Praktikum Compilerbau
Sitzung 10 – Codeerzeugung

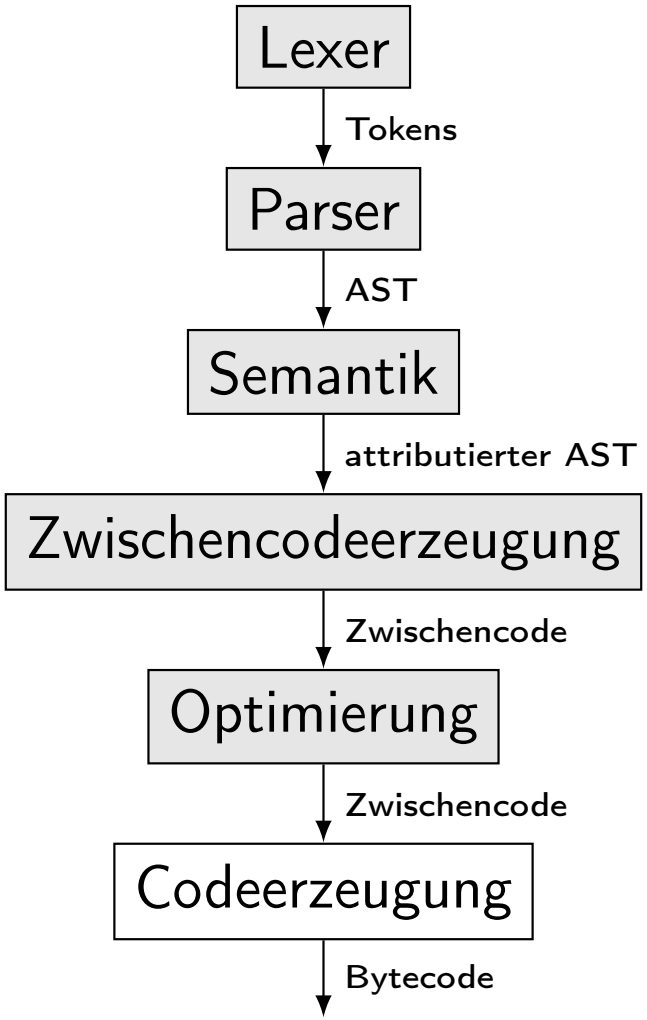
Lehrstuhl für Programmierparadigmen
Universität Karlsruhe (TH)

1. Juli 2009

Letzte Woche

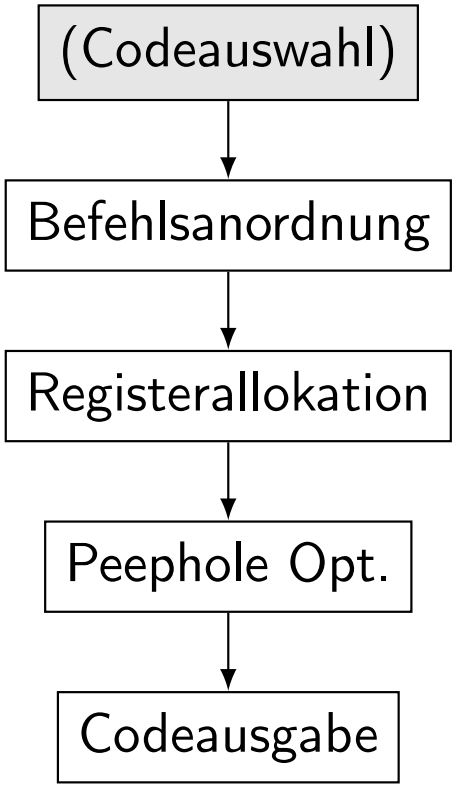
- Was waren die Probleme?
- Hat soweit alles geklappt?

Compilerphasen



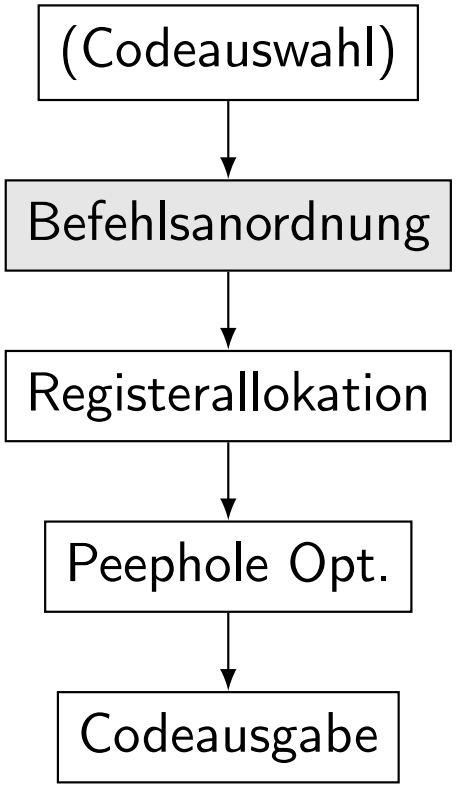
Aufbau eines Compilerbackends - Befehlsauswahl

