Kompilatorbau

ein praxisorientierter Kurs

Sven Eric Panitz
Hochschule RheinMain

Dieses Skript begleitet das Wahlmodul Compilerbau des Bachelorstudiengangs »Angewandte Informatik«. Das Modul besteht aus zwei Lehrveranstaltungen zu je 2SWS: einem Praktikum und einer Vorlesung. Zum Leistungserwerb ist eine praktische Tätigkeit mit Fachgespräch durchzuführen.

Der Kurs richtet sein Hauptaugenmerk auf die Durchführung eines eigenen kleinen Compilerbauprojektes. Theoretische Grundlagen aus dem Gebiet formale Sprachen werden in diesem Kurs nur gestreift. Diese werden insbesondere im Pflichtmodul zu formalen Sprachen und Automatentheorie vorgestellt.

Dieses Skript besteht hauptsächlich aus kommentierten Quelltext. Dieser Quelltext ist separat auch von der Webseite ladbar.

Inhaltsverzeichnis

Kapitel 1

Einführung

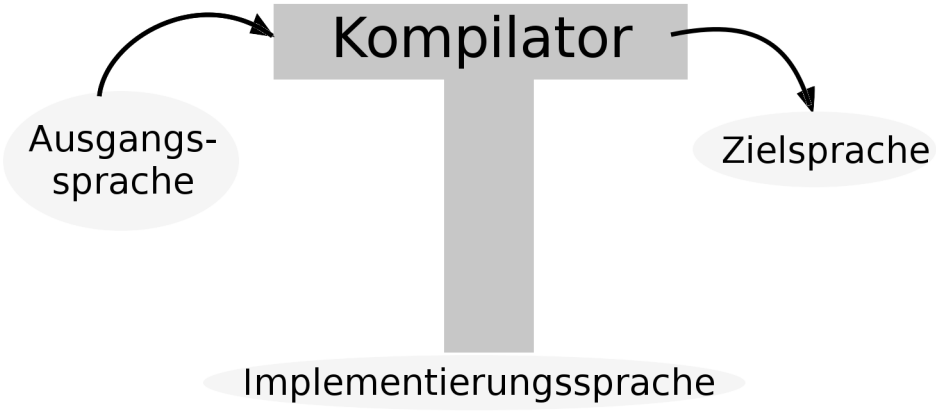
1.1 Der Name des Spiels

Das Fach, das in der englischen Sprache als *›compiler construction‹* bezeichnet wird, hat in den deutschen Stundenplänen sich unter den Namen *›Übersetzerbau‹* etabliert. Da hier aber für Außenstehende schnell ein Übersetzungsprogramm zwischen natürlichen Sprachen assoziiert wird, findet sich häufiger die deutsch-englisch Mischbezeichnung *›Compilerbau‹*.

Für das in diesem Fach erzeugte Programm, hat sich kein eigener deutscher Ausdruck etabliert. ›Übersetzer‹ wird so gut wie nicht benutzt. Auch hier bedient man sich des englischen Worts ›compiler‹. Dabei wäre es ein einfaches, hierfür ein analoges deutschen Fremdwort zu benutzen, so wie es sich im Französischen mit ›compilateur‹ durchgesetzt hat. Möchte man auf deutsch-englisches Sprachmischmasch verzichten, bietet sich ›Kompilator‹ an. Leider ist dieser Ausdruck in der deutschen Literatur nicht eingeführt.

1.2 Aufgabe eines Kompilators

Ein Kompilator ist ein Programm, das Programme die in einer bestimmte Quell- oder Ausgangssprache geschrieben sind übersetzt in ein Programm in einer bestimmten Zielsprache. Da der Kompilator selbst in einer Programmiersprache geschrieben ist, spielt diese Implementierungssprache auch noch eine bestimmte Rolle. Somit lässt sich ein Kompilator als ein T illustrieren (was auch für *translate* stehen kann), an dessen drei Enden sich eine Programmiersprache steht.



In den meisten Kompilatoren ist die Quellsprache eine höhere Programmiersprache, wobei sich in den letzten Jahrzehnten die Auffassung, was eine höhere Programmiersprache ist, doch durchaus gewandelt hat. Hatten erste Kompilatoren als Quellsprache einfache Programmiersprachen die gerade mal Funktionsaufrufe und einfache Konstrukte zum Programmfluss, wie Verzweigungen und einfache Schleifen kannten, gibt es heute Programmiersprachen mit einer Vielzahl von Konstrukten aus der objektorientierten und funktionalen Programmierung. Je komplexer die Ausdrucksstärke der Quellsprache ist, desto komplexer ist es entsprechend auch, einen Kompilator für diese Sprache zu schreiben.

Die Zielsprache der Übersetzung ist meistens eine Maschinensprache, die aus Maschinenbefehlen besteht. Sei es der Befehlssatz für eine konkrete Maschine oder aber auch der Befehlssatz einer gedachten, virtuellen (auch abstrakten) Maschine.

Schließlich spielt noch eine Rolle, in welcher Sprache der Kompilator implementiert ist. Jede Kompilator, der etwas auf sich hält, ist in seiner Quellsprache implementiert. Das hat zur Folge, dass der Kompilator sich selbst als Eingabe erhalten kann, seinen eigenen Quelltext also in eine Maschinensprache übersetzen kann. In diesem Fall wird also praktisch genau das durchgeführt, was in der Theorie der Berechenbarkeit noch als abstruses Gedankenexperiment erschien. Bei dem Beweis bezüglich des Halteproblems wird von einem Programm, das ein Terminierungsbeweiser darstellt, ausgegangen und dann zu einem Widerspruch geführt, was passiert, wenn dieses Programm sich selbst als Eingabe erhält.

Wenn auch die meisten in der herkömmlichen Softwareentwicklung verwendeten Kompilatoren von einer Hochsprache in eine maschinennahe Sprache übersetzen, gibt es doch viele Beispiele, die hiervon abweichen. Es gibt so Kompilatoren, die von einer Hochsprache in eine andere übersetzen. Gerne wird dieses verwendet, wenn es zwei Hochsprachen mit sehr ähnlichen Konzepten aber unterschiedlicher Syntax gibt.

Es gibt auch Kompilatoren, in denen die Quell- und die Zielsprache ein und dieselbe sind. Dieses wird angewendet, um bestimmte Programmierkonventionen einzu-

halten, zusätzliche Prüfungen einzubauen (z.B. NULL-Prüfungen als Vorbedingung eines Funktionsaufrufs).

Im Falle des Satzsystems LaTeX liegt ein Kompilator vor, der nicht einmal im herkömmlichen Sinne Programme übersetzt, sondern eine Dokumentbeschreibung in ein druckbares Dokumentenformat.

1.3 Semantik

Von eine Übersetzung von einer Sprache in eine andere Sprache wird erwartet, dass die Bedeutung gleich bleibt. Hierzu muss man streng genommen die Semantik der Quell- und der Zielsprache gut kennen. Diese muss hierzu sauber formuliert sein. Leider ist für die wenigsten Programmiersprachen die Semantik denotational beschrieben. Maschinensprachen haben meistens eine Zustandsübergangsbeschreibung. Jeder befehl führt von einem bestimmten Zustand in einen anderen. Hochsprachen haben oft nur eine informelle Beschreibung der Semantik. Im schlimmsten Fall ist die Semantik durch das erzeugte Maschinenprogramm eines bestimmten Kompilators festgelegt. Man spricht dann auch gerne vom *works as implemented*.

1.4 Interpretieren statt Übersetzen

Insbesondere von Skriptsprachen ist bekannt, dass gar kein eigener Maschinencode erzeugt wird, sondern der Quelltext direkt interpretiert wird. Damit fällt eine eigentliche Kompilierung weg. Trotzdem hat ein Interpreter für eine Skriptsprache auch die ersten Phasen des Kompilators. Eine lexikalische und syntaktische Analyse des Quelltextes muss vorgenommen werden. Es wird dann eine interne Datenstruktur erzeugt, auf der die Interpretation durchgeführt wird. Oft wird aber intern doch ein abstrakter Zwischencode erzeugt, für den erst ein Interpreter in Form einer virtuellen Maschine existiert.

Interpreter sind nicht allein auf Skriptsprachen beschränkt. Vielfach bieten auch Kompilatoren für statisch getypte Sprachen an, die Programme in einem Interpreter laufen zu lassen, wie z.B. der ghc für Haskell oder auch im Falle der Programmiersprache Scala.

1.5 Phasen eines Kompilators

Ein Kompilator wird in zwei große Teile eingeteilt:

- dem *frontend*, das sich mit der Analyse des Quellsprachprogramms beschäftigt. Das Quellsprachprogramm wird syntaktisch und semantisch untersucht und eine interne Baumrepräsentation des Programms erstellt.
- dem *backend*, das sich um die Erzeugung des Codes der Zielsprache kümmert.

Eine klare Trennung zwischen Frontend und Backend in der Architektur eines Kompilators ermöglicht Flexibilität in zwei Richtungen:

- zu einem Frontend können verschiedene Backends existieren: Man kann somit leichter einen Kompilators schreiben, der Code für verschiedene Zielmaschinen übersetzen kann.
- zu einem Backend lassen sich verschiedene Frontends schreiben, so daß ein Kompilators für unterschiedlichste Sprachen entstehen kann. So ist der GNU `gcc` nicht allein ein C-Kompilators, sondern in der Lage auch Java oder Fortran zu übersetzen, benutzt dabei aber immer ein und dasselbe Backend.

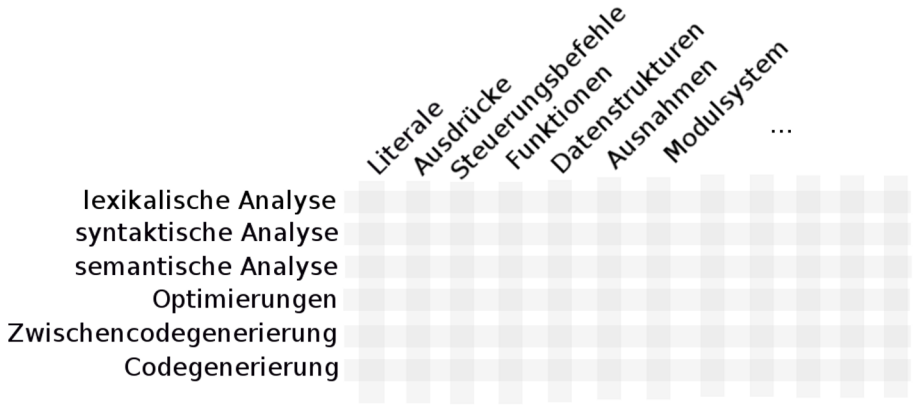
Grundvoraussetzung für eine saubere Trennung zwischen Frontend und Backend ist eine Zwischensprache. Das Frontend erzeugt aus dem Quellsprachenprogramm ein äquivalentes Zwischensprachenprogramm, für das das Backend schließen Code der Zielsprache erzeugen kann.

Im Frontend des Kompilators unterscheidet man verschiedene Schritte:

- lexikalische Analyse: die Zeichenkette des Programms wird in einzelne Wörter gesplittet. Man spricht von Token.
- syntaktische Analyse: anhand einer Grammatik wird untersucht, ob die Folge der Token als Satz der Sprache mit dieser Grammatik generiert werden kann.
- semantische Analyse: Eigenschaften des Programms über die Syntax hinaus werden untersucht. Hier kann es je nach kompilierter Programmiersprache eine vielfältige Reihe von Analysen geben. Ein paar Beispiele:
  - Konsistenzprüfungen: sind alle Variablen vor ihrer Benutzung deklariert worden.
  - Typcheck: ist die Benutzung der Variablen Typkonform.
  - Ausnahmen: werden alle Ausnahmen gefangen oder deklariert.
  - Sichtbarkeiten: wird nirgends unerlaubt auf private Eigenschaften zugegriffen.
  - Programmflußanalyse: gibt es Programmteile, die niemals erreicht werden können? Endet ein Funktionsaufruf mit Rückgabewert garantiert mit einem `return`-Befehl?...
- Transformationen, Optimierungen: das Programm kann in eine Kernsprache, die weniger Programmkonstrukte enthält, übersetzt werden. Unnötige Befehlssequenzen gelöscht, Werte konstanter Ausdrücke bereits durch den Kompilator berechnet werden ....

- Zwischencode erzeugen
- weitere Optimierungen
- Codegenerierung

Ein Kompilator besteht also aus mehreren sequentiell nacheinander folgenden Phasen. Die Quellsprache besteht aus einer bestimmten Anzahl einzelner Sprachelemente, wie arithmetische Ausdrücke, Steuerungsbefehlen wie Verzweigungen und Schleifen. So lässt sich ein Kompilator zum einen horizontal entlang der Phasen, die er durchläuft oder vertikal entlang der einzelnen Konstrukte der Quellsprache aufteilen:



In dieser Vorlesung werden wir den Kompilator vertikal aufteilen. Wir werden zunächst einen Kompilator für die Quellsprache schreiben, die nur eine einzige Konstante als Sprachelement kennt. Dieser Kompilator wird aber bereits alle Phasen bis zur Codegenerierung beinhalten. Kapitelweise wird der Kompilator dann um weitere Sprachkonstrukte der Quellsprache erweitert. Wir nennen unsere Sprache `WIP`, für *Wiesbadener Informatik Programme*.

1.6 Überblick über das Projekt

In den nachfolgenden Kapiteln werden wir schrittweise einen Kompilator entwickeln. Nicht nur das. Wir werden auch einen Interpreter für die Sprache schreiben. Wir werden eine abstrakte Maschine definieren, für die wir Code erzeugen. Die abstrakte Maschine werden wir wiederum implementieren, so dass der abstrakte Maschinencode interpretiert wird. Anschließend werden wir X86 Assembler Code generieren, um diesen mit einer in der Programmiersprache C entwickelten Laufzeitumgebung zusammen zu linken, so dass wir ein ausführbares Programm erhalten. Abbildung 1.1 gibt einen Gesamtüberblick über alle Projektteile.

Wir beginnen jeweils mit der `javacc`-Eingabedatei `WIPParser.jj`. In dieser sind die Token und die Grammatik der Sprache WIP definiert. Mit Hilfe des Programms `javacc` wird damit der Lexer und Parser generiert. Dieser ist in der Lage für einen

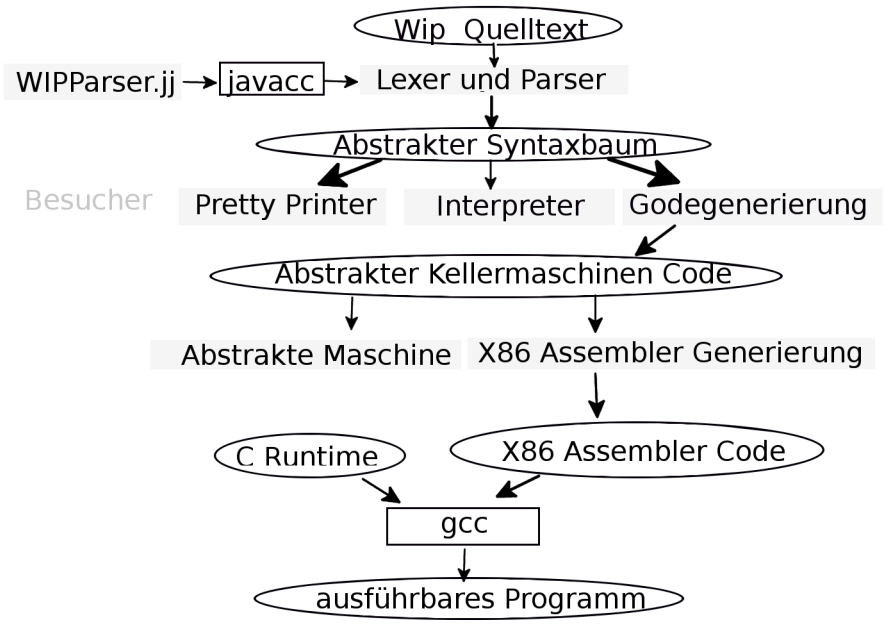


Abbildung 1.1: Überblick über die Gesamtarchitektur

WIP-Quelltext einen abstrakten Syntaxbaum (AST) zu erstellen. Auf diesem AST werden dann verschiedene Algorithmen angewendet. Dabei kommt das Besuchsmuster als Entwurfsmuster zur Anwendung.

Drei Besucher werden für den AST geschrieben:

- ein *pretty printer*, der das durch den AST dargestellte Programm wieder als Quelltext darstellt.
- ein Interpreter, der das im AST dargestellte Programm ausführt.
- eine Codegenerierung, die Code für eine abstrakte Kellermaschine erzeugt.

Der erzeugte Maschinencode für die abstrakte Maschine wird auf zwei Arten weiterverarbeitet:

- er wird interpretiert durch eine Implementierung der virtuellen Maschine.
- es wird für den abstrakten Maschinencode Assembler generiert.

Der generierte Assemblercode wird mit dem `gcc` schließlich zusammen mit einer in C geschriebenen Laufzeitumgebung zu einem ausführbaren Programm übersetzt. In der Laufzeitumgebung werden wir insbesondere die automatische dynamische Speicherverwaltung mit einer *garbage collection* umsetzen.

Kapitel 2

Version 1. Konstante 42

In unserer ersten Version des WIP-Kompilators wird unsere Quellsprache nur ein einziges Sprachkonstrukt haben: die konstante Zahl 42. Wir können also nur Programme schreiben, die genau ein Zahlenliteral haben. Trotzdem werden wir alle Phasen des Kompilators implementieren. Zunächst Lexer und Parser, für die syntaktische Analyse. Das Ergebnis wird ein abstrakter Syntaxbaum sein. Dieser wird dann weiterverarbeitet. Er wird wieder als Programm ausgegeben, er wird interpretiert. Dann werden wir einen anstrakten Maschinencode für diesen abstrakten Syntaxbaum schreiben. Dieser abstrakte Code zunächst durch einen Maschineninterpreter ausgeführt. Snchließend werden wir x86 Assembler für den abstrakten Maschinencode generieren.

2.1 Parser und Tokenizer

Die ersten beiden Schritte eines Kompilators sind die lexikalische und syntaktische Analyse. Herzu stellt man mittels regulärer Ausdrücke zunächst die einzelnen Lexeme der Sprache auf. Dieses sind die Schlüsselwörter, Zahlenliterale, Stringliterale, Bezeichner von Variablennamen, Symbole für Klammern, Interpunktionszeichen und Operatoren. Es wird auch mittels eines regulären Ausdrucks definiert, was die Zwischenräume zwischen den Lexemen sind. Der Teil des Kompilators, der die Lexeme anhand dieser Spezifikation erkennt wird als Lexer oder Tokenizer bezeichnet. Der Lexer erzeugt aus dem Eingabestrom einen Strom von Lexemen. Dieser dient als Eingabe für die zweite Phase, der syntaktischen Analyse.

In der Sprachspezifikation der Quellsprache wird eine kontextfreie Grammatik aufgestellt. Eine kontextfreie Grammatik besteht aus einer Menge von Terminalsymbolen, Nichtterminalsymbolen, von dem eines das ausgezeichnete Startsymbol ist und einer Menge von Produktionsregeln.

Eine Produktionsregel ist ein Paar aus einem Nichtterminalsymbol und einer endlichen möglicher Weise leeren Folge von Terminal- und Nichtterminalsymbolen. Alle Folgen von Terminlasymbolen, die man ausgehend vom Startsymbol durch Ersetzen der Nichtterminalsymbole entsprechend einer Produktionsregel erhalten kann, sind Sätze der Sprache.

Für eine Programmiersprache sind die Terminale dieser Grammatik die Lexeme der Sprache. Die Grammatik beschreibt, wie man Sätze der Sprache generieren kann. Die syntaktische Analyse des Kompilators geht nun umgekehrt vor. Für ein gegebenes Programm wird getestet, ob dieses mit der Grammatik der Sprache generiert werden kann. Das entsprechende Programm wird als Parser bezeichnet. Es zerteilt ein gegebenes Programm wieder in die Bestandteile, durch die es von der Grammatik erzeugt wurde. Es entsteht ein Parsbaum. An den Blättern eines Parsbaums stehen Terminalsymbole an den übrigen Knoten Nichtterminalsymbole. Die Kinder eines Knoten sind mit den Symbolen der rechten Seite einer Produktionsregel markiert.

2.1.1 Parsergeneratoren

Obwohl es gar nicht so kompliziert ist einen Lexer und einen Parser zu einer gegebenen Sprache zu programmieren, bedient man sich in der Praxis meist Werkzeugen, die diese beiden Teile eines Kompilators generieren. Man spricht dabei von Parsergeneratoren oder auch von compiler-compilern.

Der wohl am weitesten verbreitete Parsergenerator ist Yacc. Wie man seinem Namen, der für *yet another compiler compiler* steht, entnehmen kann, war dieses bei weiten nicht der erste Parsergenerator. Yacc erzeugt traditionell C-Code; es gibt aber mitlerweile auch Versionen, die Java-Code generieren. Ein weitgehendst funktionsgleiches Programm zu Yacc heißt *Bison*. Für die lexikalische Analyse steht im Zusammenspiel mit Yacc und Bison jeweils auch ein Generatorprogramm zur Verfügung, das einen Tokenizer generiert. Dieses Programm heißt `lex` bzw. `flex`. Weitere modernere und auf Java zugeschnittene Parsergeneratoren sind: `javacc`, `sablecc`, `antlr` und `gentle`. Im Prinzip kann man davon ausgehen, daß es für fast alle Sprachen einen Parsergenerator gibt, so z.B. auch *Happy* für Haskell.

Parsergeneratoren haben als Eingabe eine Beschreibung der Grammatik. Die Regeln der Grammatik sind mit Code markiert, der für einen erfolgreichen Parsvorgang auszuführen ist, um das Ergebnis, zumeist ein abstrakter Syntaxbaum, zu generieren. Viele Parsergeneratoren haben sogar eine Automatik um den Syntaxbaum zu erzeugen und generieren auch die entsprechenden Klassen hierzu.

Man kann aber auch, insbesondere in Sprachen, die höherer Ordnung sind, Bibliotheken bereitstellen, mit deren Hilfe sich Parser auf einfache Weise direkt in der Programmiersprache formulieren lassen und kein externes Werkzeug benötigt wird.

Für unseren Beispielkompilator nutzen wir das Programm `javacc` als Werkzeug zur Lexer- und Parsergenerierung. Eine `javacc`-Eingabedatei beginnt mit Informationen über die zu generierende Javaklasse, die den Parser darstellt. zunächst das Paket, der Klassenname und weitere Felder und Methoden, die in die Klasse eingefügt werden. Dieser Teil ist eingeschlossen in den Befehlen `PARSER_BEGIN` und `PARSER_END` gefolgt vom Klassennamen in Klammern.

Wir sehen für unseren ersten Parser eine Hauptmethode vor, die eine Instanz erzeugt, die als Eingabe `System.in` verarbeitet.

```
1  PARSER_BEGIN(WIPParser)
2  package v0.name.panitz.wip;
3
4  public class  WIPParser{
5    public static void main(String args[]) throws ParseException {
6      WIPParser parser = new WIPParser(System.in);
7      parser.startSymbol();
8    }
9  }
10 PARSER_END(WIPParser)
```

Listing 2.1: WIPParser.jj

In einem nächsten Abschnitt lässt sich der Weißzwischenraum der Sprache definieren. In unserem Fall definieren wir vier Zeichen als Weißzwischenraum:

```
11 SKIP :
12 {
13   " "
14 | "\t"
15 | "\n"
16 | "\r"
17 }
```

Listing 2.2: WIPParser.jj

In einem weiteren Abschnitt lassen sich die Lexeme, auch Token der Sprache, als regulären Ausdruck definieren. In unserer ersten Version gibt es nur ein einziges Token, die Konstante 42.

```
18 TOKEN :
19 {
20   <FORTY_TWO: "42">
21 }
```

Listing 2.3: WIPParser.jj

Und schließlich sind in der `javacc`-Syntax die Produktionsregeln der kontextfreien Grammatik zu formulieren. Unsere Grammatik besteht vorerst aus genau einer Regel für das Nichtterminal *startSymbol*. Die rechte Seite dieser Regel ist die Folge von zwei Terminalsymbolen. Dem Token für die Konstante 42 und dem eingebauten Token, für das Dateiende.

```
22 void startSymbol() :
23 {}
24 {
```

```
25    <FORTY_TWO> <EOF>
26  }
```

Listing 2.4: WIPParser.jj

Die regeln der Grammatik erhalten in `javacc` einen Rückgabetyp, den wir in diesem Fall auf `void` gesetzt haben. In dem ersten leeren geschweiften Klammernpaar, das wir noch leer gelassen haben. Hier können lokale Java-Variablen für die Grammatikregel deklariert werden. Die Nichtterminale werden in spitzen klammern notiert.

Damit ist unser erster Parser entwickelt. Die `.jj`-datei ist mit `javacc` zu verarbeiten. Dabei werden mehrere Klassen generiert:

```
panitz@ThinkPad-T430:~/fh/wip/student/src/v0/name/panitz/wip$ javacc WIPParser.jj
Java Compiler Compiler Version 6.1_2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file WIPParser.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
panitz@ThinkPad-T430:~/fh/wip/student/src/v0/name/panitz/wip$ ls
ParseException.java    TokenMgrError.java WIPParser.jj
SimpleCharStream.java  WIPParserConstants.java WIPParserTokenManager.java
Token.java        WIPParser.java
panitz@ThinkPad-T430:~/fh/wip/student/src/v0/name/panitz/wip$
```

Diese können wir übersetzen und das Programm dann starten. Es bricht mit einer Fehlermeldung ab, wenn auf `System.in` ein Programm gelesen wird, das nicht mit der Grammatik erzeugt werden kann.

```
panitz@ThinkPad-T430:~/fh/wip/student/src$ javac -d . v0/name/panitz/wip/*.java
panitz@ThinkPad-T430:~/fh/wip/student/src$ java v0.name.panitz.wip.WIPParser
42
panitz@ThinkPad-T430:~/fh/wip/student/src$ java v0.name.panitz.wip.WIPParser
43
Exception in thread "main" v0.name.panitz.wip.TokenMgrError: Lexical error at line 1, column 2.
Encountered: "3" (51), after : "4"
at v0.name.panitz.wip.WIPParserTokenManager.getNextToken(WIPParserTokenManager.java:142)
at v0.name.panitz.wip.WIPParser.jj_consume_token(WIPParser.java:128)
at v0.name.panitz.wip.WIPParser.input(WIPParser.java:12)
at v0.name.panitz.wip.WIPParser.main(WIPParser.java:8)
panitz@ThinkPad-T430:~/fh/wip/student/src$
```

2.2 Testprogramme

Das einziege gültige Testprogramm für unsere Sprache ist vorerst das Programm, das aus der konstanten 42 besteht.

```
1  42
```

Listing 2.5: fortytwo.wip

2.3 Syntaxbaum

Wir haben bisher zwar einen Parser, aber dieser generiert kein Ergebnis, mit dem wir weiterarbeiten können. Er akzeptiert entweder ein Eingabeprogramm oder bricht den Parsvorgang mit einer Fehlermeldung ab. Für den weiteren Verlauf unseres Kompilators benötigen wir einen abstrakten Syntaxbaum, auf dem dann weitere Phasen des kompilators durchgeführt werden können. In diesem Abschnitt werden wir nun während des Parsvorgangs einen Syntaxbaum erzeugen. Hierzu benötigen wir Klassen, die den Baum darstellen können. Wir werden dann verschiedene Algorithmen für den Syntaxbaum implementieren. In objektorientierten Sprachen ist ein gängiges Entwurfsmuster für Bäume mit unterschiedlichen Knotenarten und Algorithmen, die auf diesen Bäumen operieren, das Besuchsmuster.

2.3.1 Besuchspattern

Für das Besuchsmuster definiert man sich zunächst eine allgemeine Schnittstelle für die Baumknoten. Es wird erwartet, dass jeder Baumknoten einen Besucher willkommen heißen kann.

```
1 package v1.name.panitz.wip.tree;
2 public interface  Tree{
3   public <R> R welcome(Visitor<R> visitor) throws Exception;
4 }
```

Listing 2.6: Tree.java

Da unsere Sprache bisher nur Zahlenkonstanten kennt, brauchen wir vorerst nur genau eine Baumknotenklassen, nämlich die für Integer-Literale.

```
1 package v1.name.panitz.wip.tree;
2 public class IntLit implements Tree{
3   public int i;
4   public IntLit(int i){
5     this.i = i;
6   }
7   public String toString(){return i+"";}
```

Listing 2.7: IntLit.java

Jeder Baumknotenklasse implementiert die Methode `welcome` auf identische Art und Weise. Es wird auf dem übergebenen Besuchsobjekt die Methode `visit` mit dem `this`-zeiger aufgerufen.

```
8   public <R> R welcome(Visitor<R> visitor) throws Exception{
9     return visitor.visit(this);
10  }
11 }
```

Listing 2.8: IntLit.java

Bleibt zu klären, was ein Besucher eigentlich ist. Es ist eine Schnittstelle, in der für jede Baumknotenklasse eine Methode zum Besuchen überladen ist.

```
1 package v1.name.panitz.wip.tree;
2 public interface Visitor<R>  {
3   R visit(IntLit il) throws Exception;
4 }
```

Listing 2.9: Visitor.java

Wie man sieht, ist die Besucherschnittstelle generisch über einen Typ `R` gehalten. Dieser steht für das Ergebnis, das der Besuch erzeugt. Das wäre nicht unbedingt notwendig, kann manchmal aber den Code vereinfachen.

Ein Pretty Printer

Als ersten Besucher schreiben wir eine Klasse, die den Baum wieder in textueller Form darstellt. So ein Programm wird als *pretty printer* bezeichnet. Es geht darum, das Programm wieder möglichst sauber formatiert auszugeben. Im besten Fall handelt es sich, wenn das Ergebnis des *pretty printers* wieder geparst und mit dem *pretty printer* ausgegeben wird um eine Identität.

Wir implementieren die Klasse `PP` als einen Besucher, der eine Zahl als Ergebnis liefert. Dieses Ergebnis ist allerdings ziemlich uninteressant, wie wir sehen werden. Der Besucher bekommt im Konstruktor ein `Writer`-Objekt, auf dem der Text geschrieben werden soll. Zusätzlich wird ein internes Feld für die Einrücktiefe initalisiert.

```
1 package v1.name.panitz.wip.tree;
2 import java.io.*;
3
4 public class PP implements Visitor<Integer>{
5   int indent = 0;
6   Writer w;
7   public PP(Writer w){
8     this.w = w;
9   }
```

Listing 2.10: PP.java

Nun ist für jede Baumknotenart eine Methode `visit` zu implementieren. Da wir bisher nur eine Baumknotenart kennen, ist dieses nur eine Methode. In dieser schreiben wir den Wert der ganzahligen Konstante in das `Writer`-Objekt.

```
10   public Integer visit(IntLit il) throws Exception{
11     w.write(il.i+"");
12     return indent;
13   }
14 }
```

Listing 2.11: PP.java

Interpreter auf abstrakten Syntaxbaum

Als nächstes schreiben wir einen weiteren Besucher für den abstraktenb Syntaxbaum. Diesmal einer, der das Programm interpretiert und das Ergebnis der Interpretation zurück gibt. Dieser Besucher ist noch einfacher zu imlementieren. Er gibt für den Baumknoten der ganzzahligen Konstante genau deren Wert als Ergebnis zurück.

```
1 package v1.name.panitz.wip.tree;
2
3 public class Interpreter implements Visitor<Integer>{
4   public Integer visit(IntLit il) throws Exception{
5     return il.i;
6   }
7 }
```

Listing 2.12: Interpreter.java

2.3.2 Syntaxbaum Erzeugen

In den letzten Absätzen haben wir Klassen für den abstrakten Syntaxbaum entwickelt und mit Hilfe des Besuchsmusters zwei Algorithmen auf diesen Baumklassen umgesetzt. Zuvor haben wir einen Parser für unsere Programmiersprache entwickelt. Was wir aber brauchen ist einen Parser, der den abstrakten Syntaxbaum konstruiert.

So widmen wir uns jetzt abermals der `javacc`-Eingabedatei. Der Unterschied ist nun, dass die Regel für das Startsymbol eine Rückgabe vom Typ `Tree` hat:

```
27 PARSER_BEGIN(WIPParser)
28 package v1.name.panitz.wip;
29 import  v1.name.panitz.wip.tree.*;
30
```

```
31 public class  WIPParser{
32   public static void main(String args[]) throws ParseException {
33     WIPParser parser = new WIPParser(System.in);
34     Tree il = parser.startSymbol();
35     System.out.println(il);
36   }
37 }
38
39 PARSER_END(WIPParser)
```

Listing 2.13: WIPParser.jj

Die Abschnitte für den Weißzwischenraum und die Tokendefinitionen bleiben identisch.

In der Beschreibung der Regel für das Startsymbol wird nun Javacode jeweils in geschweiften Klammern eingeschlossen, zugefügt, der dafür sorgt, das ein Baumknoten erzeugt wird. Hierzu werden in der ersten Sektion zwei lokale Variablen angelegt. Eine vom Javacc spezifischen Typ `Token` und eine vom Ergebnistyp `IntLit`. Die Token lassen sich einer Variable vom Typ `Token` zuweisen. Der Typ `Token` hat ein Feld vom Typ String, in dem die textuelle Ausprägung des Token dargestellt ist. In unserem Fall ist das immer der String `"42"`, aber das wird sich ja bald ändern.

In geschweiften Klammern lässt sich beliebiger Javacode in die Grammatikregel schreiben, der durchgeführt wird, wenn nach dieser Regel geparst wird.

```
40 Tree startSymbol() :
41 {Token tok;
42  IntLit result = null;
43 }
44 {
45   tok=<FORTY_TWO>{result =  new IntLit(new Integer(tok.image));}
46   <EOF> {return result;}
47 }
```

Listing 2.14: WIPParser.jj

2.4 Abstrakte Kellermaschine

Bisher haben wir nur einen Interpreter geschrieben. Wir können ein Programm parsen, den abstrakten Syntaxbaum erzeugen, und diesen Baum mit unserem Interpreter-Besucher auswerten. Von einem Kompilator wird erwartet, dass er Code für eine Zielsprache erzeugt. Hierzu benötigen wir erst einmal eine Zielsprache. Die Zielsprache ist zunächst eine abstrakte Kellermaschine. Nahezu alle Maschinen basieren auf einem Stack, auf dem die Parameter für die Berechnungen abgelegt werden.

2.4.1 Befehlssatz

Bevor wir Code für unsere abstrakte Kellermaschine generieren, definieren wir zunächst die Maschine und schreiben einen Interpreter für diese. Damit haben wir dann eine virtuelle Maschine.

Als erstes definieren wir einen Befehlssatz. Ein Maschinenbefehl bestehe dabei aus einem Befehlsnamen eventuell mit einem `int`-Parameter. Wir definieren eine einheitliche Klasse für die Maschinenbefehle. Der Befehlsname wird dabei von einem Aufzählungstyp dargestellt. Vorerst kommen wir mit einem einzigen Befehl aus. Dem Befehl `PushInt`, der eine Zahl auf den Keller ablegt.

```
1  package v1.name.panitz.wip.stackmachine;
2
3  public class  Instruction{
4    static public enum Instr {PushInt};
5
6    public Instr instr;
7    public int a1;
8
9    public Instruction(Instr instr,int a1){
10     this.a1 = a1;
11     this.instr = instr;
12   }
13 }
```

Listing 2.15: Instruction.java

2.4.2 Code Generierung

Wir können jetzt einen weiteren Besucher schreiben, der nun Code für unsere virtuelle Maschine generiert. In diesem Fall ist dies besonders einfach, denn wir haben nur einen Baumknoten und einen Befehl, die eins zu eins dieselbe Bedeutung haben. So erzeugt der Besucher genau die einelementige Liste, die die den Befehl enthält, die geparste Zahl auf den Keller zu legen.

```
1  package v1.name.panitz.wip.tree;
2  import v1.name.panitz.wip.stackmachine.*;
3
4  import java.util.List;
5  import java.util.LinkedList;
6
7  public class GenCode implements Visitor<List<Instruction>>{
8    List<Instruction> result = new LinkedList<>();
9
10   public List<Instruction> visit(IntLit il){
11     result.add(new Instruction(Instruction.Instr.PushInt,il.i));
```

```
12      return result;
13    }
14  }
```

Listing 2.16: GenCode.java

Damit haben wir einen ersten kompletten Kompilator geschrieben. Allerdings existiert die abstrakte Maschine noch nicht.

2.4.3 Maschineninterpreter

Um den Code, den unser Kompilator generiert, auch zur Ausführung zu bringen, bedarf es einer Implementierung der abstrakten Maschine, die wir in diesem Abschnitt definieren.

Die Maschine benötigt folgende Komponenten:

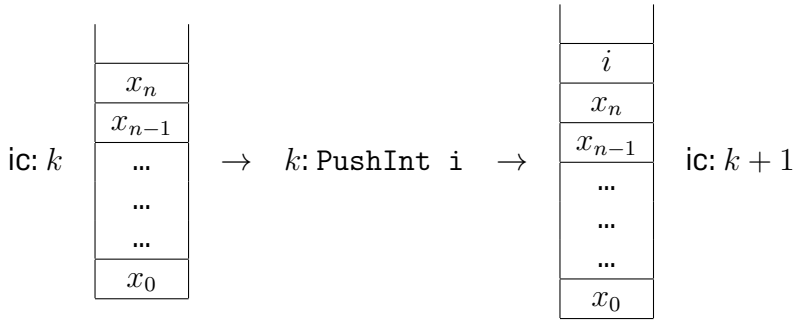
- `code`: den auszuführenden Maschinen code
- `ic`: den Instruktionszähler, der auf den aktuellen Befehl zeigt.
- `stack`: der eigentliche Keller
- `stc`: der Stackzähler, der auf das nächste freie Stackelement zeigt
- `stb`: die aktuelle Kellerbasis für einen Funktionsaufrufe. Wird vorerst noch nicht benötigt.

Und so hat unsere Klasse für die virtuelle Kellermaschine die entsprechenden Felder.

```
1  package v1.name.panitz.wip.stackmachine;
2
3  public class StackMachine{
4    public Instruction[] code;
5    int ic = 0;
6    int[] stack = new int[200];
7    int stc = 0;
8    int stb = 0;
```

Listing 2.17: StackMachine.java

Diese fünf Felder stellen den Zustand der Maschine dar. Jeder Zustand führt in einem Schritt, abhängig von dem befehl, auf den der Programmzähler zeigt, zu einen Folgezustand. Diese Zustandsübergänge sind die Semantik der Maschinenbefehle. Bisher haben wir erst einen Befehl vorgesehen. Das folgende Diagramm zeigt, wie dieser Befehl den Zustand verändert:



Dieser Zustandsübegang ist für die virtuelle Maschine zu implementieren. Wir sehen eine Methode `step` vor, die einen Zustandsübergang vornimmt. Hierbei wird ein `switch` auf den aktuellen Befehl durchgeführt:

```
9   void step(){
10    switch (code[ic].instr) {
11    case PushInt: stack[stc++] = code[ic++].a1;break;
12    }
13  }
```

Listing 2.18: StackMachine.java

In unserem Fall ist der Parameter des Befehls auf den Stack zu legen und Programmzähler und Stackzähler um eins zu erhöhen.

Wenn die Maschine läuft, werden solange Zustandsübergänge durchgeführt, bis der Programmzähler auf eine illegalen Wert zeigt.

```
14  public void run(){
15    while (ic<code.length){
16      step();
17    }
18    System.out.println(stack[0]);
19  }
```

Listing 2.19: StackMachine.java

Wir sehen hier einmal einen kleinen Testaufruf der Maschine vor.

```
20  public static void main(String[] args){
21    StackMachine interp = new StackMachine();
22    interp.code = new Instruction[1];
23    interp.code[0] = new Instruction(Instruction.Instr.PushInt,42);
24    interp.run();
25  }
26 }
```

Listing 2.20: StackMachine.java

2.5 GAS X86 Assembler Generierung

Wir sind immer noch nicht mit dem ersten Kapitel fertig. Bisher haben wir nur eine virtuelle Maschine, die durch eine Javaklasse implementiert wird. In einem letzten Schritt soll Assembler Code generiert werden. Wir nutzen hierfür den GNU Assembler, kurz GAS. Es ist der Assembler des GNU-Projekts. Er ist das Standard-Backend des `gcc`.

Der Vorteil ist, dass auch der GNU Assembler bereits eine Kellermaschine realisiert. Wir können unseren abstrakten Maschinencode für viele Instruktionen eins zu eins im GNU Assembler umsetzen. Insbesondere gibt auch die in unserem abstrakten Maschinenmodell vorgesehenen drei Zähler: Stackzähler, Kellerbasis und Befehlszähler.

Anders als in unserer abstrakten Maschinen, gibt es im GNU Assembler zusätzliche Register, auf denen die eigentlichen Befehle arbeiten. Es wird nicht auf dem Keller selbst gerechnet, sondern in den Registern. Vorerst in dieser ersten Version benötigen wir noch keine Rechnungen.

Die Befehlsnamen und die Registernamen unterscheiden sich im GNU Assembler zwischen Assembler für 32-Bit Maschinen und 64-Bit Maschinen. Wir werden dieses in den ersten Versionen des Kompilators berücksichtigen und entsprechend eines Flags Assembler-Code für beide dieser Varianten erzeugen können.

Möchte man Beispiele für GNU Assembler-Code sehen, so kann man beim `gcc` mit der Option `-S` sich für ein C-Programm den entsprechnden generierten Assemblercode ausgeben lassen. Dieses ist manchmal ein guter Weg, um Erfahrungen mit GNU Assembler-Code zu machen.

Betrachten wir eine kleine C-Datei:

```
1 int go(){
2   return 42;
3 }
```

Listing 2.21: go42.c

Für diese wird folgender Assemblercode generiert.[1]

```
1  .file   "go42.c"
2         .text
3         .globl  go
4         .type   go, @function
5 go:
6         pushl   %ebp
7         movl    %esp, %ebp
```

[1] Zusötzlich bekam der gcc hierbei die Option `-fno-asynchronous-unwind-tables` ohne die ansonsten noch eine ganze Reihe von Direktiven in den Code gewesen wären.

```
8           movl    $42, %eax
9           popl    %ebp
10          ret
11          .size   go, .-go
12          .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
13          .section        .note.GNU-stack,"",@progbits
```

Listing 2.22: go42.s

Auf ähnliche Weise, wollen wir in diesem Abschnitt Assemblercode generieren.

Wir schreiben im Folgendem die Klasse `WriteX86`, die für eine Befehlsfolge unserer abstrakten Maschine entsprechenden GNU Assembler Code in eine Datei schreibt.

Im Konstruktor wird der Code der abstrakten Maschine, für den Assemblercode generiert werden soll, übergeben:

```
1  package v1.name.panitz.wip.stackmachine;
2  import  v1.name.panitz.wip.stackmachine.Instruction.Instr;
3  import java.io.*;
4
5  public class WriteX86{
6    public Instruction[] code;
7    public WriteX86(Instruction[] code){
8      this.code = code;
9    }
```

Listing 2.23: WriteX86.java

Ein bool'sches Flag gibt an, ob es Code für eine 32 Bit oder eine 64 Bit Maschine generiert werden soll:

```
10   public boolean _32 = true;
```

Listing 2.24: WriteX86.java

Wir schreiben kleine Hilfsmethoden, die den enstprechenden Assemblercode für Assemblerbefehle und Registernamen erzeugen. Die Registenamen für den Kellerzeiger *(stack pointer)* und die Kellerbasis *(base pointer)* sind `esp` und `ebp` für 32 Bit bzw, `rsp` und `rbp` für 64 Bit Code:

```
11   String sp(){return _32?"%esp":"%rsp";}
12   String bp(){return _32?"%ebp":"%rbp";}
```

Listing 2.25: WriteX86.java

Die zwei wichtigsten Befehle, die man bei einer Kellermaschine erwartet sind natürlich `push`, um einen Wert auf den Keller zu legen und `pop`, um einen Wert vom Keller zu nehmen und in einem anderen Register zu speichern. Der GNU Assembler bietet diese natürlich an, allerdings in unterschiedlichen Varianten für 32- und 64-Bit Systemen.

```
13   String push(){return _32?"pushl":"pushq";}
14   String pop(){return _32?"popl":"popq";}
```

Listing 2.26: WriteX86.java

Ein weiterer Befehl erlaubt es, Werten von einem Register in ein anderes zu verschieben. Dieser Befehl wird als `move` bezeichnet und hat auch unterschiedliche Namen in 32- und 64-Bit Systemen.

```
15   String mov(){return _32?"movl":"movq";}
```

Listing 2.27: WriteX86.java

Es gibt schließlich weitere Register, auf denen gearbeitet werden kann. Vorerst nutzen wir nur das Register `ax`:

```
16   String ax(){return _32?"%eax":"%rax";}
```

Listing 2.28: WriteX86.java

Die eigentliche Methode dieser Klasse schreibt den Assemblercode für die Folge von Befehlen der abstrakten Maschine in ein `Writer`-Objekt.

```
17   public void writeAssembler(Writer out)throws Exception{
```

Listing 2.29: WriteX86.java

Unsere Programme der ersten Version der Sprache WIP erzeugen einen Wert als Ergebnis. Wir definieren im Assembler eine Funktion `go`, nach deren Ausführung der Wert unseres Programmes im Register `ax` gespeichert ist. Deshalb beginnen wir bei der Assemblergenerierung mit dem Assemblercode für den Start einer parameterlosen Funktion:

```
18     out.write(".globl go\n");
19     out.write("go:\n");
20     out.write("    "+push()+"    "+bp()+"\n");
21     out.write("    "+mov()+"     "+sp()+", "+bp()+"\n");
```

Listing 2.30: WriteX86.java

Diesen Code haben wir aus dem oben generierten Assembler-Coe eins zu eins abgeschaut.

