

Praktikum Compilerbau

Wintersemester 2018/19

Ulrich Schöpp

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

Übersicht

Organisatorisches

Einführung

Lexikalische Analyse

Syntaxanalyse

Semantische Analyse

Übersetzung in Zwischencode

Überblick

Aktivierungssätze (Frames)

Kanonisierung

Instruktionsauswahl

Optimierungsmöglichkeiten bei der Instruktionsauswahl

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 Maschinsprache (Assembler).

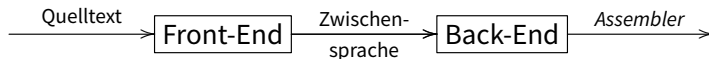
- Überprüfung, ob der Eingabetext ein korrektes MiniJava Programm ist.
 - Lexikalische Analyse und Syntaxanalyse
 - Semantische Analyse (Typüberprüfung und Sichtbarkeitsbereiche)

Ausgabe von informativen Fehlermeldungen bei inkorrektter Eingabe.

- Übersetzung in Maschinsprache
 - feste Anzahl von Maschinenregistern, wenige einfache Instruktionen, Kontrollfluss nur durch Sprünge, direkter Speicherzugriff
 - effizienter, kompakter Code
 - ...

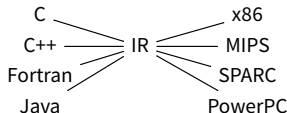
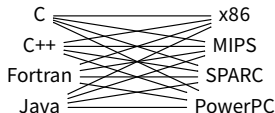
Aufbau eines Compilers

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



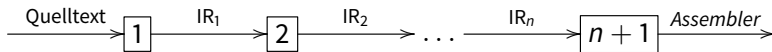
Zwischensprache(n)

- 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.

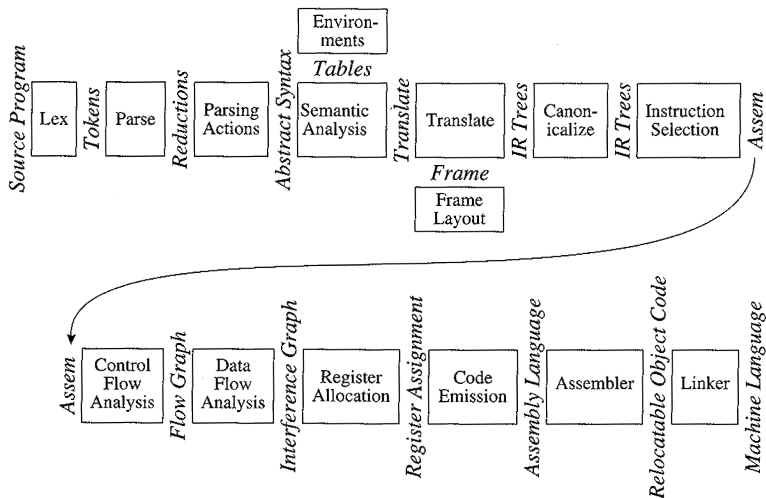
Die Aufteilung in eine Folge einfacher Transformationen hilft, die Korrektheit sicherzustellen.

- 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



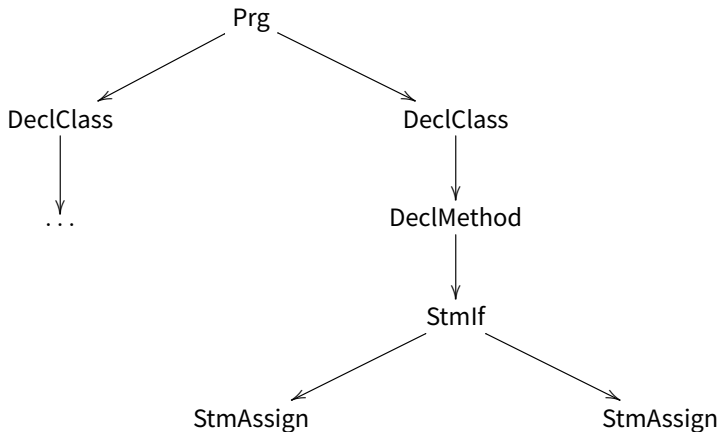
Zwischensprachen

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



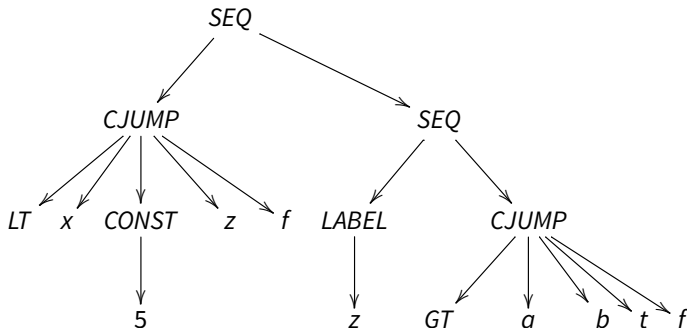
Zwischensprachen

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

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

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.

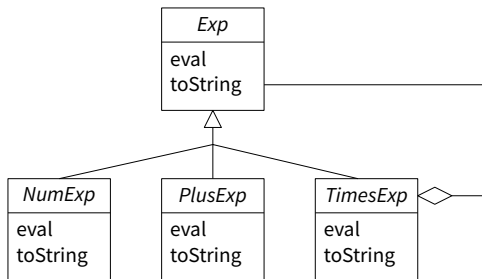
- Fördert eine *funktionsorientierte Sichtweise*: der Code für eine Operation auf einer gesamten Datenstruktur wird in einem Modul zusammengefasst.
- Fördert die Erweiterbarkeit von Programmen.
 - Implementierung neuer Operationen ohne Veränderung der Datenstruktur.
 - Beim Hinzufügen einer neuen Operation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

Visitor Pattern

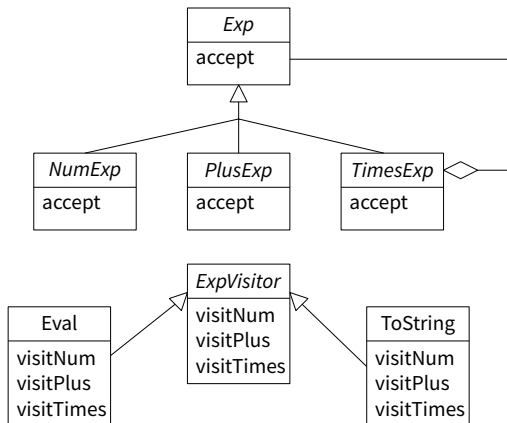
Idee:

- 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.

⇒ 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:

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

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 Programnteils 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 dabei die Ausgaben in o gemacht, der Ergebniswert ist i und am Ende ist die Umgebung ρ' .

Operationelle Semantik

Anweisungen

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

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

Ausdrücke

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

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

Programmieraufgabe: Straightline Interpreter

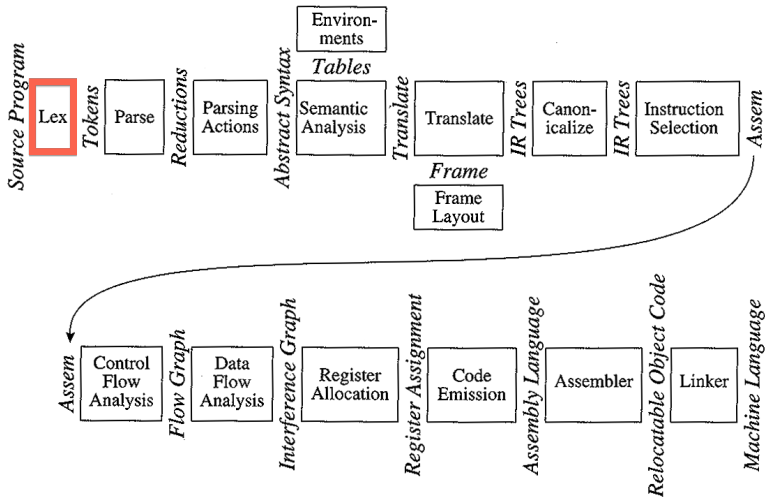
Aufgabe für heute: Implementieren Sie einen Interpreter für Straightline-Programme in Java.

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"	→	IF
"!="	→	NEQ
"("	→	LPAREN
")"	→	RPAREN
"foo"	→	ID("foo")
"73"	→	INT(73)
"66.1"	→	REAL(66.1)

Keine Beispiele:

"/* bla */"	(Kommentar)
"#define NUM 5"	(Präprozessor Direktive)
"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.

(→	LPAREN
print	→	PRINT
<i>digit digit*</i>	→	INT(ConvertToInt(yytext()))
<i>letter(letter + digit + { _ })*</i>	→	ID(yytext())

...

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

- 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 Potenzmengenkonstruktion 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

$$regex \rightarrow \{code\}$$

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

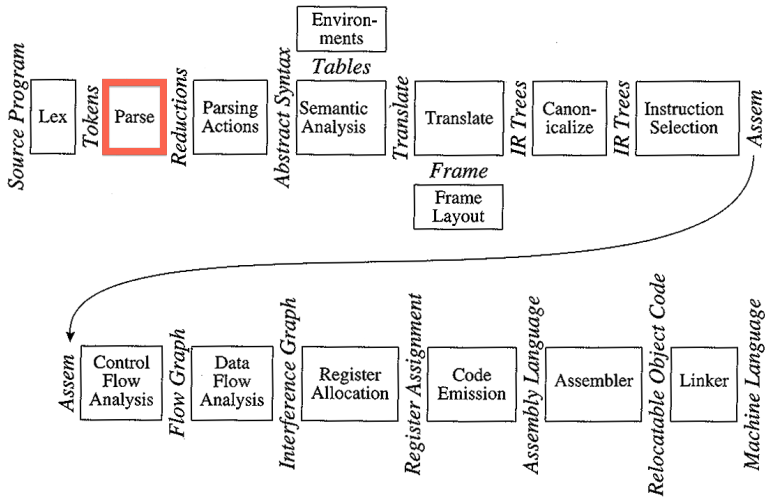
Zustände

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 `yybegin()` der Zustand gewechselt werden.
- Man verwendet das, um geschachtelte Kommentare und Stringlitterale 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: $G = (\Sigma, V, P, S)$
 - $a, b, c, \dots \in \Sigma$: Terminalsymbole (Eingabesymbole, Tokens)
 - $X, Y, Z, \dots \in V$: Nichtterminalsymbole (Variablen)
 - P : Produktionen (Regeln) der Form $X \rightarrow \gamma$
 - $S \in V$: 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 $X \rightarrow \gamma \in P$
 - Links-/Rechtsableitung: $w X \beta \Rightarrow_{\text{lm}} w \gamma \beta$ bzw. $\alpha X w \Rightarrow_{\text{rm}} \alpha \gamma w$
 - 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:

