Praktikum Compilerbau

Wintersemester 2018/19

Ulrich Schöpp

(Dank an Andreas Abel, Robert Grabowski, Martin Hofmann und Hans-Wolfgang Loidl)

Übersicht

Organisatorisches
Einführung

Organisatorisches

Das Praktikum richtet sich grob 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, Haskell, Rust, OCaml, …
- Zielarchitektur: x86

Jede Woche wird die Implementierung ein Stück weiterentwickelt.

- Vorlesungsteil (Theorie)
- Übungsteil (praktische Umsetzung)

Programmierung in Gruppen à zwei Teilnehmern.

Die Zeit in der Übung 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.

Mo	15.10.	Einführung; Interpreter
Mo	22.10.	Lexikalische Analyse und Parsing
Mo	29.10.	Abstrakte Syntax und Parser
Mo	5.11.	Semantische Analyse
		Milestone 1: Parser und Typchecker
Mo	12.11.	Zwischensprachen
Mo	19.11.	Activation records
Mo	26.11.	Basisblöcke
		Milestone 2: Übersetzung in Zwischensprache
Mo	3.12.	Instruktionsauswahl
Mo	10.12.	Aktivitätsanalyse (liveness analysis)
		Milestone 3: Übersetzung in Assembler mit Reg.variablen
Mo	17.12.	Registerverteilung
Mo	7.1.	Garbage Collection
Mo	14.1.	Static Single Assignment Form
Mo	21.1.	Optimierungen
Mo	28.1.	(optionales Thema)
		Milestone 4: Fertiger Compiler
	Feb.	Endabnahmen

Einführung

MiniJava

MiniJava ist eine kleine Teilmenge von Java

- Typen: `int`, `int[]`, `boolean`
- minimale Anzahl von Anweisungen: `if`, `while`
- Objekte und (optional) Vererbung, aber kein Überladen, keine statischen Methoden außer `main`
- einfaches Typsystem (keine Generics)
- Standardbibliothek:
  - `void System.out.println(int)`
  - `void System.out.write(int)`
  - `int System.in.read()`
- gleiche Semantik wie Java

Aufgaben eines MiniJava-Compilers

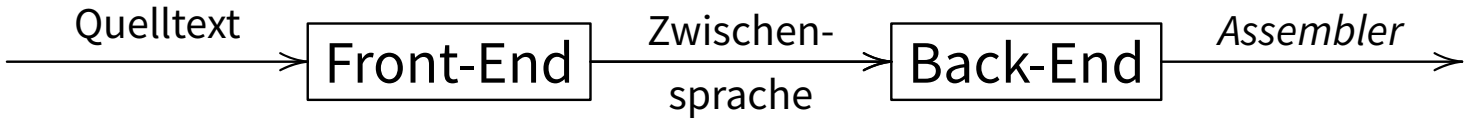
Übersetzung von Quellcode (MiniJava-Quelltext) in Maschinensprache (Assembler).

- 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
  - …

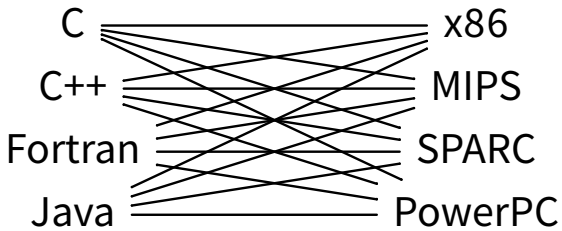
Aufbau eines Compilers

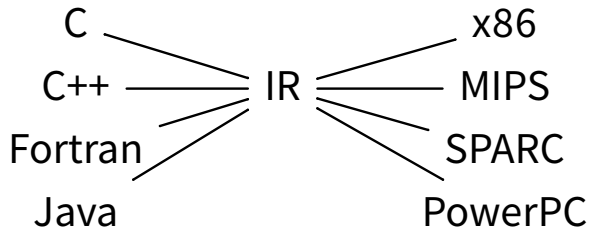
Compiler bestehen üblicherweise aus *Front-End* und *Back-End*.



