

Wiederholungsvorlesung STUPS

10.9.2009

Front End
(Analyse)



Back End
(Synthese)

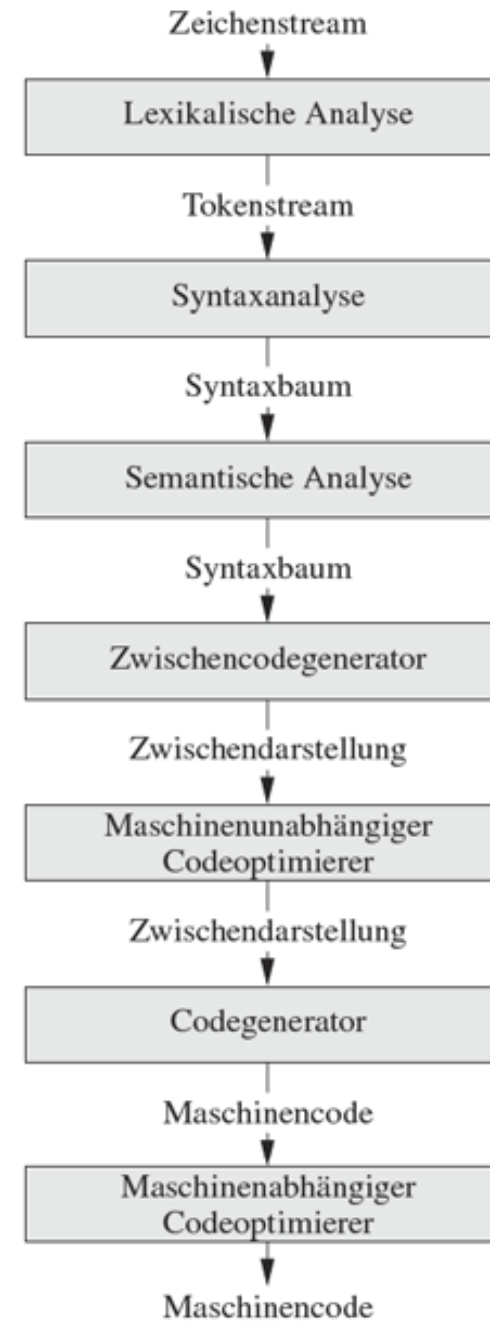


Abbildung 1.6: Phasen eines Compilers

Übersicht

| | | |
|---|----------|-----|
| 1 | position | ... |
| 2 | initial | ... |
| 3 | rate | ... |
| | | |

SYMBOLTABELLE

position = initial + rate * 60

Lexikalischer Analysator

$\langle id, 1 \rangle (=) \langle id, 2 \rangle (+) \langle id, 3 \rangle (*) \langle 60 \rangle$

Syntaktischer Analysator

```

      =
     / \
  <id,1> +
       / \
    <id,2> *
         / \
    <id,3> 60
    
```

Semantischer Analysator

```

      =
     / \
  <id,1> +
       / \
    <id,2> *
         / \
    <id,3> inttfloat
           |
           60
    
```

Zwischengeneratore

```

t1 = inttfloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
    
```

Codeoptimierer

```

t1 = id3 * 60.0
id1 = id2 + t1
    
```

Codegenerator

```

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
    
```

Lexikalische Analyse

- Prinzip: Tokens, Lexeme
- Formale Sprachen, Reguläre Ausdrücke
- Automaten: DFA, NFA
- Reguläre Ausdrücke -> NFA
- NFA -> DFA, epsilon-Hülle,
- DFA minimieren

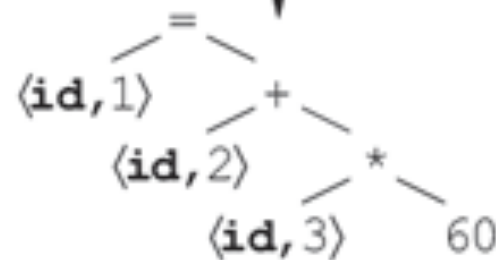


position = initial + rate * 60

Lexikalischer Analysator

$\langle \mathbf{id}, 1 \rangle \langle \mathbf{=}, 1 \rangle \langle \mathbf{id}, 2 \rangle \langle \mathbf{+}, 1 \rangle \langle \mathbf{id}, 3 \rangle \langle \mathbf{*}, 1 \rangle \langle \mathbf{60}, 1 \rangle$

Syntaktischer Analysator



| | | |
|---|----------|-----|
| 1 | position | ... |
| 2 | initial | ... |
| 3 | rate | ... |
| | | |

SYMBOLTABELLE

Ziel: Folgen von Buchstaben erkennen die als Einheit behandelt werden können

Token, Lexeme

- Ein ***Token*** ist ein Paar aus einem Namen und einem optionalen Attributwert. Der Name steht für eine Kategorie von lexikalischen Einheiten.
 - Schreibweise: <**Name**,Wert> oder <**Name**> falls kein Wert
- Ein ***Lexem*** ist eine Folge von Zeichen im Quellprogramm, die dem Muster für ein Token entspricht.

