Karlsruher Institut für Technologie

Praktikum Compilerbau
Sitzung 9 – Codeerzeugung

Prof. Dr.-Ing. Gregor Snelting
Matthias Braun

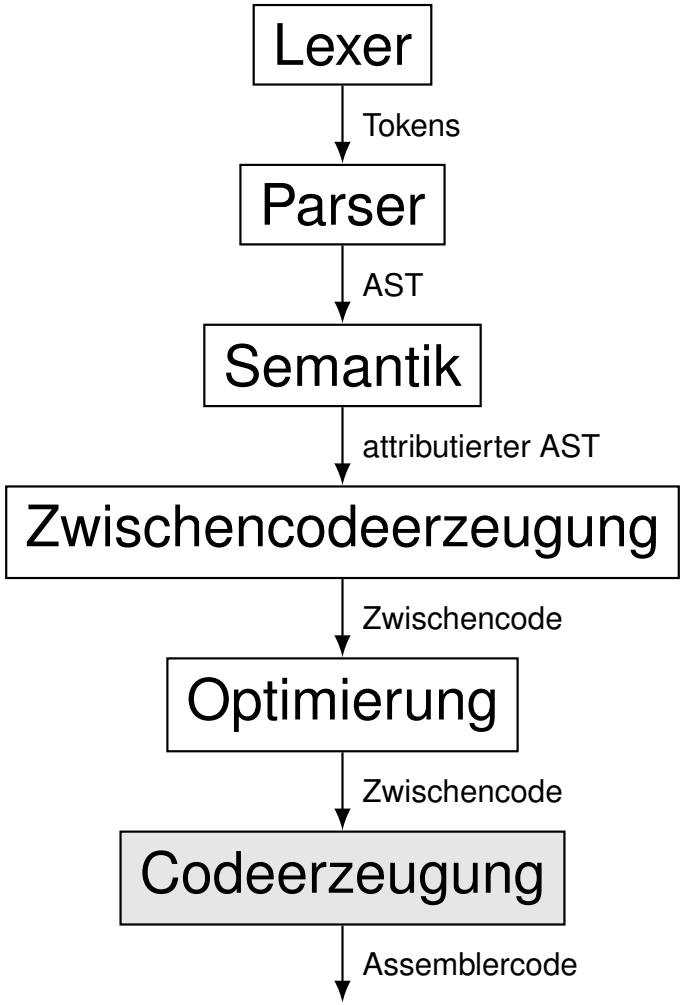
IPD Snelting, Lehrstuhl für Programmierparadigmen

1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Letzte Woche

- Was waren die Probleme?
- Hat soweit alles geklappt?

Compilerphasen



Aufbau eines Compilerbackends – Befehlsauswahl

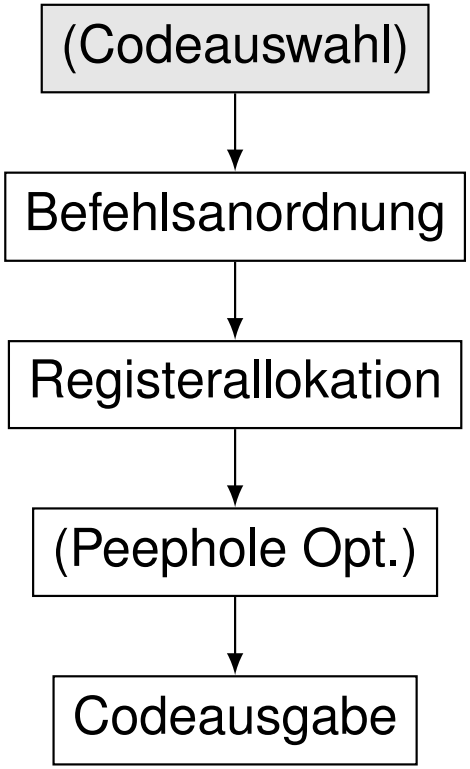
Allgemein:

- Abbilden von Zwischensprache auf Befehle der Zielmaschine. Meist $n : 1$ Abbildung Zwischensprach-:Zielbefehlen, $n : n$ selten.

Bei uns:

- Fast eine $1 : 1$ Abbildung, deshalb keine separate Codeauswahlphase! (Wo gibt es Ausnahmen zu $1 : 1$?)