- Abbilden von Befehlen der Zwischensprache auf Befehle der Zielmaschiene. Meist werden n Zwischensprachbefehle auf einen Zielmaschinenbefehl abgebildet.
- Bei uns bis auf 1 Konstrukt eine $1:1$ Abbildung. (bei welchem Konstrukt haben wir $n:1$?)
- Deshalb bei uns keine separate Codeauswahlphase!



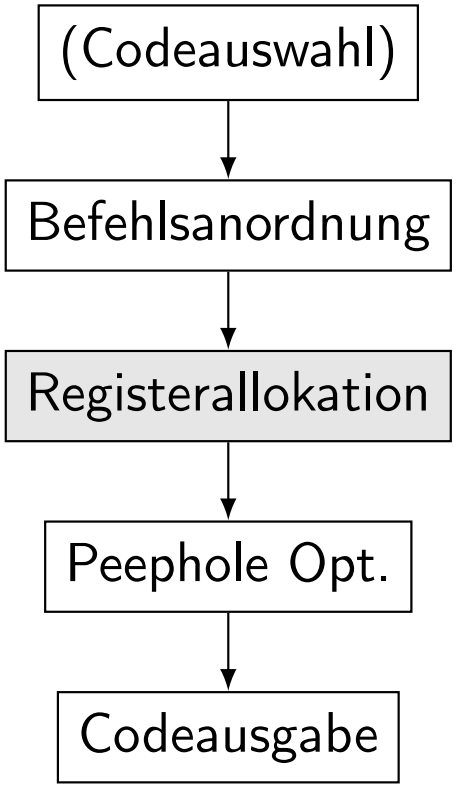
Aufbau eines Compilerbackends - Befehlsanordnung

- Bestimme abhängigkeiten zwischen Befehlen und ordne diese neu an.
- Anordnungsziel: Minimiere Resourcenbedarf und nutze Hardwareeigenschaften (Pipelining) aus.



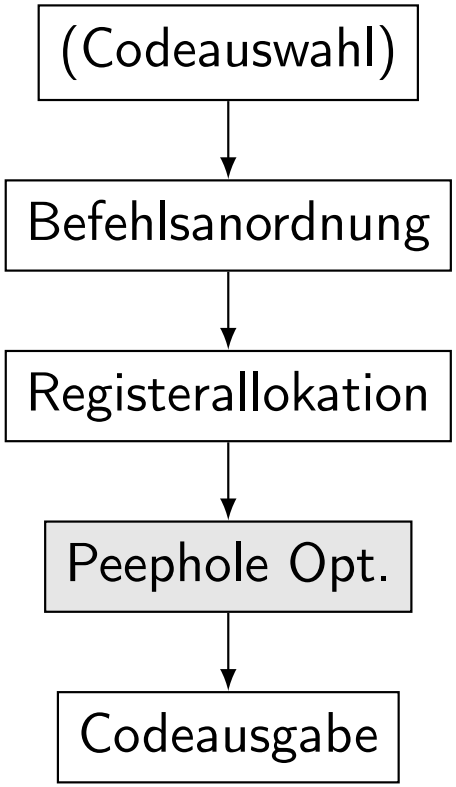
Aufbau eines Compilerbackends - Registerallokation

- Behandlung von Resourcenbeschränkungen: Register, Stackframe, etc. zuteilen.
- Bei Registermangel erzeugung von Auslagerungscode.
- Registerallokation bei uns nicht als separate Phase nötig (unbeschränkte Zahl von Variablen vorhanden).



