVrLocl:
38      case puConst:
39      case jmp :
40      case jnot:
41      case call:
42        sarg=va_arg(ap,int); // Prozedurnummer
43        wr2ToCode(sarg);
44        break;
45      /* keine Parameter */
46      default :
47        break;
48      }
49    va_end (ap);
50    return OK;
51  }
```

6.3 VARIABLEN ZUR CODEGENERIERUNG

Prozedurnummer der aktuellen Prozedur

Codeausgabebereich (mit Füllstand durch Pointer oder Index)

Konstantenblock

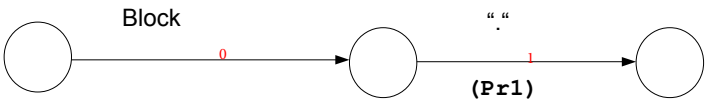
Datenstrukturen der Namensliste

6.3.1 ENTRYPROC

Vor der eigentlichen Codegenerierung ist der Befehl `entryProc(Codelen, ProcIdx, VarLen)` zu generieren, dies erfolgt in dem Nil-Bogen vor statement (siehe Unterabschnitt 6.3.3: bl6). CodeLen bleibt zunächst 0, ProcEdx ist die aktuelle Prozedur und VarLen ist dem SpzzVar zu entnehmen.

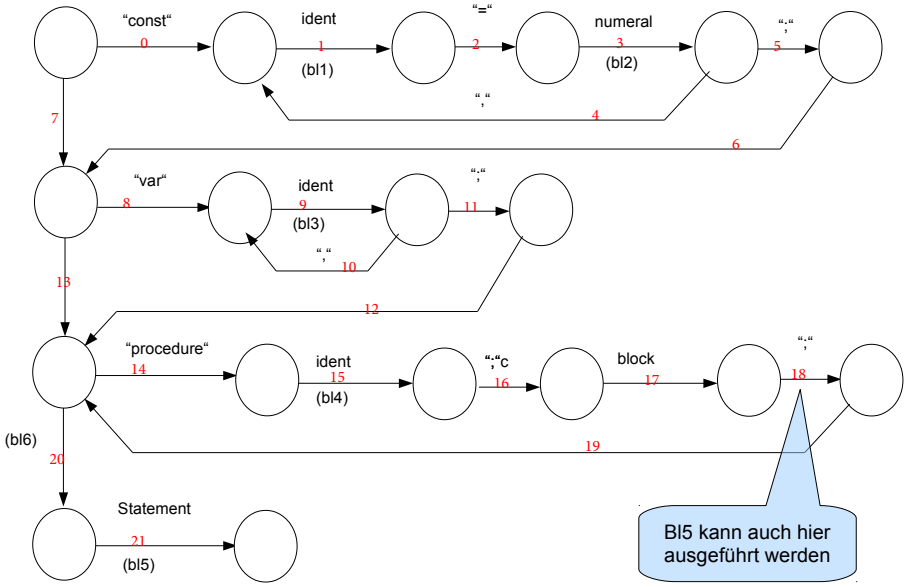
6.3.2 PROGRAMM

Nur eine Aktion, wenn die Kompilierung bis hier gekommen ist → Abschlussarbeiten.



Pr1:
- Aufruf von bl5, wenn bl5 bei ';' und nicht bei statement augerufen wurde
- Schreiben des Konstantenblocks in das Codefile
- Schreiben der Anzahl der Prozeduren in das Codefile am Anfang

6.3.3 BLOCK



bl1(Konstantenbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen

bl2 (Konstantenwert):
- Konstantenbeschreibung anlegen
- Suche nach Konstante im Konstantenblock
  - gefunden -> Index der Konstanten eintragen in Konstantenbeschreibung
  - Konstante anlegen im Konstantenblock und Index der Konstanten eintragen in Konstantenbeschreibung
- In letzten Bezeichner Zeiger auf Konstante eintragen

bl3 (Variablenbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen
- Variablenbeschreibung anlegen und Pointer in Bezeichner eintragen
- Relativadresse ermitteln aus SpzzVar, SpzzVar um 4 erhöhen (Virtuelle Maschine arbeitet mit 4 Byte langen long-Werten)

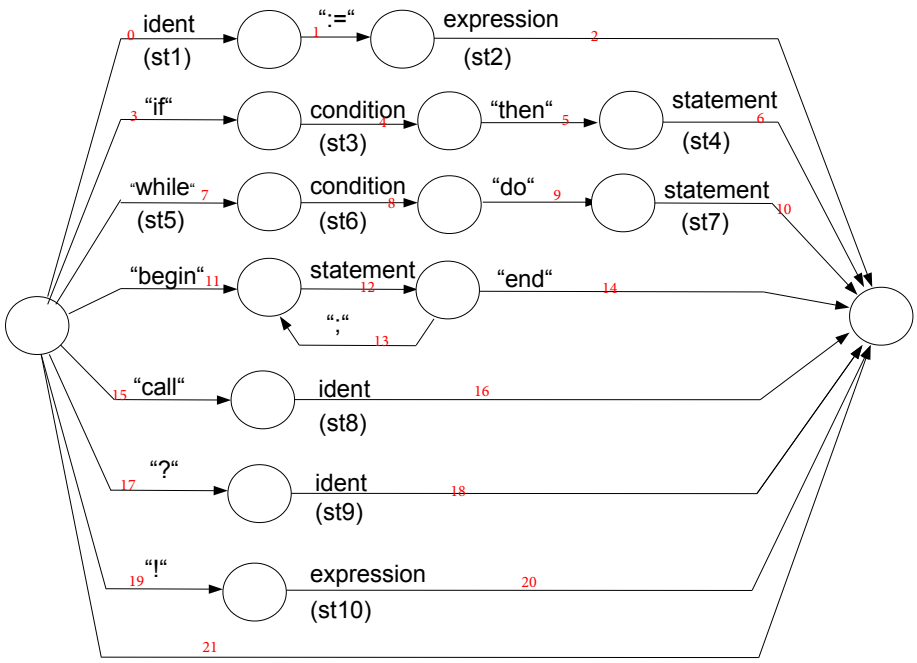
bl4(Prozedurbezeichner):
- lokale Suche nach dem Bezeichner
  - gefunden -> Fehlerbehandlung
  - nicht gefunden -> Bezeichner anlegen
- Prozedurbeschreibung anlegen
- Pointer auf Parent-Prozedur eintragen
- Pointer auf Prozedurbeschreibung in letzten Bezeichner eintragen
- Neue Prozedur ist jetzt aktuelle Prozedur

bl5 (Ende der Prozedurvereinbarung):
- Codegenerierung: **`retProc`**
- Codelänge in den Befehl **`entryProc`** als 1. Parameter nachtragen
- Code aus dem Codepuffer in die Ausgabedatei schreiben (anfügen)
- Namensliste mit allen Konstanten-, Variablen- und Prozedurbeschreibungen auflösen; die Prozedur selbst muss noch erhalten bleiben
- Die Parent-Prozedur wird die aktuelle Prozedur

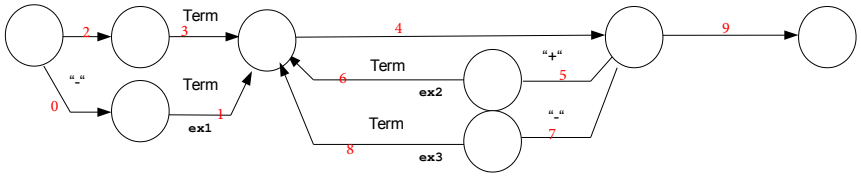
Die Aktionen von bl5 könnten auch nach Akzeptanz des Bogens Statement ausgeführt werden.

bl6(Beginn des Anweisungsteils):
Hier muss ggf. ein Nilbogen eingefügt werden, um diese semantische Aktion an dieser Stelle ausführen zu können!
Codeausgabepuffer initialisieren
Codegenerierung: entryProc(CodeLen, IdxProc, VarLen) mit CodeLen zunächst 0, IdxProc mit pCurrProc->Idx, und VarLen aus SpzzVar.

6.3.4 STATEMENT



6.3.5 EXPRESSION



