

Kapitel 6 & 8:

Codeerzeugung 1

Übersicht

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

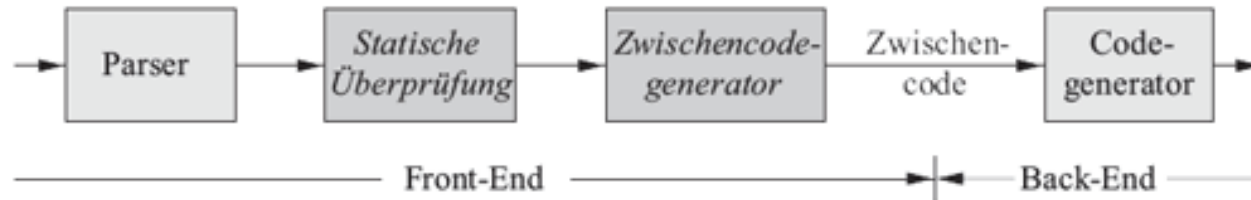


Abbildung 6.1: Logische Struktur eines Compiler-Front-End

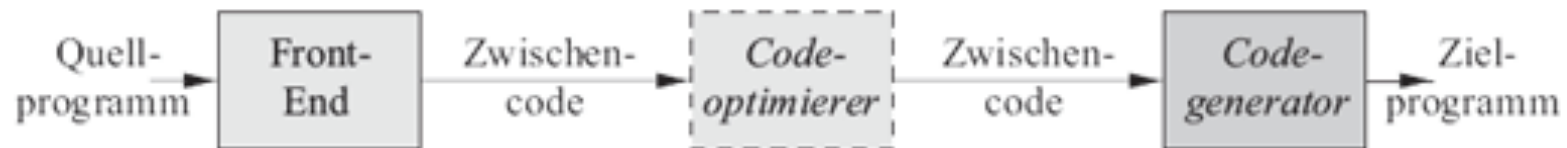


Abbildung 8.1: Die Position des Codegenerators

Naive Codeerzeugung

- $x = y+z$

```
LD R0, y      // R0 = y      (y in Register R0 laden)
ADD R0, R0, z // R0 = R0 + z  (z zu R0 addieren)
ST x, R0      // x = R0      (R0 in x speichern)
```

- $a=b+c; d=a+e$ $a=a+1$

```
LD R0, b      // R0 = b
ADD R0, R0, c // R0 = R0 + c
ST a, R0      // a = R0
LD R0, a      // R0 = a
ADD R0, R0, e // R0 = R0 + e
ST d, R0      // d = R0
```

```
LD R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST a, R0      // a = R0
```

Optimale Codeerzeugung

$t = a + b$	$t = a + b$
$t = t * c$	$t = t + c$
$t = t / d$	$t = t / d$
a	b

Abbildung 8.2: Zwei Drei-Adress-Codefolgen

		L	R0, a
L	R1, a	A	R0, b
A	R1, b	A	R0, c
M	R0, c	SRDA	R0, 32
D	R0, d	D	R0, d
ST	R1, t	ST	R1, t
a		b	

Abbildung 8.3: Optimale Maschinencodefolgen

Themen

- Darstellung von Zwischencode (Intermediate code representation)
- Generierung von Maschinencode
- Registerallokation
- Codeoptimierung

Intermediate Code

- Compilers commonly generate an *intermediate representation (IR)*
 - abstract machine language
 - separates front-end and back-end considerations
 - enhances modularity and portability
- Characteristics of IR:
 - convenient for front-end to generate
 - convenient to translate to real machine code
 - simple, straightforward, fairly low-level
 - Linear vs graphical