Compiler-Backend:



Aufbau eines Compilerbackends – Befehlsanordnung

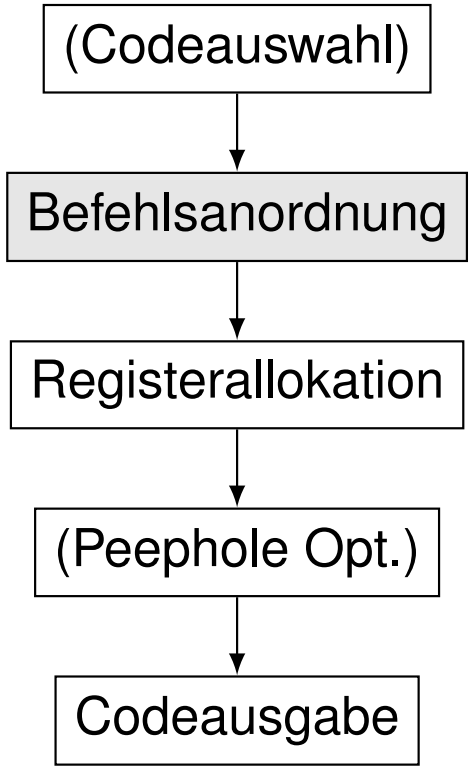
Allgemein:

- Bestimme Befehlsabhängigkeiten.
- Ordne Befehle neu an.
- Optimierungsziel z.B.: Minimaler Ressourcenbedarf (Register), parallelisierung (pipelining, superskalare CPUs).

Bei uns:

- Abhängigkeiten schon gegeben
- Ziel: Beliebige legale Anordnung
- Grundblockanordnung beliebig.

Compiler-Backend:



Aufbau eines Compilerbackends – Registerallokation

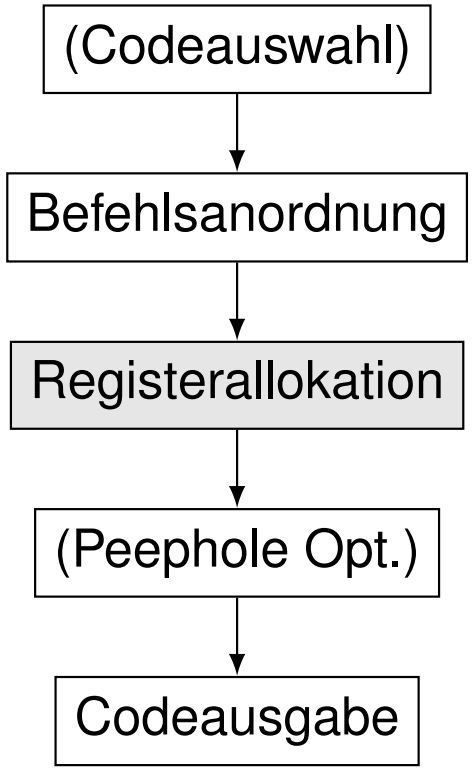
Allgemein:

- Beschränkte Ressourcen: Register, Stackframe, etc. zuteilen.
- Bei Mangel Auslagerungscode.

Bei uns:

- Zuteilung Werte zu Activation Record.
- Register nur sehr lokal nutzen (keine klassische Allokation)

Compiler-Backend:



Aufbau eines Compilerbackends – Peephole Optimierungen

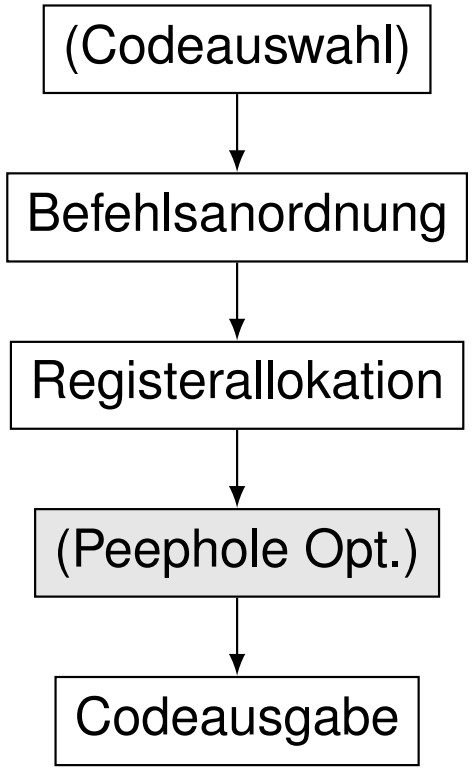
Allgemein:

- Wegen Phasenaufteilung im Backend oft Randfälle mit schlechtem Code.
- Ersetze bekannte Muster mit besserem Code.

