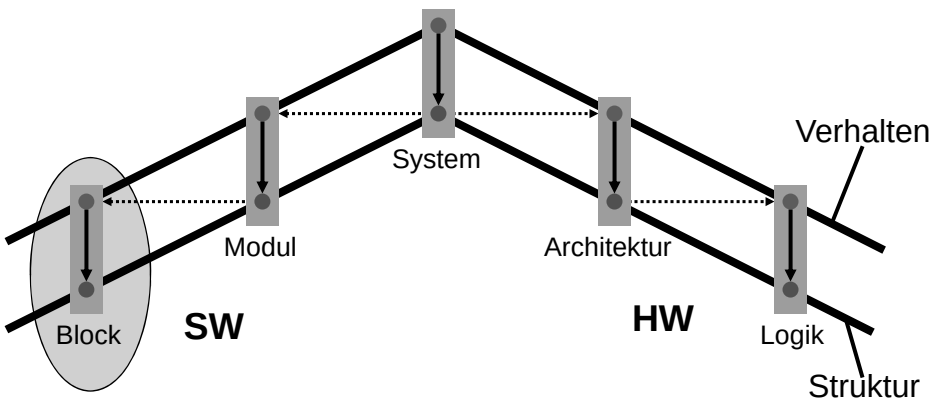
Compiler und Codegenerierung

Hw-Sw-Co-Design

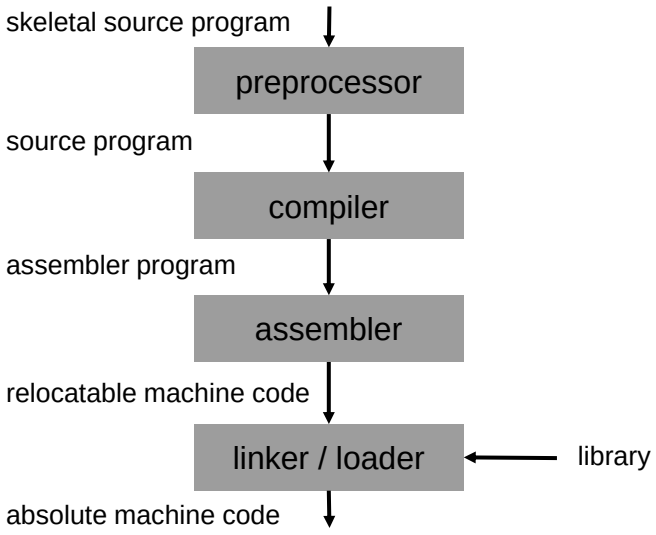
Wo sind wir?



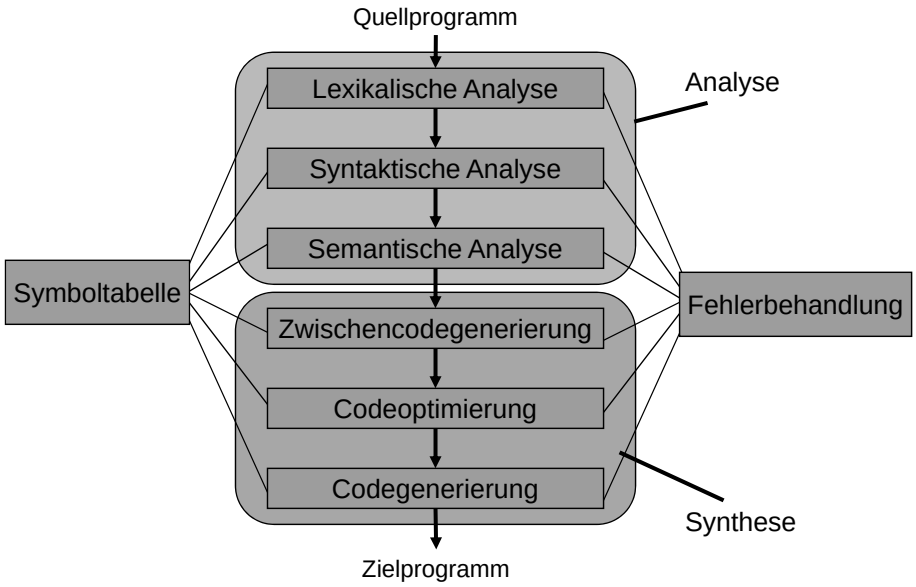
Compiler und Codegenerierung

- Compiler - Aufbau
- Codegenerierung
- Codeoptimierung
- Codegenerierung für Spezialprozessoren
- Retargetable Compiler