Als nächstes gehen wir alle Befehle der abstrakten Maschine durch und generieren für dies Assembler.

```
22      for (Instruction in:code){
23        writeAssembler(out,in);
24      }
```

Listing 2.31: WriteX86.java

Wir beschließen mit dem Standardcode für das Ende einer Funktion. Hierzu schreiben wir zunächst das Ergebnis der Funktion in Register `ax`.

```
25      out.write("    "+pop()+"     "+ax()+"\n");
26      out.write("    "+pop()+"     "+bp()+"\n");
27      out.write("    ret\n");
28      out.close();
29    }
```

Listing 2.32: WriteX86.java

Bleibt zu spezifizieren, welche Code für unsere bisher einzige abstrakte Maschinenanweisung im Assembler zu generieren ist. Eine ganze Zahl soll auf den Keller gelegt werden. Hierzu kann man sich des Assembler Befehls `push` bedienen. Zahlenkonstanten werden im Assembler mit einem Dollarzeichen eingeleitet.

```
30    public void writeAssembler(Writer out,Instruction ins)throws
       Exception{
31      switch (ins.instr) {
32      case PushInt: out.write("    "+push()+"    $"+ins.a1+"\n");break;
33      }
34    }
35  }
```

Listing 2.33: WriteX86.java

2.6 Runtime

Vorerst ist die Laufzeit, mit der wir unseren erzeugten Assemblercode zusammen linken denkbar einfach. Es ist eine C-Datei mit einer Hauptfunktion, in der die Funktion `go` aufgerufen wird und deren Ergebnis auf der Kommandozeile ausgegeben wird.

```
1  #include <stdio.h>
2
3  int main(){
4    int res = go();
5    printf("%d\n",res);
6    return 0;
7  }
```

Listing 2.34: runt.c

Um alle Komponenten unseres Kompilators zu Starten und auszuprobieren, schreiben wir uns noch eine Javaeinstiegsklasse. Wir lesen den Quelltext entweder aus einer Datei oder von der Standardeingabe, starten alle Besucher für den AST, interpretieren den abstrakten Maschinencode und generieren Assembler sowohl für 32 Bit als auch 64 Bit Architekturen.

```
1  package v1.name.panitz.wip;
2  import v1.name.panitz.wip.tree.*;
3  import v1.name.panitz.wip.stackmachine.*;
4  import java.util.*;
5  import java.io.*;
6
7  public class Main{
8    public static void main(String[] args) throws Exception{
9      WIPParser parser = null;
10     String name;
11     if (args.length == 0){
12       parser = new WIPParser(System.in);
13       name = "stdinput";
14     }else {
15       parser = new WIPParser(new FileReader(args[0]));
16       name = args[0].substring(0,args[0].indexOf('.'));
17     }
18     IntLit il = parser.startSymbol();
19
20     StringWriter out  = new StringWriter();
21     PP pp = new PP(out);
22     il.welcome(pp);
23     System.out.println(out);
24
25     System.out.println(il.welcome(new Interpreter()));
26
27     GenCode genCode = new GenCode();
28     List<Instruction> code = il.welcome(genCode);
29
30     StackMachine st = new StackMachine();
31     st.code = code.toArray(new Instruction[0]);
32     st.run();
33
34     WriteX86 asm = new WriteX86(st.code);
35     FileWriter asmf = new FileWriter(name+".s");
36     asm.writeAssembler(asmf);
37
38     WriteX86 asm64 = new WriteX86(st.code);
39     asm64._32 = false;
40     FileWriter asmf64 = new FileWriter(name+"64.s");
41     asm64.writeAssembler(asmf64);
42   }
```

```
43 ]
```

Listing 2.35: Main.java

Somit haben wir alle Komponenten unseres Kompilators umgesetzt. Nur die Quellsprache ist noch denkbar einfach. Diese werden wir jetzt schrittweise erweitern, bis sie mächtig genug ist, um in ihr ihren eigenen Kompilator zu implementieren.

Kapitel 3

Version 2: Multiplikations-Operatorausdrücke

In diesem Kapitel wird die Sprache WIP erweitert um erste Berechnung. Es soll möglich sein, Multiplikationen von Zahlenkonstanten durchzuführen.

3.1 Lexer und Parser

Zunächst einmal kennt unsere Sprache zwei neue Lexeme. Zum einen das Multiplikationssymbol, zum anderen beliebige ganzzahlige Konstanten. In der ersten Version war ja nur die Konstante `42` erlaubt. Um auszudrücken, dass ein Token aus einer endlichen Folge von mindestens einen bestimmten Zeichen besteht, benutzt auch die `javacc`-Syntax das Symbol `+`. Mit der Notation `"0"-"9"` lässt sich ausdrücken, dass ein Zeichen von `0` bis `9` erwartet wird. Token die mit einem Doppelkreuz Symbol beginnen, sind in `javacc` keine wirklichen eigenen Token, sondern nur eine Art Makro, um die Notation zu vereinfachen.

Hier also die beiden Token der zweiten Version von WIP.

```
48 TOKEN :
49 {
50    <#DIGIT: [ "0"-"9"] >
51   |<INTEGER_LITERAL: (<DIGIT>)+ >
52   |<MULT: "*">
53 }
```

Listing 3.1: WIPParser.jj

Soweit die Token. Nun haben wir erstmals eine kleine kontextfreie Grammatik. Zur Notation von kontextfreien Grammatiken sind die Backus-Naur-Form und verschiedene Erweiterungen und Modifikationen dieser etabliert. Auch die gängigen Parsergeneratoren bedienen sich dieser. Auch wir benutzen eine typographische Variante. Linke und rechte Seite einer Produktionsregel werden durch einen Pfeil getrennt. Ein Sternsymbol steht für die 0 bis n-fache Wiederholung, ein Plussymbol für die 1

bis n-fache Wiederholung. Ausdrücke in eckigen Klammern sind optional. Ausdrücke können mit Klammern gruppiert werden. Vertikale Striche trennen Alternativen. Nichtterminalsymbole sind kursiv gesetzt.

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{multiplicativeExpression} \\
\textit{multiplicativeExpression} &\rightarrow \textit{unaryExpression}\ (\texttt{MULT}\ \ \textit{unaryExpression})^* \\
\textit{unaryExpression} &\rightarrow \texttt{IntLit}
\end{aligned}
$$

Es gibt bisher drei Regeln in der Grammatik, die eins zu eins in den javacc-Quelltext umzusetzen sind. Die Einstiegsregel für das Nichtterminal *expression* geht direkt in die Regel für *multiplicativeExpression* über.

```
54 Tree expression():
55 {Tree result;}
56 {
57   result=multiplicativeExpression()  <EOF>
58   {return result;}
59 }
```

Listing 3.2: WIPParser.jj

Ein *multiplicativeExpression* ist ein *unaryExpression* gefolgt von einer optionalen Folge von Multioplikationen mit einem weiteren *unaryExpression*.

```
60 Tree multiplicativeExpression():
61 { Tree result;
62   Tree rightOperand;
63 }
64 {
65   result=unaryExpression()
66   ( <MULT> rightOperand=unaryExpression()
67       {result=new MultExpr(result,rightOperand);} )*
68   {return result;}
69 }
```

Listing 3.3: WIPParser.jj

Als *unaryExpression* sind vorerst nur einfache Ganzzahlliterale vorgesehen.

```
70 IntLit unaryExpression() :
71 {Token tok;
72  IntLit result = null;
73 }
74 {
75   tok=<INTEGER_LITERAL>{result =  new IntLit(new Integer(tok.image));}
```

```
76     [return result;]
77 ]
```

Listing 3.4: WIPParser.jj

3.2 Abstrakter Syntaxbaum

Da unsere Sprache nun ein neues Konstrukt kennt, ist für dieses ein Baumknoten zu definieren. Als zweite Klasse, die die Schnittstelle `Tree` implementiert, definieren wir eine Klasse, die die Multiplikation zweier Ausdrücke darstellt.

```
1  package v2.name.panitz.wip.tree;
2  public class MultExpr implements Tree{
3    public Tree left;
4    public Tree right;
5
6    public MultExpr(Tree left,Tree right){
7      this.left = left;
8      this.right = right;
9    }
10   public String toString(){return "MultExpr("+left+", "+right+")";}
11
12   public <R> R welcome(Visitor<R> visitor) throws Exception{
13     return visitor.visit(this);
14   }
15 }
```

Listing 3.5: MultExpr.java

Die Besucherschnittstelle verlangt nun, dass auch jeder Besucher spezifizieren muss, was er beim Besuch des neuen Knotentyps macht.

```
5  package v2.name.panitz.wip.tree;
6  public interface Visitor<R>  {
7    R visit(IntLit il) throws Exception;
8    R visit(MultExpr ex) throws Exception;
9  }
```

Listing 3.6: Visitor.java

Wir müssen also jede Besucherklasse um eine Methode `visit` für `MultExpr`-Knoten erweitern.

3.2.1 Pretty Printer

Im Besucher `PP` bedeutet es, dass bei einem Multiplikations-Ausdruck erst der linke Operand besucht werden muss, dann das Multiplikations-Symbol ausgegeben wird und schließlich der rechte Operand besucht werden muss.

```
15   public Integer visit(MultExpr ex) throws Exception{
16     ex.left.welcome(this);
17     w.write(" * ");
18     ex.right.welcome(this);
19     return indent;
20   }
21 }
```

Listing 3.7: PP.java

Man beachte, dass man zum Besuch der Kinderteilbäume im Besuchsmuster nicht die Funktion `visit` für diese aufruft, sondern quasi die Kinder bittet, den Besucher willkommen zu heißen.

3.2.2 Interpreter

Für den Interpreter-Besucher liefern die Besuche beim linken und beim rechten Operanden jeweils eine Zahl als Ergebnis. Das Ergebnis des Multiplikationsknoten ist natürlich das Produkt dieser beiden Operanden:

```
8    public Integer visit(MultExpr ex) throws Exception{
9      int op1 = ex.left.welcome(this);
10     int op2 = ex.right.welcome(this);
11     return op1 * op2;
12   }
13 }
```

Listing 3.8: Interpreter.java

3.3 Abstrakte StackMaschine

3.3.1 Befehlssatz

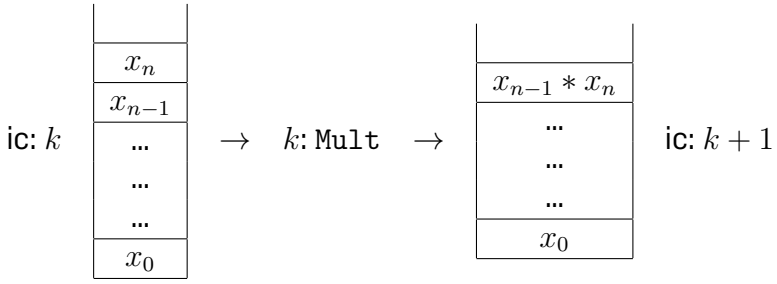
Die abstrakte Kellermaschine benötigt einen neuen Befehl, um Multiplikationen durchzuführen. Dieser Befehl wird keinen Parameter haben, sondern sich auf die oberen beiden Kellerelemente beziehen. Hierzu wird zunächst ein zweites Aufzählungsobjekt für die Instruktionen definiert:

```
14 package v2.name.panitz.wip.stackmachine;
15
16 public class  Instruction{
17   static public enum Instr {PushInt, Mult};
```

Listing 3.9: Instruction.java

3.3.2 Maschineninterpreter

Es ist zu spezifizieren, was für einen Zustandsübergang der neue Befehl auf der Kellermaschine bewirken soll. Hierzu dient wieder die schematische Darstellung.



Es werden also die oberen beiden Elemente vom Keller genommen, miteinander multipliziert und das Ergebnis auf den Keller gelegt. Der Programmzähler wird dabei um eins erhöht. Dieses lässt sich relativ einfach in dem Maschineninterpreter umsetzen.

```
27   void step(){
28     switch (code[ic].instr) {
29     case PushInt: stack[stc++] = code[ic].a1;ic++;break;
30     case Mult:
31       stack[stc-2] = stack[stc-2] * stack[stc-1];
32       stc--;
33       ic++;
34       break;
35     }
36   }
```

Listing 3.10: StackMachine.java

3.3.3 Code Generierung

Jetzt gilt es den dritten Besucher, den Besucher zur Codegenerierung um eine überladene Version der Methode `visit` für den Multiplikationsknoten zu erweitern.

Es ist zunächst Code zu generieren, der dafür sorgt, dass der linke Operand auf dem Keller geschrieben wird, dann Code zu generieren, der dafür sorgt, dass der rechte Operand auf dem Keller gelegt wird und schließlich ist der Maschinenbefehl `Mult` zu generieren.

```
15   public List<Instruction> visit(MultExpr ex) throws Exception[
16     ex.left.welcome(this);
17     ex.right.welcome(this);
18     result.add(new Instruction(Instruction.Instr.Mult));
19     return result;
20   ]
21 ]
```

Listing 3.11: GenCode.java

Es werden also erst die Operanden besucht und anschließend der `Mult`-Befehl eingefügt.

3.4 Assembler Generierung

Der X86 Assembler unterscheidet sich von unserer abstrakten Maschine in der Weise, dass Berechnungen nicht direkt auf den Stackelementen durchgeführt werden, sondern immer im Zusammenhang mit Registern.

```
36   String mul()[return _32?"mull":"mulq";]
```

Listing 3.12: WriteX86.java

Dieser Befehl hat aber anders als der Befehl `Mult` einen Parameter. Es wird der Wert aus dem Parameter mit dem Inhalt des `ax`-Registers multipliziert und das Ergebnis in das `ax`-Register geschrieben. Es wird insbesondere nicht auf dem Keller gerechnet. Um also die Semantik unserer abstrakten Maschine im Assembler umzusetzen, müssen die Operanden vom Keller genommen und in Register geschrieben werden. Dann muss die Multiplikation auf den Register ausgeführt werden und das Ergebnis aus dem `ax`-Register gelesen und auf den Keller gelegt werden

Wir nutzen als zweites Register das `bx`-Register. Die Operanden werden mit `pop` vom Keller in die beiden Register kopiert. Der Multiplikationsbefehl wird mit Register `bx` aufgerufen. Der zweite Operand wird immer aus Register `ax` genommen. Schließlich wird das Ergebnis der Multiplikation aus dem Register `ax` auf den Keller gelegt. Wir schreiben insgesamt vier Befehle:

```
37
38   public void writeAssembler(Writer out,Instruction ins)throws
       Exception[
```

```
39      switch (ins.instr) {
40      case PushInt: out.write("    "+push()+"    $"+ins.a1+"\n");break;
41      case Mult:
42        out.write("    "+pop()+"    "+bx()+"\n");
43        out.write("    "+pop()+"    "+ax()+"\n");
44        out.write("    "+mul()+"    "+bx()+"\n");
45        out.write("    "+push()+"    "+ax()+"\n");
46        break;
47      }
48    }
49  }
```

Listing 3.13: WriteX86.java

Nun können erste Rechnungen in WIP durchgeführt werden.

Kapitel 4

Version 3: Operatorausdrücke

In der dritten Version der Sprache WIP werden wir viel, wenn auch nicht viel Neues kennenlernen. Ziel ist es alle gängigen Operatorausdrücke, wie sie die meisten Programmiersprachen kennen, umzusetzen. Dabei soll vor allen Dingen die gängige Operatorpräzedenz gelten. Also die Regel, dass Multiplikationsoperatoren stärker binden als Additionsoperatoren. Wir wollen nicht nur arithmetische Operatoren sondern auch Vergleichsoperatoren und logische Operatoren fest in die Sprache einbauen. Natürlich sollen auch geklammerte Ausdrücke möglich sein.

4.1 Parser

Es versteht sich, dass wir für alle vorgesehen Operatoren Lexeme brauchen. Bei der `javacc`-Eingabedatei ist zu beachten, dass die Reihenfolge der Tokendefinitionen eine Rolle spielt. Ist ein Token das Präfix eines anderem, so sollte erst das längere Token definiert werden, denn der Lexer probiert die Lexeme in der Reihenfolge ihrer Definition.

```
78 TOKEN :
79 {
80    <#DIGIT: [ "0"-"9"] >
81   |<INTEGER_LITERAL: (<DIGIT>)+ >
82   |<MULT: "*">
83   |<DIV:  "/">
84   |<MOD:  "%">
85   |<PLUS: "+">
86   |<MINUS:"-">
87   |<EQ:   "==">
88   |<NEQ:  "!=">
89   |<LE:   "<=">
90   |<GE:   ">=">
91   |<LT:   "<">
92   |<GT:   ">">
93   |<OR:   "||">
94   |<AND:  "&&">
95
```

```
96    |<LPAR: "(">
97    |<RPAR: ")">
98  }
```

Listing 4.1: WIPParser.jj

Soweit die neuen Token in der Sprachdefinition.

Für die Präzedenz der Operatoren benutzen wir einen Trick bei der Formulierung der Grammatik. Anstatt eine einzige Produktionsregel zu definieren, die eine linken Ausdruck und einen rechten Ausdruck als Operanden hat und alle Operatoren dazwischen in gleicher Weise erlaubt, kann man eine Kaskade von Regeln definieren. Einstieg ist eine Regel für die Operatoren mit der geringsten Präzedenz. Die Operanden sind dann die Nichtterminale für die Produktionenregel der nächst höheren Präzendenz. Mit diesem Vorgehen kann man direkt einen abstrakten Syntaxbaum erzeugen, in dem die stärker bindenden Operatoren unter den schwächer bindenden sind, und somit zuerst berechnet werden.

Hier zunächst die Grammatik für Operatorausdrücke in Wip. Auf unäre Operatoren sei vorerst verzichtet.

expression	→	*andExpression*
andExpression	→	*orExpression* (`AND` *orExpression*)*
orExpression	→	*compareExpression* (`OR` *compareExpression*)*
compareExpression	→	*additiveExpression* (*compOp* *additiveExpression*)*
compOp	→	`EQ`| `NEQ`| `LE`| `LT`| `GE`| `GT`
additiveExpression	→	*multiplicativExpression* (*addOp* *multiplicativExpression*)*
addOp	→	`PLUS`| `MINUS`
multiplicativExpression	→	*unaryExpression* (*addOp* *unaryExpression*)*
multOp	→	`MULT`| `DIV`| `MOD`
unaryExpression	→	`INTLIT`|(`LPAR` *expression* `RPAR`)

Auch diese Grammatik lässt sich nun direkt für `javacc` formulieren.

der Einstiegspunkt für Ausdrücke ist nun der Ausdruck für das logische Und:

```
99  Tree expression():
100 {Tree result;}
101 {
102   result=andExpression()
103   {return result;}
104 }
```

Listing 4.2: WIPParser.jj

Die Regel für das logische Und hat das Nichtterminal der Regel für das logische Oder als Operanden.

```
105 Tree andExpression():
106 { Tree result;
107   Tree rightOperand;
108 }
109 {
110   result=orExpression()
111   ( ( <AND> rightOperand=orExpression()
112         {result=new AndExpr(result,rightOperand);}
113     )
114   )*
115   {return result;}
116 }
```

Listing 4.3: WIPParser.jj

Die Oder-Ausfdrücke haben Vergleichsausdrücke als Operanden.

```
117 Tree orExpression():
118 { Tree result;
119   Tree rightOperand;
120 }
121 {
122   result=compareExpression()
123   ( ( <OR> rightOperand=compareExpression()
124         {result=new OrExpr(result,rightOperand);}
125     )
126   )*
127   {return result;}
128 }
```

Listing 4.4: WIPParser.jj

Die Vergleichsausdrücke haben das Nichtterminal für die arihtmetischen Strichoperationen als Operanden.

```
129 Tree compareExpression():
130 { Tree result;
131   Tree rightOperand;
132 }
133 {
134   result=additiveExpression()
135   ( ( <EQ> rightOperand=additiveExpression()
136         {result=new EqExpr(result,rightOperand);}
137     |<NEQ> rightOperand=additiveExpression()
138         {result=new NeqExpr(result,rightOperand);}
139     |<LT> rightOperand=additiveExpression()
140         {result=new LtExpr(result,rightOperand);}
141     |<GT> rightOperand=additiveExpression()
```

```
142         [result=new GtExpr(result,rightOperand);]
143       |<LE> rightOperand=additiveExpression()
144         [result=new LeExpr(result,rightOperand);]
145       |<GE> rightOperand=additiveExpression()
146         [result=new GeExpr(result,rightOperand);]
147     )
148   )*
149   [return result;]
150 }
```

Listing 4.5: WIPParser.jj

Und diese schließlich führen zu den am stärksten bindenden Operatoren der Punktrechnung.

```
151 Tree additiveExpression():
152 { Tree result;
153   Tree rightOperand;
154 }
155 {
156   result=multiplicativeExpression()
157   ( ( <PLUS> rightOperand=multiplicativeExpression()
158         [result=new AddExpr(result,rightOperand);]
159       |<MINUS> rightOperand=multiplicativeExpression()
160         [result=new SubExpr(result,rightOperand);]
161     )
162   )*
163   [return result;]
164 }
```

Listing 4.6: WIPParser.jj

Die Punktrechnung kennt drei verschiedene Operatoren.

```
165 Tree multiplicativeExpression():
166 { Tree result;
167   Tree rightOperand;
168 }
169 {
170   result=unaryExpression()
171   ( ( <MULT> rightOperand=unaryExpression()
172         [result=new MultExpr(result,rightOperand);]
173       |<DIV> rightOperand=unaryExpression()
174         [result=new DivExpr(result,rightOperand);]
175       |<MOD> rightOperand=unaryExpression()
176         [result=new ModExpr(result,rightOperand);]
177     )
178   )*
179   [return result;]
180 }
```

Listing 4.7: WIPParser.jj

Schließlich landen wir in dieser Regelhierarchie ganz unten, bei der Produktionenregel ohne binären Operator. Hier können nur noch Literale oder nun auch geklammerte Ausdrücke stehen.

```
Tree unaryExpression() :
{Token tok;
 Tree result = null;
}
{
 (tok=<INTEGER_LITERAL>{result =  new IntLit(new Integer(tok.image));}
 |(<LPAR> result=expression() <RPAR>)
 )
 {return result;}
}
```

Listing 4.8: WIPParser.jj

In diesem Abschnitt ist einiges an Code entstanden, allerdings ist es immer wieder analoger Code. Ein alternativer Sprachentwurf wäre, keine Operatorpräzedenz in die Grammatik zu codieren, sondern nach dem Parsen, auf dem erzeugten Baum, die geparste Operatorenfolge nach Präzedenzen zu sortieren. Dieses ist auf jedem Fall notwendig, wenn in der Sprache die Programmierer eigene Operatoren mit einer bestimmten Präzedenz definieren können, wie z.B. in Haskell.

4.2 Abstrakter Syntaxbaum

Wir haben eine ganze Reihe neuer Ausrücke in der Grammatik definiert, die alle auch eigene Baumknoten bekommen. Allerdings sind die binären Operatorausdrücke alle gleich aufgebaut und haben zwei Operanden. deshalb definieren wir zunächst eine Baumklasse für allgemeine binäre Operatoren.

```
package v3.name.panitz.wip.tree;
import java.util.function.IntBinaryOperator;
public abstract class OpExpr implements Tree{
  public Tree left;
  public Tree right;
  public IntBinaryOperator op;
  String opName;
  public OpExpr(Tree left,Tree right,IntBinaryOperator op,String
   opName){
    this.left = left;
    this.right = right;
    this.op = op;
    this.opName = opName;
  }
  public String toString(){return getClass().getName()+"("+left+", "+
   right+")";}
```

```
15 ]
```

Listing 4.9: OpExpr.java

Ein solches Objekt enthält neben den Operanden, die Funktion, die der Operator ausführt und den textuellen Namen des Operators. Die Klasse ist allerdings abstrakt. Sie enthält nicht die Methode `welcome` aus der Schnittstelle `Tree`.

Für jeden Operator schreiben wir nun eine Unterklasse von `OpExpr`. Diese Klassen implementieren die fehlende Methode und sie initialisieren im Konstruktor mit Hilfe eines Lambda-Ausdrucks die Funktion, die sie realisieren.

Es folgen zunächst die Baumklassen für die arithmetischen Operatoren:

```
16 package v3.name.panitz.wip.tree;
17 public class MultExpr extends OpExpr{
18   public MultExpr(Tree left,Tree right){super(left,right,(x,y)->x*y,"*
       ");}
19   public <R> R welcome(Visitor<R> visitor) throws Exception{
20     return visitor.visit(this);
21   }
22 }
```

Listing 4.10: MultExpr.java

```
1 package v3.name.panitz.wip.tree;
2 public class ModExpr extends OpExpr{
3   public ModExpr(Tree left,Tree right){super(left,right,(x,y)->x%y,"%"
      );}
4   public <R> R welcome(Visitor<R> visitor) throws Exception{
5     return visitor.visit(this);
6   }
7 }
```

Listing 4.11: ModExpr.java

```
1 package v3.name.panitz.wip.tree;
2 public class DivExpr extends OpExpr{
3   public DivExpr(Tree left,Tree right){super(left,right,(x,y)->x/y,"/"
      );}
4   public <R> R welcome(Visitor<R> visitor) throws Exception{
5     return visitor.visit(this);
6   }
7 }
```

Listing 4.12: DivExpr.java

```
package v3.name.panitz.wip.tree;
public class AddExpr extends OpExpr{
  public AddExpr(Tree left,Tree right){super(left,right,(x,y)->x+y,"+");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.13: AddExpr.java

```
package v3.name.panitz.wip.tree;
public class SubExpr extends OpExpr{
  public SubExpr(Tree left,Tree right){super(left,right,(x,y)->x-y,"-");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.14: SubExpr.java

Bool'sche Werte stellen wir intern in WIP als die Zahlen 0 und 1 dar. Entsprechend sind die Lamda-Ausdrücke bei den Vergleichsoperatoren so implementiert, dass als Ergebnis kein bool'scher Wert steht, sonder eine der beiden Zahlen.

```
package v3.name.panitz.wip.tree;
public class EqExpr extends OpExpr{
  public EqExpr(Tree left,Tree right){super(left,right,(x,y)->x==y
    ?1:0,"==");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.15: EqExpr.java

```
package v3.name.panitz.wip.tree;
public class NeqExpr extends OpExpr{
  public NeqExpr(Tree left,Tree right){super(left,right,(x,y)->x!=y
    ?1:0,"!=");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.16: NeqExpr.java

```
package v3.name.panitz.wip.tree;
public class LeExpr extends OpExpr{
  public LeExpr(Tree left,Tree right){super(left,right,(x,y)->x<=y
    ?1:0,"<=");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.17: LeExpr.java

```
package v3.name.panitz.wip.tree;
public class GeExpr extends OpExpr{
  public GeExpr(Tree left,Tree right){super(left,right,(x,y)->x>=y
    ?1:0,">=");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.18: GeExpr.java

```
package v3.name.panitz.wip.tree;
public class LtExpr extends OpExpr{
  public LtExpr(Tree left,Tree right){super(left,right,(x,y)->x<y?1:0,
    "<");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.19: LtExpr.java

```
package v3.name.panitz.wip.tree;
public class GtExpr extends OpExpr{
  public GtExpr(Tree left,Tree right){super(left,right,(x,y)->x>y?1:0,
    "->");}
  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 4.20: GtExpr.java

Entsprechend sind auch die Lambda-Ausdrücke der logischen Operatoren unter diesen Gesichtspunkt zu implementieren.

```
1  package v3.name.panitz.wip.tree;
2  public class AndExpr extends OpExpr{
3    public AndExpr(Tree left,Tree right){super(left,right,(x,y)->(x==1)?
       y:1,"&&");}
4    public <R> R welcome(Visitor<R> visitor) throws Exception{
5      return visitor.visit(this);
6    }
7  }
```

Listing 4.21: AndExpr.java

```
1  package v3.name.panitz.wip.tree;
2  public class OrExpr extends OpExpr{
3    public OrExpr(Tree left,Tree right){super(left,right,(x,y)->(x==1)
       ?1:y,"||");}
4    public <R> R welcome(Visitor<R> visitor) throws Exception{
5      return visitor.visit(this);
6    }
7  }
```

Listing 4.22: OrExpr.java

Die Besucherschnittstelle wird um eine überladene Version der Methode `visit` für jeden neuen Baumknoten erweitert.

```
10 package v3.name.panitz.wip.tree;
11 public interface Visitor<R>  {
12   R visit(IntLit il) throws Exception;
13   R visit(MultExpr ex) throws Exception;
14   R visit(DivExpr ex) throws Exception;
15   R visit(ModExpr ex) throws Exception;
16   R visit(AddExpr ex) throws Exception;
17   R visit(SubExpr ex) throws Exception;
18   R visit(EqExpr ex) throws Exception;
19   R visit(NeqExpr ex) throws Exception;
20   R visit(LeExpr ex) throws Exception;
21   R visit(GeExpr ex) throws Exception;
22   R visit(LtExpr ex) throws Exception;
23   R visit(GtExpr ex) throws Exception;
24   R visit(OrExpr ex) throws Exception;
25   R visit(AndExpr ex) throws Exception;
26 }
```

Listing 4.23: Visitor.java

4.2.1 Pretty Printer

Im *pretty printer* können wir eine generische Lösung für die Operatorausdrücke implementieren.

```
22   public Integer visit(OpExpr ex) throws Exception{
23     w.write("(");
24     ex.left.welcome(this);
25     w.write(" "+ex.opName+" ");
26     ex.right.welcome(this);
27     w.write(")");
28     return indent;
29   }
```

Listing 4.24: PP.java

Trotzdem verlangt die Besucherschnitstelle, dass wir für jeden Baumknoten die Methode `visit` implementieren. Für alle Knoten von binären Operatorausdrücken sieht diese Implementierung identisch aus und ruft die allgemeine Version für Operatorausdrücke auf:

```
30   public Integer visit(MultExpr ex) throws Exception{
31     return visit((OpExpr)ex);
32   }
33   public Integer visit(DivExpr ex) throws Exception{
34     return visit((OpExpr)ex);
35   }
36   public Integer visit(ModExpr ex) throws Exception{
37     return visit((OpExpr)ex);
38   }
39   public Integer visit(AddExpr ex) throws Exception{
40     return visit((OpExpr)ex);
41   }
42   public Integer visit(SubExpr ex) throws Exception{
43     return visit((OpExpr)ex);
44   }
45   public Integer visit(EqExpr ex) throws Exception{
46     return visit((OpExpr)ex);
47   }
48   public Integer visit(NeqExpr ex) throws Exception{
49     return visit((OpExpr)ex);
50   }
51   public Integer visit(LeExpr ex) throws Exception{
52     return visit((OpExpr)ex);
53   }
54   public Integer visit(GeExpr ex) throws Exception{
55     return visit((OpExpr)ex);
56   }
57   public Integer visit(LtExpr ex) throws Exception{
58     return visit((OpExpr)ex);
59   }
60   public Integer visit(GtExpr ex) throws Exception{
61     return visit((OpExpr)ex);
62   }
63   public Integer visit(OrExpr ex) throws Exception{
64     return visit((OpExpr)ex);
```

```
65    }
66    public Integer visit(AndExpr ex) throws Exception{
67      return visit((OpExpr)ex);
68    }
69  }
```

Listing 4.25: PP.java

4.2.2 Interpreter

Analog wie bei dem ersten Besucher, lässt sich beim zweiten Besucher vorgehen. Ebenso wie ein Operatorausdruckknoten seine textuelle Darstellung enthält, enthält er seine Funktion, die er realisiert. So lässt sich für alle Ausdrücke einheitlich der Interpreter formulieren:

```
14    public Integer visit(OpExpr ex) throws Exception{
15      int op1 = ex.left.welcome(this);
16      int op2 = ex.right.welcome(this);
17      return ex.op.applyAsInt(op1, op2);
18    }
```

Listing 4.26: Interpreter.java

Alle anderen überladenen Versionen rufen diese allgemeine Methode direkt auf.

```
19    public Integer visit(MultExpr ex) throws Exception{
20      return visit((OpExpr)ex);
21    }
22    public Integer visit(DivExpr ex) throws Exception{
23      return visit((OpExpr)ex);
24    }
25    public Integer visit(ModExpr ex) throws Exception{
26      return visit((OpExpr)ex);
27    }
28    public Integer visit(AddExpr ex) throws Exception{
29      return visit((OpExpr)ex);
30    }
31    public Integer visit(SubExpr ex) throws Exception{
32      return visit((OpExpr)ex);
33    }
34    public Integer visit(EqExpr ex) throws Exception{
35      return visit((OpExpr)ex);
36    }
37    public Integer visit(NeqExpr ex) throws Exception{
38      return visit((OpExpr)ex);
39    }
40    public Integer visit(LeExpr ex) throws Exception{
41      return visit((OpExpr)ex);
42    }
```

```
43    public Integer visit(GeExpr ex) throws Exception{
44      return visit((OpExpr)ex);
45    }
46    public Integer visit(LtExpr ex) throws Exception{
47      return visit((OpExpr)ex);
48    }
49    public Integer visit(GtExpr ex) throws Exception{
50      return visit((OpExpr)ex);
51    }
52    public Integer visit(OrExpr ex) throws Exception{
53      return visit((OpExpr)ex);
54    }
55    public Integer visit(AndExpr ex) throws Exception{
56      return visit((OpExpr)ex);
57    }
58  }
```

Listing 4.27: Interpreter.java

4.3 Abstrakte StackMaschine

4.3.1 Befehlssatz

Den Befehlssatz der abstrakten Maschine erweitern wir nun für alle Operatoren:

```
18
19    static public enum Instr {PushInt, Mult, Div, Mod, Add, Sub
20                              , Eq, Neq, Lt, Gt, Le, Ge, And, Or};
```

Listing 4.28: Instruction.java

4.3.2 Maschineninterpreter

Alle diese Befehle führen ihre Rechnung auf den obersten zwei Kellerelementen aus. Wie schon bei dem Interpreter sind die bool'schen Werte mit den ganzen Zahlen 0 und 1 dargestellt, was bei der berechnung natürlich zu berücksichtigen ist.

```
37    void step(){
38      switch (code[ic].instr) {
39      case PushInt: stack[stc++] = code[ic].a1;ic++;break;
40      case Mult:
41       stack[stc-2] = stack[stc-2] * stack[stc-1];stc--;ic++;break;
42      case Mod:
43       stack[stc-2] = stack[stc-2] % stack[stc-1];stc--;ic++;break;
44      case Div:
```

```
45      stack[stc-2] = stack[stc-2] / stack[stc-1];stc--;ic++;break;
46    case Add:
47      stack[stc-2] = stack[stc-2] + stack[stc-1];stc--;ic++;break;
48    case Sub:
49      stack[stc-2] = stack[stc-2] - stack[stc-1];stc--;ic++;break;
50    case Eq:
51      stack[stc-2] = stack[stc-2] == stack[stc-1]?1:0;stc--;ic++;break;
52    case Neq:
53      stack[stc-2] = stack[stc-2] != stack[stc-1]?1:0;stc--;ic++;break;
54    case Lt:
55      stack[stc-2] = stack[stc-2] < stack[stc-1]?1:0;stc--;ic++;break;
56    case Gt:
57      stack[stc-2] = stack[stc-2] > stack[stc-1]?1:0;stc--;ic++;break;
58    case Le:
59      stack[stc-2] = stack[stc-2] <= stack[stc-1]?1:0;stc--;ic++;break;
60    case Ge:
61      stack[stc-2] = stack[stc-2] >= stack[stc-1]?1:0;stc--;ic++;break;
62    case And:
63      stack[stc-2] =(stack[stc-2]==1)?stack[stc-1]:0;stc--;ic++;break;
64    case Or:
65      stack[stc-2] =(stack[stc-2]==1)?1:stack[stc-1];stc--;ic++; break;
66    }
67  }
```

Listing 4.29: StackMachine.java

4.3.3 Code Generierung

Widmen wir uns jetzt dem dritten Besucher, der den abstrakten Maschinencode erzeugt. Hier wird für jeden Operatorausdruck eine anderer Maschinenbefehl erzeugt. Hierbei hilft das Besuchsmuster mit der überladenen Methode `visit`, um die entsprechende Fallunterscheidung zu treffen. Allen Operatorausdrücken gemein ist, dass erst Code zu erzeugen ist, der die beiden Operanden auf den Keller legt. Daher implementieren wir zunächst eine gemeinsame Methode für alle Operatorausdrücke, in der dafür gesorgt wird, dass Code für die Operanden erzeugt wird;

```
22  public void visit(OpExpr ex) throws Exception{
23    ex.left.welcome(this);
24    ex.right.welcome(this);
25  }
```

Listing 4.30: GenCode.java

Für jeden Operatorausdruck wird zunächst diese Methode aufgerufen, bevor der spezifische Maschinenbefehl für die entsprechende Berechnung generiert wird. Eine kleine Hilfmethode hilft für einen Operatorausdruck und gegebenen Maschinenbefehl den entsprechenden Code zu generieren.

```
26     private List<Instruction> gen(OpExpr ex, Instruction ins) throws
       Exception{
27      visit(ex);
28      result.add(ins);
29      return result;
30    }
```

Listing 4.31: GenCode.java

Dadurch werden die einzelnen Besuchermethoden für die Operatorausdrücke Einzeiler:

```
31    public List<Instruction> visit(MultExpr ex) throws Exception{
32      return gen(ex,new Instruction(Instruction.Instr.Mult));
33    }
34    public List<Instruction> visit(DivExpr ex) throws Exception{
35      return gen(ex,new Instruction(Instruction.Instr.Div));
36    }
37    public List<Instruction> visit(ModExpr ex) throws Exception{
38      return gen(ex,new Instruction(Instruction.Instr.Mod));
39    }
40    public List<Instruction> visit(AddExpr ex) throws Exception{
41      return gen(ex,new Instruction(Instruction.Instr.Add));
42    }
43    public List<Instruction> visit(SubExpr ex) throws Exception{
44      return gen(ex,new Instruction(Instruction.Instr.Sub));
45    }
46    public List<Instruction> visit(EqExpr ex) throws Exception{
47      return gen(ex,new Instruction(Instruction.Instr.Eq));
48    }
49    public List<Instruction> visit(NeqExpr ex) throws Exception{
50      return gen(ex,new Instruction(Instruction.Instr.Neq));
51    }
52    public List<Instruction> visit(LeExpr ex) throws Exception{
53      return gen(ex,new Instruction(Instruction.Instr.Le));
54    }
55    public List<Instruction> visit(GeExpr ex) throws Exception{
56      return gen(ex,new Instruction(Instruction.Instr.Ge));
57    }
58    public List<Instruction> visit(LtExpr ex) throws Exception{
59      return gen(ex,new Instruction(Instruction.Instr.Lt));
60    }
61    public List<Instruction> visit(GtExpr ex) throws Exception{
62      return gen(ex,new Instruction(Instruction.Instr.Gt));
63    }
64    public List<Instruction> visit(OrExpr ex) throws Exception{
65      return gen(ex,new Instruction(Instruction.Instr.Or));
66    }
67    public List<Instruction> visit(AndExpr ex) throws Exception{
68      return gen(ex,new Instruction(Instruction.Instr.And));
69    }
```

```
70 ]
```

Listing 4.32: GenCode.java

4.4 Assembler Generierung

Auch der X86 Assembler kennt Befehle für die einzelnen Operatoren. Allerdings passen Sie nicht eins zu eins zu den Befehlen unserer Maschine. Sie arbeiten auf unterschiedliche Weise mit den Registern der Maschine. Insbesondere Vergleiche werden unterschiedlich behandelt und auch die Divisionsoperatoren verdienen dabei noch eine besondere Beachtung.

Hier zunächst die Befehle des Assembler. Es gibt nur einen Divisionsoperator, mit dem sowohl die ganzzahlige Division als auch der Rest berechnet wird, also die Modulo Opertion. Für alle Vergleichsoperationen steht ein einziger Assmeblerbefehl zur Verfügung, der Befehl `cmp`.

```
50   String mul(){return _32?"mull":"mulq";}
51   String add(){return _32?"addl":"addq";}
52   String sub(){return _32?"subl":"subq";}
53   String div(){return _32?"divl":"divq";}
54   String and(){return _32?"andl":"andq";}
55   String or(){return _32?"orl":"orq";}
56   String cmp(){return _32?"cmpl":"cmpq";}
```

Listing 4.33: WriteX86.java

Um die unterschiedlichen Vergleiche unserer abstrakten Maschine umsetzen zu können, benötigen wir verschiedene Sprungbefehle des Assemblers. Die Sprungbefehle springen abhängig von dem Ergebnis des letzten Aufrufs der Verglaichsoperation zu einem angegebenen Label.

```
57   String jmp(){return "jmp";}
58   String je(){return "je";}
59   String jne(){return "jne";}
60   String jl(){return "jl";}
61   String jle(){return "jle";}
62   String jg(){return "jg";}
63   String jge(){return "jge";}
```

Listing 4.34: WriteX86.java

Ebenso werden also Sprunglabel benötigt, die wir mit einer statischen Methoden auf Bedarf generieren:

```
64   static int lbl = 0;
65   public static String label() {
66     lbl++;
67     return "label"+lbl;
68   }
```

Listing 4.35: WriteX86.java

Für eine Vergleichoperation sind die beiden Operanden vom Keller in je ein Register zu laden. Dann ist der Vergleichsbefehl `cmp` auszuführen. Anschließend wird der bedingte Sprungbefehl passend zu der durchzuführenden Vergleichsoperation auszuführen. Im positiven Fall wird zu einem Label gesprungen, bei dem die Zahl 1 auf den Keller gelegt wird, ansonsten wird die Zahl 0 auf den Keller gelegt. Dieses lässt sich alles in einer internen Hilfsmethode generieren.

```
69   private void genCompareExpr(Writer out,String jmp)throws Exception{
70     out.write("    "+pop()+"    "+bx()+"\n");
71     out.write("    "+pop()+"    "+ax()+"\n");
72     out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
73
74     String endLabel = label();
75     String posLabel = label();
76
77     out.write("    "+jmp+"    "+posLabel+"\n");
78     out.write("    "+push()+"    $0\n");
79     out.write("    "+jmp()+"    "+endLabel+"\n");
80     out.write("    "+posLabel+":\n");
81     out.write("    "+push()+"    $1\n");
82     out.write("    "+endLabel+":\n");
83   }
```

Listing 4.36: WriteX86.java

Nun sind unsere abstrakten Vergleichsoperationen durch Aufruf dieser Methode mit dem passenden bedingten Sprungbefehl aufzurufen.

```
84   public void writeAssembler(Writer out,Instruction ins)throws
      Exception{
85     switch (ins.instr) {
86     case PushInt:
87       out.write("    "+push()+"    $"+ins.a1+"\n");
88       break;
89
90     case Eq: genCompareExpr(out,je());break;
91     case Neq: genCompareExpr(out,jne()); break;
92     case Lt: genCompareExpr(out,jl()); break;
93     case Le: genCompareExpr(out,jle()); break;
94     case Gt: genCompareExpr(out,jg()); break;
95     case Ge: genCompareExpr(out,jge()); break;
```

Listing 4.37: WriteX86.java

Die Generierung der Multiplikation haben wir bereits im letzten Abschnitt gesehen:

```
96      case Mult:
97        out.write("    "+pop()+"    "+bx()+"\n");
98        out.write("    "+pop()+"    "+ax()+"\n");
99        out.write("    "+mul()+"    "+bx()+"\n");
100       out.write("    "+push()+"    "+ax()+"\n");
101       break;
```

Listing 4.38: WriteX86.java

Ähnlich lässt sich die Addition und Subtraktion umsetzen. Allerdings rufen wir die entsprechenden Befehle im Assembler mit zwei Registern als Argumente auf:

```
102     case Add:
103       out.write("    "+pop()+"    "+bx()+"\n");
104       out.write("    "+pop()+"    "+ax()+"\n");
105       out.write("    "+add()+"    "+bx()+", "+ax()+"\n");
106       out.write("    "+push()+"    "+ax()+"\n");
107       break;
108
109     case Sub:
110       out.write("    "+pop()+"    "+bx()+"\n");
111       out.write("    "+pop()+"    "+ax()+"\n");
112       out.write("    "+sub()+"    "+bx()+", "+ax()+"\n");
113       out.write("    "+push()+"    "+ax()+"\n");
114       break;
```

Listing 4.39: WriteX86.java

Die beiden Divisionsoperatoren benötigen zusätzlich das Register `dx`, auf dem der Rest der ganzzahligen Division zu finden ist.

```
115     case Div:
116       out.write("    "+pop()  +"    "+bx()+"\n");
117       out.write("    "+pop()  +"    "+ax()+"\n");
118       out.write("    "+mov()  +"    $0, "+dx()+"\n");
119       out.write("    "+div()+"    "+bx()+"\n");
120       out.write("    "+push()+"    "+ax()+"\n");
121       break;
122
123     case Mod:
124       out.write("    "+pop()  +"    "+bx()+"\n");
125       out.write("    "+pop()  +"    "+ax()+"\n");
126       out.write("    "+mov()  +"    $0, "+dx()+"\n");
127       out.write("    "+div()+"    "+bx()+"\n");
128       out.write("    "+push()+"    "+dx()+"\n");
```

```
129         break;
```

Listing 4.40: WriteX86.java

Im Assembler existieren auch Befehle entsprechend der logischen Operatoren, die wir in der abstrakten Maschine vorgesehen haben.

```
130     case And:
131       out.write("    "+pop()+"    "+bx()+"\n");
132       out.write("    "+pop()+"    "+ax()+"\n");
133       out.write("    "+and()+"    "+bx()+", "+ax()+"\n");
134       out.write("    "+push()+"    "+ax()+"\n");
135       break;
136 
137     case Or:
138       out.write("    "+pop()+"    "+bx()+"\n");
139       out.write("    "+pop()+"    "+ax()+"\n");
140       out.write("    "+or()+"    "+bx()+", "+ax()+"\n");
141       out.write("    "+push()+"    "+ax()+"\n");
142       break;
143     }
144   }
145 }
```

Listing 4.41: WriteX86.java

Kapitel 5

Version 4: Verzweigungen

Ein wichtiger Bestandteil ein jeder Programmiersprache ist die Möglichkeit eine Fallunterscheidung aufgrund eines bool'schen Wertes zu treffen. In dieser Version von WIP fügen wir der Sprache einen `if`-Ausdruck zu.

