Praktikum Compilerbau

Wintersemester 2018/19

Ulrich Schöpp

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

Übersicht

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

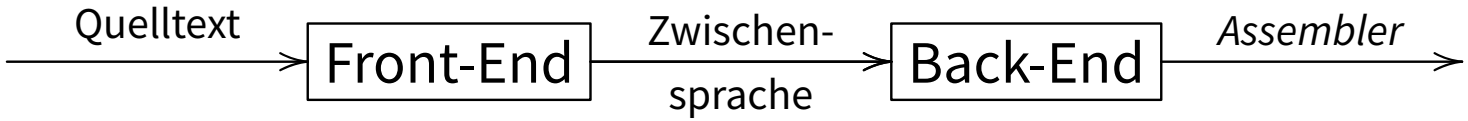
Übersetzung von Quellcode (MiniJava-Quelltext) in Maschinensprache (Assembler).

- 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
  - …

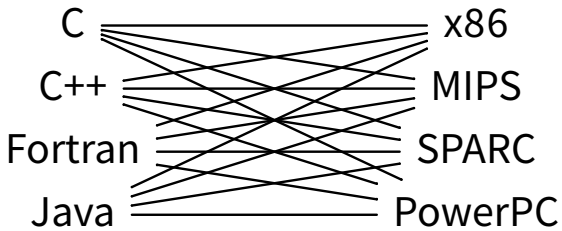
Aufbau eines Compilers

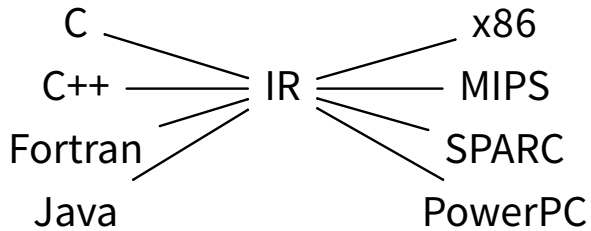
Compiler bestehen üblicherweise aus *Front-End* und *Back-End*.



Zwischensprache(n)

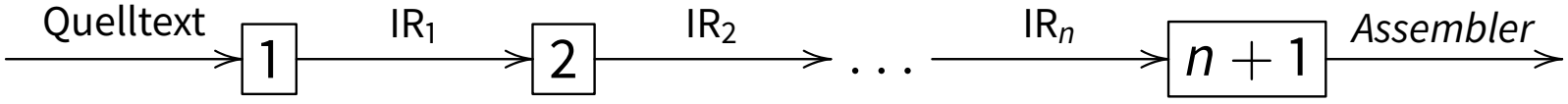
- abstrakter Datentyp, leichter zu behandeln als Strings
- weniger komplex als Eingabesprache $\Rightarrow$ Transformationen und Optimierungen leichter implementierbar
- Zusammenfassung ähnlicher Fälle, z.B. Kompilation von `for`- und `while`-Schleifen ähnlich.
- Kombination mehrerer Quellsprachen und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

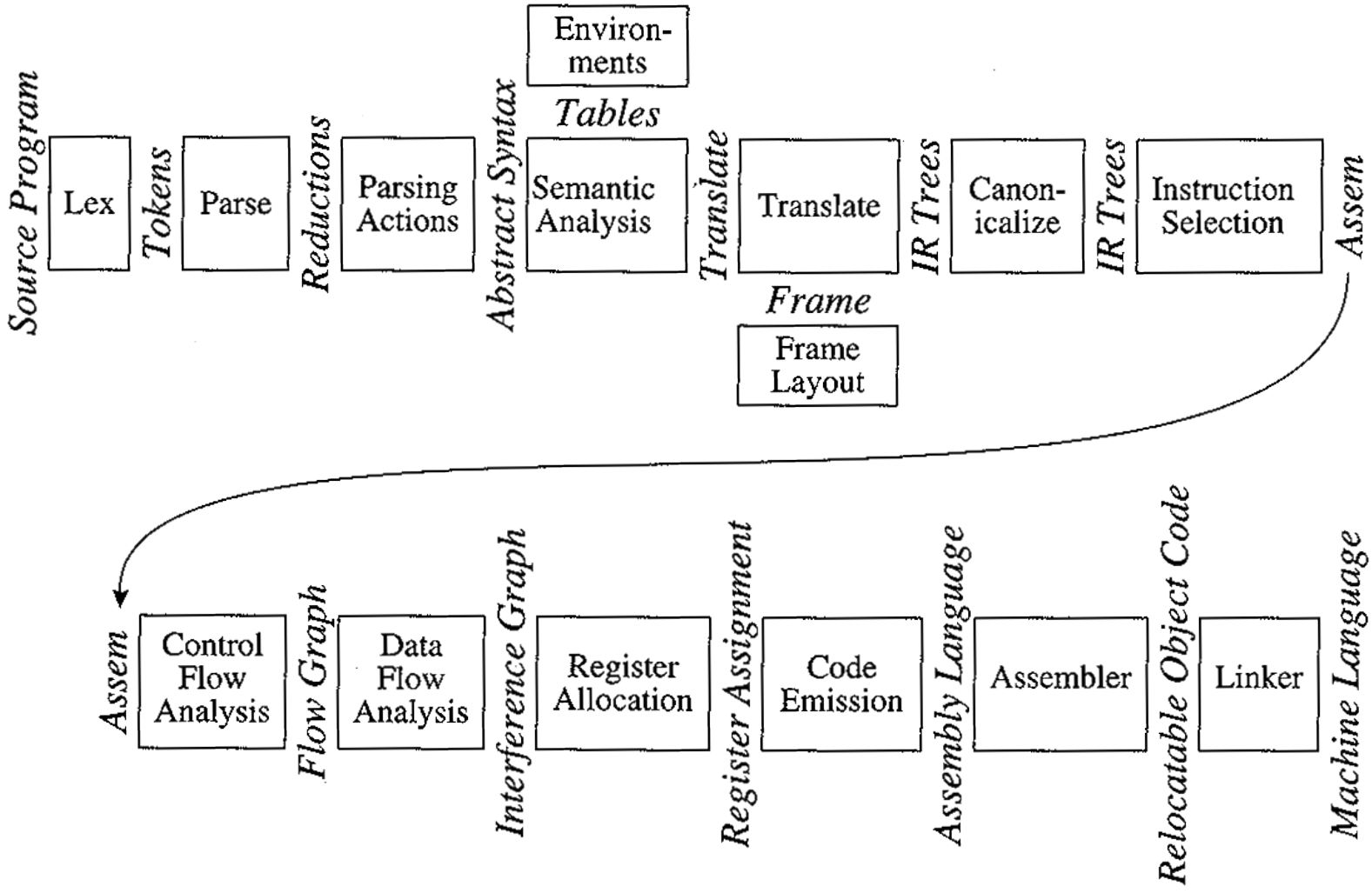
Die Aufteilung in eine Folge einfacher Transformationen hilft, die Korrektheit sicherzustellen.

