Compiler/Interpreter

Vorlesung, Praktikum, Klausur

Praktikum: besteht aus 2 Teilen

- Ausdrucksinterpreter nach dem Verfahren des rekursiven Abstiegs
- PL/0 – Compiler nach einem graphengesteuerten Verfahren

Klausur: 100 min. mit Spickzettel A4

Compiler/Interpreter

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

Grundlagen

- Einführung in die Begriffswelt, Definition von formaler Sprache
- Klassifizierung der Sprachen nach Chomsky
- Definition des Begriffes Grammatik
- Klassifizierung von Grammatiken
- Die Strategien “top down” und “bottom up”

Literatur

- Wirth, Compilerbau, Teubnerverlag
- Kopp, Compilerbau, Hanserverlag
- Aho/Sethi/Ullmann, Compilerbau, Addison-Wesley
- Bachmann/Loeper, Theorie und Technik formaler Sprachen Teubner Verlagsges.

Compiler/Interpreter

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)

Compiler

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

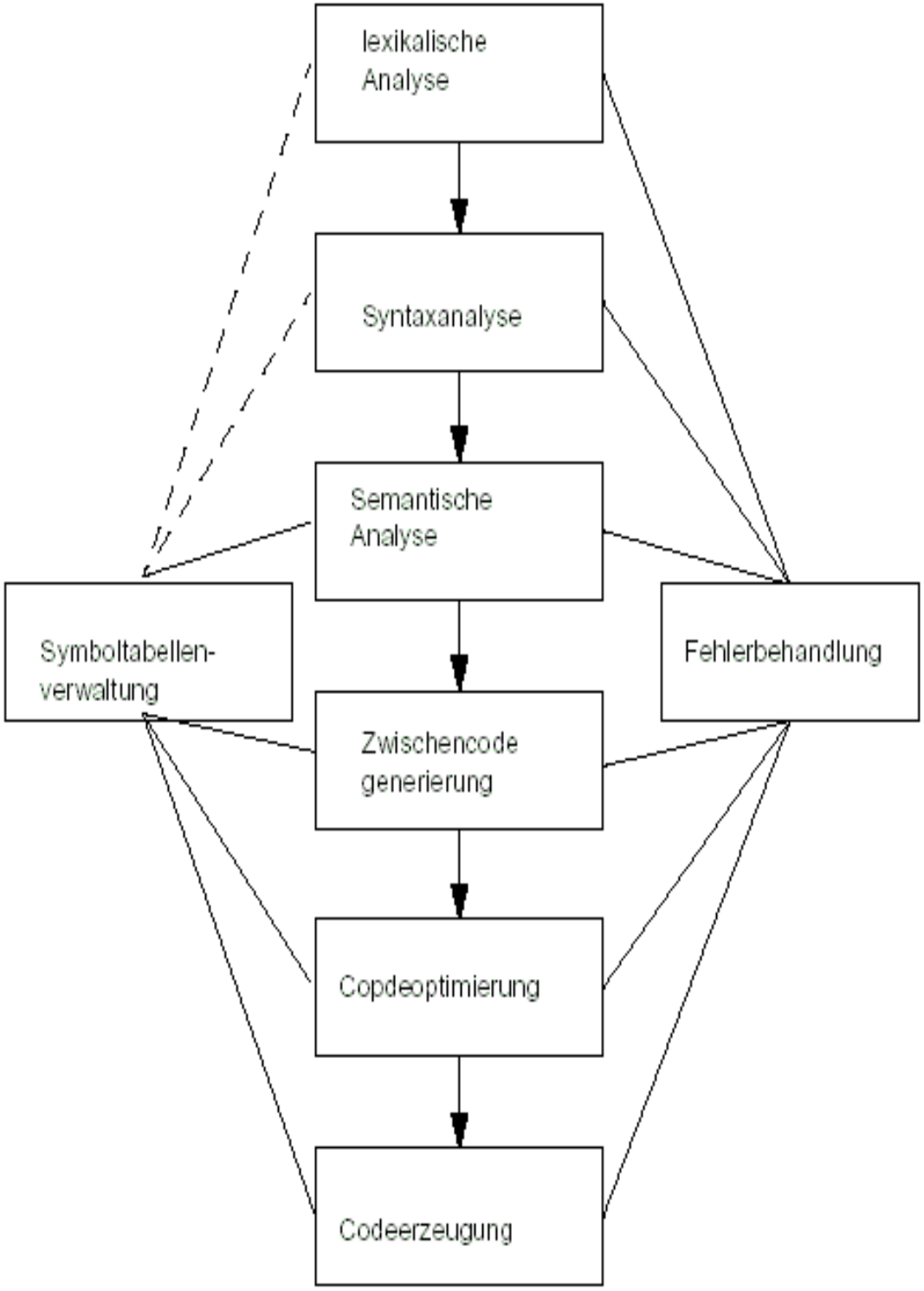
Höhere Programmiersprache -> Maschinencode
Höhere Programmiersprache -> Assemblersprache
Höhere Programmiersprache -> Zwischencode
Höhere Programmiersprache -> Höhere Sprache