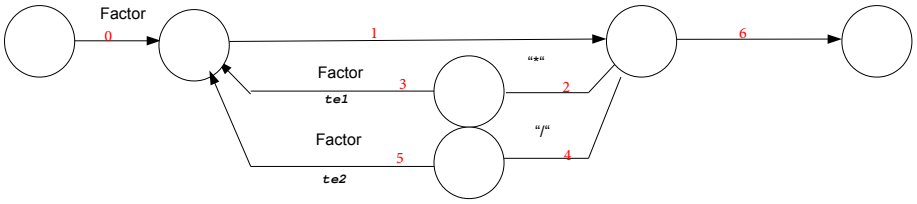
ex1 (negatives Vorzeichen)
- Codegenerierung ***vzMinus***

ex2
- Codegenerierung opAdd

ex3
- Codegenerierung ***opSub***

Anmerkung:
Es werden hier nur die additiven Operatoren generiert. Der Code zum Kellern der Operanden wurde bereits innerhalb von Term→Faktor erzeugt.

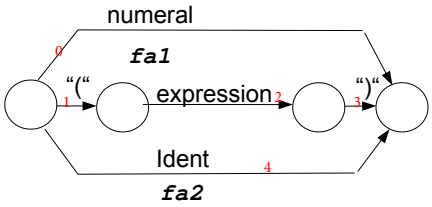
6.3.6 TERM



te1
Codegenerierung ***opMul***
te2
Codegenerierung ***opDiv***

Anmerkung:
Es werden hier nur die Multiplikativen Operatoren generiert. Der Code zum Kellern der Operanden wurde bereits innerhalb von Faktor erzeugt.

6.3.7 FACTOR

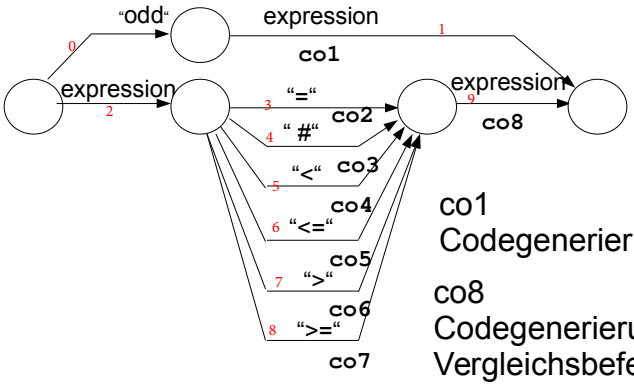


fa1 (Numeral):
suchen der Konstante
ggf. anlegen der Konstanten, wenn nicht gefunden
Codegenerierung
puConst (ConstIndex)

fa2 (Ident):
Bezeichner global suchen
nicht gefunden -> Fehlerbehandlung
gefunden -> ok.
Bezeichnet der Bezeichner eine Variable oder Konstante?
Nein, eine Prozedur -> Fehlerbehandlung
ja -> ok,
Codegenerierung :

Bezeichner gefunden	Code	Parameter
LovalVar	puValVrLocl	displ
Main Var	puValVrMain	displ
Global Var	puValVrGlob	displ, ProcedureNr
Konstante	puConst	index

6.3.8 CONDITION



co1
Codegenerierung odd

co8
Codegenerierung des gespeicherten Vergleichsbefehles

co2 – co7
der später zu generierende Code für den Vergleichsoperator wird in einer einfachen Variable gespeichert. Dies ist hier möglich, da expression keinen Vergleich enthalten kann. Anderenfalls wäre ein Umbau der Syntaxbeschreibung, ähnlich expression notwendig, oder die Operatoren müssen in einem gesonderten Stack verwaltet werden.

6.3.9 AUSZUFÜHRENDE FUNKTIONEN IN STATEMENT

6.3.9.1 ZUWEISUNGEN

stx wird jeweils ausgeführt, nachdem die beschriebenen Handlungen schon vollzogen wurden (ident, expression, . . .)

st1(Linke Seite der Zuweisung):
Bezeichner global suchen
nicht gefunden -> Fehlerbehandlung
gefunden -> ok.
Bezeichnet der Bezeichner eine Variable?
Nein, eine Konstante oder Prozedur -> Fehlerbehandlung
ja -> ok,
Codegenerierung :

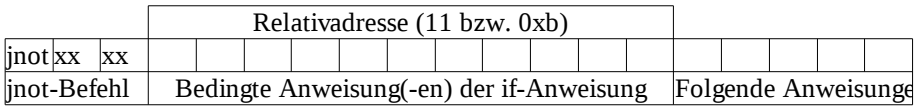
Bezeichner gefunden	Code	Parameter
Local	***PushAdrVarLocal***	Relativadresse
Im Hauptprogramm	***PushAdrVarMain***	Relativadresse
In umgebender Prozedur	***PushAdrVarGlobal***	Relativadresse, Prozedurnummer

st2 (Rechte Seite der Zuweisung):
Im Stack steht nun die Adresse der Zielvariable und der Wert des fertig berechneten Ausdrucks.
Codegenerierung : storeVal

6.3.9.2 CONDITIONAL STATEMENT

st3 (if, nach Condition)
Generieren eines Labels, es zeigt auf den nächsten freien Speicherplatz des Codeausgabepuffers.
Codegenerierung jnot mit einer vorläufigen Relativadresse 0

st4 (if, nach Statement)
Label auskellern
Relativadresse berechnen
(Es ist hilfreich, sich hier die Arbeitsweise von jmp /jnot zu verdeutlichen. Die als Parameter des jmp-Befehls übergebene Relativadresse wird zum aktuellen Instructionpointer addiert und in den Instructionpointer geschrieben.)

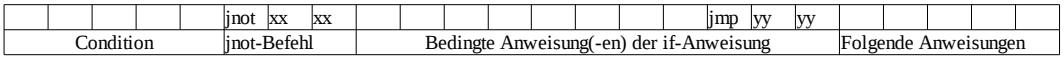


Die Relativadresse in der Skizze würde 11 (0xb) betragen
Relativadresse in jmp-Befehl eintragen.

Beispielcode:

