Kapitel 6

Zwischencode und Grundblöcke

Übersicht

- Zwischencode: Drei-Adress, Bytecode, ...
- Zielprogram: RISC, CISC, Stackbasiert
- Codegenerator:
  - Befehlsauswahl
  - Registervergabe und -zuteilung

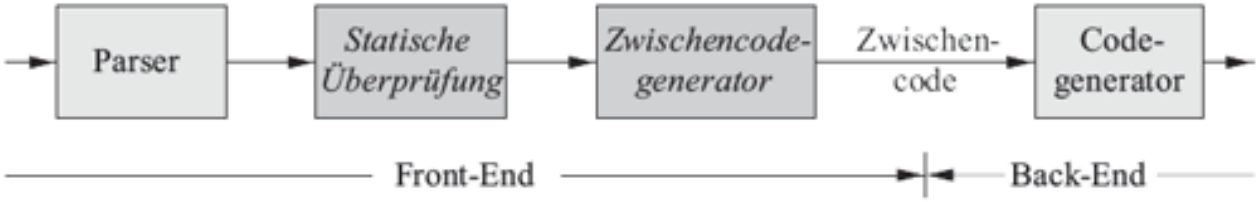


Abbildung 6.1: Logische Struktur eines Compiler-Front-End

Abbildung 8.1: Die Position des Codegenerators

Themen

- Darstellung von Zwischencode (Intermediate code representation)
- Schritte der Codegenerierung aus Zwischencode:
  - (Naive) Generierung von Maschinencode / Befehlsauswahl

    (Annahme: bel. viele Register sind vorhanden)
  - Registerallokation / Registervergabe

    (Zuordnung zu real Vorhanden Registern)
  - Scheduling und (Plattformabhängige) Codeoptimierung

Zwischencode

- Compiler erzeugen normalerweise eine Zwischendarstellung (*intermediate representation* IR)
  - Abstrakte Maschinensprache
  - Trennt zwischen Front-End und Back-End Abwägungen
  - Erhört Modularität und Protierbarkeit

- Eigenschaften von Zwischencode:
  - Kann einfach von Front-End erzeugt werden
  - Maschinencode kann einfach daraus generiert werden
  - simpel, straightforward, relativ low-level
  - Linear vs grafisch