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

Einsatz in intelligenten Editoren

Aufbau eines Compilers nach Aho, Sethi, Ullmann

Mehrpasscompiler



Lexikalische Analyse (Lexer)

- Der Lexer liest zeichenweise den Eingabetext.
- Er entfernt nicht signifikannte Zeichen.
- Er zerlegt den Eingabetext in die Token oder Morpheme, die kleinsten bedeutungtragenden Bestandteile.
- Er wird häufig auch scanner genannt.

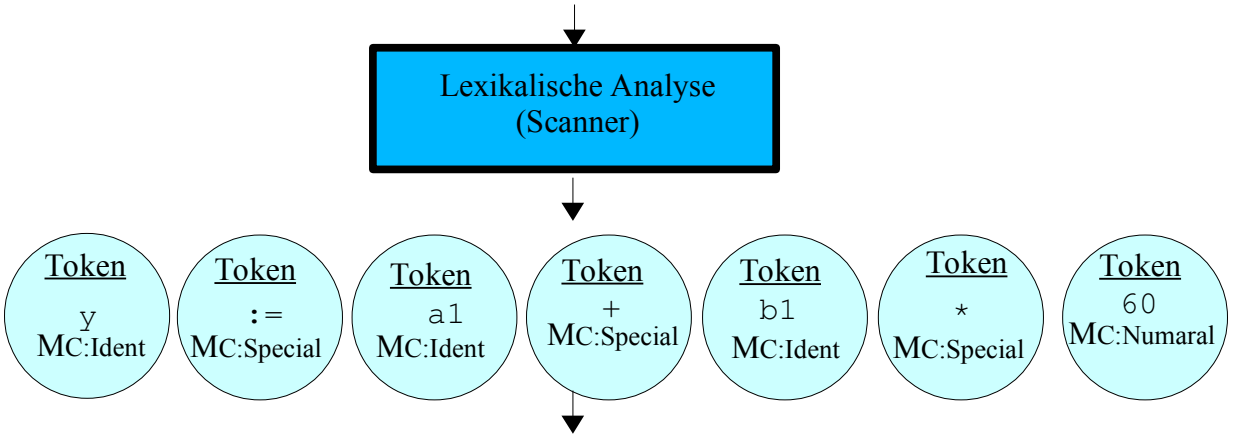
Syntaktische Analyse (Parser)

- Der Parser prüft, ob die Folge der Token oder Morpheme einen gültigen Satz der Sprache bilden.
- Die Sprache wird in einer Grammatik beschrieben, auf der der Parser basiert.
- In Mehrpasscompilern erzeugt der Parser einen Parsebaum, der dann das zu übersetzende Programm repräsentiert.
- In Einpasscompilern steuert der Parser den gesamten Übersetzungsprozess.