```
// Procedure
0000: 1A  EntryProc              0028,0000,0004
// ?a
0007: 04  PushAdrVarMain         0000
000A: 09  GetVal
// if a>10 then
000B: 01  PushValVarMain         0000
000E: 06  PushConst              0000
0011: 13  CmpGreaterThen
0012: 19  JmpNot                 0004
// !00
0015: 06  PushConst              0001
0018: 08  PutVal
// if a<=10 then
0019: 01  PushValVarMain         0000
001C: 06  PushConst              0000
001F: 14  CmpLessEqual
0020: 19  JmpNot                 0004
// !1
0023: 06  PushConst              0002
0026: 08  PutVal

0027: 17  ReturnProc
Const 0000:0010
Const 0001:0100
Const 0002:0001
```

6.3.9.3 LOOP STATEMENT



st5 (while)
Generieren eines Labels für Rücksprung am Schleifenende.

st6 (while, nach Condition)
Generieren eines Labels, es zeigt auf den nächsten freien Speicherplatz des Codeausgabepuffers.
Codegenerierung jnot mit einer vorläufigen Relativadresse 0

st7 (while, nach Statement)
Label auskellern
Relativadresse berechnen, so ähnlich, wie bei st4, jedoch müssen 3 Byte für den jmp-Befehl am Ende der while-Anweisung freihalten werden (zu der berechneneten Relativadresse muss die Länge des jmp-Befehls addiert werden addiert werden)
2. Labelauskellern und jmp-Befehl generieren. Die Relativadresse muss so berechnet werden, dass der Sprung bei dem 1. Befehl von Condition landet.

6.3.9.4 PROCEDURE CALL

st8 Prozeduraufruf:

Bezeichner global suchen
Nicht gefunden -> Fehlerbehandlung
Gefunden -> ok.

Bezeichnet der Bezeichner eine Procedure?
Nein, eine Konstante oder Variable -> Fehlerbehandlung
Ja -> ok

Codegenerierung call procedurenumber

6.3.9.5 EINGABE/AUSGABE

st9 Eingabe:
Bezeichner global suchen:
nicht gefunden -> Fehlerbehandlung
gefunden -> ok.
Bezeichnet der Bezeichner eine Variable?
Nein, eine Konstante oder Prozedur -> Fehlerbehandlung
ja -> ok,
Codegenerierung :

Bezeichner gfunden	Code	Parameter
Local	***PushAdrVarLocal***	Relativadresse
Im Hauptprogramm	***PushAdrVarMain***	Relativadresse
In umgebender Prozedur	***PushAdrVarGlobal***	Relativadresse, Prozedurnummer

Codegenerierung : getVal

st10 Ausgabe:
Auszugebender Wert steht im Stack.
Codegenerierung: putVal

7 BEISPIELCOMPILIERUNG

Vorlesung
13.12.2017

7.1 ANZULEGENDE SPEICHERBEREICHE

- Morphem
- ProcMain (Prozedurbeschreibung der Main Funktion)
- Speicherbereich für Konstanten
- Speicherbereich für Zwischencode
- Ausgabedatei

7.2 ABARBEITUNG

- Lexer erkennt Morphem
- Parser startet mit Startbogen
- Parser ruft Blockbogen auf (rekursiver Aufruf von parse)
- Vor dem erstellen einer Konstanten/Variablen: Prüfen, ob es schon einen mit gleichen Namen gibt.
- Einfügen einer Konstante: erst prüfen, ob Wert bereits vorhanden. Konstanten mit gleichem Wert zeigen auf gleichen Speicherbereich
- Zuweisung einer Variablen: pusht Konstante. D.h. Konstantenspeicherbereich wird ggf. erweitert, wenn Wert noch nicht drin (als nicht benannter Eintrag)

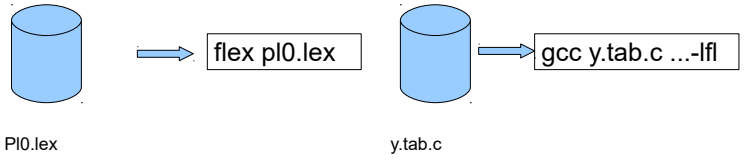
8 LEX/FLEX

8.1 GRUNDLAGEN

Vorlesung
10.01.2018

- Klassische Unix Werkzeuge
- Lexergenerator für Compiler/Interpreter
- Konzipiert für das Zusammenwirken mit den Parsergeneratoren yacc und bison
- Scannergenerator für Commandlinearguments
- Morpheme/Token werden mit Hilfe regulärer Ausdrücke beschrieben
- Lexer kann als Unterprogramm jedes erkannte Token liefern oder als Pass die gesamte Quelle scannen.
- Erzeugt ein c-Programm

8.1.1 VERWENDUNG



- Option erlauben Variationen flex -o t1.c t1.lex
- Aufpassen: Optionen vor lex-Datei angeben!!!
  - -o outputfile
  - Generierung einer C++-Scannerklasse
    (--yyclass=NAME –c++)
  - Option -i generiert einen nicht casesensitiven Scanner
  - Weitere Optionen unter man flex
  - Generierung eines c-headerfiles
    flex --header-file=lex.h tz5.lex

8.1.2 REGULÄRE AUSDRÜCKE

- Beschreiben Zeichenfolgen eines Alphabetes
- Operationen zur Beschreibung sind dabei:
  - Die Aneinanderreihung (Konkatenation)
  - Die Unterscheidung (Alternative)
  - Die Wiederholung (Iteration)
  - Die Verneinung (Negation)
- Häufig gelten dabei Vorrangregeln, wobei die Operatorpriorität von Konkatenation zu Iteration steigt.
- Im Bedarfsfall kann geklammert werden

- Neben den Operationen müssen auch die vorkommenden Zeichen beschrieben werden.
- Zeichenfolgen (abc, a1, while, 123)
- Klassen von Zeichen ([0-9], [A-Z], [1,3,5,7,9])
- Beliebiges Zeichen außer newline (.)
- Escape Sequenz \... (\n, \t, \0x41)

8.1.3 BESONDERE ZEICHEN

- . Jedes Zeichen außer \n wird akzeptiert
- ^ als erstes Zeichen: Anfang einer neuen Zeile
- [^...] alles außer … wird akzeptiert
- $ Als letztes Zeichen eines Ausdrucks wird das Zeilenende akzeptiert
- <...> Markiert am Regeleanfang (1. Zeichen) einen speziellen Status, der mit BEGIN eingestellt wird. Die Regel ist nur gültig, wenn der angegebene Status eingeschaltet ist.

8.1.4 WIEDERHOLUNGEN

- * der vor * stehende (Teil-)Ausdruck kommt 0x, 1x oder mehrfach vor
- + der vor + stehende (Teil-)Ausdruck kommt mindestens ein mal vor
- ? der vor ? stehende (Teil-)Ausdruck kann vorkommen
- {n} der vor {n} kommt n mal vor
- {n,m} der vor {n,m} kommt mindestens n, höchstens m mal vor
- "..." markiert eine Zeichenkette, die in dieser Form zu akzeptieren ist

8.1.5 REIHENFOLGE DER PATTERN

- Reihenfolge der Patternlines ist relevant
- Patternlines werden von oben nach unten getestet.
- Ist ein Token erkannt, so werden die dazugehörigen Zeichen aus dem Eingabestrom entfernt.
- Daraus folgt:
- Patternlines für Schlüsselwörter am Anfang
- Patternlines für Identifier später

8.2 AUFBAU EINER FLEX-QUELLDATEI