$$\begin{array}{llll} S \rightarrow E \$ & E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \text{id} \\ & E \rightarrow E - T & T \rightarrow T / F & F \rightarrow \text{num} \\ & E \rightarrow T & T \rightarrow F & F \rightarrow (E) \end{array}$$

```
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) $S \rightarrow F$

(2) $S \rightarrow (S+F)$

(3) $F \rightarrow a$

	()	a	+	\$
S	2		1		
F			3		

Eingabe: (a+a)

Stack	Input	Aktion
S \$	(a + a) \$	apply (2) $S \rightarrow (S+F)$
(S + F) \$	(a + a) \$	match '('
S + F) \$	a + a) \$	apply (1) $S \rightarrow F$
F + F) \$	a + a) \$	apply (3) $F \rightarrow a$
a + F) \$	a + a) \$	match 'a'
+ F) \$	+ a) \$	match '+'
F) \$	a) \$	apply (3) $F \rightarrow a$
a) \$	a) \$	apply (3) $F \rightarrow a$
) \$) \$	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): left-to-right parsing, left-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

- $\text{FIRST}(\gamma)$ ist die Menge aller Terminalsymbole, die als Anfänge von aus γ abgeleiteten Wörtern auftreten:

$$\text{FIRST}(\gamma) = \{a \mid \exists w. \gamma \Rightarrow^* aw\}$$

- $\text{FOLLOW}(X)$ 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:

$$\text{nullable}(\gamma) \iff \gamma \Rightarrow^* \epsilon.$$

Berechnung der First- und Follow-Mengen

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

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

Berechnung der First- und Follow-Mengen

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

Setze $\text{nullable}(a) := \text{false}$ und $\text{FIRST}(a) = \{a\}$ für alle Terminalsymbole a .

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

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

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

Konstruktion des Parsers

Soll die Eingabe gegen X geparkt werden (d.h. wird X erwartet) und ist das nächste Token a , so kommt die Produktion $X \rightarrow \gamma$ 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 dadurch 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 → 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

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

LL- und LR-Parser

Letztes Mal: **LL(k)**-Parsing

- **L**eft-to-right parsing
- **L**eftmost derivation
- **k** token lookahead

Heute: **LR(k)**-Parsing

- **L**eft-to-right parsing
 - **R**ightmost derivation
 - **k** token lookahead
-
- **LL**-Parser sind leicht zu implementieren (rekursiver Abstieg)
 - **LR**-Parser erfassen mehr Grammatiken, insbesondere mit Linksrekursion, sind aber schwerer zu implementieren (braucht Parsergeneratoren)

Wiederholung: LL-Parser

Grammatik:

(1) $S \rightarrow F$

(2) $S \rightarrow (S+F)$

(3) $F \rightarrow a$

	()	a	+	\$
S	2		1		
F			3		

Eingabe: (a+a)

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

Der Lauf entspricht der Linksableitung

$S \rightarrow (S + F) \rightarrow (S + a) \rightarrow (F + a) \rightarrow (a + a).$

Der Parser muss die Produktion anhand der ersten k Zeichen des von ihr generierten Wortes auswählen.

LR-Parser

Grammatik:	Stack	Input	Aktion
(1) $S \rightarrow F$		(a + a) \$	shift '('
(2) $S \rightarrow (S+F)$	(a + a) \$	shift 'a'
(3) $F \rightarrow a$	(a	+ a) \$	reduce (3) $F \rightarrow a$
	(F	+ a) \$	apply (1) $S \rightarrow F$
Eingabe: (a+a)	(S	+ a) \$	shift '+'
	(S +	a) \$	shift 'a'
	(S + a) \$	reduce (3) $F \rightarrow a$
	(S + F) \$	shift ')'
	(S + F)	\$	reduce (2) $S \rightarrow (S+F)$
	S	\$	accept

Der Lauf entspricht der Rechtsableitung

$$S \rightarrow (S + F) \rightarrow (S + a) \rightarrow (S + a) \rightarrow (F + a) \rightarrow (a + a).$$

Regeln werden rückwärts angewendet (reduziert). Die Entscheidung wird erst getroffen, wenn die rechte Seite der Regel vollständig gelesen wurden.

LR-Syntaxanalyse

Der LR-Parser kann zu jedem Zeitpunkt eine der folgenden beiden Aktionen durchführen:

- *Shift*-Aktion: Ein weiteres Eingabesymbol lesen und auf den Stack legen.
- *Reduce*-Aktion: Eine Produktion auf die oberen Stacksymbole rückwärts anwenden, also den Stackinhalt $\sigma\gamma$ durch σX ersetzen, falls eine Produktion $X \rightarrow \gamma$ vorhanden ist.

Enthält der Stack nur das Startsymbol und wurde die gesamte Eingabe eingelesen, so wird akzeptiert.

Eine Grammatik ist per Definition LR(k), wenn die Entscheidung, welche der beiden Aktionen durchzuführen ist, allein aufgrund der bisher gelesenen Eingabe, sowie der nächsten k Eingabesymbole, getroffen werden kann.

LR-Syntaxanalyse

Aus technischen Gründen erweitern wir im Folgenden Grammatik um eine Produktion $S' \rightarrow S\$$, wobei $\$$ eine Markierung für das Ende der Eingabe ist.

Akzeptanz der Eingabe entspricht Reduktion mit dieser Regel.

LR-Parser – Beispiel Linksrekursion

Grammatik:	Stack	Input	Aktion
(0) $S' \rightarrow S\$$		tt \wedge tt \$	shift 'tt'
(1) $S \rightarrow S \wedge B$	tt	\wedge tt \$	reduce (4) $B \rightarrow tt$
(2) $S \rightarrow S \vee B$	B	\wedge tt \$	reduce (3) $S \rightarrow B$
(3) $S \rightarrow B$	S	\wedge tt \$	shift ' \wedge '
(4) $B \rightarrow tt$	S \wedge	tt \$	shift 'tt'
(5) $B \rightarrow ff$	S \wedge tt	\$	reduce (4) $B \rightarrow tt$
Eingabe: tt \wedge tt	S \wedge B	\$	reduce (1) $S \rightarrow S \wedge B$
	S	\$	accept

Die Grammatik ist LR(0): *Reduce* wann immer möglich, sonst *shift*.
 Wenn Stack=S und Eingabe=\$, dann *accept*.

LR-Parser – Beispiel Rechtsrekursion

Grammatik:	Stack	Eingabe	Aktion
(0) $S' \rightarrow S\$$		tt \wedge tt \$	shift 'tt'
(1) $S \rightarrow \mathbf{B} \wedge \mathbf{S}$	tt	\wedge tt \$	reduce (4) $B \rightarrow tt$
(2) $S \rightarrow \mathbf{B} \vee \mathbf{S}$	B	\wedge tt \$	shift ' \wedge '
(3) $S \rightarrow B$	B \wedge	tt \$	shift 'tt'
(4) $B \rightarrow tt$	B \wedge tt	\$	reduce (4) $B \rightarrow tt$
(5) $B \rightarrow ff$	B \wedge B	\$	reduce (3) $S \rightarrow B$
	B \wedge S	\$	reduce (1) $S \rightarrow B \wedge S$
Eingabe: tt \wedge tt	S	\$	accept

Die Grammatik ist nicht LR(0): Reduktion mit $S \rightarrow B$ statt Schiebens von ' \wedge ' würde in Sackgasse führen.

Die Grammatik ist aber immer noch LR(1).

Faustregel: **LR**-Parser mögen **L**inks-**R**ekursion.

Wann soll der Parser mit $X \rightarrow \gamma$ reduzieren?

- Wenn Reduktion möglich ist (d.h. γ oben auf dem Stack).
- Und wenn der Parsevorgang nach der Reduktion (bei geeignetem Rest der Eingabe) zur Akzeptanz führen kann. Bei LR(k) berücksichtigt man dabei die k folgenden Eingabesymbole $a_1 a_2 \dots a_k$.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma\gamma$	$a_1 a_2 \dots a_k w \$$	

Wann soll der Parser mit $X \rightarrow \gamma$ reduzieren?

- Wenn Reduktion möglich ist (d.h. γ oben auf dem Stack).
- Und wenn der Parsevorgang nach der Reduktion (bei geeignetem Rest der Eingabe) zur Akzeptanz führen kann. Bei LR(k) berücksichtigt man dabei die k folgenden Eingabesymbole $a_1 a_2 \dots a_k$.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma\gamma$	$a_1 a_2 \dots a_k w \$$	reduce $X \rightarrow \gamma$
σX	$a_1 a_2 \dots a_k w \$$	

Wann soll der Parser mit $X \rightarrow \gamma$ reduzieren?

- Wenn Reduktion möglich ist (d.h. γ oben auf dem Stack).
- Und wenn der Parsevorgang nach der Reduktion (bei geeignetem Rest der Eingabe) zur Akzeptanz führen kann. Bei LR(k) berücksichtigt man dabei die k folgenden Eingabesymbole $a_1 a_2 \dots a_k$.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma\gamma$	$a_1 a_2 \dots a_k w \$$	reduce $X \rightarrow \gamma$
σX	$a_1 a_2 \dots a_k w \$$...
\vdots	\vdots	\vdots
S	$\$$	accept

Wann soll der Parser das nächste Symbol a shift-en?

Wenn der Parsevorgang dann noch erfolgreich abgeschlossen werden könnte, d.h. es wenn es bei geeigneter Eingabe später die Möglichkeit gibt, eine zielführende Produktion anzuwenden.

Hierbei werden keine weiteren Zeichen der Eingabe berücksichtigt.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma \gamma$	$a \ w \ \$$	

Wann soll der Parser das nächste Symbol a shift-en?

Wenn der Parsevorgang dann noch erfolgreich abgeschlossen werden könnte, d.h. es wenn es bei geeigneter Eingabe später die Möglichkeit gibt, eine zielführende Produktion anzuwenden.

Hierbei werden keine weiteren Zeichen der Eingabe berücksichtigt.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma \gamma$	a w \$	shift 'a'
$\sigma \gamma a$	w \$	

Wann soll der Parser das nächste Symbol a shift-en?

Wenn der Parsevorgang dann noch erfolgreich abgeschlossen werden könnte, d.h. es wenn es bei geeigneter Eingabe später die Möglichkeit gibt, eine zielführende Produktion anzuwenden.

Hierbei werden keine weiteren Zeichen der Eingabe berücksichtigt.

Das heißt, folgende Situation liegt für mindestens ein w vor.

Stack	Input	Aktion
$\sigma \gamma$	a w \$	shift 'a'
$\sigma \gamma a$	w \$...
\vdots	\vdots	\vdots
S	\$	accept

Das Wunder der LR(k) Syntaxanalyse

Die Entscheidung über die auszuführende Aktion kann durch einen endlichen Automaten übernommen werden.

- Der endliche Automat liest den *Stackinhalt*.
- Die Zustände des Automaten enthalten Informationen darüber, welche Produktionen im weiteren Verlauf noch erfolgreich ausgeführt werden könnten.
- Nach Lesen des Stackinhalts kann man am Zustand des Automaten erkennen, welche Aktion auszuführen ist.

Wir beschreiben den Aufbau des Automaten am Beispiel LR(0).

LR(0): Konstruktion des Automaten

Der Automat wird zunächst nichtdeterministisch konzipiert.

Die Zustände sind sogenannte **LR(0)-Items** und haben die Form

$$(X \rightarrow \alpha \cdot \beta)$$

wobei $X \rightarrow \alpha\beta$ eine Produktion ist.

Informell drückt $(X \rightarrow \alpha \cdot \beta)$ folgende Informationen über den Kellerautomaten aus:

- α liegt bereits oben auf dem Stack.
- Wenn durch weiteres Lesen der Eingabe noch β auf den Stack gelegt wird, dann kann die Reduktion mit $X \rightarrow \alpha\beta$ bei geeigneter Eingabe zur Akzeptanz führen.

LR(0): Transitionen des Automaten

Definiere die Zustandsübergänge des Automaten wie folgt:

- Startzustand des Automaten: $(S' \rightarrow .S\$)$
- $(X \rightarrow \alpha.s\beta) \xrightarrow{s} (X \rightarrow \alpha s.\beta), s \in \Sigma \cup V,$
- $(X \rightarrow \alpha.Y\beta) \xrightarrow{\epsilon} (Y \rightarrow .\delta), \text{ falls } Y \rightarrow \delta \text{ [Vervollständigung]}$

Man zeigt durch Induktion, dass der so definierte Automat tatsächlich die gewünschten Sprachen erkennt.

LR(0): Aktionen

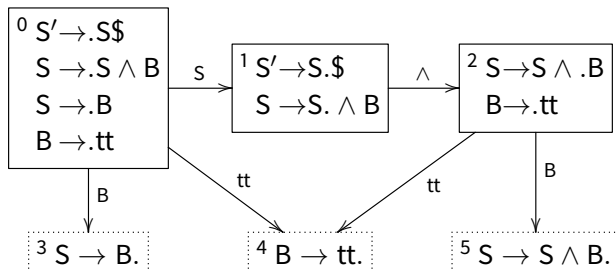
Nun determinisiert man den Automaten und erhält so *Mengen von LR(0)-Items* als Zustände des Automaten.

Der Automat kann nun auf dem Stackinhalt des Kellerautomaten ausgeführt werden, um zu entscheiden, welche Aktion auszuführen ist.

- Enthält der erreichte Zustand das Item $(X \rightarrow \gamma.)$, so reduziert man mit $X \rightarrow \gamma$.
- Enthält der erreichte Zustand das Item $(X \rightarrow \alpha.a\beta)$, so wird a von der Eingabe geshiftet.

Gibt es mehrere Möglichkeiten, so liegt ein *Shift/Reduce*, beziehungsweise ein *Reduce/Reduce-Konflikt* vor und die Grammatik ist nicht LR(0).

LR(0): Automat für Linksrekursion



Regeln:

- (0) $S' \rightarrow S\$$
- (1) $S \rightarrow S \wedge B$
- (3) $S \rightarrow B$
- (4) $B \rightarrow tt$

Zustand 0: shift (tt)

Zustand 1: shift (\wedge)

Zustand 2: shift (tt)

Zustand 3: reduce (mit Regel 3)

Zustand 4: reduce (mit Regel 4)

Zustand 5: reduce (mit Regel 1)

Beispiel

Regeln:	Stack	Input	Zustand
(0) $S' \rightarrow S\$$		tt \wedge tt \$	0
(1) $S \rightarrow S \wedge B$	tt	\wedge tt \$	4
(3) $S \rightarrow B$	B	\wedge tt \$	3
(4) $B \rightarrow tt$	S	\wedge tt \$	1
	S \wedge	tt \$	2
Input: tt \wedge tt	S \wedge tt	\$	4
	S \wedge B	\$	5
	S	\$	1

In der Spalte “Zustand” steht der Zustand des LR(0)-Automaten nach vollständigem Lesen des Stackinhalts.

Beispiel

Regeln:	Stack	Input	Zustand
(0) $S' \rightarrow S\$$	ε_0	tt \wedge tt \$	0
(1) $S \rightarrow S \wedge B$	ε_0 tt ₄	\wedge tt \$	4
(3) $S \rightarrow B$	ε_0 B ₃	\wedge tt \$	3
(4) $B \rightarrow tt$	ε_0 S ₁	\wedge tt \$	1
	ε_0 S ₁ \wedge_2	tt \$	2
Input: tt \wedge tt	ε_0 S ₁ \wedge_2 tt ₄	\$	4
	ε_0 S ₁ \wedge_2 B ₅	\$	5
	ε_0 S ₁	\$	1

Man muss nicht in jedem Schritt den gesamten Stackinhalt lesen. Man kann sich für jedes Symbol auf dem Stack den erreichten Zustand des LR(0)-Automaten merken. Dann muss man jeweils höchstens einen einzigen Schritt ausführen.

Implementierung: Parser-Tabelle

- Annotiere Stackeinträge mit erreichtem Automatenzustand (in der Praxis lässt man die ursprünglichen Stacksymbole ganz weg und arbeitet mit Automatenzuständen als Stackalphabet).
- Konstruiere Tabelle, deren Zeilen mit Zuständen und deren Spalten mit Grammatiksymbolen indiziert sind. Die Einträge enthalten eine der folgenden vier *Aktionen*:

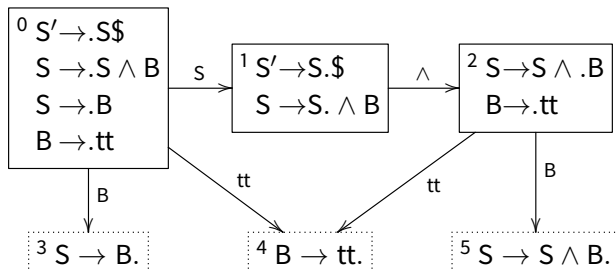
Shift (n)	„Shift“ und gehe in Zustand n ;
Reduce (k)	„Reduce“ mit Regel k ;
Goto (n)	Gehe in Zustand n ;
Accept	Akzeptiere.

Leere Einträge bedeuten Syntaxfehler.

Der LR-Algorithmus

- Ermittle Aktion aus der Tabelle anhand des obersten Stackzustands und des nächsten Symbols.
- Ist die Aktion...
 - Shift(n):** Lies ein Zeichen weiter; lege Zustand n auf den Stack.
 - Reduce(k):**
 - Entferne soviele Symbole vom Stack, wie die rechte Seite von Produktion k lang ist,
 - Sei X die linke Seite der Produktion k :
 - Finde in Tabelle unter dem nunmehr oben liegenden Zustand und X eine Aktion „**Goto(n)**“;
 - Lege n auf den Stack.
 - Accept:** Ende der Analyse, akzeptiere die Eingabe.

Beispiel: LR(0)-Automat für Linksrekursion



Regeln:

(0) $S' \rightarrow S\$$

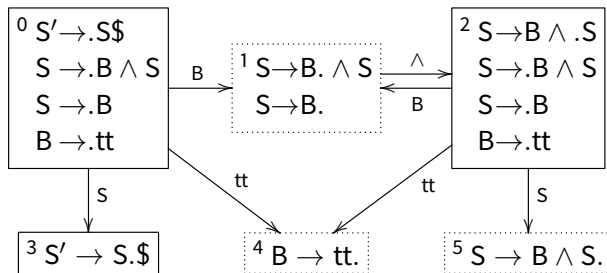
(1) $S \rightarrow S \wedge B$

(3) $S \rightarrow B$

(4) $B \rightarrow tt$

	tt	\wedge	\$	B	S
0	s4			g3	g1
1		s2	a		
2	s4			g5	
3	r3	r3	r3		
4	r4	r4	r4		
5	r1	r1	r1		

Rechtsrekursion



Regeln:

- (0) $S' \rightarrow S\$$
- (1) $S \rightarrow B \wedge S$
- (3) $S \rightarrow B$
- (4) $B \rightarrow tt$

Zustand 0: shift (tt)

Zustand 1: **Shift/Reduce-Konflikt**

Zustand 2: shift (tt)

Zustand 3:

Zustand 4: reduce (mit Regel 4)

Zustand 5: reduce (mit Regel 1)

In diesem Beispiel benötigt man einen LR(1)-Parser.

LR(1): Konstruktion des Automaten

Die Zustände haben die Form $(X \rightarrow \alpha.\beta, a)$ wobei $X \rightarrow \alpha\beta$ eine Produktion sein muss und a ein Terminalsymbol oder „?“ ist. Solch ein Zustand heißt **LR(1)-Item**.

Die Bedeutung diese Items entspricht der von $(X \rightarrow \alpha.\beta)$ mit der zusätzlichen Annahme, dass nach dem Lesen von β das nächste Eingabezeichen a ist.

Bei der Entscheidung, ob zu reduzieren ist, wird also das nächste Eingabezeichen berücksichtigt.

LR(1): Transitionen des Automaten

Die Zustandsübergänge des nichtdeterministischen Automaten sind wie folgt definiert:

- Startzustand des Automaten: $(S' \rightarrow .S\$, ?)$
- $(X \rightarrow \alpha.s\beta, a) \xrightarrow{s} (X \rightarrow \alpha s.\beta, a), s \in \Sigma \cup V,$
- $(X \rightarrow \alpha.Y\beta, a) \xrightarrow{\epsilon} (Y \rightarrow .\delta, b), \text{ falls } Y \rightarrow \delta \text{ und } b \in \text{FIRST}(\beta a)$
[Vervollständigung]

Man zeigt durch Induktion, dass der so definierte Automat tatsächlich die gewünschten Sprachen erkennt.

LR(1): Aktionen

Nun determinisiert man den Automaten und erhält so Mengen von LR(1)-Items als Zustände des Automaten. Der Automat wird auf den Stacksymbolen ausgeführt.

- Enthält der Endzustand das Item $(X \rightarrow \gamma., a)$ und ist das nächste Eingabesymbol a , so reduziert man mit $X \rightarrow \gamma$.
- Enthält der Endzustand das Item $(X \rightarrow \alpha.a\beta, c)$ und ist das nächste Eingabesymbol a , so wird geshiftet.

Gibt es mehrere Möglichkeiten, so liegt ein Shift/Reduce, beziehungsweise ein Reduce/Reduce-Konflikt vor und die Grammatik ist nicht LR(1).

LR(1): Beispiel

Grammatik:

(0) $S' \rightarrow S\$$

(1) $S \rightarrow E$

(2) $E \rightarrow E \wedge B$

(3) $E \rightarrow E \vee B$

(4) $E \rightarrow B$

(5) $B \rightarrow tt$

(6) $B \rightarrow ff$

Stack	Eingabe	Aktion
0	tt \wedge tt \$	shift 'tt'
0 2 _{tt}	\wedge tt \$	reduce (5) $B \rightarrow tt$
0 4 _B	\wedge tt \$	reduce (4) $E \rightarrow B$
0 3 _E	\wedge tt \$	shift ' \wedge '
0 3 _E 5 \wedge	tt \$	shift 'tt'
0 3 _E 5 \wedge 2 _{tt}	\$	reduce (5) $B \rightarrow tt$
0 3 _E 5 \wedge 7 _B	\$	reduce (2) $E \rightarrow E \wedge B$
0 3 _E	\$	reduce (1) $S \rightarrow E$
0 9 _S	\$	accept

Eingabe: tt \wedge tt

	\wedge	\vee	ff	tt	\$	E	B	S
0			s1	s2		g3	g4	g9
1	r6	r6			r6			
2	r5	r5			r5			
3	s5	s6			r1			
4	r4	r4			r4			
5			s1	s2			g7	
6			s1	s2			g8	
7	r2	r2			r2			
8	r3	r3			r3			
9					acc			

LALR(1) und SLR

LR(1)-Tabellen sind recht groß (mehrere tausend Zustände für typische Programmiersprache).

LALR(1) ist eine heuristische Optimierung, bei der Zustände, die sich nur durch die Vorausschau-Symbole unterscheiden, identifiziert/zusammengelegt werden. Eine Grammatik heißt LALR(1), wenn nach diesem Prozess keine Konflikte entstehen.

Bei SLR wird auf Vorausschau-Symbole in den Items verzichtet, stattdessen verwendet man FOLLOW-Mengen, um Konflikte aufzulösen.

JavaCUP

JavaCUP generiert LALR(1) Parser.

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

Von JavaCUP gemeldete Konflikte muss man ernst nehmen; in den meisten Fällen deuten sie auf Fehler in der Grammatik hin.

Mit der Option `-dump` wird der LR-Automat mit eventuellen Konflikten ausgegeben.

```
java -jar java-cup-11a.jar -dump input_file.cup
```

Man kann Konflikte durch Regelprioritäten lösen. Das ist aber nur sehr selten sinnvoll, z.B. bei „dangling else“:

$S \rightarrow \text{if } E \text{ then } S$

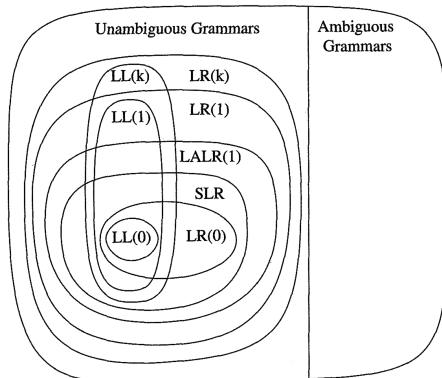
$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

Im

MiniJava-Parser sollten nach Beachtung der Operatorpräzedenzen keine Konflikte mehr enthalten sein.

Überblick

Grammatikklassen



Sprachklassen

$$LL(1) \subsetneq LL(2) \subsetneq \dots \subsetneq LR(1) = LR(2) = \dots = L(DPDA) \subsetneq L(CFG)$$

Ihre heutige Aufgabe

Stellen Sie die JavaCUP-Grammatik für MiniJava fertig.

- Lösen Sie *alle* Konflikte in Ihrer MiniJava-Grammatik.

Erweitern Sie Ihren Parser so, dass er einen abstrakten Syntaxbaum erzeugt.

- Im git-Repository finden Sie Klassen, mit denen die abstrakten Syntaxbäume von MiniJava repräsentiert werden können.
- Schreiben Sie semantische Aktionen für jede Regel, sodass entsprechende AST-Knoten erzeugt und mit den Unterknoten verbunden werden.
- Die Klasse Program hat eine Methode prettyPrint, mit der den eingelesenen Syntaxbaum wieder in einen String umwandeln kann. Testen Sie Ihren Parser damit.

Semantische Aktionen

```
/* Terminalsymbole */
terminal String IDENTIFIER;
terminal Integer INT;
terminal EQ;
...

/* Nichtterminalsymbole */
non terminal Stm Stm;
non terminal Exp Exp;
non terminal List<Exp> ExpList;

/* Grammatik */
Exp ::= Exp:e1 PLUS Exp:e2  {: RESULT = new Exp.OpExp(e1, Binop.PLUS, e2); :}
      | INT:num              {: RESULT = new Exp.NumExp(num); :}
      | LPAREN Exp:e RPAREN  {: RESULT = e; :}
      ...

ExpList ::= Exp:e             {: RESULT = Collections.singletonList(e); :}
          | ExpList:es COMMA Exp:e
            {: LinkedList<Exp> exps = new LinkedList<Exp>(es);
              exps.add(e);
              RESULT = exps;
            :}
          ;
```

Semantische Aktionen

```
/* Terminalsymbole */
terminal String IDENTIFIER;
terminal Integer INT;
terminal EQ;
...