Beispiele

| Token | Informelle Beschreibung | Beispiellexeme |
|------------|---|---------------------|
| if | Zeichen i, f | if |
| else | Zeichen e, l, s, e | else |
| comparison | < oder > oder <= oder >= oder == oder != | <=, != |
| id | Buchstabe, auf den Buchstaben oder Ziffern folgen | pi, score, D2 |
| number | Alle numerischen Konstanten | 3.14159, 0, 6.02e23 |
| literal | Alles außer Anführungszeichen, was in Anführungszeichen steht | *core dumped* |

Abbildung 3.2: Beispiele für Token

Algebraische Gesetze

| Gesetz | Beschreibung |
|----------------------------------|---|
| $r s = s r$ | ist kommutativ. |
| $r (s t) = (r s) t$ | ist assoziativ. |
| $r(st) = (rs)t$ | Konkatenation ist assoziativ. |
| $r(s t) = rs rt; (s t)r = sr tr$ | Konkatenation ist distributiv über . |
| $\epsilon r = r\epsilon = r$ | ϵ ist die Identität für Konkatenation. |
| $r^* = (r \epsilon)^*$ | ϵ ist in einer Hülle garantiert. |
| $r^{**} = r^*$ | * ist idempotent. |

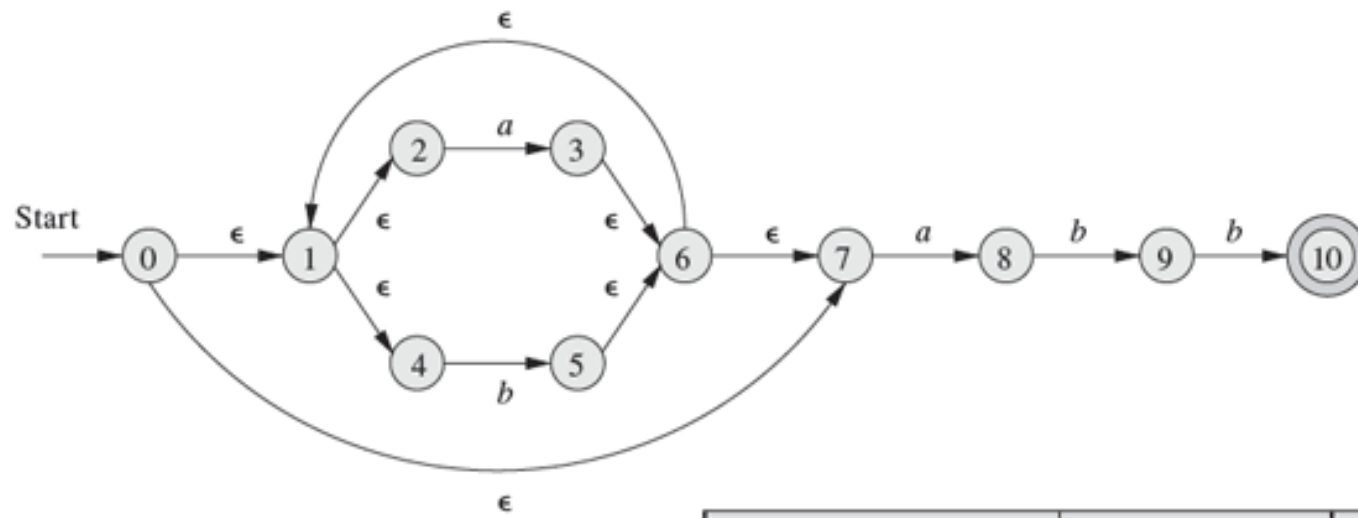
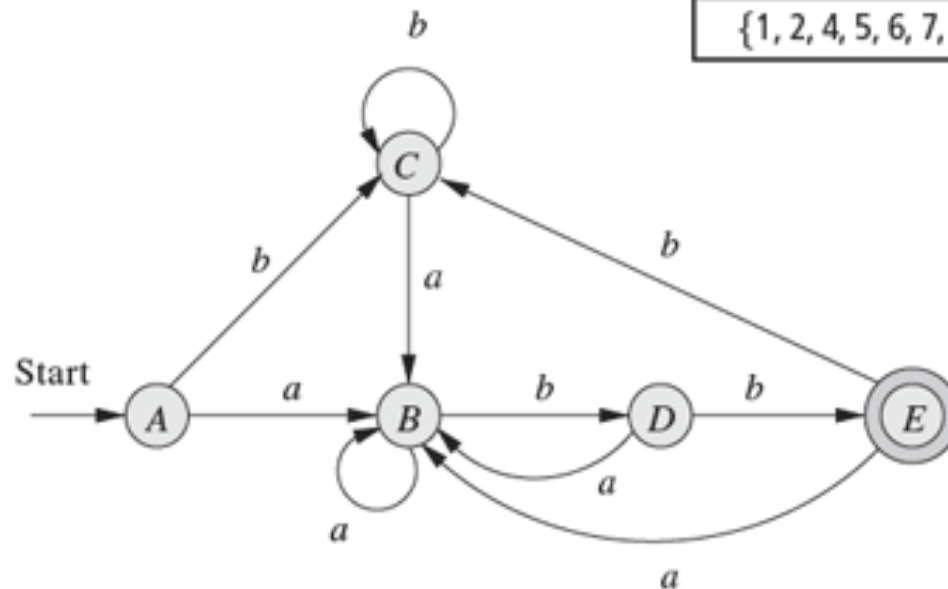
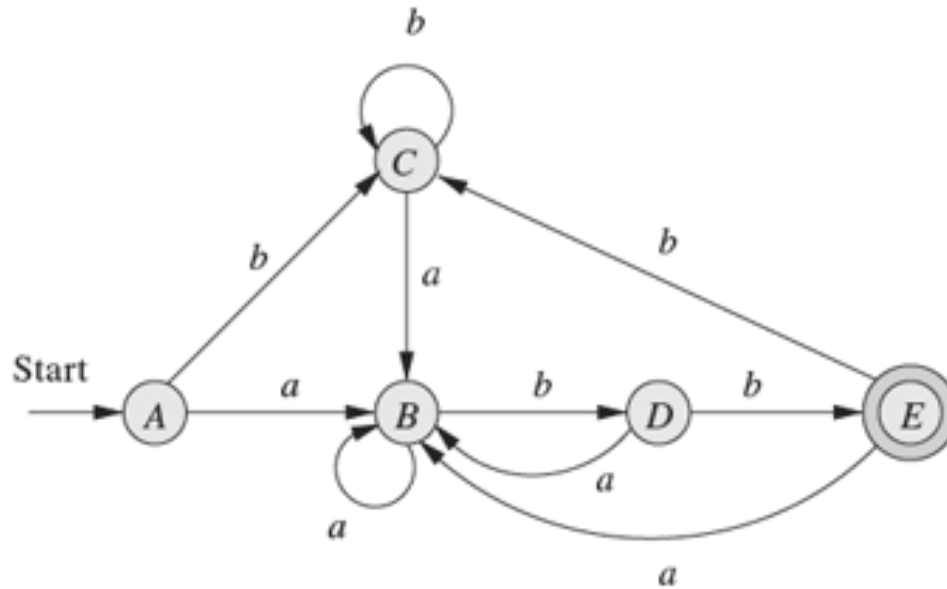


Abbildung 3.34: NFA N für $(a|b)^+abb$

| NFA-Zustand | DFA-Zustand | a | b |
|------------------------|-------------|---|---|
| {0, 1, 2, 4, 7} | A | B | C |
| {1, 2, 3, 4, 6, 7, 8} | B | B | D |
| {1, 2, 4, 5, 6, 7} | C | B | C |
| {1, 2, 4, 5, 6, 7, 9} | D | B | E |
| {1, 2, 4, 5, 6, 7, 10} | E | B | C |



Minimierung: Beispiel



Anfangspartition:
 $\{A,B,C,D\} \{E\}$

| Zustand | a | b |
|---------|---|---|
| A | B | A |
| B | B | D |
| D | B | E |
| E | B | A |

Abbildung 3.65: Übergangstabelle des Minimalzustands-DFA

Syntaktische Analyse

- Konzept: Programmstruktur als Baum darstellen
- Kontextfreie Grammatiken
- Predictive Parsing:
 - LL(1): First, Follow, Nullable; Parsertabelle
 - Parser per Hand schreiben
 - Mehrdeutigkeit, Linksrekursion eliminieren
- Bottom-Up Parsing: Shift, Reduce