Übersetzungsprozess



Phasen eines Compilers



Analyse

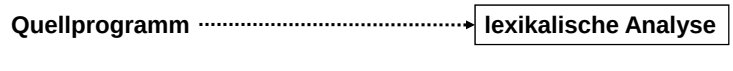
- lexikalische Analyse
  - Quellprogramm scannen und in Symbole zerlegen
  - reguläre Ausdrücke: Erkennung durch endliche Automaten
- syntaktische Analyse
  - Symbolfolgen parsen und daraus Sätze bilden
  - Sätze werden durch eine kontextfreie Grammatik beschrieben

    Z → Bezeichner := A

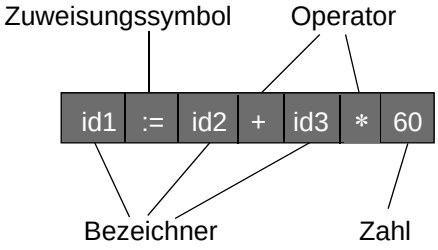
    A → A + A | A * A | Bezeichner | Zahl
- semantische Analyse
  - sicherstellen, dass die Teile sinnvoll zusammenpassen

    Bsp.: Typumwandlungen

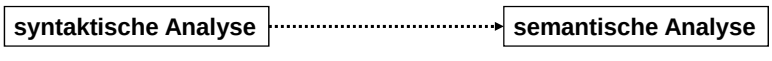
Beispiel (1)

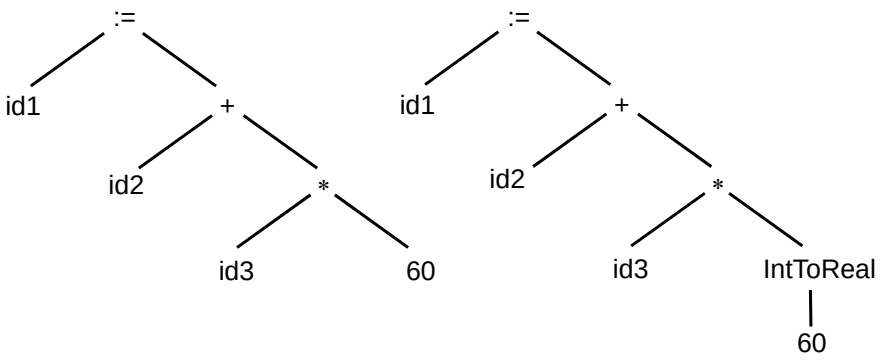


position := initial + rate * 60

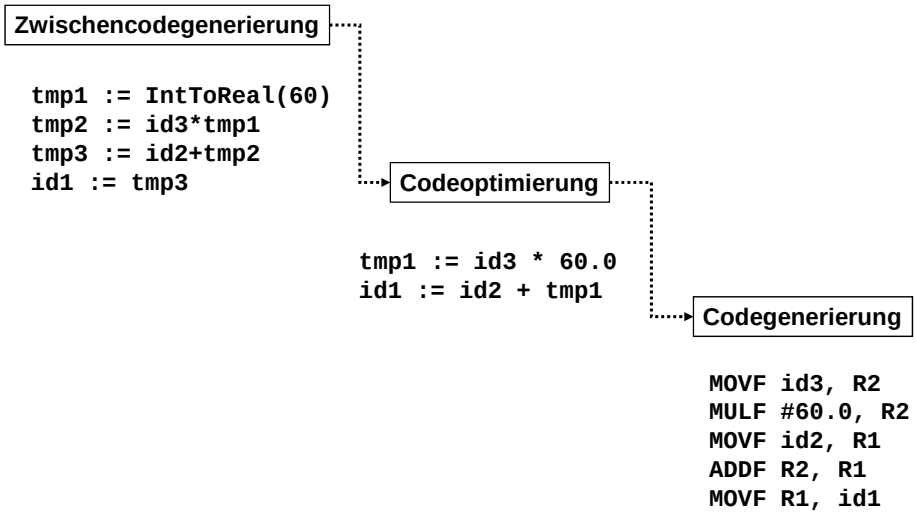


Beispiel (2)





Beispiel (3)