Zwischensprache(n)

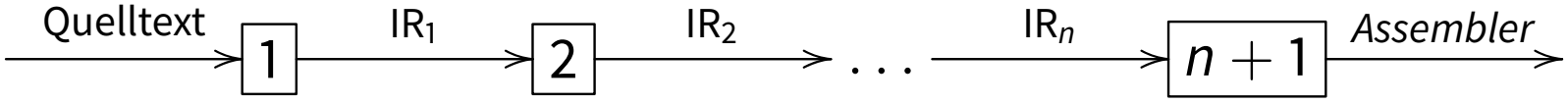
- abstrakter Datentyp, leichter zu behandeln als Strings
- weniger komplex als Eingabesprache $\Rightarrow$ Transformationen und Optimierungen leichter implementierbar
- Zusammenfassung ähnlicher Fälle, z.B. Kompilation von `for`- und `while`-Schleifen ähnlich.
- Kombination mehrerer Quellsprachen und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

Aufbau eines Compilers

Erstes Ziel beim Compilerbau ist die Korrektheit des Compilers.

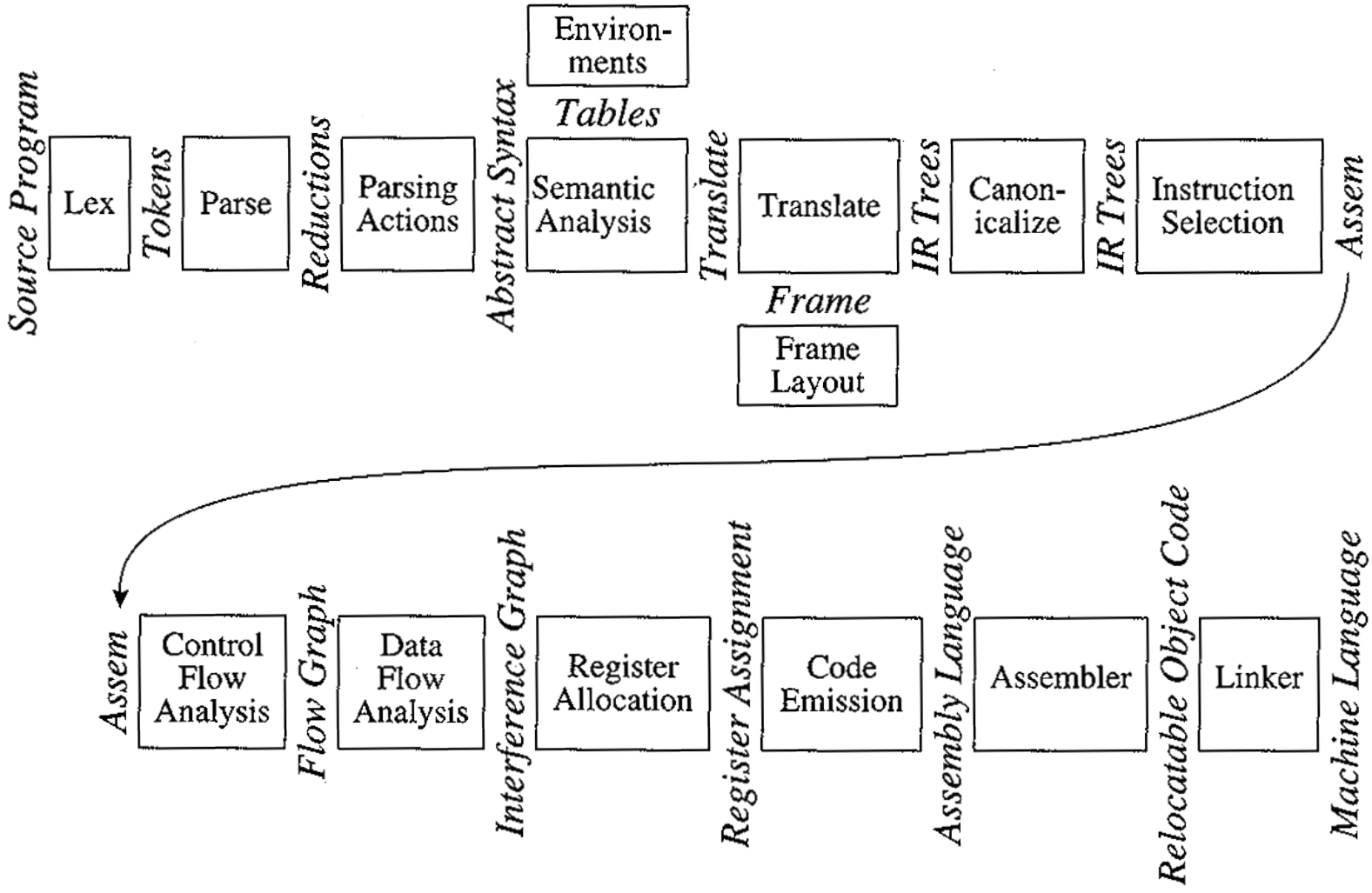
Die Aufteilung in eine Folge einfacher Transformationen hilft, die Korrektheit sicherzustellen.

