Praktikum Compilerbau

Wintersemester 2011/12

Andreas Abel und Ulrich Schöpp
(Dank an Hans-Wolfgang Loidl und Robert Grabowski)

Übersicht

Organisatorisches

Das Praktikum richtet sich nach dem Buch
Modern Compiler Implementation in Java von Andrew Appel, Cambridge University Press, 2005, 2. Auflage

Es wird ein Compiler für *MiniJava*, eine Teilmenge von Java, entwickelt.

- Implementierungssprache: Java
  (oder SML, OCaml, F#, Haskell, Scala, C#, C++, ...)
- mögliche Zielarchitekturen: x86 oder x86-64 (oder ARM, MIPS, ...)

Jede Woche wird ein Kapitel des Buchs durchgegangen; ca. 30% Vorlesung und 70% Programmierung im Beisein der Dozenten.

Programmierung in Gruppen à zwei Teilnehmern.

Die beaufsichtigte Programmierzeit wird i.A. nicht ausreichen; Sie müssen noch ca. 4h/Woche für selbstständiges Programmieren veranschlagen.

Benotung durch eine Endabnahme des Programmierprojekts.

- Anforderung: Funktionierender Compiler von MiniJava nach Assembler-Code.
- Die Abnahme wird auch mündliche Fragen zu dem in der Vorlesung vermittelten Stoff enthalten.

Einführung

Mo	17.10.	Einführung; Interpreter
Mo	24.10.	Lexikalische Analyse und Parsing
Mo	31.10.	Abstrakte Syntax und Parser
Mo	07.11.	Abstrakte Syntax und Parser
Mo	14.11.	Semantische Analyse
Mo	21.11.	Activation records
Mo	28.11.	Zwischensprachen
Mo	5.12.	Basisblöcke
Mo	12.12.	Instruktionsauswahl
Mo	19.12.	Automatisches Speichermanagement
Mo	9.1.	Aktivitätsanalyse (liveness analysis)
Mo	16.1.	Registerverteilung
Mo	23.1.	Optimierungen
Mo	30.1.	Static-Single-Assignment-Form
Mo	6.2.	Objektorientierte Sprachen

MiniJava

MiniJava ist eine Teilmenge von Java

- primitive Typen: `int`, `int[]`, `boolean`
- minimale Anzahl von Programmstatements: `if`, `while`
- Objekte und Vererbung, aber kein Überladen, keine statischen Methoden außer `main`
- einfaches Typsystem (keine Generics)
- Standardbibliothek enthält nur `System.out.println`
- gleiche Semantik wie Java

Aufgaben eines MiniJava-Compilers

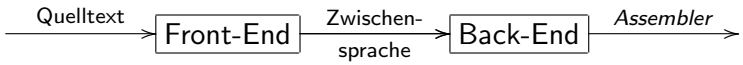
Übersetzung von Quellcode (MiniJava-Quelltext) in Maschinensprache (Assembler).

- Überprüfung, ob der Eingabetext ein korrektes MiniJava Programm ist.
  - Lexikalische Analyse und Syntaxanalyse
  - Semantische Analyse (Typüberprüfung und Sichtbarkeitsbereiche)

  Ausgabe von informativen Fehlermeldungen bei inkorrekter Eingabe.
- Übersetzung in Maschinensprache
  - feste Anzahl von Machinenregistern, wenige einfache Instruktionen, Kontrollfluss nur durch Sprünge, direkter Speicherzugriff
  - effizienter, kompakter Code
  - ...

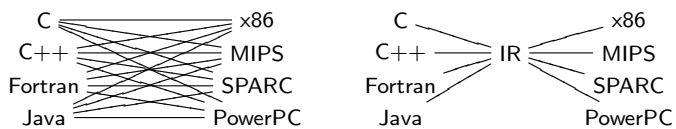
Aufbau eines Compilers

Compiler bestehen üblicherweise aus *Front-End* und *Back-End*.