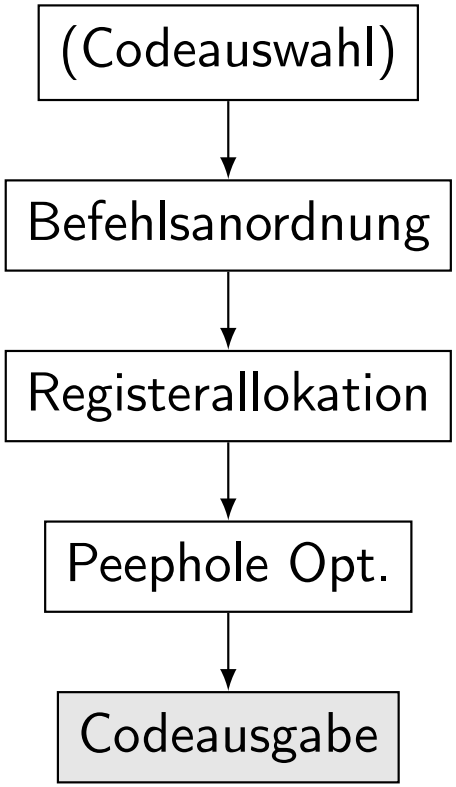
Aufbau eines Compilerbackends - Peephole Optimierungen

- Ersetze Muster von Zielsprachbefehlen durch billigere.
- Typisches Beispiele
  - `Jmp L1; L1:` weglassen
  - `iconst 1` durch `iconst_1` ersetzen
  - Ausnutzen von speziellen Adressierunsmodi
- Bei uns nicht (oder nur wenig) nötig.



Aufbau eines Compilerbackends - Codeausgabe

- Ausgaben von Assembler oder direktes erzeugen des Maschinencodes.
- Auflösen von Sprungmarken (beim direkten Erzeugen)