- Kleine Transformationen sind übersichtlicher und leichter zu entwickeln als größere.
- Die einzelnen Transformationen sind unabhängig testbar.
- Zwischenergebnisse können überprüft werden, z.B. durch Zwischensprachen mit statischer Typüberprüfung.
- Zwischensprachen liefern klare Schnittstellen zur Arbeitsteilung und Wiederverwendung.

Prioritäten

- safety-first
- small is beautiful

Aufbau des MiniJava-Compilers



Zwischensprachen

Quelltext als String

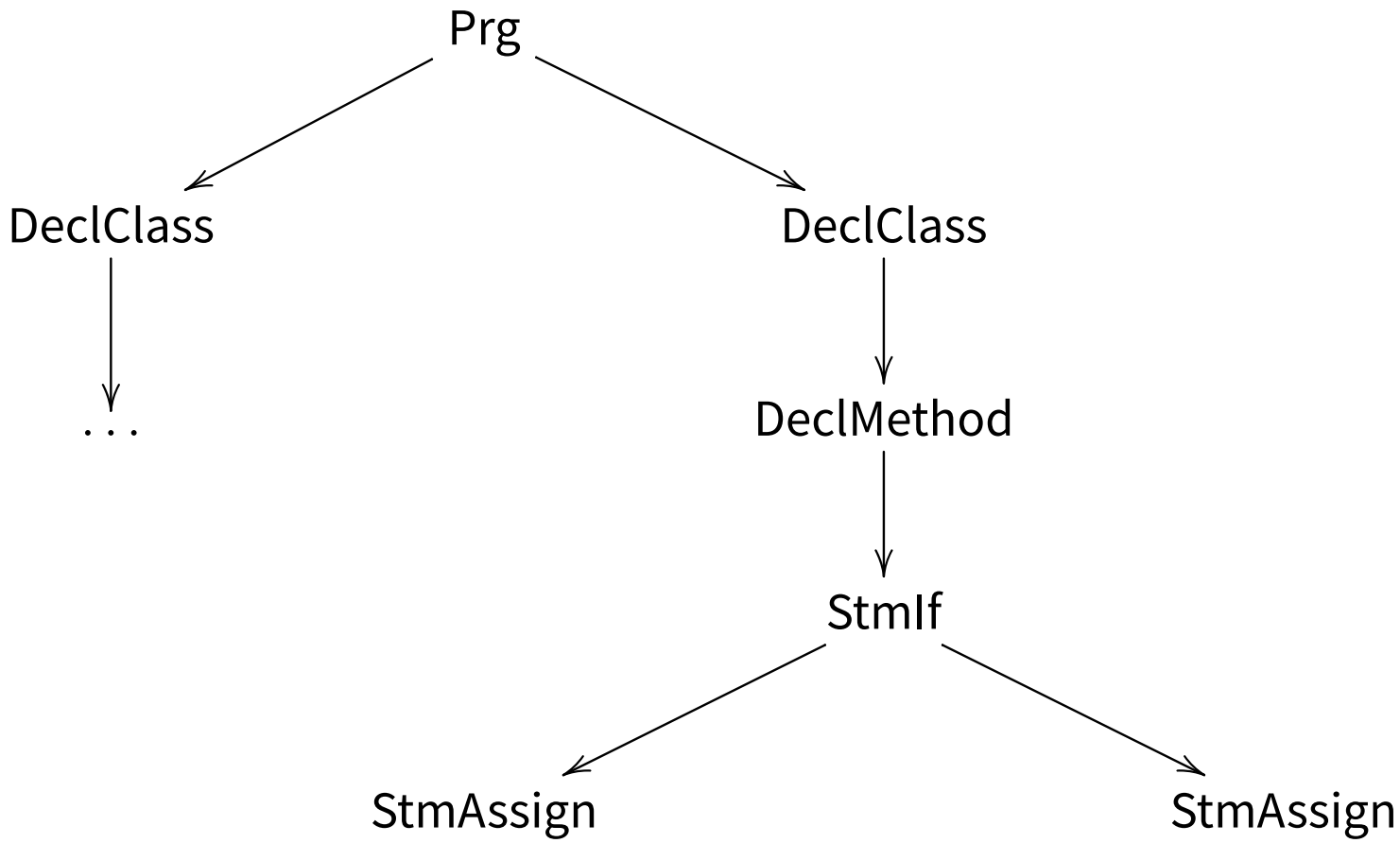
```
"class Factorial{
  public static void main(String[] a){
    System.out.println(new Fac().ComputeFac(10));
  }
}

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



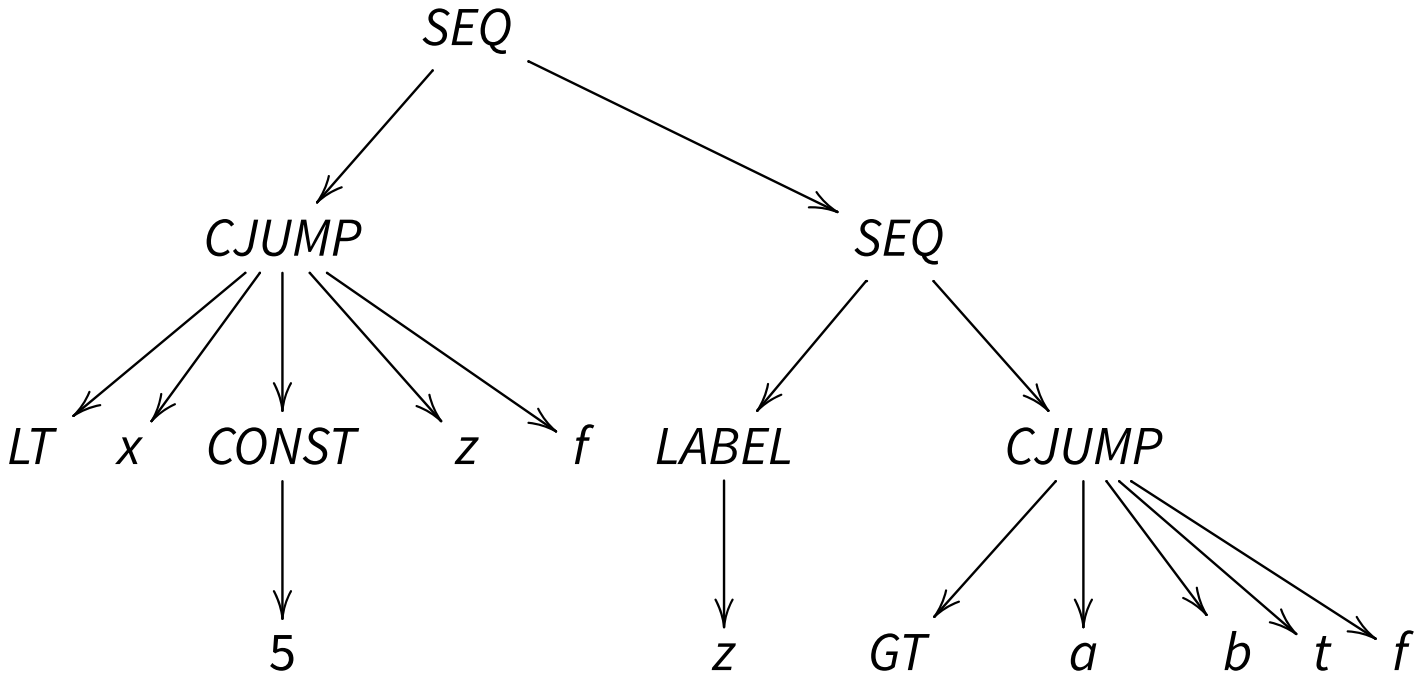
Zwischensprachen

IR: Abstraktionsebene vergleichbar mit C