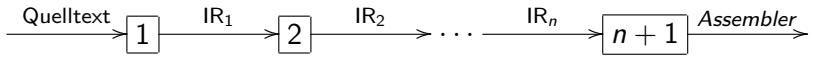
Zwischensprache(n)

- abstrakter Datentyp, leichter zu behandeln als Strings
- weniger komplex als Eingabesprache ⇒ Transformationen und Optimierungen leichter implementierbar
- Zusammenfassung ähnlicher Fälle, z.B. Kompilation von `for`- und `while`-Schleifen ähnlich.
- Kombination mehrerer Quellsprachen und Zielarchitekturen



Aufbau eines Compilers

Moderne Compiler sind als Verkettung mehrerer Transformationen zwischen verschiedenen Zwischensprachen implementiert.



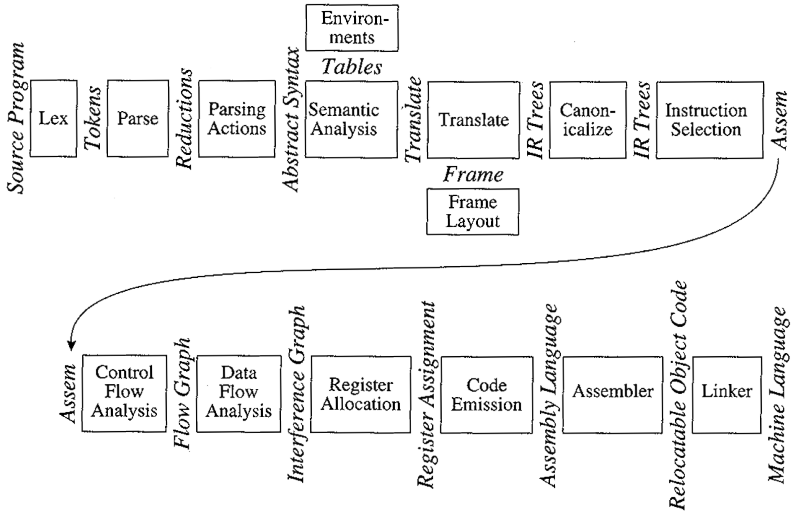
(IR — Intermediate Representation)

- Zwischensprachen nähern sich Schrittweise dem Maschinencode an.
- Optimierungsschritte auf Ebene der Zwischensprachen

Designprinzipien

- **safety first**: Fehlerfälle im Compiler möglichst früh abfangen, z.B. Typüberprüfung.
- **small is beautiful**: Mehrere, kleine Duchgänge des Compilers sind übersichtlicher und besser wartbar als wenige, komplexe Duchgänge.
- **typed intermediate languages are cool**: Typüberprüfung auf Zwischencodeebene erhöht die Sicherheit von Programmtransformationen.

Aufbau des MiniJava-Compilers



Zwischensprachen

Quelltext als String

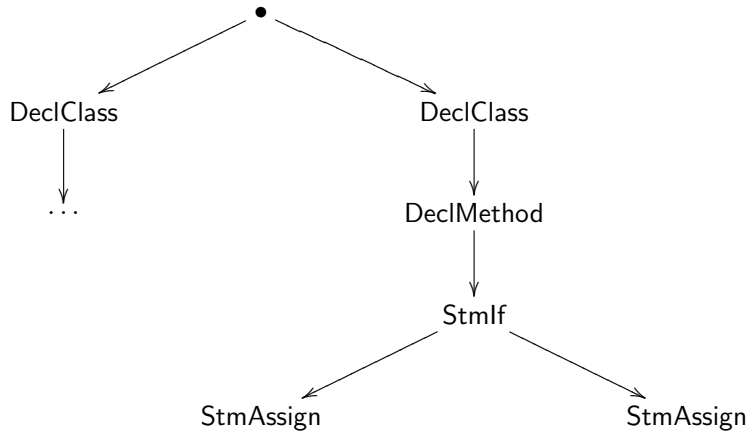
```
"class Factorial{
  public static void main(String[] a){
    System.out.println(new Fac().ComputeFac(10));
  }
}

class Fac {

  public int ComputeFac(int num){
    int num_aux;
    if (num < 1)
      num_aux = 1;
    else
      num_aux = num * (this.ComputeFac(num-1));
    return num_aux;
  }
}"
```