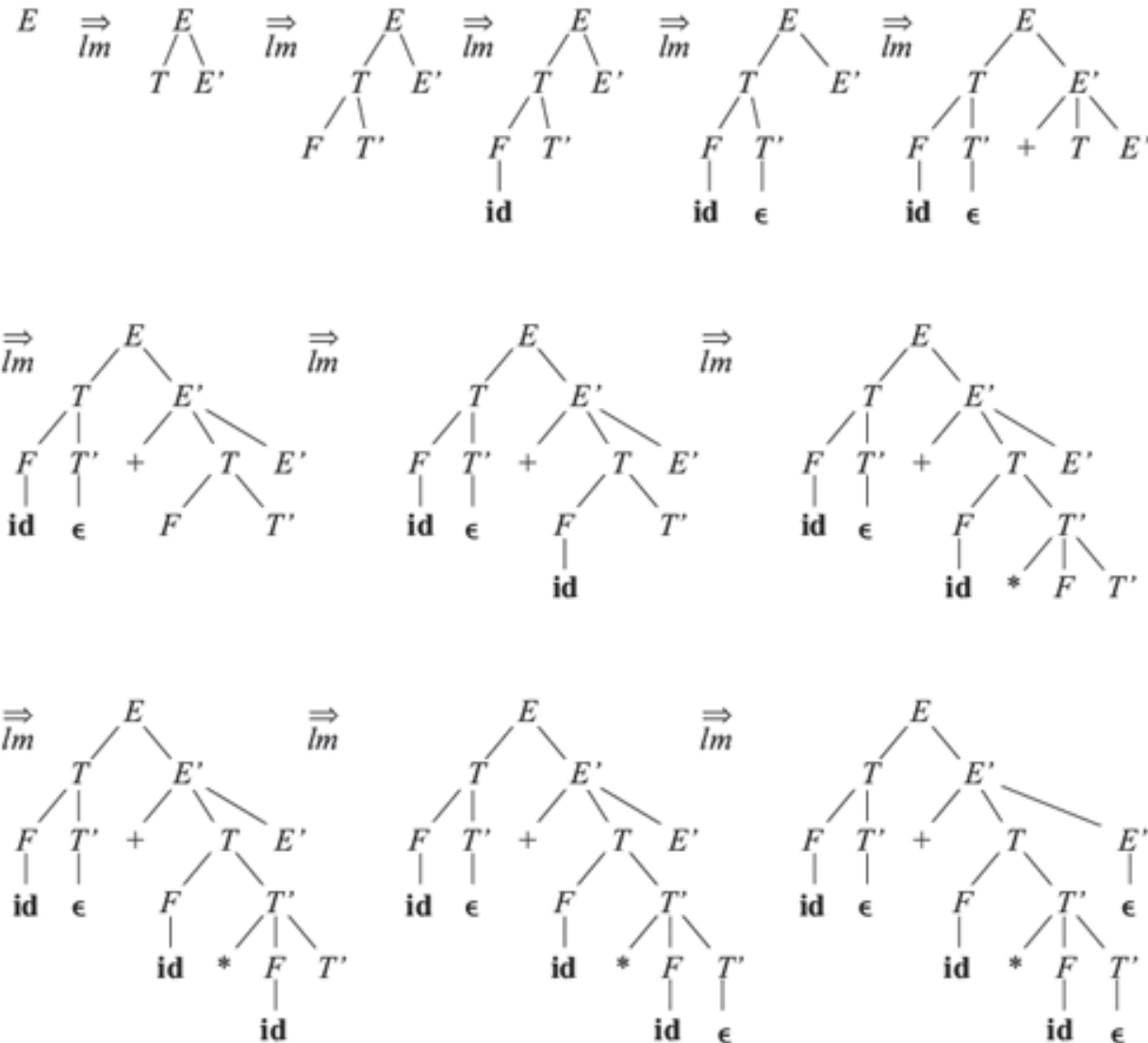
KfG

- Terminologie: Terminal, Nichtterminal, Produktion
- Ableitung, Linksableitung, Rechtsableitung
- erzeugte Sprache
- Mehrdeutigkeit
- Parsebaum

id+id*id

Top-Down Parsing

- $E \rightarrow T E'$
- $E' \rightarrow + T E' \mid \epsilon$
- $T \rightarrow F T'$
- $T' \rightarrow * F T' \mid \epsilon$
- $F \rightarrow (E) \mid id$