```
LFac$ComputeFac(2) {
  CJUMP(LT, PARAM(1), CONST(1), L$0, L$1)
  LABEL(L$1)
  MOVE(TEMP(t5), BINOP(MUL, PARAM(1),
       CALL(NAME(LFac$ComputeFac),
            PARAM(0),
            BINOP(MINUS, PARAM(1), CONST(1)))))
  JUMP(NAME(L$2), L$2)
  LABEL(L$0)
  MOVE(TEMP(t5), CONST(1))
  LABEL(L$2)
  MOVE(TEMP(t4), TEMP(t5))
  return t4
}
```

Zwischensprachen

IR Trees: Zwischensprachen brauchen keine konkrete Syntax und werden intern durch Syntaxbäume gespeichert.




```
CJUMP(LT, x, CONST(5), z, t);
LABEL(z);
CJUMP(GT, a, b, t f)
```

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

Aufwärmung am Beispiel von Straightline-Programmen

- Repräsentierung abstrakter Syntax in Java
- Visitor Pattern

Aufgabe bis zum nächsten Mal:

- Gruppen finden
- Entwicklungsumgebung einrichten
- Ein Projekt für den MiniJava-Compiler auf `gitlab.cip.ifi.lmu.de` einrichten.

Abstrakte Syntax

Beispiel: arithmetische Ausdrücke

Konkrete Syntax

```
5 + (1 + 2 * 3) * 4
```

Abstrakte Syntax

In EBNF

$$Exp \quad ::= \quad num \mid Exp \; + \; Exp \mid Exp \; * \; Exp$$

In Haskell

```
data Exp = Num Int | Plus Exp Exp | Times Exp Exp

example = Plus (Num 5) (Plus (Num 1) (Times (Num 2) (Num 3)))
```

Abstrakte Syntax (Haskell)

```
data Exp = Num Int | Plus Exp Exp | Times Exp Exp

example = Plus (Num 5) (Plus (Num 1) (Times (Num 2) (Num 3)))
```

Interpreter:

```
eval :: Exp -> Int
eval (Num x) = x
eval (Plus e1 e2) = eval e1 + eval e2
eval (Times e1 e2) = eval e1 * eval e2
```

Beispiel: `eval example` wertet zu `12` aus.

Abstrakte Syntax (Haskell)

```
data Exp = Num Int | Plus Exp Exp | Times Exp Exp

example = Plus (Num 5) (Plus (Num 1) (Times (Num 2) (Num 3)))
```

Umwandlung in Strings:

```
toString :: Exp -> String
toString (Num x) = show x
toString (Plus e1 e2) =
  "(" ++ toString e1 ++ ") + (" ++ toString e2 ++ ")"
toString (Times e1 e2) =
  "(" ++ toString e1 ++ ") * (" ++ toString e2 ++ ")"
```

Beispiel: `toString example` wertet zu
`"(5) + ((1) + ((2) * (3)))"` aus.

Abstrakte Syntax (Java)

Composite Pattern

```
abstract class Exp {}

final class NumExp extends Exp {
   final int value;
   NumExp(int value) { this.value = value; }
}

final class PlusExp extends Exp {
   final Exp left;
   final Exp right;
   PlusExp(Exp left, Exp right) {
     this.left = left;
     this.right = right;
   }
}
```

```
final class TimesExp extends Exp {
   final Exp left;
   final Exp right;
   TimesExp(Exp left, Exp right) {
     this.left = left;
     this.right = right;
   }
}
```

- Eine abstrakte Klasse für jedes Nichtterminalsymbol.
- Eine Unterklasse für jede Produktion.

Abstrakte Syntax (Java) — Funktionen

Möchte man Funktionen wie `eval` oder `toString` implementieren, kann man zu den Syntax-Klassen neue Funktionen hinzufügen.