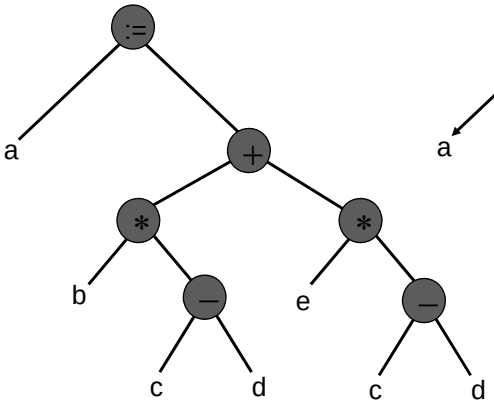
Synthese

- Zwischencode(generierung)
  - maschinenunabhängig → retargeting einfacher
  - soll leicht erzeugbar sein
  - soll leicht ins Zielprogramm übersetzbar sein
- Optimierung
  - GP-Prozessoren: schneller Code, schnelle Übersetzung
  - Spezialprozessoren: schneller Code, kurzer Code, wenig Speicherbedarf
  - Zwischencode und Zielcode optimierbar
- Codegenerierung

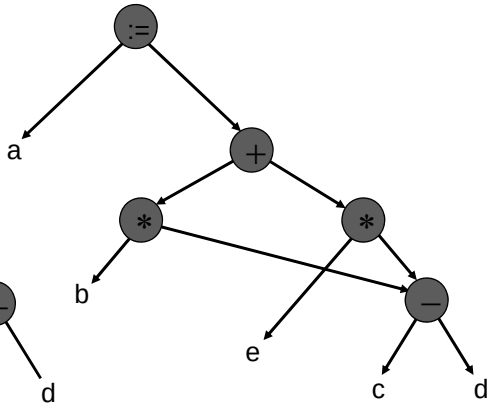
Syntaxbaum und DAG

a := b*(c-d) + e*(c-d)

Syntaxbaum



DAG (directed acyclic graph)



3-Adress-Code (1)

- 3-Adress-Befehle
  - maximal 3 Adressen (2 Operanden, 1 Resultat)
  - maximal 2 Operatoren

Zuweisungen

```
x := y op z
x := op y
x := y

x := y[i]
x[i] := y

x := &y
y := *x
*x := y
```

Kontrollflussbefehle

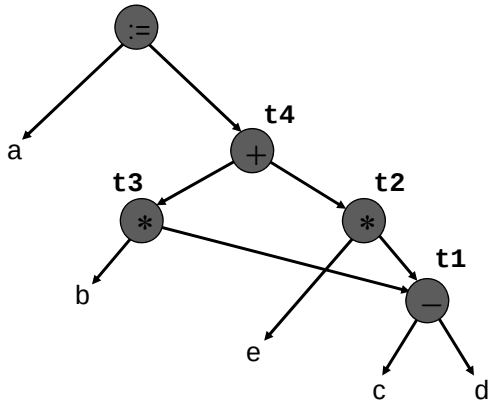
```
goto L
if x relop y goto L
```

Unterprogramme

```
param x
call p,n
return y
```

3-Adress-Code (2)

- Generierung von 3-Adress-Code




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

3-Adress-Code (3)

- Vorteile
  - Auflösung langer arithmetischer Ausdrücke und geschachtelter Schleifen
  - temporäre Namen erlauben einfache Umordnung von Befehlen
  - bereits gültiger Ablaufplan

- Definition: Ein 3-Adress-Befehl

  x := y op z

  definiert **x** und
  verwendet **y** und **z**

Grundblöcke (1)

- Definition: Ein Grundblock (basic block) ist eine Folge von fortlaufenden Anweisungen, in die der Kontrollfluss am Anfang eintritt, und die er am Ende verlässt ohne dass er dazwischen anhält oder - außer am Ende - verzweigt.

```
t1 := c - d
t2 := e * t1
t3 := b * t1
t4 := t2 + t3
if t4 < 10 goto L
```

Grundblöcke (2)

- Sequenz von 3-Adress-Befehlen → Menge von Grundblöcken:

1. Bestimmung der Blockanfänge

   Blockanfänge sind

   - **der erste Befehl**
   - **Ziele von bedingten oder unbedingten Sprüngen**
   - **Befehle, die direkt auf bedingte oder unbedingte Sprünge folgen**

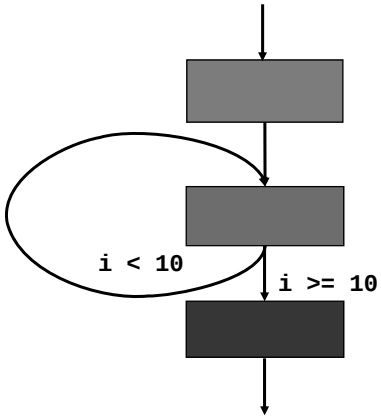
2. Bestimmung der Grundblöcke

   - **zu jedem Blockanfang gehört ein Grundblock**
   - **der Grundblock besteht aus dem Anfang selbst und geht bis zum (exklusive) nächsten Blockanfang oder bis zum Programmende**