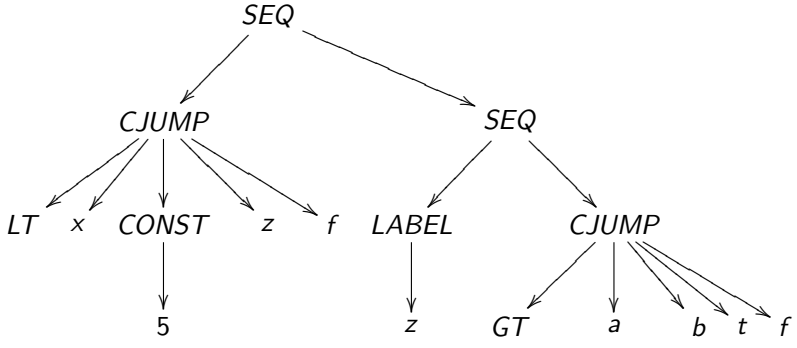

Zwischensprachen

Abstrakte Syntax



Zwischensprachen

IR Trees



Zwischensprachen

Assembler mit beliebig vielen Registern

```
Lmain:
  push %ebp
  mov %ebp, %esp
  sub %esp, 4
L$$205:
  mov t309,42
  mov t146,%ebx
  mov t147,%esi
  mov t148,%edi
  mov t310,4
  push t310
  call L_halloc_obj
  mov t311,%eax
  add %esp,4
  mov t145,t311
```


Zwischensprachen

Assembler

```
Lmain:
  push %ebp
  mov %ebp, %esp
  sub %esp, 8
L$$205:
  mov %eax,42
  mov %eax,%ebx
  mov  DWORD PTR [%ebp - 4],%eax
  mov %eax,4
  push %eax
  call L_halloc_obj
  add %esp,4
  mov %ebx,%eax
  mov %eax,4
```


Praktikumsteil heute

Wiederholung am Beispiel von Straightline-Programmen

- Repräsentierung baumartiger Strukturen in Java
- Visitor Pattern

Aufgabe bis zum nächsten Mal:

- Gruppen finden
- Entwicklungsumgebung einrichten
- SVN anlegen (siehe `http://www.rz.ifi.lmu.de/Dienste/Subversion`)

Geradlinige (Straightline) Programme

- Bestehen aus Zuweisungen, arithmetischen Ausdrücken, mehrstelligen Druckanweisungen.
- Beispiel:

  `a := 5+3; b := (print(a, a-1), 10*a); print(b)`

  Ausgabe:

  ```
  8 7
  80
  ```

  Abstrakte Syntax als BNF Grammatik:

Stm	::=	*Stm*;*Stm* | *ident*:=*Exp* | `print(`*ExpList*`)`
Exp	::=	*ident* | *num* | (*Stm*, *Exp*) | *Exp Binop Exp*
ExpList	::=	*Exp* | *Exp*, *ExpList*
Binop	::=	+ | − | ∗ | /

Abstrakte Syntax in Haskell

$$
\begin{array}{lcl}
\mathit{Stm} & ::= & \mathit{Stm}\,;\mathit{Stm} \mid \mathit{ident} := \mathit{Exp} \mid \texttt{print}\,(\mathit{ExpList}) \\
\mathit{Exp} & ::= & \mathit{ident} \mid \mathit{num} \mid (\mathit{Stm}\,,\mathit{Exp}) \mid \mathit{Exp}\ \mathit{Binop}\ \mathit{Exp} \\
\mathit{ExpList} & ::= & \mathit{Exp} \mid \mathit{Exp}\,,\mathit{ExpList} \\
\mathit{Binop} & ::= & + \mid - \mid * \mid /
\end{array}
$$

```
data Stm = CompoundStm { stm1 :: Stm, stm2 :: Stm }
         | AssignStm   { id :: String, exp :: Exp }
         | PrintStm    { exps :: [Exp] }

data Exp = IdExp  { id :: String }
         | NumExp { num :: Integer }
         | OpExp  { left :: Exp, op :: Binop, right :: Exp }
         | EseqExp{ stm :: Stm, exp :: Exp }

data Binop = PLUS | MINUS | TIMES | DIV
```