First und Follow

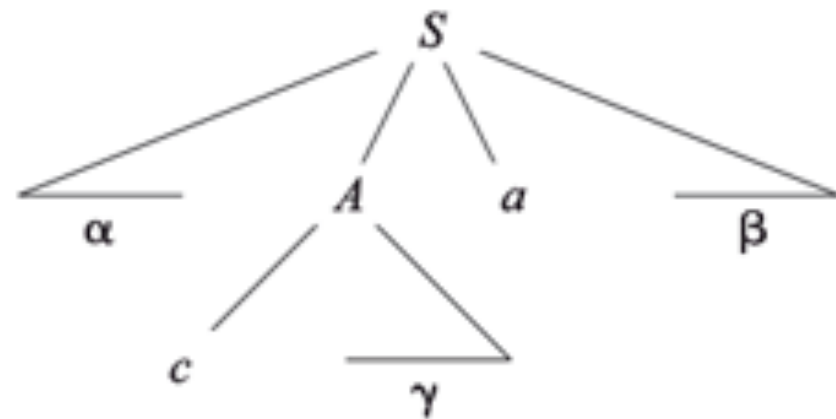


Abbildung 4.15: Terminal c ist in $\text{FIRST}(A)$ und a in $\text{FOLLOW}(A)$.

Anderes Beispiel:

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

| Nichtterminal | Eingabesymbol | | | | | |
|---------------|---------------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
| | id | + | * | (|) | \$ |
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | $E' \rightarrow +TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow \text{id}$ | | | $F \rightarrow (E)$ | | |

Linksfaktorisierung

- Wiederhole bis keine Änderung mehr:
 - Für jedes Nicht-Terminal X :
 - Finde den längsten gemeinsamen Präfix α von 2 oder mehr Alternativen
 - Falls $\alpha \neq \varepsilon$, dann ersetze

$$X \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_k \mid \gamma_1 \mid \dots \mid \gamma_n$$

• By

$$X \rightarrow \alpha X' \mid \gamma_1 \mid \dots \mid \gamma_n$$

$$X' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k$$

Eliminieren der Linksrekursion

– Umschreiben nach Rechtsrekursion:

- $E \rightarrow E + T$
- $E \rightarrow T$

- $X \rightarrow X\gamma_1$
- $X \rightarrow X\gamma_2$
- $X \rightarrow \alpha_1$
- $X \rightarrow \alpha_2$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E'$$

$$E' \rightarrow \varepsilon$$

$$X \rightarrow \alpha_1 X'$$

$$X \rightarrow \alpha_2 X'$$

$$X' \rightarrow \gamma_1 X'$$

$$X' \rightarrow \gamma_2 X'$$

$$X' \rightarrow \varepsilon$$

Mehrdeutige Grammatiken

$$\text{Ex} \rightarrow \text{Nat} \mid \text{ID} \mid (\text{Ex}) \mid \text{Ex} + \text{Ex} \mid \text{Ex} * \text{Ex}$$

- Lösung 1: Prioritäten Explizit Kodieren
 - Siehe “dangling else” Problem von vorher
- Lösung 2: Grammatik umschreiben
 - Verlangt Nachdenken
 - Nicht immer möglich

$$\begin{aligned} \text{Ex} &\rightarrow \text{Ex} + \text{Term} \mid \text{Term} \\ \text{Term} &\rightarrow \text{Term} * \text{Factor} \mid \text{Factor} \\ \text{Factor} &\rightarrow \text{Nat} \mid \text{ID} \mid (\text{Ex}) \end{aligned}$$

Beispiel 1

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

| Stack | Eingabe | Aktion |
|------------------------|--------------------------------------|--|
| \$ | id ₁ * id ₂ \$ | Verschieben (shift) |
| \$ id ₁ | * id ₂ \$ | Reduzieren durch $F \rightarrow \text{id}$ |
| \$ F | * id ₂ \$ | Reduzieren durch $T \rightarrow F$ |
| \$ T | * id ₂ \$ | Verschieben |
| \$ T * | id ₂ \$ | Verschieben |
| \$ T * id ₂ | \$ | Reduzieren durch $F \rightarrow \text{id}$ |
| \$ T * F | \$ | Reduzieren durch $T \rightarrow T * F$ |
| \$ T | \$ | Reduzieren durch $E \rightarrow T$ |
| \$ E | \$ | Akzeptieren |

Abbildung 4.28: Konfigurationen eines Shift-Reduce-Parsers bei der Eingabe id₁ * id₂

Semantische Analyse

- Semantische Werte
- Semantische Aktionen
- Prinzip der Werkzeuge (javacc, sablecc, lex + yacc)
- Konzept: Abstrakter Syntaxbaum

Backend

- Konzept: Zwischencode; 0-Adress, 3-Adress, Generierung, Grundblöcke
- Zwischencode -> Code durch Baumersetzung
- Liveness analyse, Register-Allokation

