Compilerbau

Vom Programm zum Code

Jürgen Schönwälder

schoenw@informatik.uni-osnabrueck.de

Universität Osnabrück

Welche Aufgabe hat ein Compiler?

```
#include <stdio.h>

int
main(int argc, char **argv)
{
    int c;
    int n = 0, l = 0;

    while ((c = getchar()) != EOF) {
        n++;
        if (c == '\n') l++;
    }

    printf("%7d %7d\n", l, n);
    return 0;
}
```

Quelltext

Welche Aufgabe hat ein Compiler?

```
#include <stdio.h>

int
main(int argc, char **argv)
{
    int c;
    int n = 0, l = 0;

    while ((c = getchar()) != EOF) {
        n++;
        if (c == '\n') l++;
    }

    printf("%7d %7d\n", l, n);
    return 0;
}
```

Quelltext

$\Rightarrow$

```
55 89 e5 83 ec 18 c7 45
f8 00 00 00 00 c7 45 f4
00 00 00 00 e8 af fe ff
ff 89 c0 89 45 fc 83 7d
fc ff 75 02 eb 0e ff 45
f8 83 7d fc 0a 75 03 ff
45 f4 eb e0 83 c4 fc 8b
45 f8 50 8b 45 f4 50 68
e4 84 04 08 e8 bf fe ff
ff 83 c4 10 31 c0 eb 00
c9 c3
```

Maschinencode

Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole

Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole
- Syntaktische Analyse:
  Erzeugung eines Strukturbaums für das Programm

Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole
- Syntaktische Analyse:
  Erzeugung eines Strukturbaums für das Programm
- Semantische Analyse:
  Untersuchung der statischen semantischen Merkmale

Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole
- Syntaktische Analyse:
  Erzeugung eines Strukturbaums für das Programm
- Semantische Analyse:
  Untersuchung der statischen semantischen Merkmale
- Zwischencode-Erzeugung:
  Erzeugung eines einfachen Zwischencodes

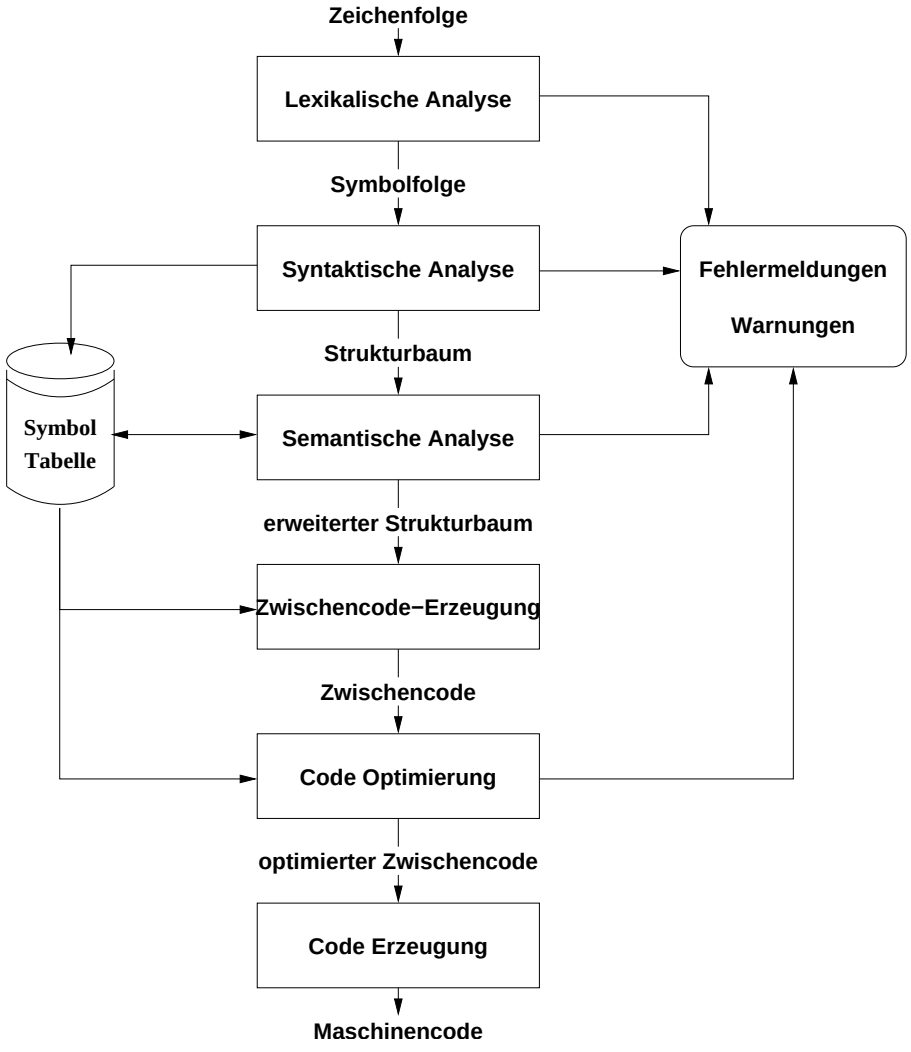
Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole
- Syntaktische Analyse:
  Erzeugung eines Strukturbaums für das Programm
- Semantische Analyse:
  Untersuchung der statischen semantischen Merkmale
- Zwischencode-Erzeugung:
  Erzeugung eines einfachen Zwischencodes
- Zwischencode-Optimierung:
  Entfernung von Redundanzen, Speicheroptimierungen