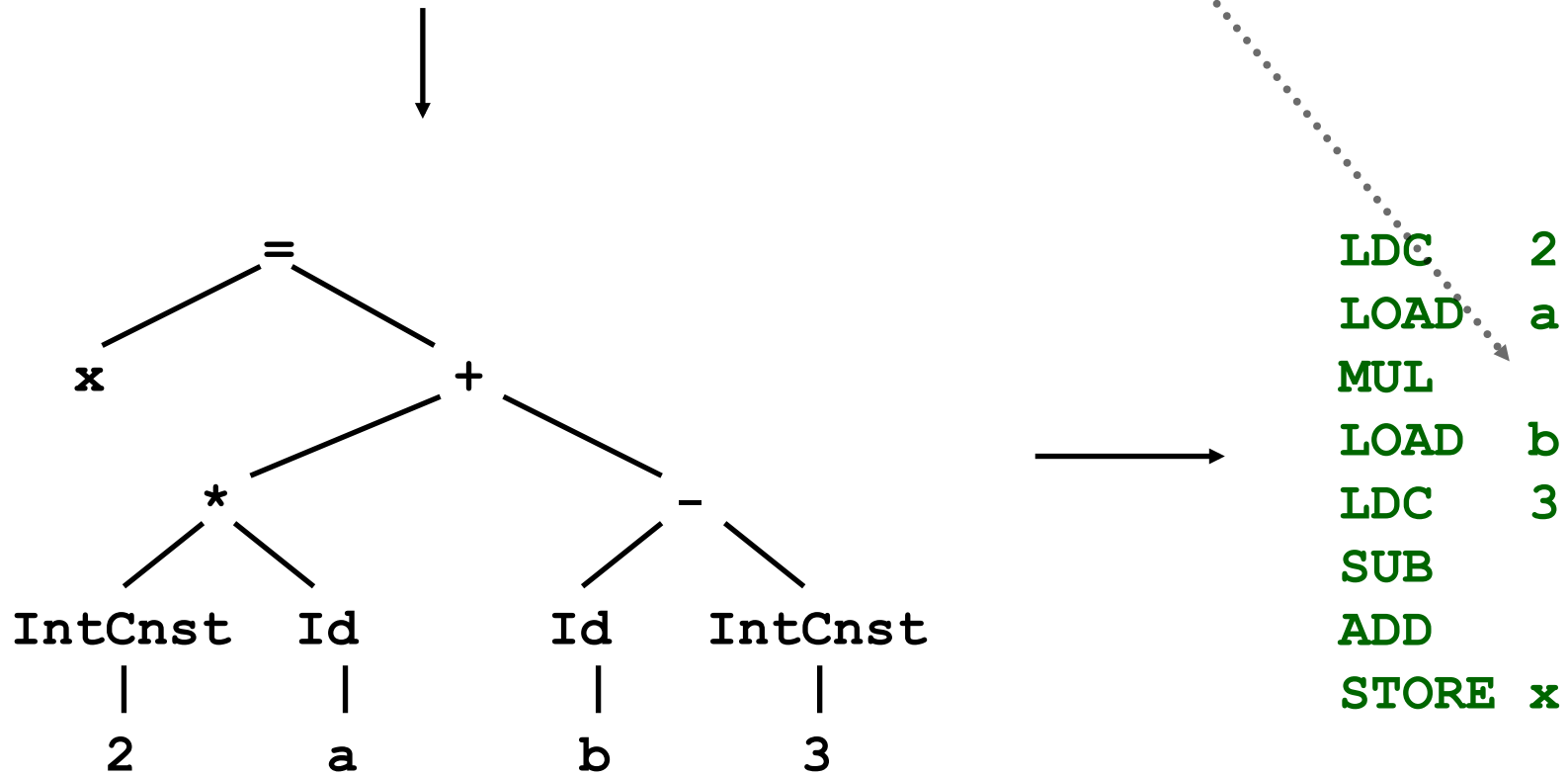
Levels of IR

- High-level Intermediate Language
 - E.g., Abstract Syntax Tree
- Medium-level Intermediate Language
 - Make explicit:
 - Source variables, temporaries, registers
 - Support for Block structure, procedures
 - Simple conditional and unconditional branches
- Low-level Intermediate Language
 - Almost one-to-one to target
 - Input for final instruction selection phase or postpass optimizer

Null-Address-Code (Zero-address code)

- abstrakter Code für eine Stackmaschine:

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



Übersetzung von Ausdrücken

- Übersetzung 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.
```

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)
```