```
definition division
%%
rules division
%%
functions division
```

flex erlaubt Kommentierung im C-Stil mit /* … */ in allen drei Sektionen.
Die Kommentarzeilen müssen mit einem Leerzeichen beginnen!
C-Code wird geklammert in
%{
%}
Oder beginnt mit wenigstens einem Leerzeichen

8.2.1 RULES DIVISION

- Besteht aus Patternlines
- Patternlines beginnen mit einem regulären Ausdruck oder einer Startcondition
- C-Code kann sich nach mind. einem Leerzeichen anschließen, bei mehr als einer Zeile als Block

8.2.2 VORDEFINIERTE SYMBOLE

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

flex:	
yy_scan_string(const char* pstr)	
yy_scan_bytes(const char *bytes, int len)	

8.3 BEISPIELE

8.3.1 BEISPIEL LEERZEICHEN

ersetzen mehrerer white spaces durch ein Leerzeichen

```
%%
[ \t]+   printf(" ");
%%
main()
{
  yylex();
}
int yywrap()
{ return 1;}
```

Aufruf des Lexers

Funktion zum Umschalten der Eingabedatei, kann entfallen, -lfl stellt dann eine defaultfunktion bereit

```
T1.dat:das  ist    ein Text       mit viel  Freiraum.

beck@Examples> lex t1.lex
beck@Examples> gcc lex.yy.c -lfl
beck@Examples> ./a.out <t1.dat
das ist ein Text mit viel Freiraum.
```

8.3.2 ZEICHEN/ZEILEN ZÄHLEN

```
%{
int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

Leider

Ein Mensch sieht schon seit Jahren klar:
Die Lage ist ganz unhaltbar.
Allein - am längsten, leider, hält
das unhaltbare auf der Welt.

```
beck@Examples> ./a.out < leider.txt
# of lines = 6, # of chars = 148
beck@Examples>
```

8.3.3 WÖRTER ZÄHLEN

```
%{
#include <string.h>
int num_lines = 0, num_nums = 0;
int num_chars = 0, num_words= 0;
%}
%%
\n          ++num_lines; ++num_chars;
[a-zA-Z]+   ++num_words; num_chars+=strlen(yytext);
[0-9]+      ++num_nums;  num_chars+=strlen(yytext);
%%
main()
{
  yylex();
  printf( "# of lines    = %d\n",num_lines);
  printf( "# of words    = %d\n",num_words);
  printf( "# of numerals = %d\n",num_nums);
  printf( "# of chars    = %d\n",num_chars );
  return 0;
}
```

+: Voranstehendes Zeichen(oder Vertreter der Klasse) muss Mindestens 1x vorkommen

```
beck@Examples> ./a.out <leider.txt
# of lines    = 6
# of words    = 23
# of numerals = 0
# of chars    = 118
```

8.3.4 BEISPIEL REGELN

```
%{
#include <math.h>
%}
%s expect

%%
floats        BEGIN(expect);

<expect>[0-9]+.[0-9]+  {
            printf( "found a float, = %f\n",
                    atof( yytext ) );
            }
<expect>\n  {
            /* end of the line, so we need another "expect-floats"
             * before we'll recognize any more numbers */
            BEGIN(INITIAL);
            }

[0-9]+      {
            printf( "found an integer, = %d\n",
                    atoi( yytext ) );
            }

"."         printf( "found a dot\n" );
```

```
floats 1.3
 found a float, = 1.300000
1.3
found an integer, = 1
found a dot
found an integer, = 3
```

8.4 LEXER FÜR PL/0 COMPILER

```
%{
/* Deklarationsteil */
/*
lexikalische Analyse mit lex für
graphengesteuerten PL/0 Einpasscompiler
*/
#include "lex.h"
#include "list.h"
#include "debug.h"
extern tMorph Morph; /* globale Morpemvariable */
FILE * pIF; /* Eingabedatei */
void MorSo(int Code); /* Function zum Bau eines SymbolTokens */
%}
%%

/* Leer- und Trennzeichen */
[ \t]+
/* Zeilenwechsel */
[\n] {Morph.PosLine++;}
/* Schluesselwoerter (Wortsymbole) */
/* werden wie Sonderzeichen behandlt */
"begin" {MorSo(zBGN);return;}
call {MorSo(zCLL);return;}
const {MorSo(zCST);return;}
do {MorSo(zDO); return;}
else {MorSo(zELS);return;}
end {MorSo(zEND);return;}
if {MorSo(zIF); return;}
odd {MorSo(zODD);return;}
procedure {MorSo(zPRC);return;}
```

```
then {MorSo(zTHN);return;}
var {MorSo(zVAR);return;}
while {MorSo(zWHL);return;}

/**********/
/* Zahlen */
/**********/
[0-9]+ {
Morph.MC=mcNumb;
Morph.Val.Numb=atol(yytext);
Morph.MLen=strlen(yytext);
return;
}
/***************/
/* Bezeichner, */
/***************/
/* muessen hinter Schluesselwoertern aufgefuehrt werden */
[A-Za-z]([A-Za-z0-9])* {
Morph.MC=mcIdent;
Morph.Val.pStr=yytext;
Morph.MLen=strlen(yytext);
return;
}

/* Sonderzeichen */
("?") {MorSo('?' );return;}
("!") {MorSo('!' );return;}
("+") {MorSo('+' );return;}
("-") {MorSo('-' );return;}
("*") {MorSo('*' );return;}
("/") {MorSo('/' );return;}
("=") {MorSo('=' );return;}
(">") {MorSo('>' );return;}
("<") {MorSo('<' );return;}
(":=") {MorSo(zErg);return;}
("<=") {MorSo(zle );return;}
(">=") {MorSo(zge );return;}
(";") {MorSo(';' );return;}
(".") {MorSo('.' );return;}
(",") {MorSo(',' );return;}
("(") {MorSo('(' );return;}
(")") {MorSo(')' );return;}
/* String */
(\".*\") {Morph.MC=mcStrng;
Morph.Val.pStr=yytext;
Morph.MLen=strlen(yytext);
return;}

%%
tMorph* Lex()
{
```

```
yylex();
return &Morph;
}
void MorSo(int Code)
{
Morph.MC=mcSymb;
Morph.Val.Symb=Code;
Morph.MLen=strlen(yytext);
return;
}
int initLex(char* fname)
{
char vName[128+1];
strcpy(vName,fname);
if (strstr(vName,".pl0")==NULL) strcat(vName,".pl0");
pIF=fopen(vName,"rt");
if (pIF!=NULL) {yyin=pIF; return OK;}
return FAIL;
}
```