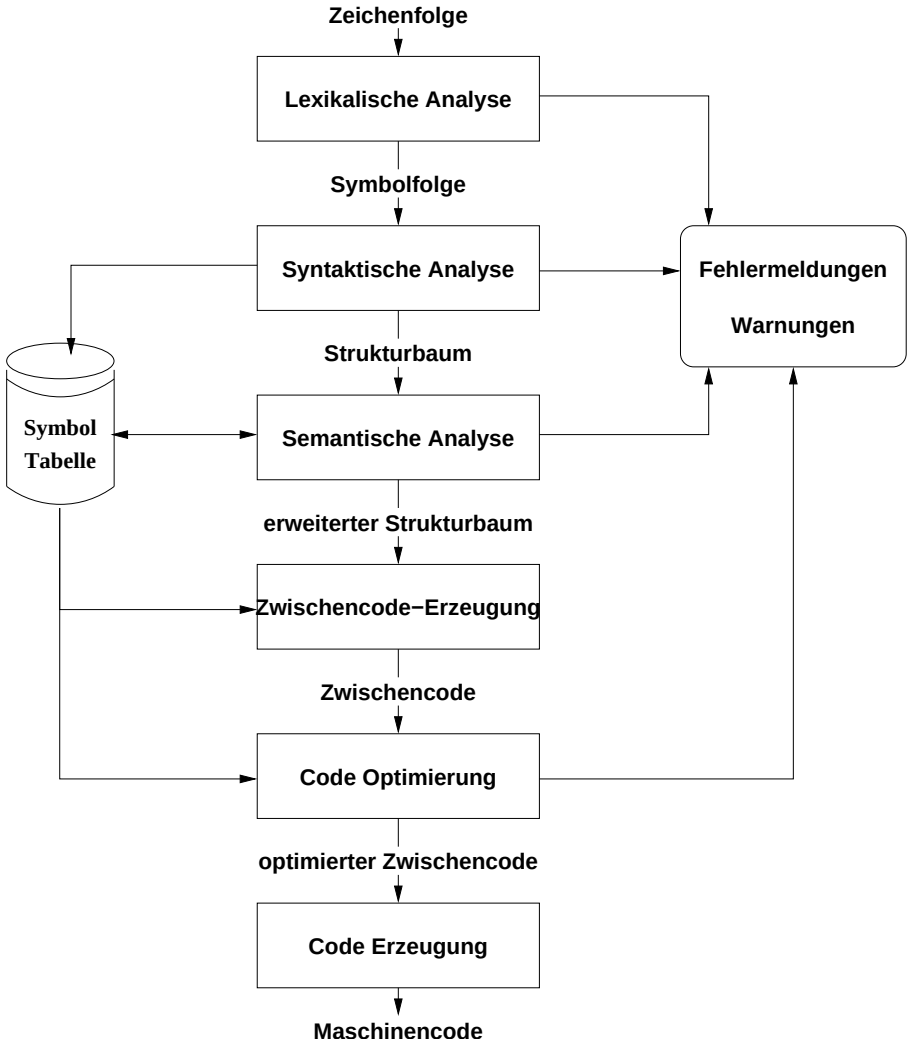
Phasen bei der Übersetzung

- Lexikalische Analyse:
  Zerlegung eines Eingabestroms in Symbole
- Syntaktische Analyse:
  Erzeugung eines Strukturbaums für das Programm
- Semantische Analyse:
  Untersuchung der statischen semantischen Merkmale
- Zwischencode-Erzeugung:
  Erzeugung eines einfachen Zwischencodes
- Zwischencode-Optimierung:
  Entfernung von Redundanzen, Speicheroptimierungen
- Code-Generierung:
  Erzeugung von ausführbarem Maschinencode