/* Nichtterminalsymbole */
non terminal Stm Stm;
non terminal Exp Exp;
non terminal List<Exp> ExpList;

/* Grammatik */
Exp ::= Exp:e1 PLUS Exp:e2 { : RESULT = new Exp.OpExp(e1, Binop.PLUS, e2); :}
      | INT:num { : RESULT = new Exp.NumExp(num); :}
      | LPAREN Exp:e RPAREN { : RESULT = e; :}
...

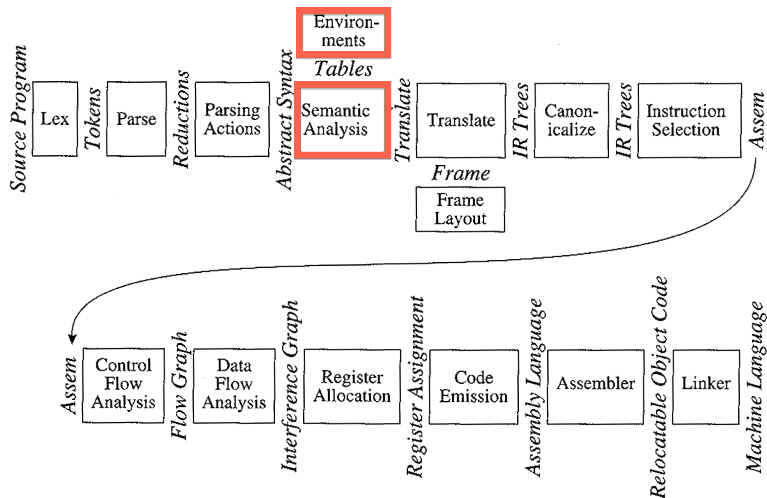
ExpList ::= Exp:e { : RESULT = Collections.singletonList(e); :}
          | ExpList:es COMMA Exp:e
            { : LinkedList<Exp> exps = new LinkedList<Exp>(es);
              exps.add(e);
              RESULT = exps;
            :}
;
;
```

Java-Typen (Typen der Ergebniswerte)

Java-Variablennamen für Teilergebnisse

Semantische Analyse

Semantische Analyse



Semantische Analyse

Fron-End muss prüfen, ob die Eingabe ein gültiges Programm der Eingabesprache ist.

Bisher:

- Lexikalische Analyse
(endlicher Automat)
- Syntaktische Analyse
(kontextfreie Grammatik)

Endliche Automaten und Kontextfreie Grammatiken sind nicht ausdrucksstark genug, um die Korrektheit von Programmen zu entscheiden. \Rightarrow Semantische Analyse

Semantische Analyse

In der semantischen Analyse im Front-End wird die abstrakte Syntax auf Gültigkeit überprüft:

- Typkorrektheit
- Alle Variablen werden deklariert bevor sie benutzt werden.
- Jede Methode hat return-Statements in jedem Lauf.
- Mögliche Exceptions sind mit throws-Klausel deklariert.
- ...

Bei möglicherweise ungewolltem Code werden Warnungen ausgegeben.

Semantische Informationen können auch später im Compiler nützlich sein, z.B. Typinformationen.

Semantische Analyse

In der ersten Phase wird das Programm üblicherweise nur auf Korrektheit geprüft. Weitere semantische Analysen können später stattfinden.

Beispiele für Analysen, die *erst später* durchgeführt werden.

- Wertebereiche genauer analysieren als durch die Typen vorgegeben, z.B. $0 < i < 100$.

```
for (int i = 1; i < 100; i++) { if (i < 1000) a[i]++; }
```

Test wird von aktuellen C-Compilern entfernt.

- Werden Zwischenergebnisse im Endergebnis überhaupt verwendet? Z.B. Entfernung von assert-Statements

```
if (debug) { ... }
```

- Wie lange kann auf Variablen zugegriffen werden? Speicherung in Register oder Hauptspeicher?
- Kontrollflussanalyse
- ...

Typüberprüfung

Ein Hauptteil der semantischen Analyse im Front-End ist die statische Typüberprüfung.

Statische Typsysteme

- Teil der Sprachdefinition
- Ausschluss von Laufzeitfehlern durch Typanalyse („Well-typed programs can't go wrong“ — Milner 1978)
- Typüberprüfung
- Typinferenz

Typinformationen im Compiler

- Generierung von besserem/effizienterem Code
- Ausschluss unsinniger Fälle
- Fehlervermeidung durch typisierte Zwischensprachen

Typüberprüfung

Für MiniJava kann die Typüberprüfung durch einen einmaligen Durchlauf des gesamten Programms implementiert werden, in dem alle Teile des Programms auf Typkorrektheit überprüft werden.

```
class C {                                ✓
    public int m () {                    ✓
        int i; D d;                     ✓
        d = new D();                    ✓
        i = d.f() + 2;                   ✓
        i = d.g(i);                      ✗
    }
}

class D {
    public int f() {...}
    public int g(boolean b) {...}
}
```

Symboltabelle

Es nützlich eine *Symboltabelle* anzulegen, in der z.B. Typinformationen für alle Identifier im Programm abgelegt werden.

```
class C {  
    public int m () {  
        int i; D d;  
        d = new D();  
        i = d.f() + 2; // Schlage hier den Typ von f in der  
                      // Symboltabelle nach (statt die Datei  
                      // nach der Definition zu durchsuchen).  
        ...  
    }  
}  
  
class D {  
    public int f() {...}  
    ...  
}
```

Symboltabellen können weitere Informationen enthalten und sind an vielen Stellen des Compilers nützlich.

Symboltabelle für MiniJava

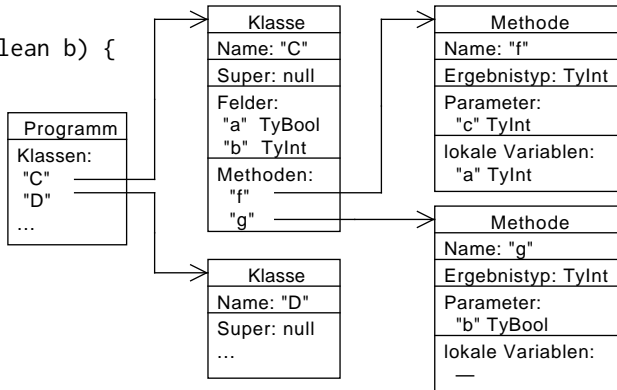
Eine Symboltabelle bildet Bezeichner ab auf „semantische Werte“.

- Semantischer Wert eines **Programms**:
 - Klassennamen mit ihren semantischen Werten
- Semantischer Wert einer **Klasse**:
 - Instanzvariablen mit ihren semantischen Werten (Ordnung wichtig)
 - Methodennamen mit ihren semantischen Werten
 - Verweis auf den Eintrag der Elternklasse
- Semantischer Wert einer **Instanzvariable**:
 - Typ
- Semantischer Wert einer **Methode**:
 - Ergebnistyp
 - Liste der Parameter mit ihren Typen (Ordnung wichtig)
 - lokale Variablen mit ihren Typen
 - Kann IOException ausgelöst werden?

Symboltabelle – Beispiel

```
class C {  
    boolean a;  
    int b;  
    public int f (int c) {  
        int a;  
        return a+b+c;  
    }  
    public int g (boolean b) {  
        return b+1;  
    }  
}
```

```
class D {  
    ...  
}
```



Typüberprüfung für MiniJava

Die Typüberprüfung selbst besteht nun aus einem Satz von Methoden, welche in Gegenwart einer Symboltabelle die verschiedenen Programmteile auf Typkorrektheit prüfen:

- Programm
- Klasse
- Methode
- Statement
- Expression

Wir betrachten zunächst MiniJava ohne Vererbung.

Typüberprüfung — Expressions

Für die Typüberprüfung ist es normalerweise nötig, den Typ eines gegebenen Ausdrucks auszurechnen (Typinferenz).

$typeOf(x)$ = Typ der Variablen x nachschlagen:

1. lokale Variablen, 2. Methodenparameter,
3. Instanzvariable der Klasse, (4. Superklasse ...)

$$typeOf(e_1 + e_2) = \begin{cases} \text{int} & \text{wenn } typeOf(e_1) = typeOf(e_2) = \text{int} \\ \perp & \text{sonst} \end{cases}$$

$$typeOf(e_1.m(e_2)) = \begin{cases} t & \text{wenn } typeOf(e_1) = C \text{ und} \\ & \text{Klasse } C \text{ hat Methode } t \text{ } m(typeOf(e_2) \ x) \\ \perp & \text{sonst} \end{cases}$$

Kompositionalität: Der Typ eines Ausdrucks ist durch die Typen seiner direkten Teilausdrücke bestimmt.

Implementierung mittels Visitor für Exp, analog zu PrettyPrint.

Typüberprüfung — Statements

Programmanweisungen (Statements) haben selbst keinen Typ, sie können nur wohlgetypt sein oder nicht.

Implementierung durch eine Funktion, die entscheidet, ob ein gegebenes Statement wohlgetypt ist:

$$typeOk(x = e) = \begin{cases} true & \text{wenn } typeOf(e) = typeOf(x) \\ false & \text{sonst} \end{cases}$$

$$typeOk(System.out.println(e)) = \begin{cases} true & \text{wenn } typeOf(e) = int \\ false & \text{sonst} \end{cases}$$

...

Typüberprüfung — Methode

```
public int f (int c, ...) {  
    int a;  
    ...  
    s  
    return e;  
}
```

Es werden folgende Bedingungen geprüft:

- Die Parameter und die lokalen Variablen haben paarweise verschiedene Namen.
Instanzvariablen dürfen aber überschattet werden.
- Die Anweisung `s` ist wohlgetypt.
- Der Typ von `e` entspricht dem Rückgabetypt der Funktion.
- Wenn in der Methode eine `IOException` ausgelöst werden kann, dann ist das auch deklariert.

Typüberprüfung — Klasse, Programm

Klasse

Um eine Klasse zu überprüfen, werden alle Methoden überprüft.

Es wird geprüft, dass keine Methode doppelt definiert ist.

Beachte: Die Methode `main` ist statisch und darf nicht auf `this` zugreifen!

Programm

Um ein Programm zu überprüfen, werden alle Klassen überprüft.

Es wird geprüft, dass keine Klasse mehrfach definiert ist.

Ihre Aufgabe

Implementieren Sie die Typüberprüfung für MiniJava ohne Vererbung.

Es bietet sich an, die Implementierung in zwei Phasen zu gliedern:

1. Erzeugung der Symboltabelle.

Die Klassen und Methoden des Eingabeprogramms werden einmal durchgegangen und die Symboltabelle wird dabei schrittweise aufgebaut.

2. Typüberprüfung.

Im zweiten Schritt wird das Programm noch einmal komplett durchgegangen. Nun kann die Symboltabelle benutzt werden, um die Eingabe auf Typkorrektheit zu überprüfen.

Ihre Aufgabe

Milestone 1: Parser und Typchecker

Schreiben Sie ein Hauptprogramm, das folgenden Anforderungen genügt:

1. Einlesen einer MiniJava-Datei, deren Dateiname als erstes Kommandozeilenargument gegeben ist.
2. Eingabe wird auf syntaktische Korrektheit sowie Typkorrektheit überprüft.
3. Ist die Eingabe ein korrektes MiniJava-Programm, soll das Programm mit Exit-Code 0 beendet werden. Andernfalls soll der Exit-Code $\neq 0$ sein. (`System.exit(exitcode);`).

Empfehlung: Automatisieren Sie Tests Ihres Compilers, mindestens mit den gegebenen Beispielprogrammen, gerne auch mit eigenen Testfällen.

Umgang mit Gültigkeitsbereichen

MiniJava hat ein sehr einfaches Typsystem, das die Behandlung von Variablen besonders einfach macht.

Nahezu alle Sprachen erlauben kompliziertere Gültigkeitsbereiche für Variablen.

```
class C {  
    int a; int b; int c;  
    public void m () {  
        System.out.println(a+c);  
        int j = a+b;  
        String a = "hello";  
        System.out.println(a);  
        for (int k = 0; k < 23; k++) {  
            System.out.println(j+k);  
        }  
        . . .  
    }  
}
```

Umgang mit Gültigkeitsbereichen

Man verwendet *Umgebungen*, um die Gültigkeit von Variablen zu verfolgen.

Eine Umgebung Γ ist eine Abbildung von Variablennamen auf semantische Werte, z.B. Typen.

class C {	$\Gamma_0 = \{g \mapsto \text{String}, a \mapsto \text{int}[]\}$
int a; int b; int c;	$\Gamma_1 = \Gamma_0 + \{a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}\}$
public void m () {	
System.out.println(a+c);	
int j = a+b;	$\Gamma_2 = \Gamma_1 + \{j \mapsto \text{int}\}$
String a = "hello";	$\Gamma_3 = \Gamma_2 + \{a \mapsto \text{String}\}$
System.out.println(a);	
for (int k = 0; k < 23; k++) {	
System.out.println(j+k);	$\Gamma_4 = \Gamma_3 + \{k \mapsto \text{int}\}$
}	Γ_3
...	

Hierbei steht $\Gamma + \Delta$ für die Umgebung, die man erhält, wenn man in Γ alle von Δ definierten Variablen durch ihre Werte in Δ überschreibt.

Typüberprüfung mit Gültigkeitsbereichen

Die Funktionen *typeOf* und *typeOk* hängen nun auch von der Umgebung ab.

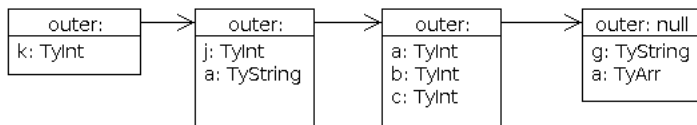
$$\text{typeOf}(\Gamma, e) = \dots$$

$$\text{typeOk}(\Gamma, s) = \dots$$

Im Fall für Variablen wird der Typ nun zunächst in der Umgebung nachgeschlagen.

Implementierung von Umgebungen

Umgebungen können als hierarchische Hash-/TreeMaps implementiert werden:



- Betreten eines Blocks durch Hinzufügen einer neuen Tabelle, deren outer-Feld zur Tabelle des aktuellen Gültigkeitsbereichs zeigt.
- Verlassen eines Gültigkeitsbereichs durch Löschen der aktuellen Tabelle. Die durch outer referenzierte Tabelle wird die neue aktuelle Tabelle.
- Um den Typ einer Variablen zu ermitteln, wird zuerst in der aktuellen Tabelle nachgeschaut, dann in der outer-Tabelle

Umgang mit Vererbung

Bei Einfachvererbung kann die Sichtbarkeit von Methoden und Feldern wie bei den Umgebungen implementiert werden (super spielt die Rolle von outer).

Die Typisierungsregeln müssen modifiziert werden, um der Unterklassenrelation \leq Rechnung zu tragen.

Beispiele

$$\frac{e_1 : C \leq D \quad D \text{ hat Methode } t'_1 m(t'_2) \text{ mit } t'_1 \leq t_1 \quad e_2 : t_2 \leq t'_2}{e_1.m(e_2) : t_1}$$

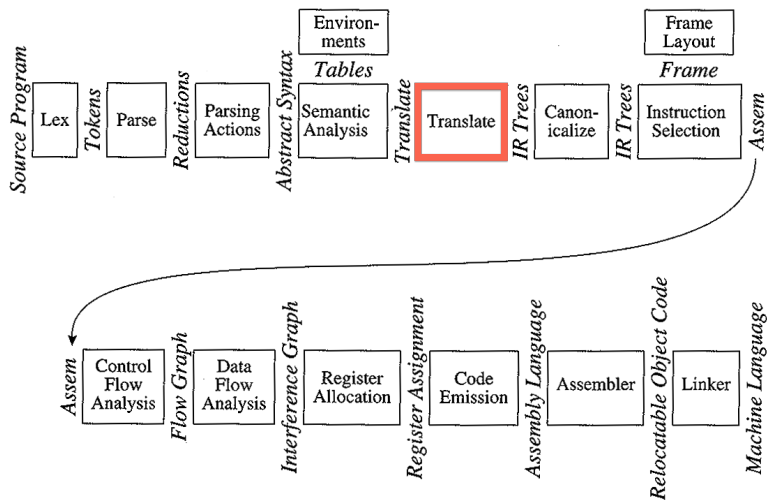
$$\frac{x : t \quad e : t' \quad t' \leq t}{x := e \text{ ok}}$$

Optionale Aufgaben

- benutzerfreundliche Fehlermeldungen
- Vererbung
- lokale Variablendeklarationen

Übersetzung in Zwischencode

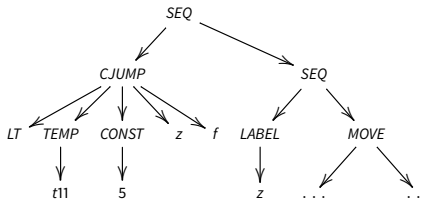
Übersetzung in Zwischencode



Überblick Zwischensprache

Im ersten Schritt der Compilierung wird in eine einfache Zwischensprache übersetzt.

- Abstraktionsebene vergleichbar mit C (portabel, aber maschinenabhängig)
- minimale Anzahl an Sprachkonstrukten (manche Compiler nennen die Zwischensprache C--)
- keine konkrete Syntax (stattdessen direkte Übersetzung in abstrakte Syntaxbäume)



Die Übersetzung in die Zwischensprache muss höhere Programmiersprachenkonstrukte (z.B. Klassen) entfernen.

Zwischensprache — Programme

Ein Programm besteht aus einer Liste von Funktionen.

```
Ladd(2) {  
    MOVE(TEMP(x), BINOP(PLUS, PARAM(0), PARAM(1)))  
    return x  
}
```

```
Ldouble_num(1) {  
    MOVE(TEMP(x), CALL(NAME(Ladd), PARAM(0), PARAM(0)))  
    return x  
}
```

Alle Werte der Zwischensprache sind 32-Bit Integer-Werte.
(Das ist eine Vereinfachung: normalerweise unterscheidet man verschiedene Typen, z.B. Integer verschiedener Breite und Pointer)

Zwischensprache — Programme

Entsprechendes C-Programm:

```
int32_t Ladd(int32_t param0, int32_t param1) {  
    int32_t x = param0 + param1;  
    return x;  
}
```

```
int32_t Ldouble_num(int32_t param0) {  
    int32_t x = Ladd(param0, param0);  
    return x;  
}
```

Zwischensprache — Funktionen

Bei Funktionen ist nur die Anzahl der Parameter angegeben (alle vom Typ `int32_t`); der Rückgabetyt ist ebenfalls `int32_t`.