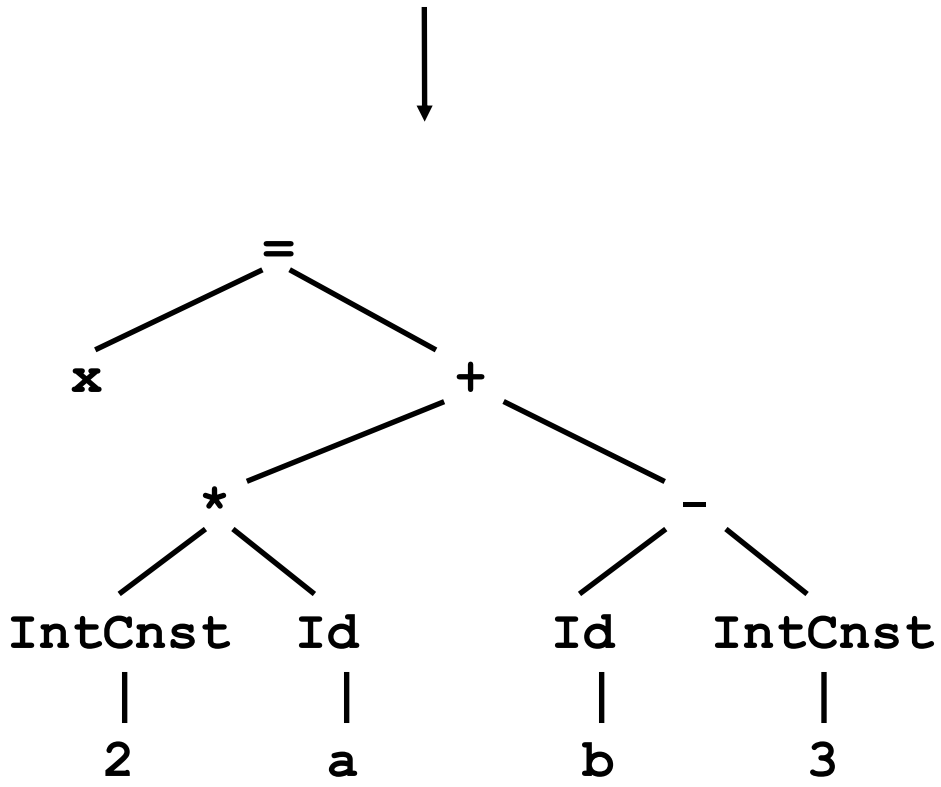
Arten von IR

- High-level Zwischencode
  - Z.B. Abstrakter Syntax Baum (AST)
- Medium-level Zwischencode
  - Explizite Darstellung von:
    - Variablen, Zwischenergebnissen, Registern
    - Unterstützung von Blockstrukturen und Funktionen
  - Einfache bedingte und unbedingte Sprünge
- Low-level Zwischencode
  - Fast eins zu eins Beziehung zum Zielcode
    - Eingabe für die finale Befehlsauswahl (instruction selection) oder nach dem Optimierer

Null-Adress-Code (Zero-address code)

- abstrakter Code für eine Stackmaschine:

```
x = 2 * a + (b - 3);
```




```
LDC   2
LOAD  a
MUL
LOAD  b
LDC   3
SUB
ADD
STORE x
```

Übersetzung von Ausdrücken

- Übersetuzng nach 0-Adress-Code ist einfach:

```
class Binop extends AST {
  private AST left, right;
  private int op;
  .
  .
  void translate() {
    left.translate();
    right.translate();
    switch (op) {
      case PLUS:
        emit0(ADD);  break;
      case MINUS:
        emit0(SUB);  break;
      case TIMES:
        emit0(MUL); break;
      ...
    }
  }
}
```

```
class IntCnst extends AST {
  private int value;
  .
  .
  void translate() {
    emit1(LDC, value);
  }
}

etc.
```

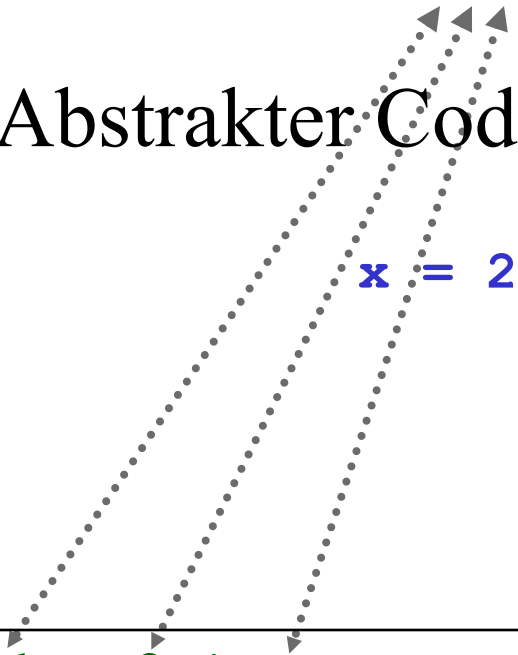
Format des Zwischencodes

- Befehle:
  - x = y **op** z, x = **op** y, x = y
  - goto L
  - if x goto L und ifFalse x goto L
  - if x relop y goto L (relop: = , <, >)
  - x = y[i] und x[i] = y
  - x = &y, x = *y, x* = y
- Adressen (d.h. x, y, z, i):
  - Namen, Konstanten, temporäre Variablen

Drei-Adress-Code

- Abstrakter Code für eine Register Maschine:

```
x = 2 * a + (b - 3);
```



```
t1 = 2 * a
t2 = b - 3
t3 = t1+t2
x = t3
```

```
(MUL, 2, a)        (1)
(SUB, b, 3)        (2)
(ADD, $1, $2)      (3)
(ST, $3,x)         (4)
```