Phasen bei der Übersetzung

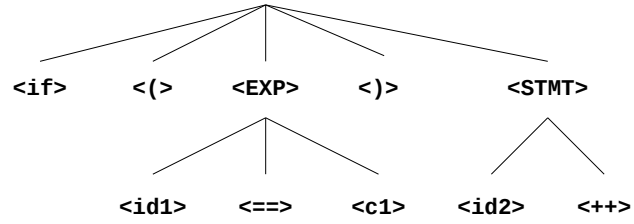


Phasen bei der Übersetzung




```
if (c == '\n') l++
```

```
<if> <(> <id1> <==> <c1> <)> <id2> <++>
```




```
    ld  r1, <id1>
    cmp r1, #<c1>
    jeq l1;
    jmp l2;
l1: ld  r1, <id2>
    inc r1
l2: nop
```

```
    cmpl $10,-4(%ebp)
    jne  L6
    incl -12(%ebp)
L6:
```

Lexikalische Analyse I

Problemstellung:

- Wie kann man effizient Symbole erkennen?
- Was passiert mit Kommentaren im Quelltext?
- Wie unterscheidet man zwischen Symbolen, deren Zeichendarstellung ähnlich anfängt (etwa + und ++)?
- Wie entscheidet man, wo Symbole aufhören? (Leerzeichen sind nicht immer Trennzeichen!)

Lexikalische Analyse I

Problemstellung:

- Wie kann man effizient Symbole erkennen?
- Was passiert mit Kommentaren im Quelltext?
- Wie unterscheidet man zwischen Symbolen, deren Zeichendarstellung ähnlich anfängt (etwa + und ++)?
- Wie entscheidet man, wo Symbole aufhören? (Leerzeichen sind nicht immer Trennzeichen!)

Ansatz:

- Beschreibung der Symbole durch *reguläre Ausdrücke*.

Formale Sprachen

Definitionen:

- Sei A eine endliche Menge von Zeichen (Alphabet).
- Eine endliche Folge von Zeichen aus einem Alphabet A bezeichnen wir als Wort von A.
- Das leere Wort mit der Länge 0 bezeichnen wir mit ϵ.
- Die Menge aller Wörter über einem Alphabet A bezeichnen wir mit A^*.
- Eine Sprache über einem Alphabet ist eine bestimmte Teilmenge von A^*.

Beispiel:

- $A = \{\text{ich}, \text{lese}, \text{trinke}\}$, $v =$ ich lese, $w =$ trinke lese

Reguläre Ausdrücke I

- Das leere Wort ϵ ist ein regulärer Ausdruck.
- Jedes Zeichen $a \in A$ ist ein regulärer Ausdruck.
- Sind r_1 und r_2 reguläre Ausdrücke, dann sind auch

  $(r_1|r_2)$ (Alternative)

  $(r_1 r_2)$ (Konkatenation)

  $(r_1)^i$ (Exponentiation mit $i \in N$)

  $(r_1)^*$ (Kleene-Abschluß)

  $(r_1)^+$ (Positiver-Abschluß)

  reguläre Ausdrücke.
- Die durch reguläre Ausdrücke beschreibbaren Sprachen nennt man reguläre Sprachen.

Reguläre Ausdrücke II

- Reguläre Ausdrücke lassen sich in sogenannte endliche erkennende Automaten übersetzen.
- Mit erkennenden Automaten kann man sehr effizient entscheiden, ob eine Zeichenfolge einem gegebenen Ausdruck entspricht.
- Reguläre Ausdrücke werden oft verwendet, um Suchmuster zu beschreiben (z.B. `egrep`).

```
egrep '[a-z](-?[a-zA-Z0-9_]+)*-?' <file>
```

Lexikalische Analyse II

- Zur lexikalischen Analyse definiert man die Symbole der Sprache durch reguläre Ausdrücke.
- Anschließend konstruiert man einen Automaten, der die Symbole erkennt.