```
abstract class Exp {
  int eval();
  String toString();
}

final class NumExp extends Exp {
   final int value;
   NumExp(int value) { this.value = value; }
   int eval() { return value; }
   String toString() { return "" + value; }
}
```

```
final class PlusExp extends Exp {
   final Exp left;
   final Exp right;
   BinaryExp(Exp left, Exp right) { ... }

   int eval() {
     return left.eval() + right.eval();
   }

   String toString() {
     return "(" + left.toString()
                + ") + ("
                + right.toString()
                + ")";
   }
}
...
```

Probleme dieses Ansatzes:

- Der Quellcode für `eval` und `toString` ist jeweils über viele verschiedene Klassen verstreut.
- Für jede neue Funktion müssen die Klassen der abstrakten Syntax geändert werden.
  (problematisch für Programmbibliotheken, Trennung in Front- und Backend im Compiler)

Standardlösung: Visitor Pattern

Visitor Pattern

Eine generischer Ansatz zum Ausführen von Operationen auf Werten einer Composite-Datenstruktur.

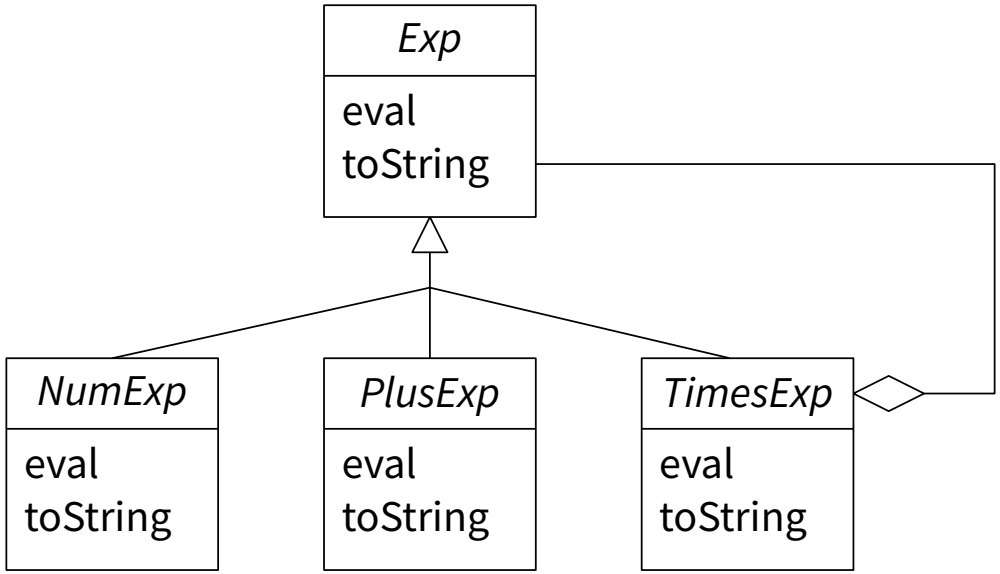
- Fördert eine *funktionsorientierte Sichtweise*: der Code für eine Operation auf einer gesamten Datenstruktur wird in einem Modul zusammengefasst.
- Fördert die Erweiterbarkeit von Programmen.
  - Implementierung neuer Operationen ohne Veränderung der Datenstruktur.
  - Beim Hinzufügen einer neuen Operation können keine Fälle vergessen werden.

