

# Compiler: Zwischenrepräsentationen

Prof. Dr. Oliver Braun

Letzte Änderung: 27.06.2017 11:13

# Eine Taxonomie von Zwischenrepräsentationen

Zwischenrepräsentationen (*Intermediate Representations (IR)*) können strukturell in drei Kategorien eingeteilt werden:

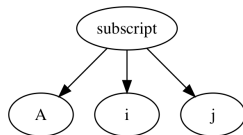
**Graphische IRs** Interne Darstellung als Graph. Die Algorithmen des Compilers werden durch Graphenalgorithmen ausgedrückt.

**Lineare IRs** Pseudocode für eine abstrakte Maschine. Die Algorithmen iterieren über einfache lineare Sequenzen von Operationen.

**Hybride IRs** Kombinieren beide Ansätze, z.B. eine low-Level lineare IR für Code-Blöcke und ein Graph der den Kontrollfluß zwischen den Blöcken ausdrückt.

# Abstraktionsebene

- ▶ gegeben sei der Array-Zugriff auf ein zweidimensionales Array:  
 $A[i, j]$
- ▶ eine baumartige Zwischenrepräsentation nah an der



Quellsprache:

- ▶ linearer Pseudocode nah an der Zielsprache

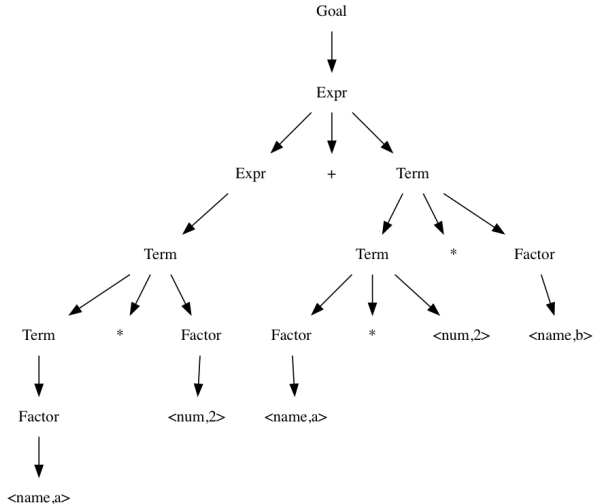
```
subI  ri, 1  => r1
multI r1, 10 => r2
subI  rj, 1  => r3
add   r2, r3 => r4
multI r4, 4  => r5
loadI @A     => r6
add   r5, r6 => r7
load  r7     => rAij
```

# Graphische IRs

# Syntax-basierte Bäume

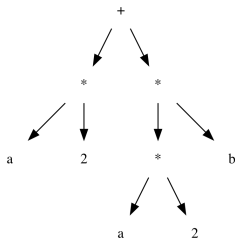
# Parse-Bäume

- ▶ repräsentieren die komplette Herleitung
- ▶ Parsebaum für  $a * 2 + a * 2 * b$



# Abstrakte Syntax-Bäume (*abstract syntax trees (AST)*)

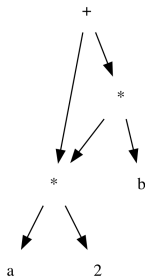
- ▶ bewahren die wesentliche Struktur der Parsebäume
- ▶ aber entfernen die überflüssigen Knoten



- ▶ AST für  $a * 2 + a * 2 * b$
- ▶ ASTs werden in vielen Compiler-Systemen genutzt
  - ▶ Source-To-Source Systems
  - ▶ syntax-gesteuerte Editoren
  - ▶ automatische Parallelisierungs-Tools

# Gerichtete kreisfreie Graphen (*directed acyclic graphs* (DAGs))

- ▶ ein DAG vermeidet doppelte Teilbäume die im AST vorkommen können durch **Sharing**



- ▶ DAG für  $a * 2 + a * 2 * b$
- ▶ durch die kompaktere Darstellung muss  $a * 2$  nur einmal berechnet werden
- ▶ ABER: Das funktioniert nur wenn sich die Werte (hier:  $a$ ) nicht verändern können