Kontrollflussgraphen

- entarteter Kontrollflussgraph (CFG)
  - die Knoten stellen ganze Grundblöcke dar

```
      i := 0
      t2 := 0
  L   t2 := t2 + i
      i := i + 1
      if i < 10 goto L
      x := t2
```



DAG eines Grundblocks

- Definition: Ein DAG eines Grundblocks ist ein gerichteter azyklischer Graph mit folgender Knotenmarkierung:
  - Blätter werden mit einem Variablen/Konstantennamen markiert. Variablen mit Initialwerten bekommen den Index 0.
  - Innere Knoten werden mit einem Operatorsymbol markiert. Aus dem Operator kann man bestimmen, ob der Wert oder die Adresse der Variablen verwendet wird.
  - Optional kann ein Knoten mit einer Sequenz von Variablennamen markiert werden. Dann wird allen Variablen der berechnete Wert zugewiesen.

Beispiel (1)

C Programm

```
int i, prod, a[20], b[20];
...
prod = 0;
i = 0;
do {
  prod = prod + a[i]*b[i];
  i++;
} while (i < 20);
```

3-Adress Code

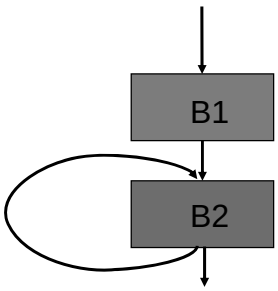
```
(1)  prod := 0
(2)  i := 0
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

Beispiel (2)

Grundblöcke

```
(1)  prod := 0          B1
(2)  i := 0
(3)  t1 := 4 * i        B2
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

Kontrollflussgraph

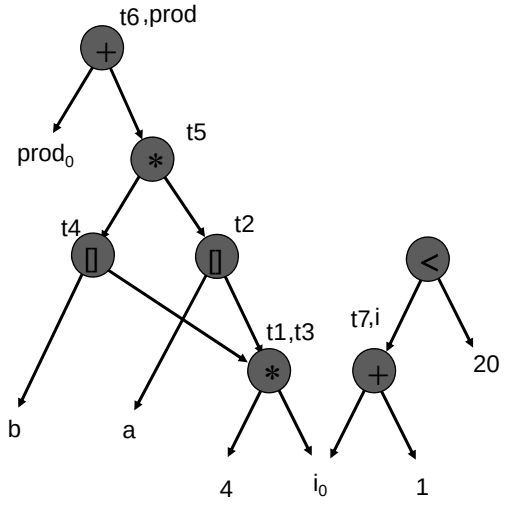


Beispiel (3)

Grundblock B2

```
t1 := 4 * i
t2 := a[t1]
t3 := 4 * i
t4 := b[t3]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i < 20 goto (3)
```

DAG für B2



Compiler und Codegenerierung

- Compiler - Aufbau
- Codegenerierung
- Codeoptimierung
- Codegenerierung für Spezialprozessoren
- Retargetable Compiler

Codegenerierung

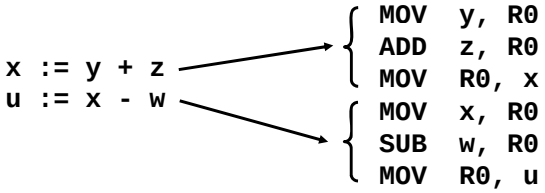
- Anforderungen
  - korrekter Code
  - effizienter Code
  - effiziente Generierung
- Codegenerierung = Softwaresynthese
  - Allokation: meist schon gegeben
  - Bindung:
    - Registervergabe, Registerzuweisung
    - Befehlsauswahl
  - Ablaufplanung
    - Instruktionsreihenfolge

Registerbindung

- Ziel: effiziente Nutzung der Register
  - Befehle mit Registeroperanden sind i.allg. kürzer und schneller ausführbar als Befehle mit Speicheroperanden
- Registervergabe, Registerzuweisung
  - bestimme für jeden Punkt im Programm die Menge der Variablen, die in Registern gehalten werden sollen
  - weise diesen Variablen bestimmte Register zu
  - das Finden einer optimalen Zuweisung ist ein NP-vollständiges Problem
  - zusätzlich gibt es Vorgaben für die Verwendung von Registern durch Prozessorarchitektur, Compiler, Betriebssystem