Abstrakte Syntax in Java I

```
abstract class Stm {}

final class CompoundStm extends Stm {
   final Stm stm1;
   final Stm stm2;
   CompoundStm(Stm s1, Stm s2) {stm1 = s1; stm2 = s2;}
}

final class AssignStm extends Stm {
   final String id;
   final Exp exp;
   AssignStm(String i, Exp e) {id = i; exp = e;}
}

final class PrintStm extends Stm {
   final List<Exp> exps;
   PrintStm(List<Exp> e) { exps = e; }
}
```


Abstrakte Syntax in Java II

```
abstract class Exp {}

final class IdExp extends Exp {
   final String id;
   IdExp(String i) {id = i;}
}

final class NumExp extends Exp {
   final int num;
   NumExp(int n) {num = n;}
}

final class OpExp extends Exp {
   enum BinOp {PLUS, MINUS, TIMES, DIV};
   final Exp left, right;
   final BinOp oper;
   OpExp(Exp l, BinOp o, Exp r) {left = l; oper = o; right = r;}
}

final class EseqExp extends Exp {
   final Stm stm;
   final Exp exp;
   EseqExp(Stm s, Exp e) {stm = s; exp = e;}
}
```


Beispielprogramm

```
        a = 5+3; b = (print (a, a-1), 10*a); print (b)

import static OpExp.Binop.*;

List<Exp> l1 = new LinkedList<Exp>();
l1.add(new IdExp("a"));
l1.add(new OpExp(new Exp.IdExp("a"), MINUS, new NumExp(1)));
List<Exp> l2 = new LinkedList<Exp>();
l2.add(new IdExp("b"));

Stm stm =
  new CompoundStm(
      new AssignStm("a",
          new OpExp(new NumExp(5), OpExp.Binop.PLUS, new NumExp(3))),
      new CompoundStm(
          new AssignStm("b",
              new EseqExp(new PrintStm(l1),
                  new OpExp(new NumExp(10), TIMES, new IdExp("a")))),
          new PrintStm(l2)));
```

Programmieraufgabe: Straightline Interpreter

Implementieren Sie in der Klasse `Stm` eine Methode:

```
Map<String, Integer> eval(Map<String, Integer> t)
```

Der Aufruf

```
Map<String, Integer> tneu = s.eval(t)
```

soll das Programm `s` in der Umgebung `t` auswerten und die daraus resultierende neue Umgebung in `tneu` zurückliefern.

Die Umgebung `t` soll nicht verändert werden.

Empfohlene Programmiertechniken

Der Interpreter dient nur als Aufwärmübung. Grundkenntnisse in Java (oder der gewählten Implementationssprache) sind Voraussetzung.

Besonders nützlich für den eigentlichen Compiler sind:

- **Datenrepräsentation:** Modellierung der abstrakten Syntax mittels einer Klassen-Hierarchie.
- **Programmiertechnik:** Iteration über diese Datenstruktur mittels eines *Visitor Pattern*.
- **Datenstrukturen:** Umgang mit unveränderlichen (immutable) Datenstrukturen.
  Die *Google Collection Library* enthält effiziente Implementierungen solcher Datenstrukturen.

Visitor Pattern

Eine generische Struktur zum Ausführen von Operationen auf allen Elementen einer komplexen Datenstruktur.