5.1 Parser

Wir definieren drei neue Token. Die Schlüsselwörter `if`, `then` und `else`.

```
191   |<IF:    "if">
192   |<THEN:  "then">
193   |<ELSE:  "else">
```

Listing 5.1: WIPParser.jj

Die Regel für Ausdrücke wird um eine Alternative für `if`-Ausdrücke erweitert:

```
194 Tree expression():
195 {Tree result;}
196 {
197   (
198    result=ifExpression()
199   |result=andExpression()
200   )
201   {return result;}
202 }
```

Listing 5.2: WIPParser.jj

Ein `if`-Ausdruck besteht immer aus einer Bedingung, einer positiven Alternative und einer negativen Alternative. Damit hat ein `if`-Asudruck immer einen Wert, der ausgerechnet werden kann und entspricht dem terziären Bedinmgungsoperator aus C-ähnlichen Sprachen.

```
Tree ifExpression():
{
  Tree cond;
  Tree alt1;
  Tree alt2;
}
{
  <IF> cond=expression() <THEN> alt1=expression() <ELSE> alt2=
    expression()
   {return new IfExpr(cond,alt1,alt2);}
}
```

Listing 5.3: WIPParser.jj

5.2 Abstrakter Syntaxbaum

Wir schreiben eine neue Knotenklasse, die `if`-Ausdrücke repräsentiert.

```
package v4.name.panitz.wip.tree;
public  class IfExpr implements Tree{
  public Tree cond;
  public Tree alt1;
  public Tree alt2;
  public IfExpr(Tree cond,Tree alt1,Tree alt2){
    this.cond = cond;
    this.alt1 = alt1;
    this.alt2 = alt2;
  }
  public String toString(){return "IfExpr("+cond+", "+alt1+", "+alt2+"
    )";}

  public <R> R welcome(Visitor<R> visitor) throws Exception{
    return visitor.visit(this);
  }
}
```

Listing 5.4: IfExpr.java

Die Besucherschnittstelle wird entsprechend um eine weitere überladene Methode `visit` erweitert:

```
package v4.name.panitz.wip.tree;
public interface Visitor<R>  {
  R visit(IntLit il) throws Exception;
  R visit(MultExpr ex) throws Exception;
  R visit(DivExpr ex) throws Exception;
  R visit(ModExpr ex) throws Exception;
```

```
33   R visit(AddExpr ex) throws Exception;
34   R visit(SubExpr ex) throws Exception;
35   R visit(EqExpr ex) throws Exception;
36   R visit(NeqExpr ex) throws Exception;
37   R visit(LeExpr ex) throws Exception;
38   R visit(GeExpr ex) throws Exception;
39   R visit(LtExpr ex) throws Exception;
40   R visit(GtExpr ex) throws Exception;
41   R visit(OrExpr ex) throws Exception;
42   R visit(AndExpr ex) throws Exception;
43   R visit(IfExpr ex) throws Exception;
44 }
```

Listing 5.5: Visitor.java

5.2.1 Pretty Printer

Um einen `if`-Ausdruck wieder textuell darzustellen, nutzen wir erstmals die Möglichkeit, den Quelltext einzurücken. Bei einem Zeilenumbruch beginnt die neue Zeile mit Leerzeichen der Länge des internen Feldes `indent`:

```
70   public Integer visit(IfExpr ex) throws Exception{
71     w.write("if");
72     indent+=2;
73     newLine();
74     ex.cond.welcome(this);
75     indent-=2;
76     newLine();
77     w.write("then");
78     indent+=2;
79     newLine();
80     ex.alt1.welcome(this);
81     indent-=2;
82     newLine();
83     w.write("else");
84     indent+=2;
85     newLine();
86     ex.alt2.welcome(this);
87     indent-=2;
88     return indent;
89   }
90   private void newLine() throws Exception{
91     w.write("\n");
92     for (int i=0;i<indent;i++) w.write(" ");
93   }
94 }
```

Listing 5.6: PP.java

5.2.2 Interpreter

Besonders einfach wird der Besucher, der den AST direkt berechnet, im Fall für `if`-Ausdrücke. Zunächst wird der Besucher zur Bedingung des `if`-Ausdrucks zu Besuch geschickt. Abhängig von dem Ergebnis wird der Besucher in die erste oder in die zweite Alternative geschickt.

```
59  public Integer visit(IfExpr ex) throws Exception{
60    int p = ex.cond.welcome(this);
61    return ((p==1) ? ex.alt1 : ex.alt2).welcome(this);
62  }
```

Listing 5.7: Interpreter.java

5.3 Abstrakte StackMaschine

5.3.1 Befehlssatz

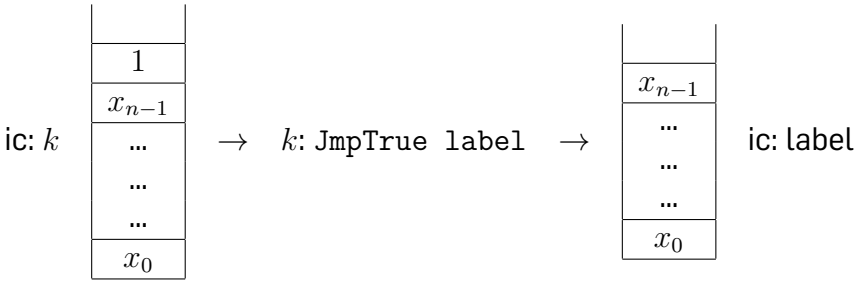
Zur Umsetzung der Fallunterscheidung benötigen wir auch in unserer abstrakten Maschine Sprungebefehle, die es erlauben zu definierten Labels im Programmcode zu springen. Wir sehen zwei Sprungbefehle vor. Ein bedingter, der abhängig vom Wert des obersten Kellerelements den Sprung durchführt und ein unbedingter Sprungbefehl. Auch die Label innerhalb des abstrakten Maschinencodes werden der Einfachheit halber als Befehle modelliert.

```
21  static public enum Instr
22    { PushInt, Mult, Div, Mod, Add, Sub
23    , Eq, Neq, Lt, Gt, Le, Ge, And, Or
24    , JmpTrue, Jmp, Label};
25
26  public Instr instr;
27  public int a1;
28  public String label;
```

Listing 5.8: Instruction.java

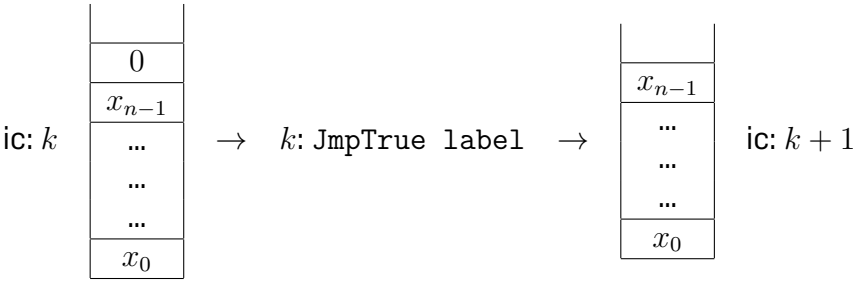
5.3.2 Maschineninterpreter

Den bedingten Sprungbefehl `JmpTrue` werden spezifiziert, indem der Zustandsübergang für diesen Befehl angegeben wird. Es gibt zwei unterschiedliche Fälle. Der Fall wenn der oberste Kellerwert die Zahl 1 darstellt:



In diesem Fall wird der Programmzähler auf die Position des Sprunglabels gesetzt. Das oberste Element wird vom Keller genommen.

Im Fall, dass das oberste Kellerelement die Zahl 0 ist, wird kein Sprung durchgeführt, sondern der Programmzähler um eins erhöht.



Auch hier wird das oberste Element vom Keller genommen.

Zur Implementierung der Sprungbefehl in der abstrakten Maschine, müssen wir für jedes Label die Position des Labels im Programmcode kennen. Diese Information wird in einem Abbildungsobjekt innerhalb der Maschine festgehalten.

```
68 package v4.name.panitz.wip.stackmachine;
69 import java.util.Map;
70 import java.util.HashMap;
71
72 public class StackMachine{
73   public Map<String,Integer> labels = new HashMap<>();
```

Listing 5.9: StackMachine.java

Bei der Umsetzung der Sprungbefehle wird in der Abbildung die Position des im Sprungbefehl als Ziel angegebenen Labels nachgeschlagen, um den Programmzähler auf diesen zu setzen.

```
74     case Jmp:  ic = labels.get(code[ic].label);break;
```

Listing 5.10: StackMachine.java

Die Umsetzung des bedingten Sprungbefehls, betrachtet zunächst den Wert des obersten Kellerelements, um abhängig davon den Programmzähler auf die Position des Labels zu setzen oder um eins zu erhöhen. Der Kellerzeiger wird anschließend um eins verringert.

```
75      case JmpTrue:
76          if (stack[stc-1]==1){
77             ic = labels.get(code[ic].label);
78           }else {
79             ic++;
80           }
81           stc--;
82           break;
```

Listing 5.11: StackMachine.java

Labels sind in unserer Maschine auch Befehle, die allerdings nichts anderes bewirken, als die Erhöhung des Programmzählers um eins.

```
83      case Label:   ic++;break;
```

Listing 5.12: StackMachine.java

5.3.3 Code Generierung

Bei der Codegenerierung sind nicht nur entsprechend der `if`-Ausdrücke Sprünge zu Labels zu generieren, sondern auch die Abbildung aufzubauen, die verzeichnet, welches Label welcher Position im erzeugten Programmcode entspricht. Hierzu erhält der Besucher intern eine Abbildung, und auch wieder eine statische Methode, um neue Labelnamen zu generieren:

```
71 public class GenCode implements Visitor<List<Instruction>>{
72   List<Instruction> result = new LinkedList<>();
73   public Map<String,Integer> labels = new HashMap<>();
74
75   static int lbl = 0;
76   public static String label() {
77     lbl++;
78     return "label"+lbl;
79   }
```

Listing 5.13: GenCode.java

```
80   public List<Instruction> visit(IfExpr ex) throws Exception{
81     String thenLabel = label();
82     String endIfLabel = label();
```

```
83
84      ex.cond.welcome(this);
85      result.add(new Instruction(Instruction.Instr.JmpTrue,thenLabel));
86      ex.alt2.welcome(this);
87      result.add(new Instruction(Instruction.Instr.Jmp,endIfLabel));
88      labels.put(thenLabel,result.size());
89      result.add(new Instruction(Instruction.Instr.Label,thenLabel));
90      ex.alt1.welcome(this);
91      labels.put(endIfLabel,result.size());
92      result.add(new Instruction(Instruction.Instr.Label,endIfLabel));
93      return result;
94    }
```

Listing 5.14: GenCode.java

5.4 Assembler Generierung

Kommen wir zur Umsetzung der drei neuen Befehle in Assemblercode:

Der einfache Sprungbefehl und die Labels lassen sich direkt in Assemblercode übersetzen.

```
146     case Jmp:
147       out.write("    "+jmp()+"    "+ins.label+"\n");
148       break;
149
150     case Label:
151       out.write(ins.label+":\n");
152       break;
```

Listing 5.15: WriteX86.java

Für den bedingten Sprungbefehl ist zunächst zu vergleichen, ob das oberste Kellerelement die Zahl 1 ist, um dann abhängig von dem Vergleich des Wertes mit 1 mit dem Assemblersprungbefehl `je` zum Label des Befehls gesprungen wird.

```
153     case JmpTrue:
154       out.write("    "+pop()+"    "+ax()+"\n");
155       out.write("    "+mov()+"    $1, "+bx()+"\n");
156       out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
157       out.write("    je     "+ins.label+"\n");
158       break;
159     }
160   }
161 }
```

Listing 5.16: WriteX86.java

Kapitel 6

Version 5: Funktionen

In diesem Kapitel erweitern wir die Sprache WIP um das wahrscheinlich wichtigste Konstrukt einjeder Programmiersprache: der Möglichkeit, Funktionen zu definieren und diese mit Argumenten aufzurufen. Bis zu diesem Kapitel war die Sprache WIP nichts weiter als ein Auswerter konstanter Ausdrücke, also eher ein kleiner Taschenrechner. Mit Ende diesem Kapitels steht uns eine Turing-berechenbare Programmiersprache zur Verfügung.

Über die Parameter der Funktionen haben wir dann auch erstmals das Konzept von Variablen in der Sprache WIP.

6.1 Parser

Wir führen zwei neue Token ein. Zunächst einmal ein Schlüsselwort für Funktionen:

```
213   |<FUN:   "fun">
```

Listing 6.1: WIPParser.jj

Desweiteren ein Token für Bezeichner, die für Funktionsnamen und Variablennamen verwendet werden:

```
214   |<IDENT: <LETTER> (<LETTER> | <DIGIT>)*>
215   |<#LETTER: ["$","A"-"Z","_","a"-"z"]>
216 ]
```

Listing 6.2: WIPParser.jj

In der Grammatik definieren wir jetzt, dass ein Programm eine Folge von Funktionsdefinitionen ist. [1].

[1] Es ist dann in der Verantwortung des Programmierers, dass es eine Funktion Namens `go()` gibt

```
217 Tree prog():
218 {
219   Tree expr;
220   FunDef fun;
221   List<FunDef> funs = new ArrayList<FunDef>();
222 }
223 {
224   (fun = funDef() {funs.add(fun);}  )*
225   <EOF>
226
227   {return new Program(funs);}
228 }
```

Listing 6.3: WIPParser.jj

Eine Funktionsdefinition wird mit dem Schlüsselwort `fun` eingeleitet. Es folgt der Funktionsname, die in Klammern eingeschlossenen Parameter und der Funktionsrumpf, der mit einem Gleichheitszeichen eingeleitet wird.

```
229 FunDef funDef():
230 {
231   Token tok;
232   String funName;
233   Tree definition;
234   List<String> args = new ArrayList<String>();
235 }
236 {
237   <FUN> tok=<IDENT> {funName = tok.image;}
238   <LPAR> [tok=<IDENT>{args.add(tok.image);} (<COMMA> tok=<IDENT> {args
      .add(tok.image);})* ] <RPAR>
239   <EQ> definition = expression()
240
241   {return new FunDef(funName,args,definition);}
242 }
```

Listing 6.4: WIPParser.jj

Die Regel für unäre Ausdrücke wird erweitert um die Möglichkeit eine Variable oder einen Funktionsaufruf aus Ausdruck zu formulieren:

```
243 Tree unaryExpression() :
244 {Token tok;
245  Tree result = null;
246  Tree arg;
247  List<Tree> args = new ArrayList<Tree>();
248 }
249 {
250    (tok=<INTEGER_LITERAL>{result =  new IntLit(new Integer(tok.image))
     ;}
251    |(<LPAR> result=expression() <RPAR>)
```

```
252     |result = funCallOrVar()
253     )
254     {return result;}
255 }
```

Listing 6.5: WIPParser.jj

Wenn nun in einem Ausdruck ein Bezeichner auftritt, so ist dieses entweder eine Variable, es sei denn es folgen in Klammern Argumente. Im letzteren Fall handelt es sich dann um einen Funktionsaufruf.

```
256 Tree funCallOrVar() :
257 {Token tok;
258  Tree result = null;
259  Tree arg;
260  List<Tree> args = new ArrayList<Tree>();
261  boolean isFunCall = false;
262 }
263 {
264   tok=<IDENT>
265   [<LPAR>
266      arg=expression(){args.add(arg);}  (<COMMA> arg=expression() {args
     .add(arg);})*
267      {
268       result = new FunCall(tok.image,args);
269       isFunCall = true;
270      }
271    <RPAR>
272   ]
273   {if (!isFunCall){result = new Var(tok.image);}
274   return result;
275   }
```

Listing 6.6: WIPParser.jj

Jetzt können wir die ersten üblichen Testprogramme in WIP formulieren, wie die Fakultät:

```
1 fun factorial(n) = if n == 0 then 1 else n*factorial(n-y)
2
3 fun go() =  factorial(5)
```

Listing 6.7: factorial.wip

Und die Berechnung der Fibonaccizahl:

```
4 fun fib(n) = if n <= 1 then n else fib(n-2)+fib(n-1)
5
6 fun go() =  fib(5)
```

Listing 6.8: factorial.wip

6.2 Abstrakter Syntaxbaum

Vier neue Klassen werden für den AST notwendig. Eine Klasse für die Funktionsdefinitionen. In dieser sind alle Informationen für eine Funktion enthalten. Der Name der Funktionen, die Namen der Parameter und den Ausdruck, der den Funktionsrumpf darstellt.

```
1  package v5.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class FunDef implements Tree{
5    public String name;
6    public List<String> args;
7    public Tree def;
8    public FunDef(String name, List<String> args, Tree def){
9      this.name = name;
10     this.args = args;
11     this.def = def;
12   }
13
14   public String toString(){return "FunDef("+name+", "+args+", "+def+")
       ";}
15
16   public <R> R welcome(Visitor<R> visitor) throws Exception{
17     return visitor.visit(this);
18   }
19 }
```

Listing 6.9: FunDef.java

Ein weiterer neuer Baumknoten stellt Funktionsaufrufe dar. Hier wird der Name der Funktion und die Liste der Argumente in Feldern repräsentiert.

```
1  package v5.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class FunCall implements Tree{
5    public String name;
6    public List<Tree> args;
7
8    public FunCall(String name, List<Tree> args){
9      this.name = name;
10     this.args = args;
11   }
12
13   public String toString(){return "FunCall("+name+", "+args+")";}
14
15   public <R> R welcome(Visitor<R> visitor) throws Exception{
16     return visitor.visit(this);
17   }
```

```
18 ]
```

Listing 6.10: FunCall.java

Ein weiterer Baumknoten steht für Variablen. Hier ist nur noch ein Variablenname zu vermerken.

```
1  package v5.name.panitz.wip.tree;
2
3  public  class Var implements Tree{
4    public String name;
5
6    public Var(String name){
7      this.name = name;
8    }
9
10   public String toString(){return "Var("+name+")";}
11
12   public <R> R welcome(Visitor<R> visitor) throws Exception{
13     return visitor.visit(this);
14   }
15 }
```

Listing 6.11: Var.java

Auch für das gesamte WIP-Programm, dass ja nun nicht mehr ein einzelner Ausdruck ist, wird ein eigener Baumknoten definiert. Dieser enthält die Liste alle Funktionsdefinitionen.

```
1  package v5.name.panitz.wip.tree;
2  import java.util.List;
3
4  public class Program implements Tree{
5    public List<FunDef> funs;
6    public Program(List<FunDef> funs){
7      this.funs = funs;
8    }
9
10   public String toString(){return "Program("+funs+")";}
11
12   public <R> R welcome(Visitor<R> visitor) throws Exception{
13     return visitor.visit(this);
14   }
15 }
```

Listing 6.12: Program.java

Zur Übersicht aller Baumknoten, die wir bisher definiert haben, sei hier die Besucherschnittstelle komplett aufgezeigt.

```
45 package v5.name.panitz.wip.tree;
46 public interface Visitor<R>  {
47   R visit(Var v) throws Exception;
48   R visit(FunCall funcall) throws Exception;
49   R visit(Program prog) throws Exception;
50   R visit(FunDef fun) throws Exception;
51   R visit(IntLit il) throws Exception;
52   R visit(MultExpr ex) throws Exception;
53   R visit(DivExpr ex) throws Exception;
54   R visit(ModExpr ex) throws Exception;
55   R visit(AddExpr ex) throws Exception;
56   R visit(SubExpr ex) throws Exception;
57   R visit(EqExpr ex) throws Exception;
58   R visit(NeqExpr ex) throws Exception;
59   R visit(LeExpr ex) throws Exception;
60   R visit(GeExpr ex) throws Exception;
61   R visit(LtExpr ex) throws Exception;
62   R visit(GtExpr ex) throws Exception;
63   R visit(OrExpr ex) throws Exception;
64   R visit(AndExpr ex) throws Exception;
65   R visit(IfExpr ex) throws Exception;
66 }
```

Listing 6.13: Visitor.java

Wir werden in den folgenden Kapiteln eine ganze Reihe weiterer kleinerer Besucher schreiben. Diese werden oft nicht den kompletten Baum durchlaufen, sondern nur bestimmte Baumknoten besuchen. Um die Implementierungen hier etwas handlicher zu machen, definieren wir einen Besucher, der in jeder Methode eine Ausnahme wirft:

```
1  package v5.name.panitz.wip.tree;
2  import java.util.Map;
3  import java.util.HashMap;
4
5  public class AbstractVisitor<R> implements Visitor<R>{
6    public R visit(Var v) throws Exception{
7      throw new UnsupportedOperationException();
8    }
9    public R visit(FunCall funcall) throws Exception{
10     throw new UnsupportedOperationException();
11   }
12   public R visit(Program prog) throws Exception{
13     throw new UnsupportedOperationException();
14   }
15   public R visit(FunDef fun) throws Exception{
16     throw new UnsupportedOperationException();
17   }
18   public R visit(IntLit il) throws Exception{
19     throw new UnsupportedOperationException();
20   }
```

```
21    public R visit(MultExpr ex) throws Exception{
22      throw new UnsupportedOperationException();
23    }
24    public R visit(DivExpr ex) throws Exception{
25      throw new UnsupportedOperationException();
26    }
27    public R visit(ModExpr ex) throws Exception{
28      throw new UnsupportedOperationException();
29    }
30    public R visit(AddExpr ex) throws Exception{
31      throw new UnsupportedOperationException();
32    }
33    public R visit(SubExpr ex) throws Exception{
34      throw new UnsupportedOperationException();
35    }
36    public R visit(EqExpr ex) throws Exception{
37      throw new UnsupportedOperationException();
38    }
39    public R visit(NeqExpr ex) throws Exception{
40      throw new UnsupportedOperationException();
41    }
42    public R visit(LeExpr ex) throws Exception{
43      throw new UnsupportedOperationException();
44    }
45    public R visit(GeExpr ex) throws Exception{
46      throw new UnsupportedOperationException();
47    }
48    public R visit(LtExpr ex) throws Exception{
49      throw new UnsupportedOperationException();
50    }
51    public R visit(GtExpr ex) throws Exception{
52      throw new UnsupportedOperationException();
53    }
54    public R visit(OrExpr ex) throws Exception{
55      throw new UnsupportedOperationException();
56    }
57    public R visit(AndExpr ex) throws Exception{
58      throw new UnsupportedOperationException();
59    }
60    public R visit(IfExpr ex) throws Exception{
61      throw new UnsupportedOperationException();
62    }
63  }
```

Listing 6.14: AbstractVisitor.java

Ein erster kleiner Besucher, der von diesem abstrakten Besucher ableitet, sammelt in einer Abbildung alle Funktionsdefinitionen eines WIP-Programms. Diese Abbildung dient in weiteren Phasen des Kompilators als schneller Zugriff auf die Funktionsdefinitionen. Der Besucher, der diese Abbildung lauft, geht nur über zwei Arten von Baumknoten und steigt von den Funktionsdefinitionen nicht rekursiv ab.

```
1  package v5.name.panitz.wip.tree;
2  import java.util.Map;
3  import java.util.HashMap;
4
5  public class CollectFunDefs
6                       extends AbstractVisitor<Map<String,FunDef>>{
7
8    Map<String,FunDef> funs = new HashMap<>();
9    public Map<String,FunDef>  visit(Program prog) throws Exception{
10     for (FunDef fun:prog.funs){
11       fun.welcome(this);
12     }
13     return funs;
14   }
15
16   public Map<String,FunDef> visit(FunDef f) throws Exception{
17     funs.put(f.name,f);
18     return funs;
19   }
20 }
```

Listing 6.15: CollectFunDefs.java

6.2.1 Pretty Printer

Mit einer gewissen Routine schreiben wir die Methoden des *pretty printers* für die neuen Baumknoten, so dass ein WIP-Programm wieder textuell ausgegeben werden kann.

```
95   public Integer visit(FunDef fun) throws Exception{
96     w.write("fun ");
97     w.write(fun.name);
98     w.write("(");
99     boolean first = true;
100    for (String arg:fun.args){
101      if (first){first=false;}else{w.write(", ");}
102      w.write(arg);
103    }
104    w.write(") = ");
105    indent+=2;
106    newLine();
107    fun.def.welcome(this);
108    indent-=2;
109    newLine();
110    newLine();
111
112    return indent;
113  }
```

Listing 6.16: PP.java

```
114   public Integer visit(FunCall fun) throws Exception{
115     w.write(fun.name);
116     w.write("(");
117     boolean first = true;
118     for (Tree arg:fun.args){
119       if (first){first=false;}else{w.write(", ");}
120       arg.welcome(this);
121     }
122     w.write(")");
123     return indent;
124   }
```

Listing 6.17: PP.java

```
125   public Integer visit(Var v) throws Exception{
126     w.write(v.name);
127     return indent;
128   }
```

Listing 6.18: PP.java

6.2.2 Interpreter

Unser Besucher, der das WIP Programm direkt interpretiert benötigt gegenüber den vorhergehenden Versionen Umgebungen, in dem die Werte von Variablen gespeichert sind und die Abbildung von Funktionsnamen auf die Funktionsdefinitionen. Hierfür werden zwei Abbildungsobjekte vorgesehen:

```
63 public class Interpreter extends AbstractVisitor<Integer>{
64   int result = 0;
65   Map<String,Integer> varEnv = new HashMap<>();;
66   Map<String,FunDef> funs = new HashMap<>();
```

Listing 6.19: Interpreter.java

Im initiale Zustand gibt es keine Variablen, so dass beide Abbildungen leer sind.

Wird ein Programm besucht, so ist zunächst mit dem zuvor geschriebenen kleinen Besucher, die Abbildung der Funktionsdefinitionen aufzusammeln. Gestartet wird dann der Interpreter mit dem Ausdruck, der die Funktion `go()` aufruft.

```
67   public Integer visit(Program prog) throws Exception{
68     funs = prog.welcome(new CollectFunDefs());
69
70     return new FunCall("go",new LinkedList<>()).welcome(this);
71   }
```

Listing 6.20: Interpreter.java

Zwei neue Baumknoten sind für den Besucher zu berücksichtigen. Zum einen die Variablen. Für diese wird der aktuelle Wert der Variablen in der Umgebung nachgeschlagen.

```
72     public Integer visit(Var v) throws Exception{
73     result = varEnv.get(v.name);
74     return result;
75   }
```

Listing 6.21: Interpreter.java

Der zweite neue Ausdruck, für den unser Besucher zu implementieren ist, sind Funktionsaufrufe. Für diese ist zunächst die Funktionsdefinition in der entsprechenden Abbildung nachzuschlagen.

```
76     public Integer visit(FunCall f) throws Exception{
77     FunDef fun = funs.get(f.name);
```

Listing 6.22: Interpreter.java

Dann wird die aktuelle Umgebung der Variablenbelegung in einer lokalen Variablen abgespeichert und eine neue Umgebung angelegt.

```
78     Map<String,Integer> oldEnv = varEnv;
79     HashMap<String,Integer> newEnv = new HashMap<>();
```

Listing 6.23: Interpreter.java

Nun wird paarweise durch die formale Parameter der Funktionsdefinition und durch die konkreten Argumente des Funktionsaufrufs iteriert. Es wird mit dem Besucher für jedes Argument das Ergebnis errechnet und diesen dann in der neuen Umgebung für den formalen Parameternamen eingetragen

```
80     Iterator<Tree> params = f.args.iterator();
81
82     for (String arg:fun.args){
83       int v = params.next().welcome(this);
84       newEnv.put(arg,v);
85     }
```

Listing 6.24: Interpreter.java

Die so neu erzeugte Umgebung wird genutzt, um den Funktionsrumpf der Funktionsdefinition auszuwerten:

```
86      varEnv = newEnv;
87      result =  fun.def.welcome(this);
```

Listing 6.25: Interpreter.java

Anschließend wird die alte Variablenumgebung wieder hergestellt und schließlich das Ergebnis des Funktionsaufrufs zurück gegeben.

```
88      varEnv = oldEnv;
89      return result;
90    }
```

Listing 6.26: Interpreter.java

6.3 Abstrakte StackMaschine

Um Funktionsdefinition und Aufrufe von Funktionen auf der abstrakten Kellermaschine zu realisieren, benötigen wir fünf neue Befehle in der Maschine.

```
29 package v5.name.panitz.wip.stackmachine;
30
31 public class  Instruction{
32   static public enum Instr
33     { PushInt, Mult, Div, Mod, Add, Sub
34     , Eq, Neq, Lt, Gt, Le, Ge, And, Or
35     , JmpTrue, Jmp, Label
36     , GlobLabel, Ret, Push, StartCall, Call};
```

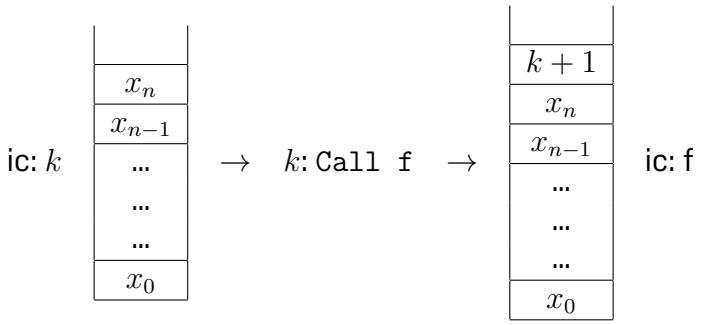
Listing 6.27: Instruction.java

- `GlobLabel`: Ein globales Label, das den Beginn des Programmcodes für eine bestimmte Funktion markiert.
- `Ret` Ein Befehl, der am Ende eines Funktionsausführung dafür sorgt, dass zurück zum Aufruf der Funktion gesprungen wird.
- `StartCall`: Ein Befehl, der dafür sorgt, dass alle notwendigen Operationen zu Beginn eines Funktionsaufrufs vorgenommen werden.
- `Call`: Der Befehl der zum Beginn einer aufzurufenden Funktion springt.
- `Push`: Der Befehl der Elemente die tiefer im Keller liegen zusätzlich ganz oben auf den Keller legen.

Wir werden mit dieser Version der Kellermaschine nicht mehr nur reine Daten auf den Keller legen, mit denen wir rechnen, sondern zusätzlich gesicherte Werte des Programmzählers und den gesicherten Wert des Kellerbasisregisters, das wir bisher noch nicht verwendet haben. Beim Aufruf einer Funktionen werden wir auf den Keller den Wert des Programmzählers legen, bei dem der Aufruf stattfand. Es wird zur Funktion gesprungen, die irgendwann einen Return-Befehl zum Rücksprung an die aufrufende Stelle hat. Dann kann der Programmzähler wieder auf den auf dem Keller gespeicherten Wert gesetzt werden. Jeder Aufruf einer Funktion führt zu einem sogenannten Stackframe, einen neuen Bereich für deie Ausführung der Funktion auf dem Keller. Hier finden sich neben der Rücksprungadresse auch die lokalen Variablen der Funktion. In unserer Sprache WIP sind vorerst als lokale Variablen nur die Argumente der Funktion vorgesehen. Das bisher noch nicht benutzte Register für die Kellerbasis zeigt an, wo der Stackframe für den aktuellen funktionsaufruf beginnt. Von diesem aus, lassen sich die lokalen Variablen des Funktionsaufrufs auf dem Keller lokalisieren.

Wir spezifizieren wieder anhand der Zustandsmaschine die neuen Befehle, um sie dann entsprechend zu implementieren.

Der Befehl `Call` bewirkt zweierlei. Zunächst wird der Programmzähler, der um eins erhöht wurde auf den Keller gelegt. Anschließend wird dieser auf die Adresse des ersten Befehls für die Funktionsdefinition der gerufenen Funktion gesetzt.



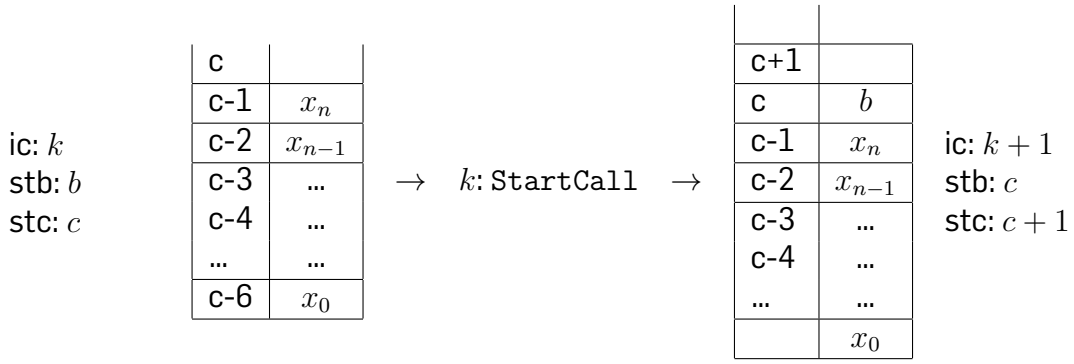
Es sind also nur zwei Operationen. In unserer Maschine hat der Befehl `Call` als Argument den Namen der zu rufenden Funktion. In einer Abbildung merken wir uns jeweils die Adresse im gesamten Programmcode, an der der erste Maschinenbefehl für diese Funktion steht.

```
84      case Call:
85        stack[stc++] = ic+1;
86        ic = labels.get(code[ic].label);
87        break;
```

Listing 6.28: StackMachine.java

Der spezielle Befehl `StartCall` sorgt dafür, dass der Wert der Kellerbasis auf dem Keller abgelegt wird. Anschließend wird die Kellerbasis auf die oberste Kellerposition gesetzt.

In der folgenden Darstellung sind die Kellerzellen durchnummeriert. Die oberste hat die Nummer c.

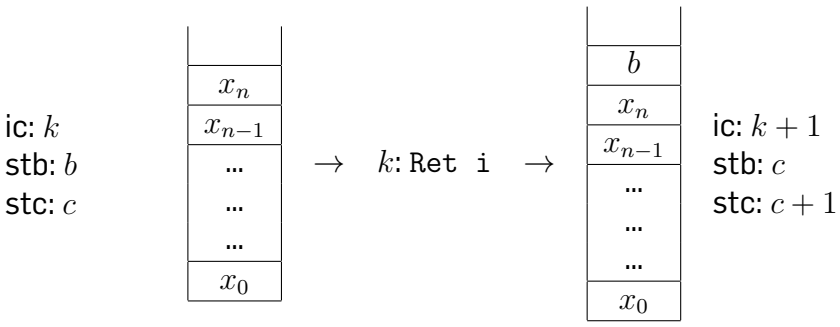


Mit dieser Spezifikation ist die Implementierung einfach umzusetzen.

```
88      case StartCall:
89        stack[stc] = stb;
90        stb = stc;
91        stc++;
92        ic++;
93        break;
```

Listing 6.29: StackMachine.java

Der Befehl `Ret` zum Rücksprung aus einem Funktionsaufruf zu




```
94      case Ret:
95        int args = code[ic].a1;
96        ic = stack[stb-1];
97        stack[stb-1-args] = stack[stc-1];
98        stc = stc-2-args;
99        stb = stack[stb];
100       break;
101     case Push:
102       stack[stc++] = stack[stb-code[ic++].a1-1];
103       break;
```

Listing 6.30: StackMachine.java

6.3.1 Code Generierung

```
95   public List<Instruction> visit(FunDef fun)  throws Exception{
96     result.add(new Instruction(Instruction.Instr.GlobLabel,fun.name));
97     labels.put(fun.name,result.size());
98     result.add(new Instruction(Instruction.Instr.StartCall));
99
100
101    int i = fun.args.size();
102    Map<String,Integer> newVarEnv = new HashMap<>();
103    for (String arg:fun.args){
104      newVarEnv.put(arg,i--);
105    }
106    Map<String,Integer> oldVarEnv = varEnv;
107    varEnv = newVarEnv;
108
109    fun.def.welcome(this);
110
111    varEnv = newVarEnv;
112    result.add(new Instruction(Instruction.Instr.Ret,fun.args.size()))
     ;
113    return result;
114  }
115
116  public List<Instruction> visit(Var v) throws Exception{
117    result.add(new Instruction(Instruction.Instr.Push,varEnv.get(v.
     name)));
118    return result;
119  }
120
121  public List<Instruction> visit(FunCall fun)  throws Exception{
122    for (Tree arg:fun.args){
123      arg.welcome(this);
124    }
125    result.add(new Instruction(Instruction.Instr.Call,fun.name));
126    return result;
127  }
```

Listing 6.31: GenCode.java

6.4 Assembler Generierung

```
162  public void writeAssembler(Writer out,Instruction ins)throws
     Exception{
163    switch (ins.instr) {
```

Listing 6.32: WriteX86.java

```
164     case GlobLabel:
165       out.write(".globl "+ins.label+"\n");
166       out.write(ins.label+":\n");
167       break;
168
169     case JmpTrue:
170       out.write("    "+pop()+"    "+ax()+"\n");
171       out.write("    "+mov()+"    $1, "+bx()+"\n");
172       out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
173       out.write("    je     "+ins.label+"\n");
174       break;
175
176     case Ret:
177       out.write("   "+pop()+"     "+ax()+"\n");
178       out.write("   "+mov()+"     "+bp()+", "+sp()+"\n");
179       out.write("   "+pop()+"     "+bp()+"\n");
180       out.write("    ret    $"+4*(ins.a1)+"\n");
181       break;
182
183     case Push:
184       out.write("    "+push()+"    "+(4*(ins.a1+1))+"("+bp()+")\n");
185       break;
186
187
188     case StartCall:
189       out.write("    "+push()+"    "+bp()+"\n");
190       out.write("    "+mov() +"    "+sp()+", "+bp()+"\n");
191       break;
192
193     case Call:
194       out.write("    call    "+ins.label+"\n");
195       out.write("    "+push()+"    "+ax()+"\n");
196       break;
197     }
198   }
199 }
```

Listing 6.33: WriteX86.java

Weitere mögliche Erweiterungen

6.4.1 Imperative Konstrukte

Wir sind an einem entscheidenden Punkt angelangt. Von nun an ist WIP eine komplette Programmiersprache. Alle mathematisch berechenbaren Funktionen lassen sich nun zumindest theoretisch in WIP programmieren. Jetzt liegt es an uns, wie es mit WIP weitergeht. Im Zuge dieses Kurses entwickelt sich WIP eher zu einer funktionalen Programmiersprache mit automatischer Speicherverwaltung. Das bedeutet insbesondere, dass es keine veränderbaren Variablen geben wird. Jetzt wäre

aber der Punkt erreicht, bei dem ein Zuweisungsbefehl zur Sprache WIP hinzugefügt werden könnte.