- 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

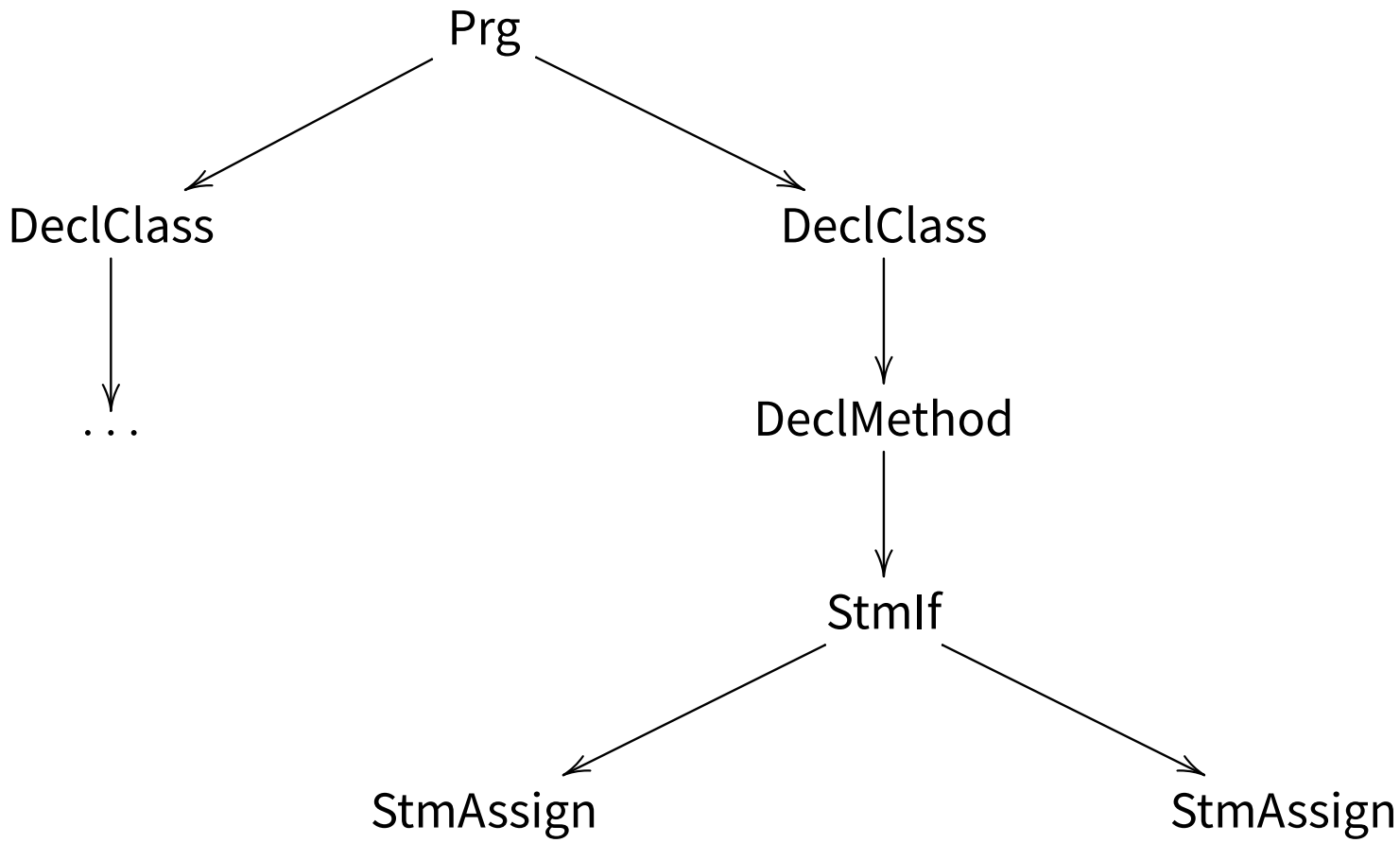
```
"class Factorial{
  public static void main(String[] a){
    System.out.println(new Fac().ComputeFac(10));
  }
}

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



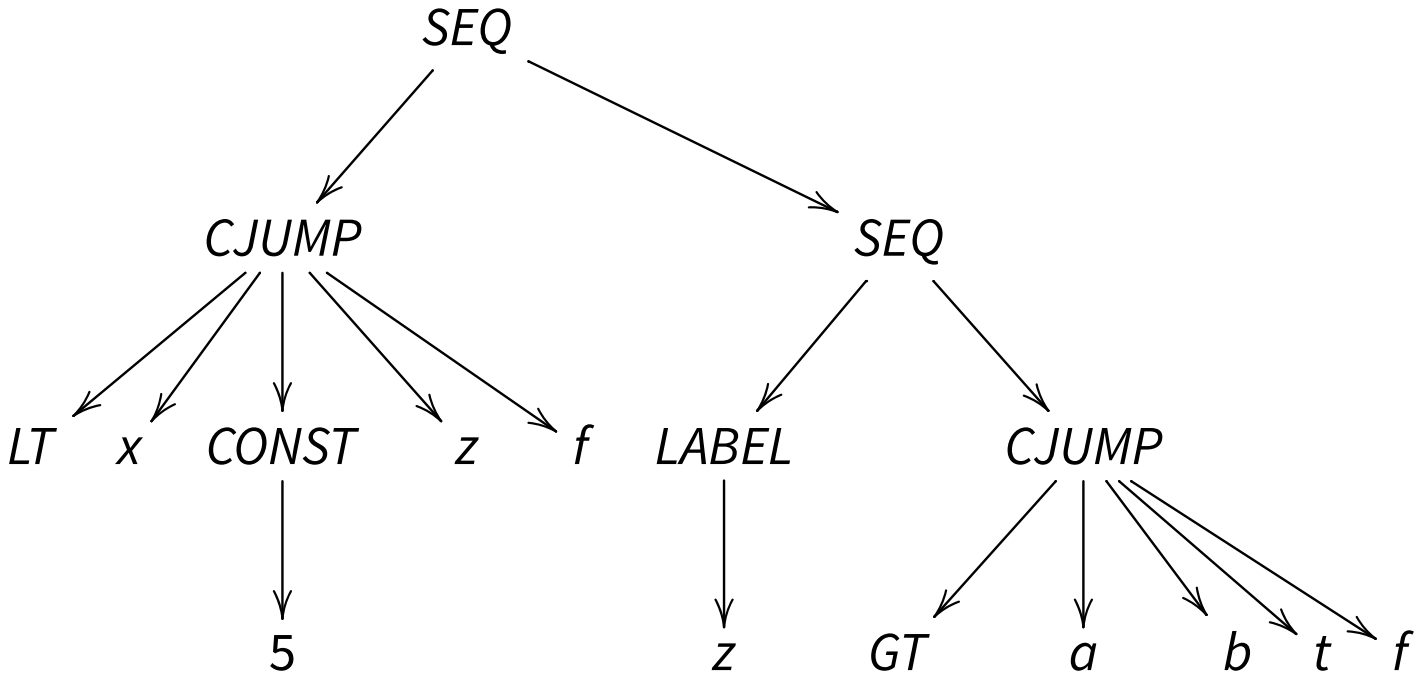
Zwischensprachen

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.

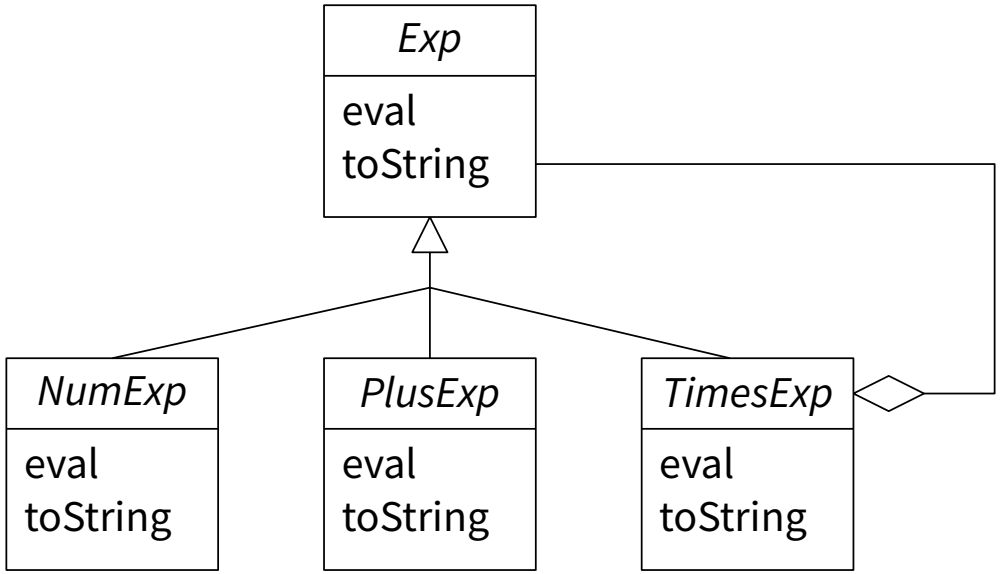
- Fördert eine *funktionsorientierte Sichtweise*: der Code für eine Operation auf einer gesamten Datenstruktur wird in einem Modul zusammengefasst.
- Fördert die Erweiterbarkeit von Programmen.
  - Implementierung neuer Operationen ohne Veränderung der Datenstruktur.
  - Beim Hinzufügen einer neuen Operation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

Visitor Pattern

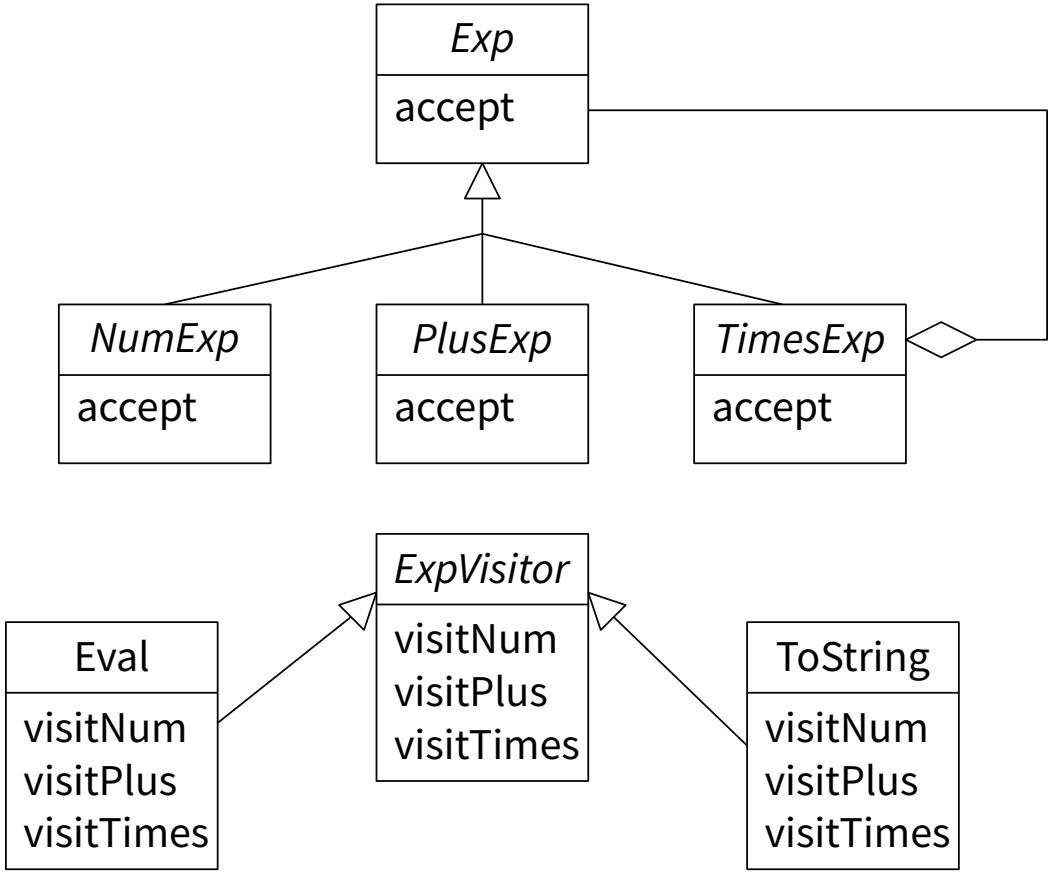
Idee:

- 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