```
Ladd(2) {  
    MOVE(TEMP(x), BINOP(PLUS, PARAM(0), PARAM(1)))  
    return x  
}
```

Eine einzelne Funktion besteht aus folgenden Daten:

- Funktionsname (im Beispiel: Ladd)
- Anzahl der Parameter (im Beispiel: 2)
- Liste von Anweisungen (im Beispiel: `MOVE(...)`)
- Rückgabevariable (im Beispiel: `x`)

Zwischensprache — Ausdrücke und Anweisungen

Wie in MiniJava gibt es auch in der Zwischensprache Ausdrücke und Anweisungen.

Anweisungen (Statements) sind Instruktionen, die keinen Rückgabewert haben und nur wegen ihrer Seiteneffekte ausgeführt werden.

Beispiel: `MOVE(TEMP(x), CONST(3))`

Ausdrücke (Expressions) liefern nach der Auswertung ein Ergebnis, einen 32-Bit Integer-Wert.

Beispiel: `BINOP(PLUS, TEMP(x), CONST(3))`

Zwischensprache — Ausdrücke (Expressions)

- $\text{CONST}(i)$: Die Integer-Konstante i .
- $\text{NAME}(l)$: Die symbolische Konstante l . (z.B. Funktionsname oder ein Label, der ein Sprungziel definiert).
- $\text{TEMP}(t)$: Die Variable t (**temporary**). Die Zwischensprache stellt beliebig viele Temporaries bereit. Temporaries können wie lokale Variablen in C verstanden werden (die ohne Deklaration verwendet werden können).
- $\text{PARAM}(i)$: Wert des i -ten Funktionsparameters.
- $\text{BINOP}(o, e_1, e_2)$: Die Anwendung des binären Operators o auf die Ausdrücke e_1 and e_2 , die zuvor in dieser Reihenfolge ausgewertet werden.

Binäre Operatoren:

- PLUS, MINUS, MUL, DIV: vorzeichenbehaftete Rechenoperationen
- AND, OR, XOR: *bitweise* logische Operationen
- LSHIFT, RSHIFT, ARSHIFT: Schiebeoperationen

Zwischensprache — Ausdrücke (Expressions)

- $\text{MEM}(e)$: Bezeichnet den Inhalt der Speicheradresse e .
- $\text{CALL}(f, e_1, \dots, e_n)$: Aufruf der Funktion f mit Argumenten e_1, \dots, e_n , die vorher in dieser Reihenfolge ausgewertet werden.
- $\text{ESEQ}(s, e)$: Ausführung von Statement s ; danach Auswertung von e .

Zwischensprache — Anweisungen (Statements)

- $\text{MOVE}(e_{dest}, e_{src})$: Zuweisung
Nicht alle Ausdrücke kommen für e_{dest} in Frage, nur sog. *Links-Ausdrücke* (*l-expressions*):
 - $\text{MOVE}(\text{TEMP } t, e)$: Auswerten von e und Abspeichern des Ergebnis in Temporary t .
 - $\text{MOVE}(\text{PARAM } i, e)$: Auswerten von e und Überschreiben des i -ten Parameters.
 - $\text{MOVE}(\text{MEM } e_1, e_2)$: Auswerten von e_1 zu Adresse a . Den Wert von e_2 in Speicheradresse a abspeichern.
 - $\text{MOVE}(\text{ESEQ}(s_1, e_1), e_2)$: Entspricht $\text{SEQ}(s_1, \text{MOVE}(e_1, e_2))$.
- $\text{SEQ}(s_1, s_2, \dots, s_n)$: Sequentielle Ausführung.
- $\text{LABEL}(l)$: Definition der symbolischen Sprungadresse l .
- $\text{JUMP}(e, labs)$: Ausführung bei Adresse e fortsetzen, wobei die Liste $labs$ alle möglichen Werte für diese Adresse angibt.

Zwischensprache — Anweisungen (Statements)

- $\text{CJUMP}(o, e_1, e_2, l_{\text{true}}, l_{\text{false}})$: Bedingter Sprung: Vergleiche die Werte von e_1 und e_2 mit Operator o . Ist der Vergleich wahr, so springe zum Label l_{true} , sonst zum Label l_{false} .

Vergleichsrelationen:

- EQ, NE, LT, GT, LE, GE: vorzeichenbehaftet
- ULT, UGT, ULE, UGE: vorzeichenfrei

Zwischensprache — Beispielfunktion

```
Lfactorial(1) {  
  CJUMP(LT, PARAM(0), CONST(1), L$0, L$1)  
  LABEL(L$1)  
  MOVE(TEMP(t5),  
    BINOP(MUL,  
      PARAM(0),  
      CALL(NAME(Lfactorial),  
        BINOP(MINUS, PARAM(0), 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  
}
```

Zwischensprache — Programme

Die Hauptfunktion heißt Lmain.

```
Lfactorial(1) {  
    ...  
}  
  
Lmain(1) {  
    MOVE(TEMP(t11), CALL(NAME(L_println_int),  
                          CALL(NAME(Lfactorial), CONST(10))))  
    MOVE(TEMP(t12), CONST(0))  
    return t12  
}
```

Ein- und Ausgabe sowie Speicherverwaltung wird durch externe Funktionen L_println_int, L_write, L_read, L_halloc und L_raise übernommen

Zwischensprache — Laufzeitbibliothek

Wir nehmen an, dass es die folgenden eingebauten Funktionen gibt:

- `L_println_int(i)`: gibt die Zahl i mit Zeilenumbruch auf dem Bildschirm aus.
- `L_write(b)`: schreibt das Byte b in `stdout`.
- `L_read()`: liest ein Byte von `stdin` und gibt es zurück.
- `L_halloc(n)`: reserviert n Bytes im Speicher, initialisiert diesen mit Nullen und liefert einen Zeiger darauf zurück.
- `L_raise(i)`: bricht das Programm mit Laufzeitfehler i ab.

Diese Funktionen werden dann die Laufzeitbibliothek unseres MiniJava-Compilers sein. Compilierte Programme werden mit einer Implementierung dieser Funktionen verlinkt.

Wir verwenden eine einfache Implementierung dieser Funktionen in C (Datei `runtime.c`).

Testen von Programmen der Zwischensprache

Auf der Praktikumshomepage finden Sie:

- Beispielprogramme in der Zwischensprache
- einen Compiler `tree2c`, der Programme in der Zwischensprache nach C übersetzt.
- Datentypen zur Repräsentierung der abstrakten Syntax der Zwischensprache.

Damit können Sie sich mit der Zwischensprache vertraut machen (und später Ihren Compiler testen).

Verwendung von `tree2c`:

```
$ tree2c programm.tree > programm.c  
$ gcc -m32 programm.c runtime.c -o programm  
$ ./programm
```

Von MiniJava in die Zwischensprache

Unser Ziel ist nun, MiniJava in die Zwischensprache zu übersetzen (abstrakte Syntax in abstrakte Syntax).

In der Zwischensprache gibt es keine Objekte oder komplizierte Datentypen, sondern nur Integer-Werte.

Alle Datentypen, insbesondere Arrays und Objekte, müssen nun in solche einfachen Typen abgebildet werden und durch die Übersetzung entsprechend umgewandelt werden.

Durch den Aufruf von externen Funktionen (z.B. `L_malloc`) können Programme der Zwischensprache Speicher allokieren. Zeiger auf Speicherbereiche werden ebenfalls als Integer-Werte gespeichert.

Übersetzung höherer Sprachkonstrukte: Objekte

Ohne Vererbung können Instanzmethoden als einfache notationelle Abkürzungen verstanden werden.

- Instanzmethoden werden auf einem Objekt (`this`) ausgeführt. Die Referenz `this` wird implizit übergeben und gesetzt.
- Macht man die Übergabe der `this`-Referenz explizit, so kann man alle Instanzmethoden durch globale statische Funktionen ersetzen.

Eine Instanzmethode einer Klasse `F`

```
public int m(int num) { ... }
```

wird zu einer statischen Funktion `F$m`, die als erstes Argument eine Referenz auf das “`this`”-Objekt erhält.

```
int F$m(F this_, int num) {  
    // ersetze this.m1(args) durch F$m1(this_, args)  
    // ersetze g.m2(args) durch G$m2(g, args)  
    ...  
}
```

Beispiel: Von MiniJava ...

```
class F {  
    int lastnum;  
    int lastresult;  
  
    public int compute(int num) {  
        int result;  
        if (num < 1) { result = 1; }  
        else { result = num * (this.compute(num - 1)); }  
        lastnum = num;  
        lastresult = result;  
        return result;  
    }  
    public int test() {  
        System.out.println(this.compute(10));  
        System.out.println(lastnum);  
        System.out.println(lastresult);  
        return 0;  
    }  
}
```


...nach MiniJava¹ ohne Instanzmethoden

```
class F {  
    int lastnum;  
    int lastresult;  
}  
  
static int F$compute(F this_, int num) {  
    int result;  
    if (num < 1) { result = 1; }  
    else { result = num * (F$compute(this_, num - 1)); }  
    this_.lastnum = num;  
    this_.lastresult = result;  
    return result;  
}  
  
static int F$test(F this_) {  
    System.out.println(F$compute(this_, 10));  
    System.out.println(this_.lastnum);  
    System.out.println(this_.lastresult);  
    return 0;  
}
```

¹ mit etwas erweiterter Syntax

Repräsentierung von Werten

Alle Werte in solchen reduzierten Programmen können durch Integer-Werte repräsentiert werden.

- `int`: Ist bereits Integer-Wert.
- `boolean`: Kodierung als Integer-Wert ($0 = \text{false}$, $1 = \text{true}$).
- `int[]`: Ein Array `int[] a` wird im Heap gespeichert und durch seine Adresse p repräsentiert. Der Zeiger kann als Integer-Wert betrachtet werden.

Adresse	Inhalt
p	<code>a.length</code>
$p + w$	<code>a[0]</code>
$p + 2w$	<code>a[1]</code>
\dots	
$p + (n + 1) \cdot w$	<code>a[n]</code>

wobei $n = \text{a.length} - 1$ und $w = 4$ oder 8 (*word size*).

Repräsentierung von Werten (ohne structs)

- Klassen: Ein Objekt obj der Klasse C wird durch einen Zeiger p in den Heap repräsentiert.

Adresse	Inhalt
p	Klassen-Id
$p + w$	feld1
$p + 2w$	feld2
...	
$p + n \cdot w$	feldn

```
class C {  
    int feld1;  
    boolean feld2;  
    ...  
    SomeClass feldn;  
  
    /* Methoden */  
}
```

Mit dieser Repräsentierung können MiniJava-Programme ohne Instanzmethoden praktisch direkt nach C übersetzt werden.

...nach C

```
#include <stdint.h>
#define MEM(x) *((int32_t*)(x))

F$compute(int32_t this, int32_t num) {
    int32_t result;
    if (num < 1) { result = 1; }
    else { result = num * (F$compute(this, num - 1)); }
    MEM(this + 4) = num;
    MEM(this + 8) = result;
    return result;
}

int32_t F$test(int32_t this) {
    printf("%i ", F$compute(this, 10));
    printf("%i ", MEM(this + 4));
    printf("%i ", MEM(this + 8));
    return 0;
}
```

Beispiel: Von MiniJava nach C

MiniJava:

```
public int compute(int num) {  
    int result;  
    if (num < 1) { result = 1; }  
    else { result = num * (this.compute(num - 1)); }  
    lastnum = num;  
    lastresult = result;  
    return result;  
}
```

C:

```
int32_t F$compute(int32_t this, int32_t num) {  
    int32_t result;  
    if (num < 1) { result = 1; }  
    else { result = num * (F$compute(this, num - 1)); }  
    MEM(this + 4) = num;  
    MEM(this + 8) = result;  
    return result;  
}
```

Ihre Aufgabe

Erweitern Sie Ihren MiniJava-Compiler, so dass er das Eingabeprogramm nach dem Parsen und Typchecken in die Zwischensprache übersetzt und das Ergebnis ausgibt.

- Überlegen Sie sich, wie MiniJava-Programme übersetzt werden können.
(evtl. zunächst auf Papier und/oder durch manuelle Übersetzung von Beispielpogrammen)
- Sie können mit der Implementierung heute beginnen.
- Für die Aufgabe ist auch noch in der nächsten Woche Zeit.

Übersetzung – Details

Sie brauchen Datentypen für die abstrakte Syntax der Zwischensprache nicht selbst zu entwickeln (siehe Praktikumshomepage).

Java-Klassen:

- TreePrg: ein gesamtes Programm der Zwischensprache
- TreeFunction: eine Funktion der Zwischensprache
- TreeStm: eine Anweisung der Zwischensprache
- TreeExp: ein Ausdruck der Zwischensprache
- Temp: ein Temporary
- Label: ein symbolisches Label

In TreePrg ist toString() so überschrieben, dass die Ausgabe mit tree2c übersetzt werden kann.

Alternative: Direkte Ausgabe von C-code mit CmmPrinter.

Übersetzung – Details

Die Übersetzung von MiniJava sollte kompositional erfolgen.

Programme

Ein MiniJava-Programm (Prg) wird zu einem TreePrg-Objekt.
Es besteht aus der Übersetzung der Methoden aller Klassen.

Methoden

Eine MiniJava-Methode wird zu einem TreeFunction-Objekt.
Dazu übersetzt man die Anweisung (Stm) im Körper der Methode
sowie den Rückgabedruck (Exp).

Anweisungen

Eine MiniJava-Anweisung (Stm) wird zu einem TreeStm-Objekt.
Verwende dazu einen StmVisitor.

Ausdrücke

Eine MiniJava-Anweisung (Exp) wird zu einem TreeExp-Objekt.
Verwende dazu einen ExpVisitor.

Methoden

Aus einer MiniJava-Methode in Klasse F

```
ty method(ty_1 x_1, ..., ty_n x_n) {  
    vardecls;  
    body;  
    return e;  
}
```

wird eine Zwischensprachen-Funktion

```
F$method( $n + 1$ ){  
    translate(body)  
    MOVE(TEMP(t), translate(e))  
    return t  
}
```

Die neue Funktion hat einen zusätzlichen Parameter. Das erste Argument enthält den this-Zeiger.

Anweisungen

MiniJava-Anweisungen können in einem Visitor-Durchlauf in TreeStm-Objekte übersetzt werden.

Im Folgenden sind einige Beispiele für mögliche Übersetzungen angegeben.

Zuweisungen

$$\text{translate}(x = e) = \text{MOVE}(\text{TEMP}(x), \text{translate}(e))$$

Anweisungen

Fallunterscheidung

Übersetzung einer **if**-Anweisung aus MiniJava:

$$\begin{aligned} \text{translate}(\mathbf{if\ } e \mathbf{\ then\ } S_1 \mathbf{\ else\ } S_2) = & \\ & \text{SEQ}(\text{CJUMP}(\text{EQ}, \text{translate}(e), \text{CONST}(1), l_{\text{true}}, l_{\text{false}}), \\ & \quad \text{LABEL}(l_{\text{true}}), \\ & \quad \text{translate}(S_1), \\ & \quad \text{JUMP}(l_{\text{exit}}), \\ & \quad \text{LABEL}(l_{\text{false}}), \\ & \quad \text{translate}(S_2), \\ & \quad \text{LABEL}(l_{\text{exit}})) \end{aligned}$$

Die Bedingung e wird mit bedingten Sprüngen ausgewertet und in den meisten Fällen in ein Temporary geschrieben. Erst dann wird das Ergebnis mit 1 verglichen und nach l_{true} oder l_{false} gesprungen.

Anweisungen

Schleifen

While-Schleifen werden ähnlich zur Fallunterscheidung mit der CJUMP-Anweisung übersetzt. Am Ende der Schleife springt man mit JUMP wieder zum Anfang, wo die Schleifenbedingung überprüft wird.

Allokation

Die Erzeugung von Objekten und Arrays wird durch den Aufruf der Bibliotheksfunktion `L_halloc(n)` realisiert, z.B.:

$$\text{translate}(\text{new } C()) = \text{CALL}(\text{NAME}(\text{L_halloc}), s)$$

Hierbei ist `s` die Größe in Bytes, die für ein Objekt der Klasse `C` im Speicher benötigt werden.

Der Rückgabewert von `L_halloc` ist ein Zeiger auf den neu allokierten Speicher.

Ausdrücke

Variablen

MiniJava-Variablen müssen entsprechend ihrer Art unterschiedlich behandelt werden.

- Lokale Variablen werden zu Temporaries $\text{TEMP}(x)$.
- Methodenparameter werden zu $\text{PARAM}(i)$.
- Instanzvariablen sind im Heap gespeichert und können durch Speicherzugriff relativ zum “this”-Parameter gelesen und geschrieben werden: $\text{MEM}(\text{BINOP}(\text{PLUS}, \text{PARAM}(0), \dots))$.

Binäroperationen

Viele MiniJava-Ausdrücke können direkt in TreeExp-Objekte übersetzt werden, z.B.:

$$\text{translate}(e_1 + e_2) = \text{BINOP}(\text{PLUS}, \text{translate}(e_1), \text{translate}(e_2))$$

Ausdrücke

Boolesche Vergleiche müssen durch Sprünge behandelt werden (analog zu bedingten Sprüngen in Assembler), z.B.:

```
translate( $e_1 < e_2$ ) =  
  ESEQ(SEQ(  
    MOVE( $t$ , CONST(0)),  
    CJUMP(LT, translate( $e_1$ ), translate( $e_2$ ),  $l_{\text{true}}$ ,  $l_{\text{false}}$ ),  
    LABEL( $l_{\text{true}}$ ), MOVE( $t$ , CONST(1)),  
    LABEL( $l_{\text{false}}$ )),  
  TEMP( $t$ ))
```

Hierbei ist t eine *neue* Variable und l_{true} , l_{false} sind *neue* Labels.

Ausdrücke

Methodenaufrufe

Methodenaufrufe werden ziemlich direkt in Funktionsaufrufe in der Zwischensprache übersetzt. Zu bemerken ist Folgendes:

- In MiniJava ohne Vererbung gibt es kein „overloading“ oder „dynamic dispatch“. Die Methodenauswahl richtet sich daher nach dem *statischen Typ* (durch semantische Analyse ermittelte Klasse) des aufgerufenen Objekts.
- Die Methode m in Klasse C wird durch eine Zwischensprachen-Funktion mit Namen $C\$m$ repräsentiert.
- Der Funktion $C\$m$ ist das aufgerufene Objekt als zusätzlicher Parameter zu übergeben.

Der Aufruf $e.m(e_1, \dots, e_n)$ wird folgendermaßen übersetzt:

$\text{CALL}(\text{NAME}(C\$m), \text{translate}(e), \text{translate}(e_1), \dots, \text{translate}(e_n))$

wobei $\text{translate}(e)$ den „this“-pointer der aufgerufenen Methode bestimmt.

Ausdrücke

Arrayzugriff

Beim Arrayzugriff $a[i]$ wird der Offset $a + w * (i + 1)$ berechnet:

$$\text{translate}(a[i]) = \text{MEM}(\text{BINOP}(\text{PLUS}, \text{translate}(a), \\ \text{BINOP}(\text{MUL}, \text{CONST}(w), \text{BINOP}(\text{PLUS}, \text{translate}(i), \text{CONST}(1))))))$$

MEM-Ausdrücke können auch als Ziel einer MOVE-Zuweisung vorkommen. Daher kann der schreibende Arrayzugriff $a[i] = e$ als

$$\text{MOVE}(\text{translate}(a[i]), \text{translate}(e))$$

übersetzt werden.

Im Allgemeinen unterscheidet man zwischen *l-exps*, Werte die links einer Wertzuweisung vorkommen (Adresse), und *r-exps*, Werte die rechts einer Wertzuweisung vorkommen (Wert).

In der Zwischensprache kann $\text{MEM}(s)$ sowohl als *l-exp* als auch als *r-exp* verwendet werden.

Ausdrücke — Hinweise

Strikte Konjunktion

Beachte die Semantik von `&&`:

Wenn der linke Teilausdruck schon falsch ist, wird der rechte nicht mehr ausgewertet.

Einhalten von Arraygrenzen

Die (Mini)Java-Semantik verlangt, dass vor einem Arrayzugriff die Einhaltung der Arraygrenzen überprüft wird.

Für einen Zugriff $a[i]$ sollte daher eine Laufzeitprüfung $0 \leq i < a.length$ im Zwischencode generiert werden, die im Fehlerfall mit der Bibliotheksfunktion `L_raise` eine entsprechende Fehlermeldung auslöst.

Allgemeine Hinweise

Temporaries und Labels möglichst immer neu erzeugen.

Für die kurzfristige Speicherung von Zwischenergebnissen immer eine *neue* Variable nehmen.

Temporäre Variablen nicht an anderer Stelle wiederverwenden.

Wiederverwendung von Variablen macht später das Leben des Registerallokators schwerer.

Weiterhin können subtile Fehler vermieden werden, in denen eine wiederverwendete Variable durch anderswo generierten Code verändert wird.

Generierung neuer Variablen und Labels durch Konstruktoren `new Temp()` und `new Label()` (diese Klassen haben einen statischen Zähler).

Optimierungsmöglichkeit: Fallunterscheidung

Übersetzung eines **if**-Statements aus MiniJava:

$$\begin{aligned} \text{translate}(\mathbf{if\ } e \mathbf{\ then\ } S_1 \mathbf{\ else\ } S_2) = \\ \text{SEQ}(\text{CJUMP}(\text{EQ}, \text{translate}(e), \text{CONST}(1), l_{\text{true}}, l_{\text{false}}), \\ \text{LABEL}(l_{\text{true}}), \text{translate}(S_1), \text{JUMP}(l_{\text{exit}}), \\ \text{LABEL}(l_{\text{false}}), \text{translate}(S_2), \text{LABEL}(l_{\text{exit}})) \end{aligned}$$

Die Bedingung e wird mit bedingten Sprüngen ausgewertet und in den meisten Fällen in ein Temporary geschrieben. Erst dann wird das Ergebnis mit 1 verglichen und nach l_{true} oder l_{false} gesprungen.

Eine mögliche Optimierung ist, die Bedingung so zu kompilieren, dass sofort nach l_{true} oder l_{false} gesprungen wird, wenn klar ist dass sie wahr bzw. falsch ist.

Optimierungsmöglichkeit: Fallunterscheidung

Verbesserte Übersetzung des **if**-Statements:

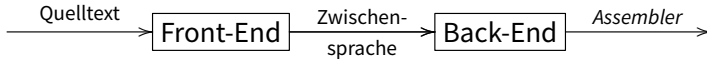
$$\begin{aligned} \text{translate}(\mathbf{if\ } e \mathbf{\ then\ } S_1 \mathbf{\ else\ } S_2) = \\ \text{SEQ}(\{\text{translateCond}(e, l_{\text{true}}, l_{\text{false}}), \\ \text{LABEL}(l_{\text{true}}), \text{translate}(S_1), \text{JUMP}(l_{\text{exit}}), \\ \text{LABEL}(l_{\text{false}}), \text{translate}(S_2), \text{LABEL}(l_{\text{exit}})\}) \end{aligned}$$

Dabei übersetzt `translateCond` Ausdrücke vom Typ `boolean` in Anweisungen der Zwischensprache:

$$\begin{aligned} \text{translateCond}(\text{true}, l_{\text{true}}, l_{\text{false}}) &= \text{JUMP}(l_{\text{true}}) \\ \text{translateCond}(!e, l_{\text{true}}, l_{\text{false}}) &= \text{translateCond}(e, l_{\text{false}}, l_{\text{true}}) \\ \text{translateCond}(e_1 \&\& e_2, l_{\text{true}}, l_{\text{false}}) &= \\ &\text{SEQ}(\text{translateCond}(e_1, l, l_{\text{false}}), \\ &\text{LABEL}(l), \\ &\text{translateCond}(e_2, l_{\text{true}}, l_{\text{false}}),) \end{aligned}$$

Überblick

Wie geht es weiter?



Front-End:

- Lexen, Parsen
- Typüberprüfung
- Übersetzung in Zwischensprache
- *Kanonisierung*

Back-End:

- *Instruktionsauswahl*
- *Programmanalyse*
- *Registerallokation*

Kanonisierung

Bringe den Zwischencode in eine Normalform, die sich leichter weiterverarbeiten lässt.

Beispiel:

```
MOVE(TEMP(t8), ESEQ(SEQ(CJUMP(LT, CONST(0), MEM(BINOP(PLUS,
  CONST(-4), TEMP(t2)))), L4, L5), SEQ(LABEL(L5),
  SEQ(MOVE(TEMP(t7), CALL(NAME(L_raise), CONST(0))),
  SEQ(LABEL(L4), SEQ()))))), MEM(TEMP(t2))))
```

wird zu

```
CJUMP(LT, CONST(0), MEM(BINOP(PLUS, CONST(-4), TEMP(t2))), L4, L5)
LABEL(L5)
MOVE(TEMP(t7), CALL(NAME(L_raise), CONST(0)))
LABEL(L4)
MOVE(TEMP(t8), MEM(TEMP(t2)))
```

Instruktionsauswahl

Zwischencode-Anweisungen werden in Maschineninstruktionen übersetzt. (Implementierung hängt vom Zielprozessor ab)

Beispiel:

CJUMP(LT, CONST(0), MEM(BINOP(PLUS, CONST(-4), TEMP(t2)))), L4, L5)

könnte zu folgenden x86-Instruktionen werden:

```
MOV t9, 0
MOV t10, t2
SUB t10, 4
CMP t9, t10
JL L4
JMP L5
```

Wir erlauben hier noch Temporaries als Platzhalter für Maschinenregister.

Instruktionsauswahl

Die Instruktionsauswahl generiert Assemblercode, in dem Funktionen manuell implementiert werden müssen.

Das Ergebnis der Instruktionsauswahl ist im Prinzip eine einzige Liste von Assemblerinstruktionen für das gesamte Programm.

- Man muss sich bei Funktionsaufrufen selbst um die Parameterübergabe, die Verwaltung von lokalen Variablen und die Rückgabe von Ergebnissen kümmern.
- Funktionsaufrufe werden meist mithilfe des Maschinen-Stack implementiert.
- Die Parameterübergabe ist oft standardisiert (calling convention). Das erlaubt den Aufruf von Code, der mit anderen Compilern übersetzt wurde (insbesondere `runtime.c`).

Registerallokation

Die Temporaries werden (möglichst optimal) auf wirkliche Maschinenregister abgebildet.

Registerallokation ist wieder unabhängig vom Zielprozessor.

Beispiel:

```
MOV t9, 0
MOV t10, t2
SUB t10, 4
CMP t9, t10
JL L4
```

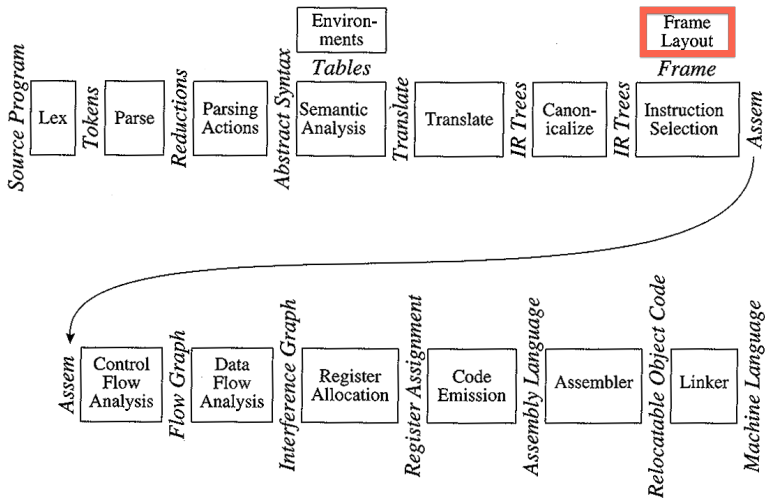
könnte zu folgendem fertigen x86-Code werden:

```
MOV eax, 0
SUB ebx, 4
CMP eax, ebx
JL L4
```

(ein MOV wurde entfernt, da t2 und t10 beide auf %ebx abgebildet wurden)

Aktivierungssätze (Frames)

Aktivierungssätze



Aktivierungssätze

Aktivierungssätze (auch **Activation Records** oder **Frames** genannt) dienen der Implementierung von Funktionsaufrufen.

- Ein Aktivierungssatz speichert Daten, die zu Funktionsaufrufen gehören:
 - konkrete Funktionsparameter
 - Werte der lokalen Variablen
 - Rücksprungadresse
 - ...
- Geschachtelte Funktionsaufrufe werden mittels eines *Stacks* von Frames realisiert.
- Das genaue Layout von Aktivierungssätzen ist hardware- und betriebssystemabhängig.

Beispiel

```
public int f(int x) {  
    int y, z;  
  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);  
        z = f(x - 1);  
        return y + z;  
    }  
}
```

- Beim Aufruf von `f` muss Speicherplatz für `x`, `y` und `z` angelegt werden.
- Der Parameter `x` muss von Aufrufer initialisiert werden.
- Aufgrund der Rekursion können mehrere Instanzen der Variablen simultan existieren.

Inhalt eines Aktivierungssatzes

Ein **Frame** speichert die lokale Umgebung der aktuellen Funktion, d.h.:

- Werte der Parameter
- Lokale Variablen
- Administrative Informationen, z.B. Rücksprungadresse
- Temporäre Variablen und zwischengespeicherte Register

Der Prozessorhersteller gibt oft eine bestimmte Aufrufkonvention vor (Parameterlayout etc.), damit Funktionen sprachübergreifend verwendet werden können.

Stack

Aktivierungssätze werden meist auf einem Stack verwaltet.

- Bei Funktionsaufruf wird ein Aktivierungssatz erzeugt und auf den Stack gelegt.
- Bei der Abarbeitung des Funktionskörpers kann auf Parameter und lokale Variablen über den Aktivierungssatz auf dem Stack zugegriffen werden.
- Beim Verlassen einer Funktion wird der Aktivierungssatz vom Stack genommen.

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 0, z = 0, \dots$

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 0, z = 0, \dots$
- Frame: $x = 1, y = 0, z = 0, \text{Rücksprungadresse} = 1$

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$
- Frame: $x = 2, y = 0, z = 0$, Rücksprungadresse = 2

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$
- Frame: $x = 2, y = 0, z = 0$, Rücksprungadresse = 2
- Frame: $x = 0, y = 0, z = 0$, Rücksprungadresse = 1

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$
- Frame: $x = 2, y = 1, z = 0$, Rücksprungadresse = 2

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$
- Frame: $x = 2, y = 1, z = 0$, Rücksprungadresse = 2
- Frame: $x = 1, y = 0, z = 0$, Rücksprungadresse = 2

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 0, \dots$
- Frame: $x = 2, y = 1, z = 1, \text{Rücksprungadresse} = 2$

Beispiel

```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

- Frame: $x = 3, y = 1, z = 2, \dots$

Beispiel

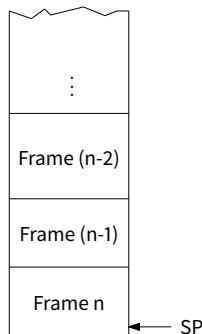
```
public int f (int x) {  
    int y, z;  
    if (x < 2) {  
        return 1;  
    } else {  
        y = f(x - 2);    // Adresse 1  
        z = f(x - 1);    // Adresse 2  
        return y + z; }  
}
```

Beispiel: Beim Aufruf von $f(3)$ entwickelt sich der Stack von Frames wie folgt:

Hardware Stack

Die meisten Architekturen unterstützen einen Hardware-Stack, auf dem die Aktivierungssätze abgelegt werden können.

- Der Stack ist ein Speicherbereich, dessen Ende durch den **Stack Pointer** (spezielles Register) markiert wird.
- Aus historischen Gründen wächst der Stack meist nach unten (hin zu niedrigeren Adressen).
- Der Stack Pointer zeigt auf das letzte Wort am Ende des Stacks.

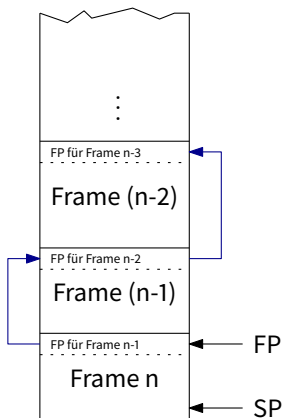


Der Frame-Pointer

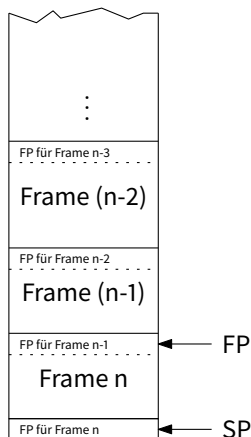
Zur Verwaltung des Stacks als eine Sequenz von Aktivierungssätzen werden zwei spezielle Register verwendet:

- **frame pointer** (FP): zeigt auf den Beginn des aktuellen Frames
- **stack pointer** (SP): zeigt auf das Ende des aktuellen Frames
- Bei Funktionsaufruf wird der Inhalt von FP auf dem Stack abgelegt (als sog. *dynamic link*) und SP nach FP übertragen.
- Ein neuer Frame wird durch Dekrementierung von SP angelegt.
- Beim Verlassen von f wird FP wieder nach SP geschrieben und FP aus dem Stack restauriert.
- Ist die Framegröße jeder Funktion zur Compilezeit bekannt, so kann FP jeweils ausgerechnet werden als $SP + \text{framesize}$.

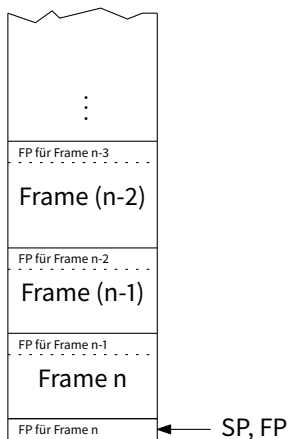
Anlegen eines neuen Frame bei Funktionsaufruf



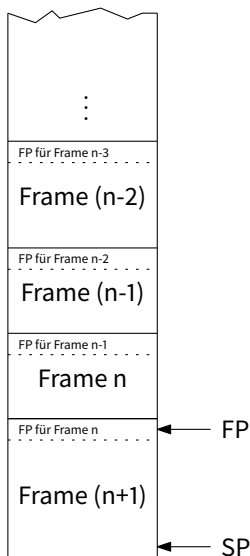
Anlegen eines neuen Frame bei Funktionsaufruf



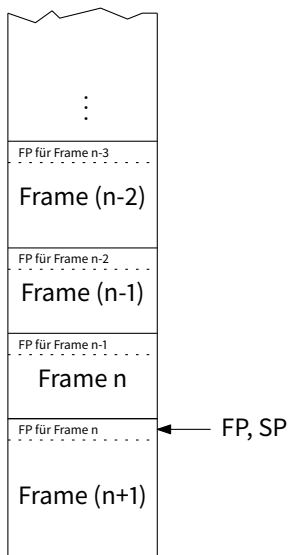
Anlegen eines neuen Frame bei Funktionsaufruf



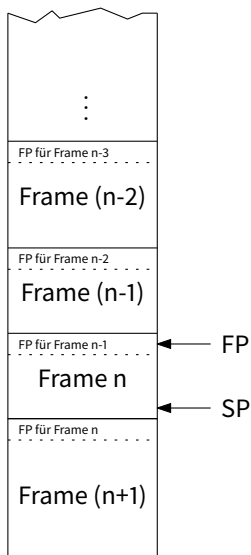
Anlegen eines neuen Frame bei Funktionsaufruf



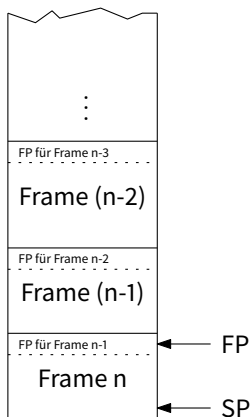
Wiederherstellen des letzten Frame bei Return



Wiederherstellen des letzten Frame bei Return



Wiederherstellen des letzten Frame bei Return



Aufrufkonventionen

Der genaue Aufbau von Frames unterscheidet sich je nach Architektur und Betriebssystem.

Die Details sind in einer **Aufrufkonvention (calling convention)** festgelegt.

- **Wo steht die Rücksprungadresse?**

Beispiel (x86-cdecl): ganz unten im Frame der aufrufenden Funktion.

- **Wo findet man die Parameter?**

Beispiel (x86-cdecl): direkt über der Rücksprungadresse im Frame der aufrufenden Funktion.

Beispiel (AMD64): die ersten sechs Parameter sind in Registern RDI, RSI, ... gespeichert; der Rest wie bei (x86-cdecl).

- ...

Wir werden x86-cdecl verwenden.

Typische Funktion in Pseudocode

```
function_name:
  // Entry Code (Frame anlegen)
  Push FP
  SP := FP - k * wordsize

  // Funktionsrumpf
  // Die Parameter sind gemäß Aufrufkonvention zugreifbar.
  // Die Speicherstellen mit Adressen
  // FP - wordsize, ..., FP - k*wordsize
  // können für lokale Variablen verwendet werden.

  // Exit Code (Frame entfernen)
  SP := FP
  POP FP
  Rückkehr zur Rücksprungadresse
```

Register

Register sind in den Prozessor eingebaute extrem schnelle Speicherelemente. RISC Architekturen (Sparc, PowerPC, MIPS) haben oft 32 Register à ein Wort; der Pentium hat acht Register (eax, ebx, ecx, edx, esi, edi, ebp, esp).