6.4.2 Erste semantische Überprüfungen

Kapitel 7

Version 6: Funktionen höherer Ordnung

7.1 Parser

```
276   |<ARROW:"->">
277   |<LAMBDA:   "\\">
```

Listing 7.1: WIPParser.jj

```
278 Tree expression():
279 {Tree result;}
280 {
281   (
282    result=ifExpression()
283   |result=andExpression()
284   |result=lambdaExpression()
285   )
286
287   {return result;}
288 }
289
290 Tree lambdaExpression():
291 {Tree result;
292  List<String> args= new ArrayList<String>();
293  Token tok;
294 }
295 {
296   <LAMBDA>
297     <LPAR> [tok=<IDENT> {args.add(tok.image);}
298        (<COMMA> tok=<IDENT> {args.add(tok.image);})*]
299     <RPAR> <ARROW> result=expression()
300 {return new LambdaExpr(args,result);}
301 }
```

Listing 7.2: WIPParser.jj

7.2 Abstrakter Syntaxbaum

```
1  package v6.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class LambdaExpr implements Tree{
5    public List<String> args;
6    public Tree def;
7    public LambdaExpr(List<String> args, Tree def){
8      this.args = args;
9      this.def = def;
10   }
11
12   public String toString(){return "LambdaExpr("+args+", "+def+")";}
13
14   public <R> R welcome(Visitor<R> visitor) throws Exception{
15     return visitor.visit(this);
16   }
17 }
```

Listing 7.3: LambdaExpr.java

```
67 package v6.name.panitz.wip.tree;
68 public interface Visitor<R>  {
69   R visit(LambdaExpr lambda) throws Exception;
70   R visit(Var v) throws Exception;
71   R visit(FunCall funcall) throws Exception;
72   R visit(Program prog) throws Exception;
73   R visit(FunDef fun) throws Exception;
74   R visit(IntLit il) throws Exception;
75   R visit(MultExpr ex) throws Exception;
76   R visit(DivExpr ex) throws Exception;
77   R visit(ModExpr ex) throws Exception;
78   R visit(AddExpr ex) throws Exception;
79   R visit(SubExpr ex) throws Exception;
80   R visit(EqExpr ex) throws Exception;
81   R visit(NeqExpr ex) throws Exception;
82   R visit(LeExpr ex) throws Exception;
83   R visit(GeExpr ex) throws Exception;
84   R visit(LtExpr ex) throws Exception;
85   R visit(GtExpr ex) throws Exception;
86   R visit(OrExpr ex) throws Exception;
87   R visit(AndExpr ex) throws Exception;
88   R visit(IfExpr ex) throws Exception;
89 }
```

Listing 7.4: Visitor.java

```
129   public Integer visit(LambdaExpr lambda) throws Exception{
130     w.write("\\(");
131     boolean first = true;
132     for (String arg:lambda.args){
133       if (first){first = false;}else{w.write(", ");}
134       w.write(arg);
135     }
136     w.write(") -> ");
137     return lambda.def.welcome(this);
138   }
```

Listing 7.5: PP.java

```
64 package v6.name.panitz.wip.tree;
65 import java.util.Map;
66 import java.util.HashMap;
67
68 public class AbstractVisitor<R> implements Visitor<R>{
69   public R visit(LambdaExpr lambda) throws Exception{
70     throw new UnsupportedOperationException();
71   }
```

Listing 7.6: AbstractVisitor.java

```
21 package v6.name.panitz.wip.tree;
22 import java.util.Map;
23 import java.util.HashMap;
24
25 public class CollectFunDefs extends AbstractVisitor<Map<String,FunDef
     >>{
26   Map<String,FunDef> funs = new HashMap<>();
27   public Map<String,FunDef>  visit(Program prog) throws Exception{
28     for (FunDef fun:prog.funs){
29       fun.welcome(this);
30     }
31     return funs;
32   }
33
34   public Map<String,FunDef> visit(FunDef f) throws Exception{
35     funs.put(f.name,f);
36     return funs;
37   }
38 }
```

Listing 7.7: CollectFunDefs.java

```
1 package v6.name.panitz.wip.tree;
2 public interface Data[]
```

Listing 7.8: Data.java

```
1  package v6.name.panitz.wip.tree;
2  public class FunName implements Data{
3    public String name;
4    public FunName(String name){this.name=name;}
5    public String toString(){return name;}
6  }
```

Listing 7.9: FunName.java

```
1  package v6.name.panitz.wip.tree;
2  public class IntVal implements Data{
3    public int i;
4    public IntVal(int i){this.i=i;}
5    public String toString(){return i+"";}
6  }
```

Listing 7.10: IntVal.java

```
91  package v6.name.panitz.wip.tree;
92
93  import java.util.List;
94  import java.util.LinkedList;
95  import java.util.Map;
96  import java.util.HashMap;
97  import java.util.Iterator;
98  import java.io.*;
99
100 public class Interpreter extends AbstractVisitor<Data>{
101   Data result = new IntVal(0);
102   Map<String,Data> varEnv = new HashMap<>();
103   Map<String,FunDef> funs = new HashMap<>();
104
105   public Data visit(Program prog) throws Exception{
106     funs = prog.welcome(new CollectFunDefs());
107
108     return new FunCall("go",new LinkedList<>()).welcome(this);
109   }
110
111   public Data visit(Var v) throws Exception{
112     result = varEnv.get(v.name);
113     if (result==null){
114       result = new FunName(v.name);
115     }
116     return result;
117   }
118
119   public Data visit(FunCall f) throws Exception{
120     FunDef fun = funs.get(f.name);
121     if (fun==null){
122       Data funvar = varEnv.get(f.name);
```

```
123       if (funvar!=null && funvar instanceof FunName){
124         fun = funs.get(((FunName)funvar).name);
125       }
126     }
127     Map<String,Data> oldEnv = varEnv;
128     HashMap<String,Data> newEnv = new HashMap<>();
129     Iterator<Tree> params = f.args.iterator();
130
131     for (String arg:fun.args){
132       Data v = params.next().welcome(this);
133       System.out.println(arg+"="+v);
134       newEnv.put(arg,v);
135     }
136     varEnv = newEnv;
137     result =  fun.def.welcome(this);
138     varEnv = oldEnv;
139     return result;
140   }
```

Listing 7.11: Interpreter.java

```
1  package v6.name.panitz.wip.tree;
2  import java.util.function.Function;
3  public interface ExceptionFunction<A,R>{
4    R apply(A a)throws Exception;
5    static <B,C> Function<B,C> tryCatch(ExceptionFunction<B,C> fe){
6      return (x)->{try{return fe.apply(x);}catch(Exception e){throw new
       RuntimeException(e);}};
7    }
8  }
```

Listing 7.12: ExceptionFunction.java

```
1  package v6.name.panitz.wip.tree;
2
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.Map;
6  import java.util.HashMap;
7  import java.util.Iterator;
8  import java.io.*;
9  import java.util.stream.Collectors;
10 import java.util.function.Function;
11 import static v6.name.panitz.wip.tree.ExceptionFunction.tryCatch;
12
13 public class GlobalizeLambda extends AbstractVisitor<Tree>{
14   List<FunDef> lambdaFuns = new LinkedList<>();
15
16   public Tree visit(Program prog) throws Exception{
17     Program p = new Program(prog.funs.stream().map(tryCatch((x)->(
     FunDef)x.welcome(this))).collect(Collectors.toList()));
```

```
18      p.funs.addAll(lambdaFuns);
19      return p;
20    }
21
22    public Tree visit(Var v) throws Exception{
23       return v;
24    }
25
26
27
28
29    public Tree visit(FunCall f) throws Exception{
30      return new FunCall(f.name, f.args.stream().map(tryCatch((x)->x.
       welcome(this))).collect(Collectors.toList()) );
31    }
32
33    public Tree visit(FunDef f) throws Exception{
34      return new FunDef(f.name, f.args, f.def.welcome(this));
35    }
36
37
38
39    public Tree visit(IntLit il) throws Exception{
40      return il;
41    }
42
43    public Tree visit(IfExpr ex) throws Exception{
44      return new IfExpr(ex.cond.welcome(this),ex.alt1.welcome(this),ex.
       alt2.welcome(this));
45    }
46
47    public Tree visit(MultExpr ex) throws Exception{
48      return new MultExpr(ex.left.welcome(this),ex.right.welcome(this));
49    }
50    public Tree visit(DivExpr ex) throws Exception{
51      return new DivExpr(ex.left.welcome(this),ex.right.welcome(this));
52    }
53    public Tree  visit(ModExpr ex) throws Exception{
54      return new ModExpr(ex.left.welcome(this),ex.right.welcome(this));
55    }
56    public Tree  visit(AddExpr ex) throws Exception{
57      return new AddExpr(ex.left.welcome(this),ex.right.welcome(this));
58    }
59    public Tree  visit(SubExpr ex) throws Exception{
60      return new SubExpr(ex.left.welcome(this),ex.right.welcome(this));
61    }
62    public Tree  visit(EqExpr ex) throws Exception{
63      return new EqExpr(ex.left.welcome(this),ex.right.welcome(this));
64    }
65    public Tree  visit(NeqExpr ex) throws Exception{
66      return new NeqExpr(ex.left.welcome(this),ex.right.welcome(this));
67    }
```

```
68    public Tree  visit(LeExpr ex) throws Exception{
69      return new LeExpr(ex.left.welcome(this),ex.right.welcome(this));
70    }
71    public Tree  visit(GeExpr ex) throws Exception{
72      return new GeExpr(ex.left.welcome(this),ex.right.welcome(this));
73    }
74    public Tree  visit(LtExpr ex) throws Exception{
75      return new LtExpr(ex.left.welcome(this),ex.right.welcome(this));
76    }
77    public Tree  visit(GtExpr ex) throws Exception{
78      return new GtExpr(ex.left.welcome(this),ex.right.welcome(this));
79    }
80    public Tree  visit(OrExpr ex) throws Exception{
81      return new OrExpr(ex.left.welcome(this),ex.right.welcome(this));
82    }
83    public Tree  visit(AndExpr ex) throws Exception{
84      return new AndExpr(ex.left.welcome(this),ex.right.welcome(this));
85    }
86
87    int nr = 0;
88    String lambdaFun(){
89      return "lambda"+(nr++);
90    }
91
92    public Tree visit(LambdaExpr lambda) throws Exception{
93      String lambdaName = lambdaFun();
94      FunDef f = new FunDef(lambdaName,lambda.args,lambda.def.welcome(
       this));
95      lambdaFuns.add(f);
96      return new Var(lambdaName);
97    }
98
99  }
```

Listing 7.13: GlobalizeLambda.java

7.3 Abstrakte StackMaschine

```
37  package v6.name.panitz.wip.stackmachine;
38
39  public class  Instruction{
40    static public enum Instr
41       { PushInt, Mult, Div, Mod, Add, Sub, Eq, Neq, Lt, Gt, Le, Ge, And
       , Or
42       , JmpTrue, Jmp, Label, GlobLabel, Ret, Push, StartCall, Call
43       , PushFun, FunPointerCall};
44
45    public Instr instr;
46    public int a1;
```

```
47   public String label;
48
49   public Instruction(Instr instr,int a1){
50     this.a1 = a1;
51     this.instr = instr;
52     this.label = "";
53   }
54
55   public Instruction(Instr instr,String l){
56     this(instr,0);
57     this.label = l;
58   }
59
60   public Instruction(Instr instr){
61     this(instr,0);
62   }
63
64   public String toString(){return instr+" "+a1+" "+label;}
65 }
```

Listing 7.14: Instruction.java

```
104     case PushFun:
105       stack[stc++] = labels.get(instruction.label);
106       ic++;
107       break;
108
109     case FunPointerCall:
110       int nextIc = ic+1;
111       ic = stack[--stc];
112       stack[stc++] = nextIc;
113       break;
```

Listing 7.15: StackMachine.java

```
128   public List<Instruction> visit(Var v) throws Exception{
129     Integer vS = varEnv.get(v.name);
130     if (vS != null){
131       result.add(new Instruction(Instruction.Instr.Push,vS));
132       return result;
133     }
134     result.add(new Instruction(Instruction.Instr.PushFun,v.name));
135     return result;
136   }
137
138
139   public List<Instruction> visit(FunCall fun)  throws Exception{
140     for (Tree arg:fun.args){
141       arg.welcome(this);
142     }
143     Integer vS = varEnv.get(fun.name);
```

```
144     if (vS!=null){
145       result.add(new Instruction(Instruction.Instr.Push,vS));
146       result.add(new Instruction(Instruction.Instr.FunPointerCall));
147     }else{
148       result.add(new Instruction(Instruction.Instr.Call,fun.name));
149     }
150     return result;
151   }
```

Listing 7.16: GenCode.java

7.4 Assembler Generierung

```
200     case PushFun:
201       out.write("    "+push()+"    $"+ins.label+"\n");
202       break;
203     case FunPointerCall:
204       out.write("    "+pop()+"    "+ax()+"\n");
205       out.write("    call    *%eax\n");
206       out.write("    "+push()+"    "+ax()+"\n");
207       break;
208 
209     }
210   }
211 }
```

Listing 7.17: WriteX86.java

Kapitel 8

Version 7: Strukturierte Daten

Wir sind an dem komplexesten Kapitel angelangt. In diesem Kapitel werden die größten Neuentwürfe in unserem Kompilator durchgeführt. Der Kompilator wird nicht nur um weitere Komponenten erweitert, sondern bestehende Komponenten werden komplett neu umgesetzt werden.

Es geht darum, strukturierte Daten in die Sprache mit aufzunehmen, also einzelne Daten beliebig miteinander zu komplexeren Daten zusammenzufassen und aus diesen komplexen Daten wieder Teile zu extrahieren. Die strukturierten Daten werden dabei auf einem Heap gespeichert, eine Komponente, die unsere Sprache bisher noch nicht kannte. Nicht mehr benötigte Daten sollen automatisch zur Laufzeit von dem Heap gelöscht werden.

8.1 Parser

```
302   |<CON:   "con">
303   |<MATCH: "match">
304   |<CASE:  "case">
```

Listing 8.1: WIPParser.jj

```
305 ConDef conDef():
306 {
307   Token tok;
308   String conName;
309   List<String> args = new ArrayList<String>();
310 }
311 {
312   <CON> tok=<IDENT> {conName = tok.image;}
313   <LPAR> [tok=<IDENT>{args.add(tok.image);}
314          (<COMMA> tok=<IDENT> {args.add(tok.image);})*
315   ] <RPAR>
316
317   {return new ConDef(conName,args);}
```

```
318 }
319
320
321 Tree expression():
322 {Tree result;}
323 {
324   (
325    result=ifExpression()
326   |result=andExpression()
327   |result=lambdaExpression()
328   |result=matchExpression()
329   )
330
331   {return result;}
332 }
333
334 Tree matchExpression():
335 {Tree expr = null;
336  Tree cexpr = null;
337  List<String> args= new ArrayList<String>();
338  List<CaseExpr> cases= new ArrayList<CaseExpr>();
339  Token tok;
340  Token conname;
341 }
342 {
343   (<MATCH> expr=expression() <LBRACK>
344   (<CASE> conname=<IDENT> {args=null;}
345         [ <LPAR> {args=new ArrayList<String>();}
346           [tok=<IDENT> {args.add(tok.image);}
347              (<COMMA> tok=<IDENT> {args.add(tok.image);})* ]
348           <RPAR>
349         ] <ARROW> cexpr=expression()
350          {if (args!=null)cases.add(new CaseExpr(conname.image,args,
    cexpr));
351           else cases.add(new CaseExpr(conname.image,cexpr));}  )+)
352   <RBRACK>
353  {return new MatchExpr(expr,cases);}
354 }
```

Listing 8.2: WIPParser.jj

8.2 Testprogramme

```
1 con Nil()
2 con Cons(head, tail)
3
4 fun map(f, xs) =
5   match xs{
6     case Nil() -> Nil()
```

```
    case Cons(y, ys) -> Cons(f(y), map(f, ys))
  ]

fun d(x) = (x * x)

fun go() =  map(d, Cons(1, Cons(2, Cons(3, Nil()))))
```

Listing 8.3: map1.wip

```
con Nil()
con Cons(head, tail)

fun map(f, xs) =
  match xs[
    case Nil() -> Nil()
    case Cons(y, ys) -> Cons(f(y), map(f, ys))
  ]

fun go() = map(\(x)->x*x, Cons(1, Cons(2, Cons(3, Nil()))))
```

Listing 8.4: map2.wip

```
con Nil()
con Cons(head,tail)
con Pair(e1,e2)

fun length(xs)
 = match xs [case Nil() -> 0 case Cons(y,ys)-> 1+length(ys)]

fun last(xs)
 = match xs [
    case Cons(y,ys) -> match ys [
      case Nil() -> y case Cons(z,zs) -> last(ys)
    ]
  ]

fun list()=Cons(17,Cons(4,Cons(42,Nil())))

fun go()=Pair(last(list()),length(list()))
```

Listing 8.5: List1.wip

```
con Nil()
con Cons(head,tail)
con Pair(e1,e2)

fun length(xs)
 = match xs [case Nil() -> 0 case Cons(y,ys)-> 1+length(ys)]
fun last(xs)
 = match xs [case Cons(y,ys) -> match ys [
```

```
9       case Nil() -> y case Cons(z,zs) -> last(ys)]]
10
11  fun list()=Cons(17,Cons(4,Cons(42,Nil())))
12
13  fun fromTo(start,end)
14   = if start>end then Nil() else Cons(start,fromTo(start+1,end))
15
16  fun go()=Pair(last(fromTo(1,10000)))
```

Listing 8.6: List2.wip

8.2.1 Sieb des Eratosthenes in WIP, C, Java und Haskell

Interessant wird auch sein, wie unsere Programmiersprache im Vergleich zu anderen Sprachen abschneidet. Zum einen ist interessant zu sehen, wie das gleiche Programm in verschiedenen Sprachen zu formulieren ist, zum anderen interessiert natürlich auch die Performance, die in unterschiedlichen Sprachen für des gleiche Programm erreicht wird.

Daher implementieren wir das Sieb des Eratosthenes als Testprogramm in WIP und zum Vergleich in C, Java und Haskell.

Zunächst die WIP Version:

```
1   con Nil()
2   con Cons(hd,tl)
3
4   fun last(xs)
5    = match xs [case Cons(y,ys) -> match ys [
6       case Nil() -> y case Cons(z,zs) -> last(ys)]]
7
8   fun fromTo(start,end)
9    = if start>end then Nil() else Cons(start,fromTo(start+1,end))
10
11  fun sieb(n,xs) = match xs [
12    case Cons(y, ys) -> if y%n > 0  then Cons(y,sieb(n,ys)) else sieb(n,
        ys)
13    case Nil() -> Nil()
14  ]
15
16  fun siebMe(xs) = match xs [
17    case Cons(y, ys) -> Cons(y,siebMe(sieb(y,ys)))
18    case Nil() -> Nil()
19  ]
20
21  fun go()=
22   (last(siebMe(fromTo(2,
23   (last(siebMe(fromTo(2,
24   (last(siebMe(fromTo(2,
25   (last(siebMe(fromTo(2,
```

```
(last(siebMe(fromTo(2,
(last(siebMe(fromTo(2,
(last(siebMe(fromTo(2,50000))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Listing 8.7: Era.wip

Die analoge Version des Programms in C ist erwartungsgemäß natürlich etwas länger. Explizit ist die Speicherverwaltung durch Freigabe nicht mehr benötigten Speichers vom Programmierer durchzuführen:

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

struct LiS {
  int hd;
  struct LiS* tl;
  bool isEmpty;
};

typedef struct LiS* Li;

Li Cons(int hd,Li tl){
  Li this = (Li)malloc(sizeof(struct LiS));
  this->hd = hd;
  this->tl = tl;
  this->isEmpty = false;
  return this;
}
Li Nil(){
  Li this = (Li)malloc(sizeof(struct LiS));
  this->isEmpty = true;
  return this;
}

Li fromToStep(int f,int t,int s){
  if (t>=f) return Cons(f,fromToStep(f+s,t,s));
  else return Nil();
}
Li fromTo(int f,int t) {return fromToStep(f,t,1);}

Li sieb(int n,Li xs){
  if (xs->isEmpty){
    free(xs);
    return Nil();
  }else {
    int y = xs->hd;
    Li ys = xs->tl;
    free(xs);

    if (y%n!=0) return Cons(y,sieb(n,ys));
    else return sieb(n,ys);
```

```
  }
}

Li siebMe(Li xs){
  if (xs->isEmpty) {
    free(xs);
    return Nil();
  }else{
    int y = xs->hd;
    Li ys = xs->tl;
    free(xs);
    return Cons(y,siebMe(sieb(y,ys)));
  }
}
int last(Li xs){
  if (xs->tl->isEmpty) {
    int y = xs->hd;
    free(xs);

    return y;
  }
  Li ys = xs->tl;
  free(xs);
  return last(ys);
}

int main(){
  printf("%d\n",
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,
    (last(siebMe(fromTo(2,50000))))))))))))))))))))))))));
  return 0;
}
```

Listing 8.8: Era.c

Syntaktisch recht ähnlich zur Version in C sieht die Version in Java aus. Die Listendatenstruktur ist nun natürlich über Objekte realisiert.

```
package v7.test;
public class Li {
  int hd;
  Li tl;
  boolean isEmpty;

  public Li(int hd, Li tl) {
    super();
    this.hd = hd;
```

```
10      this.tl = tl;
11      isEmpty = false;
12    }
13    public Li() {
14      isEmpty = true;
15    }
16
17    static Li fromToStep(int f,int t,int s) {
18      if (t>=f) {
19        return new Li(f,fromToStep(f+s,t,s));
20      }else{
21        return new Li();
22      }
23    }
24    static Li fromTo(int f,int t) {return fromToStep(f,t,1);}
25
26    static Li sieb(int n,Li xs){
27      if (xs.isEmpty){
28        return new Li();
29       } else {
30          final int hd2 = xs.hd;
31          final Li siebTl = sieb(n,xs.tl);
32          if(hd2%n!=0){
33            xs = null;
34            return new Li(hd2,siebTl);
35        }else return siebTl;
36      }
37    }
38
39    static Li siebMe(Li xs){
40      if (xs.isEmpty) return new Li();
41      else{
42        final int hd2 = xs.hd;
43        final Li siebMeTl = siebMe(sieb(hd2,xs.tl));
44        xs=null;
45        return new Li(hd2,siebMeTl);
46      }
47    }
48
49    static int last(Li xs){
50      if (xs.tl.isEmpty) return xs.hd;
51      return last(xs.tl);
52    }
53
54    public static void main(String[] args) {
55      System.out.println(
56        (last(siebMe(fromTo(2,
57        (last(siebMe(fromTo(2,
58        (last(siebMe(fromTo(2,
59        (last(siebMe(fromTo(2,
60        (last(siebMe(fromTo(2,
61        (last(siebMe(fromTo(2,
```

```
62        (last(siebMe(fromTo(2,
63        (last(siebMe(fromTo(2,50000))))))))))))))))))))))))))))))))
64      );
65    }
66  }
```

Listing 8.9: Li.java

Schließlich noch die kürzeste Version in Haskell.[1] Allerdings muss dazu gesagt werden, dass die Listenimplementierung aus dem standard Prelude von Haskell verwendet und nicht neu implementiert wird.

```
1  module Main where
2
3  sieb [] = []
4  sieb (x:xs) = x:sieb[y|y<-xs, not(y `mod` x == 0)]
5
6  fromTo f t = [f,f+1..t]
7
8  main = print$
9          last$sieb$fromTo 2$
10         last$sieb$fromTo 2$
11         last$sieb$fromTo 2$
12         last$sieb$fromTo 2$
13         last$sieb$fromTo 2$
14         last$sieb$fromTo 2$
15         last$sieb$fromTo 2 50000
```

Listing 8.10: Era.hs

8.3 Abstrakter Syntaxbaum

```
1  package v7.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class CaseExpr implements Tree{
5    public String name;
6    public List<String> args;
7    public Tree def;
8    boolean isVarCase=false;
9
10   public CaseExpr(String name,List<String> args, Tree def){
11     this.name = name;
12     this.args = args;
13     this.def = def;
14   }
```

[1] Die hätte auch noch kürzer ausfallen können.

```
15
16   public CaseExpr(String name, Tree def){
17     this.name = name;
18     this.args = null;
19     this.def = def;
20     isVarCase=true;
21   }
22
23   public String toString(){return "CaseExpr("+name+", "+args+", "+def+
       ")";}
24
25   public <R> R welcome(Visitor<R> visitor) throws Exception{
26     return visitor.visit(this);
27   }
28 }
```

Listing 8.11: CaseExpr.java

```
1  package v7.name.panitz.wip.tree;
2
3  public  class ProjectionExpr implements Tree{
4    public Tree expr;
5    public String name;
6    public ProjectionExpr( Tree expr, String name){
7      this.name = name;
8      this.expr = expr;
9    }
10   public String toString(){return "ProjectionExpr("+expr+"."+name+")"
       ;}
11
12   public <R> R welcome(Visitor<R> visitor) throws Exception{
13     return visitor.visit(this);
14   }
15 }
```

Listing 8.12: ProjectionExpr.java

```
1  package v7.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class MatchExpr implements Tree{
5    public List<CaseExpr> cases;
6    public Tree expr;
7    public MatchExpr(Tree expr,List<CaseExpr> cases ){
8      this.expr = expr;
9      this.cases = cases;
10   }
11
12   public String toString(){return "MatchExpr("+expr+", "+cases+")";}
13
14   public <R> R welcome(Visitor<R> visitor) throws Exception{
```

```
15      return visitor.visit(this);
16    }
17  }
```

Listing 8.13: MatchExpr.java

```
1  package v7.name.panitz.wip.tree;
2  import java.util.List;
3
4  public  class ConDef implements Tree{
5    public String name;
6    public List<String> args;
7    public ConDef(String name, List<String> args){
8      this.name = name;
9      this.args = args;
10   }
11
12   public String toString(){return "ConDef("+name+", "+args+")";}
13
14   public <R> R welcome(Visitor<R> visitor) throws Exception{
15     return visitor.visit(this);
16   }
17 }
```

Listing 8.14: ConDef.java

```
16 package v7.name.panitz.wip.tree;
17 import java.util.List;
18
19 public class Program implements Tree{
20   public List<ConDef> cons;
21   public List<FunDef> funs;
22   public Program(List<ConDef> cons,List<FunDef> funs){
23     this.funs = funs;
24     this.cons = cons;
25   }
26
27   public String toString(){return "Program("+cons+", "+funs+")";}
28
29   public <R> R welcome(Visitor<R> visitor) throws Exception{
30     return visitor.visit(this);
31   }
32 }
```

Listing 8.15: Program.java

```
139   public Integer visit(CaseExpr cas) throws Exception{
140     newLine();
141     w.write("case ");
142     w.write(cas.name);
```

```
143     if (!cas.isVarCase){
144       w.write("(");
145       boolean first = true;
146       for (String arg:cas.args){
147         if (first){first = false;}else{w.write(", ");}
148         w.write(arg);
149       }
150       w.write(")");
151     }
152     w.write(" -> ");
153     return cas.def.welcome(this);
154   }
155
156
157   public Integer visit(ProjectionExpr proj) throws Exception{
158     proj.expr.welcome(this);
159     w.write(".");
160     w.write(proj.name);
161     return indent;
162   }
163
164
165   public Integer visit(MatchExpr match) throws Exception{
166     w.write("match ");
167     match.expr.welcome(this);
168     w.write("[");
169     indent+=2;
170
171
172     for (CaseExpr cas:match.cases){
173       cas.welcome(this);
174     }
175     indent-=2;
176     newLine();
177     w.write("]");
178     return indent;
179   }
```

Listing 8.16: PP.java

```
1  package v7.name.panitz.wip.tree;
2  import java.util.Map;
3  import java.util.HashMap;
4
5  public class CollectConDefs extends AbstractVisitor<Map<String,ConDef
       >>{
6    Map<String,ConDef> cons = new HashMap<>();
7    public Map<String,ConDef>  visit(Program prog) throws Exception{
8      for (ConDef con:prog.cons){
9        con.welcome(this);
10     }
11     return cons;
```

```
12    }
13
14    public Map<String,ConDef> visit(ConDef f) throws Exception{
15      cons.put(f.name,f);
16      return cons;
17    }
18  }
```

Listing 8.17: CollectConDefs.java

```
3  package v7.name.panitz.wip.tree;
4  public interface Data{}
```

Listing 8.18: Data.java

```
7  package v7.name.panitz.wip.tree;
8  public class FunName implements Data{
9    public String name;
10   public FunName(String name){this.name=name;}
11   public String toString(){return name;}
12 }
```

Listing 8.19: FunName.java

```
7  package v7.name.panitz.wip.tree;
8  public class IntVal implements Data{
9    public int i;
10   public IntVal(int i){this.i=i;}
11   public String toString(){return i+"";}
12 }
```

Listing 8.20: IntVal.java

```
1  package v7.name.panitz.wip.tree;
2  import java.util.List;
3
4  public class ConData implements Data{
5    public String name;
6    public List<Data> args;
7    public ConData(String name, List<Data> args){
8      this.name=name;
9      this.args = args;
10   }
11
12   public String toString(){
13     StringBuffer result = new StringBuffer(name);
14     result.append("(");
15     boolean first = true;
16     for (Data arg:args){
```

```
17        if (first){first = false;}else{result.append(", ");}
18        result.append(arg.toString());
19      }
20      result.append(")");
21      return result+"";
22    }
23  }
```

Listing 8.21: ConData.java

```
141 package v7.name.panitz.wip.tree;
142
143 import java.util.List;
144 import java.util.LinkedList;
145 import java.util.Map;
146 import java.util.HashMap;
147 import java.util.Iterator;
148 import java.io.*;
149 import java.util.stream.Collectors;
150 import java.util.function.Function;
151 import static v6.name.panitz.wip.tree.ExceptionFunction.*;
152
153 public class Interpreter extends AbstractVisitor<Data>{
154   Data result = new IntVal(0);
155   Map<String,Data> varEnv = new HashMap<>();
156   Map<String,FunDef> funs = new HashMap<>();
157   Map<String,ConDef> cons = new HashMap<>();
158
159   public Data visit(Program prog) throws Exception{
160     funs = prog.welcome(new CollectFunDefs());
161     cons = prog.welcome(new CollectConDefs());
162
163     return new FunCall("go",new LinkedList<>()).welcome(this);
164   }
165
166   public Data visit(Var v) throws Exception{
167     result = varEnv.get(v.name);
168     if (result==null){
169       result = new FunName(v.name);
170     }
171     return result;
172   }
173
174   public Data visit(ProjectionExpr proj) throws Exception{
175     Data matchExpr = proj.expr.welcome(this);
176     if (!(matchExpr instanceof ConData)) throw new Exception("match
      data no constructor node: "+matchExpr);
177
178     ConData expr = (ConData) matchExpr;
179
180     ConDef con = cons.get(expr.name);
```

```
181     if (con==null) throw new Exception("unknown constructor node: "+
      matchExpr);
182     int i=0;
183     for (String arg:con.args){
184       if(arg.equals(proj.name)) break;
185       i++;
186     }
187     return expr.args.get(i);
188   }
189
190   public Data visit(MatchExpr match) throws Exception{
191     Data matchExpr = match.expr.welcome(this);
192     if (!(matchExpr instanceof ConData)) throw new Exception("match
      data no constructor node: "+matchExpr);
193
194     ConData expr = (ConData) matchExpr;
195     for (CaseExpr cas:match.cases){
196       if (cas.isVarCase){
197         Map<String,Data> oldEnv = varEnv;
198         HashMap<String,Data> newEnv = new HashMap<>();
199   newEnv.putAll(oldEnv);
200         newEnv.put(cas.name,expr);
201         varEnv = newEnv;
202         Data result =  cas.def.welcome(this);
203         varEnv = oldEnv;
204         return result;
205       }else if (cas.name.equals(expr.name)){
206         Map<String,Data> oldEnv = varEnv;
207         HashMap<String,Data> newEnv = new HashMap<>();
208   newEnv.putAll(oldEnv);
209
210         Iterator<Data> params = expr.args.iterator();
211
212         for (String arg:cas.args){
213           Data v = params.next();
214           //System.out.println(arg+"="+v);
215           newEnv.put(arg,v);
216         }
217         varEnv = newEnv;
218         Data result =  cas.def.welcome(this);
219         varEnv = oldEnv;
220         return result;
221       }
222     }
223     throw new Exception("no case matches: "+expr);
224
225   }
226
227   public Data visit(FunCall f) throws Exception{
228     System.out.println("Calling: "+f.name);
229
230     ConDef con = cons.get(f.name);
```

```
231       if (con!=null){
232         return new ConData(f.name, f.args.stream().map(tryCatch((x)->x.
      welcome(this))).collect(Collectors.toList()));
233       }
234
235       FunDef fun = funs.get(f.name);
236       if (fun==null){
237         Data funvar = varEnv.get(f.name);
238         if (funvar!=null && funvar instanceof FunName){
239           fun = funs.get(((FunName)funvar).name);
240         }
241       }
242       Map<String,Data> oldEnv = varEnv;
243       HashMap<String,Data> newEnv = new HashMap<>();
244       Iterator<Tree> params = f.args.iterator();
245
246       for (String arg:fun.args){
247         Data v = params.next().welcome(this);
248         //System.out.println(arg+"="+v);
249         newEnv.put(arg,v);
250       }
251       varEnv = newEnv;
252       result =  fun.def.welcome(this);
253       varEnv = oldEnv;
254       return result;
255     }
256
257     public Data visit(IntLit il) throws Exception{
258       return new IntVal(il.i);
259     }
260
261     public Data visit(IfExpr ex) throws Exception{
262       Data p = ex.cond.welcome(this);
263       if (((IntVal)p).i == 1) return ex.alt1.welcome(this);
264       return ex.alt2.welcome(this);
265     }
266
267     public Data visit(OpExpr ex) throws Exception{
268       IntVal op1 = (IntVal)ex.left.welcome(this);
269       IntVal op2 = (IntVal)ex.right.welcome(this);
270       return new IntVal(ex.op.applyAsInt(op1.i, op2.i));
271     }
272     public Data visit(MultExpr ex) throws Exception{
273       return visit((OpExpr)ex);
274     }
275     public Data visit(DivExpr ex) throws Exception{
276       return visit((OpExpr)ex);
277     }
278     public Data  visit(ModExpr ex) throws Exception{
279       return visit((OpExpr)ex);
280     }
281     public Data  visit(AddExpr ex) throws Exception{
```

```
282     return visit((OpExpr)ex);
283   }
284   public Data  visit(SubExpr ex) throws Exception{
285     return visit((OpExpr)ex);
286   }
287   public Data  visit(EqExpr ex) throws Exception{
288     return visit((OpExpr)ex);
289   }
290   public Data  visit(NeqExpr ex) throws Exception{
291     return visit((OpExpr)ex);
292   }
293   public Data  visit(LeExpr ex) throws Exception{
294     return visit((OpExpr)ex);
295   }
296   public Data  visit(GeExpr ex) throws Exception{
297     return visit((OpExpr)ex);
298   }
299   public Data  visit(LtExpr ex) throws Exception{
300     return visit((OpExpr)ex);
301   }
302   public Data  visit(GtExpr ex) throws Exception{
303     return visit((OpExpr)ex);
304   }
305   public Data  visit(OrExpr ex) throws Exception{
306     return visit((OpExpr)ex);
307   }
308   public Data  visit(AndExpr ex) throws Exception{
309     return visit((OpExpr)ex);
310   }
311 }
```

Listing 8.22: Interpreter.java

```
9  package v7.name.panitz.wip.tree;
10 import java.util.function.Function;
11 public  interface ExceptionFunction<A,R>{
12   R apply(A a)throws Exception;
13   static <B,C> Function<B,C> tryCatch(ExceptionFunction<B,C> fe){
14     return (x)->{try{return fe.apply(x);}catch(Exception e){throw new
       RuntimeException(e);}};
15   }
16 }
```

Listing 8.23: ExceptionFunction.java

```
1   public Tree visit(Program prog) throws Exception{
2     Program p
3       = new Program
4           (prog.cons
5           ,prog.funs.stream()
6                     .map(tryCatch((x)->(FunDef)x.welcome(this)))
```

```
                    .collect(Collectors.toList())
          );
    return p;
  }

  public Tree visit(ProjectionExpr proj) throws Exception{
    return new ProjectionExpr(proj.expr.welcome(this),proj.name);
  }

  public Tree visit(MatchExpr match) throws Exception{
    return new MatchExpr
     (match.expr.welcome(this)
     ,match.cases.stream()
                 .map(tryCatch((x)->(CaseExpr)x.welcome(this)))
                 .collect(Collectors.toList())
     );
  }

  public Tree visit(CaseExpr cas) throws Exception{
    CaseExpr result =  new CaseExpr(cas.name,cas.args,cas.def.welcome(
    this));
    result.isVarCase = cas.isVarCase;
    return result;
  }
```

Listing 8.24: Copy.java

```
package v7.name.panitz.wip.tree;

import java.util.List;
import java.util.LinkedList;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;
import java.io.*;
import java.util.stream.Collectors;
import static v7.name.panitz.wip.tree.ExceptionFunction.tryCatch;

public class GlobalizeCases extends Copy{
  List<FunDef> caseFuns = new LinkedList<>();
  Set<String> funNames=new HashSet<>();
  Set<String> conNames=new HashSet<>();

  public Tree visit(Program prog) throws Exception{
    for (FunDef fun:prog.funs) funNames.add(fun.name);
    for (ConDef con:prog.cons) conNames.add(con.name);

    Program p = (Program)super.visit(prog);
    p.funs.addAll(caseFuns);
    return p;
```

```
26    ]
27
28    public Tree visit(CaseExpr cas) throws Exception{
29      Set<String> boundVars = new HashSet<>();
30      if (cas.isVarCase)boundVars.add(cas.name); else boundVars.addAll(
       cas.args);
31      Set<String> freeVars = cas.def.welcome(new CollectFreeVars(
       conNames,funNames,boundVars));
32      String caseName = caseFun();
33
34      List<String> funArgs = new LinkedList<>();
35
36      if (cas.isVarCase){
37        funArgs.add(cas.name);
38      }else{
39        funArgs.add("this");
40        funArgs.addAll(cas.args);
41      }
42      funArgs.addAll(freeVars);
43
44      FunDef f = new FunDef(caseName,funArgs,cas.def.welcome(this));
45      f.isSingleCalled = true;
46
47      caseFuns.add(f);
48      funNames.add(caseName);
49      CaseExpr result = new CaseExpr
50        (cas.name,cas.args
51        ,new FunCall
52          (caseName
53          ,funArgs.stream()
54                  .map(x->new Var(x)).collect(Collectors.toList())));
55
56      result.isVarCase = cas.isVarCase;
57      return result;
58    }
59
60    int nr = 0;
61    synchronized String caseFun(){
62      return "case"+(nr++);
63    }
64  }
```

Listing 8.25: GlobalizeCases.java

```
1  package v7.name.panitz.wip.tree;
2  import java.util.Set;
3  import java.util.HashSet;
4
5  public class CollectFreeVars extends AbstractVisitor<Set<String>>{
6    Set<String> boundVars;
7    Set<String> freeVars = new HashSet<String>();
8
```

```
9   Set<String> funNames;
10  Set<String> conNames;
11
12  public CollectFreeVars(Set<String> conNames,Set<String> funNames,Set
     <String> boundVars){
13    this.boundVars = boundVars;
14    this.funNames = funNames;
15    this.conNames = conNames;
16  }
17
18  public Set<String> visit(CaseExpr cas) throws Exception{
19    Set<String> newBound=new HashSet<>();
20    if (!cas.isVarCase){
21      for (String arg:cas.args){
22        if (!boundVars.contains(arg)) newBound.add(arg);
23      }
24    }else if (!boundVars.contains(cas.name)){
25      newBound.add(cas.name);
26    }
27    boundVars.addAll(newBound);
28    cas.def.welcome(this);
29    boundVars.removeAll(newBound);
30    return freeVars;
31  }
32
33  public Set<String> visit(ProjectionExpr match) throws Exception{
34    match.expr.welcome(this);
35    return freeVars;
36  }
37
38  public Set<String> visit(MatchExpr match) throws Exception{
39    match.expr.welcome(this);
40    for (CaseExpr cas:match.cases){
41      cas.welcome(this);
42    }
43    return freeVars;
44  }
45  public Set<String> visit(LambdaExpr lambda) throws Exception{
46    Set<String> newBound=new HashSet<>();
47    for (String arg:lambda.args){
48      if (!boundVars.contains(arg)) newBound.add(arg);
49    }
50    boundVars.addAll(newBound);
51    lambda.def.welcome(this);
52    boundVars.removeAll(newBound);
53    return freeVars;
54  }
55
56  public Set<String> visit(Var v) throws Exception{
57    if(!boundVars.contains(v.name))
58      freeVars.add(v.name);
59    return freeVars;
```

```
60    }
61    public Set<String> visit(FunCall funCall) throws Exception{
62      if (!conNames.contains(funCall.name)&&!funNames.contains(funCall.
       name) && !boundVars.contains(funCall.name))
63        freeVars.add(funCall.name);
64
65      for (Tree arg:funCall.args)arg.welcome(this);
66      return freeVars;
67    }
68
69    public Set<String> visit(IntLit il) throws Exception{
70      return freeVars;
71    }
72
73    public Set<String> visit(MultExpr ex) throws Exception{
74      ex.left.welcome(this);
75      ex.right.welcome(this);
76      return freeVars;
77    }
78    public Set<String> visit(DivExpr ex) throws Exception{
79      ex.left.welcome(this);
80      ex.right.welcome(this);
81      return freeVars;
82    }
83    public Set<String> visit(ModExpr ex) throws Exception{
84      ex.left.welcome(this);
85      ex.right.welcome(this);
86      return freeVars;
87    }
88    public Set<String> visit(AddExpr ex) throws Exception{
89      ex.left.welcome(this);
90      ex.right.welcome(this);
91      return freeVars;
92    }
93    public Set<String> visit(SubExpr ex) throws Exception{
94      ex.left.welcome(this);
95      ex.right.welcome(this);
96      return freeVars;
97    }
98    public Set<String> visit(EqExpr ex) throws Exception{
99      ex.left.welcome(this);
100     ex.right.welcome(this);
101     return freeVars;
102   }
103   public Set<String> visit(NeqExpr ex) throws Exception{
104     ex.left.welcome(this);
105     ex.right.welcome(this);
106     return freeVars;
107   }
108   public Set<String> visit(LeExpr ex) throws Exception{
109     ex.left.welcome(this);
110     ex.right.welcome(this);
```

```
111     return freeVars;
112   }
113   public Set<String> visit(GeExpr ex) throws Exception{
114     ex.left.welcome(this);
115     ex.right.welcome(this);
116     return freeVars;
117   }
118   public Set<String> visit(LtExpr ex) throws Exception{
119     ex.left.welcome(this);
120     ex.right.welcome(this);
121     return freeVars;
122   }
123   public Set<String> visit(GtExpr ex) throws Exception{
124     ex.left.welcome(this);
125     ex.right.welcome(this);
126     return freeVars;
127   }
128   public Set<String> visit(OrExpr ex) throws Exception{
129     ex.left.welcome(this);
130     ex.right.welcome(this);
131     return freeVars;
132   }
133   public Set<String> visit(AndExpr ex) throws Exception{
134     ex.left.welcome(this);
135     ex.right.welcome(this);
136     return freeVars;
137   }
138   public Set<String> visit(IfExpr ex) throws Exception{
139     ex.cond.welcome(this);
140     ex.alt1.welcome(this);
141     ex.alt2.welcome(this);
142     return freeVars;
143   }
144 }
```

Listing 8.26: CollectFreeVars.java

8.4 Abstrakte StackMaschine

```
66 package v7.name.panitz.wip.stackmachine;
67
68 public class  Instruction{
69   static public enum Instr
70     { PushInt, Mult, Div, Mod, Add, Sub, Eq, Neq, Lt, Gt, Le, Ge, And,
       Or
71     , JmpTrue, JmpGt, Jmp, Label, GlobLabel, Ret, Push, StartCall,
      Call, PushFun, FunPointerCall
72     , Pack, Unpack, Pop, PopTops, PushTop, Project};
73
```

```
74    public Instr instr;
75    public int a1;
76    public int a2;
77    public String label;
78
79    public Instruction(Instr instr,int a1){
80      this.a1 = a1;
81      this.instr = instr;
82      this.label = "";
83    }
84
85
86    public Instruction(Instr instr,int a1,int a2){
87      this.a1 = a1;
88      this.a2 = a2;
89      this.instr = instr;
90      this.label = "";
91    }
92
93    public Instruction(Instr instr,String l){
94      this(instr,0);
95      this.label = l;
96    }
97
98    public Instruction(Instr instr){
99      this(instr,0);
100   }
101
102   public String toString(){return instr+" "+a1+" "+label;}
103 }
```

Listing 8.27: Instruction.java

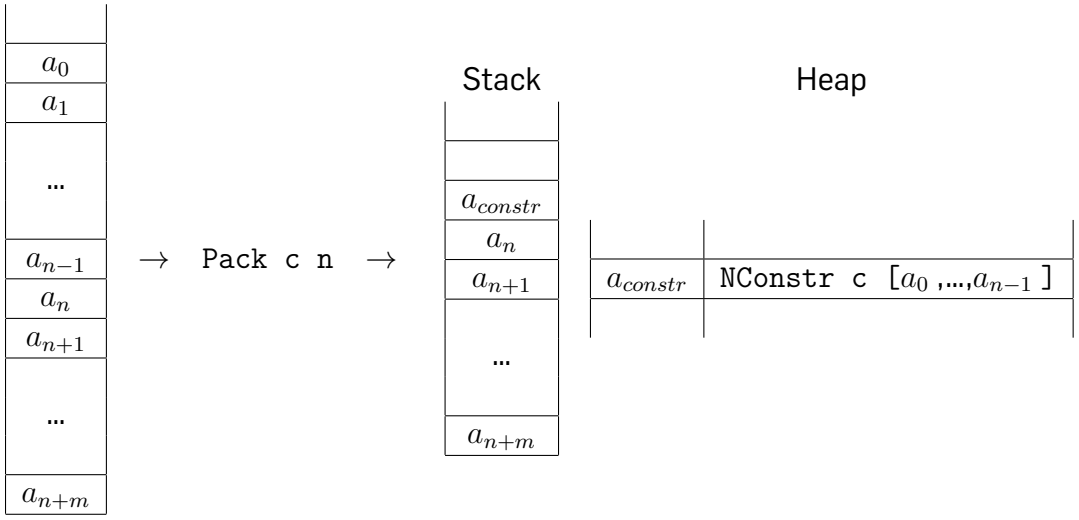
```
1  package v7.name.panitz.wip.stackmachine;
2  public class RefOrInt{
3    boolean isRef;
4    int val;
5    public RefOrInt(boolean isRef,int val){
6      this.isRef = isRef;
7      this.val = val;
8    }
9    public RefOrInt(int val){this(false,val);}
10   public String toString(){
11     return isRef?"(-> "+val+")":(val+"");
12   }
13 }
```

Listing 8.28: RefOrInt.java

```
1  package v7.name.panitz.wip.stackmachine;
2  import java.util.List;
```

```
3  class HeapNode{
4    public int name;
5    public RefOrInt[] args;
6
7    public HeapNode(int name,RefOrInt[] args){
8      this.name = name;
9      this.args = args;
10   }
11   public String toString(){
12     StringBuffer result = new StringBuffer();
13     result.append(name);
14     result.append("[");
15     boolean first = true;
16     for (RefOrInt arg: args){
17       if (first) { first = false;}else{result.append(", ");}
18       result.append(arg+"");
19     }
20     result.append("]");
21     return result.toString();
22   }
23 }
```