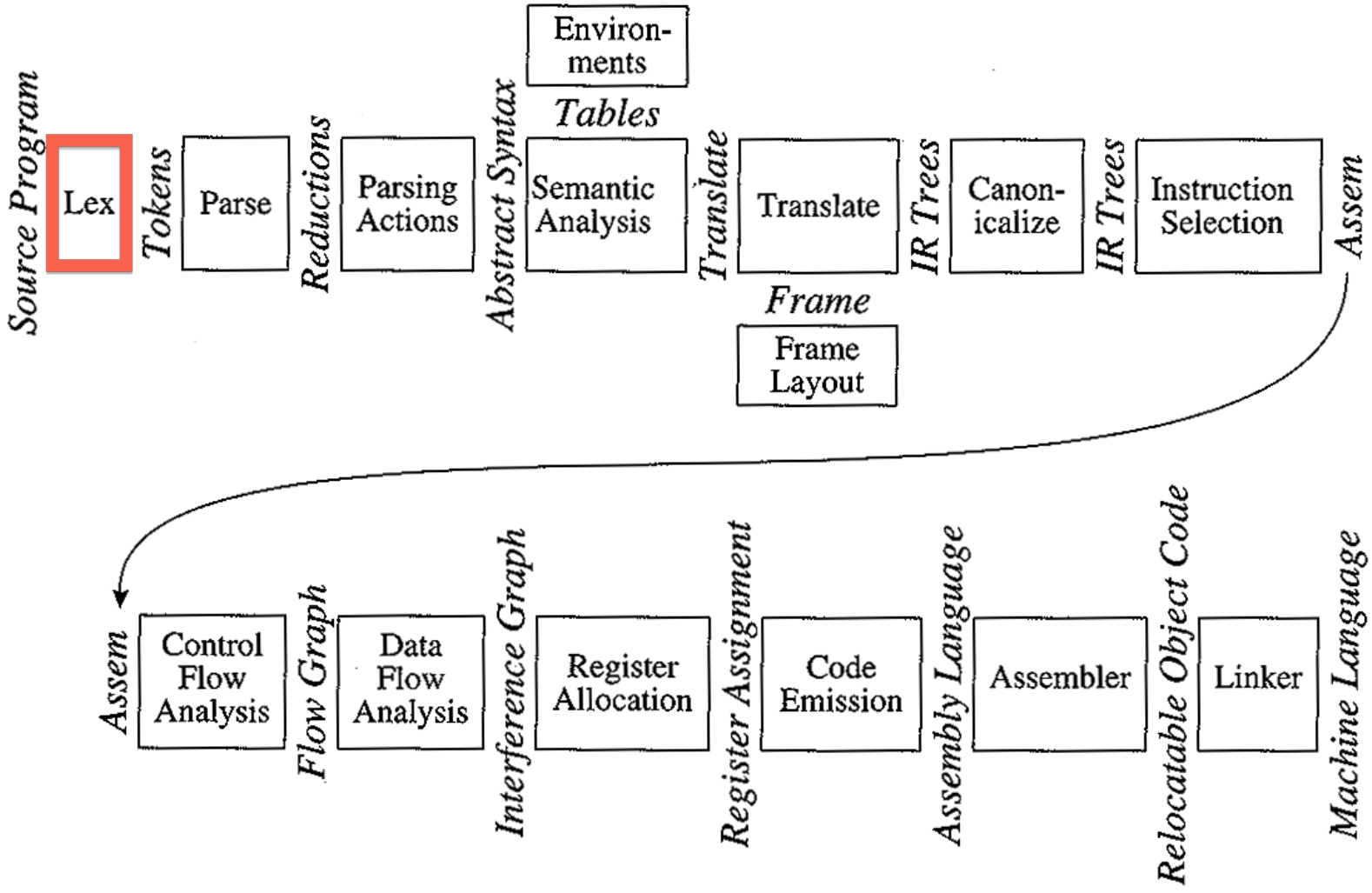
Aufgabe für heute: Implementieren Sie einen Interpreter für Straightline-Programme in Java.

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*.

Lexikalische Analyse

Lexikalische Analyse



Lexikalische Analyse

- Erste Phase der Kompilierung
- Die String-Eingabe (Quelltext) wird in eine Folge von *Tokens* umgewandelt.
- Leerzeichen und Kommentare werden dabei ignoriert.
- Die Tokens entsprechen den *Terminalsymbolen* der Grammatik der Sprache, können jedoch auch mit Werten versehen sein.

Was sind Tokens?

Beispiele:

"`if`"	$\rightarrow$	IF
"`!=`"	$\rightarrow$	NEQ
"`(`"	$\rightarrow$	LPAREN
"`)`"	$\rightarrow$	RPAREN
"`foo`"	$\rightarrow$	ID("foo")
"`73`"	$\rightarrow$	INT(73)
"`66.1`"	$\rightarrow$	REAL(66.1)

Keine Beispiele:

"`/* bla */`"	(Kommentar)
"`#define NUM 5`"	(Präprozessordirektive)
"NUM"	(Makro)

Beispiel für die lexikalische Analyse

