Ein kleiner Compiler

Sven Eric Panitz

31. Mai 2019

Inhaltsverzeichnis

1 Compilerbau

In diesem Projektblatt des Moduls *»Programmiermethoden und Techniken«* wird ein Compiler für eine kleine Programmierprache entwickelt.

Ein Compiler ist ein Computerprogramm, das andere Prorgamme als Eingabe erhält und wiederum Computerprogramme als Ausgabe erzeugt. Wir sprechen also bei einem Compiler über drei beteiligte Programmiersprachen:

- der Quellsprache des Programms, das der Compiler als Eingabe erhält.
- die Zielsprache, in die das Eingabeprogrammm übersetzt wird. Dieses ist oft eine Maschinensprache in Form von Assembler.
- die Implementierungssprache, in der der Compiler entwickelt wurde.

Für dieses Projektblatt nutzen wir als Implementierungssprache Java und als Quellsprache eine einfache Sprache mit Bedingungen, Schleifen und Funktionsaufrufen.[1] Zielsprache ist Gnu Assembler (GAS) für x86-64-Architekturen.

1.1 Phasen eines Compilers

Ein Compiler besteht aus mehreren Phasen:

- Die lexikographische Analyse: Hier werden aus einem String die einzelnen Wörter der Sprache erzeugt. Diese werden auch als Token bezeichnet. Wörter sind z.B. Schlüsselwörter, Literale und Symbole der Sprache. Dieser Teil des Compilers wird als Lexer oder auch als Tokenizer bezeichnet. Das Ergebnis der lexikographischen Analyse ist eine Folge von Token.
- Die syntaktische Analyse: Dieser Teil des Compilers wird als Parser bezeichnet. Das Ergebnis ist ein abstrakter Syntaxbaum (kurz AST), der die Struktur des gelesenen Programms darstellt.
- Statische Analysen: In dieser Phase prüft der Compiler den Quelltext auf Konsistenz. Das ist bei statisch getypten Sprache zunächst der Typcheck. Bei einer Sprache wie Java fallen hier eine Vielzahl von Prüfungen an, wie:
  - sind alle Exceptions gefangen oder im `throws` deklariert?
  - werden Sichtbarkeiten beim Verwenden von Eigenschaften beachtet?
  - gibt es garantiert eine `return`-Anweisung bei Methoden mit Rückgabewert?
  - werden alle abstrakten Methoden implementiert?
  - gibt es keine zyklische Vererbung?
  - etc etc.
- Code-Generierung: In dieser Phase wird nun der eigentlich Code der Zielsprache generiert.

[1] Wir haben keinen Namen für die Quellsprache. Das Javapaket, in der sie definiert ist heißt `longStack`, da nur mit dem Datentyp `long int` auf dem Stack gearbeitet wird. Das Anwenderprogramm trägt in nostalgischer Anwandlung an das Prograann *hugs* den Namen TUGS.

Viel Compiler haben noch weitere Phasen, wie die Übersetzung des Quelltextes in eine kleinere Kernsprache, den sogenannten Entzuckern von Konstrukten, die den Quelltext bequemer machen, einen abstrakten Zwischencode und unterschiedlichen Optimierungsphasen.

In diesem Projektblatt ist der abstrakte Syntaxbaum vorgegeben. Ein Lexer und ein Parser liegen als Klassen bereits vor.

Auf den Syntaxbaum werden drei Algorithmen realisiert:

- Ein Pretty-Printer, der das Programm, das durch den AST dargestellt wird, wieder als Text formatiert darstellt.
- Ein Interpreter, der Ausdrücke direkt auswertet.
- Eine Code-Generierung, die Assembler generiert, der mit dem gcc zusammen mit anderen C-Programmen zu einem ausführbaren Programm gelinkt werden kann.

Als Hilfsalgorithmus werden in einer Funktion alle lokalen Variablen einer Funktion ermittelt.

1.2 Importierte Klassen

Wir benötigen ein rudimentäres IO, eine Reihe von Container-Klassen und zwei funktionale Schnittstellen.

Wir werden folgende Standardklassen verwenden.

Zunächst einige Klassen des Pakets `java.io`:

Java: AST

```
package name.panitz.longStack;

import java.io.IOException;
import java.io.StringWriter;
import java.io.Writer;
import java.io.FileWriter;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.BufferedReader;
```

Dann ein paar Sammlungsklassen des Pakets `java.util`:

Java: AST

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
```

Und ein paar funktionale Schnittstellen des Pakets `java.util.function`:

Java: AST

```
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Consumer;
```

Um alle in der Aufgabe implementierten Methoden manuell auszuprobieren, ist eine Methode `main` am Ende dieses Papiers definiert.

2 Abstrakter Syntaxbaum

Die Umsetzung des AST und der drei Algorithmen des AST bündeln wir in einer einzigen Java Schnittstelle. Alle Klassen, die wir benötigen, sind innere statische Klassen dieser Schnittstelle. Funktionalitäten sind als `default`-Methoden der Schnittstelle realisiert.

Java: AST

```
@SuppressWarnings("unchecked")
public interface AST {
```

2.1 Operatoren der Sprache

Bevor wir die eigentlichen Baumknoten des AST definieren, seien die Operatoren der Quellsprache in Form einer Aufzählungsklasse definiert.

Java: AST

```
public static enum BinOP{
  add ((x,y)->x+y,"+"),
  sub ((x,y)->x-y,"-"),
  mult((x,y)->x*y,"*"),
  eq  ((x,y)->x==y?1L:0L,"=");

  BinOP(BinaryOperator<Long> op,String name){
    this.op = op;
    this.name = name;
  }
  BinaryOperator<Long> op;
  String name;
}
```

Die vier Werte dieser Aufzählungsklasse beinhalten den textuellen Namen des Operators und die Funktion, die der Operator ausdrückt.

2.2 Baumknoten

Nun definieren wir Klassen, die die Schnittstelle `AST` implementieren. Diese Klassen beschreiben als Baumknoten jeweils bestimmte Konstrukte unserer Quellsprache. Anweisungen, die aus mehreren anderen Ausdrücken oder Anweisungen zusammengesetzt sind, haben diese als Kindknoten.

Zusammen mit den Baumknoten geben wir jeweils ein Beispiel für das Sprachkontrukt im Quelltext.

2.2.1 Zahlenliterale

Der einfachste Ausdruck unserer Quellsprache sind Zahlenliterale. Wir verarbeiten als einzigen Datentyp ganze Zahlen des Typs `long`, also 64 Bit lange Zahlen.

Der Baumknoten für Zahlenliterale hat keine Kinder und enthält den Wert des Zahlenliterals.

Java: AST

```
public static class IntLiteral implements AST{
  long n;
  public IntLiteral(long n) {
    this.n = n;
  }
}
```

2.2.2 Variablen

Unsere Quellsprache kennt Variablen. Solche Variablen können lokale Variablen einer Funktion sein, oder Parameter der Funktion. Der Baumknoten für Variablen hat keine Kinder und enthält den Variablennamen.

Java: AST

```
public static class Var implements AST{
  String name;
  public Var(String name) {
    this.name = name;
  }
}
```

2.2.3 Zuweisung

In der Quellsprache können Variablen Werte zugewiesen werden. Als Zuweisungsoperator dient der Operator `:=`. Auf der rechten Seite kann ein beliebiger Ausdruck unserer Quellsprache stehen.