9 YACC

9.1 GRUNDLAGEN

Vorlesung
17.01.2018

- Onlinedocumentation:
  - http://dinosaur.compilertools.net/yacc/index.html
  - http://epaperpress.com/lexandyacc/
- Yet another compiler compiler
- Klassisches Unix Werkzeug, bison ist die GNU-Version von yacc
- Generiert LALR bottom up Parser (look ahead, left recursive Parser)

9.1.1 AUFBAU EINER YACC-DATEI

- Aufbau ist ähnlich lex
- Programmbuild:

```
%{

 ...c definitions ...
%}
...definitions (tokens ...)

%%

... rules ...

%%

... c-functions ...
```

- yacc -d xyz.y
- lex xyz.lex
- gcc lex.yy.c y.tab.c
- yacc -d erzeugt Headerfile für lex mit Tokenbeschreibungen
- Arbeitsschritte:
  - yacc-Datei aus Grammatik bauen
  - yacc -d xyz.y
  - Lex-Datei bauen, verwendet y.tab.h
  - lex xyz.lex
  - compilieren

9.1.2 ZUSAMMENSPIEL LEX – YACC

- Yacc ruft die Funktion `int yylex()` immer auf, wenn ein Token benötigt wird.
- Der Tokencode, bei Sonderzeichen oft das Zeichen selbst, wird als Returnwert zurückgegeben, viele Token sind damit hinreichend beschrieben.
- Ein darüber hinausgehender Tokenwert kann über yylval übergeben werden. Der Wert wird in der Aktion einer Regel nach yylval geschrieben.
- Der Defaulttyp von yylval ist int, sollen Werte verschiedener Typen übergeben werden, so erfolgt dies über einen union Typ.

9.2 YACC-TEILE

9.2.1 DEFINITIONSTEIL

- C-Code, der in das zu generierende c-File übernommen wird, eingeschlossen in %{ .. %}
  - Includes, defines, prototypes, Variablendefinitionen
- Yacc Dekarationen
  - Tokendefinitionen (%token numeral) daraus wird dann das Includefile y.tab.h generiert bei Aufruf von yacc -d xyz.y
  - Präzedenzdeklarationen
  - Spezielle union Definition, aus der YYSTYPE generiert wird.

9.2.2 REGELTEIL

- Regeln in BNF ähnlicher Notation beginnen linksbündig mit einem Metasymbol
- Alternativen beginnen üblicher Weise auf neuer Zeile mit '|', nicht unbedingt linksbündig
- Nach der Regel kann eine Aktion als Block angegeben werden, dieser kann über mehrere Zeilen gehen. Die Zeilen dürfen nicht linksbündig beginnen!
- Werte der einzelnen Zeichen der rechten Seite der Regel stehen in den Aktionen unter $1, $2 … entsprechend Ihrer Position zur Verfügung. Ein Returnwert kann durch $$ gebildet werden.
- Jede Regel wird mit ';' abgeschlossen

9.2.3 BEISPIEL 1

```
1  %{
2    #include <stdio.h>
3    int yylex(void);
4    void yyerror(char *);
5  %}
6  %token INTEGER
7  %%
8  program:  program expr '\n'   { printf("%d\n", $2); }
9            |
10           ;
11 expr:     INTEGER
12           | expr '+' expr      { $$ = $1 + $3; }
13           | expr '-' expr      { $$ = $1 - $3; }
14           ;
15 %%
16 void yyerror(char *s) {
```

```
17   fprintf(stderr, "%s\n", s);
18 }
19 int main(void) {
20   yyparse();
21   return 0;
22 }
```

Mit `yacc -d xyz.y` generiert sich `y.tab.h` (gekürzt):

```
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE

   enum yytokentype {
     INTEGER = 258
   };
#endif
/* Tokens.  */
#define INTEGER 258

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef int YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn
*/
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Tokencode

Typ des Tokenwertes, hier int
In anderen Fällen ein union

Mit der dazugehörigen `lex`-Datei:

```
    /* calculator #1 */
%{
    #include "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      { return *yytext; }

[ \t]       ;       /* skip whitespace */

.           yyerror("Unknown character");

%%

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

Tokenwert

Tokencode

'+', '-' oder '\n' wird als
Token zurückgegeben

9.2.4 BEISPIEL 2: EXPR, TERM, FACTOR

```
1 %{
2   #include <stdio.h> /* Printf */
3   #include <stdlib.h>/* exit */
4   #define YYSTYPE int
5   int yyparse(void);
6   int yylex(void);
```

```
  void yyerror(char *mes);
%}
%token number
%token QUIT 254
%%
 // Kommentar eins eingerückt
program : program expr '\n'   {printf("= %d\n", $2);}
          |
          ;
expr :    expr '+' term       {puts("expr 1");$$ = $1 + $3;}
          | expr '-' term     {puts("expr 2");$$ = $1 - $3;}
          | term              {puts("expr 3");$$ = $1;}
          | QUIT              {exit(0);}
          ;
term :    term '*' fact       {puts("term 1");$$ = $1 * $3;}
          | term '/' fact     {puts("term 2");$$ = $1 / $3;}
          | fact              {puts("term 3");$$ = $1;}
          ;
fact :    number              {puts("fact 1");$$ = $1;}
          | '-' number        {puts("fact 1");$$ = -$2;}
          | '(' expr ')'      {puts("fact 2");$$ = $2;}
          ;
%%
int main() {
  printf("Enter expression with + - * / ( ) \n");
  yyparse(); return 0;
}
void yyerror(char *mes) {fprintf(stderr,"%s\n", mes);}
```