```
void match0(char *s) /* find a zero */
{ if (!strncmp (s, ''0.0'', 3))
    return 0.;
}
```

VOID ID(”match0”) LPAREN CHAR STAR
ID(s) RPAREN LBRACE IF LPAREN
BANG ID(”strncmp”) LPAREN ID(”s”)
COMMA STRING(”0.0”) COMMA INT(3)
RPAREN RPAREN RETURN REAL(0.0)
SEMI RBRACE EOF

Implementierung eines Lexers

Schreibe einen Lexer von Hand

- nicht schwer, aber fehleranfällig und aufwändig

Benutze einen Lexer-Generator

- schnelle Implementierung eines Lexers
- liefert korrekten und effizienten Lexer
- Benutzung des Lexer-Generators benötigt Einarbeitungszeit

Lexer-Generatoren

Tokens werden durch reguläre Ausdrücke spezifiziert.

`(`	$\rightarrow$	LPAREN
`print`	$\rightarrow$	PRINT
digit digit*	$\rightarrow$	INT(ConvertToInt(yytext()))
letter(*letter* + *digit* + {_})*	$\rightarrow$	ID(yytext())
...		

wobei *digit* $= \{$`0`, ..., `9`$\}$ und *letter* $= \{$`a`, ..., `z`, `A`, ..., `Z`$\}$.

- Die regulären Ausdrücke werden vom Lexer-Generator in einen endlichen Automaten umgewandelt, der zum Lesen der eigentlichen Eingabe benutzt wird.
- Automatische Lexergeneratoren wie flex, JFlex, alex, Ocamllex wandeln die Spezifikation in ein Lexer-Programm (als Quelltext) um, das einen entsprechenden Automaten simuliert.

Auflösung von Mehrdeutigkeiten

I.A. gibt es mehrere Möglichkeiten, eine Folge in Tokens zu zerlegen. Lexergeneratoren verwenden die folgenden zwei Regeln:

- **Längste Übereinstimmung:** Das längste Präfix, das zu irgendeinem der regulären Ausdrücke passt, wird das nächste Token.
- **Regelpriorität:** Wenn das nicht hilft, kommt die weiter oben stehende Regel zum Zug. Die Reihenfolge der Regeln spielt also eine Rolle.

`print0`	:	ID(”print0”) *nicht* PRINT INT(0)
`print`	:	PRINT *nicht* ID(”print”)

Funktionsweise von Lexer-Generatoren

Idee: Implementiere die lexikalische Analyse durch Abarbeitung der Eingabe mit einem endlichen Automaten (DFA). Der Automat kann ein Token am Anfang der Eingabe erkennen. $\Rightarrow$ Zerlegung der Eingabe durch wiederholtes Ansetzen des Automaten.

- Jeder reguläre Ausdruck wird erst in einen NFA umgewandelt.
- Aus den einzelnen NFAs wird ein einziger NFA konstruiert, dessen Sprache die Vereinigung der Sprachen der NFAs ist.
- Mit der Potenzmengenkonstuktion und Minimierung wird der NFA in einen minimalen DFA umgewandelt.
- Jedem regulären Ausdruck ist ein Token zugeordnet. Die Endzustände des DFA für die Vereinigung werden mit den Tokens beschriftet, die beim Erreichen dieses Zustands gelesen worden sind. Mehrdeutigkeiten werden durch die Regelpriorität aufgelöst.

Funktionsweise von Lexer-Generatoren

Lexikalische Analyse

- Beginne mit dem Startzustand des Automaten und verfolge einen Lauf des Automaten.
- Zur Implementierung von „längste Übereinstimmung“ merkt man sich stets die Eingabeposition, bei der das letzte Mal ein Endzustand erreicht wurde.
- Kommt man in eine Sackgasse, d.h. kann das Wort nicht mehr vom Automaten akzeptiert werden, so bestimmt der letzte erreichte Endzustand das Token.
  Man fängt dann an der folgenden Position wieder im Startzustand an, um das nächste Token zu lesen.

Lexergeneratoren

Ein Lexergenerator erzeugt aus einer Spezifikations-Datei einen Lexer in Form einer Java-Klasse (bzw. Haskell-Datei, ...) mit einer Methode/Funktion `next_token()`.

Jeder Aufruf von `next_token()` liefert das „Ergebnis“ zurück, welches zum nächsten verarbeiteten Teilwortes der Eingabe gehört. Normalerweise besteht dieses „Ergebnis“ aus dem Namen des Tokens und seinem Wert.