```
x := 42
```

Die Baumknoten der Zuweisung enthalten zwei Kinder. Einen Variablenknoten und einen beliebigen AST für die rechte Seite der Zuweisung.

Java: AST

```
public static class Assign implements AST{
  Var v;
  AST right;
  public Assign(Var left, AST right) {
    this.v = left;
    this.right = right;
  }
}
```

2.2.4 Operator-Ausdrücke

Die binären Operatoren haben wir bereits in einer Aufzählungsklasse definiert. Es gibt im Quelltext die Operatoren `+`, `-`, `*` und `=`. Auf weitere Operatoren haben wir der Einfachheit halber verzichtet. Insbesondere die Divisionsoperationen sind im Assembler ein klein wenig komplexer.

Im Quelltext gilt die übliche Operatorpräzendenz und es können Ausdrücke geklammert sein, so dass der folgende Ausdruck ein gültiger Ausdruck ist.

`(17+4)*2=26*2-10`

Ein Baumknoten für einen Operatorausdruck hat als Kinder den linken und den rechten Operanden und ein Objekt für den Operator.

Java: AST

```
public static class OpExpr implements AST{
  AST left, right;
  BinOP op;
  public OpExpr(AST left, BinOP op,AST right) {
    this.left = left;
    this.op = op;
    this.right = right;
  }
}
```

2.2.5 Bedingungsausdrücke

Die Quellsprache kennt if-Bedingungen. Diese haben immer auch den else-Fall zu spezifizieren. Syntaktisch werden die drei Schlüsselwörter `if`, `then` und `else` verwendet, um die Bedingung und die beiden Alternativen zu trennen.

`if 17+4*2=x-1 then 1 else 42`

Die Bedigung wird als wahr bezeichnet, wenn sie nicht den Wert `0` hat.

Der Baumknoten hat entsprechend drei Kinder: die Bedingung und die beiden Alternativen.

Java: AST

```
public static class IfExpr implements AST{
  AST cond, alt1, alt2;
  public IfExpr(AST cond, AST alt1, AST alt2) {
    this.cond = cond;
    this.alt1 = alt1;
    this.alt2 = alt2;
  }
}
```

2.2.6 Schleifen

Die Quellsprache kennt eine typische `while`-Schleife aus einer Bedingung und einem Schleifenrumpf. Zur Trennung werden syntaktisch die Schlüsselwörter `while` und `do` verwendet.

```
while x+y=8 do x:=x-1
```

Entsprechend hat der Baumknoten für Schleifen zwei Kinder. In einem Kind steht der AST für die Bedingung im anderen Kind der AST für den Schleifenrumpf.

Java: AST

```java
public static class WhileExpr implements AST{
    AST cond, body;
    public WhileExpr(AST cond, AST body) {
      this.cond = cond;
      this.body = body;
    }
  }
```

2.2.7 Code-Blöcke

Ein Code-Block ist die Sequenz von mehreren Ausdrücken.

Syntaktisch sind Code-Blöcke in geschweiften Klammern eingeschlossen und die Ausdrücke des Code-Blocks durch Semikolons getrennt. Das ist anders als in Java. Die Semikolons stehen nur zwischen zwei Ausdrücken, nicht auch noch nach dem letzten Ausdruck. Sie markieren also nicht das Ende eines Ausdrucks.

Der Wert eines Code-Blocks sei der Wert des letzten Ausdrucks in der Folge von Ausdrücken.

```
{x:=5
;r:=1
;while (x)
  {r := r*x
  ;x := x -1
  }
;r
}
```

Der Baumknoten für Code-Blöcke kann eine beliebige Anzahl von Kindern haben. Daher enthält er eine Liste von Kindern.

Java: AST

```java
public static class Sequence implements AST{
  List<AST> sts;
  public Sequence(List<AST> sts) {
    this.sts = sts;
  }
}
```

2.2.8 Funktionsaufrufe

Unsere Quellsprache kennt auch Funktionen, die in herkömmlicher Weise aufgerufen werden können.

```
if x=0 then 1 else x*f(x-1)
```

Der Baumknoten enthält den Namen der aufgerufenen Funktion und eine Liste von Argumenten.

Java: AST

```
public static class FunCall implements AST{
  String name;
  List<AST> args;
  public FunCall(String name, List<AST> args) {
    this.name = name;
    this.args = args;
  }
}
```

2.2.9 Funktionsdefinitionen

In der Quellsprache ist es auch möglich, Funktionen zu definieren. Hierzu dient das Schlüsselwort `fun`. Eine Funktion hat dann einen Namen und eine Parameterliste. Ein Gleichheitszeichen läutet den Funktionsrumpf ein.

```
fun fac(x) = if x=0 then 1 else x*f(x-1)
```

Der Name der Funktion und der Name der Argumente sind im Knoten als Strings gespeichert. Der Rumpf ist ein Kindknoten im Baum. Die Klasse, die Funktionsdefinitionen beschreibt, ist kein Teil des AST. Es gibt also vorerst auch keine lokalen Funktionsdefinitionen.

Java: AST

```
static class Fun{
  String name;
  List<String> args;
  AST body;
  public Fun(String name, List<String> args, AST body) {
    this.name = name;
    this.args = args;
    this.body = body;
  }
}
```

2.3 Beispielprogramme

Um ein besseres Gefühl für die Sprache und den AST zu bekommen, betrachten wir ein paar Beispielprogramme.

2.3.1 Iterative Fakultät

Zunächst eine Umsetzung der Fakultätsfunktion mit Hilfe einer Schleife. Es gibt eine Ergebnisvariable `r` und es wird auf dem Parameter `x` gerechnet.