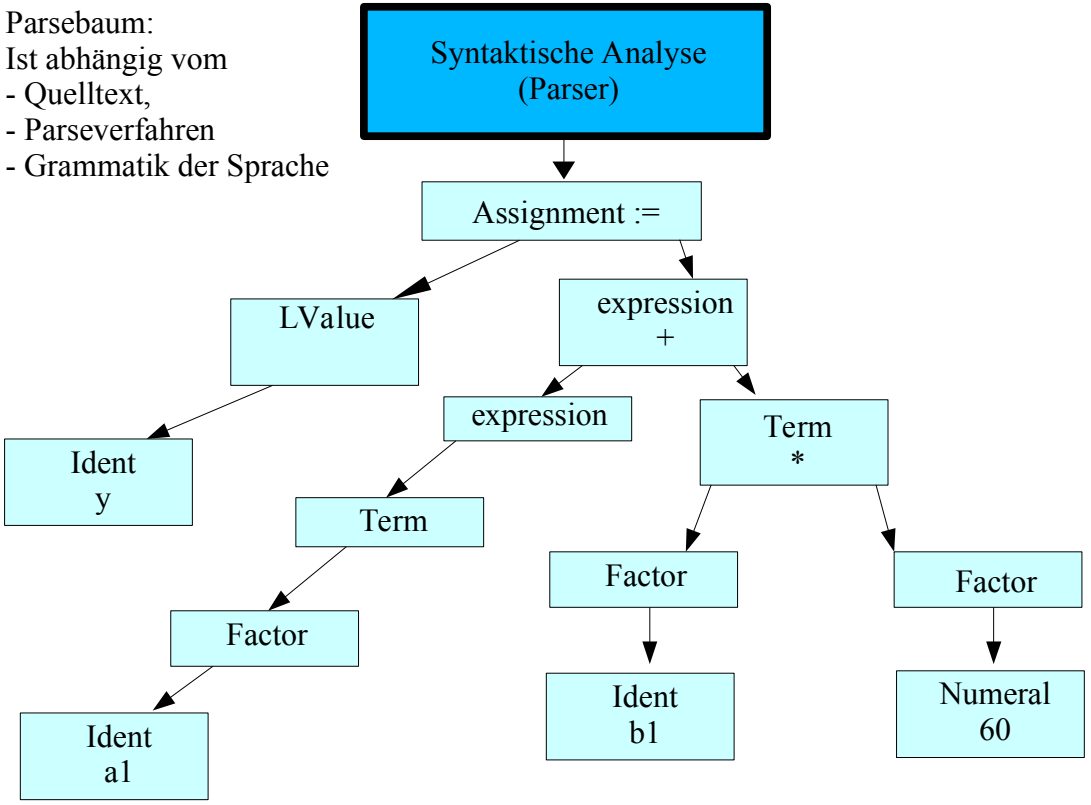
Semantische Analyse

- führt zusätzliche Prüfungen durch, wie die Überprüfung von Bezeichnern und Datentypverträglichkeit.