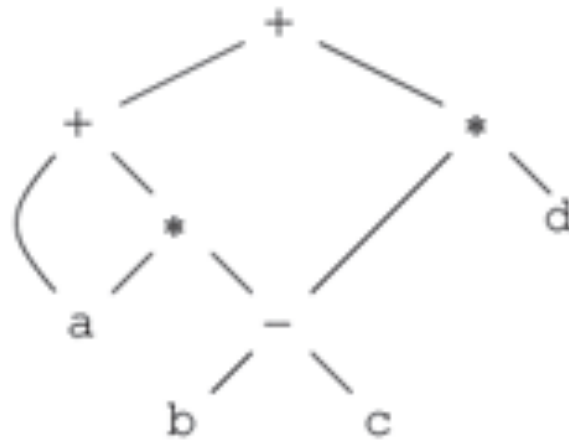
Result referred to by position

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

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

b Drei-Adress-Code

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

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

Darstellungsweisen I

- **Quadrupel**
- Tripel
- indirekte Tripel

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

a Drei-Adress-Code

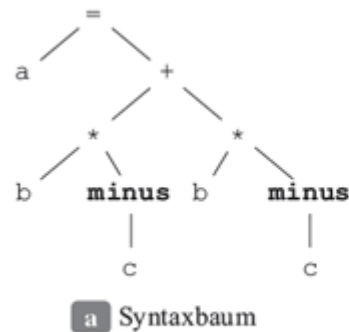
	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
0	minus	c		t ₁
1	*	b	t ₁	t ₂
2	minus	c		t ₃
3	*	b	t ₃	t ₄
4	+	t ₂	t ₄	t ₅
5	=	t ₅		a
		...		

b Quadrupel

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

Darstellungsweisen II

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



	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
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 $a = b * -c + b * -c$

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

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>
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

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

a

Drei-Adress-Code

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$

b

Statische Einzelzuweisungsform

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

Generierung des Zwischencodes

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

Produktion	Semantische Regeln
$S \rightarrow id = E ;$	$S.code = E.code \parallel$ $gen(top.get(id:lexeme) '=' E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = newTemp()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr '=' E_1.addr '+' E_2.addr)$
$ -E_1$	$E.addr = newTemp()$ $E.code = E_1.code \parallel$ $gen(E.addr '=' 'minus' E_1.addr)$
$ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$ id$	$E.addr = top.get(id:lexeme)$ $E.code = ''$