```
fun fac(x) =
  {r := 1
  ;while x
   do
     {r := r*x
     ;x := x-1
     }
  ;r
  }
```

Folgendes ist de AST dieses Programms.

Java: AST

```
static Fun factorial= new Fun("fac",List.of("x"),
  new Sequence(List.of
    (new Assign(new Var("r"), new IntLiteral(1))
    ,new WhileExpr
      (new Var("x")
      ,new Sequence(List.of
        (new Assign
          (new Var("r")
          ,new OpExpr(new Var("r"), BinOP.mult,new Var("x")))
        ,new Assign
          (new Var("x")
          ,new OpExpr(new Var("x"), BinOP.sub, new IntLiteral(1)))
        ))
      )
    ,new Var("r")
    ))
  );
```

2.3.2 Rekursive Fakultät

Die rekursive Version der Fakultätsfunktion lässt sich elegant in einer Zeile schreiben:

```
fun f(x) = if x = 0 then 1 else x*f(x-1)
```

Entsprechend ist auch der AST für dieses Programm etwas kürzer.

Java: AST

```
static Fun factorialRek = new Fun("f",List.of("x"),
  new IfExpr(new OpExpr(new Var("x"),BinOP.eq,new IntLiteral(0))
      , new IntLiteral(1)
      , new OpExpr(new Var("x"),BinOP.mult
        ,new FunCall("f"
          , List.of(new OpExpr(new Var("x")
             , BinOP.sub, new IntLiteral(1)))))));
```

2.3.3 Zwei Parameter

Die folgende Funktion soll dazu dienen, zu testen, ob die Parameter nicht vertauscht werden. Dieses passiert gerne einmal bei der Code-Generierung.

```
fun minus(x,y) = x-y
```

Hier noch der AST für diese Funktion.

Java: AST

```
static Fun minus =
  new Fun("minus"
         ,List.of("x","y")
         ,new OpExpr(new Var("x"), BinOP.sub, new Var("y")));
```

2.3.4 Fibonaccizahlen

Als letztes darf die Funktion der Fibonaccizahlen nicht fehlen. Auch diese kommt ganz ohne einen längeren Code-Block aus.

```
fun fic(x) = if x=0 then 0 else if x = 1 then 1 else fib(x-2)+fib(x-1)
```

Und auch hierzu sei der AST in Java definiert.

Java: AST

```
static Fun fib
  = new Fun("fib",List.of("x")
     ,new IfExpr(new OpExpr(new IntLiteral(0),BinOP.eq,new Var("x"))
        ,new IntLiteral(0)
        ,new IfExpr(new OpExpr(new IntLiteral(1),BinOP.eq,new Var("x"))
        ,new IntLiteral(1)
        ,new OpExpr
           (new FunCall("fib"
           ,List.of
              (new OpExpr(new Var("x"),BinOP.sub, new IntLiteral(2))))
             ,BinOP.add
             ,new FunCall("fib"
             ,List.of
              (new OpExpr(new Var("x"),BinOP.sub, new IntLiteral(1))))
      ))));
```

3 Pattern Matching für Baumknoten

Im Folgenden sollen mehrere Algorithmen für den AST geschrieben werden. In einer rein objektorientierten Lösung würde man hierzu in der Schnittstelle AST jeweils eine abstrakte Methode definieren und diese in jeder Baumklasse implementieren.

Eine solche Umsetzung hat den Nachteil, dass es keinen einheitlichen Blick auf einen komplexen Algorithmus gibt. Die Code-Generierung ist ein Algorithmus, den man lieber gebündelt aufschreibt, um einen kompletten Überblick über sie zu bekommen.

Somit ist eine Lösung, für die Algorithmen eine Methode zu schreiben, die ein Objekt des Typs AST bekommt. Diese Methode muss dann eine Fallunterscheidung treffen. Von welcher Klasse, die die Schnittstelle `AST` implementiert, ist das Argument?

Um eine unschöne if-else-Kaskade zu vermeiden, bietet sich in solchen Fällen das Besuchsmuster an[GHJV95], das auch typischer Weise in Compilerbau bei einer objektorientierten Implementierungssprache verwendet wird. Hierzu müssen die Baumklassen eine Methode beinhalten, die ein Objekt einer sogenannten Besucherklasse als Argument erwarten. Die Besucherklasse beinhaltet dann den Algorithmus, der über den Baum traversiert.

Wir gehen in dieser Projektaufgabe einen alternativen Weg, der ein wenig in die Zukunft von Java weist. Das Besuchsmuster ersetzt ein fehlendes eher funktionales Konstrukt, das sogenannte Pattern-Patching. Pattern-Matching kann als eine Form sehr ausgeklügelter switch-cases gesehen werden. Moderne Sprachen wie zum Beispiel die Programmiersprache Scala enthalten das Konzept des Pattern-Matching.

Bereits mit Java 12 gibt es eine leicht erweiterte switch-case-Anweisung[Par18]. Es ist angedacht dieses in der Zukunft auf ein volles Pattern-Matching auf reinen Datenhal-

tungsklassen zu erweitern[BG18]. Als Datenhaltungsklassen sind Klassen zu verstehen, die keine Methoden haben, sondern nur Felder und einen Konstruktor, der diese Felder initialisiert[Goe19]. Unsere Baumklassen sind solche reinen Datenhaltungsklassen.

Nun würden wir gerne ein Java-Konstrukt verwenden, das wahrscheinlich frühestens mit der nächsten Hauptversion von Java kommt (wahrscheinlich Java17).

Wir können uns aber mit einer kleinen eigenen reinen Java 10 Lösung begnügen, die einfachstes Pattern-Matching, das Typ-Pattern, realisiert. Der Schlüssel für solche Lösungen sind die seit Java 8 zur Verfügung stehenden λ-Ausdrücke. Mit diesen lassen sich viele nicht in der Sprache integrierte Konzepte als Bibliothek realisieren. Ein ausführliches Beispiel findet sich in [DW19].

Wir definieren für die Schnittstelle `AST` eine default-Methode `match`. Diese vergleicht das this-Objekt mit einer Folge von Typ-Pattern. Das erste Pattern dieser Folge, das auf das this-Objekt zutrifft, wird dann für das Ergebnis verwendet.

Java: AST

```
default <R> R match(PatternCase<? extends AST,R>... pts){
  for (var pt:pts) {
    if (pt.matches(this)) return pt.eval(this);
  }
  throw new RuntimeException("unmatched pattern case: "+this);
}
```

3.1 Pattern Matching für Baumknoten

Ein Pattern ist ein Objekt, mit dem getestet werden kann, ob ein Objekt das Pattern matcht und im Erfolgsfall das Ergebnis für den Match berechnet wird.

Java: AST