Listing 8.29: HeapNode.java




```
114 package v7.name.panitz.wip.stackmachine;
115 import java.util.Map;
116 import java.util.HashMap;
117 import java.io.*;
118
119 public class StackMachine{
120   public Instruction[] code;
121   public int ic = 0;
122
123   RefOrInt[] stack = new RefOrInt[2000];
```

```
124   int stc = 0;
125   int stb = 0;
126
127   HeapNode[] heap = new HeapNode[909];
128   int hc = 0;
129
130   public String[] constructors;
131
132   public Map<String,Integer> labels = new HashMap<>();
133
134   void step(){
135     Instruction instruction = code[ic];
136
137     switch (instruction.instr) {
138     case PushInt: stack[stc++] = new RefOrInt(instruction.a1);ic++;
      break;
139     case Pop: stc--;ic++;break;
140     case PopTops: stc-=stack[--stc].val;ic++;break;
141     case Mult: stack[stc-2] = new RefOrInt(stack[stc-2].val * stack[
      stc-1].val);stc--;ic++;break;
142     case Mod:  stack[stc-2] = new RefOrInt(stack[stc-2].val % stack[
      stc-1].val);stc--;ic++;break;
143     case Div:  stack[stc-2] = new RefOrInt(stack[stc-2].val / stack[
      stc-1].val);stc--;ic++;break;
144     case Add:  stack[stc-2] = new RefOrInt(stack[stc-2].val + stack[
      stc-1].val);stc--;ic++;break;
145     case Sub:  stack[stc-2] = new RefOrInt(stack[stc-2].val - stack[
      stc-1].val);stc--;ic++;break;
146     case Eq:   stack[stc-2] = new RefOrInt(stack[stc-2].val == stack[
      stc-1].val?1:0);stc--;ic++;break;
147     case Neq:  stack[stc-2] = new RefOrInt(stack[stc-2].val != stack[
      stc-1].val?1:0);stc--;ic++;break;
148     case Lt:   stack[stc-2] = new RefOrInt(stack[stc-2].val < stack[
      stc-1].val?1:0);stc--;ic++;break;
149     case Gt:   stack[stc-2] = new RefOrInt(stack[stc-2].val > stack[
      stc-1].val?1:0);stc--;ic++;break;
150     case Le:   stack[stc-2] = new RefOrInt(stack[stc-2].val <= stack[
      stc-1].val?1:0);stc--;ic++;break;
151     case Ge:   stack[stc-2] = new RefOrInt(stack[stc-2].val >= stack[
      stc-1].val?1:0);stc--;ic++;break;
152     case And:  stack[stc-2] = new RefOrInt((stack[stc-2].val==1 && 1==
      stack[stc-1].val)?1:0);stc--;ic++;break;
153     case Or:   stack[stc-2] = new RefOrInt((stack[stc-2].val==1 || 1==
      stack[stc-1].val)?1:0);stc--;ic++;break;
154     case Jmp:  ic = labels.get(instruction.label);break;
155     case JmpTrue:
156         if (stack[stc-1].val==1){
157           ic = labels.get(instruction.label);
158         }else {
159           ic++;
160         }
161         stc--;
```

```
162         break;
163       case Label:  ic++;break;
164       case GlobLabel:  ic++;break;
165       case Ret: {
166         int args = instruction.a1;
167         if (instruction.a2 != 42) ic = stack[stb-1].val;
168         else ic++;
169
170         stack[stb-1-args] = stack[stc-1];
171         stc = stc-2-args;
172         stb = stack[stb].val;
173
174         break;
175       }
176       case Push:
177         stack[stc++] = stack[stb-code[ic++].a1-1];
178         break;
179       case PushTop:
180         stack[stc] = stack[stc-code[ic++].a1-1];
181         stc++;
182         break;
183       case StartCall:
184         if (instruction.a1==42)  stack[stc++] = new RefOrInt(ic+1);
185         stack[stc] = new RefOrInt(stb);
186         stb = stc;
187         stc++;
188         ic++;
189         break;
190       case Call:
191         stack[stc++] = new RefOrInt(ic+1);
192         ic = labels.get(instruction.label);
193         break;
194
195       case PushFun:
196         stack[stc++] = new RefOrInt(labels.get(instruction.label));
197         ic++;
198         break;
199
200       case FunPointerCall:
201         int nextIc = ic+1;
202         ic = stack[--stc].val;
203         stack[stc++] = new RefOrInt(nextIc);
204         break;
205
206       case Pack:{
207         int argsNr = stack[--stc].val;
208         RefOrInt[] args = new RefOrInt[argsNr];
209
210         for (int i= argsNr-1;i>=0;i--){
211           args[i]= stack[--stc];
212         }
213
```

```
214       stack[stc++] = new RefOrInt(true,hc);
215       heap[hc++] = new HeapNode(instruction.a1,args);
216       ic++;
217       break;
218     }
219     case Unpack:
220       int heapNr = stack[--stc].val;
221       HeapNode hp = heap[heapNr];
222       for (RefOrInt arg:hp.args){
223         stack[stc++] = arg;
224       }
225       stack[stc++] = new RefOrInt(hp.name);
226       ic++;
227       break;
228     }
229   }
230
231   String showHeap(){
232     StringBuffer result = new StringBuffer();
233     for (int i=0;i<hc;i++){
234       result.append(i+": "+heap[i]+"\n");
235     }
236     result.append("\n");
237     return result.toString();
238   }
239
240   String showStack(){
241     StringBuffer result = new StringBuffer();
242     for (int i=0;i<stc;i++){
243       result.append(stack[i]+" ");
244     }
245     result.append("\n");
246     result.append("PC: "+ic+" BS: "+stb+" SP:"+stc+"\n");
247     result.append(code[ic]+"\n");
248     return result.toString();
249   }
250
251   public void run() throws IOException{
252     while (ic<code.length){
253     System.out.println(showHeap());
254       System.out.println(showStack());
255       step();
256     }
257     System.out.println(stack[0]);
258     Writer w = new PrintWriter(System.out);
259     printNode(stack[0],w);
260    w.write("\n");
261    w.flush();
262
263
264   }
265
```

```
266   public void printNode(RefOrInt n,Writer w)throws IOException{
267     if (!n.isRef){ w.write(n.toString());}
268     else{
269       w.write(constructors[heap[n.val].name]);
270       w.write("(");
271       boolean first = true;
272       for (RefOrInt arg:heap[n.val].args){
273         if (first) first = false; else w.write(", ");
274         printNode(arg,w);
275       }
276       w.write(")");
277     }
278   }
279
280   public static void main(String[] args)throws IOException{
281     StackMachine interp = new StackMachine();
282     interp.code = new Instruction[1];
283     interp.code[0] = new Instruction(Instruction.Instr.PushInt,42);
284     interp.run();
285   }
286
287   public String toString(){
288     StringBuffer result = new StringBuffer();
289     int i = 0;
290     for (Instruction x:code){
291       result.append((i++)+" "+x+"\n");
292     }
293     result.append("PC:  "+ic);
294
295     return result.toString();
296   }
297 }
```

Listing 8.30: StackMachine.java

```
152 package v7.name.panitz.wip.tree;
153 import v7.name.panitz.wip.stackmachine.*;
154
155 import java.util.List;
156 import java.util.LinkedList;
157 import java.util.Map;
158 import java.util.HashMap;
159
160 public class GenCode extends AbstractVisitor<List<Instruction>>{
161   List<Instruction> result = new LinkedList<>();
162
163   public Map<String,Integer> labels = new HashMap<>();
164   Map<String,Integer> varEnv = new HashMap<>();
165   Map<String,FunDef> funs = new HashMap<>();
166   Map<String,ConDef> cons = new HashMap<>();
167   String[] conNames;
168   Map<String,Integer> conNumbers = new HashMap<>();
```

```
169
170   int start=0;
171
172   public StackMachine getStackMachine(){
173     StackMachine res = new StackMachine();
174     List<Instruction> instr= new LinkedList<>();
175     instr.addAll(result);
176     instr.add(new Instruction(Instruction.Instr.Call,"go"));
177     res.code = instr.toArray(new Instruction[0]);
178     res.labels = labels;
179     res.ic = result.size();
180     res.constructors = conNames;
181     return res;
182   }
183
184   static int lbl = 0;
185   public static String label() {
186     lbl++;
187     return "label"+lbl;
188   }
189
190
191   public List<Instruction> visit(Program prog) throws Exception{
192     funs = prog.welcome(new CollectFunDefs());
193     cons = prog.welcome(new CollectConDefs());
194     conNames = new String[prog.cons.size()];
195     int i=0;
196     for (ConDef con:prog.cons){
197       conNames[i]=con.name;
198       conNumbers.put(con.name,i);
199       i++;
200     }
201
202     for (FunDef fun:prog.funs){
203       fun.welcome(this);
204     }
205     start = result.size();
206     return result;
207   }
208
209
210   public List<Instruction> visit(ProjectionExpr proj)  throws
      Exception{
211     proj.expr.welcome(this);
212
213     int i=0;
214
215     breakLabel: for (ConDef con:cons.values()){
216       i=0;
217       for (String arg:con.args){
218         if (arg.equals(proj.name)){
219     i = con.args.size()-i-1;
```

```
220       break breakLabel;
221           ]
222     i++;
223         ]
224       ]
225       result.add(new Instruction(Instruction.Instr.Project,i));
226       return result;
227     }
228
229     public List<Instruction> visit(MatchExpr match)  throws Exception{
230       String endMatch = label();
231       match.expr.welcome(this);
232       result.add(new Instruction(Instruction.Instr.PushTop,0));
233       result.add(new Instruction(Instruction.Instr.Unpack));
234       for (CaseExpr cas:match.cases){
235           if (cas.isVarCase){
236             result.add(new Instruction(Instruction.Instr.Pop));
237             result.add(new Instruction(Instruction.Instr.PopTops));
238             cas.welcome(this);
239             result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
240           }else{
241             String endCase = label();
242             result.add(new Instruction(Instruction.Instr.PushTop,0));
243             result.add(new Instruction(Instruction.Instr.PushInt,
    conNumbers.get(cas.name)));
244             result.add(new Instruction(Instruction.Instr.Neq));
245             result.add(new Instruction(Instruction.Instr.JmpTrue,endCase)
    );
246             result.add(new Instruction(Instruction.Instr.Pop));
247             result.add(new Instruction(Instruction.Instr.Pop));
248             cas.welcome(this);
249             result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
250             result.add(new Instruction(Instruction.Instr.Label,endCase));
251             labels.put(endCase,result.size());
252       }
253       ]
254       result.add(new Instruction(Instruction.Instr.Label,endMatch));
255       labels.put(endMatch,result.size());
256       return result;
257     }
258
259     public List<Instruction> visit(CaseExpr cas)  throws Exception{
260       FunCall theCase = (FunCall) cas.def;
261       if (!cas.isVarCase){
262         for (Tree arg:theCase.args.subList(cas.args.size()+1,theCase.
    args.size())){
263           arg.welcome(this);
264         }
265       } else{
266         for (Tree arg:theCase.args.subList(1,theCase.args.size())){
267           arg.welcome(this);
268         }
```

```
269       }
270       result.add(new Instruction(Instruction.Instr.Jmp,theCase.name));
271
272       result.add(new Instruction(Instruction.Instr.Label,theCase.name+"
      callReturn"));
273       labels.put(theCase.name+"callReturn",result.size());
274       return result;
275     }
276
277     public List<Instruction> visit(FunDef fun)  throws Exception{
278       if(fun.isSingleCalled){
279         result.add(new Instruction(Instruction.Instr.Label,fun.name));
280         labels.put(fun.name,result.size());
281         result.add(new Instruction(Instruction.Instr.StartCall,42));
282       }else{
283         result.add(new Instruction(Instruction.Instr.GlobLabel,fun.name)
      );
284         labels.put(fun.name,result.size());
285         result.add(new Instruction(Instruction.Instr.StartCall));
286       }
287       int i = fun.args.size();
288       Map<String,Integer> newVarEnv = new HashMap<>();
289       for (String arg:fun.args){
290         newVarEnv.put(arg,i--);
291       }
292       Map<String,Integer> oldVarEnv = varEnv;
293       varEnv = newVarEnv;
294
295       fun.def.welcome(this);
296
297       varEnv = newVarEnv;
298
299       if(fun.isSingleCalled){
300          result.add(new Instruction(Instruction.Instr.Ret,fun.args.size
      (),42));
301          result.add(new Instruction(Instruction.Instr.Jmp,fun.name+"
      callReturn"));
302       }else{
303         result.add(new Instruction(Instruction.Instr.Ret,fun.args.size()
      ));
304       }
305       return result;
306     }
307
308     public List<Instruction> visit(Var v) throws Exception{
309       Integer vS = varEnv.get(v.name);
310       if (vS != null){
311         result.add(new Instruction(Instruction.Instr.Push,vS));
312         return result;
313       }
314       result.add(new Instruction(Instruction.Instr.PushFun,v.name));
315       return result;
```

```
316   }
317
318   public List<Instruction> visit(FunCall fun)  throws Exception{
319     for (Tree arg:fun.args){
320       arg.welcome(this);
321     }
322     Integer vS = varEnv.get(fun.name);
323     if (vS!=null){
324       result.add(new Instruction(Instruction.Instr.Push,vS));
325       result.add(new Instruction(Instruction.Instr.FunPointerCall));
326     }else {
327       Integer conNr = conNumbers.get(fun.name);
328       if (conNr!=null){
329         result.add(new Instruction(Instruction.Instr.PushInt,fun.args.
      size()));
330         result.add(new Instruction(Instruction.Instr.Pack,conNr,fun.
      args.size()));
331       }else{
332         result.add(new Instruction(Instruction.Instr.Call,fun.name));
333       }
334     }
335     return result;
336   }
337
338   public List<Instruction> visit(IntLit il) {
339     result.add(new Instruction(Instruction.Instr.PushInt,il.i));
340     return result;
341   }
342
343   public List<Instruction> visit(IfExpr ex) throws Exception{
344     if (ex.cond instanceof GtExpr){
345       String thenLabel = label();
346       String endIfLabel = label();
347       visit((OpExpr)ex.cond);
348       result.add(new Instruction(Instruction.Instr.JmpGt,thenLabel));
349       ex.alt2.welcome(this);
350       result.add(new Instruction(Instruction.Instr.Jmp,endIfLabel));
351       labels.put(thenLabel,result.size());
352       result.add(new Instruction(Instruction.Instr.Label,thenLabel));
353       ex.alt1.welcome(this);
354       labels.put(endIfLabel,result.size());
355       result.add(new Instruction(Instruction.Instr.Label,endIfLabel));
356     }else{
357
358       String thenLabel = label();
359       String endIfLabel = label();
360
361       ex.cond.welcome(this);
362       result.add(new Instruction(Instruction.Instr.JmpTrue,thenLabel))
      ;
363       ex.alt2.welcome(this);
364       result.add(new Instruction(Instruction.Instr.Jmp,endIfLabel));
```

```
365         labels.put(thenLabel,result.size());
366         result.add(new Instruction(Instruction.Instr.Label,thenLabel));
367         ex.alt1.welcome(this);
368         labels.put(endIfLabel,result.size());
369         result.add(new Instruction(Instruction.Instr.Label,endIfLabel));
370       }
371       return result;
372     }
373
374     public void visit(OpExpr ex) throws Exception{
375       ex.left.welcome(this);
376       ex.right.welcome(this);
377     }
378     public List<Instruction> visit(MultExpr ex) throws Exception{
379       visit((OpExpr) ex);
380       result.add(new Instruction(Instruction.Instr.Mult));
381       return result;
382     }
383
384     public List<Instruction>  visit(DivExpr ex) throws Exception{
385       visit((OpExpr) ex);
386       result.add(new Instruction(Instruction.Instr.Div));
387       return result;
388     }
389     public List<Instruction>  visit(ModExpr ex) throws Exception{
390       visit((OpExpr) ex);
391       result.add(new Instruction(Instruction.Instr.Mod));
392       return result;
393     }
394     public List<Instruction>  visit(AddExpr ex) throws Exception{
395       visit((OpExpr) ex);
396       result.add(new Instruction(Instruction.Instr.Add));
397       return result;
398     }
399     public List<Instruction>  visit(SubExpr ex) throws Exception{
400       visit((OpExpr) ex);
401       result.add(new Instruction(Instruction.Instr.Sub));
402       return result;
403     }
404     public List<Instruction>  visit(EqExpr ex) throws Exception{
405       visit((OpExpr) ex);
406       result.add(new Instruction(Instruction.Instr.Eq));
407       return result;
408     }
409     public List<Instruction>  visit(NeqExpr ex) throws Exception{
410       visit((OpExpr) ex);
411       result.add(new Instruction(Instruction.Instr.Neq));
412       return result;
413     }
414     public List<Instruction>  visit(LeExpr ex) throws Exception{
415       visit((OpExpr) ex);
416       result.add(new Instruction(Instruction.Instr.Le));
```

```
417     return result;
418   }
419   public List<Instruction>  visit(GeExpr ex) throws Exception{
420     visit((OpExpr) ex);
421     result.add(new Instruction(Instruction.Instr.Ge));
422     return result;
423   }
424   public List<Instruction>  visit(LtExpr ex) throws Exception{
425     visit((OpExpr) ex);
426     result.add(new Instruction(Instruction.Instr.Lt));
427     return result;
428   }
429   public List<Instruction>  visit(GtExpr ex) throws Exception{
430     visit((OpExpr) ex);
431     result.add(new Instruction(Instruction.Instr.Gt));
432     return result;
433   }
434   public List<Instruction>  visit(OrExpr ex) throws Exception{
435     visit((OpExpr) ex);
436     result.add(new Instruction(Instruction.Instr.Or));
437     return result;
438   }
439   public List<Instruction>  visit(AndExpr ex) throws Exception{
440     visit((OpExpr) ex);
441     result.add(new Instruction(Instruction.Instr.And));
442     return result;
443   }
444 }
```

Listing 8.31: GenCode.java

8.5 Runtime

```
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <stdbool.h>
11
12 #include <sys/resource.h>
13
14
15 #define HEAP_SIZE 20*1048576
16 #define MAX_SIZE 14*1048576
17 #define STACK_SIZE 2*1048576
18
19 typedef enum {Ref,Prim} RefPrimFlag;
20
21 typedef struct {
22   RefPrimFlag refprimflag;
23   int refprimval;
```

```
24 } RefPrim;
25
26 void printRefPrim(RefPrim* rp){
27   switch (rp->refprimflag ){
28     case Ref:  printf("(-> %d)",rp->refprimval);break;
29     case Prim: printf(" %d",rp->refprimval);break;
30   }
31 }
32
33 void printStack();
34 void printHeap();
35
36 int callingArg;
37
38
39 typedef struct {
40   int constrNr;
41   int arity;
42   RefPrim* args;
43 } HeapNode;
44
45 void printHeapNode(HeapNode* h){
46   printf("Constr<%d %d>(",h->constrNr,h->arity);
47   int i;
48   for (i=0;i<h->arity;i++){
49     if (i>0) printf(",");
50       printRefPrim(h->args+i);
51   }
52   printf(")");
53 }
54
55
56 HeapNode** heap;
57 HeapNode** gcheap;
58
59 unsigned int hp=0;
60 unsigned int heapSize = 909;
61 unsigned int lastUsedheapSize = 909;;
62
63 int nr;
64 int arity;
65
66 HeapNode* newCell(unsigned int arity,int nr){
67   HeapNode* newNode = (HeapNode*)malloc(sizeof(HeapNode));
68   newNode->constrNr = nr;
69   newNode->arity = arity;
70   return newNode;
71 }
72
73 void gc();
74
75 HeapNode* newHeapCell(){
```

```
76    if (hp>=heapSize-1){
77      gc();
78    }
79
80    RefPrim* args = (RefPrim*)malloc(sizeof(RefPrim)*arity);
81    HeapNode* newNode = (HeapNode*)malloc(sizeof(HeapNode));
82
83    newNode->constrNr = nr;
84    newNode->arity = arity;
85    newNode->args = args;
86
87    heap[hp] = newNode;
88    hp++;
89
90    return newNode;
91  }
92
93
94  RefPrim* stack;
95  RefPrim* baseptr;
96  unsigned int sp = 0;
97  unsigned int sb = 0;
98
99  void unpack(){
100   int heapRef = stack[--sp].refprimval;
101   int nr = heap[heapRef]->constrNr;
102   int i;
103
104   if (sp+heap[heapRef]->arity+1>=STACK_SIZE){
105     printf("stack full\n");
106     exit(-1);
107   }
108
109   for (i=heap[heapRef]->arity-1;i>=0;i--){
110     stack[sp].refprimflag = heap[heapRef]->args[i].refprimflag;
111     stack[sp].refprimval = heap[heapRef]->args[i].refprimval;
112     sp++;
113   }
114   stack[sp].refprimflag = Prim;
115   stack[sp].refprimval = heap[heapRef]->arity;
116   sp++;
117   stack[sp].refprimflag = Prim;
118   stack[sp].refprimval = nr;
119   sp++;
120 }
121
122
123 int heapRef;
124 void project(){
125   heapRef = stack[--sp].refprimval;
126   stack[sp].refprimflag = heap[heapRef]->args[callingArg].refprimflag;
127   stack[sp].refprimval = heap[heapRef]->args[callingArg].refprimval;
```

```
128   sp++;
129 }
130
131
132 void pushPrim(){
133   if (sp+1>STACK_SIZE){
134     printf("stack full\n");
135     exit(-1);
136   }
137   stack[sp].refprimflag=Prim;
138   stack[sp].refprimval=callingArg;
139   sp++;
140 }
141
142 void pushRef(){
143   if (sp+1>STACK_SIZE){
144     printf("stack full\n");
145     exit(-1);
146   }
147   stack[sp].refprimflag=Ref;
148   stack[sp].refprimval=callingArg;
149   sp++;
150 }
151
152 void pop(){
153   sp--;
154 }
155
156 void popTops(){
157   sp = sp - stack[--sp].refprimval-1;
158 }
159
160 void push(){
161   if (sp+1>STACK_SIZE){
162     printf("stack full\n");
163     exit(-1);
164   }
165   stack[sp].refprimflag = stack[sb-callingArg].refprimflag;
166   stack[sp].refprimval  = stack[sb-callingArg].refprimval;
167   sp++;
168 }
169
170
171 void pushtop(){
172   if (sp+1>STACK_SIZE){
173     printf("stack full\n");
174     exit(-1);
175   }
176   stack[sp].refprimflag = stack[sp-callingArg-1].refprimflag;
177   stack[sp].refprimval  = stack[sp-callingArg-1].refprimval;
178   sp++;
179 }
```

```
180
181 HeapNode** allocHeap(unsigned int size) {
182   HeapNode** result = (HeapNode**) malloc(size*sizeof(HeapNode*));
183   if (!result) {
184     printf("could not allocate heap\n");
185     exit(-1);
186   }
187   return result;
188 }
189
190 unsigned int copy(unsigned int heapIndex){
191   HeapNode* oldcell = heap[heapIndex];
192
193   if (oldcell->constrNr == -1){
194     return oldcell->arity;
195   }else{
196     HeapNode* newcell = newCell(oldcell->arity,oldcell->constrNr);
197     newcell->args=oldcell->args;
198     oldcell->constrNr = -1;
199     oldcell->arity = lastUsedheapSize;
200     unsigned int result = lastUsedheapSize;
201     gcheap[lastUsedheapSize] = newcell;
202     lastUsedheapSize++;
203
204     int ar=newcell->arity;
205     int ai;
206     for (ai=0;ai< ar;ai++){
207       newcell->args[ai].refprimflag = oldcell->args[ai].refprimflag;
208       if (oldcell->args[ai].refprimflag == Ref){
209         newcell->args[ai].refprimval = copy(oldcell->args[ai].
    refprimval);
210       }else{
211         newcell->args[ai].refprimval = oldcell->args[ai].refprimval;
212       }
213     }
214
215     return result;
216   }
217 }
218
219 void gc(){
220   printf("Starting gc. old heap: %d",hp-1);
221   int luh3  = lastUsedheapSize*3;
222   int newSize = luh3 > heapSize ? luh3 : heapSize;
223   if (newSize>MAX_SIZE) newSize = MAX_SIZE;
224   printf(" new size: %d\n",newSize);
225   gcheap = allocHeap(newSize);
226   lastUsedheapSize = 0;
227
228   int i;
229   for (i=0;i<sp;i++ ){
230     if ((stack+i)->refprimflag==Ref){
```

```
231       (stack+i)->refprimval = copy((stack+i)->refprimval);
232     }
233   }
234
235   //alten heap löschen
236   //int i;
237   for (i=0;i<hp;i++){
238     if (heap[i]->constrNr!=-1)free(heap[i]->args);
239     free(*(heap+i));
240   }
241   free(heap);
242
243   //neuen heap setzen
244   heap = gcheap;
245   heapSize = newSize;
246   hp = lastUsedheapSize;
247   printf("  Ending gc. new heap: %d\n",hp);
248 }
249
250 void printStack() {
251   int i;
252   for (i = 0; i < sp; i++) {
253     printRefPrim(&stack[i]);
254     printf("\n");
255   }
256 }
257
258 void printHeap(){
259   printf("\nHEAP\n");
260   int i;
261   for (i=0;i<hp;i++){
262     printf("%d:  ",i);
263     printHeapNode(heap[i]);
264     printf("\n");
265   }
266 }
267
268 void ret(){
269   RefPrim result = stack[--sp];
270   sb = stack[--sp].refprimval;
271   sp -= callingArg;
272   stack[sp++] = result;
273 }
274
275
276 int main (int argc, char **argv)
277 {
278   const rlim_t kStackSize = 125 * 1024 * 1024;
279   struct rlimit rl;
280   int result;
281
282   result = getrlimit(RLIMIT_STACK, &rl);
```

```
283   if (result == 0){
284     if (rl.rlim_cur < kStackSize){
285       rl.rlim_cur = kStackSize;
286       result = setrlimit(RLIMIT_STACK, &rl);
287       if (result != 0){
288   fprintf(stderr, "setrlimit returned result = %d\n", result);
289       }
290     }
291   }
292   stack=(RefPrim*)malloc(STACK_SIZE*sizeof(RefPrim));
293   baseptr=stack;
294
295   heap = allocHeap(heapSize * sizeof(HeapNode*));
296   go();
297   gc();
298   printf("fertig\n");
299   printf("\nSTACK\n");
300   printStack();
301   printHeap();
302   return 0;
303 }
```

Listing 8.32: runt.c

```
44 package v7.name.panitz.wip;
45 import v7.name.panitz.wip.tree.*;
46 import v7.name.panitz.wip.stackmachine.*;
47 import java.util.*;
48 import java.io.*;
49
50 public class Main{
51   public static void main(String[] args) throws Exception{
52     WIPParser parser = null;
53     if (args.length == 0){
54       parser = new WIPParser(System.in);
55     }else {
56       parser = new WIPParser(new FileReader(args[0]));
57     }
58     Tree il = parser.prog();
59
60     StringWriter out  = new StringWriter();
61     PP pp = new PP(out);
62     il.welcome(pp);
63     System.out.println(out);
64
65     System.out.println(il.welcome(new Interpreter()));
66
67     GenCode genCode = new GenCode();
68     List<Instruction> code = il.welcome(genCode);
69
70     StackMachine st = genCode.getStackMachine();
71     System.out.println(st);
```

```
72
73 //     st.run();
74
75     WriteX86 asm = new WriteX86(st.code);
76     FileWriter asmf = new FileWriter("test.s");
77     asm.writeAssembler(asmf);
78
79     WriteX86 asm64 = new WriteX86(st.code);
80     asm64._32 = false;
81     FileWriter asmf64 = new FileWriter("test64.s");
82     asm64.writeAssembler(asmf64);
83   }
84 }
```

Listing 8.33: Main.java

8.6 Assembler Generierung

```
212 package v7.name.panitz.wip.stackmachine;
213 import  v7.name.panitz.wip.stackmachine.Instruction.Instr;
214 import java.io.*;
215 import static v7.name.panitz.wip.tree.GenCode.*;
216
217 public class WriteX86{
218   public boolean _32 = true;
219
220   String mov(){return _32?"movl":"movq";}
221   String ax(){return _32?"%eax":"%rax";}
222   String bx(){return _32?"%ebx":"%rbx";}
223   String cx(){return _32?"%ecx":"%rcx";}
224   String dx(){return _32?"%edx":"%rdx";}
225   String sp(){return _32?"%esp":"%rsp";}
226   String bp(){return _32?"%ebp":"%rbp";}
227   String push(){return _32?"pushl":"pushq";}
228   String pop(){return _32?"popl":"popq";}
229   String mul(){return _32?"mull":"mulq";}
230   String add(){return _32?"addl":"addq";}
231   String sub(){return _32?"subl":"subq";}
232   String div(){return _32?"divl":"divq";}
233   String and(){return _32?"andl":"andq";}
234   String or(){return _32?"orl":"orq";}
235
236   String cmp(){return _32?"cmpl":"cmpq";}
237   String jmp(){return "jmp";}
238   String jne(){return "jne";}
239   String jnl(){return "jnl";}
240
241
242   public Instruction[] code;
```

```
243   public WriteX86(Instruction[] code){
244     this.code = code;
245   }
246
247   private void genCompareExpr(Writer out,String jmp)throws Exception{
248     genPop(out,"%ecx");
249     genPop(out,"%edx");
250     out.write("    movl    %edx, %ebx\n");
251     out.write("    movl    %ecx, %eax\n");
252     out.write("   "+cmp()+"   "+ax()+", "+bx()+"\n");
253
254     String endLabel = label();
255     String posLabel = label();
256
257     out.write("   "+jmp+"   "+posLabel+"\n");
258     pushInt(out,0);
259     out.write("   "+jmp()+"   "+endLabel+"\n");
260     out.write("   "+posLabel+":\n");
261     pushInt(out,1);
262     out.write("   "+endLabel+":\n");
263   }
264
265   public void writeAssembler(Writer out)throws Exception{
266     for (Instruction in:code){
267       writeAssembler(out,in);
268     }
269     out.close();
270   }
271
272   public void pushInt(Writer out,int prim)throws Exception{
273     out.write("    movl stack, %eax\n");
274     out.write("    movl sp, %edx\n");
275     out.write("    sall $3, %edx\n");
276     out.write("    addl %edx, %eax\n");
277     out.write("    movl $1, (%eax)\n");
278     out.write("    movl stack, %eax\n");
279     out.write("    movl sp, %edx\n");
280     out.write("    sall $3, %edx\n");
281     out.write("    addl %eax, %edx\n");
282     out.write("    movl $"+prim+", %eax\n");
283     out.write("    movl %eax, 4(%edx)\n");
284     out.write("    movl sp, %eax\n");
285     out.write("    addl $1, %eax\n");
286     out.write("    movl %eax, sp\n");
287   }
288
289   private void pushInt(Writer out,String reg)throws Exception{
290     out.write("    movl "+reg+", %ebx\n");
291     out.write("    movl stack, %eax\n");
292     out.write("    movl sp, %edx\n");
293     out.write("    sall $3, %edx\n");
294     out.write("    addl %edx, %eax\n");
```

```
295     out.write("    movl $1, (%eax)\n");
296     out.write("    movl stack, %eax\n");
297     out.write("    movl sp, %edx\n");
298     out.write("    sall $3, %edx\n");
299     out.write("    addl %eax, %edx\n");
300     out.write("    movl %ebx, %eax\n");
301     out.write("    movl %eax, 4(%edx)\n");
302     out.write("    movl sp, %eax\n");
303     out.write("    addl $1, %eax\n");
304     out.write("    movl %eax, sp\n");
305   ]
306
307   private void genPushRef(Writer out,String reg)throws Exception[
308     out.write("    movl "+reg+", %ebx\n");
309     out.write("    movl stack, %eax\n");
310     out.write("    movl sp, %edx\n");
311     out.write("    sall $3, %edx\n");
312     out.write("    addl %edx, %eax\n");
313     out.write("    movl $0, (%eax)\n");
314     out.write("    movl stack, %eax\n");
315     out.write("    movl sp, %edx\n");
316     out.write("    sall $3, %edx\n");
317     out.write("    addl %eax, %edx\n");
318     out.write("    movl %ebx, %eax\n");
319     out.write("    movl %eax, 4(%edx)\n");
320     out.write("    movl sp, %eax\n");
321     out.write("    addl $1, %eax\n");
322     out.write("    movl %eax, sp\n");
323
324
325   ]
326
327   private void genPop(Writer out)throws Exception[
328     out.write("    movl    sp, %eax\n");
329     out.write("    subl    $1, %eax\n");
330     out.write("    movl    %eax, sp\n");
331   ]
332
333
334   private void genPop(Writer out,String reg)throws Exception[
335     genPop(out);
336     out.write("    movl    sp, %eax  #save pop in reg\n");
337     out.write("    sall    $3, %eax\n");
338     out.write("    addl    stack, %eax\n");
339     out.write("    addl    $4, %eax\n");
340     out.write("    movl    (%eax), "+reg+"\n");
341   ]
342
343   public void writeAssembler(Writer out,Instruction ins)throws
      Exception[
344     switch (ins.instr) {
345     case PushInt: pushInt(out,ins.a1); break;
```

```
346
347     case Mult:
348       genPop(out,"%ecx");
349       genPop(out,"%edx");
350       out.write("     movl    %edx, %ebx\n");
351       out.write("     movl    %ecx, %eax\n");
352       out.write("     mull    %ebx\n");
353       pushInt(out,"%eax");
354       break;
355
356     case Add:
357       genPop(out,"%ecx");
358       genPop(out,"%edx");
359       out.write("     addl    %ecx, %edx\n");
360       pushInt(out,"%edx");
361       break;
362
363     case Sub:
364       genPop(out,"%ecx");
365       genPop(out,"%edx");
366       out.write("     subl    %ecx, %edx\n");
367       pushInt(out,"%edx");
368       break;
369
370     case Div:
371       genPop(out,"%edx");
372       genPop(out,"%ecx");
373       out.write("     movl    %edx, %ebx\n");
374       out.write("     movl    %ecx, %eax\n");
375       out.write("     movl $0, %edx\n");
376       out.write("     idivl %ebx\n");
377       pushInt(out,"%eax");
378       break;
379
380     case Mod:
381       genPop(out,"%edx");
382       genPop(out,"%ecx");
383       out.write("     movl    %edx, %ebx\n");
384       out.write("     movl    %ecx, %eax\n");
385       out.write("     movl $0, %edx\n");
386       out.write("     idivl %ebx\n");
387       pushInt(out,"%edx");
388       break;
389
390     case Eq:
391       genCompareExpr(out,"je");
392       break;
393
394     case Neq:
395       genCompareExpr(out,"jne");
396       break;
397
```

```
398     case Lt:
399       genCompareExpr(out,"jl");
400       break;
401 
402     case Le:
403       genCompareExpr(out,"jle");
404       break;
405 
406     case Gt:
407       genCompareExpr(out,"jg");
408       break;
409 
410     case Ge:
411       genCompareExpr(out,"jge");
412       break;
413 
414     case And:
415       genPop(out,"%ebx");
416       genPop(out,"%eax");
417       out.write("    "+and()+"    "+bx()+", "+ax()+"\n");
418       pushInt(out,"%eax");
419       break;
420 
421     case Or:
422       genPop(out,"%ebx");
423       genPop(out,"%eax");
424       out.write("    "+or()+"    "+bx()+", "+ax()+"\n");
425       pushInt(out,"%eax");
426       break;
427 
428     case Jmp:
429       out.write("    "+jmp()+"    "+ins.label+"\n");
430       break;
431 
432     case Label:
433       out.write(ins.label+":\n");
434       break;
435 
436     case GlobLabel:
437       out.write(".globl "+ins.label+"\n");
438       out.write(ins.label+":\n");
439       break;
440 
441     case JmpTrue:
442       genPop(out,"%eax");
443       out.write("    "+mov()+"    $1, "+bx()+"\n");
444       out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
445       out.write("    je     "+ins.label+"\n");
446       break;
447 
448     case JmpGt:
449       genPop(out,"%ebx");
```

```
450         genPop(out,"%eax");
451 //        out.write("     movl    %edx, %ebx\n");
452 //        out.write("     movl    %ecx, %eax\n");
453         out.write("    "+cmp()+"    "+bx()+", "+ax()+"\n");
454         out.write("    jg     "+ins.label+"\n");
455         break;
456 
457       case Ret:
458         out.write("   "+mov()+"    $"+ins.a1+", callingArg\n");
459         out.write("    call ret\n");
460         if (ins.a2!=42){
461           out.write("    ret    \n");
462         }
463         break;
464 
465       case Push:
466         out.write("   movl $"+(ins.a1+1)+", callingArg\n");
467         out.write("   movl stack, %eax\n");
468         out.write("   movl sp, %edx\n");
469         out.write("   sall $3, %edx\n");
470         out.write("   addl %eax, %edx\n");
471         out.write("   movl stack, %eax\n");
472         out.write("   movl sb, %ebx\n");
473         out.write("   movl callingArg, %ecx\n");
474         out.write("   subl %ecx, %ebx\n");
475         out.write("   movl %ebx, %ecx\n");
476         out.write("   sall $3, %ecx\n");
477         out.write("   addl %ecx, %eax\n");
478         out.write("   movl (%eax), %eax\n");
479         out.write("   movl %eax, (%edx)\n");
480         out.write("   movl stack, %eax\n");
481         out.write("   movl sp, %edx\n");
482         out.write("   sall $3, %edx\n");
483         out.write("   addl %eax, %edx\n");
484         out.write("   movl stack, %eax\n");
485         out.write("   movl sb, %ebx\n");
486         out.write("   movl callingArg, %ecx\n");
487         out.write("   subl %ecx, %ebx\n");
488         out.write("   movl %ebx, %ecx\n");
489         out.write("   sall $3, %ecx\n");
490         out.write("   addl %ecx, %eax\n");
491         out.write("   movl 4(%eax), %eax\n");
492         out.write("   movl %eax, 4(%edx)\n");
493         out.write("   movl sp, %eax\n");
494         out.write("   addl $1, %eax\n");
495         out.write("   movl %eax, sp\n");
496         break;
497 
498       case StartCall:
499         out.write("    "+mov() +"    sb,"+cx()+"\n");
500         pushInt(out,"%ecx");
501         out.write("    "+mov() +"    sp,"+dx()+"\n");
```

```
502       out.write("    "+mov() +"    "+dx()+", sb\n");
503       break;
504
505     case Call:
506       out.write("    call    "+ins.label+"\n");
507       break;
508
509     case Pack:
510       genPop(out,"%ecx");
511       out.write("     movl    %ecx, arity\n");
512       out.write("     movl    $"+ins.a1+", nr\n");
513       out.write("     call newHeapCell\n");
514
515       out.write("     movl    hp, %ecx\n");
516       out.write("     decl    %ecx\n");
517       // ecx ist heapindex des neuen Knoten
518
519       //den Multiplizieren wir mit zwei, weil vier wegen 4 bytes sind
    eine Adresse
520        out.write("     sall    $2, %ecx\n");
521
522       //darauf wird die startadresse des Heap addiert. Wir haben die
    Adresse!
523       out.write("     addl    heap, %ecx\n");
524
525        //und gehen zu diesen neuen Heapknoten
526
527       out.write("     movl    (%ecx), %ecx\n");
528       //nun also zum Array der Argumente (acht weiter. erst nr und
    arity)
529       out.write("     addl    $8, %ecx\n");
530       out.write("     movl    (%ecx), %ecx\n");
531
532       for (int i=0;i<ins.a2;i++){
533         genPop(out);
534         //hole argument vom stack
535         out.write("     movl    sp, %eax\n");
536         out.write("     sall    $3, %eax\n");
537         out.write("     addl    stack, %eax\n");
538
539         out.write("     movl    (%eax), %edx\n");
540
541         out.write("     movl    %edx, (%ecx)\n");
542   out.write("     addl    $4, %ecx\n");
543
544         out.write("     movl    4(%eax), %edx\n");
545
546         out.write("     movl    %edx, (%ecx)\n");
547   out.write("     addl    $4, %ecx\n");
548       }
549
550       out.write("     movl    hp, %ecx\n");
```

```
551         out.write("    decl    %ecx\n");
552         genPushRef(out,"%ecx"); //die Adresse im Heap drauf
553         break;
554 
555       case Unpack:
556         out.write("    call unpack\n");
557         break;
558 
559       case Project:
560         out.write("  "+mov()+"    $"+ins.a1+", callingArg\n");
561         out.write("    call project\n");
562         break;
563 
564       case Pop:
565         genPop(out);
566         break;
567 
568       case PopTops:
569         out.write("    call popTops\n");
570         break;
571 
572       case PushTop:
573         out.write("  movl $"+(ins.a1)+", %ebx\n");
574         out.write("  movl sp, %edx\n");
575         out.write("  movl %ebx, %eax\n");
576         out.write("  subl %eax, %edx\n");
577         out.write("  movl %edx, %eax\n");
578         out.write("  subl $1, %eax\n");
579         out.write("  movl %eax, %ebx\n");
580         out.write("  movl stack, %eax\n");
581         out.write("  movl sp, %edx\n");
582         out.write("  sall $3, %edx\n");
583         out.write("  addl %eax, %edx\n");
584         out.write("  movl stack, %eax\n");
585         out.write("  movl %ebx, %ecx\n");
586         out.write("  sall $3, %ecx\n");
587         out.write("  addl %ecx, %eax\n");
588         out.write("  movl (%eax), %eax\n");
589         out.write("  movl %eax, (%edx)\n");
590         out.write("  movl stack, %eax\n");
591         out.write("  movl sp, %edx\n");
592         out.write("  sall $3, %edx\n");
593         out.write("  addl %eax, %edx\n");
594         out.write("  movl stack, %eax\n");
595         out.write("  movl %ebx, %ecx\n");
596         out.write("  sall $3, %ecx\n");
597         out.write("  addl %ecx, %eax\n");
598         out.write("  movl 4(%eax), %eax\n");
599         out.write("  movl %eax, 4(%edx)\n");
600         out.write("  movl sp, %eax\n");
601         out.write("  addl $1, %eax\n");
602         out.write("  movl %eax, sp\n");
```

```
603
604 //       out.write("  call pushtop\n");
605       break;
606
607     case PushFun:
608       pushInt(out,"$"+ins.label);
609       break;
610
611     case FunPointerCall:
612       genPop(out,ax());
613       out.write("    call    *%eax\n");
614       break;
615
616     }
617   }
618 }
```

Listing 8.34: WriteX86.java

Kapitel 9

Entrümpelung (Garbage Collection)

Kapitel 10

Strings und Interaktion mit C

10.1 Parser

```
355 PARSER_BEGIN(WIPParser)
356 package v10.name.panitz.wip;
357 import  v10.name.panitz.wip.tree.*;
358 import java.util.List;
359 import java.util.ArrayList;
360
361 public class  WIPParser{}
362
363 PARSER_END(WIPParser)
364
365 SKIP :
366 {
367   " "
368 | "\t"
369 | "\n"
370 | "\r"
371 }
372
373
374
375 SKIP :
376 {
377   "/*" : WithinComment
378 }
379
380 <WithinComment> SKIP :
381 {
382   "*/" : DEFAULT
383 }
384
385 <WithinComment> MORE :
386 {
387   <□[]>
388 }
389
```

```
390
391 TOKEN:
392 [
393   <STRING_LITERAL:"\""((~["\"", "\\", "\n", "\r"])
394       | ("\\"(["n", "t", "b", "r", "f", "\\", "'", "\""]
395       | ["0"-"7"](["0"-"7"])?
396       | ["0"-"3"]["0"-"7"]["0"-"7"]))*"\"">
397 ]
```

Listing 10.1: WIPParser.jj

```
398 Tree unaryExpression() :
399 {Token tok;
400  Tree result = null;
401  Tree arg;
402  List<Tree> args = new ArrayList<Tree>();
403 }
404 {
405    (tok=<INTEGER_LITERAL>{result =  new IntLit(new Integer(tok.image))
     ;}
406    |tok=<STRING_LITERAL>{result =  new StringLit(tok.image.substring
     (1,tok.image.length()-1));}
407    |(<LPAR> result=expression() <RPAR>)
408    |result = funCallOrVar()
409    )(<DOT> tok=<IDENT>{result = new ProjectionExpr(result,tok.image)
     ;})*
410    {return result;}
411 }
```

Listing 10.2: WIPParser.jj

10.2 Testprogramme

10.3 Abstrakter Syntaxbaum

```
1  package v10.name.panitz.wip.tree;
2  public class Pair<A,B>{
3    public A e1;
4    public B e2;
5    public Pair(A e1,B e2){
6      this.e1 = e1;
7      this.e2 = e2;
8    }
9    public String toString(){return "("+e1+", "+e2+")";}
10 }
```