Mit dem Lexer:

```
%{
  #include "t2.h" /* eigentlich y.tab.h !! */
  #include <stdlib.h> /* yacc -d -o t2.c erzeugt*/
  void yyerror(char *); /* auch t2.h */
%}

%%
[09]+       { yylval = atoi(yytext);
              return number;
            }
q(uit)?     return QUIT;
[-+()=/*\n] { return *yytext; }
[ \t]       ; /* skip whitespace */
.           yyerror("Unknown character");
%%
int yywrap(void) {
  return 1;
}
```

9.3 PRÄZENDENZREGELN

- Präzendenz: Vorrang, Priorität
- Präzedenzregeln von yacc regeln die Priotrität und die Assoziativität von Operatoren
- Die Präzedenzregeln werden im Definitionsteil der yacc Datei angegeben und gelten für jedes Vorkommen des betreffenden Tokens
- Präzedenz steigt zeilenweise von oben nach unten (→ Beispiel)

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
```

* und / hat höhere Präzedenz als + und -

9.3.1 AUSNAHME

- Für das unäre '-' klappt das nicht! Das negative Vorzeichen hat eine höhere Priorität, als die multiplikativen Operatoren.
- Vergabe einer neuen Präzedenz für das Token '-' in der betreffenden Regel.

```
expr    :       expr  '+'  expr
        |       expr  '-'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |  '-'  expr %prec  '*'
```

9.3.2 BEISPIEL

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

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{

void yyerror(char *);
int yylex(void);
int sym[26];
%}
%%
. . . → next Page

%%
void yyerror(char *s) {fprintf(stderr, "%s\n", s);}
int main(void)
{
 yyparse();  return 0;
}
```

Präzedenzregeln

Array für 26 Variable

```
Program:   program statement '\n'
         |
         ;
Statement: expr { printf("%d\n", $1); }
         | VARIABLE '=' expr { sym[$1] = $3; }
         ;
Expr: INTEGER
    | VARIABLE              { $$ = sym[$1]; }
    | expr '+' expr         { $$ = $1 + $3; }
    | expr '-' expr         { $$ = $1 - $3; }
    | expr '*' expr         { $$ = $1 * $3; }
    | expr '/' expr         { $$ = $1 / $3; }
    | '(' expr ')'          { $$ = $2; }
    | '-'  expr  %prec '*'  { $$ =-$2; }
    ;
```

Unäres '-' hat höheren Vorrang als '*'

```
%{
//    #include "t2.h"       /* eigentlich y.tab.h !!  */
    #include "y.tab.h"      /* eigentlich y.tab.h !  */
    #include <stdlib.h>   /* yacc -d -o t2.c erzeugt*/
    void yyerror(char *); /* auch t2.h              */
%}

%%
[0-9]+      { yylval = atoi(yytext);
              return INTEGER;
            }
quit      return QUIT;
[a-z]+      { yylval = *yytext-'a';
              return VARIABLE;
            }
[-+()=/*\n] { return *yytext; }
[ \t]   ;   /* skip whitespace */
.           yyerror("Unknown character");

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

Berechnen des Variablenindex aus dem Buchstaben

Damit ergibt $12 + 3 * 4 = 24$, aber $12 + 3 * -4 = 0$, da $12 + (3 * -4) = 0$.

9.4 PL/0 MIT YACC UND LEX

- Umformung der Grammatik in einfache BNF
- Erstellen der yacc Datei
- Erstellen des Headerfiles
- Erstellen der lex-Datei
- Alles zusammenbauen → Parser fertig
- Semantikroutinen anpassen (Parameterübergaben)
- Alles zusammenbauen → Compiler fertig

9.4.1 PL0.Y

9.4.1.1 VEREINBARUNGSTEIL

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
#include "code.h"
#include "semrtype.h"
#include "semr.h"
%}
%union{
  long num;
  char*strg;
}
```

```
%token T_BEGIN          276
%token T_CALL           257
%token T_CONST          258
%token T_DO             259
%token T_ELSE           260
%token T_END            261
%token T_IF             262
%token T_ODD            263
%token T_PROCEDURE      264
%token T_THEN           265
%token T_VAR            266
%token T_WHILE          267
%token <strg>T_Ident    268
%token <num>T_Num       269
%token T_ERG            270
%token T_LEQ            271
%token T_GEQ            272
%token <strg>T_String   273
```

9.4.1.2 FUNKTIONENTEIL

```
%%
main(int argc, char**argv)
{
 if (argc>1)
  if (initLex(argv[1]))
  {
    initMain();
    openOFile(argv[1]);
    yyparse();
    closeOFile();
  }
  else printf("Dateifehler\n");
  return 0;
}

yyerror(char* ps) {  printf("%s\n",ps); }
```

Öffnen der Quelldatei und Initialisierung des Lexers

Anlegen aller erforderlichen Datenstrukturen

9.4.1.3 REGELTEIL

```
Program  : block '.'                        {AdestroyPl0();};

Block    : blockDecl statement;

BlockDecl: constDecl varDecl procDeclList  {AenterProc();};

ConstDecl: T_CONST constList ';'

         | ;

ConstList: constList ',' T_Ident '=' T_Num {AcreateConst($5,$3);}

         | T_Ident '=' T_Num              {AcreateConst($3,$1);};

varDecl:   T_VAR varList ';'

         | ;

varList:   varList ',' T_Ident             {AcreateVar($3);}

         | T_Ident                         {AcreateVar($1);}
```

```
ProcDeclList: procDeclList procDecl
            | ;

ProcDecl: T_PROCEDURE T_Ident ';'         {AcreateProc($2);}
          Block ';'                       {AdestroyProc();};

StatementList: statementList ';' statement
            |  statement
            |  ;

Statement:    T_BEGIN statementList T_END
            | T_Ident T_ERG {AassLeft($1);}
              expression                      {Aass();}
            | T_IF condition T_THEN           {Aif1();}
              Statement                       {Aif2();}
            | T_WHILE                         {Awhile1();}
              condition T_DO                  {Awhile2();}
              Statement                       {Awhile();}
            | T_CALL T_Ident                  {Acall($2);}
            | '?' T_Ident                     {Ain($2);}
            | '!' expression                  {AstOutExpr();}
            | '!' T_String                    {AstOutStrng($2);};


    expression: term1 '+' exprList      {Aoperator(0);}
              | term1 '-' exprList      {Aoperator(1);}
              | term1;
    term1:      '-' term                {AvzMinus();}
              | '+' term
              | term;
    ExprList:   exprList '+' term       {Aoperator(0);}
              | exprList '-' term       {Aoperator(1);}
              | term;

    Term:       term '*' factor         {Aoperator(2);}
              | term '/' factor         {Aoperator(3);}
              | factor;

    Factor:     T_Num                   {AfacNum($1);}
              | T_Ident                 {AfacId ($1);}
              | '(' expression ')';
    CmpOP:      '='                     {AcondPush(cmpEQ);}
              | '#'                     {AcondPush(cmpNE);}
              | '<'                     {AcondPush(cmpLT);}
              | '>'                     {AcondPush(cmpGT);}
              | T_LEQ                   {AcondPush(cmpLE);}
              | T_GEQ                   {AcondPush(cmpGE);};


   Condition: T_ODD expression              {AcondOdd();}
            | expression cmpOP expression   {Acond();};
```