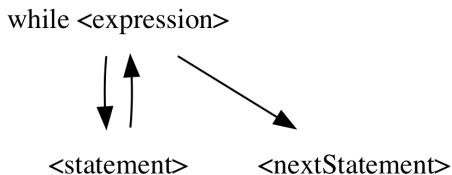
# Graphen

# Kontroll-Fluß-Graphen (*control-flow graphs (CFGs)*)

- ▶ ein CFG modelliert den Kontrollfluß in einem Programm
- ▶ ein CFG ist ein gerichteter Graph  $G = (N, E)$ 
  - ▶ jeder Knoten  $n \in N$  repräsentiert einen Grundblock (sequenzieller Code ohne Verzweigungen)
  - ▶ jede Kante  $e = (n_i, n_j) \in E$  korrespondiert zu einem möglichen Kontrollübergang von  $n_i$  zu  $n_j$

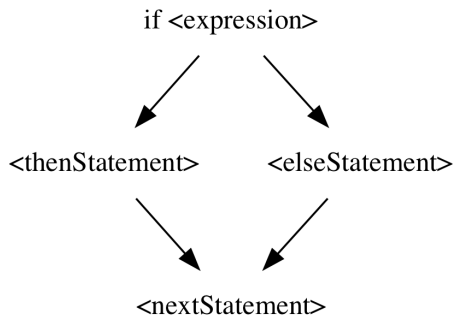
## Beispiel: while-Schleife

```
while (<expression>
    <statement>
<nextStatement>
```



## Beispiel: if-then-else

```
if (<expression>
    then <thenStatement>
    else <elseStatement>
<nextStatement>
```

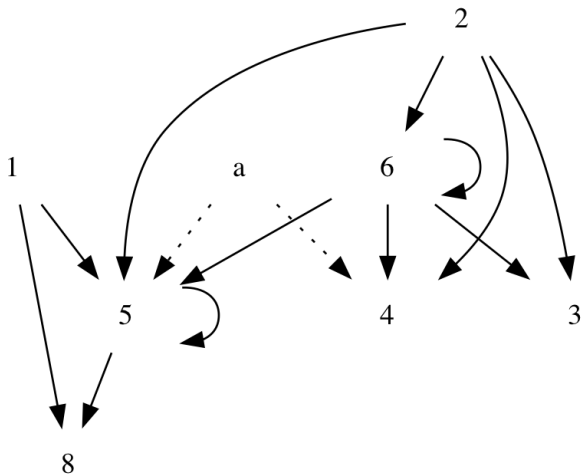


## Abhängigkeitsgraph (*dependence graph*)

- ▶ Abhängigkeiten zwischen Definition und Nutzung eines Wertes
- ▶ wie sieht der data-dependence graph zu folgendem Code aus?

```
1  int x = 0;
2  int i = 1;
3  while (i < 100) {
4      if (a[i] > 0)
5          x += a[i];
6      i++;
7  }
8  System.out.println(x);
```

# Data-Dependence Graph



# Aufruf-Graphen (*call graph*)

- ▶ für interprozedurale Analyse und Optimierung
- ▶ Knoten für jede Prozedur
- ▶ Kante für jeden Prozeduraufruf
- ▶ Probleme
  - ▶ getrennte Compilierung
  - ▶ Funktionen höherer Ordnung
    - ▶ bei jedem Aufruf kann eine andere Funktion übergeben werden
  - ▶ Vererbung

# Lineare IRs



# Lineare IRs

- ▶ i.d.R. Assembler-Code für eine abstrakte Maschine
- ▶ viele frühe Compiler nutzten lineare IRs
- ▶ lineare IRs haben eine implizite Ordnung
  - ▶ Abhängigkeitsgraphen haben eine partielle Ordnung die mehrere Ausführungsreihenfolgen zulässt
- ▶ in einem Compiler muss eine lineare IR einen Mechanismus beinhalten, der den Kontrollfluß beschreibt
  - ▶ z.B. Blöcke und Sprünge

# Stack-Maschine

- ▶ Spezialform von Ein-Adress-Code (Null-Adress-Maschine)
- ▶ Beispiel

```
push 2  
push b  
multiply  
push a  
subtract
```

- ▶ kompakter, einfach zu generierender und auszuführender Code
- ▶ Smalltalk 80 und Java nutzen einen Bytecode der einer Stack-Maschine entspricht