Listing 10.3: Pair.java

```
1  package v10.name.panitz.wip.tree;
2  public class StringLit implements Tree{
3    public String s;
4    public StringLit(String s){
5      this.s = s;
6    }
7    public String toString(){return "\""+s+"\"";}
8
9    public <R> R welcome(Visitor<R> visitor) throws Exception{
10     return visitor.visit(this);
11   }
12 }
```

Listing 10.4: StringLit.java

```
180   public Integer visit(StringLit il) throws Exception{
181     w.write("\""+il.s+"\"");
182     return indent;
183   }
```

Listing 10.5: PP.java

```
1  package v10.name.panitz.wip.tree;
2  public class CharVal implements Data{
3    public int c;
4    public CharVal(char c){this.c=c;}
5    public String toString(){return "\'"+c+"\"";}
6  }
```

Listing 10.6: CharVal.java

```
312   public Data visit(StringLit li) throws Exception{
313     List<Data> args = new LinkedList<>();
314     Data result = new ConData("Nil",args);
315     for (int i = li.s.length()-1;i>=0;i--){
316       args = new LinkedList<>();
317       args.add(new CharVal(li.s.charAt(i)));
318       args.add(result);
319       result = new ConData("Cons",args);
320     }
321     return result;
322   }
```

Listing 10.7: Interpreter.java

10.4 Abstrakte StackMaschine

```
104 package v10.name.panitz.wip.stackmachine;
105
106 public class  Instruction{
107   static public enum Instr
108     { PushInt, Mult, Div, Mod, Add, Sub, Eq, Neq, Lt, Gt, Le, Ge, And,
      Or
109     , JmpTrue, JmpGt, Jmp, Label, GlobLabel, Ret, Push, StartCall,
     Call, PushFun, FunPointerCall
110     , Pack,PackReverse, Unpack, Pop, PopTops, PushTop, Project,
     PushChar};
111
112   public Instr instr;
113   public int a1;
114   public int a2;
115   public String label;
116
117   public Instruction(Instr instr,int a1){
118     this.a1 = a1;
119     this.instr = instr;
120     this.label = "";
121   }
122
123
124   public Instruction(Instr instr,int a1,int a2){
125     this.a1 = a1;
126     this.a2 = a2;
127     this.instr = instr;
128     this.label = "";
129   }
130
131   public Instruction(Instr instr,String l){
132     this(instr,0);
133     this.label = l;
134   }
135
136   public Instruction(Instr instr){
137     this(instr,0);
138   }
139
140   public String toString(){return instr+" "+a1+" "+label;}
141 }
```

Listing 10.8: Instruction.java

```
14 package v10.name.panitz.wip.stackmachine;
15 public class RefOrInt{
16   boolean isRef;
17   int val;
18   boolean isChar = false;
19   public RefOrInt(boolean isRef,int val){
20     this.isRef = isRef;
```

```
21      this.val = val;
22    }
23    public RefOrInt(int val){this(false,val);}
24    public RefOrInt(char val){this(false,val);isChar=true;}
25
26    public String toString(){
27      if (isChar) return ""+((char)val);
28      return isRef?"(-> "+val+")":(val+"");
29    }
30  }
```

Listing 10.9: RefOrInt.java

```
24  package v10.name.panitz.wip.stackmachine;
25  import java.util.List;
26  class HeapNode{
27    public int name;
28    public RefOrInt[] args;
29
30    public HeapNode(int name,RefOrInt[] args){
31      this.name = name;
32      this.args = args;
33    }
34    public String toString(){
35      StringBuffer result = new StringBuffer();
36      result.append(name);
37      result.append("[");
38      boolean first = true;
39      for (RefOrInt arg: args){
40        if (first) { first = false;}else{result.append(", ");}
41        result.append(arg+"");
42      }
43      result.append("]");
44      return result.toString();
45    }
46  }
```

Listing 10.10: HeapNode.java

```
298 package v10.name.panitz.wip.stackmachine;
299 import java.util.Map;
300 import java.util.HashMap;
301 import java.io.*;
302
303 public class StackMachine{
304   public Instruction[] code;
305   public int ic = 0;
306
307   RefOrInt[] stack = new RefOrInt[2000];
308   int stc = 0;
309   int stb = 0;
```

```
310
311   HeapNode[] heap = new HeapNode[1909];
312   int hc = 0;
313
314   public String[] constructors;
315
316   public Map<String,Integer> labels = new HashMap<>();
317
318   void step(){
319     Instruction instruction = code[ic];
320     //System.out.println(ic+" "+instruction);
321
322     switch (instruction.instr) {
323     case PushInt: stack[stc++] = new RefOrInt(instruction.a1);ic++;
      break;
324     case PushChar: stack[stc++] = new RefOrInt((char)instruction.a1);
      ic++;break;
325     case Pop: stc--;ic++;break;
326     case PopTops: stc-=stack[--stc].val;ic++;break;
327     case Mult: stack[stc-2] = new RefOrInt(stack[stc-2].val * stack[
      stc-1].val);stc--;ic++;break;
328     case Mod:  stack[stc-2] = new RefOrInt(stack[stc-2].val % stack[
      stc-1].val);stc--;ic++;break;
329     case Div:  stack[stc-2] = new RefOrInt(stack[stc-2].val / stack[
      stc-1].val);stc--;ic++;break;
330     case Add:  stack[stc-2] = new RefOrInt(stack[stc-2].val + stack[
      stc-1].val);stc--;ic++;break;
331     case Sub:  stack[stc-2] = new RefOrInt(stack[stc-2].val - stack[
      stc-1].val);stc--;ic++;break;
332     case Eq:   stack[stc-2] = new RefOrInt(stack[stc-2].val == stack[
      stc-1].val?1:0);stc--;ic++;break;
333     case Neq:  stack[stc-2] = new RefOrInt(stack[stc-2].val != stack[
      stc-1].val?1:0);stc--;ic++;break;
334     case Lt:   stack[stc-2] = new RefOrInt(stack[stc-2].val < stack[
      stc-1].val?1:0);stc--;ic++;break;
335     case Gt:   stack[stc-2] = new RefOrInt(stack[stc-2].val > stack[
      stc-1].val?1:0);stc--;ic++;break;
336     case Le:   stack[stc-2] = new RefOrInt(stack[stc-2].val <= stack[
      stc-1].val?1:0);stc--;ic++;break;
337     case Ge:   stack[stc-2] = new RefOrInt(stack[stc-2].val >= stack[
      stc-1].val?1:0);stc--;ic++;break;
338     case And:  stack[stc-2] = new RefOrInt((stack[stc-2].val==1 && 1==
      stack[stc-1].val)?1:0);stc--;ic++;break;
339     case Or:   stack[stc-2] = new RefOrInt((stack[stc-2].val==1 || 1==
      stack[stc-1].val)?1:0);stc--;ic++;break;
340     case Jmp:  ic = labels.get(instruction.label);break;
341     case JmpTrue:
342         if (stack[stc-1].val==1){
343           ic = labels.get(instruction.label);
344          }else {
345           ic++;
346          }
```

```
347           stc--;
348           break;
349       case Label:  ic++;break;
350       case GlobLabel:  ic++;break;
351       case Ret: {
352         int args = instruction.a1;
353         if (instruction.a2 != 42) ic = stack[stb-1].val;
354         else ic++;
355
356         stack[stb-1-args] = stack[stc-1];
357         stc = stc-2-args;
358         stb = stack[stb].val;
359
360         break;
361       }
362       case Push:
363         stack[stc++] = stack[stb-code[ic++].a1-1];
364         break;
365       case PushTop:
366         stack[stc] = stack[stc-code[ic++].a1-1];
367         stc++;
368         break;
369       case StartCall:
370         if (instruction.a1==42)  stack[stc++] = new RefOrInt(ic+1);
371         stack[stc] = new RefOrInt(stb);
372         stb = stc;
373         stc++;
374         ic++;
375         break;
376       case Call:
377         if (instruction.label.equals("readFile")){
378           ic++;
379   break;
380         }
381         stack[stc++] = new RefOrInt(ic+1);
382         ic = labels.get(instruction.label);
383         break;
384
385       case PushFun:
386         stack[stc++] = new RefOrInt(labels.get(instruction.label));
387         ic++;
388         break;
389
390       case FunPointerCall:
391         int nextIc = ic+1;
392         ic = stack[--stc].val;
393         stack[stc++] = new RefOrInt(nextIc);
394         break;
395
396       case Pack:{
397         int argsNr = stack[--stc].val;
398         RefOrInt[] args = new RefOrInt[argsNr];
```

```
399
400       for (int i= argsNr-1;i>=0;i--){
401         args[i]= stack[--stc];
402       }
403
404       stack[stc++] = new RefOrInt(true,hc);
405       heap[hc++] = new HeapNode(instruction.a1,args);
406       ic++;
407       break;
408     }
409     case PackReverse:{
410       int argsNr = stack[--stc].val;
411       RefOrInt[] args = new RefOrInt[argsNr];
412
413       for (int i= 0;i<argsNr;i++){
414         args[i]= stack[--stc];
415       }
416
417       stack[stc++] = new RefOrInt(true,hc);
418       heap[hc++] = new HeapNode(instruction.a1,args);
419       ic++;
420       break;
421     }
422     case Unpack:
423       int heapNr = stack[--stc].val;
424       HeapNode hp = heap[heapNr];
425
426       for (RefOrInt arg:hp.args){
427         stack[stc++] = arg;
428       }
429       stack[stc++] = new RefOrInt(hp.args.length);
430       stack[stc++] = new RefOrInt(hp.name);
431       ic++;
432       break;
433     }
434   }
435
436   String showHeap(){
437     StringBuffer result = new StringBuffer();
438     for (int i=0;i<hc;i++){
439       result.append(i+": "+heap[i]+"\n");
440     }
441     result.append("\n");
442     return result.toString();
443   }
444
445   String showStack(){
446     StringBuffer result = new StringBuffer();
447     for (int i=0;i<stc;i++){
448       result.append(stack[i]+" ");
449     }
450     result.append("\n");
```

```
451       result.append("PC: "+ic+" BS: "+stb+" SP:"+stc+"\n");
452       result.append(code[ic]+"\n");
453       return result.toString();
454     }
455
456     public void run() throws IOException{
457       while (ic<code.length){
458 //        System.out.println(showHeap());
459 //        System.out.println(showStack());
460         step();
461       }
462       System.out.println(stack[0]);
463       Writer w = new PrintWriter(System.out);
464       printNode(stack[0],w);
465       w.write("\n");
466       w.flush();
467
468
469     }
470
471     public void printNode(RefOrInt n,Writer w)throws IOException{
472       if (!n.isRef){ w.write(n.toString());}
473       else{
474         w.write(constructors[heap[n.val].name]);
475         w.write("(");
476         boolean first = true;
477         for (RefOrInt arg:heap[n.val].args){
478           if (first) first = false; else w.write(", ");
479           printNode(arg,w);
480         }
481         w.write(")");
482       }
483     }
484
485     public static void main(String[] args)throws IOException{
486       StackMachine interp = new StackMachine();
487       interp.code = new Instruction[1];
488       interp.code[0] = new Instruction(Instruction.Instr.PushInt,42);
489       interp.run();
490     }
491
492     public String toString(){
493       StringBuffer result = new StringBuffer();
494       int i = 0;
495       for (Instruction x:code){
496         result.append((i++)+" "+x+"\n");
497       }
498       result.append("PC:  "+ic);
499
500       return result.toString();
501     }
502 }
```

Listing 10.11: StackMachine.java

```
445 package v10.name.panitz.wip.tree;
446 import v10.name.panitz.wip.stackmachine.*;
447
448 import java.util.List;
449 import java.util.LinkedList;
450 import java.util.Map;
451 import java.util.HashMap;
452
453 public class GenCode extends AbstractVisitor<List<Instruction>>{
454   List<Instruction> result = new LinkedList<>();
455
456   public Map<String,Integer> labels = new HashMap<>();
457   Map<String,Integer> varEnv = new HashMap<>();
458   Map<String,FunDef> funs = new HashMap<>();
459   Map<String,ConDef> cons = new HashMap<>();
460   String[] conNames;
461   Map<String,Integer> conNumbers = new HashMap<>();
462
463   int start=0;
464
465   public StackMachine getStackMachine(){
466     StackMachine res = new StackMachine();
467     List<Instruction> instr= new LinkedList<>();
468     instr.addAll(result);
469     instr.add(new Instruction(Instruction.Instr.Call,"go"));
470     res.code = instr.toArray(new Instruction[0]);
471     res.labels = labels;
472     res.ic = result.size();
473     res.constructors = conNames;
474     return res;
475   }
476
477   static int lbl = 0;
478   public static String label() {
479     lbl++;
480     return "label"+lbl;
481   }
482
483
484   public List<Instruction> visit(Program prog) throws Exception{
485     funs = prog.welcome(new CollectFunDefs());
486     cons = prog.welcome(new CollectConDefs());
487     conNames = new String[prog.cons.size()];
488     int i=0;
489     for (ConDef con:prog.cons){
490       conNames[i]=con.name;
491       conNumbers.put(con.name,i);
492       i++;
```

```
493       }
494
495       for (FunDef fun:prog.funs){
496         fun.welcome(this);
497       }
498       start = result.size();
499       return result;
500     }
501
502
503     public List<Instruction> visit(ProjectionExpr proj)  throws
          Exception{
504       proj.expr.welcome(this);
505
506       int i=0;
507
508       breakLabel: for (ConDef con:cons.values()){
509         i=0;
510         for (String arg:con.args){
511           if (arg.equals(proj.name)){
512       i = con.args.size()-i-1;
513       break breakLabel;
514           }
515     i++;
516         }
517       }
518       result.add(new Instruction(Instruction.Instr.Project,i));
519       return result;
520     }
521
522
523     public List<Instruction> visit(MatchExpr match)  throws Exception{
524       String endMatch = label();
525       match.expr.welcome(this);
526       result.add(new Instruction(Instruction.Instr.PushTop,0));
527       result.add(new Instruction(Instruction.Instr.Unpack));
528       for (CaseExpr cas:match.cases){
529          if (cas.isVarCase){
530            result.add(new Instruction(Instruction.Instr.Pop));
531            result.add(new Instruction(Instruction.Instr.PopTops));
532            cas.welcome(this);
533            result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
534          }else{
535            String endCase = label();
536            result.add(new Instruction(Instruction.Instr.PushTop,0));
537            result.add(new Instruction(Instruction.Instr.PushInt,
          conNumbers.get(cas.name)));
538            result.add(new Instruction(Instruction.Instr.Neq));
539            result.add(new Instruction(Instruction.Instr.JmpTrue,endCase)
          );
540            result.add(new Instruction(Instruction.Instr.Pop));
541            result.add(new Instruction(Instruction.Instr.Pop));
```

```
542           cas.welcome(this);
543           result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
544           result.add(new Instruction(Instruction.Instr.Label,endCase));
545           labels.put(endCase,result.size());
546         }
547       }
548       result.add(new Instruction(Instruction.Instr.Label,endMatch));
549       labels.put(endMatch,result.size());
550       return result;
551     }
552
553     public List<Instruction> visit(CaseExpr cas)  throws Exception{
554       FunCall theCase = (FunCall) cas.def;
555       if (!cas.isVarCase){
556         for (Tree arg:theCase.args.subList(cas.args.size()+1,theCase.
      args.size())){
557           arg.welcome(this);
558         }
559       } else{
560         for (Tree arg:theCase.args.subList(1,theCase.args.size())){
561           arg.welcome(this);
562         }
563       }
564       result.add(new Instruction(Instruction.Instr.Jmp,theCase.name));
565
566       result.add(new Instruction(Instruction.Instr.Label,theCase.name+"
      callReturn"));
567       labels.put(theCase.name+"callReturn",result.size());
568       return result;
569     }
570
571 /*
572
573     public List<Instruction> visit(MatchExpr match)  throws Exception{
574       String endMatch = label();
575       match.expr.welcome(this);
576       //nochmal drauf
577       result.add(new Instruction(Instruction.Instr.PushTop,0));
578       //auspacken
579       result.add(new Instruction(Instruction.Instr.Unpack));
580       for (CaseExpr cas:match.cases){
581          if (cas.isVarCase){
582           //Con Nr runter, jetzt sind argumente plus argumentanzahl
      drauf
583           result.add(new Instruction(Instruction.Instr.Pop));
584           //Argumente runter, nur noch das verpackte ist drauf
585           result.add(new Instruction(Instruction.Instr.PopTops));
586           //schaue zum case und mache dort alles!
587           cas.welcome(this);
588           //hat gematcht also fertig mit match
589           result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
590           // es gibt kein endCase Label, da danach kein case mehr
```

```
    matchen kann
591         }else{
592           String endCase = label();
593           //nummer drauf
594           result.add(new Instruction(Instruction.Instr.PushTop,0));
595           //zum vergleichen mit case Nummer
596           result.add(new Instruction(Instruction.Instr.PushInt,
    conNumbers.get(cas.name)));
597           //teste ungleichheit
598           result.add(new Instruction(Instruction.Instr.Neq));
599           //bei ungleichen zum naechsten case
600           result.add(new Instruction(Instruction.Instr.JmpTrue,endCase)
    );
601     //Nummer runter
602           result.add(new Instruction(Instruction.Instr.Pop));
603           //anzahl der Args runter
604           result.add(new Instruction(Instruction.Instr.Pop));
605           //schaue zum case und mache dort alles!
606           cas.welcome(this);
607           //hat gematcht also fertig mit match
608           result.add(new Instruction(Instruction.Instr.Jmp,endMatch));
609           result.add(new Instruction(Instruction.Instr.Label,endCase));
610           labels.put(endCase,result.size());
611       }
612     }
613     result.add(new Instruction(Instruction.Instr.Label,endMatch));
614     labels.put(endMatch,result.size());
615     return result;
616   }
617
618   public List<Instruction> visit(CaseExpr cas)  throws Exception{
619     FunCall theCase = (FunCall) cas.def;
620     if (!cas.isVarCase){
621       for (Tree arg:theCase.args.subList(cas.args.size()+1,theCase.
    args.size())){
622         arg.welcome(this);
623       }
624     } else{
625       for (Tree arg:theCase.args.subList(1,theCase.args.size())){
626         arg.welcome(this);
627       }
628     }
629     result.add(new Instruction(Instruction.Instr.Jmp,theCase.name));
630     result.add(new Instruction(Instruction.Instr.Label,theCase.name+"
    callReturn"));
631     labels.put(theCase.name+"callReturn",result.size());
632     return result;
633   }
634
635
636   public List<Instruction> visit(LetExpr let)  throws Exception{
637     for(Pair<String,Tree> arg:let.args) arg.e2.welcome(this);
```

```
638     FunCall theExpr = (FunCall) let.expr;
639     for (Tree arg:theExpr.args.subList(let.args.size(),theExpr.args.
        size())){
640         arg.welcome(this);
641     }
642     result.add(new Instruction(Instruction.Instr.Jmp,theExpr.name));
643 
644     result.add(new Instruction(Instruction.Instr.Label,theExpr.name+"
        callReturn"));
645     labels.put(theExpr.name+"callReturn",result.size());
646     return result;
647   }
648 
649 */
650   public List<Instruction> visit(FunDef fun)  throws Exception{
651     if(fun.isSingleCalled){
652       result.add(new Instruction(Instruction.Instr.Label,fun.name));
653       labels.put(fun.name,result.size());
654       result.add(new Instruction(Instruction.Instr.StartCall,42));
655     }else{
656       result.add(new Instruction(Instruction.Instr.GlobLabel,fun.name)
        );
657       labels.put(fun.name,result.size());
658       result.add(new Instruction(Instruction.Instr.StartCall));
659     }
660     int i = fun.args.size();
661     Map<String,Integer> newVarEnv = new HashMap<>();
662     for (String arg:fun.args){
663       newVarEnv.put(arg,i--);
664     }
665     Map<String,Integer> oldVarEnv = varEnv;
666     varEnv = newVarEnv;
667 
668     fun.def.welcome(this);
669 
670     varEnv = newVarEnv;
671 
672     if(fun.isSingleCalled){
673       result.add(new Instruction(Instruction.Instr.Ret,fun.args.size
        (),42));
674       result.add(new Instruction(Instruction.Instr.Jmp,fun.name+"
        callReturn"));
675     }else{
676       result.add(new Instruction(Instruction.Instr.Ret,fun.args.size()
        ));
677     }
678     return result;
679   }
680 
681   public List<Instruction> visit(Var v) throws Exception{
682     Integer vS = varEnv.get(v.name);
683     if (vS != null){
```

```
684       result.add(new Instruction(Instruction.Instr.Push,vS));
685       return result;
686     }
687     result.add(new Instruction(Instruction.Instr.PushFun,v.name));
688     return result;
689   }
690
691   public List<Instruction> visit(FunCall fun)  throws Exception{
692     for (Tree arg:fun.args){
693       arg.welcome(this);
694     }
695     Integer vS = varEnv.get(fun.name);
696     if (vS!=null){
697       result.add(new Instruction(Instruction.Instr.Push,vS));
698       result.add(new Instruction(Instruction.Instr.FunPointerCall));
699     }else {
700       Integer conNr = conNumbers.get(fun.name);
701       if (conNr!=null){
702         result.add(new Instruction(Instruction.Instr.PushInt,fun.args.
    size()));
703         result.add(new Instruction(Instruction.Instr.Pack,conNr,fun.
    args.size()));
704       }else{
705         result.add(new Instruction(Instruction.Instr.Call,fun.name));
706       }
707     }
708     return result;
709   }
710
711   public List<Instruction> visit(IntLit il) {
712     result.add(new Instruction(Instruction.Instr.PushInt,il.i));
713     return result;
714   }
715
716   public List<Instruction> visit(StringLit li) {
717     String str = li.s;
718     str=str.replace("\\\\", "\\");
719     str=str.replace("\\n", "\n");
720     str=str.replace("\\t", "\t");
721     str=str.replace("\\\"", "\"");
722     str=str.replace("\\\'", "\'");
723
724     result.add(new Instruction(Instruction.Instr.PushInt,0));
725     result.add(new Instruction(Instruction.Instr.Pack,0,0));
726     for (int i=str.length()-1;i>=0;i--){
727       result.add(new Instruction(Instruction.Instr.PushChar,str.charAt
    (i)));
728       result.add(new Instruction(Instruction.Instr.PushInt,2));
729       result.add(new Instruction(Instruction.Instr.PackReverse,1,2));
730     }
731     return result;
732   }
```

```
733
734   public List<Instruction> visit(IfExpr ex) throws Exception{
735     if (ex.cond instanceof GtExpr){
736       String thenLabel = label();
737       String endIfLabel = label();
738       visit((OpExpr)ex.cond);
739       result.add(new Instruction(Instruction.Instr.JmpGt,thenLabel));
740       ex.alt2.welcome(this);
741       result.add(new Instruction(Instruction.Instr.Jmp,endIfLabel));
742       labels.put(thenLabel,result.size());
743       result.add(new Instruction(Instruction.Instr.Label,thenLabel));
744       ex.alt1.welcome(this);
745       labels.put(endIfLabel,result.size());
746       result.add(new Instruction(Instruction.Instr.Label,endIfLabel));
747     }else{
748
749       String thenLabel = label();
750       String endIfLabel = label();
751
752       ex.cond.welcome(this);
753       result.add(new Instruction(Instruction.Instr.JmpTrue,thenLabel))
      ;
754       ex.alt2.welcome(this);
755       result.add(new Instruction(Instruction.Instr.Jmp,endIfLabel));
756       labels.put(thenLabel,result.size());
757       result.add(new Instruction(Instruction.Instr.Label,thenLabel));
758       ex.alt1.welcome(this);
759       labels.put(endIfLabel,result.size());
760       result.add(new Instruction(Instruction.Instr.Label,endIfLabel));
761     }
762     return result;
763   }
764
765   public void visit(OpExpr ex) throws Exception{
766     ex.left.welcome(this);
767     ex.right.welcome(this);
768   }
769   public List<Instruction> visit(MultExpr ex) throws Exception{
770     visit((OpExpr) ex);
771     result.add(new Instruction(Instruction.Instr.Mult));
772     return result;
773   }
774
775   public List<Instruction>  visit(DivExpr ex) throws Exception{
776     visit((OpExpr) ex);
777     result.add(new Instruction(Instruction.Instr.Div));
778     return result;
779   }
780   public List<Instruction>  visit(ModExpr ex) throws Exception{
781     visit((OpExpr) ex);
782     result.add(new Instruction(Instruction.Instr.Mod));
783     return result;
```

```
784     }
785     public List<Instruction>  visit(AddExpr ex) throws Exception{
786       visit((OpExpr) ex);
787       result.add(new Instruction(Instruction.Instr.Add));
788       return result;
789     }
790     public List<Instruction>  visit(SubExpr ex) throws Exception{
791       visit((OpExpr) ex);
792       result.add(new Instruction(Instruction.Instr.Sub));
793       return result;
794     }
795     public List<Instruction>  visit(EqExpr ex) throws Exception{
796       visit((OpExpr) ex);
797       result.add(new Instruction(Instruction.Instr.Eq));
798       return result;
799     }
800     public List<Instruction>  visit(NeqExpr ex) throws Exception{
801       visit((OpExpr) ex);
802       result.add(new Instruction(Instruction.Instr.Neq));
803       return result;
804     }
805     public List<Instruction>  visit(LeExpr ex) throws Exception{
806       visit((OpExpr) ex);
807       result.add(new Instruction(Instruction.Instr.Le));
808       return result;
809     }
810     public List<Instruction>  visit(GeExpr ex) throws Exception{
811       visit((OpExpr) ex);
812       result.add(new Instruction(Instruction.Instr.Ge));
813       return result;
814     }
815     public List<Instruction>  visit(LtExpr ex) throws Exception{
816       visit((OpExpr) ex);
817       result.add(new Instruction(Instruction.Instr.Lt));
818       return result;
819     }
820     public List<Instruction>  visit(GtExpr ex) throws Exception{
821       visit((OpExpr) ex);
822       result.add(new Instruction(Instruction.Instr.Gt));
823       return result;
824     }
825     public List<Instruction>  visit(OrExpr ex) throws Exception{
826       visit((OpExpr) ex);
827       result.add(new Instruction(Instruction.Instr.Or));
828       return result;
829     }
830     public List<Instruction>  visit(AndExpr ex) throws Exception{
831       visit((OpExpr) ex);
832       result.add(new Instruction(Instruction.Instr.And));
833       return result;
834     }
835 }
```

Listing 10.12: GenCode.java

10.5 Runtime

```
304 #include <stdlib.h>
305 #include <stdio.h>
306 #include <stdbool.h>
307
308 #include <sys/resource.h>
309 #include <sys/stat.h>
310
311 #define HEAP_SIZE 20*1048576
312 #define MAX_SIZE 14*1048576
313 #define STACK_SIZE 2*1048576
314
315
316
317 #define NIL 0
318 #define CONS 1
319
320 extern char* constructors[];
321
322
323 typedef enum {
324   Ref, Prim, Char
325 } RefPrimFlag;
326
327
328
329 typedef struct {
330   RefPrimFlag refprimflag;
331   int refprimval;
332 } RefPrim;
333
334 void printRefPrim(RefPrim* rp){
335   switch (rp->refprimflag ){
336     case Ref:  printf("(-> %d)",rp->refprimval);break;
337     case Prim: printf(" %d",rp->refprimval);break;
338     case Char: printf(" %c",(char)rp->refprimval);break;
339   }
340 }
341
342 void printStack();
343 void printHeap();
344
345 int callingArg;
346
```

```
347
348 typedef struct [
349   int constrNr;
350   int arity;
351   RefPrim* args;
352 ] HeapNode;
353
354 void printHeapNode(HeapNode* h){
355   printf("Constr<%d %d>(",h->constrNr,h->arity);
356   int i;
357   for (i=0;i<h->arity;i++){
358     if (i>0) printf(",");
359       printRefPrim(h->args+i);
360   }
361   printf(")");
362 ]
363
364 HeapNode** heap;
365 HeapNode** gcheap;
366
367 unsigned int hp=0;
368 unsigned int heapSize = 909;
369 unsigned int lastUsedheapSize = 909;;
370
371 int nr;
372 int arity;
373
374 HeapNode* newCell(unsigned int arity,int nr){
375   HeapNode* newNode = (HeapNode*)malloc(sizeof(HeapNode));
376   newNode->constrNr = nr;
377   newNode->arity = arity;
378   return newNode;
379 ]
380
381 void gc();
382
383 HeapNode* newHeapCell(){
384   if (hp>=heapSize-1){
385     gc();
386   }
387
388   RefPrim* args = (RefPrim*)malloc(sizeof(RefPrim)*arity);
389   HeapNode* newNode = (HeapNode*)malloc(sizeof(HeapNode));
390
391   newNode->constrNr = nr;
392   newNode->arity = arity;
393   newNode->args = args;
394
395   heap[hp] = newNode;
396   hp++;
397
398   return newNode;
```

```
399 }
400
401
402 RefPrim* stack;
403 RefPrim* baseptr;
404 unsigned int sp = 0;
405 unsigned int sb = 0;
406
407 void unpack(){
408   int heapRef = stack[--sp].refprimval;
409   int nr = heap[heapRef]->constrNr;
410   int i;
411
412   if (sp+heap[heapRef]->arity+1>=STACK_SIZE){
413     printf("stack full\n");
414     exit(-1);
415   }
416
417   for (i=heap[heapRef]->arity-1;i>=0;i--){
418     stack[sp].refprimflag = heap[heapRef]->args[i].refprimflag;
419     stack[sp].refprimval = heap[heapRef]->args[i].refprimval;
420     sp++;
421   }
422   stack[sp].refprimflag = Prim;
423   stack[sp].refprimval = heap[heapRef]->arity;
424   sp++;
425   stack[sp].refprimflag = Prim;
426   stack[sp].refprimval = nr;
427   sp++;
428 }
429
430
431 int heapRef;
432 void project(){
433   heapRef = stack[--sp].refprimval;
434   stack[sp].refprimflag = heap[heapRef]->args[callingArg].refprimflag;
435   stack[sp].refprimval = heap[heapRef]->args[callingArg].refprimval;
436   sp++;
437 }
438
439
440 void pushPrim(){
441   if (sp+1>STACK_SIZE){
442     printf("stack full\n");
443     exit(-1);
444   }
445   stack[sp].refprimflag=Prim;
446   stack[sp].refprimval=callingArg;
447   sp++;
448 }
449
450 void pushRef(){
```

```
451   if (sp+1>STACK_SIZE){
452     printf("stack full\n");
453     exit(-1);
454   }
455   stack[sp].refprimflag=Ref;
456   stack[sp].refprimval=callingArg;
457   sp++;
458 }
459
460 void pop(){
461   sp--;
462 }
463
464 void popTops(){
465   sp = sp - stack[--sp].refprimval-1;
466 }
467
468 void push(){
469   if (sp+1>STACK_SIZE){
470     printf("stack full\n");
471     exit(-1);
472   }
473   stack[sp].refprimflag = stack[sb-callingArg].refprimflag;
474   stack[sp].refprimval  = stack[sb-callingArg].refprimval;
475   sp++;
476 }
477
478
479 void pushtop(){
480   if (sp+1>STACK_SIZE){
481     printf("stack full\n");
482     exit(-1);
483   }
484   stack[sp].refprimflag = stack[sp-callingArg-1].refprimflag;
485   stack[sp].refprimval  = stack[sp-callingArg-1].refprimval;
486   sp++;
487 }
488
489 HeapNode** allocHeap(unsigned int size) {
490   HeapNode** result = (HeapNode**) malloc(size*sizeof(HeapNode*));
491   if (!result) {
492     printf("could not allocate heap\n");
493     exit(-1);
494   }
495   return result;
496 }
497
498 unsigned int copy(unsigned int heapIndex){
499   HeapNode* oldcell = heap[heapIndex];
500
501   if (oldcell->constrNr == -1){
502     return oldcell->arity;
```

```
503   }else{
504     HeapNode* newcell = newCell(oldcell->arity,oldcell->constrNr);
505     newcell->args=oldcell->args;
506     oldcell->constrNr = -1;
507     oldcell->arity = lastUsedheapSize;
508     unsigned int result = lastUsedheapSize;
509     gcheap[lastUsedheapSize] = newcell;
510     lastUsedheapSize++;
511
512     int ar=newcell->arity;
513     int ai;
514     for (ai=0;ai< ar;ai++){
515       newcell->args[ai].refprimflag = oldcell->args[ai].refprimflag;
516       if (oldcell->args[ai].refprimflag == Ref){
517         newcell->args[ai].refprimval = copy(oldcell->args[ai].
    refprimval);
518       }else{
519         newcell->args[ai].refprimval = oldcell->args[ai].refprimval;
520       }
521     }
522
523     return result;
524   }
525 }
526
527 void gc(){
528   printf("Starting gc. old heap: %d",hp-1);
529   int luh3  = lastUsedheapSize*3;
530   int newSize = luh3 > heapSize ? luh3 : heapSize;
531   if (newSize>MAX_SIZE) newSize = MAX_SIZE;
532   printf(" new size: %d\n",newSize);
533   gcheap = allocHeap(newSize);
534   lastUsedheapSize = 0;
535
536   int i;
537   for (i=0;i<sp;i++ ){
538     if ((stack+i)->refprimflag==Ref){
539       (stack+i)->refprimval = copy((stack+i)->refprimval);
540     }
541   }
542
543   //alten heap löschen
544   //int i;
545   for (i=0;i<hp;i++){
546     if (heap[i]->constrNr!=-1)free(heap[i]->args);
547     free(*(heap+i));
548   }
549   free(heap);
550
551   //neuen heap setzen
552   heap = gcheap;
553   heapSize = newSize;
```

```
554   hp = lastUsedheapSize;
555   printf("  Ending gc. new heap: %d\n",hp);
556 }
557
558 void printStack() {
559   int i;
560   for (i = 0; i < sp; i++) {
561     printRefPrim(&stack[i]);
562     printf("\n");
563   }
564 }
565
566 void printHeap(){
567   printf("\nHEAP\n");
568   int i;
569   for (i=0;i<hp;i++){
570     printf("%d:  ",i);
571     printHeapNode(heap[i]);
572     printf("\n");
573   }
574 }
575
576 void ret(){
577   RefPrim result = stack[--sp];
578   sb = stack[--sp].refprimval;
579   sp -= callingArg;
580   stack[sp++] = result;
581 }
582
583
584
585 void printString(unsigned int node){
586   printf("\"");
587   while (heap[node]->constrNr!=NIL) {
588     printf("%c",heap[node]->args[1].refprimval);
589     node = heap[node]->args[0].refprimval;
590   }
591   printf("\"");
592 }
593
594
595 void printHeapCell(unsigned int node) {
596   if (    heap[node]->constrNr==CONS
597       &&   heap[node]->args[1].refprimflag==Char){
598     printString(node);
599     return;
600   }
601   printf("%s(", constructors[heap[node]->constrNr]);
602   int i;
603   for (i = heap[node]->arity - 1; i >= 0; i--) {
604     if (i < heap[node]->arity - 1) printf(", ");
605     if (heap[node]->args[i].refprimflag == Prim) {
```

```
606       printf("%d", heap[node]->args[i].refprimval);
607     } else if (heap[node]->args[i].refprimflag == Char) {
608       printf("%c", (char) heap[node]->args[i].refprimval);
609     } else {
610       printHeapCell(heap[node]->args[i].refprimval);
611     }
612   }
613   printf(")");
614 }
615
616
617 void printResult() {
618   if (stack[0].refprimflag == Prim)
619     printf("%d", stack[0].refprimval);
620   else if (stack[0].refprimflag == Char)
621     printf("%c", (char) stack[0].refprimval);
622   else {
623     printHeapCell(stack[0].refprimval);
624   }
625 }
626
627
628
629 void mkString(char* cs, unsigned int size) {
630   arity = 0;
631   nr = NIL;
632   HeapNode* newNode = newHeapCell();
633   stack[sp].refprimflag = Ref;
634   stack[sp].refprimval = hp-1;
635   sp++;
636
637   int i;
638   for (i = size - 1; i >= 0; i--) {
639     arity = 2;
640     nr = CONS;
641     newNode = newHeapCell();
642
643     newNode->args[0].refprimflag = Ref;
644     newNode->args[0].refprimval = stack[sp-1].refprimval ;
645     newNode->args[1].refprimflag = Char;
646     newNode->args[1].refprimval = cs[i];
647     stack[sp-1].refprimflag = Ref;
648     stack[sp-1].refprimval = hp-1;
649   }
650 }
651
652
653 char* getString() {
654   int i = 0;
655   RefPrim constr = stack[sp-1];
656   while (heap[constr.refprimval]->constrNr != NIL) {
657     constr = heap[constr.refprimval]->args[0];
```

```
658     i++;
659   }
660   char* result = malloc((i + 1) * sizeof(char));
661   constr = stack[sp-1];
662   int j;
663   for (j = 0; j < i; j++) {
664     result[j] = (char) heap[constr.refprimval]->args[1].refprimval;
665     constr = heap[constr.refprimval]->args[0];
666   }
667   result[i] = '\0';
668   return result;
669 }
670
671
672 typedef unsigned int sizetype;
673
674 void readFile() {
675   printf("Datei soll gelesen werden\n");
676
677   char* fileName = getString();
678   struct stat fileStat;
679   printf("Datei: %s\n",fileName);
680
681   if (stat(fileName, &fileStat) == 0) {
682     // Dateigröße ermitteln
683     sizetype fileSize = fileStat.st_size;
684     printf("%d\n",fileSize);
685     // String mit Größe erstellen
686     char* content = malloc((fileSize + 1) * sizeof(char));
687     if (!content) {
688       printf("could not allocate String constant\n");
689       exit(-1);
690     }
691     content[fileSize] = '\0';
692
693     // Datei öffnen (nur Lesezugriff symbolisiert durch das "r")
694     FILE* file = fopen(fileName, "r");
695     printf("open file %s\n",fileName);
696     if (file) {
697       printf("opened file %s\n",fileName);
698       int i;
699       for (i = 0; !feof(file); i++) {
700         char c = fgetc(file);
701         if (c != EOF) {
702           content[i] = c;
703         }
704       }
705
706       printf("making String of content\n");
707       sp--;
708       mkString(content,fileSize+1);
709       printf("free content\n");
```

```
710       free(content);
711       printf("ready\n");
712     ]
713   } else {
714     mkString("geht nicht",0);
715     printf("not ready\n");
716   }
717 }
718
719 void swap() {
720   RefPrimFlag rp = stack[sp - 1].refprimflag;
721   int v = stack[sp - 1].refprimval;
722   stack[sp - 1].refprimflag = stack[sp - 2].refprimflag;
723   stack[sp - 1].refprimval = stack[sp - 2].refprimval;
724   stack[sp - 2].refprimflag = rp;
725   stack[sp - 2].refprimval = v;
726 }
727
728
729 void writeFile(){
730   swap();
731   char* fileName = getString();
732   printf("Datei: %s\n",fileName);
733   sp--;
734   FILE* file = fopen(fileName,"w");
735
736   // Checken ob Datei geöffnet werden konnte und dann Fehlercode
       zurückgeben
737   if (file==NULL){
738     sp--;
739     pushPrim(-1);
740     return ;
741   }
742
743   HeapNode constr = *(heap[stack[sp - 1].refprimval]);
744   while (constr.constrNr != NIL) {
745     fprintf(file,"%c",(char) constr.args[1].refprimval);
746     constr = *(heap[constr.args[0].refprimval]);
747   }
748   fclose(file);
749   pushPrim(0);
750 }
751
752
753
754
755 void appendFile(){
756   swap();
757   char* fileName = getString();
758   sp--;
759   FILE* file = fopen(fileName,"a");
760
```

```
761   // Checken ob Datei geöffnet werden konnte und dann Fehlercode
        zurückgeben
762   if (file==NULL){
763     sp--;
764     pushPrim(-1);
765     return ;
766   }
767
768   HeapNode constr = *(heap[stack[sp - 1].refprimval]);
769   while (constr.constrNr != NIL) {
770     fprintf(file,"%c",(char) constr.args[1].refprimval);
771     constr = *(heap[constr.args[0].refprimval]);
772   }
773   fclose(file);
774   pushPrim(0);
775 }
776
777
778
779
780 int main (int argc, char **argv)
781 {
782   const rlim_t kStackSize = 125 * 1024 * 1024;
783   struct rlimit rl;
784   int result;
785
786   result = getrlimit(RLIMIT_STACK, &rl);
787   if (result == 0){
788     if (rl.rlim_cur < kStackSize){
789       rl.rlim_cur = kStackSize;
790       result = setrlimit(RLIMIT_STACK, &rl);
791       if (result != 0){
792   fprintf(stderr, "setrlimit returned result = %d\n", result);
793       }
794     }
795   }
796   stack=(RefPrim*)malloc(STACK_SIZE*sizeof(RefPrim));
797   baseptr=stack;
798
799   heap = allocHeap(heapSize * sizeof(HeapNode*));
800   go();
801   gc();
802   printResult();
803   printf("\n");
804 //   printf("\nSTACK\n");
805 //   printStack();
806 //   printHeap();
807   return 0;
808 }
```

Listing 10.13: runt.c

```
85  package v10.name.panitz.wip;
86  import v10.name.panitz.wip.tree.*;
87  import v10.name.panitz.wip.stackmachine.*;
88  import java.util.*;
89  import java.io.*;
90
91  public class Main{
92    public static void main(String[] args) throws Exception{
93      WIPParser parser = null;
94      if (args.length == 0){
95        parser = new WIPParser(System.in);
96      }else {
97        parser = new WIPParser(new FileReader(args[0]));
98      }
99      Tree il = parser.prog();
100
101     StringWriter out  = new StringWriter();
102     PP pp = new PP(out);
103     il.welcome(pp);
104     System.out.println(out);
105
106
107     GenCode genCode = new GenCode();
108     List<Instruction> code = il.welcome(genCode);
109
110     StackMachine st = genCode.getStackMachine();
111     //System.out.println(st);
112
113     WriteX86 asm = new WriteX86(st.code,st.constructors);
114     FileWriter asmf = new FileWriter("test.s");
115     asm.writeAssembler(asmf);
116
117     WriteX86 asm64 = new WriteX86(st.code,st.constructors);
118     asm64._32 = false;
119 //    FileWriter asmf64 = new FileWriter("test64.s");
120 //    asm64.writeAssembler(asmf64);
121
122
123 //    System.out.println(il.welcome(new Interpreter()));
124
125 //    st.run();
126
127   }
128 }
```