9.4.2 LEXER

```
%{
/*
lexikalische Analyse mit lex fuer PL/0
*/
#include "y.tab.h"
#include "list.h"
#include "debug.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

FILE * pIF;
int Line;
%}
%%
```


9.4.2.1 FUNKTIONENTEIL

```
%%

int Lex()
{
  return yylex();;
}

int yywrap(){ return 1;}

int initLex(char* fname)
{
  char vName[128+1];

  strcpy(vName,fname);
  if (!strstr(vName,".pl0")) strcat(vName,".pl0");

  pIF=fopen(vName,"rt");
  if (pIF) {yyin=pIF; return OK;}
  return FAIL;
}
```

9.4.3 REGELTEIL

```
%%
 /* rules division */

 /*************************************************/
 /* return beendet die lexikalische Analyse nach dem*/
 /* Erkennen eines Morphems.                      */
 /*************************************************/

 /*************************/
 /* Leer- und Trennzeichen */
 /*************************/
[ \t]+

 /*****************/
 /* Zeilenwechsel */
 /*****************/
[\n]      {Line++;}


        /************************************/
        /* Schluesselwoerter (Wortsymbole)   */
        /* werden wie Sonderzeichen behandlt */
        /************************************/
       "begin"   {return T_BEGIN;}
       call      {return T_CALL;}
       const     {return T_CONST;}
       do        {return T_DO;}
       else      {return T_ELSE;}
       end       {return T_END;}
       if        {return T_IF;}
       odd       {return T_ODD;}
       procedure {return T_PROCEDURE;}
       then      {return T_THEN;}
       "var"     {return T_VAR;}
       while     {return T_WHILE;}
```


```
 /**********/
 /* Zahlen */
 /**********/
[0-9]+       {
                yylval.num=atol(yytext);
                return T_Num;
             }
 /***************/
 /* Bezeichner, */
 /***************/

 /* muessen hinter Schluesselwoertern aufgefuehrt werden */
[A-Za-z]([A-Za-z0-9])* {
                 ylval.idnt=malloc(strlen(yytext)+1);
                 strcpy(yylval.idnt,yytext);/*yytext;*/
                 printf("<%s>\n",yylval.idnt);
                 return T_Ident;
                 }


      /*****************/
      /* Sonderzeichen */
      /*****************/

     [?]         {return '?';}
     [!]         {return '!';}
     "+"         {return '+';}
     '-'         {return '-';}
     "*"         {return '*';}
     "/"         {return '/';}
     ">"         {return '>';}
     "<"         {return '<';}
     "="         {return '=';}
     ":="        {return T_ERG;}
     "<="        {return T_LEQ;}
     ">="        {return T_GEQ;}
     ";"         {return ';';}
     "."         {return '.';}
     ","         {return ',';}
     "("         {return '(';}
     ")"         {return ')';}
     (\".*\")    { yylval.strg=yytext;return T_String; }
```

9.4.4 DIE FUNKTIONEN

```
 Factor: T_Num    {AfacNum($1);};
       | T_Ident  {AfacId ($1);};
       | '(' expression ')';
 *********************************************/

 int AfacId (char* pName)
 {
   tBez *pB;
   if ((pB=searchBezGlobal(pName))==NULL) Error(EBezNtFnd);
   if (((tVar*)pB->pObj)->Kz==KzVar)
   {
     if (pB->IdxProc==0)
        code(puValVrMain,((tVar*)pB->pObj)->Dspl); else
     if (pB->IdxProc==pCurrPr->IdxProc)
        code(puValVrLocl,((tVar*)pB->pObj)->Dspl); else
        code(puValVrGlob,((tVar*)pB->pObj)->Dspl,pB->IdxProc);
        return OK;
   }else
   if (((tConst*)pB->pObj)->Kz==KzConst)
   {
     code(puConst,((tConst*)pB->pObj)->Idx);
     return OK;
   }
   else Error(ENoNumVal);
   return 0;
 }
```

```
int AfacNum(long Numb)
{
  tConst *pCnst;
  if ((pCnst=searchConst(Numb))!=NULL) ;
  else
  {
    pCnst=createConst(Numb);
    if (pCnst==NULL) return FAIL;
  }
  code(puConst,pCnst->Idx);
  return OK;
}
```


10 TABELLENGESTEUERTE VERFAHREN

10.1 GRUNDLAGEN

Vorlesung
17.01.2018

- Es gibt tabellengesteuerte Verfahren nach den beiden Strategien, top down und bottom up.
- Die Syntaxregeln der Sprache werden in Tabellen implementiert, die den Stackautomaten dann steuern.
- Für einen minimalen Parser wird eine Automatentabelle, ein Kellerspeicher, ein Analysealgorithmus und ein einfacher Lexer benötigt.
- Tabellengesteuerte bottom up Parser sind komplizierter.

10.2 SCHRITTWEISE AUFBAU EINES TOP-DOWN PARSERS FÜR AUSDRÜCKE

- Umformung der Grammatik in einfache BNF
- Umformung der Regeln, so dass keine Linksrekursionen enthalten sind
- Bestimmung der terminalen Anfänge der Regeln
- Ursprüngliche Grammatik für Ausdrücke:

```
<expr> ::= <expr> '+' <term> | <expr> '-' <term> | <term>
<term> ::= <term> '*' <fact> | <term> '/' <fact> | <fact>
<fact> ::= '-' num | num | '(' <expr> ')'
```

10.2.1 UMGEFORMTE REGELN

```
<expr>  :: <term> <rexpr>

<rexpr> :: '+' <term> <rexpr>
         | '-' <term> <rexpr>
         | nil

<term>  :: <fact> <rterm>

<rterm> :: '*' <fact> <rterm>
         | '/' <fact> <rterm>
         | nil

<fact>  :: '-' <num>
         | <num>
         | '(' <expr> ')'
```

10.2.2 BESTIMMUNG DER TERMINALEN ANFÄNGE

```
<expr>  -> <term> -> <fact> -> ['-' | '(' | num]
<rexpr> -> ['+' | '-' | epsilon]
<term>  -> <fact> -> ['-' | '(' | num]
<rterm> -> ['*' | '/' | epsilon]
<fact>  -> ['-' | '(' | num]
```