```
static interface PatternCase<C,R>{
  boolean matches(Object o);
  R eval(Object o);
}
```

Für eine beliebige Klassen kann die folgende Methode ein `PatternCase`-Objekt erzeugen. Dafür benötigt man eine Klasse, auf die gematcht werden soll und eine Funtion, die dann für das gematchte Objekt ein Ergebnis erzeugt.

Java: AST

```
static <C,R> PatternCase<C,R> mkPattern(Class<C> cl,Function<C, R> f){
  return new PatternCase<C,R>() {
    public boolean matches(Object o) {return cl.isInstance(o);}
    public R eval(Object o){return f.apply(cl.cast(o));}
  };
}
```

3.2 Die Pattern für die Klassen des AST

Mit dieser Hilfsfunktion erzeugen wir jetzt Funktionen, die jeweils für eine Baumknotenklasse ein Pattern-Case erzeugen.

Wir verletzten in diesem Fall die eigentliche Namenskonvention für Java-Methoden. Die Methoden heißen jeweils wie die Klassen, auf die gematcht wird mit einem vorangestellten Dollar-Symbol.

Java: AST

```
static <R> PatternCase<OpExpr,R> $OpExpr(Function<OpExpr, R> f) {
  return mkPattern(OpExpr.class, f);
}
static<R>PatternCase<IntLiteral,R> $IntLiteral(Function<IntLiteral,R>f){
  return mkPattern(IntLiteral.class, f);
}
static <R> PatternCase<Var,R> $Var(Function<Var, R> f){
  return mkPattern(Var.class, f);
}
static <R> PatternCase<Assign,R> $Assign(Function<Assign, R> f){
  return mkPattern(Assign.class, f);
}
static <R> PatternCase<IfExpr ,R> $IfExpr(Function<IfExpr, R> f){
  return mkPattern(IfExpr.class, f);
}
static <R> PatternCase<WhileExpr,R> $WhileExpr(Function<WhileExpr, R> f){
  return mkPattern(WhileExpr.class, f);
}
static <R> PatternCase<FunCall,R> $FunCall(Function<FunCall, R> f){
  return mkPattern(FunCall.class, f);
}
static <R> PatternCase<Sequence,R> $Sequence(Function<Sequence, R> f) {
  return mkPattern(Sequence.class, f);
}
```

Ein letztes Pattern-Case trifft auf alle Baumknoten zu. Man kann dieses in der Methode `match` als den Standardfall verwenden.

Java: AST

```
static <R> PatternCase<AST,R> $(Function<AST, R> f) {
  return mkPattern(AST.class, f);
}
```

3.3 Einfaches erstes Beispiel

Wir geben ein einfaches Beispiel für eine default Methode in der Schnittstelle `AST`, die mit der Methode `match` auf die Baumknotenklassen unterscheidet und einen String erzeugt.

Java: AST

```
default String whatAreYou() {
  return match
    ($Var       (v -> "Variable mit Namen »"+v.name+"«")
    ,$IntLiteral(il-> "Literal mit Wert "+il.n)
    ,$FunCall   (fc-> "Aufruf der Funktion: "+fc.name)
    ,$          (x -> "Irgend ein anderer Baumknoten")
    );
}
```

Manchmal soll eine Fallunterscheidung nach Baumknoten gemacht werden, aber kein Ergebnis erzeugt werden, sondern nur ein `Consumer`-Objekt auf den Baumknoten angewendet werden. Hierzu kann man sich der künstlichen Klasse `Void` als Ergebnistyp bedienen. Der Ergebnistyp des Pattern-Case ist dann der Typ `Void`. Trotzdem müssen Sie in der Funktion ein Ergebnis zurück geben, das von der Klasse `Void` ist.

Aus Bequemlichkeit definieren wir jetzt noch einmal für jede Baumklasse eine Funktion, die ein `Comsumer`-Objekt erhält und für dieses ein Pattern-Case erzeugt.

Wir stellen als Methodennamen den Unterstrich jetzt dem Klassennamen vor. Die entsprechenden Methoden finden sich in Abbildung 1

4 Pretty Printing

In diesem Abschnitt wird in Form einer Aufgabe nun der erste Algorithmus für den AST geschrieben. Die erste Aufgabe wird sein, aus dem AST wieder den Quelltext unserer Quellsprache zu schreiben.

4.1 Exception freier Rahmen für `Writer`

Hierzu schreiben wir uns eine Hilfsklasse, die es vereinfachen soll, Text in einen `Writer` zu schreiben. Diese Klasse bietet einen kleinen Rahmen um einen `Writer` herum. Beim

Java: AST

```
static PatternCase<OpExpr,Void> _OpExpr(Consumer<OpExpr> f) {
  return mkPattern(OpExpr.class, x->{f.accept(x);return null;});
}
static PatternCase<IntLiteral,Void> _IntLiteral(Consumer<IntLiteral> f){
  return mkPattern(IntLiteral.class, x->{f.accept(x);return null;});
}
static PatternCase<Var,Void> _Var(Consumer<Var> f){
  return mkPattern(Var.class, x->{f.accept(x);return null;});
}
static PatternCase<Assign,Void> _Assign(Consumer<Assign> f){
  return mkPattern(Assign.class, x->{f.accept(x);return null;});
}
static PatternCase<IfExpr ,Void> _IfExpr(Consumer<IfExpr> f){
  return mkPattern(IfExpr.class, x->{f.accept(x);return null;});
}
static PatternCase<WhileExpr,Void> _WhileExpr(Consumer<WhileExpr> f){
  return mkPattern(WhileExpr.class, x->{f.accept(x);return null;});
}
static PatternCase<FunCall,Void> _FunCall(Consumer<FunCall> f){
  return mkPattern(FunCall.class, x->{f.accept(x);return null;});
}
static PatternCase<Sequence,Void> _Sequence(Consumer<Sequence> f) {
  return mkPattern(Sequence.class, x->{f.accept(x);return null;});
}
static PatternCase<AST,Void> __(Consumer<AST> f) {
  return mkPattern(AST.class, x->{f.accept(x);return null;});
}
```

Abbildung 1: PatternCases mit Void als Ergebnistyp.

Schreiben werden eventuelle Exceptions gefangen und wieder als `RuntimeException` geworfen. Dieses ist notwendig, weil die Funktionen, die im Pattern-Matching verwendet werden, keine allgemeinen Ausnahmen werfen dürfen.

Zusätzlich bietet die Hilfsklasse die Möglichkeit, eine neue Zeile mit einer Einrückung zu schreiben. Die Einrückung kann erhöht und wieder verringert werden.

Als dritte Zusatzfunktion generiert die Klasse die Folge der natürlichen Zahlen. Diese werden wir bei der Code-Generierung verwenden, um dort unterschiedliche Labelnamen zu generieren. Die komplette Hilfsklasse befindet sich in Abbildung 2.

4.2 Stringdarstellung

Die Hilfsklasse `ExWriter` kommt bei unseren ersten Algorithmus für den AST in Einsatz. Für einfache Test wird ein `StringWriter` in die Klasse `ExWriter` gesteckt.

Java: AST