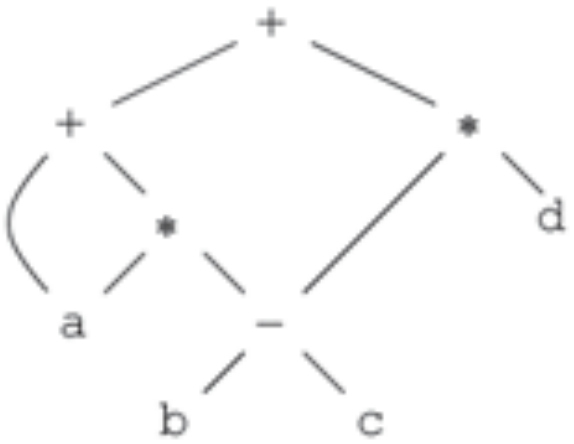
Result referred to by position

Triples

Anmerkung: maximal ein Operator auf der rechten Seite

Erzeugung des Zwischencodes

- 1. DAG Generierung
- 2. Drei-Adress Code:
  - lineare Darstellung des DAGs
  - interne Knoten → explizite Namen



a DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

b Drei-Adress-Code

Abbildung 6.8: Ein DAG mit entsprechendem Drei-Adress-Code

Beispiel

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

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

Darstellungsweisen I

- **Quadrupel**
- Tripel
- indirekte Tripel

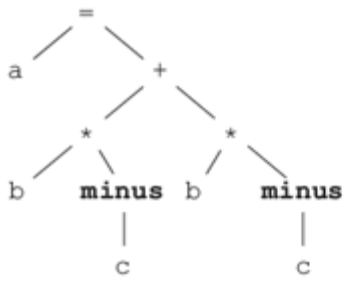
a Drei-Adress-Code

b Quadrupel

Abbildung 6.10: Drei-Adress-Code und seine Darstellung in Quadrupeln

Darstellungsweisen II

- Quadrupel
- **Tripel**
- **indirekte Tripel**



a Syntaxbaum

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

b Tripel

Abbildung 6.11: Darstellung von $a = b * -c + b * -c$

	Befehl
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Abbildung 6.12: Drei-Adress-Code als indirekte Tripel-Darstellung

SSA: Static Single Assignment

- Alle Zuweisungen: verschiedene Variable

Abbildung 6.13: Zwischenprogramm in Drei-Adress-Code und SSA

Generierung des Zwischencodes

- Durchlauf des ASTs bzw. DAGs
  - Zwei Attribute:
    - code: Drei-Adress-Code
    - addr: Wo wird das Ergebnis gespeichert

Produktion	Semantische Regeln
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \parallel$ *gen*(*top.get*(**id**.*lexeme*) '=' $E.addr$)
$E \rightarrow E_1 + E_2$	$E.addr =$ **new***Temp*() $E.code = E_1.code \parallel E_2.code \parallel$ *gen*($E.addr$ '=' $E_1.addr$ '+' $E_2.addr$)
$\mid -E_1$	$E.addr =$ **new***Temp*() $E.code = E_1.code \parallel$ *gen*($E.addr$ '=' '**minus**' $E_1.addr$)
$\mid (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
| **id**	$E.addr =$ *top.get*(**id**.*lexeme*) $E.code =$ ''

Abbildung 6.19: Drei-Adress-Code für Ausdrücke

a = b + - c

t1 = minus c
t2 = b + t1
a = t2

top: Symboltabelle

Generierung des Zwischencodes 2

- Beispiel: c+a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

```
S → id = E ;   { gen(top.get(id.lexeme) '=' E.addr); }
  | L = E ;    { gen(L.array.base '[' L.addr ']' '=' E.addr); }
E → E1 + E2    { E.addr = new Temp();
                 gen(E.addr '=' E1.addr '+' E2.addr); }
  | id         { E.addr = top.get(id.lexeme); }
  | L          { E.addr = new Temp();
                 gen(E.addr '=' L.array.base '[' L.addr ']'); }
L → id [ E ]   { L.array = top.get(id.lexeme);
                 L.type = L.array.type.elem;
                 L.addr = new Temp();
                 gen(L.addr '=' E.addr '*' L.type.width); }
  | L1 [ E ]   { L.array = L1.array;
                 L.type = L1.type.elem;
                 t = new Temp();
                 L.addr = new Temp();
                 gen(t '=' E.addr '*' L.type.width);
                 gen(L.addr '=' L1.addr '+' t); }
```