Listing 10.14: Main.java

10.6 Assembler Generierung

```
619 package v10.name.panitz.wip.stackmachine;
620 import  v10.name.panitz.wip.stackmachine.Instruction.Instr;
621 import java.io.*;
622 import static v10.name.panitz.wip.tree.GenCode.*;
623
624 public class WriteX86{
625   public boolean _32 = true;
626
627   String mov(){return _32?"movl":"movq";}
628   String ax(){return _32?"%eax":"%rax";}
629   String bx(){return _32?"%ebx":"%rbx";}
630   String cx(){return _32?"%ecx":"%rcx";}
631   String dx(){return _32?"%edx":"%rdx";}
632   String sp(){return _32?"%esp":"%rsp";}
633   String bp(){return _32?"%ebp":"%rbp";}
634   String push(){return _32?"pushl":"pushq";}
635   String pop(){return _32?"popl":"popq";}
636   String mul(){return _32?"mull":"mulq";}
637   String add(){return _32?"addl":"addq";}
638   String sub(){return _32?"subl":"subq";}
639   String div(){return _32?"divl":"divq";}
640   String and(){return _32?"andl":"andq";}
641   String or(){return _32?"orl":"orq";}
642
643   String cmp(){return _32?"cmpl":"cmpq";}
644   String jmp(){return "jmp";}
645   String jne(){return "jne";}
646   String jnl(){return "jnl";}
647
648
649   String[] conNames;
650   public Instruction[] code;
651
652   public WriteX86(Instruction[] code,  String[] conNames){
653     this.code = code;
654     this.conNames = conNames;
655   }
656
657   public void generateConstructorArray(Writer out)throws IOException{
658     out.write(".globl  constructors\n");
659     out.write(".section        .rodata\n");
660     int i=0;
661     for (String c:conNames){
662        out.write(".LC"+i+":\n");
663        out.write(".string \""+c+"\"\n");
664        i++;
665     }
666     out.write(".data\n");
667     out.write(".align 4\n");
668     out.write(".type   constructors, @object\n");
669     out.write(".size   constructors, "+(4*conNames.length)+"\n");
670     out.write("constructors:\n");
```

```
671     i=0;
672     for (String c:conNames){
673       out.write(".long    .LC"+i+"\n");
674       i++;
675     }
676   }
677
678
679   private void genCompareExpr(Writer out,String jmp)throws Exception{
680     genPop(out,"%ecx");
681     genPop(out,"%edx");
682     out.write("     movl    %edx, %ebx\n");
683     out.write("     movl    %ecx, %eax\n");
684     out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
685
686     String endLabel = label();
687     String posLabel = label();
688
689     out.write("    "+jmp+"    "+posLabel+"\n");
690     pushInt(out,0);
691     out.write("    "+jmp()+"    "+endLabel+"\n");
692     out.write("    "+posLabel+":\n");
693     pushInt(out,1);
694     out.write("    "+endLabel+":\n");
695   }
696
697   public void writeAssembler(Writer out)throws Exception{
698     generateConstructorArray(out);
699     for (Instruction in:code){
700       writeAssembler(out,in);
701     }
702     out.close();
703   }
704
705   public void pushInt(Writer out,int prim)throws Exception{
706     out.write("     movl stack, %eax\n");
707     out.write("     movl sp, %edx\n");
708     out.write("     sall $3, %edx\n");
709     out.write("     addl %edx, %eax\n");
710     out.write("     movl $1, (%eax)\n");
711     out.write("     movl stack, %eax\n");
712     out.write("     movl sp, %edx\n");
713     out.write("     sall $3, %edx\n");
714     out.write("     addl %eax, %edx\n");
715     out.write("     movl $"+prim+", %eax\n");
716     out.write("     movl %eax, 4(%edx)\n");
717     out.write("     movl sp, %eax\n");
718     out.write("     addl $1, %eax\n");
719     out.write("     movl %eax, sp\n");
720   }
721
722   public void pushChar(Writer out,int prim)throws Exception{
```

```
723     out.write("    movl stack, %eax\n");
724     out.write("    movl sp, %edx\n");
725     out.write("    sall $3, %edx\n");
726     out.write("    addl %edx, %eax\n");
727     out.write("    movl $2, (%eax)\n");
728     out.write("    movl stack, %eax\n");
729     out.write("    movl sp, %edx\n");
730     out.write("    sall $3, %edx\n");
731     out.write("    addl %eax, %edx\n");
732     out.write("    movl $"+prim+", %eax\n");
733     out.write("    movl %eax, 4(%edx)\n");
734     out.write("    movl sp, %eax\n");
735     out.write("    addl $1, %eax\n");
736     out.write("    movl %eax, sp\n");
737   }
738
739
740   private void pushInt(Writer out,String reg)throws Exception{
741     out.write("    movl "+reg+", %ebx\n");
742     out.write("    movl stack, %eax\n");
743     out.write("    movl sp, %edx\n");
744     out.write("    sall $3, %edx\n");
745     out.write("    addl %edx, %eax\n");
746     out.write("    movl $1, (%eax)\n");
747     out.write("    movl stack, %eax\n");
748     out.write("    movl sp, %edx\n");
749     out.write("    sall $3, %edx\n");
750     out.write("    addl %eax, %edx\n");
751     out.write("    movl %ebx, %eax\n");
752     out.write("    movl %eax, 4(%edx)\n");
753     out.write("    movl sp, %eax\n");
754     out.write("    addl $1, %eax\n");
755     out.write("    movl %eax, sp\n");
756   }
757
758   private void genPushRef(Writer out,String reg)throws Exception{
759     out.write("    movl "+reg+", %ebx\n");
760     out.write("    movl stack, %eax\n");
761     out.write("    movl sp, %edx\n");
762     out.write("    sall $3, %edx\n");
763     out.write("    addl %edx, %eax\n");
764     out.write("    movl $0, (%eax)\n");
765     out.write("    movl stack, %eax\n");
766     out.write("    movl sp, %edx\n");
767     out.write("    sall $3, %edx\n");
768     out.write("    addl %eax, %edx\n");
769     out.write("    movl %ebx, %eax\n");
770     out.write("    movl %eax, 4(%edx)\n");
771     out.write("    movl sp, %eax\n");
772     out.write("    addl $1, %eax\n");
773     out.write("    movl %eax, sp\n");
774
```

```
775
776   }
777
778   private void genPop(Writer out)throws Exception{
779     out.write("    movl    sp, %eax\n");
780     out.write("    subl    $1, %eax\n");
781     out.write("    movl    %eax, sp\n");
782   }
783
784
785   private void genPop(Writer out,String reg)throws Exception{
786     genPop(out);
787     out.write("    movl    sp, %eax  #save pop in reg\n");
788     out.write("    sall    $3, %eax\n");
789     out.write("    addl    stack, %eax\n");
790     out.write("    addl    $4, %eax\n");
791     out.write("    movl    (%eax), "+reg+"\n");
792   }
793
794   public void writeAssembler(Writer out,Instruction ins)throws
       Exception{
795     switch (ins.instr) {
796     case PushInt: pushInt(out,ins.a1); break;
797     case PushChar: pushChar(out,ins.a1); break;
798
799     case Mult:
800       genPop(out,"%ecx");
801       genPop(out,"%edx");
802       out.write("    movl    %edx, %ebx\n");
803       out.write("    movl    %ecx, %eax\n");
804       out.write("    mull    %ebx\n");
805       pushInt(out,"%eax");
806       break;
807
808     case Add:
809       genPop(out,"%ecx");
810       genPop(out,"%edx");
811       out.write("    addl    %ecx, %edx\n");
812       pushInt(out,"%edx");
813       break;
814
815     case Sub:
816       genPop(out,"%ecx");
817       genPop(out,"%edx");
818       out.write("    subl    %ecx, %edx\n");
819       pushInt(out,"%edx");
820       break;
821
822     case Div:
823       genPop(out,"%edx");
824       genPop(out,"%ecx");
825       out.write("    movl    %edx, %ebx\n");
```

```
826         out.write("    movl    %ecx, %eax\n");
827         out.write("    movl $0, %edx\n");
828         out.write("    idivl %ebx\n");
829         pushInt(out,"%eax");
830         break;
831 
832       case Mod:
833         genPop(out,"%edx");
834         genPop(out,"%ecx");
835         out.write("    movl    %edx, %ebx\n");
836         out.write("    movl    %ecx, %eax\n");
837         out.write("    movl $0, %edx\n");
838         out.write("    idivl %ebx\n");
839         pushInt(out,"%edx");
840         break;
841 
842       case Eq:
843         genCompareExpr(out,"je");
844         break;
845 
846       case Neq:
847         genCompareExpr(out,"jne");
848         break;
849 
850       case Lt:
851         genCompareExpr(out,"jl");
852         break;
853 
854       case Le:
855         genCompareExpr(out,"jle");
856         break;
857 
858       case Gt:
859         genCompareExpr(out,"jg");
860         break;
861 
862       case Ge:
863         genCompareExpr(out,"jge");
864         break;
865 
866       case And:
867         genPop(out,"%ebx");
868         genPop(out,"%eax");
869         out.write("    "+and()+"    "+bx()+", "+ax()+"\n");
870         pushInt(out,"%eax");
871         break;
872 
873       case Or:
874         genPop(out,"%ebx");
875         genPop(out,"%eax");
876         out.write("    "+or()+"    "+bx()+", "+ax()+"\n");
877         pushInt(out,"%eax");
```

```
878       break;
879
880     case Jmp:
881       out.write("    "+jmp()+"    "+ins.label+"\n");
882       break;
883
884     case Label:
885       out.write(ins.label+":\n");
886       break;
887
888     case GlobLabel:
889       out.write(".globl "+ins.label+"\n");
890       out.write(ins.label+":\n");
891       break;
892
893     case JmpTrue:
894       genPop(out,"%eax");
895       out.write("    "+mov()+"    $1, "+bx()+"\n");
896       out.write("    "+cmp()+"    "+ax()+", "+bx()+"\n");
897       out.write("    je     "+ins.label+"\n");
898       break;
899
900     case JmpGt:
901       genPop(out,"%ebx");
902       genPop(out,"%eax");
903 //      out.write("     movl    %edx, %ebx\n");
904 //      out.write("     movl    %ecx, %eax\n");
905       out.write("    "+cmp()+"    "+bx()+", "+ax()+"\n");
906       out.write("    jg     "+ins.label+"\n");
907       break;
908
909     case Ret:
910       out.write("   "+mov()+"    $"+ins.a1+", callingArg\n");
911       out.write("    call ret\n");
912       if (ins.a2!=42){
913         out.write("    ret    \n");
914       }
915       break;
916
917     case Push:
918       out.write("   movl $"+(ins.a1+1)+", callingArg #start push\n");
919       out.write("   movl stack, %eax\n");
920       out.write("   movl sp, %edx\n");
921       out.write("   sall $3, %edx\n");
922       out.write("   addl %eax, %edx\n");
923       out.write("   movl stack, %eax\n");
924       out.write("   movl sb, %ebx\n");
925       out.write("   movl callingArg, %ecx\n");
926       out.write("   subl %ecx, %ebx\n");
927       out.write("   movl %ebx, %ecx\n");
928       out.write("   sall $3, %ecx\n");
929       out.write("   addl %ecx, %eax\n");
```

```
930       out.write("  movl (%eax), %eax\n");
931       out.write("  movl %eax, (%edx)\n");
932       out.write("  movl stack, %eax\n");
933       out.write("  movl sp, %edx\n");
934       out.write("  sall $3, %edx\n");
935       out.write("  addl %eax, %edx\n");
936       out.write("  movl stack, %eax\n");
937       out.write("  movl sb, %ebx\n");
938       out.write("  movl callingArg, %ecx\n");
939       out.write("  subl %ecx, %ebx\n");
940       out.write("  movl %ebx, %ecx\n");
941       out.write("  sall $3, %ecx\n");
942       out.write("  addl %ecx, %eax\n");
943       out.write("  movl 4(%eax), %eax\n");
944       out.write("  movl %eax, 4(%edx)\n");
945       out.write("  movl sp, %eax\n");
946       out.write("  addl $1, %eax\n");
947       out.write("  movl %eax, sp\n");
948       break;
949
950     case StartCall:
951       out.write("   "+mov() +"   sb,"+cx()+"\n");
952       pushInt(out,"%ecx");
953       out.write("   "+mov() +"   sp,"+dx()+"\n");
954       out.write("   "+mov() +"   "+dx()+", sb\n");
955       break;
956
957     case Call:
958       out.write("   call   "+ins.label+"\n");
959       break;
960
961     case Pack:
962       genPop(out,"%ecx");
963       out.write("    movl    %ecx, arity\n");
964       out.write("    movl    $"+ins.a1+", nr\n");
965       out.write("    call newHeapCell\n");
966
967       out.write("    movl    hp, %ecx\n");
968       out.write("    decl    %ecx\n");
969       // ecx ist heapindex des neuen Knoten
970
971       //den Multiplizieren wir mit zwei, weil vier wegen 4 bytes sind
    eine Adresse
972        out.write("    sall    $2, %ecx\n");
973
974       //darauf wird die startadresse des Heap addiert. Wir haben die
    Adresse!
975       out.write("    addl    heap, %ecx\n");
976
977        //und gehen zu diesen neuen Heapknoten
978
979       out.write("    movl    (%ecx), %ecx\n");
```

```
980       //nun also zum Array der Argumente (acht weiter. erst nr und
      arity)
981       out.write("    addl    $8, %ecx\n");
982       out.write("    movl    (%ecx), %ecx\n");
983
984       for (int i=0;i<ins.a2;i++){
985         genPop(out);
986         //hole argument vom stack
987         out.write("    movl    sp, %eax\n");
988         out.write("    sall    $3, %eax\n");
989         out.write("    addl    stack, %eax\n");
990
991         out.write("    movl    (%eax), %edx\n");
992
993         out.write("    movl    %edx, (%ecx)\n");
994   out.write("    addl    $4, %ecx\n");
995
996         out.write("    movl    4(%eax), %edx\n");
997
998         out.write("    movl    %edx, (%ecx)\n");
999   out.write("    addl    $4, %ecx\n");
1000      }
1001
1002      out.write("    movl    hp, %ecx\n");
1003      out.write("    decl    %ecx\n");
1004      genPushRef(out,"%ecx"); //die Adresse im Heap drauf
1005      break;
1006
1007
1008    case PackReverse:
1009      genPop(out,"%ecx");
1010      out.write("    movl    %ecx, arity\n");
1011      out.write("    movl    $"+ins.a1+", nr\n");
1012      out.write("    call newHeapCell\n");
1013
1014      out.write("    movl    hp, %ecx\n");
1015      out.write("    decl    %ecx\n");
1016      // ecx ist heapindex des neuen Knoten
1017
1018      //den Multiplizieren wir mit zwei, weil vier wegen 4 bytes sind
      eine Adresse
1019       out.write("    sall    $2, %ecx\n");
1020
1021      //darauf wird die startadresse des Heap addiert. Wir haben die
      Adresse!
1022      out.write("    addl    heap, %ecx\n");
1023
1024       //und gehen zu diesen neuen Heapknoten
1025
1026      out.write("    movl    (%ecx), %ecx\n");
1027      //nun also zum Array der Argumente (acht weiter. erst nr und
      arity)
```

```
1028        out.write("    addl    $8, %ecx\n");
1029        out.write("    movl    (%ecx), %ecx\n");
1030        out.write("    addl    $"+(8*(ins.a2-1))+", %ecx\n");
1031
1032        for (int i=0;i<ins.a2;i++){
1033          genPop(out);
1034          //hole argument vom stack
1035          out.write("    movl    sp, %eax\n");
1036          out.write("    sall    $3, %eax\n");
1037          out.write("    addl    stack, %eax\n");
1038
1039          out.write("    movl    (%eax), %edx\n");
1040
1041          out.write("    movl    %edx, (%ecx)\n");
1042    out.write("    addl    $4, %ecx\n");
1043
1044          out.write("    movl    4(%eax), %edx\n");
1045
1046          out.write("    movl    %edx, (%ecx)\n");
1047    out.write("    subl    $12, %ecx\n");
1048        }
1049
1050        out.write("    movl    hp, %ecx\n");
1051        out.write("    decl    %ecx\n");
1052        genPushRef(out,"%ecx"); //die Adresse im Heap drauf
1053        break;
1054
1055      case Unpack:
1056        out.write("    call unpack\n");
1057        break;
1058
1059      case Project:
1060        out.write("  "+mov()+"    $"+ins.a1+", callingArg\n");
1061        out.write("    call project\n");
1062        break;
1063
1064      case Pop:
1065        genPop(out);
1066        break;
1067
1068      case PopTops:
1069        out.write("    call popTops\n");
1070        break;
1071
1072      case PushTop:
1073        out.write("  movl $"+(ins.a1)+", %ebx\n");
1074        out.write("  movl sp, %edx\n");
1075        out.write("  movl %ebx, %eax\n");
1076        out.write("  subl %eax, %edx\n");
1077        out.write("  movl %edx, %eax\n");
1078        out.write("  subl $1, %eax\n");
1079        out.write("  movl %eax, %ebx\n");
```

```
1080        out.write("  movl stack, %eax\n");
1081        out.write("  movl sp, %edx\n");
1082        out.write("  sall $3, %edx\n");
1083        out.write("  addl %eax, %edx\n");
1084        out.write("  movl stack, %eax\n");
1085        out.write("  movl %ebx, %ecx\n");
1086        out.write("  sall $3, %ecx\n");
1087        out.write("  addl %ecx, %eax\n");
1088        out.write("  movl (%eax), %eax\n");
1089        out.write("  movl %eax, (%edx)\n");
1090        out.write("  movl stack, %eax\n");
1091        out.write("  movl sp, %edx\n");
1092        out.write("  sall $3, %edx\n");
1093        out.write("  addl %eax, %edx\n");
1094        out.write("  movl stack, %eax\n");
1095        out.write("  movl %ebx, %ecx\n");
1096        out.write("  sall $3, %ecx\n");
1097        out.write("  addl %ecx, %eax\n");
1098        out.write("  movl 4(%eax), %eax\n");
1099        out.write("  movl %eax, 4(%edx)\n");
1100        out.write("  movl sp, %eax\n");
1101        out.write("  addl $1, %eax\n");
1102        out.write("  movl %eax, sp\n");
1103
1104 //       out.write("  call pushtop\n");
1105        break;
1106
1107     case PushFun:
1108        pushInt(out,"$"+ins.label);
1109        break;
1110
1111     case FunPointerCall:
1112        genPop(out,ax());
1113        out.write("   call   *%eax\n");
1114
1115        break;
1116
1117     }
1118   }
1119 }
```

Listing 10.15: WriteX86.java

Kapitel 11

Bootstrapping... das Münchhausen-Prinzip

```
con Nil()
con Cons(head,tail)

con Pair(fst,snd)

fun not(x)=if x==1 then 0 else 1

fun fst(p)= match p [
  case Pair(a,b) -> a
]

fun snd(p)= match p [
  case Pair(a,b) -> b
]


/*  standard list functions
 */
fun head(xs) = match xs [
  case Cons(y, ys) -> y
]

fun tail(xs) = match xs [
  case Cons(y, ys) -> ys
]

fun drop(n,xs) = if n==0 then xs else match xs[
  case Cons(y,ys) -> drop(n-1,ys)
  case Nil() -> Nil()
]

fun elem(x,xs) = match xs [
  case Nil() -> 0
  case Cons(y, ys) -> if (x==y) then 1 else elem(x,ys)
]

```

```
37 fun elemWith(eq,x,xs) = match xs [
38   case Nil() -> 0
39   case Cons(y, ys) -> if eq(x,y) then 1 else elemWith(eq,x,ys)
40 ]
41
42 fun elemStr(x,xs) = elemWith(strEq,x,xs)
43
44 fun get(i,xs) = match xs[
45   case Cons(y, ys) -> if (i==0) then y else get(i-1,ys)
46 ]
47
48 fun size(xs) = match xs [
49   case Nil() -> 0
50   case Cons(y, ys) -> 1+size(ys)
51 ]
52
53 fun map(f,xs) = match xs [
54   case Nil() -> Nil()
55   case Cons(y, ys) -> Cons(f(y),map(f,ys))
56 ]
57
58 fun append(xs,ys)= match xs [
59   case Nil() -> ys
60   case Cons(z, zs) -> Cons(z,append(zs,ys))
61 ]
62
63 fun reverse(xs) = match xs [
64   case Nil() -> Nil()
65   case Cons(y, ys) -> append(reverse(ys),Cons(y,Nil()))
66 ]
67
68 fun concat(xss) = match xss [
69   case Nil() -> Nil()
70   case Cons(ys, yss) -> append(ys,concat(yss))
71 ]
72
73 fun concatWith(sep,xss) = match xss [
74   case Nil() -> Nil()
75   case Cons(ys, yss) -> append(append(ys,sep),concatWith(sep,yss))
76 ]
77
78 fun strEq(xs,ys)=match xs[
79   case Nil() -> match ys[
80     case Nil() -> 1
81     case Cons(y1,ys1) -> 0
82   ]
83   case Cons(x1,xs1) -> match ys[
84     case Nil() -> 0
85     case Cons(y1,ys1) -> x1==y1 && strEq(xs1,ys1)
86   ]
87 ]
88
```

```
89  con Nothing()
90  con Just(x)
91
92  fun lookup(map,x)=match map[
93    case Nil() -> Nothing()
94    case Cons(p,map1) -> match p[
95      case Pair(key,value) ->
96       if strEq(key,x) then Just(value) else lookup(map1,x)
97    ]
98  ]
99
100 /*here we add column and row number to the characters*/
101 fun numerate(c,r,xs)
102  = match xs [
103     case Nil() -> Nil()
104     case Cons(y, ys) ->
105       if   y!=10
106       then Cons(Pair(y,Pair(c,r)),numerate(c+1,r,ys))
107       else Cons(Pair(y,Pair(c,r)),numerate(1,r+1,ys))
108 ]
109
110
111 /*we provide simple constructors for the token of the language*/
112 /*Keywords*/
113 con ConToken(pos)
114 con FunToken(pos)
115 con IfToken(pos)
116 con ThenToken(pos)
117 con ElseToken(pos)
118 con CaseToken(pos)
119 con MatchToken(pos)
120
121 /*Symbols*/
122 con ArrowToken(pos)
123 con LambdaToken(pos)
124 con LParToken(pos)
125 con RParToken(pos)
126 con LBrackToken(pos)
127 con RBrackToken(pos)
128 con CommaToken(pos)
129 con DotToken(pos)
130 con PlusToken(pos)
131 con MinusToken(pos)
132 con MultToken(pos)
133 con ModToken(pos)
134 con DivToken(pos)
135 con EqToken(pos)
136 con EqEqToken(pos)
137 con LtToken(pos)
138 con GtToken(pos)
139 con LeToken(pos)
140 con GeToken(pos)
```

```
141 con NeqToken(pos)
142 con AndToken(pos)
143 con OrToken(pos)
144
145 /*Literals*/
146 con StringLiteralToken(pos,s)
147 con NumberLiteralToken(pos,i)
148 con IdentToken(pos,s)
149
150 fun tokenValue(tok)= match tok [
151   case ConToken( p) -> "con"
152   case FunToken( p) -> "fun"
153   case IfToken( p) -> "if"
154   case ThenToken( p) -> "then"
155   case ElseToken( p) -> "else"
156   case CaseToken( p) -> "case"
157   case MatchToken(p) -> "match"
158   case ArrowToken( p) -> "->"
159   case LambdaToken( p) -> "\\"
160   case LParToken( p) -> "("
161   case RParToken( p) -> ")"
162   case LBrackToken( p) -> "["
163   case RBrackToken( p) -> "]"
164   case CommaToken( p) -> ","
165   case DotToken( p) -> ".\n"
166   case PlusToken( p) -> "+"
167   case MinusToken( p) -> "-"
168   case MultToken( p) -> "*"
169   case ModToken( p) -> "%"
170   case DivToken( p) -> "/"
171   case EqToken( p) -> "="
172   case EqEqToken( p) -> "=="
173   case NeqToken( p) -> "!="
174   case LeToken( p) -> "<="
175   case GeToken( p) -> ">="
176   case LtToken( p) -> "<"
177   case GtToken( p) -> ">"
178   case AndToken( p) -> "&&"
179   case OrToken( p) -> "||"
180   case StringLiteralToken( p, v) -> append("\"",append(v,"\""))
181   case NumberLiteralToken( p, v) -> v
182   case IdentToken( p, v) -> v
183 ]
184
185 fun tokenPos(tok) = match tok [
186   case ConToken( p) ->p
187   case FunToken( p) ->p
188   case IfToken( p) -> p
189   case ThenToken( p) ->p
190   case ElseToken( p) ->p
191   case CaseToken( p) ->p
192   case MatchToken(p) ->p
```

```
193   case ArrowToken( p) =>p
194   case LambdaToken( p) =>p
195   case LParToken( p) => p
196   case RParToken( p) => p
197   case LBrackToken( p) => p
198   case RBrackToken( p) =>p
199   case CommaToken( p) =>p
200   case DotToken( p) => p
201   case PlusToken( p) => p
202   case MinusToken( p) =>p
203   case MultToken( p) => p
204   case ModToken( p) => p
205   case DivToken( p) => p
206   case EqToken( p) => p
207   case EqEqToken( p) => p
208   case NeqToken( p) => p
209   case LeToken( p) => p
210   case GeToken( p) => p
211   case LtToken( p) => p
212   case GtToken( p) => p
213   case AndToken( p) => p
214   case OrToken( p) => p
215   case StringLiteralToken( p, v) => p
216   case NumberLiteralToken( p, v) => p
217   case IdentToken( p, v) => p
218 ]
219
220 fun tokenTest(t1,t2) = match t1 [
221   case ConToken( p) =>  match t2
222     [case ConToken(p2) => 1 case otherwise => 0]
223   case FunToken( p) =>  match t2
224     [case FunToken(p2)=> 1 case otherwise => 0]
225   case IfToken( p) =>   match t2
226     [case IfToken(p2)=> 1 case otherwise =>0]
227   case ThenToken( p) => match t2
228     [case ThenToken(p2)=> 1 case otherwise =>0]
229   case ElseToken( p) => match t2
230     [case ElseToken(p2)=> 1 case otherwise =>0]
231   case CaseToken( p) => match t2
232     [case CaseToken(p2)=> 1 case otherwise =>0]
233   case MatchToken(p) => match t2
234     [case MatchToken(p2)=> 1 case otherwise =>0]
235   case ArrowToken( p) =>  match t2
236     [case ArrowToken(p2)=> 1 case otherwise =>0]
237   case LambdaToken( p) =>  match t2
238     [case LambdaToken(p2)=> 1 case otherwise =>0]
239   case LParToken( p) =>  match t2
240     [case LParToken(p2)=> 1 case otherwise =>0]
241   case RParToken( p) =>  match t2
242     [case RParToken(p2)=> 1 case otherwise =>0]
243   case LBrackToken( p) => match t2
244     [case LBrackToken(p2)=> 1 case otherwise =>0]
```

```
245   case RBrackToken( p) -> match t2
246     [case RBrackToken(p2)-> 1 case otherwise ->0]
247   case CommaToken( p) ->  match t2
248     [case CommaToken(p2)-> 1 case otherwise ->0]
249   case DotToken( p) ->   match t2
250     [case DotToken(p2)-> 1 case otherwise ->0]
251   case PlusToken( p) ->  match t2
252     [case PlusToken(p2)-> 1 case otherwise ->0]
253   case MinusToken( p) -> match t2
254     [case MinusToken(p2)-> 1 case otherwise ->0]
255   case MultToken( p) ->  match t2
256     [case MultToken(p2)-> 1 case otherwise ->0]
257   case ModToken( p) ->   match t2
258     [case ModToken(p2)-> 1 case otherwise ->0]
259   case DivToken( p) ->   match t2
260     [case DivToken(p2)-> 1 case otherwise ->0]
261   case EqToken( p) ->   match t2
262     [case EqToken(p2)-> 1 case otherwise ->0]
263   case EqEqToken( p) ->  match t2
264     [case EqEqToken(p2)-> 1 case otherwise-> 0]
265   case NeqToken( p) ->  match t2
266     [case NeqToken(p2)-> 1 case otherwise-> 0]
267   case LeToken( p) ->  match t2
268     [case LeToken(p2)-> 1 case otherwise-> 0]
269   case GeToken( p) ->  match t2
270     [case GeToken(p2)-> 1 case otherwise-> 0]
271   case LtToken( p) ->  match t2
272     [case LtToken(p2)-> 1 case otherwise-> 0]
273   case GtToken( p) ->  match t2
274     [case GtToken(p2)-> 1 case otherwise-> 0]
275   case AndToken( p) ->   match t2
276     [case AndToken(p2)-> 1 case otherwise ->0]
277   case OrToken( p) ->   match t2
278     [case OrToken(p2)-> 1 case otherwise ->0]
279   case StringLiteralToken( p, v) ->  match t2
280     [case StringLiteralToken(p2,s)-> 1 case otherwise ->0]
281   case NumberLiteralToken( p, v) ->  match t2
282     [case NumberLiteralToken(p2,s)-> 1 case otherwise ->0]
283   case IdentToken( p, v) ->  match t2
284     [case IdentToken(p2,s)-> 1 case otherwise ->0]
285 ]
286
287 fun mapTokenValue(xs) = match xs [
288   case Nil -> Nil()
289   case Cons( y, ys) -> Cons(tokenValue(y),mapTokenValue(ys))
290 ]
291
292 fun whitespace() = " \t\n"
293 fun isWhite(x) = elem(x,whitespace())
294
295 fun digits() = "0123456789"
296 fun isDigit(x) = elem(x,digits())
```

```
297
298 fun alphas() = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
299 fun isAlpha(x) = elem(x,alphas())
300
301
302 fun tokenize(xs)
303   = match xs [
304       case Cons( y, ys) ->  match y [
305         case Pair( c, pos) ->
306             if isWhite(c)
307           then tokenize(ys)
308           else if (c==head("."))
309           then  Cons(DotToken(pos),tokenize(ys))
310           else if (c==head(","))
311           then  Cons(CommaToken(pos),tokenize(ys))
312           else if (c==head("("))
313           then Cons(LParToken(pos),tokenize(ys))
314           else if (c==head(")"))
315           then Cons(RParToken(pos),tokenize(ys))
316           else if (c==head("["))
317           then Cons(LBrackToken(pos),tokenize(ys))
318           else if (c==head("]"))
319           then Cons(RBrackToken(pos),tokenize(ys))
320           else if (c==head("+"))
321           then Cons(PlusToken(pos),tokenize(ys))
322           else if (c==head("*"))
323           then Cons(MultToken(pos),tokenize(ys))
324           else if (c==head("%"))
325           then Cons(ModToken(pos),tokenize(ys))
326           else if (c==head("\\"))
327           then Cons(LambdaToken(pos),tokenize(ys))
328           else if (c==head("-"))
329           then tokenizeMinus(pos,ys)
330           else if (c==head("="))
331           then tokenizeEq(pos,ys)
332           else if (c==head("<"))
333           then tokenizeLt(pos,ys)
334           else if (c==head(">"))
335           then tokenizeGt(pos,ys)
336           else if (c==head("/"))
337           then tokenizeDiv(pos,ys)
338           else if (c==head("&"))
339           then tokenizeKeyWord(pos,"",AndToken(pos),"&&",xs)
340           else if (c==head("|"))
341           then tokenizeKeyWord(pos,"",OrToken(pos),"||",xs)
342           else if (isDigit(c))
343           then tokenizeDigit(pos,Cons(c,Nil()),ys)
344           else if (c==head("!"))
345           then tokenizeKeyWord(pos,"",NeqToken(pos),"!=",xs)
346           else if (c==head("i"))
347           then tokenizeKeyWord(pos,"",IfToken(pos),"if",xs)
348           else if (c==head("t"))
```

```
            then tokenizeKeyWord(pos,"",ThenToken(pos),"then",xs)
            else if (c==head("e"))
            then tokenizeKeyWord(pos,"",ElseToken(pos),"else",xs)
            else if (c==head("f"))
            then tokenizeKeyWord(pos,"",FunToken(pos),"fun",xs)
            else if (c==head("c"))
            then tokenizeCKeyWord(pos,ys)
            else if (c==head("m"))
            then tokenizeKeyWord(pos,"",MatchToken(pos),"match",xs)
            else if (c==head("\""))
            then tokenizeStringLiteral(pos,"",ys)
            else if (isAlpha(c))
            then tokenizeIdentifier(pos,"",xs)
          else tokenize(ys)
          ]
        case Nil() -> Nil()
]


fun tokenizeStringLiteral(pos,found,xs) = match xs [
  case Nil() -> Nil()
  case Cons( y, ys) -> match y [
    case Pair( c, p) ->
      if   head("\"")==c
      then Cons(StringLiteralToken(pos,reverse(found)),tokenize(ys))
      else if head("\\a")==c
      then tokenizeStringLiteral(pos,Cons(fst(head(ys)),Cons(c,found))
    ,tail(ys))
      else tokenizeStringLiteral(pos,Cons(c,found),ys)
   ]
]


fun tokenizeCKeyWord(pos,xs) = match xs [
  case Nil() -> Cons(IdentToken(pos,"c"),Nil())
  case Cons(y, ys) -> match y [
    case Pair( c, p) ->
      if   head("o") == c
      then tokenizeKeyWord(pos,"oc",ConToken(pos),"n",ys)
      else if head("a") == c
      then tokenizeKeyWord(pos,"ac",CaseToken(pos),"se",ys)
      else tokenizeIdentifier(pos,"c",xs)
  ]
]

fun tokenizeMinus(pos,xs)= match xs [
  case Nil() -> Cons(MinusToken(pos),Nil())
  case Cons( y, ys) -> match y [
    case Pair( c, p)  ->
      if (c==head(">"))
      then Cons(ArrowToken(pos),tokenize(ys))
      else Cons(MinusToken(pos),tokenize(xs))
```

```
  ]
]

fun tokenizeEq(pos,xs)= match xs [
  case Nil() -> Cons(EqToken(pos),Nil())
  case Cons( y, ys) -> match y [
    case Pair( c, p)  ->
      if (c==head("="))
      then Cons(EqEqToken(pos),tokenize(ys))
      else Cons(EqToken(pos),tokenize(xs))
  ]
]

fun tokenizeLt(pos,xs)= match xs [
  case Nil() -> Cons(LtToken(pos),Nil())
  case Cons( y, ys) -> match y [
    case Pair( c, p)  ->
      if (c==head("="))
      then Cons(LeToken(pos),tokenize(ys))
      else Cons(LtToken(pos),tokenize(xs))
  ]
]

fun tokenizeGt(pos,xs)= match xs [
  case Nil() -> Cons(GtToken(pos),Nil())
  case Cons( y, ys) -> match y [
    case Pair( c, p)  ->
      if (c==head("="))
      then Cons(GeToken(pos),tokenize(ys))
      else Cons(GtToken(pos),tokenize(xs))
  ]
]

fun tokenizeDiv(pos,xs)= match xs [
  case Nil() -> Cons(DivToken(pos),Nil())
  case Cons( y, ys) -> match y [
    case Pair( c, p)  ->
      if (c == head("*"))
      then tokenizeComment(pos,ys)
      else Cons(DivToken(pos),tokenize(xs))
  ]
]

fun tokenizeComment(pos,xs) = match xs [
  case Cons( y, ys) -> match y [
    case Pair( c, p) ->
        if (c == head("*"))
      then tokenizeCommentEnd(pos,ys)
      else tokenizeComment(pos,ys)
  ]
]