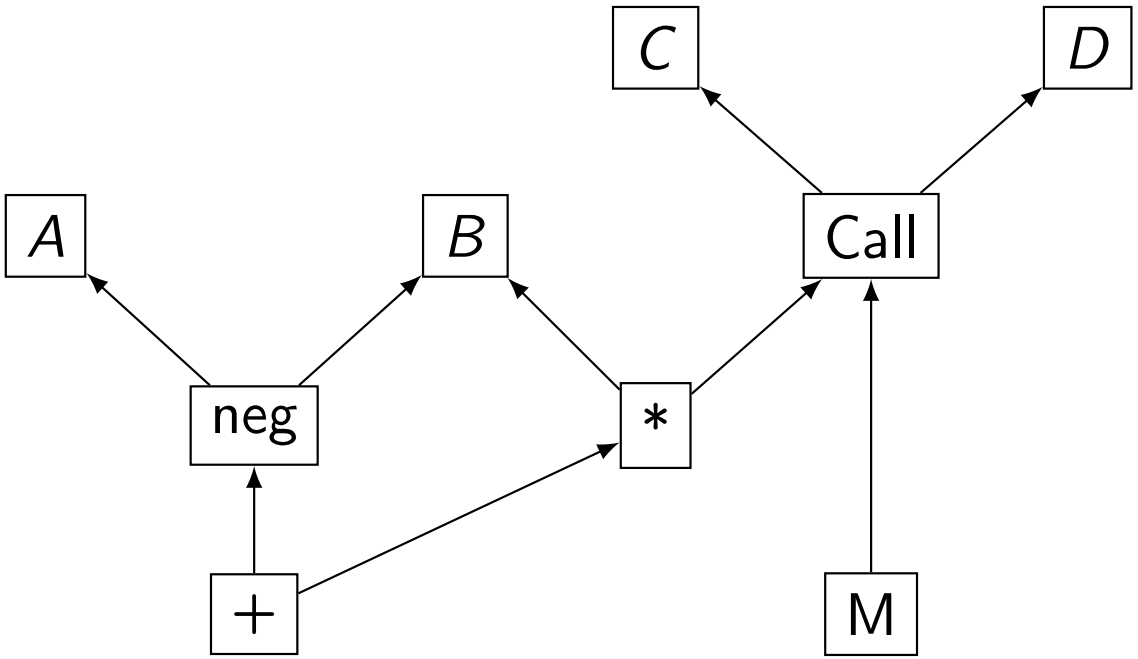


Befehlsreihenfolge in einem Grundblock

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

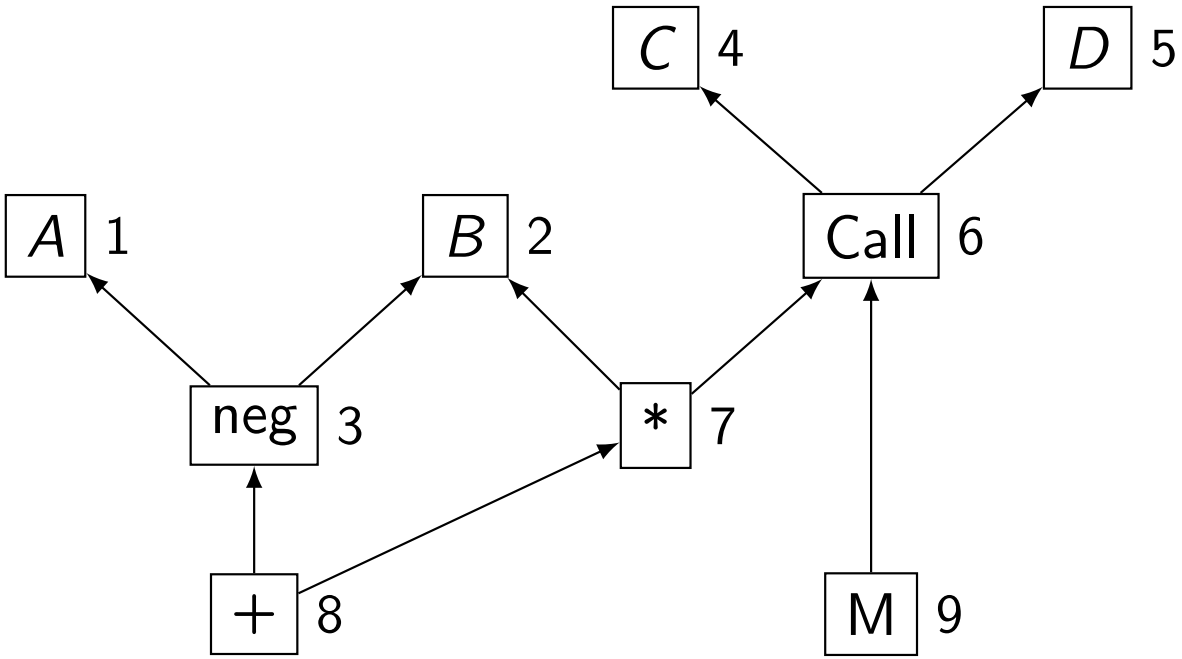
Reverse Postorder

- 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.

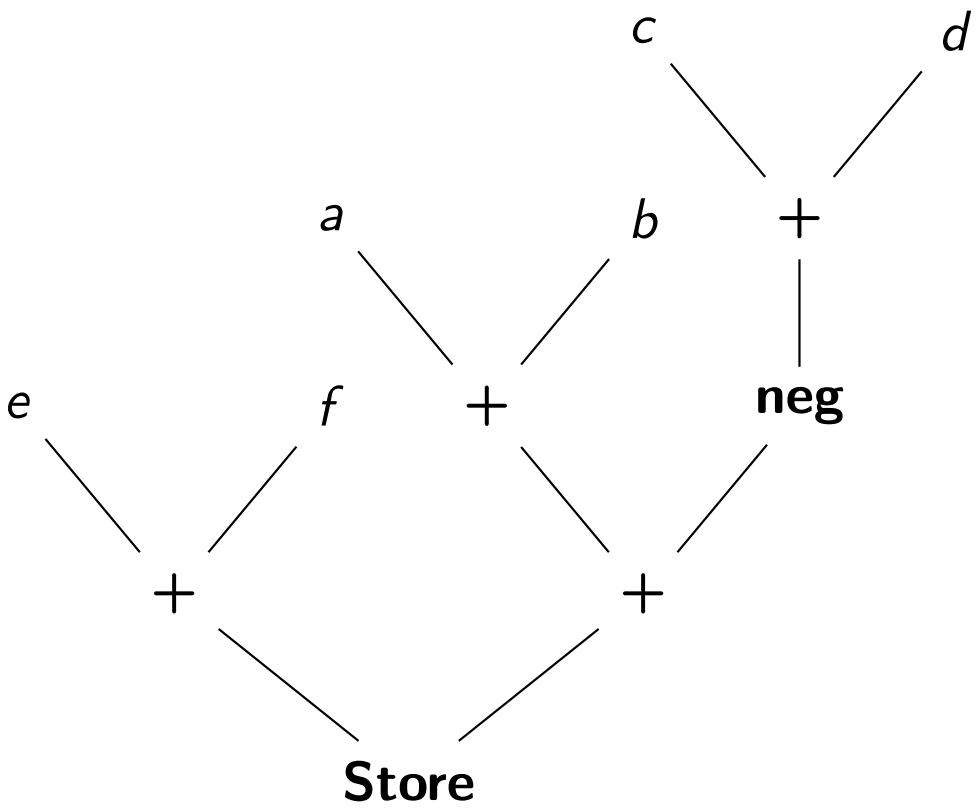


Resourcenverteilung

- Bytecode ist stackbasiert.
- Anordnung garantiert aber nicht, dass die Operanden für jeden Befehl auf der Spitze des Stacks liegen. $\Rightarrow$ Ausweichen auf Variablen.
- Naive Lösung: Eine Variable für jeden Firm-Knoten. Vor Operation Operanden aus Variablen laden, nach Operation Variable schreiben.