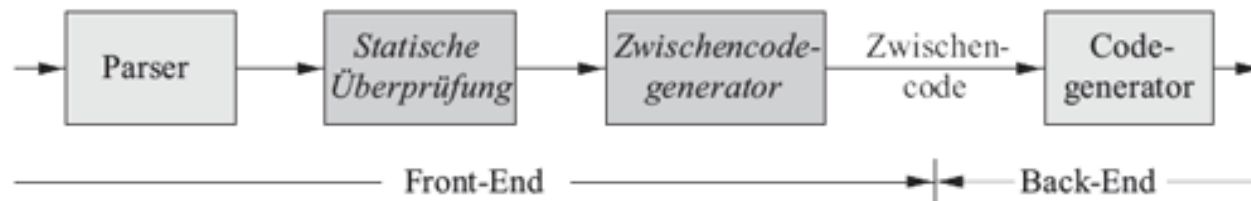


Abbildung 6.1: Logische Struktur eines Compiler-Front-End

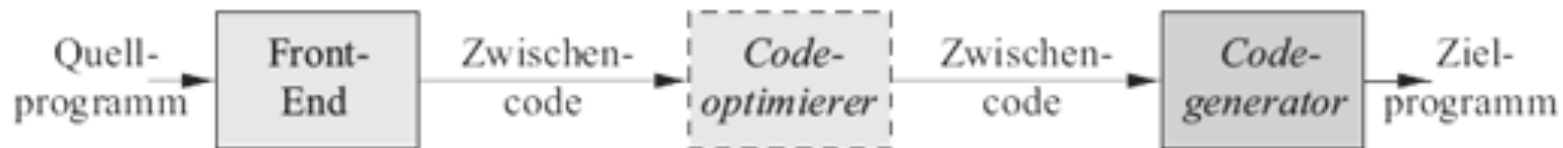


Abbildung 8.1: Die Position des Codegenerators

Format des Zwischencodes

- Befehle:
 - $x = y \text{ op } z$, $x = \text{op } y$, $x = y$
 - goto L
 - if x goto L und ifFalse x goto L
 - if x relop y goto L
 - $x = y[i]$ und $x[i] = y$
 - $x = \&y$, $x = *y$, $x* = y$
- Adressen:
 - Namen, Konstanten, temporäre Variablen

Beispiel

- do i=i+1; while (a[i]<v);

```
L:  t1 = i + 1  
    i  = t1  
    t2 = i * 8  
    t3 = a [ t2 ]  
    if t3 < v goto L
```

a Symbolische Bezeichnungen

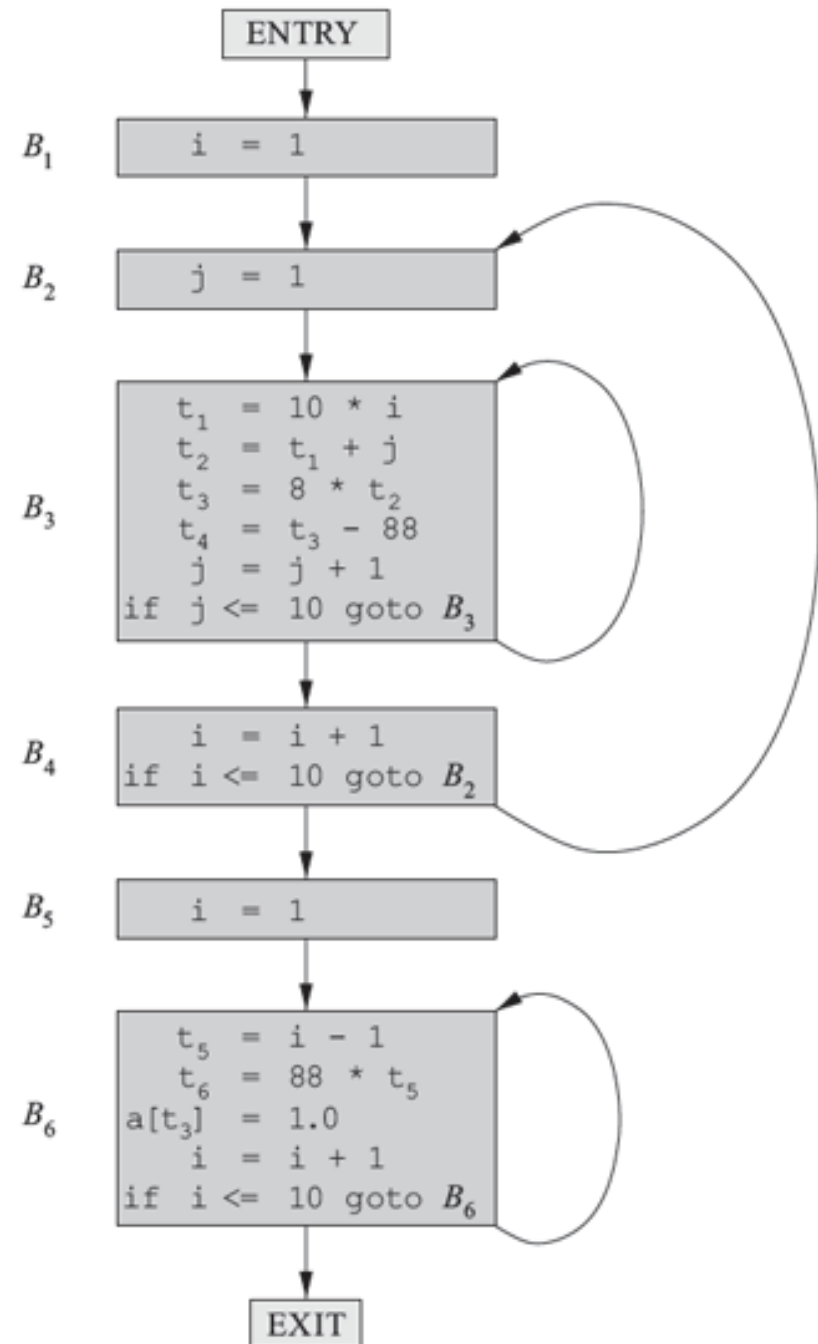
```
100: t1 = i + 1  
101: i  = t1  
102: t2 = i * 8  
103: t3 = a [ t2 ]  
104: if t3 < v goto 100
```