ung 6.22: Semantische Aktionen für Arrayreferenzen

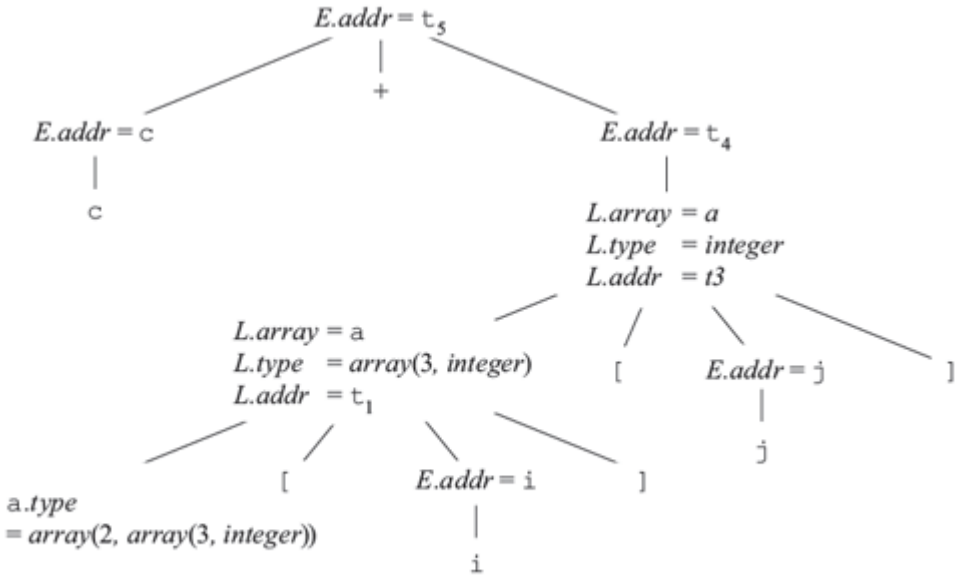


Abbildung 6.23: Annotierter Parse-Baum für c +a[i][j]