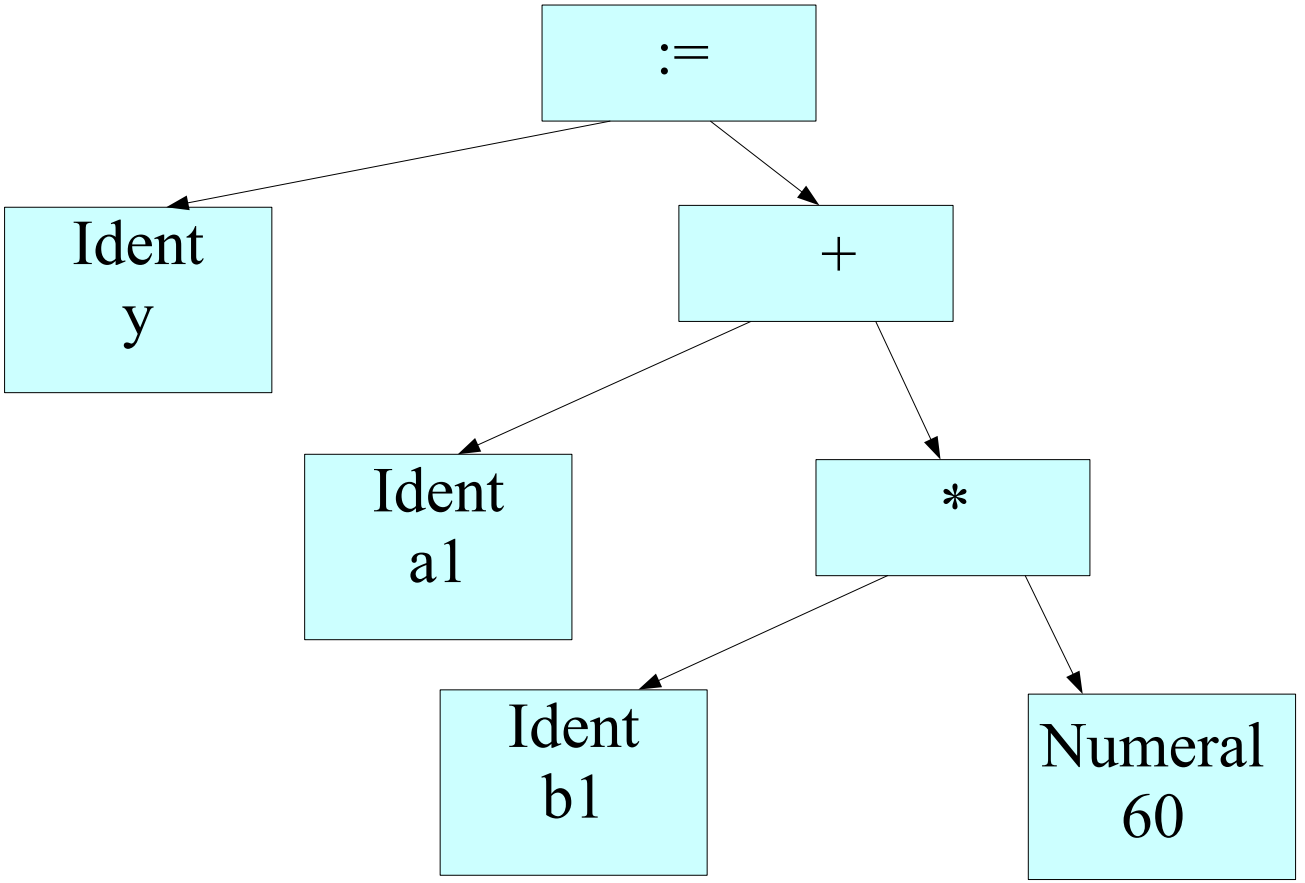
y:=a1+b1*60



Parsebaum:
Ist abhängig vom
- Quelltext,
- Parseverfahren
- Grammatik der Sprache



Abstrakter Syntaxbaum



Zwischencode

Dreiadressmaschine

```
temp1:=inttoreal(60)
temp2:=var3*temp1
temp3:=var2+temp2
var1 := temp3
```

Stackmaschine

```
pushAdr   y
pushVal   a1
pushVal   b1
pushConst 60.0
mul
add
store
```

Codeoptimierung
(Zwischencode)