Visitor Pattern

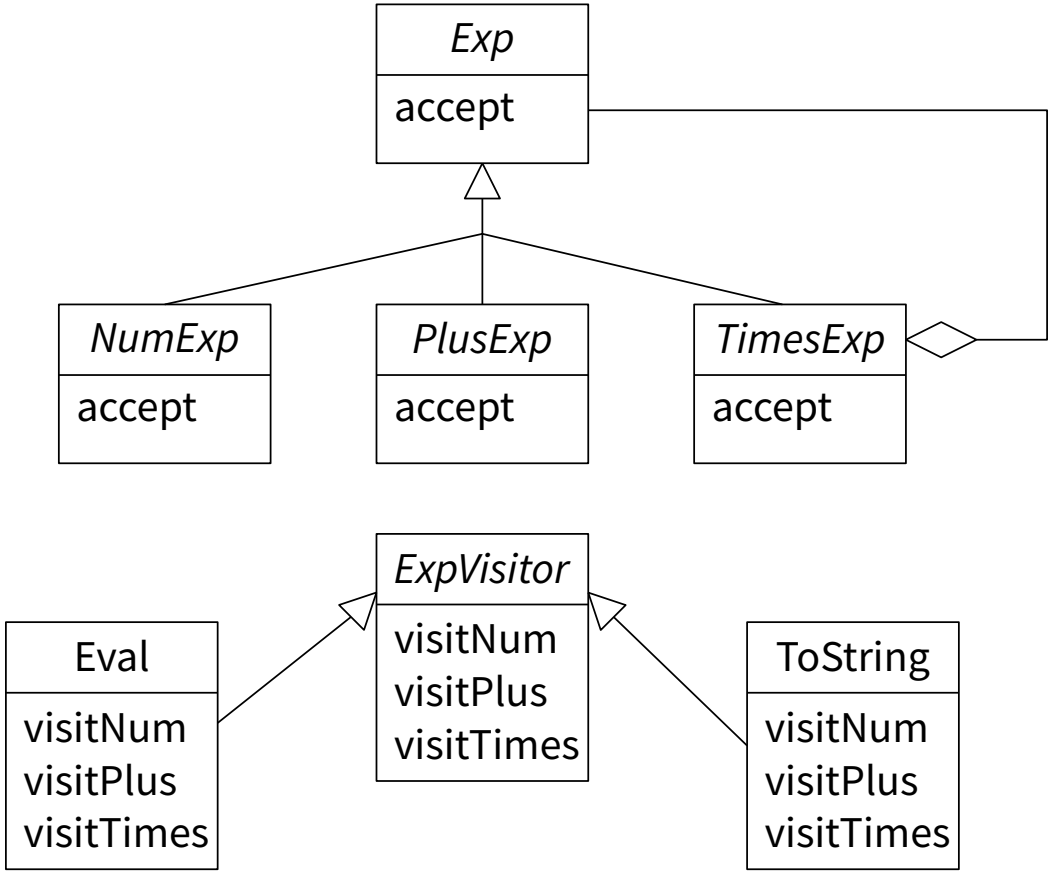
Idee:

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

Unser Beispiel



wird zu:



`accept(v)` in `NumExp` ist durch `{v.visitNum(this)}` implementiert, usw.

$\Rightarrow$ Auslagerung der Methodendefinitionen in Visitor-Objekte.

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

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

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

final class TimesExp extends Exp {
  ...
  <T> T accept(ExpVisitor<T> v) { return v.visitTimes(this);}
}
```

```
abstract class ExpVisitor<T> {
    abstract T visitNum(NumExp c);
    abstract T visitPlus(PlusExp p);
    abstract T visitTimes(TimesExp t);
}
```

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

```
class EvalVisitor implements ExpVisitor<Integer> {

  public Integer visitNum(NumExp c) {
    return c.value;
  }
  public Integer visitPlus(PlusExp p) {
    return p.left.accept(this) + p.right.accept(this);
  }
  public Integer visitTimes(PlusExp p) {
    return p.left.accept(this) * p.right.accept(this);
  }
}
```

Verwendung:

```
Exp example = new ExpPlus(new ExpNum(4), new ExpNum(3));
int v = example.visit(new EvalVisitor());
// v == 7
```

Straightline-Programme

Praktikumsaufgabe für heute: Implementierung eines Interpreters für *Straightline-Programme*.