Generierung des Zwischencodes 3

- Für Kontrollflussanweisungen: true und false Labels

$S \rightarrow$ `if ( B )` S_1

$S \rightarrow$ `if ( B )` S_1 `else` S_2

$S \rightarrow$ `while ( B )` S_1

Mehr Details in Kapitel 6

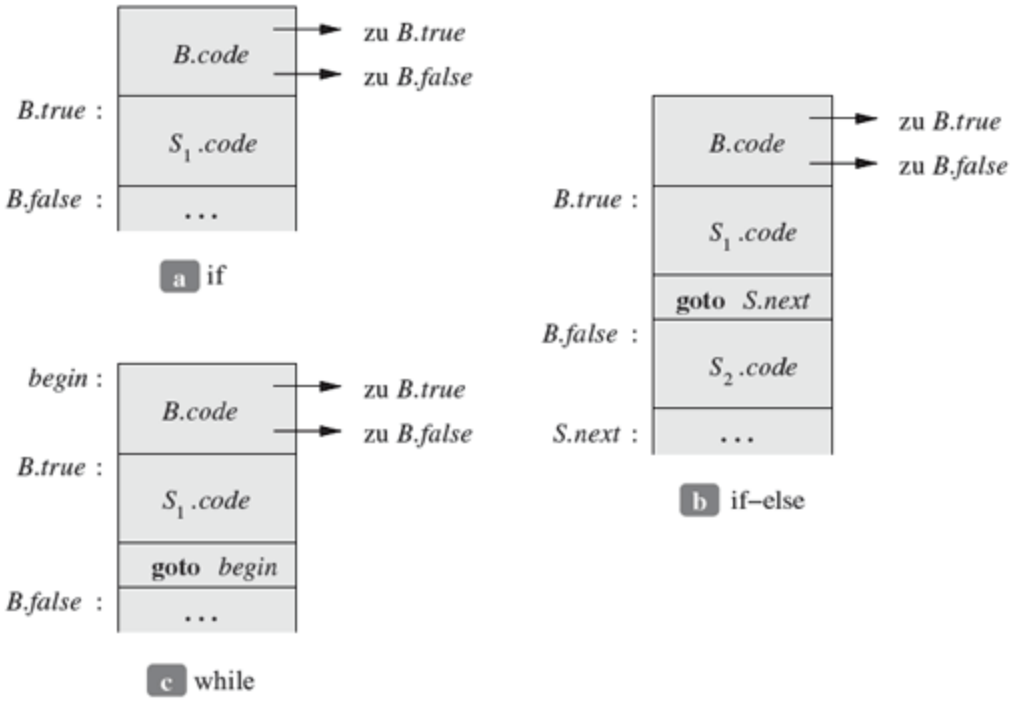


Abbildung 6.35: Code für if-, if-else- und while-Anweisungen

Generierung des Zwischencodes 4

Produktion	Semantische Regeln
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen($**'goto'** $S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen($**'goto'** $begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

bildung 6.36: Syntaxgerichtete Definition für Kontrollflussanweisungen

Produktion	Semantische Regeln
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1$ && B_2	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow$! B_1	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($**'if'** $E_1.addr$ **rel**$.op$ $E_2.addr$ **'goto'** $B.true)$ $\mid\mid gen($**'goto'** $B.false)$
$B \rightarrow$ **true**	$B.code = gen($**'goto'** $B.true)$
$B \rightarrow$ **false**	$B.code = gen($**'goto'** $B.false)$

Abbildung 6.37: Generierung von Drei-Adress-Code für boolesche Ausdrücke

Mehr Details in Kapitel 6

Grundblöcke (Drachenbuch Kap. 8.4)

- Maximale konsekutive Sequenz von Drei-Adress Anweisungen so dass:
  - Kontrollfluss kann nur durch den ersten Befehl in den Block gelangen
  - Steuerung verlässt den Block ohne Halt/Verzweigung mit Ausnahme des letzten Befehls
- Werden in Flussgraphen zusammengefasst

Ermitteln der Grundblöcke

METHODE: Zuerst ermitteln wir die Befehle im Zwischencode, die als *Anführer* fungieren, also die ersten Befehle in einem Grundblock. Die Anführer werden anhand folgender Regeln bestimmt:

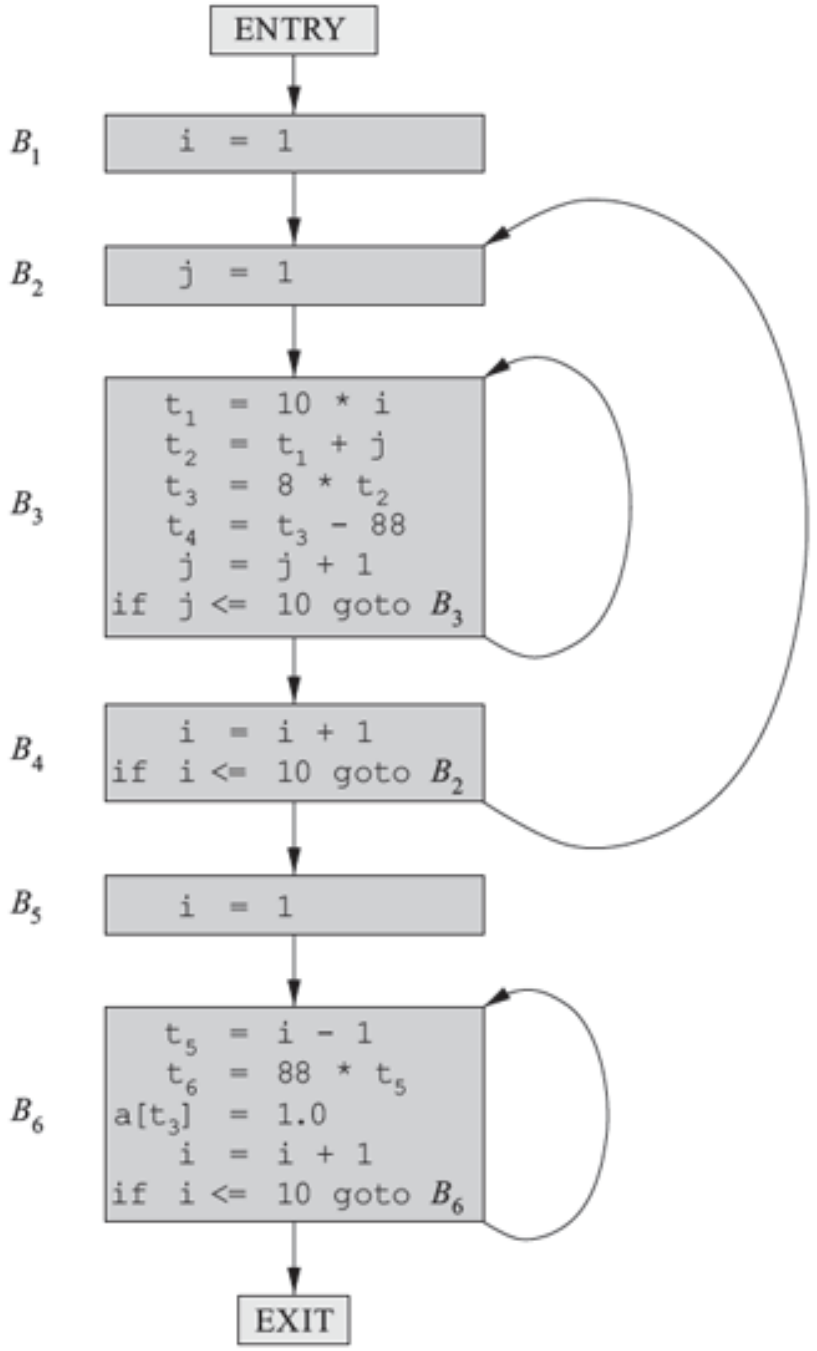
1. Der erste Befehl im Drei-Adress-Zwischencode ist ein Anführer.
2. Jeder Befehl, der Ziel eines bedingten oder unbedingten Sprunges ist, ist ein Anführer.
3. Jeder Befehl, der direkt auf einen bedingten oder unbedingten Sprung folgt, ist ein Anführer.

Der Grundblock für einen Anführer besteht dann aus diesem selbst sowie allen Befehlen bis zum nächsten Anführer, ohne diesen einzuschließen, oder bis zum Ende des Zwischenprogramms. □

Beispiel

```
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;
```

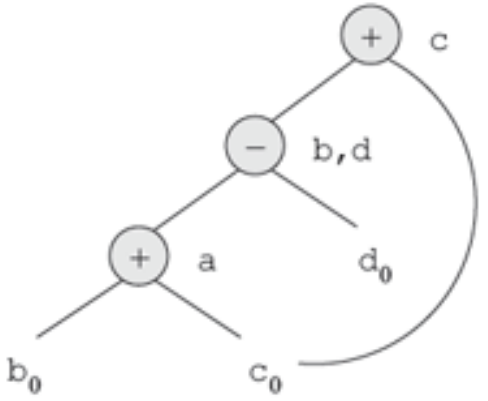
```
 1) i = 1
 2) j = 1
 3) t1 = 10 * i
 4) t2 = t1 + j
 5) t3 = 8 * t2
 6) t4 = t3 - 88
 7) a[t4] = 0.0
 8) j = j + 1
 9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Optimierungen von Grundblöcken

- DAG-Darstellung (8.5.1 ff); lokale gemeinsame Ausdrücke

```
a = b + c
b = a - d
c = b + c
d = a - d
```



- Dead Code Elimination
- Algebraische Identitäten
  - x+0 = 0+x = x , ...

Zusammenfassung:

- Motivation von Zwischencode
- Arten und Darstellung:
  - AST, DAG, 0-Address Code 3-Address Code
- Generierung durch syntaxgerichtete Definition
- IR und Grundblöcke