Lexikalische Analyse II

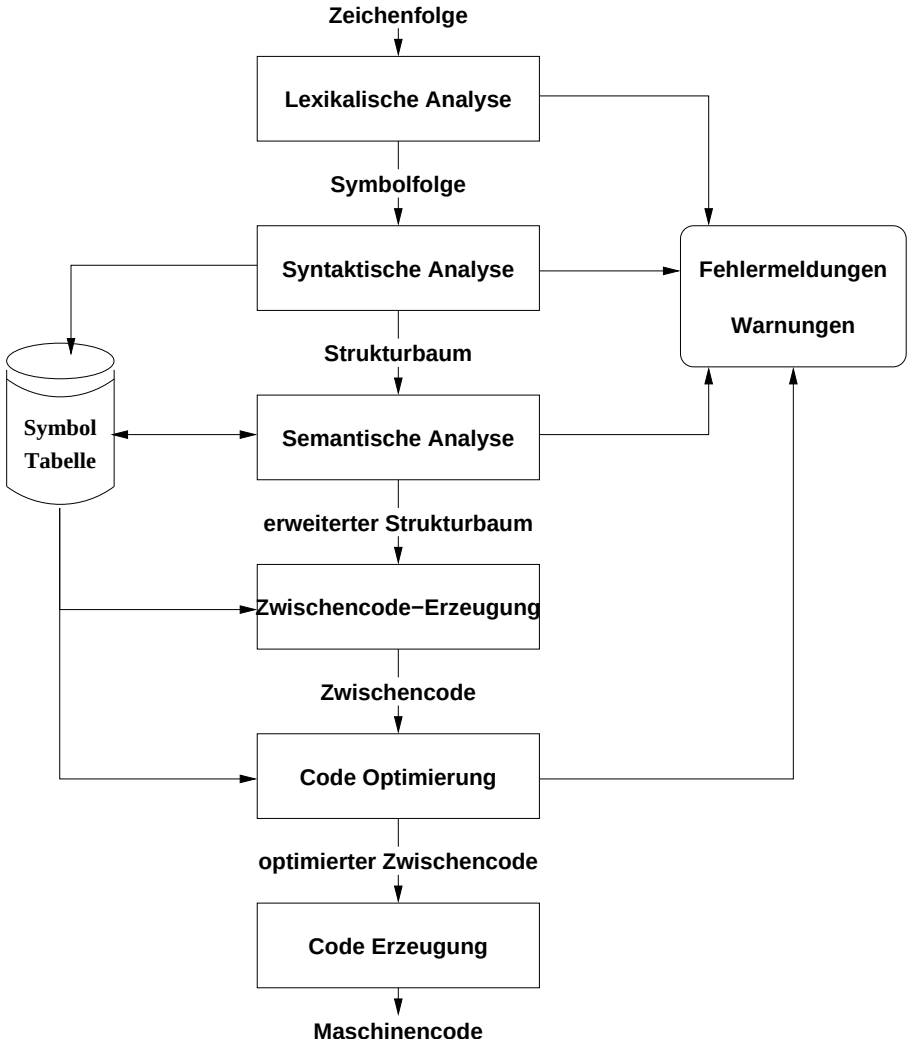
- Zur lexikalischen Analyse definiert man die Symbole der Sprache durch reguläre Ausdrücke.
- Anschließend konstruiert man einen Automaten, der die Symbole erkennt.

$\Rightarrow$ Als richtiger Informatiker schreibt man sich ein Programm zur Konstruktion des Automaten!

Fast Lexical Analyzer Generator (flex)

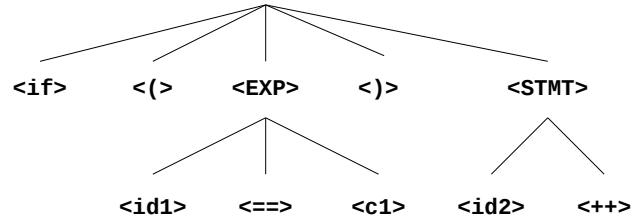
```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

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

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

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

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

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

%%
```

Phasen bei der Übersetzung




```
if (c == '\n') l++
```

```
<if> <(> <id1> <==> <c1> <)> <id2> <++>
```




```
    ld  r1, <id1>
    cmp r1, #<c1>
    jeq l1;
    jmp l2;
l1: ld  r1, <id2>
    inc r1
l2: nop
```

```
    cmpl $10,-4(%ebp)
    jne  L6
    incl -12(%ebp)
L6:
```

Syntaktische Analyse I

Problemstellung:

- Wie erzeuge ich aus dem Strom von Symbolen einen Strukturbaum?
- Wie beschreibe ich überhaupt was ein gültiger Strukturbaum ist?
- Ist die Konstruktion eines Strukturbaums immer eindeutig?