```

```
452 fun tokenizeCommentEnd(pos,xs) = match xs [
453   case Cons( y, ys) -> match y [
454     case Pair( c, p) ->
455         if (c == head("/"))
456       then tokenize(ys)
457       else tokenizeComment(pos,xs)
458   ]
459 ]
460
461 fun tokenizeDigit(pos,is,xs)= match xs [
462   case Nil() -> Cons(NumberLiteralToken(pos,reverse(is)),Nil())
463   case Cons( y, ys) -> match y [
464     case Pair( c, p) -> if isDigit(c)
465               then tokenizeDigit(pos,Cons(c,is),ys)
466               else Cons(NumberLiteralToken(pos,reverse(is)),tokenize(
    xs))
467   ]
468 ]
469
470 fun tokenizeKeyWord(pos,found,resultToken,key,xs) = match key [
471   case Nil() -> match xs [
472     case Nil() -> Cons(resultToken,Nil())
473     case Cons( y, ys) -> match y [
474       case Pair( c, p) -> if isAlpha(c)
475                 then tokenizeIdentifier(pos,found,xs)
476                 else Cons(resultToken,tokenize(xs))
477     ]
478   ]
479   case Cons( k, ey) -> match xs [
480     case Nil() -> tokenizeIdentifier(pos,found,xs)
481     case Cons( y, ys) -> match y [
482       case Pair( c, p) -> if (k==c)
483                 then tokenizeKeyWord(pos,Cons(k,found),resultToken,ey,
    ys)
484                 else tokenizeIdentifier(pos,found,xs)
485     ]
486
487   ]
488 ]
489
490 fun tokenizeIdentifier(pos,found,xs) = match xs [
491   case Nil() -> Cons(IdentToken(pos,reverse(found)),Nil())
492   case Cons( y, ys) -> match y [
493     case Pair( c, p) -> if isAlpha(c) || isDigit(c)
494               then tokenizeIdentifier(pos,Cons(c,found),ys)
495               else Cons(IdentToken(pos,reverse(found)),tokenize(xs))
496   ]
497 ]
498
499
500 /*PARSER*/
501
```

```
502 con Fail(reason)
503 con Success(result)
504
505
506 fun pToken(token,xs)=match xs [
507   case Nil()-> Fail("end of token stream")
508   case Cons(y,ys) ->
509    if tokenTest(token, y)
510    then Success(Pair(y,ys))
511    else Fail(append("expected Token: ", tokenValue(token)))
512 ]
513
514 fun pConT(xs) = pToken(ConToken(Pair(0,0)),xs)
515 fun pFunT(xs) = pToken(FunToken(Pair(0,0)),xs)
516 fun pIfT(xs) = pToken(IfToken(Pair(0,0)),xs)
517 fun pThenT(xs) = pToken(ThenToken(Pair(0,0)),xs)
518 fun pElseT(xs) = pToken(ElseToken(Pair(0,0)),xs)
519 fun pMatchT(xs) = pToken(MatchToken(Pair(0,0)),xs)
520 fun pCaseT(xs) = pToken(CaseToken(Pair(0,0)),xs)
521
522 fun pLParT(xs) = pToken(LParToken(Pair(0,0)),xs)
523 fun pRParT(xs) = pToken(RParToken(Pair(0,0)),xs)
524 fun pLBrackT(xs) = pToken(LBrackToken(Pair(0,0)),xs)
525 fun pRBrackT(xs) = pToken(RBrackToken(Pair(0,0)),xs)
526 fun pArrowT(xs) = pToken(ArrowToken(Pair(0,0)),xs)
527 fun pLambdaT(xs) = pToken(LambdaToken(Pair(0,0)),xs)
528 fun pCommaT(xs) = pToken(CommaToken(Pair(0,0)),xs)
529 fun pEqT(xs) = pToken(EqToken(Pair(0,0)),xs)
530 fun pEqEqT(xs) = pToken(EqEqToken(Pair(0,0)),xs)
531 fun pNeqT(xs) = pToken(NeqToken(Pair(0,0)),xs)
532 fun pGeT(xs) = pToken(GeToken(Pair(0,0)),xs)
533 fun pLeT(xs) = pToken(LeToken(Pair(0,0)),xs)
534 fun pGtT(xs) = pToken(GtToken(Pair(0,0)),xs)
535 fun pLtT(xs) = pToken(LtToken(Pair(0,0)),xs)
536 fun pAndT(xs) = pToken(AndToken(Pair(0,0)),xs)
537 fun pOrT(xs) = pToken(OrToken(Pair(0,0)),xs)
538 fun pPlusT(xs) = pToken(PlusToken(Pair(0,0)),xs)
539 fun pMinusT(xs) = pToken(MinusToken(Pair(0,0)),xs)
540 fun pMultT(xs) = pToken(MultToken(Pair(0,0)),xs)
541 fun pDivT(xs) = pToken(DivToken(Pair(0,0)),xs)
542 fun pModT(xs) = pToken(ModToken(Pair(0,0)),xs)
543
544 fun pIdentT(xs) = pToken(IdentToken(Pair(0,0),"IDENTIFIER"),xs)
545
546 fun pNumberLitT(xs) = pToken(NumberLiteralToken(Pair(0,0),"42"),xs)
547 fun pStringLitT(xs) = pToken(StringLiteralToken(Pair(0,0),"STRING"),xs
    )
548
549 fun seq(p1,p2,xs)=match p1(xs) [
550   case Success(succ)->match succ [
551     case Pair(result,rest) -> match p2(rest)[
552       case Fail(reason)->Fail(reason)
```

```
553       case Success(succ2)->match succ2 [
554         case Pair(result2,rest2) -> Success(Pair(Pair(result,result2),
    rest2))
555       ]
556     ]
557   ]
558   case Fail(ff) -> Fail(ff)
559 ]
560
561 fun seqs(ps,xs)=seqsAcc(ps,xs,Nil())
562 fun seqsAcc(ps,xs,result)=match ps[
563   case Nil()->Success(Pair(reverse(result),xs))
564   case Cons(p,further)-> match p(xs)[
565     case Success(succ) -> match succ [
566       case Pair(r1,rest) -> seqsAcc(further,rest,Cons(r1,result))
567     ]
568     case fail -> fail
569   ]
570 ]
571
572
573 fun alt(p1,p2,xs)=match p1(xs) [
574   case Fail(reason)->p2(xs)
575   case result -> result
576 ]
577
578 fun alts(ps,xs)=match ps [
579   case Nil() -> Fail("no matching parser alternative")
580   case Cons(p,further) -> match p(xs) [
581     case Fail(reason) -> alts(further,xs)
582     case result -> result
583   ]
584 ]
585
586
587 fun zeroOrMore(p,xs) = match p(xs) [
588   case Fail(reason1) -> Success(Pair(Nil(),xs))
589   case Success(succ) -> match succ [
590     case Pair(r1, toks) -> match zeroOrMore(p,toks) [
591       case Success(succrest) -> match succrest [
592         case Pair(result,restToken) -> Success(Pair(Cons(r1,result),
    restToken))
593       ]
594       case Fail(reason2) -> Success(Pair(Cons(r1,Nil()),toks))
595     ]
596   ]
597 ]
598
599 fun oneOrMore(p,xs) = match p(xs) [
600   case Fail(reason1) -> Fail(reason1)
601   case Success(succ) -> match succ [
602     case Pair(r1, toks) -> match zeroOrMore(p,toks) [
```

```
603       case Success(succrest) -> match succrest [
604         case Pair(result,restToken) -> Success(Pair(Cons(r1,result),
    restToken))
605       ]
606       case Fail(reason2) -> Success(Pair(Cons(r1,Nil()),toks))
607     ]
608   ]
609 ]
610
611
612
613 fun mapSuccess(f,sf) = match sf [
614   case Success(succ) -> match succ [
615     case Pair(result,rest) -> Success(Pair(f(result),rest))
616   ]
617   case fail -> fail
618 ]
619
620 fun pCommaIdent(xs) = seq(pCommaT,pIdentT,xs)
621 fun pCommaIdentArgs(xs)=zeroOrMore(pCommaIdent,xs)
622
623 fun pArgs(xs) = sepList(pCommaT,pIdentT,xs)
624 fun pParArgs(xs) = pInPar(pArgs,xs)
625
626
627 fun pInPar(p,xs) =
628  mapSuccess(\ (r)-> get(1,r),seqs(Cons(pLParT,Cons(p,Cons(pRParT,Nil()
    ))),xs))
629
630 fun sepList(sep,p,xs)=match p(xs) [
631   case Fail(ff) -> Success(Pair(Nil(),xs))
632   case Success(succ)-> match succ[
633     case Pair(res,toks) -> match sepListAux(sep,p,toks)[
634       case Success(su) -> match su [
635         case Pair(result,rest) -> Success(Pair(Cons(res,result),rest))
636       ]
637       case Fail(ff) -> Success(Pair(Nil(),xs))
638     ]
639   ]
640 ]
641
642 fun sepListAux(sep,p,xs)=match seq(sep,p,xs)[
643   case Fail(ff) -> Success(Pair(Nil(),xs))
644   case Success(succ) -> match succ [
645     case Pair(res,toks) -> match res [
646       case Pair(resSep,resP) -> match sepListAux(sep,p,toks)[
647         case Success(resultToken)-> match resultToken [
648           case Pair(result,token) -> Success(Pair(Cons(resP,result),
    token))
649         ]
650         case Fail(ff) -> Success(Pair(Cons(resP,Nil()),xs))
651       ]
```

```
652     ]
653   ]
654 ]
655
656
657 /*Abstrakter Syntax Baum */
658 con ConDef(pos,name,args)
659 con FunDef(pos,name,args,def)
660 con SingleCalledFunDef(pos,name,args,def)
661 con LetExpr(pos,defs,expr)
662 con MatchExpr(pos,expr,cases)
663 con CaseExpr(pos,name,args,expr)
664 con VarCaseExpr(pos,name,expr)
665 con ProjectionExpr(pos,expr,name)
666 con LambdaExpr(pos,args,exp)
667 con FunCall(pos,name,args)
668 con IntLit(pos, i)
669 con StringLit(pos,s)
670 con MultExpr(pos,op1,op2)
671 con DivExpr(pos,op1,op2)
672 con ModExpr(pos,op1,op2)
673 con AddExpr(pos,op1,op2)
674 con SubExpr(pos,op1,op2)
675 con EqExpr(pos,op1,op2)
676 con NeqExpr(pos,op1,op2)
677 con LeExpr(pos,op1,op2)
678 con GeExpr(pos,op1,op2)
679 con LtExpr(pos,op1,op2)
680 con GtExpr(pos,op1,op2)
681 con OrExpr(pos,op1,op2)
682 con AndExpr(pos,op1,op2)
683 con IfExpr(pos,cond,a1,a2)
684 con Var(pos,name)
685 con Prog(cons,funs)
686
687 fun getTreePosition(tree) = match tree[
688   case ConDef(pos,name,args) -> pos
689   case FunDef(pos,name,args,def) -> pos
690   case SingleCalledFunDef(pos,name,args,def) -> pos
691   case LetExpr(pos,defs,expr) -> pos
692   case MatchExpr(pos,expr,cases) -> pos
693   case CaseExpr(pos,name,args,expr) -> pos
694   case VarCaseExpr(pos,name,expr) -> pos
695   case ProjectionExpr(pos,expr,name) -> pos
696   case LambdaExpr(pos,args,exp) -> pos
697   case FunCall(pos,name,args) -> pos
698   case IntLit(pos, i) -> pos
699   case StringLit(pos,s) -> pos
700   case MultExpr(pos,op1,op2) -> pos
701   case DivExpr(pos,op1,op2) -> pos
702   case ModExpr(pos,op1,op2) -> pos
703   case AddExpr(pos,op1,op2) -> pos
```

```
704   case SubExpr(pos,op1,op2) -> pos
705   case EqExpr(pos,op1,op2) -> pos
706   case NeqExpr(pos,op1,op2) -> pos
707   case LeExpr(pos,op1,op2) -> pos
708   case GeExpr(pos,op1,op2) -> pos
709   case LtExpr(pos,op1,op2) -> pos
710   case GtExpr(pos,op1,op2) -> pos
711   case OrExpr(pos,op1,op2) -> pos
712   case AndExpr(pos,op1,op2) -> pos
713   case IfExpr(pos,cond,a1,a2) -> pos
714   case Var(pos,name) -> pos
715   case Prog(cons,funs) -> Pair(0,0)
716 ]
717
718 fun mkProg(defs)=mkProgAux(defs,Nil(),Nil())
719
720 fun mkProgAux(defs,cons,funs) = match defs [
721   case Nil() ->  Prog(reverse(cons),reverse(funs))
722   case Cons(def,defs2) -> match def [
723     case ConDef(p,n,args) ->  mkProgAux(defs2,Cons(def,cons),funs)
724     case FunDef(p,n,args,e) -> mkProgAux(defs2,cons,Cons(def,funs))
725     case SingleCalledFunDef(p,n,args,e) -> mkProgAux(defs2,cons,Cons(
    def,funs))
726   ]
727 ]
728
729 fun pProg(xs)=mapSuccess(mkProg ,zeroOrMore(pFunOrConDef,xs))
730
731 fun pFunOrConDef(xs)=alt(pFunDef,pConDef,xs)
732
733 fun pConDef(xs) =
734     mkConDef( seqs
735       (Cons(pConT
736       ,Cons(pIdentT
737       ,Cons(pLParT
738       ,Cons(pArgs
739       ,Cons(pRParT,Nil())))))),xs))
740
741 fun mkConDef(parseResult)=mapSuccess(\ (result)-> match head(tail(
    result)) [
742   case IdentToken(pos,s) -> ConDef
743       (pos
744       ,s
745       ,map(\ (x)->match x [case IdentToken(p,ident)->ident],get(3,
    result))
746       )
747   ],parseResult)
748
749 fun pFunDef(xs) =
750   mkFunDef( seqs
751      (Cons(pFunT
752      ,Cons(pIdentT
```

```
753       ,Cons(pLParT
754       ,Cons(pArgs
755       ,Cons(pRParT
756       ,Cons(pEqT
757       ,Cons(pExpression,Nil()))))))),xs))
758
759 fun mkFunDef(parseResult)=mapSuccess(\ (result)-> match head(tail(
      result)) [
760   case IdentToken(pos,s) -> FunDef
761       (pos
762       ,s
763       ,map(\ (x)->match x [case IdentToken(p,ident)->ident],get(3,
      result))
764       ,get(6,result))
765   ],parseResult)
766
767
768 fun pExpression(xs) =
769  alts(Cons(pAndExpr
770       ,Cons(pIfExpr
771       ,Cons(pLambdaExpr
772       ,Cons(pMatchExpr
773       ,Cons(pLetExpr,Nil()))))),xs)
774
775 fun pIfExpr(xs)=
776   mapSuccess
777     (\(r)->IfExpr(tokenPos(get(0,r)),get(1,r),get(3,r),get(5,r))
778     ,seqs(Cons(pIfT,Cons(pExpression
779          ,Cons(pThenT,Cons(pExpression
780          ,Cons(pElseT,Cons(pExpression,Nil()))))),xs)
781     )
782
783 fun pMatchExpr(xs)=
784   mapSuccess
785     (\(r)->MatchExpr(tokenPos(get(0,r)),get(1,r),get(3,r))
786    ,seqs(Cons(pMatchT,Cons( pExpression,Cons( pLBrackT
787         ,Cons( pCases,Cons( pRBrackT,Nil()))))),xs)
788   )
789
790 fun pCases(xs) = oneOrMore(pCase,xs)
791
792 fun pCase(xs) =
793   mapSuccess
794     ( \ (r)->
795       if    isVarPattern(get(1,r))
796       then VarCaseExpr(tokenPos(get(0,r)),getPatternName(get(1,r)),get
      (3,r))
797       else CaseExpr(tokenPos(get(0,r))
798                    ,getPatternName(get(1,r))
799                    ,getPatternArgs(get(1,r))
800                    ,get(3,r))
```

```
801     , seqs(Cons(pCaseT,Cons(pPattern,Cons(pArrowT,Cons(pExpression,Nil
        ()))))),xs))
802
803 fun pPattern(xs) = match pIdentT(xs)[
804   case Fail(f) -> Fail(f)
805   case Success(succ1) -> match succ1 [
806     case Pair(ident,rest) -> match ident [
807       case IdentToken(p,n) -> match  pInPar(pArgs,rest)[
808         case Fail(f2) -> Success(Pair(VarPattern(n),rest))
809         case Success(succ2) -> match succ2 [
810           case Pair(args,rest2) -> Success(Pair(ConPattern(n,map(\(x)
        ->match x[case IdentToken(pp,v)->v],args)),rest2))
811         ]
812       ]
813     ]
814   ]
815 ]
816
817 con VarPattern(n)
818 con ConPattern(n,args)
819
820 fun isVarPattern(p)=match p [
821   case VarPattern(n) -> 1
822   case otherwise -> 0
823 ]
824 fun getPatternName(p)=match p [
825   case VarPattern(n) -> n
826   case ConPattern(n,agrs) -> n
827 ]
828 fun getPatternArgs(p) = match p [case ConPattern(n,args) -> args]
829
830 fun pLambdaExpr(xs)=
831   mapSuccess
832     (\(r) -> LambdaExpr(tokenPos(get(0,r)),map(tokenValue,get(1,r)),get
        (3,r))
833     ,seqs(Cons(pLambdaT,Cons(pParArgs,Cons(pArrowT,Cons(pExpression,Nil
        ()))))),xs)
834     )
835
836 fun pLetExpr(xs)= pVar(xs)
837
838
839 fun pOperatorExpression(pOperand,pSeqOperatorOperand,xs) = match
        pOperand(xs)[
840   case Fail(f1) -> Fail(f1)
841   case Success(succ1) -> match succ1 [
842     case Pair(op1,rest1) -> match zeroOrMore(pSeqOperatorOperand,rest1
        ) [
843       case Fail(f2) -> Fail(f2)
844       case Success(succ2) -> match succ2[
845         case Pair(ops,rest2) -> Success(Pair(buildOps(op1,ops),rest2))
846       ]
```

```
    ]
  ]
]

fun buildOps(op,ops)= match ops [
  case Nil() -> op
  case Cons(op2,ops2) -> match op2 [
    case Pair(operator,operand) -> match operator [
      case AndToken(p) -> buildOps(AndExpr(p,op,operand),ops2)
      case OrToken(p) -> buildOps(OrExpr(p,op,operand),ops2)
      case EqEqToken(p) -> buildOps(EqExpr(p,op,operand),ops2)
      case NeqToken(p) -> buildOps(NeqExpr(p,op,operand),ops2)
      case GeToken(p) -> buildOps(GeExpr(p,op,operand),ops2)
      case LeToken(p) -> buildOps(LeExpr(p,op,operand),ops2)
      case GtToken(p) -> buildOps(GtExpr(p,op,operand),ops2)
      case LtToken(p) -> buildOps(LtExpr(p,op,operand),ops2)
      case PlusToken(p) -> buildOps(AddExpr(p,op,operand),ops2)
      case MinusToken(p) -> buildOps(SubExpr(p,op,operand),ops2)
      case MultToken(p) -> buildOps(MultExpr(p,op,operand),ops2)
      case DivToken(p) -> buildOps(DivExpr(p,op,operand),ops2)
      case ModToken(p) -> buildOps(ModExpr(p,op,operand),ops2)
    ]
  ]
]

fun pAndOps(xs)=alts(Cons(pAndT,Nil()),xs)
fun pAndExpr(xs) =
 pOperatorExpression(pOrExpr,\(ys)->seq(pAndOps,pOrExpr,ys),xs)

fun pOrOps(xs)=alts(Cons(pOrT,Nil()),xs)
fun pOrExpr(xs) =
 pOperatorExpression(pCompExpr,\(ys)->seq(pOrOps,pCompExpr,ys),xs)

fun pCompOps(xs)=
  alts(Cons(pLtT
       ,Cons(pGtT
       ,Cons(pLeT
       ,Cons(pGeT
       ,Cons(pNeqT
       ,Cons(pEqEqT,Nil()))))))),xs)
fun pCompExpr(xs) =
 pOperatorExpression(pAddExpr,\(ys)->seq(pCompOps,pAddExpr,ys),xs)

fun pAddOps(xs)=alts(Cons(pMinusT,Cons(pPlusT,Nil())),xs)
fun pAddExpr(xs) =
 pOperatorExpression(pMultExpr,\(ys)->seq(pAddOps,pMultExpr,ys),xs)

fun pMultOps(xs)=alts(Cons(pModT,Cons(pMultT,Cons(pDivT,Nil()))),xs)
fun pMultExpr(xs) =
 pOperatorExpression(pUnaryExpr,\(ys)->seq(pMultOps,pUnaryExpr,ys),xs)

fun pUnaryExpr(xs) =
```

```
899   alts(Cons(pIntLit
900        ,Cons(pStringLit
901        ,Cons(pParExpr
902        ,Cons(pFunCallOrVar,Nil()))))),xs)
903
904 fun pFunCallOrVar(xs) = match pIdentT(xs)[
905   case Fail(f) -> Fail(f)
906   case Success(succ) -> match succ [
907     case Pair(vt,rest) -> match vt [
908       case IdentToken(p,v) -> match pFunCallArgs(rest)[
909         case Fail(f2) -> Success(Pair(Var(p,v),rest))
910         case Success(succ2) -> match succ2[
911           case Pair(args,rest2) -> Success(Pair(FunCall(p,v,args),
    rest2))
912         ]
913       ]
914     ]
915   ]
916 ]
917
918 fun pFunArgs(xs) = sepList(pCommaT,pExpression,xs)
919
920 fun pFunCallArgs(xs) = pInPar(pFunArgs,xs)
921
922 fun pVar(xs) = mapSuccess
923   ( \(r)-> match r [case IdentToken(pos,name)->Var(pos,name)]
924   , pIdentT(xs))
925
926 fun pIntLit(xs) = mapSuccess
927   ( \(r)-> match r [case NumberLiteralToken(pos,name)->IntLit(pos,name
    )]
928   , pNumberLitT(xs))
929
930 fun pStringLit(xs) = mapSuccess
931   ( \(r)-> match r [case StringLiteralToken(pos,name)->StringLit(pos,
    name)]
932   , pStringLitT(xs))
933
934 fun pParExpr(xs) = pInPar(pExpression,xs)
935
936 /*Maschinenbefehle */
937 con PushInt()
938 con Mult()
939 con Div()
940 con Mod()
941 con Add()
942 con Sub()
943 con Eq()
944 con Neq()
945 con Lt()
946 con Gt()
947 con Le()
```

```
948 con Ge()
949 con And()
950 con Or()
951 con JmpTrue()
952 con JmpGt()
953 con Jmp()
954 con Label()
955 con GlobLabel()
956 con Ret()
957 con Push()
958 con StartCall()
959 con Call()
960 con PushFun()
961 con FunPointerCall()
962 con Pack()
963 con PackReverse()
964 con Unpack()
965 con Pop()
966 con PopTops()
967 con PushTop()
968 con Project()
969 con PushChar()
970
971 con Instruction(ins,a1,a2)
972
973
974 fun genLabel(prefix, pos)=match pos[
975   case Pair(c,r) ->
976    append(prefix,append("_",append(show(r),append("_",append(show(c),"
     _")))))
977 ]
978
979 fun genCode(prog) = match prog [
980   case Prog(cs,fs) ->
981    concat
982      (genCodeProg
983        (fs
984        ,Nil()
985        ,map(\ (f)->match f [case FunDef(p,n,args,e)->Pair(n,f) case
     SingleCalledFunDef(p,n,args,e)->Pair(n,f)],fs)
986        ,map(\ (c)->match c [case ConDef(p,n,args)->Pair(n,c)],cs)
987        ,map(\ (p)->match p[
988          case Pair(nr,c)
989           ->match c [case ConDef(p,n,args)->Pair(n,nr)]],giveNumbers
     (0,cs))
990        )
991      )
992 ]
993
994 fun genCodeProg( fs, varEnv, funs, cons, conNumbers  )
995  = map(reverse,mapGenCodeAux(varEnv,funs,cons,conNumbers,fs))
996
```

```
997 fun mapGenCodeAux(varEnv,funs,cons,conNumbers,fs)= match fs[
998   case Nil() -> Nil()
999   case Cons(f,fs2) ->
1000      Cons(genCodeAux(f,varEnv,funs,cons,conNumbers)
1001           ,mapGenCodeAux(varEnv,funs,cons,conNumbers,fs2))
1002 ]
1003
1004 fun genCodeAux
1005   ( tree
1006   , varEnv      /* Map<String,int>*/
1007   , funs        /* Map<String,FunDef>*/
1008   , cons        /* Map<String,ConDef>*/
1009   , conNumbers /* Map<String,int>*/
1010   )
1011  =  match tree [
1012   case IntLit(p,i) -> Cons(Instruction(PushInt(),i,0),Nil())
1013
1014   case AddExpr(p,op1,op2) ->
1015     Cons(Instruction(Add(),0,0),
1016     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1017     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1018
1019   case MultExpr(p,op1,op2) ->
1020     Cons(Instruction(Mult(),0,0),
1021     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1022     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1023
1024   case DivExpr(pos,op1,op2)->
1025     Cons(Instruction(Div(),0,0),
1026     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1027     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1028
1029   case ModExpr(pos,op1,op2)->
1030     Cons(Instruction(Mod(),0,0),
1031     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1032     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1033
1034   case SubExpr(pos,op1,op2)->
1035     Cons(Instruction(Sub(),0,0),
1036     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1037     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1038
1039   case EqExpr(pos,op1,op2)->
1040     Cons(Instruction(Eq(),0,0),
1041     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1042     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1043
1044   case NeqExpr(pos,op1,op2)->
1045     Cons(Instruction(Neq(),0,0),
1046     append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1047     genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1048
```

```
1049    case LeExpr(pos,op1,op2)->
1050      Cons(Instruction(Le(),0,0),
1051      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1052      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1053
1054    case GeExpr(pos,op1,op2)->
1055      Cons(Instruction(Ge(),0,0),
1056      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1057      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1058
1059    case LtExpr(pos,op1,op2)->
1060      Cons(Instruction(Lt(),0,0),
1061      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1062      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1063
1064    case GtExpr(pos,op1,op2)->
1065      Cons(Instruction(Gt(),0,0),
1066      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1067      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1068
1069    case OrExpr(pos,op1,op2)->
1070      Cons(Instruction(Or(),0,0),
1071      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1072      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1073
1074    case AndExpr(pos,op1,op2)->
1075      Cons(Instruction(And(),0,0),
1076      append(genCodeAux(op2,varEnv,funs,cons,conNumbers),
1077      genCodeAux(op1,varEnv,funs,cons,conNumbers)))
1078
1079    case FunDef(p,n,args,def) ->
1080     Cons(Instruction(Ret(),size(args),0),
1081     append(genCodeAux
1082       ( def
1083       , map( \(p)->match p[case Pair(e1,e2)->Pair(e2,e1)]
1084            , giveNumbers(1,reverse(args)))
1085       , funs, cons, conNumbers),
1086     Cons(Instruction(StartCall(),0,0),
1087     Cons(Instruction(GlobLabel(),0,n),
1088     Nil()))))
1089
1090
1091    case SingleCalledFunDef(p,n,args,def) ->
1092     Cons(Instruction(Ret(),size(args),0),
1093     append(genCodeAux
1094       ( def
1095       , map( \(p)->match p[case Pair(e1,e2)->Pair(e2,e1)]
1096            , giveNumbers(1,reverse(args)))
1097       , funs, cons, conNumbers),
1098     Cons(Instruction(StartCall(),0,0),
1099     Cons(Instruction(Label(),0,n),
1100     Nil()))))
```

```
1101
1102  case FunCall(p,n,args) ->
1103    match concat(mapGenCodeAux(varEnv,funs,cons,conNumbers,reverse(
      args)))[
1104    case argsCode -> match lookup(varEnv,n)[
1105      case Just(vS) ->
1106        Cons(Instruction(FunPointerCall(),0,0),
1107        Cons(Instruction(Push(),vS,0),
1108        argsCode))
1109
1110      case Nothing() -> match lookup(conNumbers,n)[
1111        case Nothing() -> Cons(Instruction(Call(),0,n),argsCode)
1112        case Just(conNr) ->
1113          Cons(Instruction(Pack(),conNr,size(args)),
1114          Cons(Instruction(PushInt(),show(size(args)),0),
1115          argsCode))
1116      ]
1117    ]
1118  ]
1119
1120  case Var(p,v) -> match lookup(varEnv,v)[
1121    case Nothing() -> Cons(Instruction(PushFun(),0,v),Nil())
1122    case Just(vS) -> Cons(Instruction(Push(),vS,0),Nil())
1123  ]
1124
1125  case IfExpr(p,cond,a1,a2) -> match genLabel("endthen",p)[
1126    case endIfLabel -> match genLabel("then",p)[
1127      case thenLabel ->
1128        Cons(Instruction(Label(),0,endIfLabel),
1129        append(genCodeAux(a1,varEnv,funs,cons,conNumbers),
1130        Cons(Instruction(Label(),0,thenLabel),
1131        Cons(Instruction(Jmp(),0,endIfLabel),
1132        append(genCodeAux(a2,varEnv,funs,cons,conNumbers),
1133        Cons(Instruction(JmpTrue(),0,thenLabel),
1134        genCodeAux(cond,varEnv,funs,cons,conNumbers)))))))
1135    ]
1136  ]
1137
1138  case MatchExpr(p,expr,cases) ->
1139       match genCodeAux(expr,varEnv,funs,cons,conNumbers)[
1140    case exprCode -> match genLabel("endMatch",p)[
1141      case endMatch ->
1142        Cons(Instruction(Label(),0,endMatch),
1143        append(concat(
1144          genCodeCases(reverse(cases),endMatch ,varEnv,funs,cons,
      conNumbers)),
1145        Cons(Instruction(Unpack(),0,0),
1146        Cons(Instruction(PushTop(),0,0),
1147        exprCode))))
1148    ]
1149  ]
1150
```

```
1151   case VarCaseExpr(pos,name,expr) -> match expr[
1152     case FunCall(p,n,as) -> match concat(mapGenCodeAux(varEnv,funs,
       cons,conNumbers,reverse(tail(as))))[
1153       case argsCode ->
1154         Cons(Instruction(Label(),0,append(n,"callReturn")),
1155         Cons(Instruction(Jmp(),0,name),
1156         argsCode))
1157     ]
1158     case otherwise -> Nil()
1159   ]
1160
1161   case CaseExpr(pos,name,args,expr) ->  match expr [
1162     case FunCall(p,n,as) -> match concat(mapGenCodeAux(varEnv,funs,
       cons,conNumbers, reverse(drop(1+size(args),as))))[
1163       case argsCode ->
1164         Cons(Instruction(Label(),0,append(n,"callReturn")),
1165         Cons(Instruction(Jmp(),0,name),
1166         argsCode))
1167     ]
1168     case otherwise -> Nil()
1169   ]
1170
1171   case otherwise -> Nil()
1172 ]
1173
1174 fun genCodeCases(cases,endMatch ,varEnv,funs,cons,conNumbers)=match
       cases[
1175   case Nil() -> Nil()
1176   case Cons(cas,cases2) ->  match cas[
1177     case VarCaseExpr(pos,name,expr) ->
1178      Cons(
1179         Cons(Instruction(Jmp(),0,endMatch),
1180         append(genCodeAux(cas,varEnv,funs,cons,conNumbers),
1181         Cons(Instruction(PopTops(),0,0),
1182         Cons(Instruction(Pop(),0,0),Nil()))))
1183        ,genCodeCases(cases2,endMatch ,varEnv,funs,cons,conNumbers)
1184        )
1185     case CaseExpr(pos,name,args,expr) -> match  genLabel("endCase",pos
       )[
1186       case endCase -> match lookup(conNumbers,name) [
1187         case Nothing() -> Nil()
1188         case Just(conNr) ->
1189          Cons(
1190           Cons(Instruction(Label(),0,endCase),
1191           Cons(Instruction(Jmp(),0,endMatch),
1192           append(genCodeAux(cas,varEnv,funs,cons,conNumbers),
1193           Cons(Instruction(Pop(),0,0),
1194           Cons(Instruction(Pop(),0,0),
1195           Cons(Instruction(JmpTrue(),0,endCase),
1196           Cons(Instruction(Neq(),0,0),
1197           Cons(Instruction(PushInt(),show(conNr),0),
1198           Cons(Instruction(PushTop(),0,0),
```

```
1199            Nil()))))))))))),
1200          genCodeCases(cases2,endMatch  ,varEnv,funs,cons,conNumbers) )
1201
1202       ]
1203     ]
1204   ]
1205 ]
1206
1207 fun collectFreeVarsList(boundVars,conNames,funNames,xs)=match xs[
1208   case Cons(y,ys) ->
1209     append(collectFreeVars(boundVars,conNames,funNames,y)
1210           ,collectFreeVarsList(boundVars,conNames,funNames,ys))
1211   case Nil() -> Nil()
1212 ]
1213
1214
1215
1216 fun collectFreeVars(boundVars,conNames,funNames,tree) = match tree[
1217   case ConDef(pos,name,args) -> Nil()
1218   case FunDef(pos,name,args,def) -> Nil()
1219   case SingleCalledFunDef(pos,name,args,def) -> Nil()
1220   case LetExpr(pos,defs,expr) -> Nil()
1221   case MatchExpr(pos,expr,cases) ->
1222      append(collectFreeVars(boundVars,conNames,funNames,expr)
1223            ,collectFreeVarsList(boundVars,conNames,funNames,cases))
1224
1225   case CaseExpr(pos,name,args,expr) ->
1226     collectFreeVars(append(args,boundVars),conNames,funNames,expr)
1227
1228   case VarCaseExpr(pos,name,expr) ->
1229     collectFreeVars(Cons(name,boundVars),conNames,funNames,expr)
1230
1231   case ProjectionExpr(pos,expr,name) ->
1232     collectFreeVars(boundVars,conNames,funNames,expr)
1233
1234   case LambdaExpr(pos,args,expr) ->
1235     collectFreeVars(append(args,boundVars),conNames,funNames,expr)
1236
1237   case FunCall(pos,name,args) ->
1238     if not(elemStr(name,conNames)) && not(elemStr(name,funNames)) &&
      not(elemStr(name,boundVars))
1239     then Cons(name,collectFreeVarsList(boundVars,conNames,funNames,
      args))
1240     else collectFreeVarsList(boundVars,conNames,funNames,args)
1241
1242   case IntLit(pos, i) -> Nil()
1243
1244   case StringLit(pos,s) -> Nil()
1245
1246   case MultExpr(pos,op1,op2) ->
1247    append(collectFreeVars(boundVars,conNames,funNames,op1)
1248          ,collectFreeVars(boundVars,conNames,funNames,op2))
```

```
1249
1250    case DivExpr(pos,op1,op2) ->
1251     append(collectFreeVars(boundVars,conNames,funNames,op1)
1252            ,collectFreeVars(boundVars,conNames,funNames,op2))
1253
1254    case ModExpr(pos,op1,op2) ->
1255     append(collectFreeVars(boundVars,conNames,funNames,op1)
1256            ,collectFreeVars(boundVars,conNames,funNames,op2))
1257
1258    case AddExpr(pos,op1,op2) ->
1259     append(collectFreeVars(boundVars,conNames,funNames,op1)
1260            ,collectFreeVars(boundVars,conNames,funNames,op2))
1261
1262    case SubExpr(pos,op1,op2) ->
1263     append(collectFreeVars(boundVars,conNames,funNames,op1)
1264            ,collectFreeVars(boundVars,conNames,funNames,op2))
1265
1266    case EqExpr(pos,op1,op2) ->
1267     append(collectFreeVars(boundVars,conNames,funNames,op1)
1268            ,collectFreeVars(boundVars,conNames,funNames,op2))
1269
1270    case NeqExpr(pos,op1,op2) ->
1271     append(collectFreeVars(boundVars,conNames,funNames,op1)
1272            ,collectFreeVars(boundVars,conNames,funNames,op2))
1273
1274    case LeExpr(pos,op1,op2) ->
1275     append(collectFreeVars(boundVars,conNames,funNames,op1)
1276            ,collectFreeVars(boundVars,conNames,funNames,op2))
1277
1278    case GeExpr(pos,op1,op2) ->
1279     append(collectFreeVars(boundVars,conNames,funNames,op1)
1280            ,collectFreeVars(boundVars,conNames,funNames,op2))
1281
1282    case LtExpr(pos,op1,op2) ->
1283     append(collectFreeVars(boundVars,conNames,funNames,op1)
1284            ,collectFreeVars(boundVars,conNames,funNames,op2))
1285
1286    case GtExpr(pos,op1,op2) ->
1287     append(collectFreeVars(boundVars,conNames,funNames,op1)
1288            ,collectFreeVars(boundVars,conNames,funNames,op2))
1289
1290    case OrExpr(pos,op1,op2) ->
1291     append(collectFreeVars(boundVars,conNames,funNames,op1)
1292            ,collectFreeVars(boundVars,conNames,funNames,op2))
1293
1294    case AndExpr(pos,op1,op2) ->
1295     append(collectFreeVars(boundVars,conNames,funNames,op1)
1296            ,collectFreeVars(boundVars,conNames,funNames,op2))
1297
1298    case IfExpr(pos,cond,a1,a2) ->
1299     append(collectFreeVars(boundVars,conNames,funNames,cond),
1300     append(collectFreeVars(boundVars,conNames,funNames,a1),
```

```
1301     collectFreeVars(boundVars,conNames,funNames,a2)))
1302
1303   case Var(pos,name) ->
1304     if not(elemStr(name,boundVars)) then Cons(name,Nil()) else Nil()
1305
1306   case Prog(cons,funs) -> Nil()
1307 ]
1308
1309 fun globalizeCasesList(conNames,funNames,xs)=match xs[
1310   case Cons(y,ys) -> match globalizeCases(conNames,funNames,y) [
1311     case Pair(f1s,yNew) -> match globalizeCasesList(conNames,funNames,
      ys)[
1312       case Pair(f2s,ysNew) -> Pair(append(f1s,f2s),Cons(yNew,ysNew))
1313     ]
1314   ]
1315   case Nil() -> Pair(Nil(),Nil())
1316 ]
1317
1318 fun globalizeCases(conNames,funNames,tree)=match tree[
1319   case ConDef(pos,name,args) -> Pair(Nil(),tree)
1320
1321   case FunDef(pos,name,args,def) -> match globalizeCases(conNames,
      funNames,def)[
1322     case Pair(fs,newDef) -> Pair(fs,FunDef(pos,name,args,newDef))
1323   ]
1324
1325
1326   case SingleCalledFunDef(pos,name,args,def) -> match globalizeCases(
      conNames,funNames,def)[
1327     case Pair(fs,newDef) -> Pair(fs,SingleCalledFunDef(pos,name,args,
      newDef))
1328   ]
1329
1330   case LetExpr(pos,defs,expr) -> Pair(Nil(),tree)
1331
1332   case MatchExpr(pos,expr,cases) -> match globalizeCases(conNames,
      funNames,expr)[
1333     case Pair(f1s,expNew) -> match globalizeCasesList(conNames,
      funNames,cases)[
1334       case Pair(f2s,casesNew) -> Pair(append(f1s,f2s),MatchExpr(pos,
      expNew,casesNew))
1335     ]
1336   ]
1337
1338   case CaseExpr(pos,name,args,expr) -> match collectFreeVars(Nil(),
      conNames,funNames,expr)[
1339     case freeVars -> match genLabel("case_",pos)[
1340       case casename -> match Cons("this",append(args,freeVars)) [
1341         case funArgs -> match globalizeCases(conNames,funNames,expr)[
1342           case Pair(fs,exprNew) ->
1343             Pair(Cons(SingleCalledFunDef(getTreePosition(expr),
      casename,funArgs,exprNew),fs)
```

```
1344                  ,CaseExpr(pos,name,args,FunCall(getTreePosition(expr)
1345                                                  ,casename
1346                                                  ,map(\ (x)->Var(Pair
    (0,0),x),funArgs) )))
1347          ]
1348        ]
1349      ]
1350    ]
1351
1352    case VarCaseExpr(pos,name,expr) -> match collectFreeVars(Nil(),
     conNames,funNames,expr)[
1353      case freeVars -> match genLabel("case_",pos)[
1354        case casename -> match Cons("this",freeVars) [
1355          case funArgs -> match globalizeCases(conNames,funNames,expr)[
1356            case Pair(fs,exprNew) ->
1357              Pair(Cons(SingleCalledFunDef(getTreePosition(expr),
     casename,funArgs,exprNew),fs)
1358                  ,VarCaseExpr(pos,name,FunCall(getTreePosition(expr)
1359                                                ,casename
1360                                                ,map(\ (x)->Var(Pair(0,0)
    ,x),funArgs) )))
1361          ]
1362        ]
1363      ]
1364    ]
1365
1366    case ProjectionExpr(pos,expr,name) -> match globalizeCases(conNames,
     funNames,expr)[
1367      case Pair(fs,expNew) -> Pair(fs,ProjectionExpr(pos,expNew,name))
1368    ]
1369
1370    case LambdaExpr(pos,args,expr) -> match globalizeCases(conNames,
     funNames,expr)[
1371      case Pair(fs,expNew) -> Pair(fs,LambdaExpr(pos,args,expNew))
1372    ]
1373
1374    case FunCall(pos,name,args) -> match globalizeCasesList(conNames,
     funNames,args)[
1375      case Pair(fs,argsNew) -> Pair(fs,FunCall(pos,name,argsNew))
1376    ]
1377
1378    case IntLit(pos, i) -> Pair(Nil(),tree)
1379    case StringLit(pos,s) -> Pair(Nil(),tree)
1380    case MultExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames
     ,op1)[
1381      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
     op2)[
1382        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),MultExpr(pos,
     op1New,op2New))
1383      ]
1384    ]
```

```
1385    case DivExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1386      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1387        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),DivExpr(pos,op1New
          ,op2New))
1388      ]
1389    ]
1390    case ModExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1391      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1392        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),ModExpr(pos,op1New
          ,op2New))
1393      ]
1394    ]
1395    case AddExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1396      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1397        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),AddExpr(pos,op1New
          ,op2New))
1398      ]
1399    ]
1400    case SubExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1401      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1402        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),SubExpr(pos,op1New
          ,op2New))
1403      ]
1404    ]
1405    case EqExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1406      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1407        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),EqExpr(pos,op1New,
          op2New))
1408      ]
1409    ]
1410    case NeqExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1411      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
1412        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),NeqExpr(pos,op1New
          ,op2New))
1413      ]
1414    ]
1415    case LeExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
          op1)[
1416      case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
          op2)[
```

```
1417        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),LeExpr(pos,op1New,
      op2New))
1418       ]
1419    ]
1420    case GeExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
      op1)[
1421     case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
      op2)[
1422        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),GeExpr(pos,op1New,
      op2New))
1423       ]
1424    ]
1425    case LtExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
      op1)[
1426     case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
      op2)[
1427        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),LtExpr(pos,op1New,
      op2New))
1428       ]
1429    ]
1430    case GtExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
      op1)[
1431     case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
      op2)[
1432        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),GtExpr(pos,op1New,
      op2New))
1433       ]
1434    ]
1435    case OrExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
      op1)[
1436     case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
      op2)[
1437        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),OrExpr(pos,op1New,
      op2New))
1438       ]
1439    ]
1440    case AndExpr(pos,op1,op2) -> match globalizeCases(conNames,funNames,
      op1)[
1441     case Pair(f1s,op1New) -> match globalizeCases(conNames,funNames,
      op2)[
1442        case Pair(f2s,op2New) -> Pair(append(f1s,f2s),AndExpr(pos,op1New
      ,op2New))
1443       ]
1444    ]
1445    case IfExpr(pos,cond,a1,a2) -> Pair(Nil(),tree)
1446    case Var(pos,name) -> Pair(Nil(),tree)
1447    case Prog(cons,funs) -> match globalizeCasesList(map(\ (c)->match c[
      case ConDef(p,n,args)->n],cons)
1448                                                     ,map(\ (f)->match f[
      case FunDef(p,n,a,d)->n case SingleCalledFunDef(p,n,a,d)->n],funs)
1449                                                     ,funs)[
1450       case Pair(fs,newFuns) -> Pair(fs,Prog(cons,newFuns))
```

```
  ]
]

fun pushInt(prim) =
  Cons(append("    movl ",append(prim,", %ebx\n")),
  Cons(append("    movl stack, %eax #PushInt ",append(prim," \n")),
  Cons("    movl sp, %edx\n",
  Cons("    sall $3, %edx\n",
  Cons("    addl %edx, %eax\n",
  Cons("    movl $1, (%eax)\n",
  Cons("    movl stack, %eax\n",
  Cons("    movl sp, %edx\n",
  Cons("    sall $3, %edx\n",
  Cons("    addl %eax, %edx\n",
  Cons("    movl %ebx, %eax\n",
  Cons("    movl %eax, 4(%edx)\n",
  Cons("    movl sp, %eax\n",
  Cons("    addl $1, %eax\n",
  Cons("    movl %eax, sp\n",Nil())))))))))))))))

fun  genPop()=
  Cons("    movl    sp, %eax  #start pop\n",
  Cons("    subl    $1, %eax\n",
  Cons("    movl    %eax, sp\n",Nil())))

fun genPopReg( reg)=
  append(genPop(),
  Cons("    movl    sp, %eax  #save pop in reg\n",
  Cons("    sall    $3, %eax\n",
  Cons("    addl    stack, %eax\n",
  Cons("    addl    $4, %eax\n",
  Cons(append("    movl    (%eax), ",append(reg,"\n")),Nil()))))))

fun genX86(fileName,cons,funs,code)
 = match Success(writeFile(fileName,genConstructorArray(cons))) [
  case written -> genX86ForInstructions(fileName,giveNumbers(1,code))
]

fun genX86ForInstructions(fileName,codes) = match codes [
  case Nil() -> Success("ready")
  case Cons(nrcode,codes2)
         -> match Success(appendFile(fileName,concat(
    genX86ForInstruction(nrcode))))[
    case written -> genX86ForInstructions(fileName,codes2)
  ]
]



fun giveNumbers(i,xs)=match xs[
  case Nil() -> Nil()
```

```
1502   case Cons(y,ys) -> Cons(Pair(i,y),giveNumbers(i+1,ys))
1503 ]
1504
1505 fun genConstructorArray(cons)=
1506   concat(
1507     Cons(".globl  constructors\n",
1508     Cons(".section        .rodata\n",
1509     Cons(concat
1510        (map(\ (p)->match p[
1511              case Pair(i,co)->match co [
1512                case ConDef(p,n,args)->
1513                 append(".LC",append(show(i),append(":\n.string \"",
     append(n,"\"\n"))))]]
1514             ,giveNumbers(0,cons))),
1515     Cons(".data\n",
1516     Cons(".align 4\n",
1517     Cons(".type   constructors, @object\n",
1518     Cons(append(".size   constructors, ",append(show(4*size(cons)),"\n
     ")),
1519     Cons("constructors:\n",
1520     Cons(concat
1521       (map(\ (p)->match p[case Pair(i,co)->append(".long   .LC",append
     (show(i),"\n"))]
1522          ,giveNumbers(0,cons))), Nil()
1523     )))))))))
1524   )
1525
1526 fun genCompareExpr(jmp,nr)=
1527    append( genPopReg("%ecx"),
1528    append( genPopReg("%edx"),
1529    Cons("    movl    %edx, %ebx\n",
1530    Cons("    movl    %ecx, %eax\n",
1531    Cons("    cmpl    %eax, %ebx\n",
1532    Cons(append("    ",append(jmp,append("     posLabel",append(show(nr
     ),"\n")))),
1533    append(pushInt("$0"),
1534    Cons(append("    jmp",append("   endLabel",append(show(nr),"\n"))),
1535    Cons(append("  posLabel",append(show(nr),":\n")),
1536    append(pushInt("$1"),
1537    Cons(append("   endLabel",append(show(nr),":\n")),Nil()))))))))))))
1538
1539
1540
1541 fun genX86ForInstruction(p)=match p[
1542   case Pair(nr,instr)->match instr [
1543    case Instruction(ins,a1,a2) -> match ins [
1544     case GlobLabel() ->
1545       Cons(append(".globl ",append(a2,"\n")),
1546       Cons(append(a2,":\n"),Nil()))
1547
1548     case StartCall() ->
1549       Cons("    movl sb, %ecx\n",
```

```
1550        append(pushInt("%ecx"),
1551        Cons("    movl sp, %edx\n",
1552        Cons("    movl %edx, sb\n",Nil())))
1553
1554      case PushInt() -> pushInt(append("$",a1))
1555
1556      case Add() ->
1557        Cons("#ADD \n",
1558        append(genPopReg("%ecx"),
1559        append(genPopReg("%edx"),
1560        Cons("    addl    %ecx, %edx\n",
1561        pushInt("%edx")))))
1562
1563      case Mult() ->
1564        append(genPopReg("%ecx"),
1565        append(genPopReg("%edx"),
1566        Cons("    movl    %edx, %ebx\n",
1567        Cons("    movl    %ecx, %eax\n",
1568        Cons("    mull    %ebx\n",
1569        pushInt("%eax"))))))
1570
1571      case Sub() ->
1572        Cons("#Sub \n",
1573        append(genPopReg("%ecx"),
1574        append(genPopReg("%edx"),
1575        Cons("    subl    %ecx, %edx\n",
1576        pushInt("%edx")))))
1577
1578      case Mod() ->
1579        append(genPopReg("%ecx"),
1580        append(genPopReg("%edx"),
1581        Cons("    movl    %edx, %ebx\n",
1582        Cons("    movl    %ecx, %eax\n",
1583        Cons("    movl $0, %edx\n"),
1584        Cons("    divl    %ebx\n",
1585        pushInt("%edx"))))))
1586
1587      case Div() ->
1588        append(genPopReg("%ecx"),
1589        append(genPopReg("%edx"),
1590        Cons("    movl    %edx, %ebx\n",
1591        Cons("    movl    %ecx, %eax\n",
1592        Cons("    movl $0, %edx\n"),
1593        Cons("    divl    %ebx\n",
1594        pushInt("%eax"))))))
1595
1596      case Eq() -> genCompareExpr("je",nr)
1597      case Neq()-> genCompareExpr("jne",nr)
1598      case Lt() -> genCompareExpr("jl",nr)
1599      case Gt() -> genCompareExpr("jg",nr)
1600      case Le() -> genCompareExpr("jle",nr)
1601      case Ge() -> genCompareExpr("jge",nr)
```

```
1602      case Or() -> Nil()
1603      case And() -> Nil()
1604
1605      case Ret() ->
1606        Cons(append("    movl $",append(show(a1),", callingArg\n")),
1607        Cons("    call ret\n",
1608        Cons("    ret   \n",Nil())))
1609
1610      case Call() ->
1611        Cons(append("    call   ", append(a2,"\n")),Nil())
1612
1613      case Push() ->
1614        Cons(append("    movl $",append(show(a1+1),", callingArg #start
        push\n")),
1615        Cons("    movl stack, %eax\n",
1616        Cons("    movl sp, %edx\n",
1617        Cons("    sall $3, %edx\n",
1618        Cons("    addl %eax, %edx\n",
1619        Cons("    movl stack, %eax\n",
1620        Cons("    movl sb, %ebx\n",
1621        Cons("    movl callingArg, %ecx\n",
1622        Cons("    subl %ecx, %ebx\n",
1623        Cons("    movl %ebx, %ecx\n",
1624        Cons("    sall $3, %ecx\n",
1625        Cons("    addl %ecx, %eax\n",
1626        Cons("    movl (%eax), %eax\n",
1627        Cons("    movl %eax, (%edx)\n",
1628        Cons("    movl stack, %eax\n",
1629        Cons("    movl sp, %edx\n",
1630        Cons("    sall $3, %edx\n",
1631        Cons("    addl %eax, %edx\n",
1632        Cons("    movl stack, %eax\n",
1633        Cons("    movl sb, %ebx\n",
1634        Cons("    movl callingArg, %ecx\n",
1635        Cons("    subl %ecx, %ebx\n",
1636        Cons("    movl %ebx, %ecx\n",
1637        Cons("    sall $3, %ecx\n",
1638        Cons("    addl %ecx, %eax\n",
1639        Cons("    movl 4(%eax), %eax\n",
1640        Cons("    movl %eax, 4(%edx)\n",
1641        Cons("    movl sp, %eax\n",
1642        Cons("    addl $1, %eax\n",
1643        Cons("    movl %eax, sp\n",Nil()))))))))))))))))))))))))))))))))
1644
1645      case Label() -> Cons(append(a2,":\n"),Nil())
1646
1647      case Jmp() -> Cons(append("    jmp ",append(a2,"\n")),Nil())
1648
1649      case JmpTrue() ->
1650        append(genPopReg("%eax"),
1651        Cons("    movl $1, %ebx\n",
1652        Cons("    cmpl %eax, %ebx\n",
```

```
1653        Cons(append("    je ",append(a2,"\n")),Nil())))
1654
1655      case PushFun() ->  Cons("#not implemented\n",Nil())
1656
1657      case otherwise -> Cons("#not implemented\n",Nil())
1658    ]
1659  ]
1660 ]
1661
1662 fun show(x)=
1663   if x<0 then append("-",show(0-x))
1664   else if x==0 then "0"
1665   else if (x<10) then Cons(head("0")+x,Nil())
1666   else append(show(x/10),show(x%10))
1667
1668 fun a3(f,x,y,z)=match f(x,y,z)[
1669   case Fail(f)-> 42
1670   case Success(succ) -> fst(succ)
1671 ]
1672
1673 fun singleton(x)=Cons(x,Nil())
1674
1675 fun compile(s)= match pProg(tokenize(numerate(1,1,s)))[
1676   case Fail(f) -> f
1677   case Success(succ) -> match succ [
1678     case Pair(prog,rest) -> match globalizeCases(Nil(),Nil(),prog)[
1679       case Pair(fs,progNew) -> match progNew [
1680         case Prog(cons,funs)->  genX86("ttt.s",cons,append(funs,fs),
    genCode(Prog(cons,append(funs,fs))) )
1681       ]
1682     ]
1683   ]
1684 ]
1685
1686 fun go()=
1687    /* lookup(Cons(Pair("murx",17),Cons(Pair("witzelbritz",42),Nil()))
    ,"witzelbritz")*/
1688    compile(readFile("src/wipc.wip"))
1689 /*   compile("con Nil() con Cons(head,tail) fun fib(n)= if n<=1 then 1
     else fib(n-1)+fib(n-2) fun f(x)= if x==0 then 1 else x*f(x-1)  fun
     go()=fib(6)")
1690
1691 /*   compile("con Nil() con Cons(head,tail) fun go()=match \"hallo\" [
    case Nil()->\"nix\" case Cons(y,ys)->ys]")
1692 /*   compile("con Nil() con Cons(head,tail) fun go()= if 17==1 then 17
     else 42")*/
1693
1694 /*  match pProg(tokenize(numerate(1,1,readFile("src/wipc.wip")))) [
1695     case Success(succ) -> match succ[
1696       case Pair(prog,rest)->globalizeCases(Nil(),Nil(),prog)]]
1697   /*seqs(Cons(pConT,Cons(pItentT,Cons(pLParT,Cons(pArgs,Cons(pRParT,
    Nil()))))))*/
```

```
1698  /*(tokenize(numerate(1,1,"\\(xs) -> con Test( a,b,c)")) )*/
1699
1700  /*pProg(tokenize(numerate(1,1,"con Test( a,b,c) fun f( a,b,c) = if (
       u==v) then 42 else result && a == d && b == i+j*k || c || d && f
       (1,2,3)")) )*/
1701
1702  /*map(singleton,"hello")*/
1703  /*pCondef(tokenize(numerate(1,1,"con test(a, b, c, d,dd,rdfq, tzu)")
       )) */
1704
1705  /*pCommaIdentArgs(tokenize(numerate(1,1,",a, b, c, d,dd,rdfq tzu")))
       */
```

Listing 11.1: wipc.wip