10.2.3 AUFBAU DER TABELLE

- Die Spalten der Tabelle werden durch die Eingabezeichen gebildet.
- Zeilen werden durch die linken Seiten der Regeln, durch die nichtterminalen Symbole gebildet.
- In der Zelle steht
  - die rechte Seite der Regel zu einem nichtterminalen Symbol und dem jeweiligen terminalen Anfangssymbol.
  - Error: bei korrekter Syntax der Eingabe gelangt man hierher nicht
  - Pop: Es bleibt nichts zu tun

10.2.4 DER ALGORITHMUS

- Stackinitialisierung mit push <expr>
- Auskellern des obersten Stackelements
  - Nichtterminal :
    - Ermitteln der Zelle aus dem Nichtterminal (Zeile) und dem Eingabesymbol (Spalte)
    - Error: zu dieser Zelle gelangt man im Normalfall nicht.
    - Pop: an dem Stack ändert sich nun nichts mehr.
    - Regel: Einkellern aller Symbole der rechten Seite, so dass das am weitesten links stehende Symbol oben auf dem Stack liegt.
  - Terminal:
    - Übereinstimmung mit Eingabezeichen? (accept)-> ok/Error
- Nächster Analyseschritt

10.2.5 BEISPIEL

Man schreibt die rechte Seite der Regeln hin (siehe Umgeformte Regeln), in denen eine Regel einen terminalen Anfang besitzt.

		First								
		nix(ε)	+	-	*	/	(	)	num	leer
Regel	expr									
	rexpr									
	term									
	rterm									
	fac									

LÖSUNG AUTOMATENTABELLE

		First								
		nix	+	-	*	/	(	)	num	leer
Regel	expr	error	error	rexpr term	error	error	rexpr term	error	rexpr term	error
	rexpr	error	rexpr term '+'	rexpr term '-'	error	error	error	pop	error	pop
	term	error	error	rterm fac	error	error	rterm fac	error	rterm fac	error
	rterm	error	pop	pop	rterm fac '*'	rterm fac '/'	error	pop	error	pop
	fac	error	error	num '-'	error	error	')' expr '('	error	num	error

Hinweis: nix entspricht keiner Eingabe beim Start, sonst ist leer.

ALGORITHMUSABARBEITUNG

Stack				Operation	Current Token
<expr>					Token=num(2)
<term>	<rexpr>				
<fact>	<rterm>	<rexpr>			
num	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(2)	Token='+'
<rexpr>				pop	
+'	<term>	<rexpr>			
<term>	<rexpr>			accepted '+'	Token=num(3)
<fact>	<rterm>	<rexpr>			
<num>	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(3)	Token='*'
*'	<fact>	<rterm>	<rexpr>		
<fact>	<rterm>	<rexpr>		Accepted '*'	Token=num(4)
<num>	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(3)	empty
<rexpr>				pop	
empty				accepted all ok	

→ so lange entsprechend der Automatentabelle ersetzen, bis empty und alle akzeptiert.

10.3 BOTTOM-UP ANALYSE

- Ist geeignet, linksrekursive Grammatiken zu verarbeiten
- LR-Parser basieren auf bottom-up Verfahren

- Basiert auf
- zwei Tabellen
  - Aktionstabelle
  - Sprungtabelle
- zwei Kellerspeichern
  - Symbolstack
  - Zustandsstack
- zwei Operationen
  - shift
  - reduce

10.3.1 SHIFT/REDUCE

Shift:

Die Shiftoperation wird immer dann ausgeführt, wenn in dem adressierten Feld der Automatentabelle ein neuer Zustand Qx aufgeführt ist.

Es wird ein Zeichen aus dem Eingabestrom in den Symbolstack übernommen und der in der Automatentabelle angegebene Zustand in den Zustandsstack gespeichert.

Reduce:

Die Reduceoperation wird ausgeführt, wenn in der Automatentabelle eine Regel gefunden wird. Dazu werden die Symbole auf der rechten Seite der Regel aus dem Symbolstack und ebenso viele Einträge aus dem Zustandsstack entfernt.

Dabei müssen die aus dem Stack entfernten Symbole mit denen der anzuwendenden Regel übereinstimmen. Danach wird das Symbol auf der linken Seite der Regel in den Symbolkeller geschrieben. Das jetzt oberste Symbol und der jetzt oberste Zustand bilden die Indizes für die Sprungtabelle. Der hier gefundene Zustand wird in den Zustandskeller geschrieben.

10.3.2 AUTOMATENTABELLE

	Aktionstabelle						***Sprungtabelle***		
	Num	+	*	(	)	$	E	T	F
Q0	Q5			Q4			Q1	Q2	Q3
Q1		Q6				ACCEPT			
Q2		E::=T	Q7		E::=T	E::=T			
Q3		T::=F	T::=F		T::=F	T::=F			
Q4	Q5			Q4			Q8	Q2	Q3
Q5		F::=num	F::=num		F::=num	F::=num			
Q6	Q5			Q4				Q9	Q3
Q7	Q5			Q4					Q10
Q8		Q6			Q11				
Q9		E::=E+T	Q7		E::=E+T	E::=E+T			
Q10		T::=T*F	T::=T*F		T::=T*F	T::=T*F			
Q11		F::=(E)	F::=(E)		F::=(E)	F::=(E)			

10.3.3 BEISPIEL

2+3*4									
$	Q0								

1. Schritt: Mit aktuellem Eingabesymbol 2 (Num) zum nächsten Zustand Q5 (shift)
2. Schritt: Mit aktuellem Eingabesymbol + zum nächsten Zustand/Reduce Anweisung: num mit

F ersetzen und nächsten Sprung mit Sprungtabelle einfügen.
3. Schritt: F mit T ersetzen, Q2 als Sprung.
4. Schritt: T mit E ersetzen, Q1 als Sprung.
usw. . . Es wird immer geshiftet oder reduced. Steht ein Qx drin, so wird geshifted, wobei das nächste Zeichen der Eingabe das aktuelle Token ist (mit dem man bei den Zuständen in die entsprechende Spalte guckt).
Beim Reduzieren in späteren Schritten: so viele Zustände raus nehmen, wie in der Regel auf der rechten Seite stehen. Wie immer kommt ein neuer Zustand hinzu.

2+3*4									
$	Q0								
num	Q0	Q5							
F	Q0	Q3							
T	Q0	Q2							
E	Q0	Q1							
E+	Q0	Q1	Q6						
E+num	Q0	Q1	Q6	Q5					
E+F	Q0	Q1	Q6	Q3					
E+T	Q0	Q1	Q6	Q9					
E+T*	Q0	Q1	Q6	Q9	Q7				
E+T*num	Q0	Q1	Q6	Q9	Q7	Q5			
E+T*F	Q0	Q1	Q6	Q9	Q7	Q10			
E+T	Q0	Q1	Q6	Q9					
E	Q0	Q1							
$	Accept								