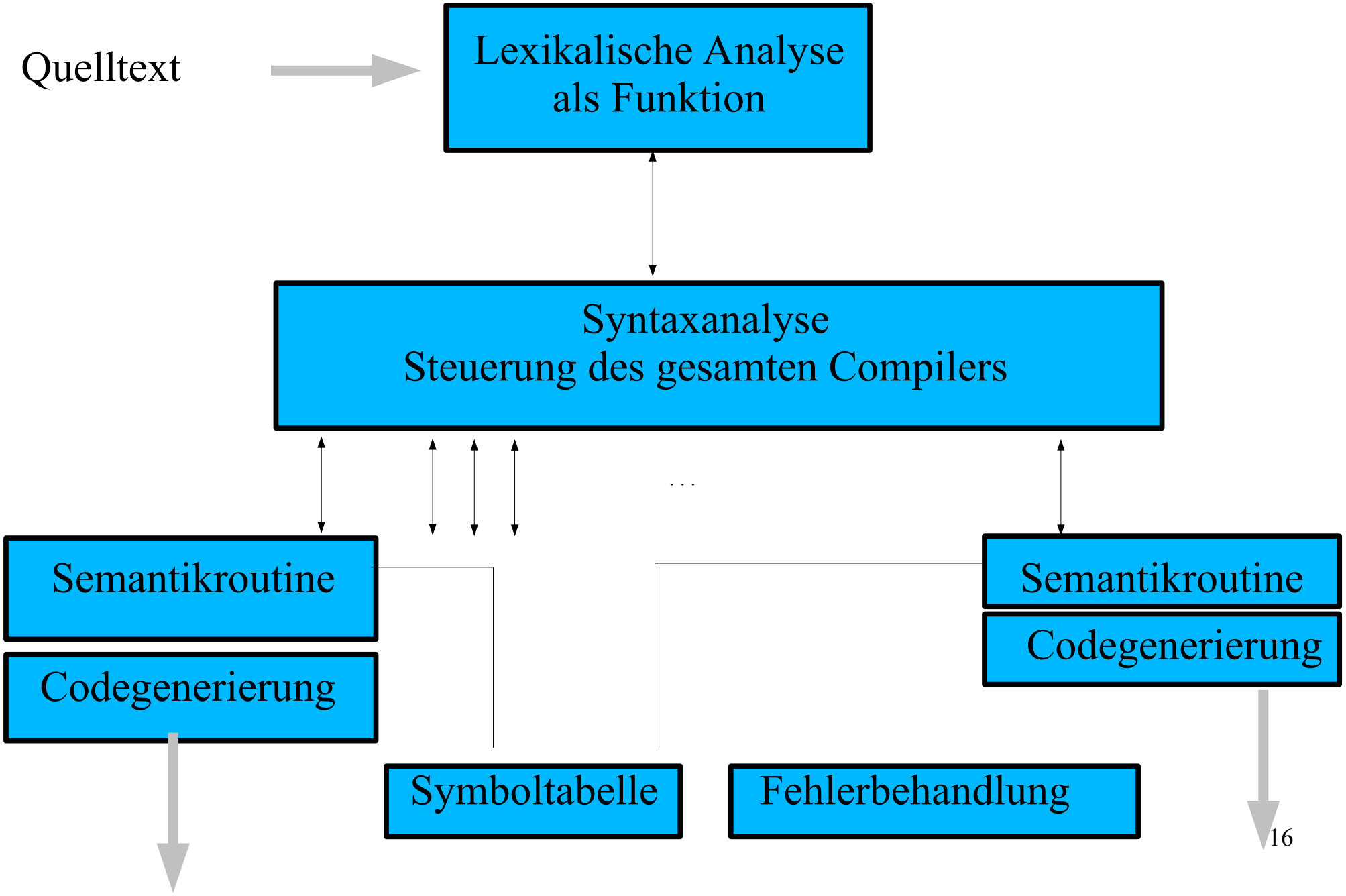
Dreiadressmaschine	Stackmaschine
temp1:=var3*inttoreal(60)	push 60.0
Var1 :=var2+temp1	mul b1
	add a1
	Store y

Maschinencodegenerierung
(Assemblercode)

Sytaxgesteuerter Einpasscompiler

- Bisher haben wir Mehrpasscompiler betrachtet. Ein Pass bildet dabei einen Transformationsschritt, in dem das gesamte zu übersetzende Programm gelesen, und nach der Verarbeitung in einer anderen (Zwischen-)Form wieder geschrieben wird.
- Ein Einpasscompiler liest die Eingabe und generiert Code in einem Durchlauf. Es erfolgt keine Speicherung eines Zwischenformates.
- Der Parser ist das zentrale Element, er ruft den Lexer und Routinen zu semantischen Prüfumngen und zur Codegenerierung als Unterprogramme (Funktionen/Prozeduren) auf.

Syntaxgesteuerter Einpasscompiler



Grundlagen und Begriffe

Sprache

- Sprache:

- Sprache dient der Kommunikation, Programmierprachen 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 von Syntaxregeln bilden lassen.**

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 hier als Zeichen aufgefasst (Wortsymbole, wie if, while, ... oder Bezeichner und Zahlen oder zusammengesetzte Symbole wie <= sind Zeichen in diesem Sinne).

- **Terminales Alphabet (T)**:

  ist die Menge der Zeichen, aus denen die Sätze der Spache gebildet werden, man bezeichnet diese auch als Atome, Morpheme, Token.