- Lokale Variablen, Parameter, etc. sollen soweit wie möglich in Registern gehalten werden.
- Oft können arithmetische Operationen nur in Registern durchgeführt werden.

Temporaries werden später soweit wie möglich in Registern gespeichert.

Wenn die Register nicht ausreichen, werden Temporaries in den Frame ausgelagert.

Aufrufkonventionen: Register

Register sind globale Variablen, deren Wert sich durch einen Funktionsaufruf ändern kann.

- Die Aufrufkonvention gibt vor, welche Register **caller-save**, d.h., von der aufgerufenen Funktion beschrieben werden dürfen; und welche **callee-save** sind, also von der aufgerufenen Funktion in den ursprünglichen Zustand wiederherzustellen sind.
- Befindet sich eine lokale Variable o.ä. in einem caller-save Register, so muss sie der Aufrufer vor einem Funktionsaufruf im Frame abspeichern, es sei denn, der Wert wird nach Rückkehr der Funktion nicht mehr gebraucht.
- Der Pentium hat drei caller-save Register (eax, ecx, edx) und vier callee-save (ebx, esi, edi, ebp) Register.

Aufrufkonventionen: Parameterübergabe

Je nach Architektur werden die ersten k Parameter in Registern übergeben; die verbleibenden über den Frame. Beim Pentium werden i.d.R. alle Parameter über den Frame übergeben, allerdings kann man bei GCC auch Werte $0 < k \leq 3$ einstellen.

Bei den Frameparametern hängt der Ort von der Aufrufkonvention ab:

- `cdecl`: in den untersten Frameadressen des Aufrufers, so dass sie in der aufgerufenen Funktion dann jenseits des Frame-Pointers sichtbar sind (aufrufende Funktion entfernt Parameter);
- `stdcall`: direkt im neuen Frame (aufgerufene Funktion entfernt Parameter).

Aufrufkonventionen: Rücksprungadressen

Wurde g von Adresse a aus aufgerufen, so muss nach Abarbeitung von g an Adresse $a + \text{wordSize}$ fortgefahren werden. Diese Adresse muss daher vor Aufruf irgendwo abgespeichert werden.

Im x86-Befehlssatz gibt es dafür spezielle Befehle:

- `call addr`: Lege die Adresse des folgenden Befehls auf den Stack und springe zur Adresse `addr`.
- `ret`: Nimm eine Adresse vom Stack und springe dorthin.

Die Rücksprungadresse liegt also im Frame.

Eine andere Möglichkeit besteht darin, die Rücksprungadresse in einem Register abzulegen, welches dann ggf. von der aufgerufenen Funktion im Frame zu sichern ist. Beispiel: ARM

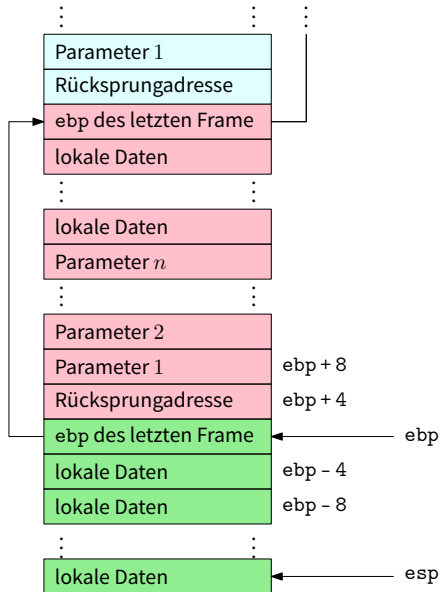
Aufrufkonventionen: Rückgabewerte

Die aufrufende Funktion muss den Rückgabewert eines beendeten Funktionsaufrufs auffinden können.

Rückgabewerte werden oft in einem speziellen Register zurückgegeben.

Bei x86 werden Integer- und Zeigerwerte im Register `eax` zurückgegeben.

x86-cdecl: Stacklayout



x86-cdecl: Funktionsdefinition

```
function_name:
```

```
    push ebp
```

```
    mov ebp, esp
```

```
    sub esp, k * wordsize
```

```
    ...
```

```
    // Parameter: [ebp + 8], [ebp + 12], ...
```

```
    // Lokale Variablen: [ebp - 4], [ebp - 8], ...
```

```
    mov eax, return_value
```

```
    mov esp, ebp
```

```
    pop ebp
```

```
    ret
```

x86-cdecl: Funktionsaufruf

```
push param_n  
push param_(n-1)  
...  
push param_1  
call function_name  
add esp, n*4
```

Mit der letzten Instruktion werden die Parameter wieder vom Stack entfernt. Bei cdecl ist das Aufgabe des Callers.

So kann man auch in C implementierte externe Funktionen aufrufen (z.B. `runtime.c`).

Escapes

Lokale Variablen und Parameter dürfen nicht in Registern abgelegt werden, wenn vom Programm auf ihre Adresse zugegriffen wird. Das ist der Fall bei Verwendung von Cs Adressoperator, Pascals Call-by-Reference, und bei der Verwendung von Zeigerarithmetik.

Solche Parameter („escapes“) können nur über den Frame übergeben werden.

In MiniJava gibt es keine Escapes.

Verschachtelte Funktionen

In manchen Sprachen (z.B. Pascal oder ML) können lokale Funktionen innerhalb eines Blocks deklariert werden. In lokalen Funktionen kann auf lokale Variablen des äußeren Blocks Bezug genommen werden.

Der inneren Funktion muss daher mitgeteilt werden, wo diese Variablen zu finden sind.

- *static link*: im aktuellen Frame wird ein Verweis auf den Frame der äußeren Funktion abgelegt.
- *lambda lifting*: man übergibt alle lokalen Variablen der äußeren Funktion als eigene Parameter.

Verschachtelte Funktionen gibt es in MiniJava nicht.

Höherstufige Funktionen

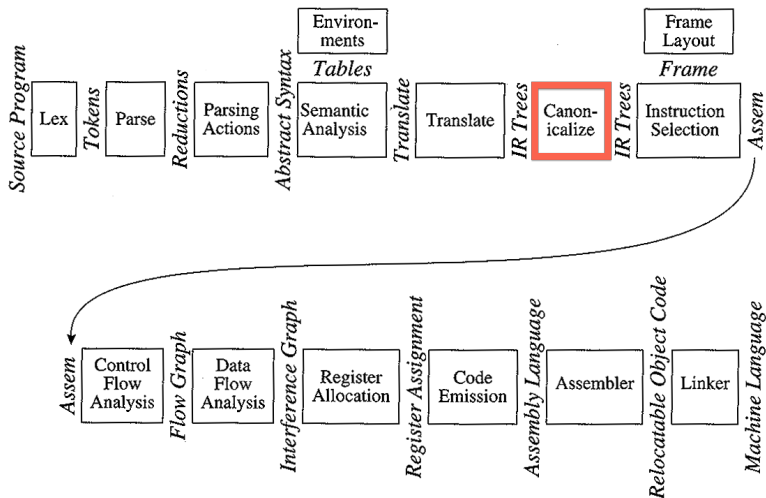
Können Funktionen nach außen gegeben werden, so bleiben unter Umständen lokale Variablen nach Verlassen einer Funktion am Leben:

```
fun incBy x =  
  let fun g y = x+y  
    in g  
  end  
let incBy3 = incBy 3  
let z = incBy3 1
```

Wird `incBy` verlassen, so darf `x` noch nicht zerstört werden!
In MiniJava gibt es allerdings keine höherstufigen Funktionen.

Kanonisierung

Kanonisierung



Kanonisierung

Jede Funktion des Zwischencodes wird kanonisiert, um die Weiterverarbeitung im Backend zu vereinfachen.

Definition: Eine Anweisung s heißt **kanonisch**, wenn sie folgende Eigenschaften hat.

- In s kommt weder SEQ noch ESEQ vor.
- In s gibt es höchstens ein Vorkommen von CALL.
- Wenn in s ein CALL vorkommt, dann hat s die Form $\text{MOVE}(\text{TEMP}(t), \text{CALL}(\vec{e}))$.

Definition: Eine Liste von Anweisungen \vec{s} ist in **Normalform**, falls gilt:

- In \vec{s} kommen nur kanonische Anweisungen vor.
- Wenn $\text{CJUMP}(r, e_1, e_2, l_1, l_2)$ in \vec{s} vorkommt, dann folgt direkt darauf die Anweisung $\text{LABEL}(l_2)$.

Kanonisierung

Zwischencode in Normalform ist leichter weiterzuverarbeiten:

- Bei der Übersetzung in Maschinencode im Backend sind weniger Fälle zu behandeln.
- Zwischencode in Normalform macht die Reihenfolge von Seiteneffekten explizit.
- Bei der Auswertung eines Ausdrucks können die Teilausdrücke in beliebiger Reihenfolge ausgewertet werden.
 - Ausdrücke ohne CALL haben keine Seiteneffekte und können in beliebiger Reihenfolge ausgewertet werden.¹
 - Ausdrücke mit CALL müssen die Form $\text{CALL}(\vec{e})$ haben und dürfen keinen weiteren CALL enthalten.
Die Argumente können dann ebenfalls in beliebiger Reihenfolge ausgewertet werden.

¹Genau genommen auch die Division einen Seiteneffekt haben und bei Division durch 0 zu einer Exception führen. Dafür ist die Reihenfolge aber ebenfalls nicht wichtig. Allgemein werden Operationen mit Seiteneffekten wie CALL behandelt.

Kanonisierung

Jede Funktion in der Zwischensprache wird einzeln kanonisiert.

Die Kanonisierung erfolgt in drei Schritten:

1. Kanonisierung aller Anweisungen.
2. Gruppierung der Instruktionen in *Basisblöcke*.
3. Anordnung der Basisblöcke durch „*Tracing*“.

Kanonisierung von Ausdrücken

Ausdrücke können Anweisungen enthalten, was in kanonisierten Programmen nicht mehr erlaubt ist.

```
MOVE(TEMP(t8),  
    ESEQ(SEQ(CJUMP(LT, CONST(0), MEM(BINOP(PLUS,  
        CONST(-4), TEMP(t2)))), L4, L5), SEQ(LABEL(L5),  
        SEQ(MOVE(TEMP(t7), CALL(NAME(L_raise), CONST(0)))),  
        SEQ(LABEL(L4), SEQ())))),  
    MEM(TEMP(t2))))
```

Die Anweisungen werden aus den Ausdrücken herausgezogen und vorher ausgeführt.

```
CJUMP(LT, CONST(0), MEM(BINOP(PLUS, CONST(-4), TEMP(t2)))), L4, L5)  
LABEL(L5)  
MOVE(TEMP(t7), CALL(NAME(L_raise), CONST(0)))  
LABEL(L4)  
MOVE(TEMP(t8), MEM(TEMP(t2)))
```

Kanonisierung von Ausdrücken

Kanonisierung eines Ausdrucks: $e \longrightarrow (\vec{s}, e')$:

- \vec{s} ist eine Liste von kanonischen Anweisungen
- e' enthält weder ESEQ noch CALL

Das Paar (\vec{s}, e') soll so aufgebaut sein, dass man e im Programm durch $\text{ESEQ}(\text{SEQ}(\vec{s}), e')$ ersetzen könnte.

Definition durch Induktion über den Ausdruck e .

Beispiele:

$$\begin{array}{c}
 \frac{}{\text{CONST}(i) \longrightarrow (\varepsilon, \text{CONST}(i))} \qquad \frac{e \longrightarrow (\vec{s}, e')}{\text{MEM}(e) \longrightarrow (\vec{s}, \text{MEM}(e'))} \\
 \\
 \frac{e \longrightarrow (\vec{s}, e') \quad e_1 \longrightarrow (\vec{s}_1, e'_1) \quad t, t_1, r \text{ frisch}}{\text{CALL}(e, e_1) \longrightarrow (\vec{s}, \text{MOVE}(\text{TEMP}(t), e'), \vec{s}_1, \text{MOVE}(\text{TEMP}(t_1), e'_1) \\
 \qquad \qquad \qquad \text{MOVE}(\text{TEMP}(r), \text{CALL}(t, t_1)), \\
 \qquad \qquad \qquad \text{TEMP}(r))}
 \end{array}$$

Kanonisierung von Anweisungen (I)

Kanonisierung einer Anweisung: $s \longrightarrow \vec{s}$

$$\frac{e \longrightarrow (\vec{s}, e')}{\text{MOVE}(\text{TEMP}(t), e) \longrightarrow \vec{s}, \text{MOVE}(\text{TEMP}(t), e')}$$

$$\frac{e \longrightarrow (\vec{s}, e')}{\text{MOVE}(\text{PARAM}(i), e) \longrightarrow \vec{s}, \text{MOVE}(\text{PARAM}(i), e')}$$

$$\frac{e_1 \longrightarrow (\vec{s}_1, e'_1) \quad e_2 \longrightarrow (\vec{s}_2, e'_2) \quad t, t_1, r \text{ frisch}}{\text{MOVE}(\text{MEM}(e_1), e_2) \longrightarrow \vec{s}_1, \text{MOVE}(\text{TEMP}(t), e_1), \vec{s}_2, \text{MOVE}(\text{MEM}(\text{TEMP}(t)), e'_2)}$$

$$\frac{s \longrightarrow \vec{t} \quad \text{MOVE}(e_1, e_2) \longrightarrow \vec{s}}{\text{MOVE}(\text{ESEQ}(s, e_1), e_2) \longrightarrow \vec{t}, \vec{s}}$$

Kanonisierung von Anweisungen (II)

Kanonisierung einer Anweisung: $s \longrightarrow \vec{s}$

$$\frac{e \longrightarrow (\vec{s}, e')}{\text{JUMP}(e, \vec{l}) \longrightarrow \vec{s}, \text{JUMP}(e', \vec{l})}$$

$$\frac{e_1 \longrightarrow (\vec{s}_1, e'_1) \quad e_2 \longrightarrow (\vec{s}_2, e'_2)}{\text{CJUMP}(o, e_1, e_2, l_1, l_2) \longrightarrow \vec{s}_1, \text{MOVE}(\text{TEMP}(t), e'_1), \vec{s}_2, \text{CJUMP}(o, t, e'_2, l_1, l_2)} \quad t \text{ frisch}$$

$$\frac{s_1 \longrightarrow \vec{t}_1 \quad s'_2 \longrightarrow \vec{t}_2}{\text{SEQ}(s_1, s_2) \longrightarrow \vec{t}_1, \vec{t}_2}$$

Vereinfachungen

Die angegebene Kanonisierungsmethode kann in vielen Fällen noch verbessert werden, z.B. um einfachere Ausdrücke zu liefern.

Bei der Verbesserung der Kanonisierung müssen vor allem Seiteneffekte beachtet werden.

Beispiel: Die Ausdrücke

$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2))$ und $\text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

liefern i.a. nicht das gleiche Ergebnis, z.B. wenn durch s Temporaries in e_1 verändert werden.

Die folgenden Ausdrücke liefern jedoch das gleiche Ergebnis.

$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2))$ und $\text{ESEQ}(\text{SEQ}(\text{MOVE}(\text{TEMP}(t), e_1), s),$
 $\text{BINOP}(op, \text{TEMP}(t), e_2))$

2. Basisblöcke

In vielen Compilern wird Zwischencode nicht als Liste, sondern als Menge von **Basisblöcken** gespeichert und verarbeitet.

Ein Basisblock besteht aus Instruktionen mit linearem Kontrollfluss:

- Er beginnt mit einem LABEL;
- die letzte Instruktion ist ein JUMP oder CJUMP;
- ansonsten enthält er keine JUMP oder CJUMP Instruktionen.

Der Code einer Funktion kann dann durch eine *Menge* von Basisblöcken mit einem Anfangs- und einem Endlabel gespeichert werden.

Wir verwenden Basisblöcke nur in diesem Schritt, um den Zwischencode neu anzuordnen.

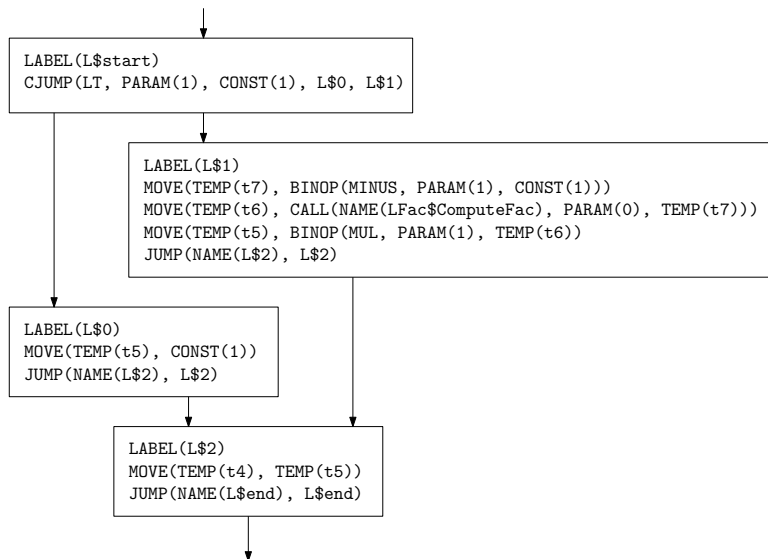
2. Basisblöcke – Beispiel

Die folgende lineare Liste von Anweisungen

```
CJUMP(LT, PARAM(1), CONST(1), L$0, L$1)
LABEL(L$1)
  MOVE(TEMP(t7), BINOP(MINUS, PARAM(1), CONST(1)))
  MOVE(TEMP(t6), CALL(NAME(LFac$ComputeFac), PARAM(0), TEMP(t7)))
  MOVE(TEMP(t5), BINOP(MUL, PARAM(1), TEMP(t6)))
  JUMP(NAME(L$2), L$2)
LABEL(L$0)
  MOVE(TEMP(t5), CONST(1))
LABEL(L$2)
  MOVE(TEMP(t4), TEMP(t5))
```

wird zu folgenden Basisblöcken...

2. Basisblöcke – Beispiel



2. Basisblöcke

Erstellung der Basisblöcke: Linearer Durchlauf über die Liste der Instruktionen, wobei ein LABEL einen neuen Basisblock beginnt und ein JUMP oder CJUMP einen Basisblock beendet.

Falls...

- einem Label kein Sprung vorhergeht, wird ein Sprung auf dieses Label eingefügt.
- einem Sprung kein Label folgt, wurde „dead code“ gefunden.
- der erste Block nicht mit einem Anfangslabel anfängt, wird eines erfunden.
- der letzte Block nicht mit einem Sprung endet, wird ein Sprung auf ein End-Label eingefügt, welches später am Ende angehängt wird.

3. Tracing

Wir ordnen die Basisblöcke nun so linear an, dass am Ende ein Programm in Normalform entsteht.