$a = b + - c$

$t1 = \text{minus } c$
 $t2 = b + t1$
 $a = t2$

top: Symboltabelle

Generierung des Zwischencodes II

- 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); }
    
```

Abbildung 6.22: Semantische Aktionen für Arrayreferenzen

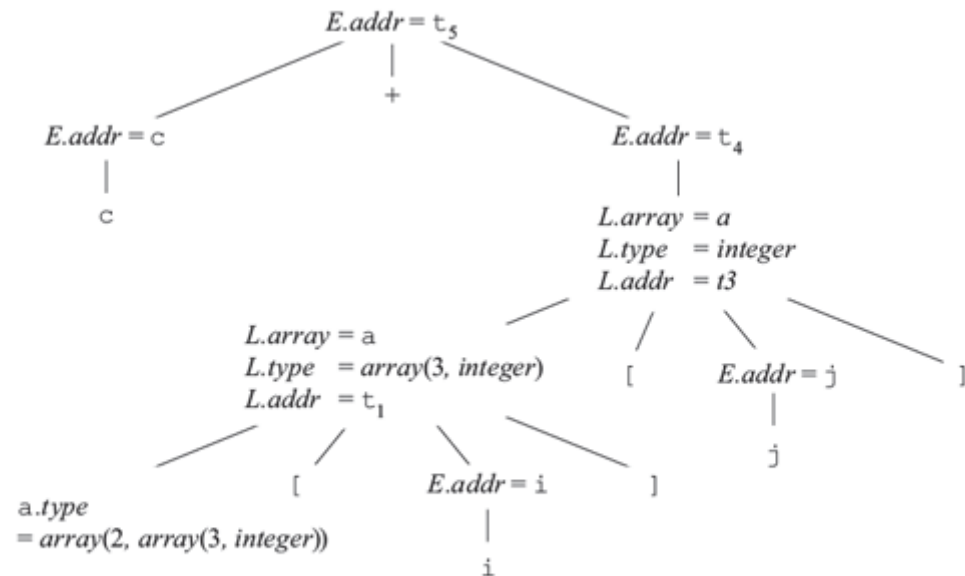


Abbildung 6.23: Annotierter Parse-Baum für $c + a[i][j]$

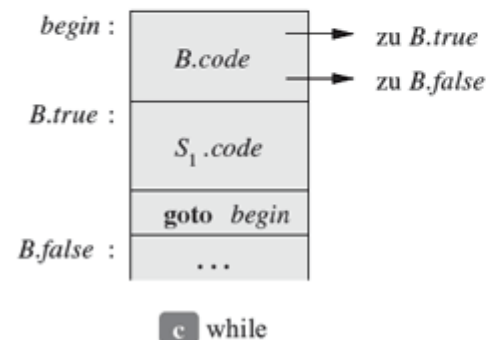
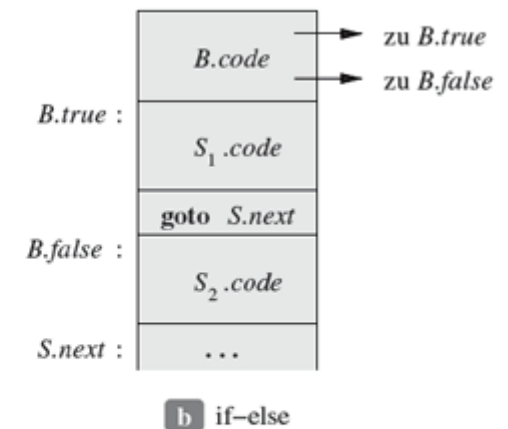
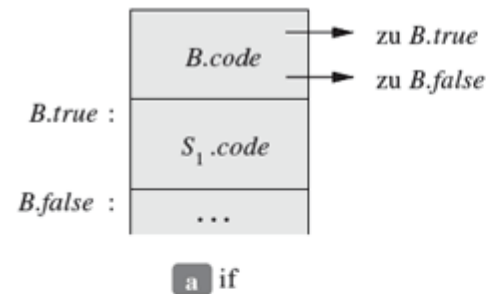
Generierung des Zwischencodes

- Für Kontrollflussanweisungen: true und false Labels

$S \rightarrow \text{if} (B) S_1$

$S \rightarrow \text{if} (B) S_1 \text{ else } S_2$

$S \rightarrow \text{while} (B) S_1$



Mehr Details in
Kapitel 6

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

Generierung des Zwischencodes

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

bildung 6.36: Syntaxgerichtete Definition für Kontrollflussanweisungen

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

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

Mehr Details in Kapitel 6

Typen I

- Typüberprüfung

- Typsynthese:

- Typ eines Ausdrucks anhand der Teilausdrücke abgeleitet
 - Falls f den Typ $s \rightarrow t$ hat und x den Typ s dann hat $f(x)$ den Typ t

- Typinferenz

- Type wird anhand der Verwendungsweise festgestellt
 - $f(x)$ kann vom Typ β sein falls f vom Typ $\alpha \rightarrow \beta$ sein kann und x vom Typ α sein kann
 - basierend auf Unifikation und Typvariablen (α, β, \dots)

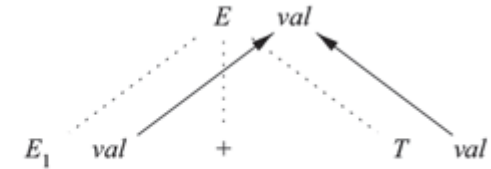
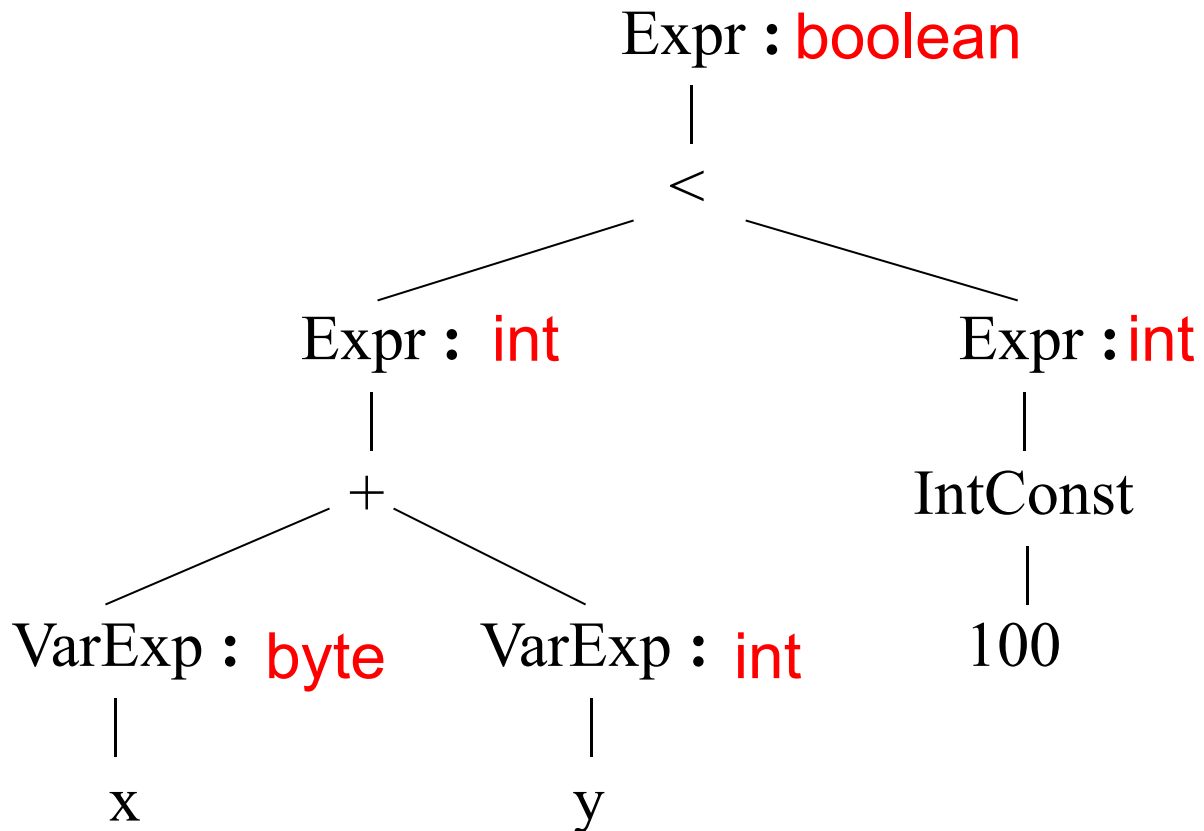


Abbildung 5.6: $E.val$ wird aus $E_1.val$ und $E_2.val$ synthetisiert

Typsynthese

- Bottom-up Analyse der Ausdrücke



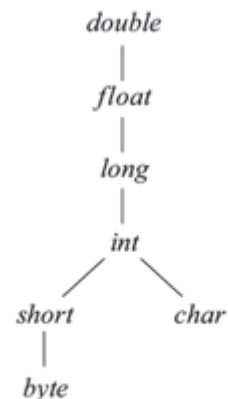
Funktionsaufrufe