- **Nichtterminales Alphabet (N)**:

  ist 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 }
```

Zeichenkette

Eine Aneinanderreihung von Zeichen eines Alphabetes wird **Zeichenkette** genannt. Sie kann sowohl Elemente des terminalen, als auch 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

Freie Halbgruppe (V*)

- 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.

Syntaxregeln

- Aufbau von Syntaxregeln:

  Linke Seite SYMBOL Rechte Seite

- Linke und Rechte Seite sind Zeichenketten des Alphabetes.
- Die Linke Seite muss wenigstens ein nichtterminales Symbol enthalten, das bei Anwendung der Regel durch eine Zeichenkette ersetzt wird.
- Übliche Symbole sind:
  - -> (wird abgeleitet nach)
  - ::
  - ::=

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
```

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 eine Zeichenkette, die 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.

Chomsky Typ 0

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

Chomsky Typ 1

Die Produktionen sind von der Form **`s -> t`** mit **`s,t ∈ V*`** 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∈V*`** und **`#a>0`** annehmen.

Chomsky Typ 2

Die Produktionen haben die Form **A->s** mit **A∈N** und **s∈V***. Bei der Ableitung werden einzelne Nichtterminale durch eine Zeichenkette aus **V*** ersetzt. Grammatiken vom Typ 2 heißen kontextfreie Grammatiken, da sie nichtterminale Zeichen in einer Zeichenkette 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.

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.

Chomsky Typ 4

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

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)

Morpheme, Token, Atome

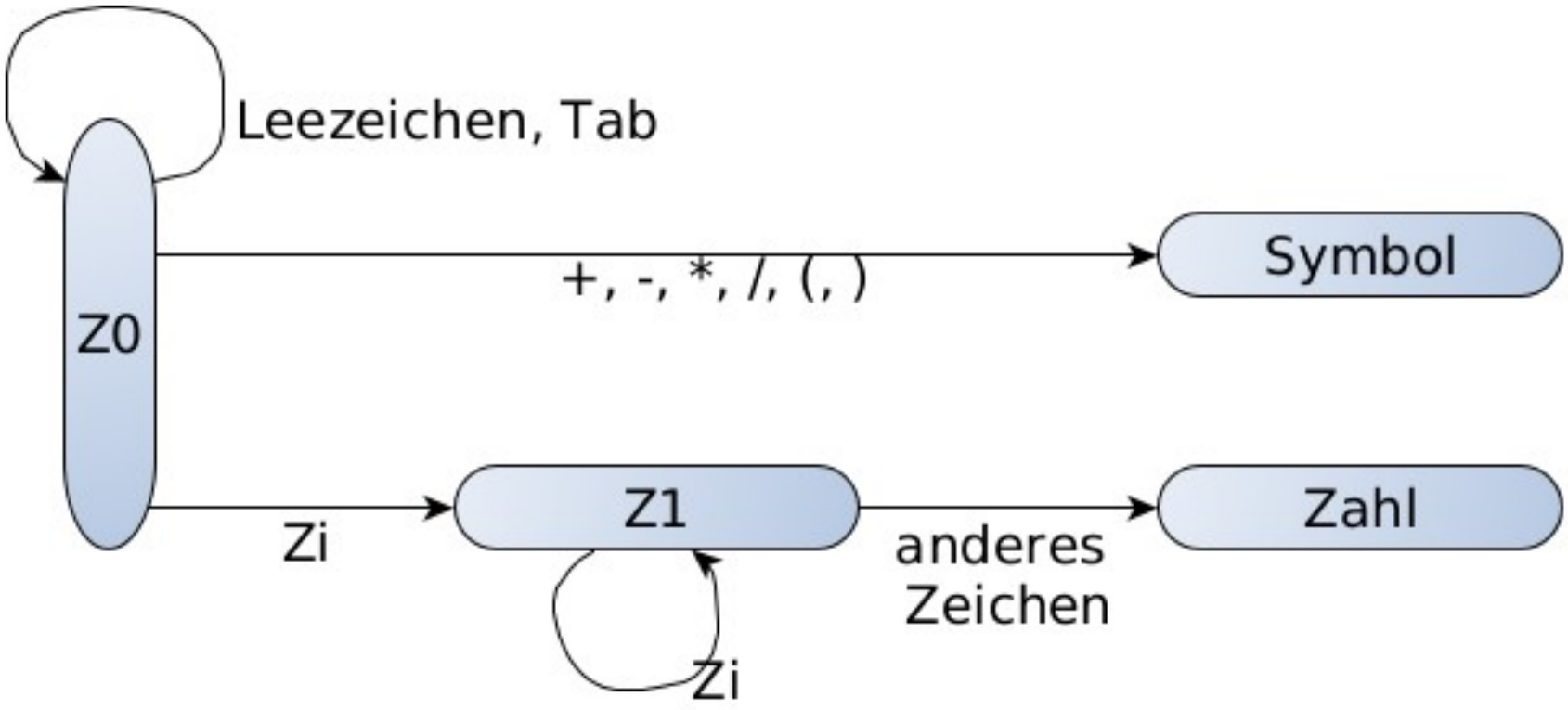
Sind die kleinsten Bedeutung tragenden Bestandteile einer Programmiersprache (Zahlen, Symbole, wie +, - .., aber auch Wortsymbole und Bezeichener)
Eigenschaften sind:

- Morphemtyp
- Morpheminhalt
- Morphemstatus (bereits verarbeitet?)
- Morphemlänge, Morphemposition ...

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 ::= +,-,*,/,(,)

Morpheme einfacher Ausdrücke



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

Ein einfacher Lexer (ohne – und /)

```
typedef struct morph
{
  int     mc;
  union
  {
  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=='.')
    // numeral
    {
      m.dval=strtod(px,&px);
      m.mc  =mdbl;
    } else
    // Symbol
    switch (*px)
    {
      case '+':
      case '*':
      case '(':
      case ')':m.cval=*px++; m.mc=mop;
               break;
      // end or wrong char
      default :m.mc=mempty;
    }
  }
}
```

Grammatik für einfache Ausdrücke (G1)

Grammatik G1 findet man häufig in der Literatur als Grammatik für einfache Ausdrücke, es fehlen – und / für die Subtraktion und Division.

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

Verfahren des Rekursiven Abstiegs (recursive descend parsing)

- Sehr einfaches Verfahren, um einen Parser zu bauen
- Jedes Nichtterminal von G wird durch eine Funktion repräsentiert, die den Namen der linken Seite trägt und die rechte Seite implementiert.
- 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)

Alternativen

- Bei alternativen Regeln muss an Hand des aktuellen lookahead entschieden werden können, ob eine Regel angewendet werden kann.
- Grammatiken müssen vom Typ LL(1) sein (wird später genauer erklärt)

Anwendung des Verfahrens auf G1

- Es werden 3 Funktionen benötigt
- expr, term, factor
- Die erste Regel lautet:

```
Expression  ::= Expression '+' Term | Term
double expr()
{
   double tmp=expr();
   ...
}
```

- Die Funktion expr ruft als erstes die Funktion expr, also sich selbst rekursiv auf

Verfahren funktioniert mit G1 nicht!

Anforderungen an die Grammatik

- 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

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.
- Bei LL(1) Grammatiken kann man anhand eines Eingabezeichens (aktuelles Token) die Alternative bestimmen.
- Um dies zu prüfen, werden die Mengen **first** und **follow** zu jedem Nichtterminal bestimmt.

- First: Die Menge der terminalen Anfänge aller Zeichenktten, 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.

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

Grammatik für einfache Ausdrücke (G2) keine Linksrekursion mehr

Aber irgendetwas geht hier schief!

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

oder

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

Die Funktion Expression (G2)

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

Die Funktion Faktor (G2)

```
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;
}
```

main

```
int main(int argc, char*argv[])
{

  char *pBuf=argv[1];
  printf("%s\n",pBuf);
  lex(pBuf);
  printf("%-10.4f\n",expr());
  free(pBuf);
  return 0;
}
```

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

Cut

Das müssen wir jetzt im Praktikum erst mal mit allen 4 Grundrechenarten probieren!

Erweitern Sie das Programm, so dass auch die Subtraktion und die Division, ev. Auch Modulo berechnet werden.