Syntaktische Analyse I

Problemstellung:

- Wie erzeuge ich aus dem Strom von Symbolen einen Strukturbaum?
- Wie beschreibe ich überhaupt was ein gültiger Strukturbaum ist?
- Ist die Konstruktion eines Strukturbaums immer eindeutig?

Ansatz:

- Beschreibung der Programmiersprache durch *kontextfreie Sprachen*.

Kontextfreie Sprache I

Eine kontextfreie Grammatik ist ein Quadrupel $G = (V_N, V_T, P, S)$ mit

- V_N endliche Menge von Nichtterminalsymbolen
- V_T endliche Menge von Terminalsymbolen
- $P \subseteq V_N \times (V_N \cup V_T)*$ endliche Menge von Poduktionen
- $S \in V_N$ ein ausgezeichnetes Startsymbol

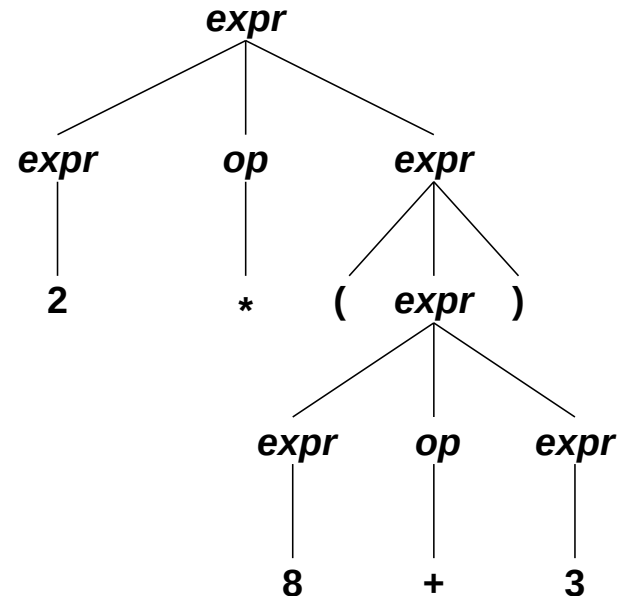
Bemerkung:

- Produktionen $(A, a) \in P$ schreibt man üblicherweise in der From $A \rightarrow a$.
- Eine kontextfreie Grammatik definiert eine kontextfreie Sprache.

Kontextfreie Sprache II

Die Grammatik $G = (V_N, V_T, P, S)$ mit $V_N = \{e^{xpr}, op, (,)\}$, $V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, \}$, $S = expr$ und den unten angegebenen Produktionen P definiert die Struktur arithmetischer Ausdrücke.

$$
\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -expr \\
expr &\rightarrow 0 \\
&\ \ \vdots \\
expr &\rightarrow 9 \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow /
\end{aligned}
$$



Syntaktische Analyse II

- Man definiert die Programmiersprache als kontextfreie Grammatik.
- Die gelesenen Symbole werden durch Anwendung der Produktionsregeln so reduziert, dass am Ende das Startsymbol der Grammatik übrig bleibt.
- Geht das mit jeder kontextfreien Grammatik?
- Ist die anzuwendende Produktion in jedem Reduktionsschritt eindeutig?