Die Spezifikations-Datei enthält Regeln der Form

$$\textit{regexp} \;\rightarrow\; \{\textit{code}\}$$

wobei *code* Java-Code (oder Haskell-Code, ...) ist, der das „Ergebnis“ berechnet. Dieses Codefragment kann sich auf den (durch *regexp*) verarbeiteten Text durch spezielle Funktionen und Variablen beziehen, z.B.: $\mathrm{yytext}()$ und yypos. Siehe Beispiel + Doku.

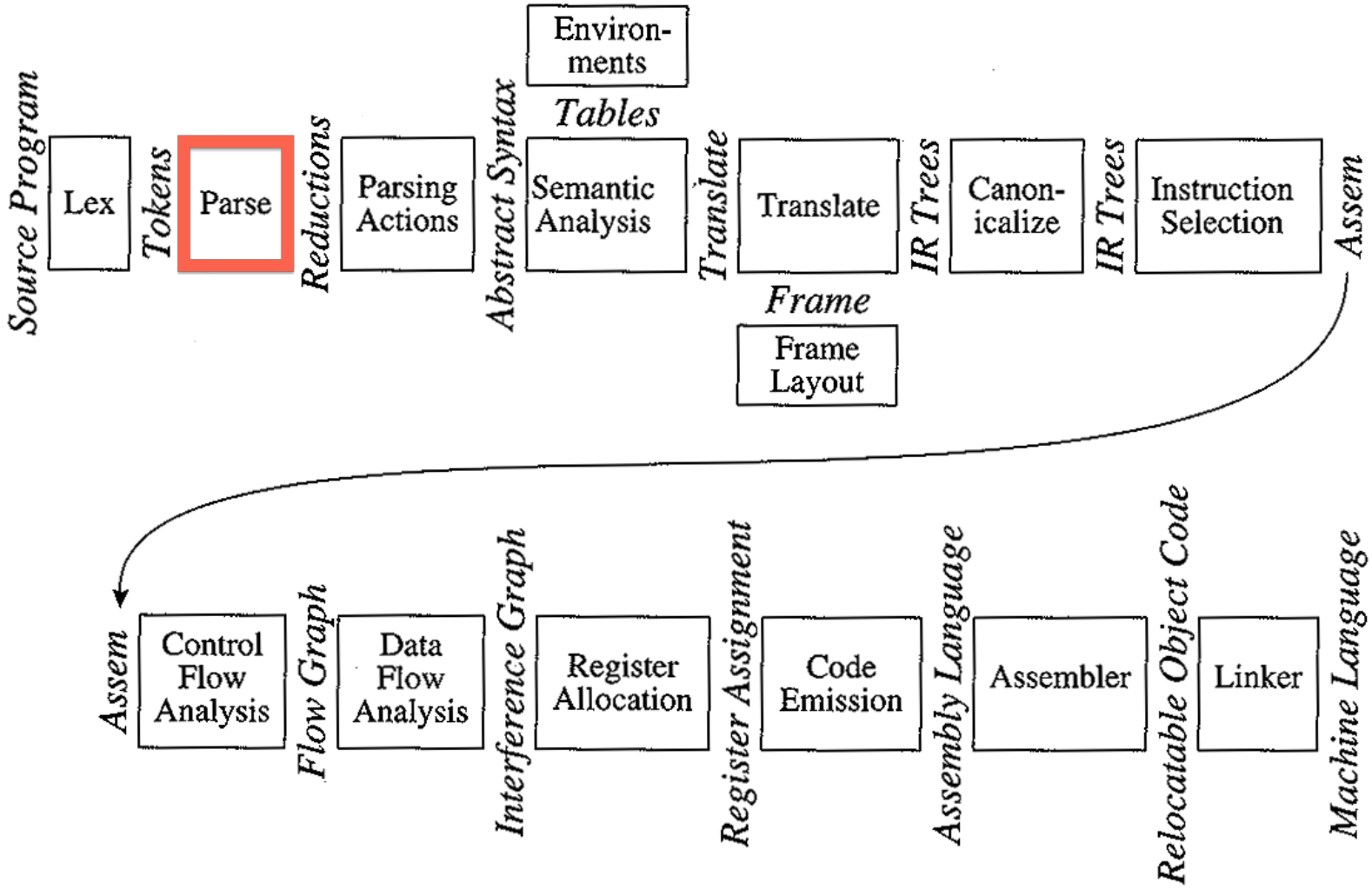
Zustände

In der Spezifikations-Datei können auch Zustände angegeben werden:

- Regeln können mit Zuständen beschriftet werden; sie dürfen dann nur in diesem Zustand angewandt werden.
- Im *code*-Abschnitt kann durch spezielle Befehle wie $\mathrm{yybegin}()$ der Zustand gewechselt werden.
- Man verwendet das, um geschachtelte Kommentare und Stringliterale zu verarbeiten:
  - `/*` führt in einen „Kommentarzustand“.
  - Kommt `/*` im Kommentarzustand vor, so inkrementiere einen Tiefenzähler.
  - `*/` dekrementiert den Tiefenzähler oder führt wieder in den „Normalzustand“ zurück.
  - Ebenso führt " (Anführungszeichen) zu einem „Stringzustand“…

Syntaxanalyse

Syntaxanalyse



Syntaxanalyse

- Zweite Phase der Kompilierung
- Erkennen einer Folge von Tokens als eine Ableitung einer kontextfreien Grammatik
  - Überprüfung, ob Eingabe syntaktisch korrekt ist in Bezug auf die Programmiersprache
  - Umwandlung in einen Syntaxbaum (abstrakte Syntax, *abstract syntax tree*, AST).

Parsertechniken und -generatoren

- Laufzeit soll linear in der Eingabegröße sein:
  - Einschränkung auf spezielle Grammatiken (LL(1), LR(1), LALR(1)), für die effiziente Analysealgorithmen verfügbar sind.
- Diese Algorithmen (*Parser*) können automatisch aus einer formalen Grammatik erzeugt werden:
  - Parsergeneratoren: z.B. yacc, bison, ML-yacc, JavaCUP, JavaCC, ANTLR.
- Hier vorgestellt:
  - Parsertechniken LL(1), LR(1), LALR(1)
  - Parsergenerator JavaCUP

Wiederholung: Kontextfreie Grammatiken

- Kontextfreie Grammatik: $G = (\Sigma, V, P, S)$
  - $a, b, c, \ldots \in \Sigma$: Terminalsymbole (Eingabesymbole, Tokens)
  - $X, Y, Z, \ldots \in V$: Nichtterminalsymbole (Variablen)
  - P: Produktionen (Regeln) der Form $X \to \gamma$
  - $S \in V$: Startsymbol der Grammatik
- Symbolfolgen:
  - $u, v, w \in \Sigma^*$: Wörter (Folge von Nichtterminalsymbolen)
  - ϵ: leeres Wort
  - $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$: Satzform (Folge von Terminal- und Nichtterminalsymbolen)
- Ableitungen:
  - Ableitungsrelation: $\alpha X \beta \Rightarrow \alpha\gamma\beta$ falls $X \to \gamma \in P$
  - Links-/Rechtsableitung: $wX\beta \Rightarrow_{\mathbf{lm}} w\gamma\beta$ bzw. $\alpha Xw \Rightarrow_{\mathbf{rm}} \alpha\gamma w$
  - Sprache der Grammatik: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$
  - Ableitungsbaum: Symbole aus γ als Unterknoten von X

LL(1)-Syntaxanalyse

Beispiel-Grammatik:

1. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
2. $S \rightarrow \text{begin } S\ L$
3. $S \rightarrow \text{print } E$
4. $L \rightarrow \text{end}$
5. $L \rightarrow ;\ S\ L$
6. $E \rightarrow \text{num} = \text{num}$

Ein Parser für diese Grammatik kann mit der Methode des rekursiven Abstiegs (*recursive descent*) gewonnen werden: Für jedes Nichtterminalsymbol gibt es eine Funktion, die gegen dieses analysiert.

In C

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();}}
void L(void) {switch(tok) {
        case END:  eat(END); break;
        case SEMI: eat(SEMI); S(); L(); break;
        default:   error();}}
void E(void) { eat(NUM); eat(EQ); eat(NUM); }
```