Bei uns:

- Freiwillig.

Compiler-Backend:



Aufbau eines Compilerbackends – Codeausgabe

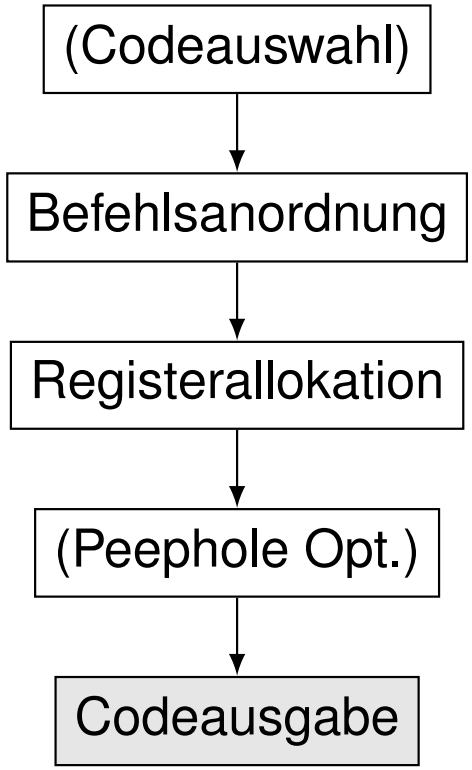
Allgemein:

- Ausgaben von Assembler/Maschinencode.
- Falls nötig Auflösen von Sprungmarken und Referenzen.

Bei uns:

- Ausgabe von x86 assembler (AT&T für GNU Binutils)

Compiler-Backend:



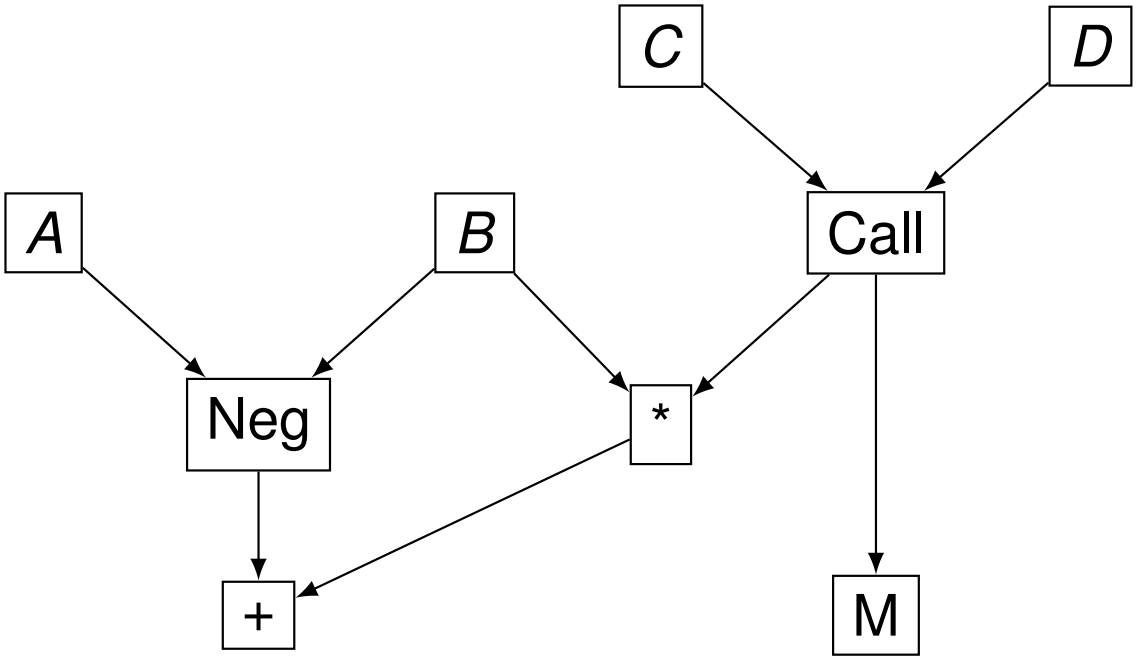
1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Befehlsreihenfolge in einem Grundblock