# Drei-Adress-Code

- ▶ Operationen haben in der Regel die Form

`i ← j op k`

- ▶ Beispiel

`t1 ← 2`

`t2 ← b`

`t3 ← t1 * t2`

`t4 ← a`

`t5 ← t4 - t3`

# Speichermodelle

- ▶ der Compiler muss für jeden Wert im Code entscheiden ob er in einem Register oder im Hauptspeicher liegen soll
- ▶ für den ausführbaren Code muss sogar klar sein, ob der Wert z.B. in Register `r13` oder in den ersten 16 Byte des Labels `L0089` steht
- ▶ bis kurz vor der Code-Generierung kann der Compiler natürlich symbolische Adressen verwenden
- ▶ im Wesentlichen gibt es zwei Speichermodelle die Verwendung finden

# Register-to-Register-Modell

- ▶ der Compiler nutzt für alles Register
- ▶ unabhängig von den tatsächlichen, physikalischen Einschränkungen
- ▶ Werte werden nur dann in den Hauptspeicher geschrieben, wenn die Semantik des Programms dies erfordert
  - ▶ z.B. bei einem Prozeduraufruf wenn ein Parameter “by reference” übergeben wird

# Memory-to-Memory-Modell

- ▶ der Compiler geht davon aus, dass alle Werte im Hauptspeicher gehalten werden
- ▶ die Werte müssen vor der Benutzung in Register und nach der Definition in den Hauptspeicher verschoben werden
- ▶ es reicht dann eine sehr kleine Anzahl von Registernamen in der IR
- ▶ oft werden in der IR memory-to-memory-Operationen verwendet

# Auswahl eines Speichermodells

- ▶ die Auswahl des Speichermodells ist meist orthogonal zur Auswahl der IR
- ▶ aber es hat einen Einfluss auf den Rest des Compilers
- ▶ bei einem Register-to-Register-Modell nutzt der Compiler i.d.R. mehr Register als tatsächlich vorhanden sind
  - ▶ der Register-Allokator muss dann eine Menge von virtuellen Registern auf die physikalischen Register abbilden
- ▶ bei einem Memory-to-Memory-Modell werden üblicherweise viel weniger Register genutzt wie in einer modernen Hardware vorhanden sind
  - ▶ der Register-Allokator sucht dann nach im Hauptspeicher gehaltenen Werten, die über einen längeren Zeitraum in ein Register geschrieben werden können

# Symboltabellen

- ▶ während der Übersetzung sammelt ein Compiler verschiedene Informationen, die er an anderer Stelle wieder benötigt, z.B.
  - ▶ Variablen mit Datentyp, Speicherklasse und Name
  - ▶ Arrays mit Dimensionen, untere und obere Grenze für jede Dimension
- ▶ der Compiler kann diese Informationen wenn er Sie benötigt immer wieder berechnen
- ▶ oder sie beim ersten Auftreten speichern
  - ▶ in der IR aufwändig wieder zu finden
  - ▶ extra in einer Symboltabelle



# Verschachtelte Gültigkeitsbereiche

- ▶ viele Programmiersprachen haben die Möglichkeit Variablen lokal in einem Gültigkeitsbereich (*scope*) zu definieren
- ▶ bei verschachtelten Gültigkeitsbereichen (*nested scopes*) könnte die Variable auf die zugegriffen wird in jedem der umschließenden Gültigkeitsbereiche definiert worden sein
- ▶ für jeden lexikalischen Scope eine eigene Symboltabelle
- ▶ bei einem Lookup muss *von innen nach außen* gesucht werden

# Verlinkte Tabellen für Namensauflösung in OO

- ▶ in einer objektorientierten Sprache gibt es neben den lexikalischen Gültigkeitsbereichen eine Vererbungshierarchie in der gesucht werden muss
- ▶ eine einfache Implementierung hat eine Symboltabelle für jede Klasse mit zwei verschachtelten Hierarchien
  - ▶ eine für lexikalisches Scoping innerhalb von Methoden und
  - ▶ die andere die der Vererbungshierarchie für jede Klasse folgt

# Namensauflösung in OO

- ▶ um einen Namen `foo` aufzulösen
  - ▶ wird erst in der lexikalischen Tabelle gesucht
  - ▶ dann in der Klasse und ihren Oberklassen
  - ▶ und schließlich in der globalen Tabelle