Manchmal funktioniert das nicht:

$S \to E\,\$$

$E \to E + T$
$E \to E - T$
$E \to T$

$T \to T * F$
$T \to T \,/\, F$
$T \to F$

$F \to \mathrm{id}$
$F \to \mathrm{num}$
$F \to (\ E\ )$

```
void S(void) {  E(); eat(EOF);  }
void E(void) {switch(tok) {
        case ?: E(); eat(PLUS); T(); break;
        case ?: E(); eat(MINUS); T(); break;
        case ?: T(); break;
        default:   error(); }}
void T(void) {switch(tok) {
        case ?: T(); eat(TIMES); F(); break;
        case ?: T(); eat(DIV); F(); break;
        case ?: F(); break;
        default:   error();  }}
```

LL(1)-Syntaxanalyse

- Eine Grammatik heißt LL(1), wenn ein Parse-Algorithmus basierend auf dem Prinzip des rekursiven Abstiegs für sie existiert.
- Der Parser muss anhand des nächsten zu lesenden Tokens (und der erwarteten linken Seite) in der Lage sein zu entscheiden, welche Produktion zu wählen ist.
- Die zu wählende Produktion kann zur Optimierung in einer Parser-Tabelle abgespeichert werden.
- Die LL(1) Sprachen sind genau die Sprachen, die von einem deterministischen Kellerautomaten mit einem Zustand akzeptiert werden.