- Abhängigkeiten in Graph vorhanden
- Abhängigkeiten ergeben Halbordnung der Befehle
- Bilden einer Totalordnung nötig (*Topologisches Sortieren*)
- Die einfachste Möglichkeit für DAGs: Reverse Postorder (für alle Wurzeln).

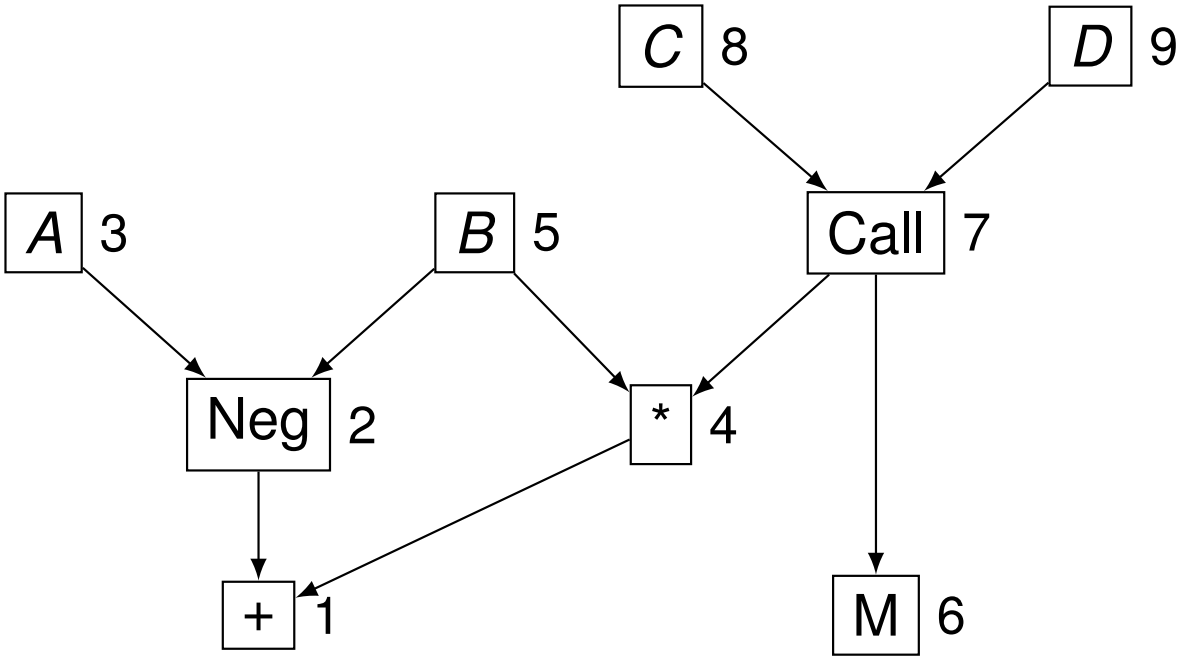
Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