- Straightline-Programme bestehen aus Zuweisungen, arithmetischen Ausdrücken, mehrstelligen Print-Anweisungen.
- Beispiel:

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

  Ausgabe:

  ```
  8 7
  80
  ```

- Abstrakte Syntax als BNF Grammatik:

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

Straightline Interpreter

Aufgabe für heute: Implementierung eines Interpreters für Straightline-Programme in Java.

Ein Programmrahmen ist im Praktikums-git gegeben.

Gegeben sind:

- Klassen für die abstrakte Syntax.
- Parser, der Straightline-Programmdateien in abstrakte Syntax einliest.
- Implementierung von `ToString` als Beispiel eines Visitors.
- Für die Auswertung von Programmen soll ein `Eval`-Visitor implementiert werden.

Operationelle Semantik

Die operationelle Semantik legt fest, wie sich Programme bei der Ausführung verhalten sollen.

Wichtiges Prinzip: *Kompositionalität*
Das Verhalten eines Programmteils wird aus dem Verhalten seiner Bestandteile erklärt.

Beispiel: Auswertung von e_1+e_2

- Werte e_1 aus. Das Ergebnis ist eine Zahl i_1.
- Werte e_2 aus. Das Ergebnis ist eine Zahl i_2.
- Das Ergebnis der Auswertung ist die Zahl $i_1 + i_2$.

Beispiel: Auswertung von (s, e)

- Werte s aus. Dabei können Variablenwerte verändert werden.
- Werte e aus. Das Ergebnis ist eine Zahl i.
- Das Ergebnis der Auswertung ist die Zahl i.

Big-Step Reduktionsrelationen

Formalisierung der Auswertung durch Reduktionsrelationen.

- $s, \rho \Downarrow o, \rho'$: Wenn man die Anweisung s mit der Anfangsumgebung ρ (endliche Abbildung von Variablen auf Zahlen) ausführt, dann werden die Ausgabe in o (endlicher String) gemacht und am Ende ist die Umgebung ρ'.
- $e, \rho \Downarrow i, o, \rho'$: Wenn man den Ausdruck e in der Anfangsumgebung ρ auswertet, dann werden dabe die Ausgaben in o gemacht, der Ergebniswert ist i und am Ende ist die Umgebung ρ'.

Operationelle Semantik

Anweisungen

$$\frac{s_1, \rho_1 \Downarrow o_1, \rho_2 \qquad s_2, \rho_2 \Downarrow o_2, \rho_3}{s_1 ; s_2, \rho_1 \Downarrow o_1 o_2, \rho_3} \qquad \frac{e, \rho_1 \Downarrow i, o, \rho_2}{x := e, \rho_1 \Downarrow o, \rho_2[x := i]}$$

$$\frac{e_i, \rho_i \Downarrow v_i, o_i, \rho_{i+1} \text{ für } i = 1, \ldots, n}{\texttt{print}(e_1, \ldots, e_n), \rho_1 \Downarrow o_1 \ldots o_n v_1 \texttt{␣} v_2 \texttt{␣} \ldots \texttt{␣} v_n \texttt{\textbackslash n}, \rho_{n+1}}$$

Ausdrücke

$$\frac{e_1, \rho_1 \Downarrow i_1, o_1, \rho_2 \qquad e_2, \rho_2 \Downarrow i_2, o_2, \rho_3}{e_1 + e_2, \rho_1 \Downarrow i_1 + i_2, o_1 o_2, \rho_3}$$

$$\frac{s, \rho_1 \Downarrow o_1, \rho_2 \qquad e, \rho_2 \Downarrow i, o_2, \rho_3}{(s, e), \rho_1 \Downarrow i, o_1 o_2, \rho_3}$$

Programmieraufgabe: Straightline Interpreter

Aufgabe für heute: Implementieren Sie einen Interpreter für Straightline-Programme in Java.

Der Interpreter dient als Aufwärmübung. Besonders nützlich für den eigentlichen Compiler sind:

- **Datenrepräsentation:** Modellierung der abstrakten Syntax mittels einer Klassenhierarchie (*Composite Pattern*).
- **Programmiertechnik:** Iteration über diese Datenstruktur mittels eines *Visitor Pattern*.