```
static class ExWriter{
  Writer w;
  String indent = "";
  void addIndent(){indent=indent+"  ";}
  void subIndent(){indent=indent.substring(2);}
  void nl(){write("\n"+indent);}

  int lbl=0;
  int next(){return lbl++;}

  public ExWriter(Writer w) {
    this.w = w;
  }
  void lnwrite(Object o) {
    nl();
    write(o);
  }
  void write(Object o) {
    try {
      w.write(o+"");
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Abbildung 2: Hilfsklasse zum formattierten Schreiben mit abgefangenen Ausnahmen.

Java: AST

```
default String show() {
  var r = new ExWriter(new StringWriter());
  show(r);
  return r.w.toString();
}
```

Die folgende Methode schreibt ein Funktionsdefinition:

Java: AST

```
static String show(Fun fd) {
  var r = new ExWriter(new StringWriter());
  r.write("fun ");
  r.write(fd.name);
  r.write("(");
  var first=true;
  for (var arg:fd.args){
    if (first) first=false;else r.write(", ");
    r.write(arg);
  }
  r.write(") = ");
  r.addIndent();
  r.nl();
  fd.body.show(r);
  return r.w.toString();
}
```

4.3 Schreiben des Programms mit Einrückungen

Jetzt können wir alle Knoten des AST mit Einrückungen schön schreiben.

Aufgabe 1

Vervollständigen Sie die Methode `show`, so dass das durch den AST dargestellte Programm vollständig als Quelltext geschrieben wird. Nutzen Sie dabei die Syntax, wie sie informell mit den Baumklassen vorgestellt wurde.
Berücksichtigen Sie, dass bei Operatorausdrücken wegen der Operatorpräzedenzen eventuell Klammern gesetzt werden müssen.

Java: AST

```
default Void show(ExWriter r) {
  return match
    (_Var       (v -> {r.write(v.name);})
    ,_IntLiteral(il-> {r.write(il.n+"");})
    ,__         (x ->
      {r.write("not yet implemented. Show: "+x);})
    );
}
```

Zum Ausprobieren der Lösung kann die `main`-Methode verwendet werden. Hier eine Beispielsitzung:

```
$ java -cp classes/ name.panitz.longStack.AST -i t1.ls
```

```
TUGS (Tiny Usable Great System) :? for help
> :s fac
fun fac(x) =
  if x=0
    then 1
    else x*fac(x-1)
> :s facIt
fun facIt(x) =
  {r := 1
  ;while not(x=0)
    do
      {r := r*x
      ;x := x-1
      }
  ;r
  }
> :q
```

5 Interpreter

Als nächstes soll ein Programm unserer Quellsprache direkt ausgewertet werden. Der einzige Datentyp, der verarbeitet wird, sind dabei ganze 64-Bit Zahlen. Wir überladen die default-Methode `ev` (für *evaluate*).

Die erste Version wertet einen Ausdruck aus, ohne dabei auf Funktionsdefinitionen zuzugreifen. Das Ergebnis ist dann die Zahl, die der Ausdruck berechnet.

Java: AST

```java
default long ev() {return ev(List.of());}
```

Diese ruft die überladenen Version auf, die als zusätzlichen Parameter eine Abbildung von Strings auf Funktionsdefinitionen enthält.

Java: AST

```java
default long ev(List<Fun> funs) {
  HashMap<String, Fun> fs = new HashMap<>();
  funs.forEach(fun->fs.put(fun.name, fun));
  return ev(fs,new HashMap<>());
}
```

Und diese schließlich ruft die Version auf, die zusätzlich eine Umgebung (*environment*) enthält, in der für Variablennnamen aktuell Werte gespeichert sind. Aus dieser Umgebung

können die aktuellen Werte für Variablen während der Auswertung gelesen werden und bei einer Zuweisung können dort neue Werte abgespeichert werden.

Aufgabe 2

Ergänzen Sie die fehlenden Fälle, zur Auswertung eines Ausdrucks.

Java: AST

```
default long ev(Map<String,Fun> fs, Map<String, Long> env) {
  return match
      ($IntLiteral(il -> il.n)
      ,$Var       (v  -> env.get(v.name))
      ,$OpExpr    (ae -> 42L)
      ,$IfExpr    (ie -> 42L)
      ,$Assign    (as -> {return 42L;})
      ,$WhileExpr (we -> {return 0L;})
      ,$Sequence  (sq -> {
        var r = 42L;
        /*TO DO*/;
        return r;
      })
      ,$FunCall   (fc -> {
        return 42L;
      })
  );
}
```

Mit Hilfe der `main`-Methode sollten Sie jetzt in der Lage sein, beliebige Ausdrücke auswerten zu lassen. Hier ein Beispielsitzung:

```
$ java -cp classes/ name/panitz/longStack/AST -i src/t1.ls
TUGS (Tiny Usable Great System) :? for help
> 17+4*2
25
> (17+4)*2
42
> fac(5)
120
> fib(10)
55
> :q
$
```

6 X86 Code Generierung

Wir werden in diesem Kapitel nun Assembler Code generieren. Der generierte Assembler ist in der Syntax für den Gnu Assembler. Er ist für eine 64-Bit Maschine mit X86 Befehlssatz. Der generierte Assembler kann von C-Programmen aus aufgerufen werden. Er hält sich also insbesondere an die Aufrufkonvention (engl. calling convention) des gcc.

Der Assembler arbeitet mit Registern und mit einem Stack. Gerechnet wird in der Regel mit Werten, die in Registern gespeichert sind. Zwischenergebnisse und ebenso lokale Variablen und Funktionsargumente werden auf dem Stack gespeichert.

6.1 Register

Wir betrachten zunächst die relevanten Register des Assemblers.

6.1.1 Register für Stackzeiger

Die beiden Register `%rsp` (Stackpointer) und `%rbp` (Basepointer) werden ausschließlich dafür verwendet, um auf Adressen des Stacks zu verweisen.