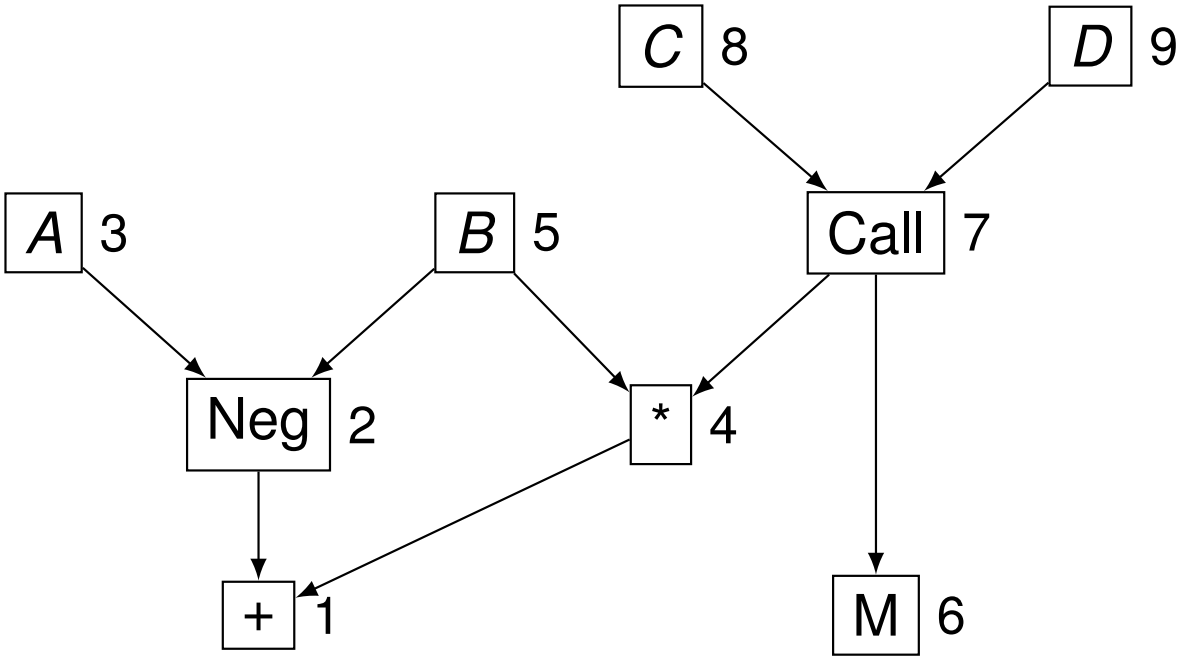
Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



Reverse Postorder

- Für jede Wurzel Tiefensuche auf dem Graph, Nummern beim Verlassen von Knoten vergeben.



Reverse: *D*, *C*, Call, M, *B*, ∗, *A*, Neg, +

Reverse Postorder

- Problem: Firm-Graphen sind keine DAGs. Aber:
  - Schleifen enthalten eine Φ-Operation (Datenschleifen) oder einen Grundblock (Steuerflussschleifen)
  - Φ- und Block-Eingänge nicht relevant beim Anordnen innerhalb eines Grundblocks.
- In jFirm: `Graph.walkTopological()`

Ausnahmen

- Sprungbefehle ans Ende anordnen.
- Proj-Knoten nicht anordnen.
- Phi-Knoten erzeugen Code am Ende der Vorgängerblöcke (siehe spätere Folie).

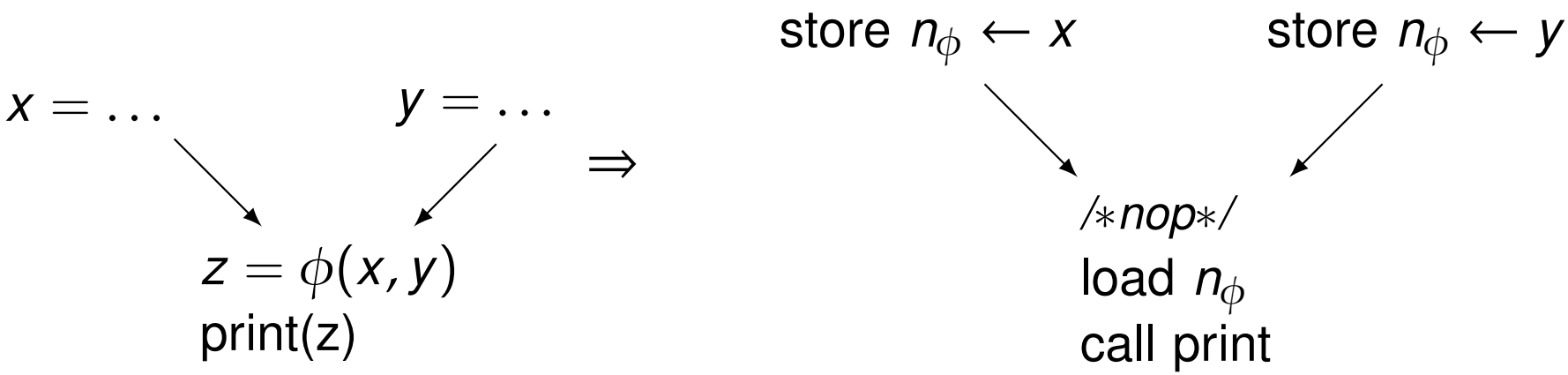
1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Ressourcenverteilung