Verfeinerung: Erzeugen von Stackcode aus Bäumen

Erzeugung von Stackcode aus Bäumen durch Postfixform:
Für jeden Knoten, rufe Erzeugerfunktion rekursiv für alle Kinder auf und erzeuge dann die Operation.
Beispiel:



Schematisch:

e
f
add
a
b
add
c
d
add
neg
add
store

Φ-Knoten Behandlung

- Teile Variablen für Φ-Knoten zu.
- Am Ende der Vorgängerblöcke eines Φ-Knotens speichere Argumente in (die gleiche) Variable.
- Ersetze Φ-Knoten durch Ladebefehl für die Variable.

Teilgraphen mit Baumstruktur

- DAGs haben aufspannende Bäume als Teilgraphen. („Bäume mit Querkanten ;-)")
- Codeerzeugung also durch Tiefensuche/Postfixform mit Sonderbehandlung der Kanten die nicht zum Baum gehören.
- Diese Kanten haben mehrere Benutzer oder sind Wurzel eines Teilbaums. Variablen nur für Knoten an einer solchen Kanten zuweisen.

Grobes Schema

- Suche alle Klassen im Programm:

```
for(Type type : Program.getTypes()) {
if (! (type instanceof ClassType)) { continue; }
/* ... */
}
```

- Gib für jede Klasse einen Klassenheader und Standardkonstruktor aus.
- Erzeuge jasmin Definition für alle Felder und Methoden in der Klasse.

```
for(Entity entity : classType.getMembers()) {
if (entity.getType() instanceof MethodType) { continue; }
emitField(entity);
}
for(Entity entity : classType.getMembers()) {
if (! (entity.getType() instanceof MethodType)) { continue; }
emitMethod(entity.getGraph());
}
```

Methoden: Erzeugen von Befehlslisten

- Benutze `Graph.walkPostOrder` um Graph in reverse Postorder zu durchlaufen. (Firm-Graph ist bereits in umgekehrter Reihenfolge)
- Ordne Befehle dabei in Befehlslisten ein. Eine Liste pro Block.
- Achtung: Sprungbefehle sind in Firm nicht geordnet, müssen bei der Ausgabe aber als letztes im Grundblock erscheinen.

Methoden: Baumwurzeln markieren

Als Wurzel markieren und Zuteilung von Variablennummern an:

- Alle Knoten mit mehreren Verwendern (`BackEdge.getNOuts(node)>1`)
- Verwender aus anderen Grundblöcken
- Paramter-Projs – Paramter liegen in (vorgegebenen) Variablen
- Φ-Knoten benötigen eine Variable.
- Vorsicht bei Proj-Knoten von `Call`/`Load`/`Store`: Diese erzeugen nicht wirklich Code/Werte. Hier kommt es auf den entsprechenden `mode_T` Knoten an.

Als Wurzeln markieren ohne eigene Variablennummer: `Store`, Sprungbefehle

Ausgabe

Für jeden Grundblock:

- `BlockXX:` ausgeben
- Befehlsliste durchgehen. Für Wurzelknoten Code erzeugen:
  - Sprungbefehle ausgeben
  - Für Store:

    ```
    pushValue(skipSel(store.getPtr()));
    pushValue(store.getValue());
    printf("putfield %s\n", getFieldSpec(store.getPtr()));
    ```

  - Sonst: `createValue(node); /* emit astore/istore */`

    ```
    createValue(node);
    String storecmd = node.getMode().isReference() ? "astore" : "istore";
    printf("%sstore %d\n", storecmd, varnum);
    ```

-

Schema create/pushValue

```
private void pushValue(Node node) {
  if (varAssigned(node)) {
    /* emit aload/iload varnumber */
    return;
  }
  createValue(node);
}
private void createValue(Node node) {
  switch (node.getOpCode()) {
  case iro_Const: /* ... */ break;
  case iro_Add:
    Add add = (Add) node;
    pushValue(add.getLeft());
    pushValue(add.getRight());
    println("\tiadd");
    break;
  }
}
```

Feedback! Fragen? Probleme?

- Anmerkungen?
- Probleme?
- Fragen?