LL(1)-Parser – Beispiel

Grammatik:
(1) S→F
(2) S→(S+F)
(3) F→a

Eingabe: (a+a)

	(	)	a	+	$
S	2		1		
F			3		

Stack	Input	Aktion
S $	(a + a) $	apply (2) S→(S+F)
(S + F) $	(a + a) $	match '('
S + F) $	a + a) $	apply (1) S→F
F + F) $	a + a) $	apply (3) F→a
a + F) $	a + a) $	match 'a'
+ F) $	+ a) $	match '+'
F) $	a) $	apply (3) F→a
a) $	a) $	apply (3) F→a
) $	) $	match ')'
$	$	match '$' = accept

LL(1)-Syntaxanalyse

- Die Eingabe wird von links nach rechts verarbeitet, dabei wird eine Linksableitung vollzogen, und die Regelauswahl wird anhand des ersten Symbols der verbleibenden Eingabe und des obersten Stacksymbols entschieden.
  - LL(1): **l**eft-to-right parsing, **l**eft-most derivation, 1 token lookahead
- Verarbeitung beginnt beim Startsymbol (Top-down-Ansatz).
- Welche Produktion soll gewählt werden, bzw. wie wird die Tabelle erzeugt?
  - Ansatz: Wenn das nächste Eingabesymbol a ist und X auf Stack liegt (also erwartet wird), kommt diejenige Produktion für X infrage, die zu einer Ableitung mit a an der ersten Stelle führt.

Die First- und Follow-Mengen

- FIRST(γ) ist die Menge aller Terminalsymbole, die als Anfänge von aus γ abgeleiteten Wörtern auftreten:
  $\text{FIRST}(\gamma) = \{a \mid \exists w.\gamma \Rightarrow^* aw\}$
- FOLLOW(X) ist die Menge der Terminalsymbole, die unmittelbar auf X folgen können:
  $\text{FOLLOW}(X) = \{a \mid \exists \alpha, \beta.S \Rightarrow^* \alpha X a \beta\}$.
- $\text{nullable}(\gamma)$ bedeutet, dass das leere Wort aus γ abgeleitet werden kann:
  $\text{nullable}(\gamma) \Longleftrightarrow \gamma \Rightarrow^* \epsilon$.

Berechnung der First- und Follow-Mengen

Es genügt $\mathrm{FIRST}(X)$ und $\mathrm{nullable}(X)$ für Terminal- und Nichtterminalsymbole zu berechnen. Die Werte auf Satzformen sind davon eindeutig bestimmt.

- $\mathrm{FIRST}(\epsilon) = \emptyset$, $\mathrm{FIRST}(a\gamma) = \{a\}$, $\mathrm{FIRST}(X\gamma) =$ **if** $\mathrm{nullable}(X)$ **then** $\mathrm{FIRST}(X) \cup \mathrm{FIRST}(\gamma)$ **else** $\mathrm{FIRST}(X)$.
- $\mathrm{nullable}(\epsilon) = \mathit{true}$, $\mathrm{nullable}(a\gamma) = \mathit{false}$, $\mathrm{nullable}(X\gamma) = \mathrm{nullable}(X) \wedge \mathrm{nullable}(\gamma)$.

Berechnung der First- und Follow-Mengen

Die First- und Follow-Mengen, sowie das Nullable-Prädikat, lassen sich wie folgt iterativ berechnen:

Setze $\mathrm{nullable}(a) := \mathit{false}$ und $\mathrm{FIRST}(a) = \{a\}$ für alle Terminalsymbole a.
Setze $\mathrm{nullable}(X) := \mathit{false}$ und $\mathrm{FIRST}(X) = \mathrm{FOLLOW}(X) = \emptyset$ für alle Nichtterminalsymbole X.

Gehe dann jede Produktion $X \to \gamma$ durch und ändere die Werte wie folgt, solange bis sich nichts mehr ändert:

- Wenn $\mathrm{nullable}(\gamma)$, dann setze $\mathrm{nullable}(X) := \mathit{true}$.
- $\mathrm{FIRST}(X) := \mathrm{FIRST}(X) \cup \mathrm{FIRST}(\gamma)$.
- Für alle α, Y und β, so dass $\gamma = \alpha Y \beta$ und $\mathrm{nullable}(\beta)$ gilt, setze $\mathrm{FOLLOW}(Y) := \mathrm{FOLLOW}(Y) \cup \mathrm{FOLLOW}(X)$.
- Für alle α, Y, β, δ sowie $u \in \Sigma \cup V$, so dass $\gamma = \alpha Y \beta u \delta$ und $\mathrm{nullable}(\beta)$ gilt, setze $\mathrm{FOLLOW}(Y) := \mathrm{FOLLOW}(Y) \cup \mathrm{FIRST}(u)$.

Konstruktion des Parsers

Soll die Eingabe gegen X geparst werden (d.h. wird X erwartet) und ist das nächste Token a, so kommt die Produktion $X \to \gamma$ in Frage, wenn

- $a \in \text{FIRST}(\gamma)$ oder
- $\text{nullable}(\gamma)$ und $a \in \text{FOLLOW}(X)$.

Die in Frage kommenden Produktionen werden in die LL-Tabelle in Zeile X und Spalte a geschrieben.

Kommen aufgrund dieser Regeln mehrere Produktionen in Frage, so ist die Grammatik nicht LL(1). Ansonsten spezifiziert die Tabelle den LL(1)-Parser, der die Grammatik erkennt.

Von LL(1) zu LL(k)

Der LL(1)-Parser entscheidet aufgrund des nächsten Tokens und der erwarteten linken Seite, welche Produktion zu wählen ist. Bei LL(k) bezieht man in diese Entscheidung die k nächsten Token mit ein.

Dementsprechend bestehen die First- und Follow-Mengen aus Wörtern der Länge k und werden dadurch recht groß. Durch Einschränkung der k-weiten Vorausschau auf bestimmte benutzerspezifizierte Stellen, lässt sich der Aufwand beherrschbar halten (z.B. bei ANTLR und JavaCC).

Wenn die Grammatik nicht LL(k) ist

Wenn eine Grammatik nicht LL(k) ist, gibt es dennoch Möglichkeiten, LL-Parser einzusetzen.