Im `%rsp` steht die aktuelle Adresse des ersten freien Platzes auf dem Stack notiert. Dabei ist übrigens zu beachten, dass der Stack nach unten wächst.

Im `%rbp` steht eine Adresse im Stack zu der relativ die Argumente und lokalen Variablen der Funktion zu finden sind, die aktuell ausgeführt wird. Es ist quasi der Zeiger auf die Basis des gerade ausgeführten Funktionsaufrufs auf dem Stack.

Zum Lösen der Aufgabe werden sie diese beiden Register nicht manuell verändern. Sie werden aber über den Basepointer auf die Argumente und lokalen Variablen einer Funktion auf dem Stack zugreifen.

6.1.2 Rechenregister

Wir werden ausschließlich Rechenoperationen auf den beiden Registern `%rax` und `%rbx` ausführen. Dem Register `%rax` kommt dabei die besondere Rolle zu, dass es immer das zuletzt berechnete Ergebnis enthalten wird. Insbesondere nach Ende der Ausführung einer Funktion steht das Ergebnis der Funktion in diesem Register.

6.1.3 Argumentregister

Sechs weitere Register werden verwendet, um Argumente an eine Funktion zu übergeben. Hat eine Funktion mehr als 6 Argumente, so werden weitere Argumente auf den Stack gespeichert. Funktionen, die ihrerseits im Rumpf andere Funktionen aufrufen, benötigen die sechs Register, um deren Argumente dort zu übergeben. Daher werden wir zu Beginn

einer Funktion alle die Argumente, die in einem Register übergeben wurden, auch direkt auf den Stack ablegen.

Die sechs Register für die Argumentübergabe sind namentlich in folgendem Array genannt.

Java: AST

```
String[] rs = {"%rdi","%rsi","%rdx","%rcx","%r8","%r9"};
```

6.2 Instruktionen

Nun betrachten wir die relevanten Assembler-Anweisungen für diese Aufgabe und deren Argumente.

6.2.1 Argumente

Die Anweisungen des Assemblers, die wir verwenden, haben 0, 1, oder 2 Argumente. Diese werden in vier verschiedene Formen auftreten.

Zahlenkonstanten Zahlenkonstanten können direkt als Literale geschrieben werden. Ihnen wird das Dollarzeichen vorangestellt. So bezeichnet also `$42` die Zahl 42.

Registerwerte Register sind Speicherzellen, in denen Zahlen stehen. Soll in einer Anweisung direkt mit einer solchen Speicherstelle gearbeitet werden, um dort eine Zahl hineinzuschreiben, oder herauszulesen, so wird direkt der Name des Registers notiert, zum Beispiel `%rax`.

Man nennt dieses auch direkte Adressierung.

Registeradressen Wenn die Zahl, die in einem Register gespeichert ist, erst die Adresse einer weiteren Speicherzelle ist, mit der gearbeitet werden soll, so schreibt man das Register in Klammern also z.B. `(%rax)`. Dieses entspricht der Pointer-Dereferenzierung in C. Man nennt dieses die indirekte Adressierung.

Man kann sogar noch einen Schritt weiter gehen. Dem geklammerten Register kann noch eine Zahl vorangestellt werden, also zum Beispiel `8(%rax)`. Dieses bedeutet: nimm die Zahl aus Register `%rax` als Adresse im Hauptspeicher. Addiere auf diese Adresse noch die acht drauf, gehe also 8 Byte weiter im Hauptspeicher und arbeite mit dieser Speicherzelle.

Label Es gibt Sprungbefehle, die zu einem im Assemblercode gesetzten Label springen. Dann ist das Argument der Anweisung ein Label. Label beginnen bei uns mit einem Punkt. Sie können einen beliebigen Bezeichner haben. Wir werden für Label den Buchstaben `L` benutzen und dann durchnummerieren, also `.L1`, `.L2`, `.L3`....

6.2.2 Stack Instruktionen

pushq Mit der Anweisung `pushq` können neue Daten auf den Stack gelegt werden. Es bewirkt, dass dabei auch das Register `%rsp` geändert wird. Die Anweisung hat ein Argument, mit dem spezifiziert wird, welche Daten auf den Stack gelegt werden sollen.

Wir geben ein paar Beispiele:

- Um zum Beispiel die Konstante 42 auf den Stack zu legen, verwendet man:
  `pushq $42`
- Um den Wert, der im Register `%rax` steht, auf den Stack zu legen, verwendet man:
  `pushq %rax`
- Um den Wert, der im Stack zwei Plätze oberhalb der Adresse in `%rbp` gespeichert ist, auf den Stack zu legen, verwendet man:
  `pusq 16(%rbp)`

  Die 16, weil wir ausschließlich mit 8-Byte Daten operieren.

popq Um Daten, die direkt als oberstes auf dem Stack liegen, wieder von Stack zu nehmen und woanders zu speichern, dient die Anweisung `popq`. Das Argument ist dann das Ziel, in dem die Daten gespeichert werden. So wird mit `popq %rax` das oberste Stackelement in das Register `%rax` gespeichert.

6.2.3 Daten verschieben

Die beiden Stackbefehle verschieben schon Daten, allerdings immer vom Stack oder auf den Stack. Um zwischen beliegen Speicherstellen Daten zu verschieben, gibt es die Anweisung `moveq`. Sie hat zwei Argumente: die Quelle und das Ziel der Daten. Die beiden Argumente werden durch ein Komma getrennt.

Wir geben ein paar Beispiele:

- `moveq %rax %rbx` schreibt die in Register `%rax` gespeicherte Zahl in das Register `%rbx`.
- `moveq (%rax) %rbx` schreibt die Zahl, die in der Speicherzelle gespeichert ist, deren Adresse in Register `%rax` staht, in das Register `%rbx`.

6.2.4 Rechnen

Arithmetische Operationen Für die drei arithmetischen Operatoren unserer Sprache gibt es die direkten drei Anweisungen, die jeweils zwei Parameter haben: `imulq`, `subq` und `addq`. Sie verrechnen die Daten der beiden Argumente und speichern das Ergebnis in das Register des zweiten Arguments. So addiert z.B. `addq %rbx, %rax` die Zahlen in den Registern `%rbx` und `%rax` und speichert das Ergebnis in Register `%rax`.