b Positionsnummern

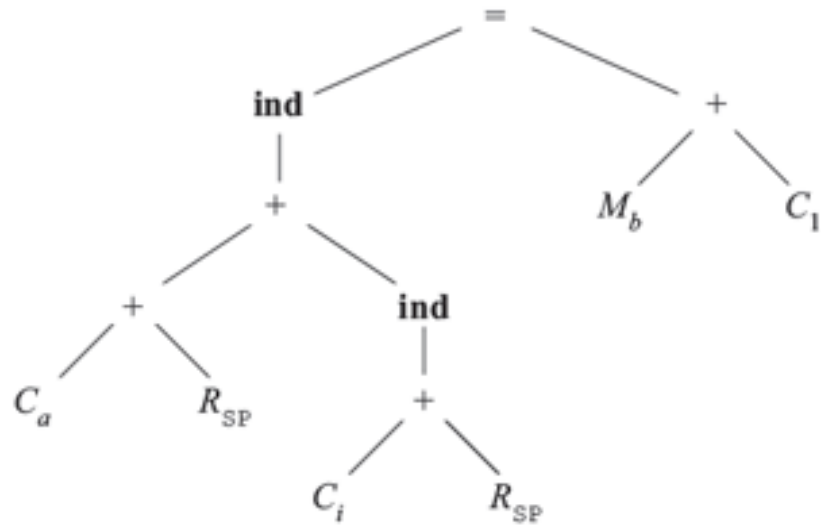
Abbildung 6.9: Zwei Arten, Drei-Adress-Anweisungen zu benennen

Beispiel

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
for i from 1 to 10 do
  for j from 1 to 10
    a[i, j] = 0.0;
for i from 1 to 10 do
  a[i, i] = 1.0;
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t3] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Beispiel



1) $R_0 \leftarrow C_a$ { LD R0, #a }

7) $R_0 \leftarrow$ { ADD R0, R0, SP }

| | | |
|----|------------------------|-----------------------|
| 1) | $R_i \leftarrow C_a$ | { LD Ri, #a } |
| | | |
| 6) | $R_i \leftarrow$ R_j | { ADD Ri, Ri, a(Rj) } |
| 7) | $R_i \leftarrow$ R_j | { ADD Ri, Ri, Rj } |
| 8) | $R_i \leftarrow$ C_1 | { INC Ri } |