Wir beginnen mit dem Block mit Anfangslabel und hängen dann einen Block nach dem anderen an.

- Wenn der letzte angehängte Block mit einem JUMP endet, versuchen wir den Zielblock als nächstes einzufügen. Dann kann der Sprung entfernt werden.
Wenn der Zielblock schon weiter oben eingefügt wurde, bleibt es bei dem Sprung.
- Wenn der letzte Block mit einem CJUMP endet, dann:
 - Wenn der else-Block noch verfügbar ist, wird er angehängt.
 - Wenn nur noch der then-Zweig verfügbar ist, kann die CJUMP-Bedingung negiert und die Labels vertauscht werden.
 - Wenn beide Zweige nicht mehr verfügbar sind, muss ein Dummy-else-Zweig mit einem Sprung auf den echten else-Zweig eingefügt werden (damit Normalform erreicht wird).

Heutige Aufgabe

Erweitern Sie Ihren Compiler so, dass der Zwischencode nach der Übersetzung kanonisiert und nach Tracing angeordnet wird.

- Implementieren Sie die Umwandlung des Zwischencodes in Normalform.
- Im git-Repository finden Sie eine *einfache* Implementierung der Kanonisierung für Anweisungen.
- Implementieren Sie die Aufteilung von Funktionen in Basisblöcke.
- Bringen Sie das Programm durch optimierte Anordnung der Basisblöcke in Normalform.
- Sie können die Ausgabe wieder mit `tree2c` testen.

Instruktionsauswahl

Instruktionsauswahl

Ziel: Übersetzung der Zwischensprache in Assemblercode mit (beliebig vielen) abstrakten Registern.

- x86 (Pentium) Assembler in Intel Syntax
- Instruktionsauswahl als Kachelung von Bäumen
- Greedy-Algorithmus (Maximal-Munch) und optimaler Algorithmus mit dynamischer Programmierung
- Instruktionsauswahl im MiniJava-Compiler
- Implementierungsaufgabe

x86 Assembler: Register

- Der x86 (im 32bit Modus) hat 8 Register: eax, ebx, ecx, edx, esi, edi, esp, ebp
- esp ist der *stack pointer*; nicht anderweitig verwendbar.
- ebp kann als *frame pointer* benutzt werden. Wird kein frame pointer benötigt, so ist ebp allgemein verfügbar.
- eax, edx, ecx sind *caller-save*
- ebx, esi, edi, ebp sind *callee-save*

x86 Assembler: Operanden

Ein *Operand* ist entweder

- ein Register, oder
- eine Konstante, oder
- eine Speicherstelle (effective address).

Eine *Konstante* wird geschrieben als n wobei n eine Zahl ist. Der *Assembler* gestattet hier auch arithmetische Ausdrücke mit Konstanten und Abkürzungen.

Eine *Speicherstelle* wird geschrieben als $[reg_1 + reg_2 * s + n]$, wobei

- n eine konstante Zahl,
- $s \in \{1, 2, 4, 8\}$,
- $reg_{1,2}$ Register sind ($reg_2 \neq esp$).

Die Art des Inhalts kann mit dem Vorsatz BYTE/WORD/DWORD PTR disambiguiert werden.

x86 Assembler: Verschiebepfehle

Ein Verschiebepfehl hat die Form

```
mov      dst, src
```

Er verschiebt den Inhalt von src nach dst. Hier kann src ein beliebiger Operand sein; dst darf natürlich keine Konstante sein. Außerdem dürfen src und dst nicht beide zugleich Speicherstellen sein.

```
mov      [esp+48-4], ebx  
mov      DWORD PTR [eax], 33  
mov      esi, [ecx]
```

x86 Assembler: Arithmetische Befehle

Instruktionen für Addition, Subtraktion und Multiplikation sind:

add	dst, src
sub	dst, src
imul	dst, src

Der Effekt ist jeweils:

$$\begin{aligned} dst &\leftarrow dst + src \\ dst &\leftarrow dst - src \\ dst &\leftarrow dst * src \end{aligned}$$

Die Operanden sind denselben Einschränkungen wie bei den Verschiebepfehlen unterworfen. Bei `imul` muss `dst` ein Register sein.

x86 Assembler: Multiplikation (Variante)

Es gibt auch eine einstellige Multiplikation:

```
imul    src
```

Der erste Operand (32 Bit) muss sich in `eax` befinden, der zweite Operand (wieder 32 Bit) wird als `src` übergeben.

Das Ergebnis (64 Bit) liegt dann in `edx:eax`.

Falls das Ergebnis in 32 Bit passt, ist `eax` der korrekte Wert. Für unsere Zwecke ist also die einstellige Form auch passend.

x86 Assembler: Division

Das Format des Divisionsbefehls ist:

```
cdq  
idiv    src
```

Dies platziert das Ergebnis der Division von `eax` durch `src` in `eax`.
Der Rest der Division wird in das Register `edx` geschrieben.

Im Divisionsbefehl darf `src` keine Konstante sein.

x86 Assembler: Sonderbefehle

Als Beispiel für einen der vielen Sonderbefehle geben wir hier den nützlichen Befehl *load effective address*:

```
lea    dst, [reg1 + reg2 * s + n]
```

(Hierbei ist n eine Konstante oder ein Register plus eine Konstante und $s \in \{1, 2, 4, 8\}$).

Das Ergebnis von *load effective address* ist eine simple Zuweisung

$$\text{dst} \leftarrow \text{reg1} + \text{reg2} \times s + n$$

Anstelle eine Speicherzelle zu laden, wird hier nur ihre Adresse berechnet.

x86 Assembler: Bitverschiebepfehle

Diese haben die Form

sop dst, n

sop ist entweder eine Bitverschiebung auf vorzeichenfreien, shl (left shift) und shr (right shift), oder vorzeichenbehafteten Ganzzahloperanden, sal (arithmetic left shift, synonym zu shl) und sar (arithmetic right shift). Der Operand op wird um n Bits verschoben.

Beispiele:

- Multiplikation von eax mit 4:

```
shl     eax, 2
```

- Die Instruktion cdq kann implementiert werden durch:

```
mov     edx, eax  
sar     edx, 31
```

x86: Labels und unbedingte Sprünge

Sprungadressen können im Assemblercode symbolisch durch einen Bezeichners mit Doppelpunkt markiert (*labelled*) werden.

```
f:      sub     esp, 24
L73:
      mov     eax, 35
```

Nach dem Doppelpunkt ist ein Zeilenumbruch erlaubt.

Ein unbedingter Sprungbefehl hat die Form

`jmp label`

x86 Assembler: Bedingte Sprünge

Ein bedingter Sprung hat die Form:

```
cmp op1 , op2  
jop label
```

Hier ist *jop* eines von je (equal), jne (not equal), jl (less than), jg (greater than), jle (less or equal), jge (greater or equal), jb (unsigned less), jbe (unsigned less or equal), ja (unsigned greater), jae (unsigned greater or equal).

Es wird gesprungen, falls die entsprechende Bedingung zutrifft, ansonsten an der nächsten Instruktion weitergearbeitet.

x86 Assembler: Funktionsaufruf

Die Instruktion

`call label`

legt die Rücksprungadresse auf den Stack, subtrahiert also 4 vom Stack-Pointer, und springt dann nach *label* und

Die Instruktion

`ret`

springt zur Adresse ganz oben auf dem Stack und addiert 4 zum Stack-Pointer.

Parameterübergabe und Rückgabe von Funktionsergebnissen wird entsprechend der Calling-Convention durchgeführt.

Funktionsprolog und -epilog

Prolog: Sichern von ebp und Platz für n lokale Variablen:

```
push    ebp
mov     ebp, esp
sub     esp, 4n
```

Epilog: Entfernen der lokalen Variablen, Rücksichern von %ebp, Rücksprung:

```
mov     esp, ebp
pop     ebp
ret
```

Für diese Befehlsfolgen gibt es sogar eigene Instruktionen (enter 4n und leave). Diese sind kürzer im Binärcode, aber teilweise langsamer. (Siehe http://en.wikipedia.org/wiki/X86_assembly_language.)

Assemblerprogramme und runtime.c

Beispiel:

```
.intel_syntax noprefix  
.global Lmain
```

Lmain:

```
    push ebp  
    mov ebp, esp
```

```
    push 23  
    call L_println_int  
    add esp, 4
```

```
    mov esp, ebp  
    pop ebp  
    ret
```

Übersetzen und Linken mit:

```
gcc -m32 -o runme prog.s runtime.c
```

Assemblerprogramme und runtime.c

Die Direktiven

```
.intel_syntax noprefix  
.global Lmain
```

sorgen dafür, dass der Assembler Intel Syntax liest und dass Lmain in der Symboltabelle der compilierten Objektdatei erscheint.

Im Assembler können Funktionen mit `call` aufgerufen werden, auch in C geschriebene. Argumente für diese werden in `[esp]`, `[esp+4]`, `[esp+8]` übergeben.

Unter Windows und macOS stellen C-Compiler allen Funktionsnamen einen Unterstrich voran.

Übersetzung von Zwischencode in Assemblercode

Wir übersetzen zunächst in Assemblercode, in dem auch Temporaries als Register verwendet werden können.

Maschinenprogramme haben die gleiche Struktur wie die Programme der Zwischensprache.

- Ein Maschinenprogramm besteht aus einer Liste von Maschinenfunktionen.
- Eine Maschinenfunktion besteht aus einer Liste von Maschineninstruktionen (= Assemblerbefehle).

Bei der Übersetzung muss jede Funktion der Zwischensprache in eine Maschinenfunktion übersetzt werden.

Übersetzung einer Zwischensprachenfunktion

F\$method:

```
push    ebp
mov     ebp, esp
; brauchen noch keinen Platz im Frame
```

Funktion

```
F$method(n){
    body
    return t
}
```

```
mov     t1, ebx    ; callee-save
mov     t2, esi    ; Register
mov     t3, edi    ; speichern

translate(body)
mov     eax, t
```

wird zu:

```
mov     ebx, t1    ; callee-save
mov     esi, t2    ; Register
mov     edi, t3    ; wiederherstellen

mov     esp, ebp
pop     ebp
ret
```

Instruktionsauswahl

Die Liste der Anweisungen im Funktionsrumpf muss durch Assemblerinstruktionen implementiert werden.

Dazu müssen alle TreeExp-Ausdrücke und TreeStm-Anweisungen in Assembler übersetzt werden.

Ansätze zur Erzeugung effizienten Maschinencodes:

1. Ordne den Assemblerinstruktionen Baummuster zu und zerteile den Baum in diese Muster.
2. Übersetze den Baum auf naive Weise in Assemblerinstruktionen und wende dann lokale Optimierungen auf dem Code an (peephole optimization).

Beide Ansätze sind in der Praxis erfolgreich. Wir betrachten hier den ersten.

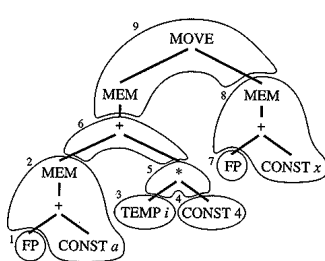
Instruktionsauswahl durch Baumkachelung

Ordne Assemblerinstruktionen Baummuster zu:

Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\ \quad \quad \quad \\ + \quad + \quad \text{CONST} \quad \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\ \quad \end{array}$

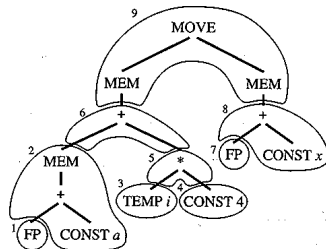
Instruktionsauswahl durch Baumkachelung

Übersetze einen gegebenen Baum durch Kachelung mit den Baummustern. Schreibe die Ergebnisse jeweils in Temporaries.



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)



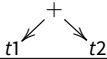

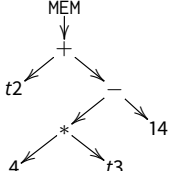
2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

Instruktionsauswahl durch Baumkachelung

Eine sinnvolle Wahl der Baummuster sollte es ermöglichen, jeden Baum damit zu kacheln.

Bei Prozessoren mit eingeschränkten Instruktionen wie x86 kann es nötig sein, zusätzliche Kopierinstruktionen zu benutzen.

<code>mov r, t1</code> <code>add r, t2</code>	 <pre>graph BT; plus[+] --> t1; plus --> t2</pre>
<code>mov eax, t1</code> <code>imul t2</code> <code>mov r, eax</code>	 <pre>graph BT; mul[*] --> t1; mul --> t2</pre>
<code>mov r, DWORD PTR [t2 + 4*t3 - 14]</code>	 <pre>graph BT; MEM[MEM] --> plus[+]; plus --> t2; plus --> minus[-]; minus --> mul[*]; mul --> 4; mul --> t3; minus --> 14</pre>

Algorithmen zur Baumkachelung

Ziel ist nun, eine möglichst gute Kachelung für einen gegebenen IR-Baum zu berechnen.

Der zu minimierende Preis einer Kachelung sei hier zunächst einfach die Anzahl der Kacheln.

Maximal Munch

- Greedy-Algorithmus
- Wähle die größte Kachel, die an der Wurzel des Baumes passt.
- Wende den Algorithmus rekursiv für die Teilbäume an den Blättern der Kachel an.

Dieser Algorithmus findet eine Kachelung, die sich nicht durch Ersetzung zweier benachbarter Kacheln durch eine einzige verbessern lässt.

Die so gefundene Kachelung muss aber nicht optimal sein.

Algorithmen zur Baumkachelung

Optimale Kachelung durch dynamisches Programmieren

- Lege Tabelle an, in der für jeden Knoten des gegebenen IR-Baumes eine optimale Kachelung (mit minimalen Kosten) abgelegt werden soll.
- Fülle Tabelle bottom-up oder rekursiv: Blätter sind klar. Bei inneren Knoten werden alle passenden Kacheln probiert. Die optimalen Kachelungen der Unterbäume an den Blättern der Kacheln sind schon in der Tabelle eingetragen.
- Die Kosten können für jede Kachel einzeln festgelegt werden. Es wird eine Kachelung mit minimalen Kosten gefunden.
- Das Prinzip ist recht leicht; die Implementierung aber etwas aufwendig, weswegen es auch für diese Aufgabe codeerzeugende Werkzeuge gibt („code generator generator“). Diese lösen sogar eine etwas allgemeinere Aufgabe, die sich stellt, wenn Register nicht allgemein verwendbar sind.

Instruktionsauswahl im MiniJava-Compiler

Wir verwenden drei *Interfaces* zur Repräsentierung von Assembler-Programmen:

- MachinePrg: ein ganzes Assembler-Programm
- MachineFunction: eine einzige in Assembler übersetzte Funktion
- MachineInstruction: eine einzige Assemblerinstruktion

Ein weiteres Interface CodeGenerator stellt die eigentliche Übersetzungsfunktion (von TreePrg nach MachinePrg) bereit.

Wir verwenden Interfaces (und nicht Klassen wie in der Tree-Zwischensprache), weil es im Compiler verschiedene Backends für verschiedene Architekturen geben kann.

Jede Architektur implementiert diese Interfaces geeignet. Gegeben sind rudimentäre Implementierungen I386Prg, I386Function und Klassen für x86-Assemblerinstruktionen.

Konkrete Instruktionsklassen für x86

Wir modellieren Assemblerinstruktionen durch Klassen, die das Interface `MachineInstruction` implementieren.

Beispiel:

```
class InstrBinary implements MachineInstruction {  
    enum Kind {MOV, ADD, SUB, SHL, SHR, SAL, SAR, AND, OR, ...}  
  
    // Konstruktor  
    InstrBinary(Kind kind, Operand dst, Operand src) {  
        ...  
    }  
}
```

Dabei ist `Operand` eine abstrakte Klasse mit drei konkreten Unterklassen `Imm` (immediate), `Reg` (register) und `Mem` (memory), welche die verschiedenen möglichen Operanden repräsentieren.

Benutzung:

```
emit(new InstrBinary(MOV, new Reg(eax), new Reg(1)));  
emit(new InstrUnary(IMUL, new Reg(r)));  
emit(new InstrBinary(MOV, new Reg(t), new Reg(eax)));
```

Maschinenregister

Für die Maschinenregister `eax`, `ebx`, `ecx`, . . . sollten spezielle Temp-Objekte reserviert werden.

Diese sollen zunächst nur benutzt werden, wenn dies absolut unvermeidbar ist, z.B. weil `idiv` sein Ergebnis immer in `eax` zurückliefert.

Ansonsten verwenden wir immer frische Temporaries. Ergebnisse, die in speziellen Maschinenregistern zurückgegeben werden, werden sofort in Temporaries umkopiert.

Erst bei der Registerverteilung werden diese Temporaries dann auf die entsprechenden Maschinenregister abgebildet. Unnötige Zuweisungen können dort entfernt werden.

Implementierung von Maximal Munch

Wir schreiben zwei wechselseitig rekursive Methoden:

```
Operand munchExp(TreeExp exp);  
void munchStm(TreeStm stm);
```

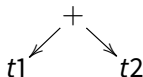
Diese erzeugen als Seiteneffekt eine Liste von Assemblerinstruktionen (mit `void emit(MachineInstruction i);`).

Der von `munchExp` zurückgegebene Operand (Register, Speicher, oder Konstante) enthält nach Ausführung der mit `emit` erzeugten Instruktionssequenz den Wert des Ausdrucks `exp`.

Die Funktionen `munchExp` und `munchStm` wählen jeweils diejenige Kachel aus, die von der Wurzel des übergebenen Baums das meiste abdeckt. Die beiden Funktionen rufen dann rekursiv die entsprechende Funktion für Teilbäume an den Blättern der Kachel auf. Damit werden Assemblerinstruktionen für die Teilbäume erzeugt. Danach werden die Assemblerinstruktionen der gewählten Kachel ausgegeben.

Implementierung von Maximal Munch

Hinweis: Implementieren Sie zunächst den einfachsten Fall von minimalen Kacheln aus nur einem Knoten.



Das entspricht dann wieder einer Übersetzung durch einen einzigen Visitor-Durchlauf:

Für jede Art von TreeExp bzw. TreeStm

- Übersetze die direkten Kindknoten rekursiv.
- Mit den Ergebnissen kann dann Code für die Wurzel des Baums erzeugt werden.

Kompliziertere Kacheln können später noch speziell behandelt werden.

Hinweis zu Funktionsaufrufe

Funktionsaufrufe können als `call` Instruktion übersetzt werden.

Wir nehmen an, dass alle Temporaries, die keine Maschinenregister sind, durch einen Funktionsaufruf unverändert bleiben.

Wir verlassen uns darauf, dass sich alle Funktionen an die Calling-Convention halten und Caller-Save-Register nicht verändern (oder wiederherstellen).