- Produktionsauswahl:
  - Kommen mehrere Produktionen infrage, kann man sich entweder auf eine festlegen, oder alle durchprobieren.
  - Damit ist der Parser aber nicht mehr unbedingt „vollständig“ in Bezug auf die Grammatik, d.h. es werden evtl. nicht alle gültigen Ableitungen erkannt.
  - Außerdem muss darauf geachtet werden, dass keine unendliche Rekursion stattfindet (bei linksrekursiven Grammatiken).
- Umformung der Grammatik:
  - Die Grammatik kann u.U. in eine LL(k)-Grammatik umgeformt werden, die die gleiche Sprache beschreibt.
  - Beispiel: Elimination von Linksrekursion.
  - Die Ableitungsbäume ändern sich daduch natürlich.

Zusammenfassung LL(1)-Syntaxanalyse

- **Top-down parsing:** Das nächste Eingabesymbol und die erwartete linke Seite entscheiden, welche Produktion anzuwenden ist.
- Eine **LL(1)-Grammatik** liegt vor, wenn diese Entscheidung in eindeutiger Weise möglich ist. Die Entscheidung lässt sich dann mithilfe der First- und Follow-Mengen automatisieren.
- Der große **Vorteil** des LL(1)-Parsing ist die leichte Implementierbarkeit: auch ohne Parsergenerator kann ein rekursiver LL(1)-Parser leicht von Hand geschrieben werden, denn die Tabelle kann relativ einfach berechnet werden.
- Der **Nachteil** ist, dass die Grammatik vieler Programmiersprachen (auch MiniJava) nicht LL(k) ist.

Parsergenerator JavaCUP

- Parsergenerator: Grammatikspezifikation $\rightarrow$ Parser-Quellcode
- JavaCUP generiert LALR(1)-Parser als Java-Code
  (LALR-/LR-Parser werden nächstes Mal vorgestellt)
- Grammatikspezifikationen sind in die folgenden Abschnitte gegliedert:
  Benutzerdeklarationen (z.B. package statements, Hilfsfunktionen)
  Parserdeklarationen (z.B. Mengen von Grammatiksymbolen)
  Produktionen
- Beispiel für eine Produktion:
  `exp ::= exp PLUS exp` `{:` $Semantische\ Aktion$ `:}`
  Die „semantische Aktion“ (Java-Code) wird ausgeführt, wenn die entsprechende Regel „feuert“. Üblicherweise wird dabei die AST-Repräsentation zusammengesetzt.

Beispielgrammatik

Stm	$\rightarrow$	*Stm*;*Stm*	(CompoundStm)
Stm	$\rightarrow$	`id :=`*Exp*	(AssignStm)
Stm	$\rightarrow$	`print (`*ExpList*`)`	(PrintStm)
Exp	$\rightarrow$	`id`	(IdExp)
Exp	$\rightarrow$	`num`	(NumExp)
Exp	$\rightarrow$	*Exp BinOp Exp*	(OpExp)
Exp	$\rightarrow$	`(`*Stm* `,` *Exp*`)`	(EseqExp)
ExpList	$\rightarrow$	*Exp* `,` *ExpList*	(PairExpList)
ExpList	$\rightarrow$	*Exp*	(LastExpList)
BinOp	$\rightarrow$	`+`	(Plus)
BinOp	$\rightarrow$	`-`	(Minus)
BinOp	$\rightarrow$	`*`	(Times)
BinOp	$\rightarrow$	`/`	(Div)

Implementierung in JavaCUP

```
package straightline;
import java_cup.runtime.*;

parser code  {: <Hilfsfunktionen für den Parser> :}

terminal String ID; terminal Integer INT;
terminal COMMA, SEMI, LPAREN, RPAREN, PLUS, MINUS,
         TIMES, DIVIDE, ASSIGN, PRINT;

non terminal exp;
non terminal explist;
non terminal stm;

precedence left SEMI;
precedence nonassoc ASSIGN;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

Implementierung in JavaCUP

```
start with stm;

exp ::=  exp TIMES exp {: :}
       | exp DIVIDE exp {: :}
       | exp PLUS exp {: :}
       | exp MINUS exp {: :}
       | INT {: :}
       | ID {: :}
       | LPAREN stm COMMA exp RPAREN {: :}
       | LPAREN exp RPAREN {: :}
       ;

explist ::=  exp {: :}
           | exp COMMA explist {: :}
           ;

stm ::=  stm SEMI stm {: :}
       | PRINT LPAREN explist RPAREN {: :}
       | ID ASSIGN exp {: :}
       ;
```

Präzedenzdirektiven

Die obige Grammatik ist mehrdeutig. Präzedenzdirektiven werden hier zur Auflösung der Konflikte verwendet.

Die Direktive

```
precedence left SEMI;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

besagt, dass `TIMES, DIVIDE` stärker als `PLUS, MINUS` binden, welche wiederum stärker als `SEMI`. Alle Operatoren assoziieren nach links.

Also wird $s1; s2; s3, x + y + z * w$ zu $(s1; (s2; s3)), ((x + y) + (z * w))$.

Verwendung von JavaCUP

JavaCUP wird als Kommandozeilentool verwendet.

```
java -jar java-cup-11a.jar --help
```

Der Aufruf

```
java -jar java-cup-11a.jar Parser.cup
```

- liest die Grammatik von `Parser.cup`,
- generiert eine Klasse `sym.java` mit Konstanten für alle Terminalsymbole,
- generiert eine Klasse `parser.java` mit dem eigentlichen Parser.

Die Namen von `sym` und `parser` können durch Kommandozeilenparameter geändert werden.

```
java -jar java-cup-11a.jar -parser Parser -symbols Parser_sym Parser.cup
```

Ihre heutige Aufgabe

Schreiben eines MiniJava-Lexers und -Parsers:

- MiniJava-Grammatik ist auf der Vorlesungsseite verlinkt.
- Verwenden Sie Lexer und Parser für StraightLine-Programme als Ausgangspunkt.
- Passen Sie den Lexer entsprechend der MiniJava-Syntax an.
- Passen Sie die Grammatik der MiniJava-Grammatik an.
- Parser soll zunächst nur akzeptierend sein:
  - keine semantischen Aktionen, keine AST-Klassenhierarchie
  - Operatorpräzedenzen wie in Java
  - Probleme wie Shift/Reduce-Konflikte können noch nicht behoben werden: Detaillierte Erklärung in der nächsten Sitzung.