Abbildung 8.20: Baumersetzungsregeln für einige Zielmaschinenbefehle

Example: Liveness Analysis

Defs

Uses

t7

t101

t102

t103

t104

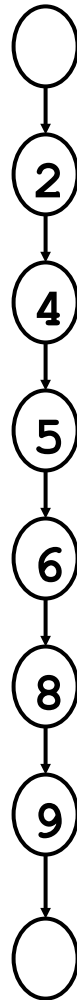
t105

t102

t101, t103

t105, t104

t7

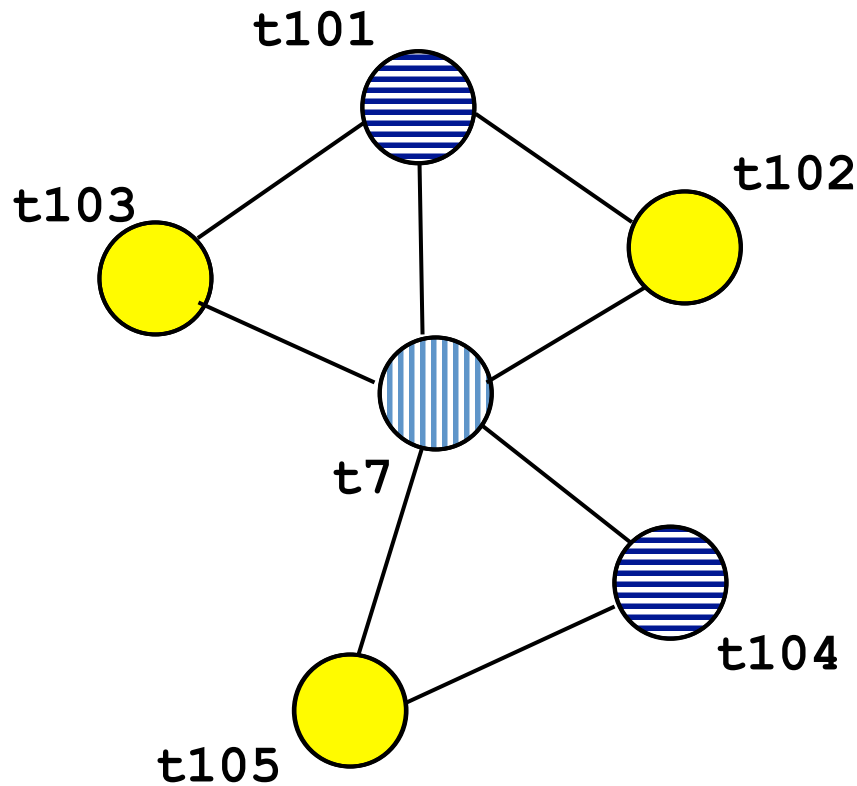



t7 t101 t102 t103 t104 t105


```

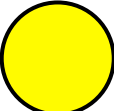
(2) LOAD t101, a(fp)
(4) ADDI t102, r0, 4
(5) MUL t103, t7, t102
(6) ADD t104, t101, t103
(8) LOAD t105, x(fp)
(9) STORE t105, 0(t104)
  
```

Example: Interference Graph



 = r_i

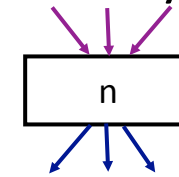
 = r_1

 = r_2

```
LOAD  r1, a(fp)
ADDI  r2, r0, 4
MUL   r2, ri, r2
ADD   r1, r1, r2
LOAD  r2, x(fp)
STORE r2, 0(r1)
```

Computing Liveness

- **Def**(n) = set of vars defined in node n
- **Use**(n) = set of vars used
- Var **Live** on edge: directed path from that edge to use not via any def
- **in**[n] = set of vars live on any in-edge (**live-in**)
- **out**[n] = set of vars live on any out-edge (**live-out**)



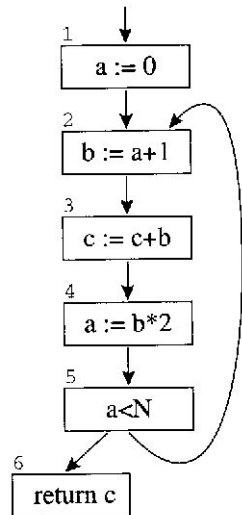
$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

EQUATIONS 10.3. Dataflow equations for liveness analysis.

```

a ← 0
L1 : b ← a + 1
      c ← c + b
      a ← b * 2
      if a < N goto L1
      return c
    
```



for each n

$in[n] \leftarrow \{\}; out[n] \leftarrow \{\}$

repeat

for each n

$in'[n] \leftarrow in[n]; out'[n] \leftarrow out[n]$

$in[n] \leftarrow use[n] \cup (out[n] - def[n])$

$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$

until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

ALGORITHM 10.4. Computation of liveness by iteration.

Prolog: Theorie

- Aussagenlogik, Prädikatenlogik
 - Interpretation, Modelle
 - logische Konsequenz, Äquivalenz
 - Refutationsbeweis, Resolution, KNF
 - Unifikation, Unifikationsalgorithmus
- Prolog
 - Prolog \leftrightarrow Logik (KNF), Horn Klauseln
 - SLD Resolution, SLD-Bäume

Prolog: Praxis

- Siehe Übungen
- Listenmanipulation
- Suche: Tiefensuche, iterative Deepening, Breitensuche, A*
- (Minimax)
- Datenstrukturen als Prologterme darstellen