Syntaktische Analyse II

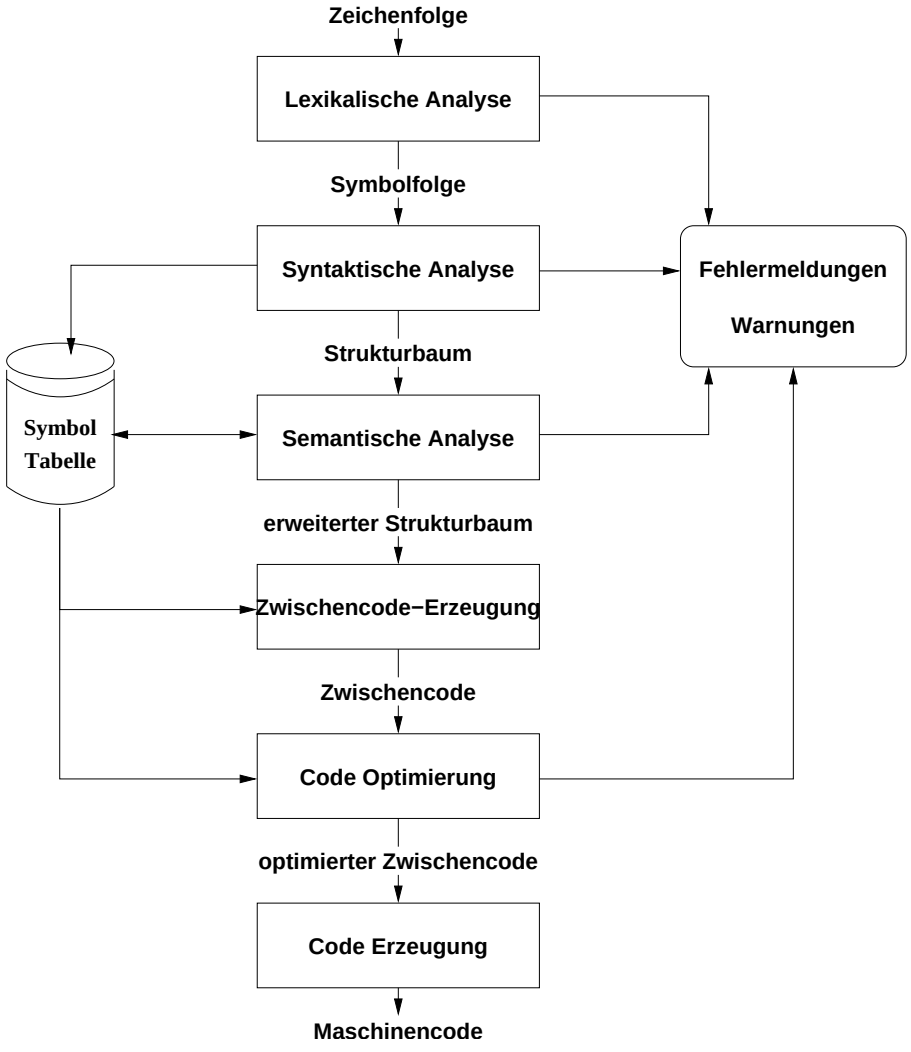
- Man definiert die Programmiersprache als kontextfreie Grammatik.
- Die gelesenen Symbole werden durch Anwendung der Produktionsregeln so reduziert, dass am Ende das Startsymbol der Grammatik übrig bleibt.
- Geht das mit jeder kontextfreien Grammatik?
- Ist die anzuwendende Produktion in jedem Reduktionsschritt eindeutig?

$\Rightarrow$ Als richtiger Informatiker schreibt man sich ein Programm zur Konstruktion eines Programms zur Reduktion der Symbole!

Compiler-Generatoren (yacc, bison)

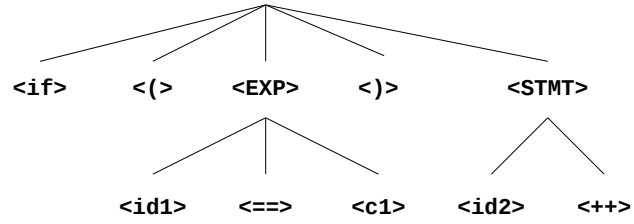
```
%%
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n' { printf("\t%.10g\n", $1); }
;

exp:      NUM { $$ = $1; }
        | exp exp '+' { $$ = $1 + $2; }
        | exp exp '-' { $$ = $1 - $2; }
        | exp exp '*' { $$ = $1 * $2; }
        | exp exp '/' { $$ = $1 / $2; }
        /* Unary minus    */
        | exp 'n' { $$ = -$1; }
;
%%
```

Phasen bei der Übersetzung




```
if (c == '\n') l++
```

```
<if> <(> <id1> <==> <c1> <)> <id2> <++>
```




```
    ld  r1, <id1>
    cmp r1, #<c1>
    jeq l1;
    jmp l2;
l1: ld  r1, <id2>
    inc r1
l2: nop
```

```
    cmpl $10,-4(%ebp)
    jne  L6
    incl -12(%ebp)
L6:
```

Danke für die Aufmerksamkeit!
Eventuell mehr im WS 2002/2003.