```
boolean greater (int x, int y) {  
    return (x > y);  
}  
  
.  
.  
  
    if (greater(a, b)) { ... }
```

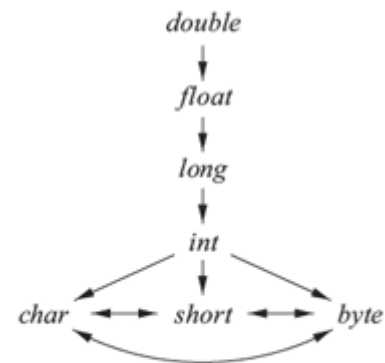
- Typ von `greater` ist **`int × int → boolean`**
- Um Anwendung zu überprüfen:
 - Typ von `greater` in Symboltabelle nachschauen
 - Typen der Parameter überprüfen
 - Ergebnistyp der Funktion (**`boolean`**) wird Ergebnistyp des Aufrufs

Typen II

- Typkonvertierung
 - Implizit (coercion) oder Explizit
 - Erweiterend oder Einengend



a Erweiternde Konvertierung



b Einengende Konvertierung

Abbildung 6.25: Konvertierungen zwischen primitiven Typen in Java

Code Generierung

Zwischencode → Zielcode

Drei-Adress RISC Maschine

- LD dst, addr (LD r,x oder LD r1,r2)
- ST x,r
- OP dst, src1, src2 (SUB r1,r2,r3: $r1=r2-r3$)
- BR lbl
- Bcond r, L
- Adressen:
 - x : Variablennamen
 - a(reg): a Variablenname
 - cst(reg), *r, *cst(r)
 - #cst (LD r1, #100)

dies ist die
Zielsprache

Codeerzeugung: Beispiele

- $x=y-z$

```
LD R1, y      // R1 = y
LD R2, z      // R2 = z
SUB R1, R1, R2 // R1 = R1 - R2
ST x, R1      // x = R1
```

- $b=a[i]$

```
LD R1, i      // R1 = i
MUL R1, R1, 8  // R1 = R1 * 8
LD R2, a(R1)  // R2 = contents(a + contents(R1))
ST b, R2      // b = R2
```

Codeerzeugung: Beispiele

- $a[i]=c$

- if $x < y$ goto L