Vergleichsoperatoren Zwei Werte können mit mit der Anweisung `cmpq` verglichen werden. Das Ergebnis ist in Form eine Subtraktion. Für zum Beispiel `cmpq %rax, %rbx` gilt: Wenn die Zahl in `%rax` größer ist, ist das Ergebnis positiv, wenn sie kleiner ist negativ und bei zwei gleichen Zahlen ist es 0.

6.2.5 Sprünge

bedingte Sprünge Für bedingte Sprünge verwenden wir ausschließlich die Anweisung `jne`. Sie hat einen Parameter. Dieser ist ein Label. Wenn die vorhergehende `cmpq`-Anweisung nicht die Gleichheit ergeben hat, dann wird als nächste Anweisung zum angegebenen Label.

unbedingte Sprünge Wenn man unter jeder Bedingung zu einem bestimmten Label springen will, kann man die Anweisung `jmp` verwenden, die das entsprechende Label als Argument hat.

6.3 Berechnung der lokalen Variablen

Bevor wir beginnen können, Assembler zu generieren, benötigen wir die Menge aller lokalen Variablen innerhalb einer Funktion. Hierfür ist eine eigene Methode zu schreiben, die diese aus einem AST extrahiert.

Java: AST

```
default Set<String> getVars(){
  var r = new TreeSet<String>();
  getVars(r);
  return r;
}
```

Lokale Variablen brauchen mindestens eine Zuweisung innerhalb der Funktionsrumpfs. Somit wird bei jeder Zuweisung eine potentielle weitere lokale Variable auf der linken Seite gefunden. Ansonsten ist der gante AST zu traversieren, um nach weiteren Zuweisungen zu suchen.

Aufgabe 3

Ergänzen Sie die fehlenden Fälle, um die Menge der Variablen eines Programms zu berechnen.

Java: AST

```
default void getVars(TreeSet<String> r) {
  match
  (_Assign    (as -> {r.add(as.v.name);as.right.getVars(r);})
   ,__         (x  -> {})
  );
}
```

6.4 X86 Code für Funktionen

Wir können nun beginnen, Assembler zu schreiben. Da Funktionen mit der Parameterübergabe für den Anfänger etwas zu komplex sind, ist die Code-Generierung für Funktionsdefinitionen und Funktionsaufrufe bereits umgesetzt. Alle anderen Sprachkonstrukte unserer Quellsprache sind als Aufgabe umzusetzen.

In diesen Abschnitt folgt die Beschreibung der Code-Generierung für Funktionsdefinitionen. Für eine Funktionsdefinition ist Assembler-Code zu generieren und in einen Writerobjekt zu schreiben.

Java: AST

```
static void asm(Fun f,ExWriter r) {
```

6.4.1 Funktionskopf

Den Beginn einer Funktion beschreiben im Assembler ein paar globale Labels, in denen der Funktionsname gesetzt wird.

Java: AST

```
r.nl();
r.lnwrite(".globl  "+f.name);
r.lnwrite(".type  "+f.name+", @function");
r.lnwrite(f.name+":");
r.addIndent();
```

6.4.2 Stackpointer verwalten

Die erste Aufgabe ist es, die Stackpointer zu verwalten. Hierzu wird der Wert des aktuellen Basepointers auf den Stack geschrieben. Der aktuelle Stackpointer wird dann zum Basepointer während der Ausführung der Funktion.

Java: AST

```
r.lnwrite("pushq %rbp");
r.lnwrite("movq %rsp, %rbp");
```

Am Ende, beim Verlassen der Funktion wird wieder der alte Zustand der beiden Register hergestellt werden.

6.4.3 Parameter von Registern auf den Stack

Die Aufrufkonvention lädt die ersten 6 Argumente in die Register, die in der Reihung `rs` benannt sind. Die weiteren Argumente befinden sich auf dem Stack. Dieses ist eine effizientere Aufrufvariante, als wenn alle Argumente auf den Stack gelegt werden. Allerdings funktioniert diese Variante nur, wenn unsere Funktion selbst keine andere Funktion aufruft. Um hier einheitlich arbeiten zu können legen wir somit als erstes auch die Argumente, die in Registern übergeben werden, auf den Stack.

Dabei bauen wir eine Abbildung von Variablen und Parameternamen zu Zahlen auf. Diese Abbildung sagt für eine Variable, wie viel Schritte vom Basepointer entfernt, sich die Variable auf dem Stack gespeichert befindet.

Als erstes schauen wir, wie viele Argumente über Register übergeben wurden und legen die Abbildung an.

Java: AST

```
var registerArgs = Math.min(rs.length, f.args.size());
var env = new HashMap<String,Integer>();
```

Nun werden nacheinander die Argumente aus den Registern auf den Stack gelegt und in der Abbildung vermerkt, wie viele Speicherplätze vom Basepointer sie entfernt liegen. Das erste Argument liegt direkt über den Basepointer. Deshalb beginnen wir mit -8.

Java: AST

```
var sp=-8;
for (var i=0;i<registerArgs;i++) {
  r.lnwrite("movq "+rs[i]+", "+sp+"(%rbp)");
  env.put(f.args.get(i), sp);
  sp = sp-8;
}
```

6.4.4 Lokale Variablen in Umgebung vermerken

Nun sind die per Register übergebenen Argumente auf dem Stack und in der Umgebung vermerkt. Als nächstes sind Stackpositionen der lokalen Variablen zu vermerken. Dieses sind die Variablen, denen etwas im Funktionsrumpf zugewiesen wird, ohne die Argumente.

Wir haben zwar noch keine iunitialen Werte, für diese Variablen, aber müssen der Stackpointer entsprechend weiter setzen, damit dieser Platz auf dem Stack nicht durch weitere push-Befehle überschrieben wird.

Java: AST

```
var vs = f.body.getVars();
vs.removeAll(f.args);
for (var v:vs) {
  env.put(v, sp);
  sp = sp-8;
}
r.lnwrite("subq $"+(-sp)+", %rsp");
```

6.4.5 Stackparameter in Umgebung vermerken

Gibt es mehr Argumente, als in den Registern übergeben werden kann, liegen die weiteren Argumenten bereits auf dem Stack. Auch dieses ist noch in der Umgebung zu vermerken.

Java: AST

```
sp = 16;
for (var i=registerArgs;i<f.args.size();i++) {
  env.put(f.args.get(i), sp);
  sp+=8;
}
```

6.4.6 Code für Funktionsrumpf

Jetzt kann der eigentliche Assembler-Code für den Funktionsrumpf generiert werden.