Befehlsauswahl

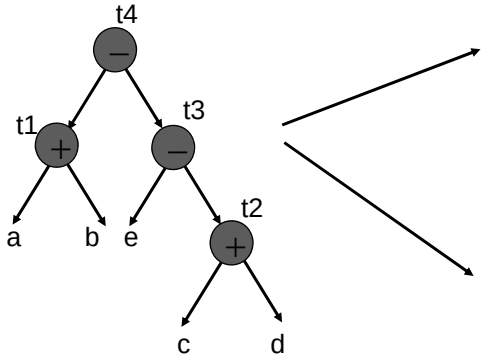
- Codemuster für jeden 3-Adress Befehl



- Probleme
  - oft ineffizienter Code generiert → Codeoptimierung
  - es kann viele passende Zielinstruktionen geben
  - manche Instruktionen nur mit bestimmten Registern möglich
  - Ausnutzung von speziellen Prozessoreigenschaften
    Bsp.: autoinkrement / dekrement Adressierung

Ablaufplanung (1)

- Ziel: effiziente Ausführungsreihenfolge
  - möglichst kurz und möglichst wenige Register




```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

Ablaufplanung (2)

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```



```
MOV  c, R0
ADD  d, R0
MOV  e, R1
SUB  R0, R1
MOV  a, R0
ADD  b, R0
SUB  R1, R0
MOV  R0, t4
```

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```



```
MOV  a, R0
ADD  b, R0
MOV  c, R1
ADD  d, R1
MOV  R0, t1
MOV  e, R0
SUB  R1, R0
MOV  t1, R1
SUB  R0, R1
MOV  R1, t4
```

Maschinenmodell

- *n* Register R0 … Rn-1
- Byte-adressierbar, 4 Byte bilden ein Wort
- Instruktionsformat

```
op source, destination
```

  - Bsp.: MOV R0, a
    ADD R1, R0
  - jede Instruktion hat Kosten von 1
  - Adressen von Speicheroperanden befinden sich in den folgenden Instruktionsworten

Adressierungsarten

mode	form	address	added cost
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + *contents*(R)	1
indirect register	*R	*contents*(R)	0
indirect indexed	*c(R)	*contents*(c+*contents*(R))	1
immediate	#c	c	1

Lebenszeit von Variablen

- Definition: Ein Name in einem Grundblock ist aktiv an einem bestimmten Punkt, wenn sein Wert nach diesem Punkt noch verwendet wird (möglicherweise in einem anderen Grundblock).
- Bestimmung der Lebenszeit
  1. gehe zum Ende des Grundblocks und notiere, welche Namen am Ausgang aktiv sein müssen
  2. gehe rückwärts zum Blockanfang; für jeden Befehl `(I)  x := y op z`
     - binde die Informationen über `x, y, z` an `(I)`
     - setze `x` auf nicht-aktiv und keine-nächste-Verwendung
     - setze `y, z` auf aktiv und nächste-Verwendung auf `(I)`

Einfacher Codegenerator (1)

- aktuelle Stelle einer Variablen **t**
  - Register oder Speicher; wenn **t** sowohl in einem Register als auch im Speicher steht, bevorzuge das Register
- für jeden 3-Adress-Befehl **x := y op z**
  1. bestimme durch *getreg(***x***)* die Stelle **L**, wo das Ergebnis **x** abgelegt werden soll
  2. bestimme **y'**, die aktuelle Stelle von **y**; falls **y'** nicht **L** ist, generiere **MOV y', L**
  3. bestimme **z'**, die aktuelle Stelle von **z**, und generiere **op z', L**

Einfacher Codegenerator (2)

- **L** = *getreg(***x***)* für den Befehl **x := y op z**
  1. wenn **y** in einem Register **R** ist, das sonst keine Variablen hält und **y** keine nächste Verwendung hat: return(**R**)
  2. wenn ein leeres Register **R** existiert: return(**R**)
  3. wenn **x** eine nächste Verwendung hat oder **op** ein Operator ist, der ein Register benötigt (z.Bsp. indirekte Adressierung), finde ein belegtes Register **R,** speichere das Register (**MOV R, M)** und return(**R**)
  4. wenn **x** nicht mehr im Grundblock benutzt wird oder kein passendes Register gefunden wurde: return($\mathbf{M_x}$)