```
LD R1, x      // R1 = x
LD R2, y      // R2 = y
SUB R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M    // if R1 < 0 springe zu M
```

Grundblöcke (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

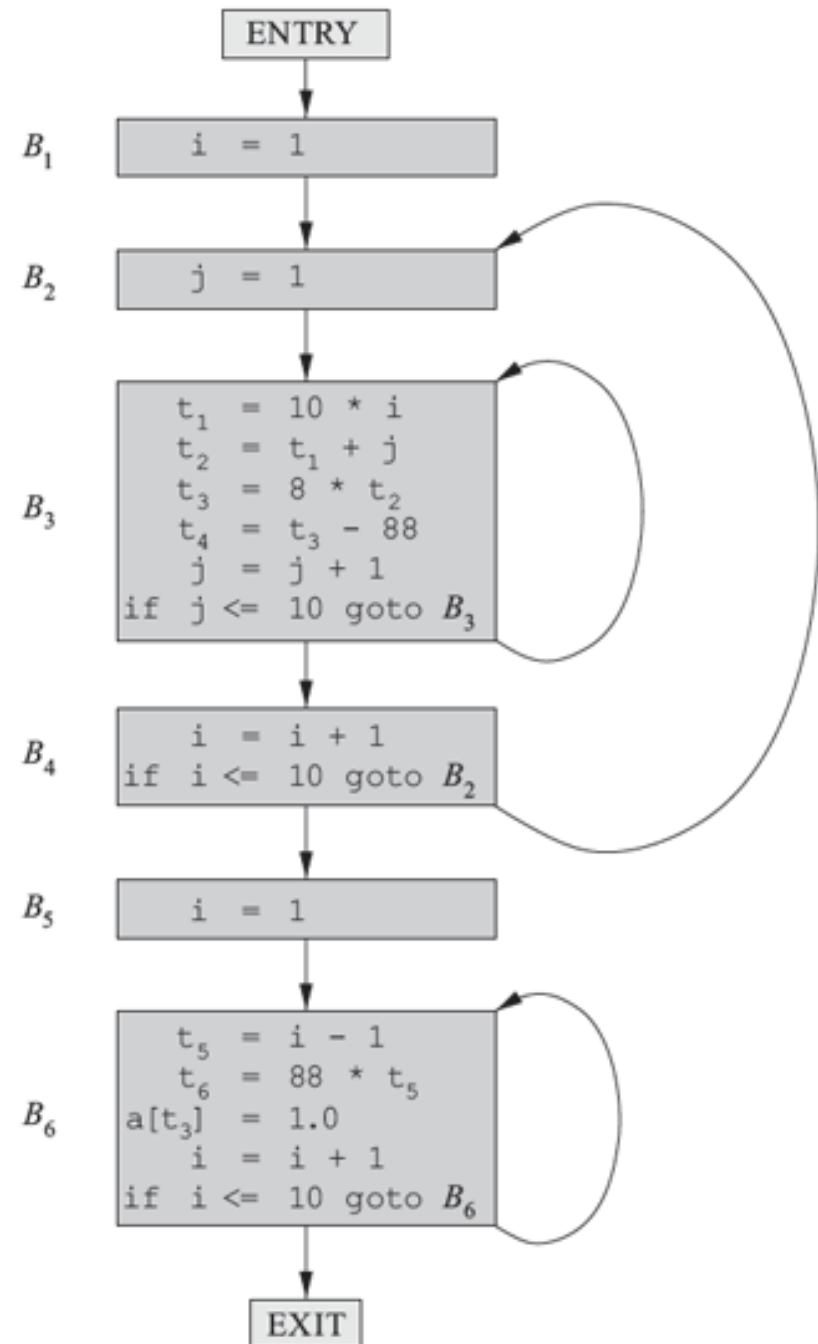
METHODE: Zuerst ermitteln wir die Befehle im Zwischencode, die als *Anführer* fungieren, also die ersten Befehle in einem Grundblock. Die Anführer werden anhand folgender Regeln bestimmt:

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

```
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)
```



Optimierungen von Grundblöcken

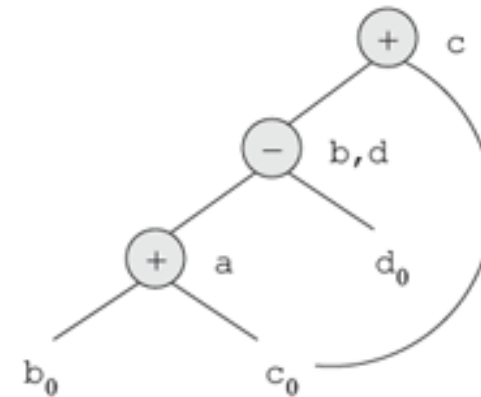
- DAG-Darstellung (8.5.1 ff); lokale gemeinsame Ausdrücke

$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - d$$



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