- Mehr Zwischenergebnisse berechnet als Register.
- $\Rightarrow$ Auslagern in Activation-Record.
- Ansatz:
  - Eine Variable im Activation-Record pro Firm-Knoten.
  - Vor Operation Operanden in Register laden
  - Nach Operation Variable schreiben.

ϕ-Knoten Behandlung

- Jeder ϕ-Knoten hat Variable im Activation Record.
- Am Ende eines Vorgängerblocks zu einem ϕ-Knotens speichere Argumente in die entsprechende Variable.



„Swap“-Problem

Achtung: Semantik von ϕ-Funktionen erfordert gleichzeitige Auswertung. Klassisches Beispiel:

x_0 = ...
y_0 = ...
while (**true**) {
  x_1 = phi(x_0, y_1)
  y_1 = phi(y_0, x_1)
  print(x_1, y_1)
}

Mögliches Vorgehen: Erst ϕ-Operanden besorgen, dann Zielvariablen schreiben.

„Swap“-Problem

Achtung: Semantik von ϕ-Funktionen erfordert gleichzeitige Auswertung. Klassisches Beispiel:

x_0 = ...
y_0 = ...
while (**true**) {
  x_1 = phi(x_0, y_1)
  y_1 = phi(y_0, x_1)
  print(x_1, y_1)
}

Mögliches Vorgehen: Erst ϕ-Operanden besorgen, dann Zielvariablen schreiben.

Verfeinerungen (freiwillig)

Verbesserung: Falls Registerverbrauch 1, Werte direkt erzeugen statt zu laden und zu speichern.

- Bei Konstanten
- Bei unären Operationen

Registerallokation

- Zum Beispiel für baumförmige (nur ein Benutzer am Knoten) Teilgraphen anwenden.

1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Vorgehen

- Codeerzeugung pro Funktion

  for (Graph graph : Program.getGraphs()) { /*...*/ }

- Phase 1 – Vorbereitung:
  - Befehlslisten pro Grundblock erzeugen
  - Knoten auf Offsets im Activation Record zuweisen
- Phase 2 – Assembler Ausgabe

Tipp

- Unterscheide Operationen die (nur) Datenwerte erzeugen von Operationen mit Seiteneffekten.
- Unterscheide Abrufen eines Datenwertes (`getValue`) von dessen Erzeugung (`createValue`).

Schema getValue

```
/**
 * Wert fuer Knoten in Register legen
 */
private void getValue(Node node, String destRegister) {
   if (variableAssigned(node)) {
      int offset = getVariableOffset(node);
      printf("\tmovl %d(%%ebp), %s # reload for %s\n",
               offset, destRegister, node);
      return;
   }
   createValue(node, destRegister);
}
```

Schema createValue

```
/**
 * Wert fuer Knoten berechnen und in Register legen.
 */
private void createValue(Node node, String destRegister) {
   switch (node.getOpCode()) {
   case iro_Const: /* ... */ break;
   case iro_Add:
      Add add = (Add) node;
      getValue(add.getLeft(), destRegister);
      // Achtung: bug falls destRegister==ebx!
      getValue(add.getRight(), "%ebx");
      printf("\taddl %%ebx, %s\n", destRegister);
      break;
   }
}
```

Schema Grundblockausgabe

Für jeden Grundblock:

- `.LXX:` ausgeben
- Befehlsliste durchgehen und Code erzeugen.
- Schreibe dabei Werte in Activation Record:

```
if (variableAssigned(node)) {
   createValue(node, "%eax");
   int offset = getVariableOffset(node);
   printf("\tmovl %s, %d(%%ebp) # spill for %s\n",
          destRegister, offset, node);
   return;
}
```

1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Assembler

Nutze Assembler statt direkter Codeausgabe:

- Unterstützung von Sprungmarken.
- Erzeugt Objekt-Dateiformate (ELF, Mach-O, COFF)
- Menschenlesbar: Erleichtert Debugging