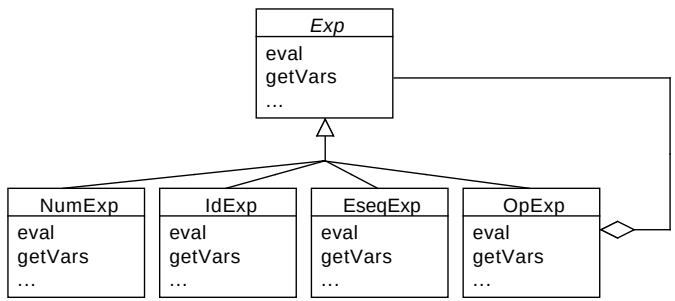
- Fördert eine *funktionsorientierte Sichtweise*: der Code für eine Operation auf einer gesamten Datenstruktur wird in einem Modul zusammengefasst (z.B. Typüberprüfung auf der abstrakten Syntax)
- Es werden 2 Klassen-Hierarchien aufgebaut: eine für die Objekt-Klassen (Daten) und eine für die Visitor-Operationen (Code).
- Geeignet für Anwendung mit fixer Datenstruktur (z.B. abstrakter Syntax) und verschiedenen Operationen, die darauf arbeiten.

Visitor Pattern

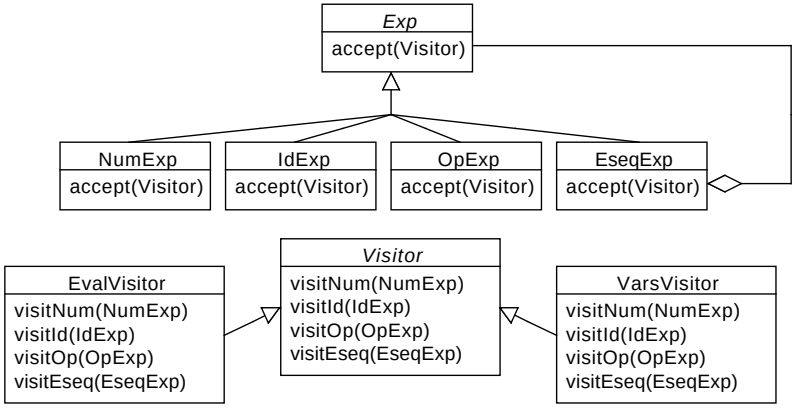
Idee:

- Sammle die Definitionen der Operationen auf der Objektstruktur in einem Visitor-Objekt.
- Ersetze die verschiedenen Operationen durch eine einzige `accept`-Methode.

Beispiel



Visitor Pattern



`accept(v)` in `NumExp` durch {`v.visitNum(this)`} implementiert, usw. ⇒ Auslagerung der Methodendefinitionen in Visitor-Objekte.

Visitor Pattern

```
abstract class Exp {
  abstract <T> T accept(ExpVisitor<T> v);
}

final class IdExp extends Exp {
  ...
  <T> T accept(ExpVisitor<T> v) { return v.visitVar(this); }
}

final class NumExp extends Exp {
  ...
  <T> T accept(ExpVisitor<T> v) { return v.visitConst(this); }
}

...
```


Visitor Pattern

```
abstract class ExpVisitor<T> {
    abstract T visitId(IdExp v);
    abstract T visitNum(NumExp c);
    abstract T visitOp(OpExp p);
    abstract T visitEseq(EseqExp p);
}
```

Funktionen für arithmetische Ausdrücke können nun in zentral in einem `ExpVisitor`-Objekt aufgeschrieben werden, ohne die Syntax ändern zu müssen.

Unveränderbare Objekte

Ein Objekt ist unveränderbar (immutable) wenn sich sein interner Zustand nicht ändern kann.

Unveränderbare Objekte:

- einfach
- thread-sicher
- können beliebig weitergegeben werden
- eignen sich gut als Bausteine für größere Objekte
- führen evtl. zu zusätzlichen Kopieroperationen

Unveränderbare Objekte sollten veränderbaren wenn möglich vorgezogen werden.

Unveränderbare Objekte

Beispiel

Klasse `Assem` repräsentiert Assembler-Instruktionen wie zum Beispiel `mov t12, %eax`.

Möglichkeiten zur Modellierung einer Funktion `Assem.rename` für die Umbenennung der Registernamen in einer Instruktion:

1. `public void rename(Map<...> m)`
   Verändert das Objekt.
2. `public Assem rename(Map<...> m)`
   Konstruiert ein neues Objekt, das die umbenannte Instruktion repräsentiert. Das Objekt selbst bleibt unverändert.