Die Callee-Save-Register sollten im Funktionsrumpf in Temporaries gespeichert und am Ende wiederhergestellt werden. Das ermöglicht es dem Registerallocator, diese Register im Bedarfsfall auch anderweitig zu nutzen (es werden nur Temporaries auf Maschinenregister verteilt, nicht aber Maschinenregister umkopiert).

Programmieraufgabe

Implementieren Sie die Übersetzung von der Zwischensprache in Maschinencode (mit unendlich vielen Registern).

- Insgesamt soll ein `TreePrg` in ein `MachinePrg` übersetzt werden.
- Klassen zur Repräsentierung von x86-Assemblerprogrammen finden Sie im git-Repository.
- Es muss die Methode `codeGen` in `I386CodeGenerator` implementiert werden.

Einen Simulator für x86-Assemblerprogramme mit unendlich vielen Registern finden Sie ebenfalls im git-Repository.

- Der Simulator implementiert nur einen kleinen Teil der Intel-Assemblerbefehle.
- Der Simulator erlaubt keine Zugriffe auf uninitialisierte Speicherstellen usw., zur leichteren Fehlersuche.
- Eine Trace kann mit Option `-v` angezeigt werden.

Optimierungsmöglichkeiten bei der Instruktionsauswahl

Instruktionsauswahl

Einige Möglichkeiten zur Optimierung bei der Instruktionsauswahl

- Komplexe Instruktionen: `leaq`
- Division durch Zweierpotenzen
- Division durch Konstanten

Komplexe Instruktionen: lea

Beispiele:

- Multiplikation und Addition in einem (nützlich z.B. für Arrayzugriffe).

```
lea ecx, [ebx + 4*eax]
```

- Addition zweiter Register in ein drittes:

```
lea eax, [ebx + ecx]
```

- Multiplikation mit 2, 3, 4, 5, 8 und 9 (mit niedrigerer Latenz als imul).

```
lea eax, [2*eax]           # eax := 2 * eax
```

```
lea eax, [eax + 2*eax]     # eax := 3 * eax
```

```
lea eax, [4*eax]           # eax := 4 * eax
```

```
lea eax, [eax + 4*eax]     # eax := 5 * eax
```

```
lea eax, [8*eax]           # eax := 8 * eax
```

```
lea eax, [eax + 8*eax]     # eax := 9 * eax
```

Komplexe Instruktionen: lea

Beachte: Seit Sandy Bridge (2011) ist die lea-Instruktionen mit drei Operanden auf Intel-Prozessoren weniger effizient als die anderen Varianten.

```
lea t1, [t2 + 4*t2 + 23]
```

Dann ist folgende Instruktionssequenz besser:

```
lea t1, [t2 + 4*t2]  
add t1, 23
```

Komplexe Instruktionen: lea

Wie soll man mit komplexen Instruktionen im Compiler umgehen?

Problem: große Anzahl von Fällen bei der Mustererkennung

$$4 + t \implies [t + 4]$$

$$4 * t + 2 \implies [4*t + 2]$$

$$4 * t + 2 - (1 * t + 3) \implies [3*t - 1]$$

$$(t + 1) + t' + (t + 1) \implies [t' + 2*t + 2]$$

\vdots

Normalformen

Zur Reduktion der Anzahl der Fälle, ist es oft günstig ein Programm in eine geeignete Normalform zu bringen.

Eine Linearkombination hat die Form

$$c_1 \cdot t_1 + \dots + c_k \cdot t_k + c \ ,$$

wobei c und alle c_i Integer und die t_i Temporaries sind.

- Addition:
 $(4t_1 + 5t_2 + 4) + (4t_1 + 5t_3 + 5) = (8t_1 + 5t_2 + 5t_3 + 9)$
- Multiplikation:
 $(4t_1 + 5t_2 + 4) * 5 = (20t_1 + 25t_2 + 20)$
 $(4t_1 + 5t_2 + 4) * (4t_1 + 5t_3 + 5) = \text{keine Linearkomb.}$
- Speicherung z.B. als HashMap, die jedem Temporary t_i den Koeffizienten c_i zuordnet.

Normalformen

Mit einem Datentyp für Linearkombinationen kann man TreeExp-Ausdrücke in Linearkombinationen normalisieren.

Versuche eine TreeExp in eine Linearkombination von Temporaries zu überführen.

Beispiele:

- $\text{lincomb}(\text{CONST}(n)) = n$
- $\text{lincomb}(\text{BINOP}(\text{PLUS}(e_1, e_2))) = \text{lincomb}(e_1) + \text{lincomb}(e_2)$
- $\text{lincomb}(\text{CALL}(\dots)) = \text{fail}$; ist keine Linearkombination

Wenn man einen Ausdruck so in eine Linearkombination überführen kann, dann kann man daraus leicht(er) komplexe Instruktionen wie `lea` erzeugen.

Die Normalisierung in Linearkombinationen könnte auch als Teil der Kanonisierung durchgeführt werden.

Verwendung der Normalform

Verwendung von `lea`:

- Einfachster Fall:
Verwende `lea` nur für Linearkombinationen, die direkt dargestellt werden können.
 - $s + 4t$ wird zu `lea u, [s + 4*t]`
 - $s + 5t$ wird ohne `lea` übersetzt.
- Kompliziertere Linearkombinationen könnten gezielt in mehrere Instruktionen zerlegt werden.

Beispiel: $s + 4t + 2r$ könnte zu

`lea u, [s + 4t]` $\# u := s + 4t$

`lea u, [u + 2r]` $\# u := u + 4r$

werden.

Speicherzugriffe:

Linearkombinationen sind auch nützlich bei der Instruktionsauswahl für `MEM` und anderen Speicherzugriffen.

`mov u, [t + 4*i]` $\# u := \text{MEM}(t + 4i)$

Optimierung der Division

Verglichen mit anderen arithmetischen Operationen ist die `idiv`-Instruktion besonders teuer und sollte vermieden werden.

Division durch Konstanten kann mit Schiebeoperationen und Multiplikation implementiert werden.

Beispiel: gcc generiert für “`x/7`” folgenden Code

```
mov  edx, 613566757
mov  eax, ecx
mul  edx
mov  eax, ecx
sub  eax, edx
shr  eax
add  eax, edx
shr  eax, 2
```

(auf meinem Rechner etwa doppelt so schnell wie `idiv`)

Division durch Zweierpotenzen

Unsigned Integer

Division durch 2^k entspricht Rechtsshift um k Stellen.

```
mov eax, 123
shr eax, 2
# Wert von eax ist jetzt 123/4 = 30
```

Signed Integer

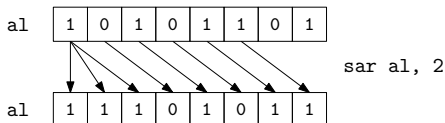
Für negative Zahlen (Zweierkomplementdarstellung) ist shr inkorrekt, da links mit Nullen aufgefüllt wird.

```
mov eax, -123
shr eax, 2
# Wert von eax ist jetzt 1073741793
```

Division durch Zweierpotenzen

Signed Integer

Die *arithmetische Shift-Instruktion* `sar` füllt links mit dem Wert des höchstwertigen Bits auf.



```
mov eax, 123
sar eax, 2
# Wert von eax ist jetzt 30
mov eax, -123
sar eax, 2
# Wert von eax ist jetzt -31
```

Problem:

Es wird immer in Richtung $-\infty$ gerundet. Java rundet in Richtung 0.

Division durch Zweierpotenzen

Signed Integer

Um in Richtung 0 zu runden, kann man bei negativen Zahlen vor der Division $2^k - 1$ addieren.

Division durch 2^k mit Runden in Richtung 0:

```
mov  eax, n
mov  ecx, eax
sar  ecx, 31
and  ecx, (2^k-1)
add  eax, ecx
sar  eax, k
```

Division durch Konstanten

Wir betrachten jetzt die Division durch andere Konstanten und wollen z.B. Division durch 7 folgendermaßen implementieren.

```
# Wert n ist in ecx
mov edx, 613566757
mov eax, ecx
mul edx
mov eax, ecx
sub eax, edx
shr eax
add eax, edx
shr eax, 2
```

Betrachte zuerst den Fall für **unsigned Integer**.

Division durch Konstanten (Unsigned, 32 Bit)

Idee: Division durch Multiplikation mit dem Inversen.

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{1}{d} \cdot n \right\rfloor = \left\lfloor \frac{2^k}{d} \cdot \frac{n}{2^k} \right\rfloor$$

Wir möchten aber Integer-Multiplikation verwenden.

Betrachte

$$\left\lfloor m \cdot \frac{n}{2^k} \right\rfloor \quad \text{für} \quad m := \left\lceil \frac{2^k}{d} \right\rceil .$$

Die Zahl m ist größer-gleich dem eigentlichen Faktor, aber danach wird abgerundet.

Indem wir k groß genug wählen, können wir

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{2^k}{d} \cdot \frac{n}{2^k} \right\rfloor = \left\lfloor m \cdot \frac{n}{2^k} \right\rfloor$$

für alle (unsigned) 32-Bit-Zahlen n erreichen.

Division durch Konstanten (Unsigned, 32 Bit)

Wie groß muss k sein, damit für alle 32-Bit n gilt:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor m \cdot \frac{n}{2^k} \right\rfloor?$$

Die Differenz $\frac{n}{d} - \left\lfloor \frac{n}{d} \right\rfloor$ ist höchstens $\frac{d-1}{d}$.

Um die Differenz $m \cdot \frac{n}{2^k} - \left\lfloor \frac{n}{d} \right\rfloor$ abzuschätzen, ist es hilfreich, zunächst m als $\frac{2^k + e}{d}$ schreiben. Wegen $m = \left\lceil \frac{2^k}{d} \right\rceil$ gilt $0 \leq e < d$.
Damit:

$$\begin{aligned} m \cdot \frac{n}{2^k} - \left\lfloor \frac{n}{d} \right\rfloor &= \left(\frac{2^k + e}{d} \right) \cdot \frac{n}{2^k} - \left\lfloor \frac{n}{d} \right\rfloor = \frac{n}{d} + \frac{e \cdot n}{d \cdot 2^k} - \left\lfloor \frac{n}{d} \right\rfloor \\ &< \frac{d-1}{d} + \frac{e \cdot n}{d \cdot 2^k} < \frac{d-1}{d} + \frac{d \cdot n}{d \cdot 2^k} \\ &= \frac{d-1}{d} + \frac{n}{2^k} \end{aligned}$$

Die gewünschte Gleichung gilt falls $m \cdot \frac{n}{2^k} - \lfloor \frac{n}{d} \rfloor < 1$.
Dafür genügt zu zeigen:

$$\begin{aligned} & \frac{d-1}{d} + \frac{n}{2^k} < 1 \\ \iff & d-1 + \frac{d \cdot n}{2^k} < d \\ \iff & \frac{d \cdot n}{2^k} < 1 \\ \iff & d \cdot n < 2^k \end{aligned}$$

Da n eine 32-Bit-Zahl ist, ist dafür Folgendes hinreichend:

$$d \cdot 2^{32} < 2^k$$

Wegen $d < 2^{\lceil \log_2 d \rceil}$ genügt es also

$$k := 32 + \lceil \log_2 d \rceil$$

zu wählen.

Zwischenzusammenfassung (Unsigned, 32 Bit)

Wenn wir für eine gegebene Zahl $d \geq 0$

$$p := \lceil \log_2 d \rceil$$

$$m := \left\lceil \frac{2^{32+p}}{d} \right\rceil$$

setzen, dann gilt

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor \frac{m \cdot n}{2^{32+p}} \right\rfloor$$

für jede nichtnegative 32-Bit-Zahl n .

Wir können also die Division durch eine Integer-Multiplikation und eine Shift-Operation implementieren.

Wir nehmen im Folgenden an, dass d keine Zweierpotenz ist.

Implementierung (Unsigned, 32 Bit)

Wie viele Bits brauchen wir für die Multiplikation?

Wenn d keine Zweierpotenz ist, dann gilt $2^{32} < m < 2^{33}$.

Denn aus $2^{\lfloor \log_2 d \rfloor} < d < 2^{\lceil \log_2 d \rceil}$ folgt

$$2^{32} = \left\lceil \frac{2^k}{2^{\lceil \log_2 d \rceil}} \right\rceil < m < \left\lceil \frac{2^k}{2^{\lfloor \log_2 d \rfloor}} \right\rceil = 2^{33}.$$

Das bedeutet also, dass m eine 33-Bit-Zahl ist, in der das oberste Bit gesetzt ist. D.h.

$$m = 2^{32} + m'$$

für eine 32-Bit-Zahl m' .

Für die Implementierung der von $m \cdot n$ ist es günstig, zuerst $m' \cdot n$ zu berechnen (64-Bit-Multiplikation) und den fehlenden Teil $2^{32} \cdot n$ später noch zu addieren.

Implementierung (Unsigned, 32 Bit)

Die Implementierung auf der Maschine benutzt m' :

$$\left\lfloor \frac{m \cdot n}{2^{32+p}} \right\rfloor = \left\lfloor \frac{m' \cdot n + 2^{32} \cdot n}{2^{32+p}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{m' \cdot n}{2^{32}} \right\rfloor + n}{2^p} \right\rfloor$$

Algorithmus für Division durch d :

- Die Werte von p und $m' = m - 2^{32}$ sind vorberechnet.
- $x_1 := \lfloor (m' \cdot n) / 2^{32} \rfloor$
- $x_2 := \lfloor (x_1 + n) / 2 \rfloor$
- $x_3 := \lfloor x_2 / 2^{p-1} \rfloor$

Das Ergebnis ist dann in x_3 .

Bei der Berechnung von x_2 muss man aufpassen, da die Addition überlaufen könnte. Deswegen wird dort schon durch 2 geteilt. Man kann $\lfloor (x_1 + n) / 2 \rfloor$ überlauffrei durch $x_1 + \lfloor (n - x_1) / 2 \rfloor$ berechnen.

Implementierung in x86-Assembler

Beispiel: $d = 7$

$$p := \lceil \log_2 d \rceil = 3 \quad m' := \left\lceil \frac{2^{32+p}}{d} \right\rceil - 2^{32} = 613566757$$

Division durch 7 wird implementiert durch:

```
# Anfangs: ecx = n
mov edx, 613566757
mov eax, ecx
mul edx          # edx := m' * n / 2^32 = x1
mov eax, ecx
sub eax, edx     # eax := n - x1
shr eax          # eax := (n - x1) / 2
add eax, edx     # eax := x1 + (n - x1) / 2 = x2
shr eax, 2       # eax := x2 / 2^2 = x3 = n / 7
```

Division durch Konstante (Signed, 32 Bit)

MiniJava verwendet [signed Integer](#). Auch zur Division vorzeichenbehafteter Integer kann man die gleiche Idee verwenden.

Für alle $2 < d < 2^{32} - 1$ gibt es Zahlen $0 \leq m < 2^{32}$ und $k \geq 32$, so dass gilt:

$$\left\lfloor \frac{m \cdot n}{2^k} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \quad \text{falls } 0 \leq n < 2^{31}$$

$$\left\lfloor \frac{m \cdot n}{2^k} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \quad \text{falls } -2^{31} \leq n < 0$$

Für $k := 31 + \lceil \log_2 d \rceil$ und $m := \lceil 2^k / d \rceil$ ist diese Aussage erfüllt.

Details siehe: [\[Warren, Hacker's Delight, Addison Wesley 2013\]](#)

Division durch Konstante (Signed, 32 Bit)

Berechnung des *kleinsten* m mit der gewünschten Eigenschaft:

Beginne mit $k = 32$ und inkrementiere k so lange, bis

$$2^k > n_c \cdot (d - (2^k \bmod d))$$

gilt, wobei $n_c = \lfloor 2^{31}/d \rfloor \cdot d - 1$.

Dann hat man k und kann m wie bisher berechnen:

$$m = \left\lceil \frac{2^k}{d} \right\rceil$$

Beispiel: Im Fall $d = 3$ erhalten wir $k = 32$, obwohl $31 + \lceil \log_2 3 \rceil = 33$. Wir können bei der Implementierung der Division durch 3 also ohne Shift-Instruktion am Ende auskommen.

Division durch Konstante (Signed, 32 Bit)

Die Implementierung erfolgt jetzt wieder analog zu vorher.

Ist der Divisor negativ, so muss das Ergebnis inkrementiert werden, um die korrekte Rundung zu erreichen.

Das einzige Problem ist, dass wir $0 \leq m < 2^{32}$ haben, aber mit vorzeichenbehafteten Integer nur Werte von -2^{31} bis $2^{31} - 1$ speichern können.

Wir betrachten deshalb die folgenden zwei Fälle.

Fall: $0 \leq m < 2^{31}$

Im Fall $0 \leq m < 2^{31}$ brauchen wir einfach nur vorzeichenbehaftete Operationen (imul und sar statt mul und shr) zu verwenden.

Division durch $d = 65$:

Es gilt $k = 37$ und $m = 2114445439 < 2^{31}$.

Anfangs: ecx = n

```
mov eax, ecx
```

```
mov edx, 2114445439
```

```
imul edx           # edx = m * n / 2^32
```

```
sar edx, 5         # edx = m * n / 2^37
```

```
sar ecx, 31        # addiere 1 falls n < 0
```

```
sub edx, ecx       # Ergebnis in edx
```

Fall: $2^{31} \leq m < 2^{32}$

Andernfalls multiplizieren wir n mit $m' = m - 2^{32}$ und addieren danach noch $2^{32}n$ zur Korrektur. Überlauf ist kein Problem, da n und $m' \cdot n$ unterschiedliche Vorzeichen haben.

Beachte $-2^{31} \leq m' < 2^{31}$, also kann man m' im Gegensatz zu m als Signed Integer repräsentieren.

Division durch $d = 187$: Es gilt $k = 39$ und $m = 2939870663 \geq 2^{31}$. Benutze $m' = m - 2^{32} = -1355096633$.

```
# Anfangs: ecx = n
mov eax, ecx
mov edx, -1355096633
imul edx                # edx = m' * n / 2^32
add edx, ecx            # edx = m * n / 2^32
sar edx, 7              # edx = m * n / 2^39
sar ecx, 31             # addiere 1 falls n < 0
sub edx, ecx            # Ergebnis in edx
```


Optionale Aufgabe

Implementieren Sie Optimierungen in der Instruktionsauswahl.

- Effektive Adressen
- Verwendung von `lea`.
- Division durch Zweierpotenzen
- Multiplikation mit Zweierpotenzen
- Division durch Konstanten
- ...

Weitere Optimierungen, z.B. Instruction Scheduling, benötigen größeren zusätzlichen Aufwand.