Assemblieren + Linken mit libc in der Praxis:

```
gcc my_file.s -o output
```

Segmente

Einteilung der Daten in Segmente:

- Code-Segment (`.text`), nur lesbar, shared
- Daten-Segment (`.data`), les/schreibbar
- Zero-Segment (`.bss`), wie data aber Null-Initialisiert
- Read-Only-Daten-Segment (`.rodata`), nur lesbar, shared
- weitere Segmente für Dinge wie Thread-Local-Storage, globale Konstruktoren, C++ template Code, . . . (Unterschiedlich je nach Loader/Linker)

Segment direktiven (`.text, .data, ...`) leiten Ausgabe in entsprechendes Segment um

Assembler Labels

Benennung von Daten- und Codeteilen:

- Exportieren und Importieren von Funktionen, globalen Variablen
- Anzeige der Namen in Debugger oder Crash-Handlern.

Einfachste Form:
labelName:

Interne Labels (z.B. Beginn von Grundblöcken) mit .L-Prefix (ELF) bzw. L-Prefix (Mach-O) versehen.

Exportieren von Labels für Linker mit .globl-Direktive. Manche Loader (Mach-O, COFF) erwarten Unterstriche als Präfix für Funktionsnamen.

Funktion in ELF

Executable and **L**inkable **F**ormat
Weite Verbreitung auf Unix-Systemen.

```
  .text
  .p2align 4,,15
.globl MyFunction
  .type MyFunction, @function
MyFunction:
  # ...
  .size MyFunction, .-MyFunction
```

Programmstart in Funktion `main`

Funktion in Mach-O

Mach Objekt File Format
Apple Darwin (basierend auf Mach Microkernel)

```
  .text
  .p2align 4,0x90,15
.globl _MyFunction
_MyFunction:
  # ...
```

Programmstart in Funktion `_main`

Tips/Hilfsmittel

- `objdump` zeigt Inhalt von ELF-Dateien an
- `otool` zeigt Inhalt von Mach-O-Dateien an
- Mit `gcc -g3` assemblieren, dann mit `gdb` debuggen

Wiederholung x86

Prolog:

```
pushl %ebp
movl %esp, %ebp
# allocate XX bytes for activation record
subl $XX, %esp
```

Epilog:

```
# return value is in %eax by now
# copy basepointer to stackpointer (free stack)
movl %ebp, %esp
# restore previous base pointer
popl %ebp
# jump to return address (and remove it from stack)
ret
```

Wiederholung x86

Karlsruher Institut für Technologie

Laden aus Activation Record

```
# Load from offset XX into register %eax
movl XX(%ebp), %eax
```

Schreiben in Activation Record

```
# Store value in register %eax
# into activation record offset XXX
movl %eax, XX(%ebp)
```

Offsets:

- Funktionsparameter: 8, 12, ...
- Variablen: -4, -8, -12, ...

Wiederholung x86

Karlsruher Institut für Technologie

Funktionsaufrufe

```
# push value for 2nd parameter (right-to-left order)
pushl %ebx
# push value for 1st parameter
pushl %eax
# call
call MyFunction
# remove arguments from stack
addl $8, %esp
# Return value is in %eax now
```

Achtung: Aufgerufene Funktion verändert Werte in Registern!
(Stichwort: calling convention, callee-/caller-save Register)

1. Letzte Woche
2. Backends
3. Befehlsanordnung
4. Resourcenverteilung
5. Codeausgabe, Backendschema
6. Assembler, Linker, x86
7. Optimierungen
8. Sonstiges

Weiter Optimierungen (freiwillig)

- Adressierungsmodi nutzen
- Peephole Optimierungen. Beispiele
  - `jmp .L177; .L177:` weglassen
  - `movl 0, %eax` durch `xorl %eax, %eax` ersetzen
  - `movl %eax, $var; movl $var, %ebx` ersetzen durch `movl %eax, %ebx`
- Geschickte Grundblockanordnung (fallthroughs erzeugen)
- (Einfache) Registerallokation
- Code ohne Rahmenzeiger erzeugen

Feedback! Fragen? Probleme?

Karlsruher Institut für Technologie

- Anmerkungen?
- Probleme?
- Fragen?