Java: AST

```
f.body.asm(Map.of(),env,r);
```

6.4.7 Stackpointer Zurücksetzen beim Verlassen

Beim Verlassen der Funktion, wird der alte Zustand des Stackpointers und des Basepointers wieder hergestellt. Die Funktion schließt mit der Anweisung `ret`, die dafür sorgt, dass zurück zur Aufrufstelle des Funktionsaufrufs gesprungen wird.

Java: AST

```
  r.lnwrite("movq %rbp, %rsp");
  r.lnwrite("popq %rbp");
  r.lnwrite("ret");
  r.subIndent();
}
```

6.5 Code für Ausdrücke

Jetzt kommt der spannende teil. Für alle Ausdrücke der Quellsprache ist Assembler zu generieren.

Für einfache Test, ohne Funktionsaufrufe und Variablen ist die folgende Methode.

Java: AST

```
default String asm() {
  var r = new ExWriter(new StringWriter());
  asm(new HashMap<>(),new HashMap<>(),r);
  return r.w.toString();
}
```

Eine Statische Methode sei für ein ganzes Programm, das aus einer Liste von Funktionsdefinitionen besteht, gegeben.

Java: AST

```
static String asm(List<Fun> fs) {
  var r = new ExWriter(new StringWriter());
  fs.forEach(f->asm(f,r));
  return r.w.toString();
}
```

Die eigentliche Funktion zur Codegenerierung hat drei Argumente:

- Der Abbildung, die für Funktionsnamen die Funktionsdefinition bereit hält.
- Die Umgebung: das ist die Abbildung von Variablennamen auf Positionen relativ zum Basepointer.
- Das Objekt, in das der Assembler-Code textuell geschrieben wird.

Java: AST

```
default void asm(Map<String,Fun> fs,Map<String, Integer> e, ExWriter r){
  match
```

Der Code, der für einen Ausdruck generiert wird, soll immer Code sein, der das Ergebnis für den Ausdruck berechnet, so dass dieses anschließend im Register `%rax` gespeichert ist.

6.5.1 Zahlenkonstanten

Der Assembler-Code für Zahlenkonstanten ist dann denkbar einfach. Die entsprechende Konstante wird in das Register `%rax` gespeichert.

Java: AST

```
(_IntLiteral(il -> {r.lnwrite("movq $"+il.n+", %rax");})
```

6.5.2 Varieablenzugriff

Bei einem Variablenzugriff wird in der Umgebung `e` die relative Stack-Adresse vom Basepointer ausgehend nachgeschlagen. Aus dieser wird dann vom Stack der Wert der Variablen in das Register `%rax` geladen

Java: AST

```
,_Var       (v -> {r.lnwrite("movq "+e.get(v.name)+"(%rbp), %rax");})
```

Soweit die ersten beiden Fälle für Ausdrücke. Die weiteren werden jetzt als Aufgabe formuliert. Um ein wenig Hinweise für bestimme Fälle zu bekommen, kann man eine C-Funktion schreiben, die ein äquivalentes Konstrukt enthält, und sich mit dem `gcc` durch die Option `-S` den Assembler generieren lassen. Dort lässt sich dann analog betrachten, was für ein Code zu generieren ist.

6.5.3 Zuweisungen

Bei einer Zuweisung ist der Code für die linke Seite der Zuweisung zu generieren. Dieser Code wird dafür sorgen, dass im Register `%rax` das Ergebnis der linken Seite steht. Dieses ist dann auf dem Stack an der Position zu speichern, die für die Variable auf der linken Seite der Zuweisung in der Umgebung vermerkt ist.

Aufgabe 4

Schreiben Sie in der Code-Generierung das Pattern-Case für die Zuweisung.

6.5.4 Operator-Ausdrücke

Für einen Operatorausdruck müssen wir für die beiden Operanden Code erzeugen.

Es empfiehlt sich wie folgt vorzugehen:

- Erzeuge Code für den rechten Operanden.
- Push das Ergebnis des rechten Operanden auf den Stack.
- Erzeuge Code für den linken Operanden.
- Hole das Ergebnis des rechten Operanden vom Stack und lade es in Register `%rbx`.
- Generiere die Anweisung für den Operator. Für den Operator = ist dabei folgende Anweisungsfolge zu generieren:

```
cmpq %rax, %rbx
sete %al
movzbl %al, %eax
```

Aufgabe 5

Schreiben Sie in der Code-Generierung das Pattern-Case für Operatorausdrücke.

6.5.5 Code-Blöcke

Bei einer Sequenz von Ausdrücken in Form eins Code-Blocks ist für diese nacheinander Code zu generieren. Damit steht abschließend das Ergebnis des letzten Ausdrucks der Sequenz in Register `%rax`.

Aufgabe 6

Schreiben Sie in der Code-Generierung das Pattern-Case für Code-Blöcke.

6.5.6 While-Schleifen

Bei der Codegenerierung für Schleifen sind zwei Label zu generieren. Eines für den Beginn des Schleifenrumpfes und eines für den Beginn des Bedingungstests.

Es empfiehlt sich folgendes Vorgehen:

- Erzeuge zwei neue Label `l1` und `l2`.
- Generiere den Befehl der zu Label `l1` springt.
- Generiere im Assembler das Label `l2`.
- Generiere Code für den Schleifenrumpf.

- Generiere im Assembler das Label `l1`.
- Generiere Code für die Schleifenbedingung.
- Generiere den Code, der prüft, ob die Schleifenbedingung 0 als Ergebnis hatte: `cmpq $0, %rax`.
- Generiere den bedingten Sprung zum Label `l2`, wenn der Test zuvor nicht wahr war.

Aufgabe 7

Schreiben Sie in der Code-Generierung das Pattern-Case für Schleifen.

6.5.7 If-Bedingungen

Auch für die Realisierung der Bedingungsausdrücke sind zwei Sprunglabel `l1` und `l2` zu erzeugen. Label `l1` steht zu Beginn der then-Alternative, das Label `l2` am Ende des Gesamtausdrucks.

Gehen Sie wie folgt vor:

- Generieren Sie Code für die Bedingung.
- Generieren Sie die Anweisung, die das Ergebnis der Bedingung mit 0 vergleicht.
- Generieren Sie einen bedingten Sprung zu Label `l1`, wenn das letzte Ergebnis nicht gleich ergab.
- Generieren Sie Code für die else-Alternative.
- Generieren Sie einen Sprung zu Label `l2`.
- Schreiben Sie das Label `l2`.
- Generieren Sie Code für die then-Alternative.
- Schreiben Sie das Label `l2`.

Aufgabe 8

Schreiben Sie in der Code-Generierung das Pattern-Case für Bedingungen.

6.5.8 Funktionsaufrufe

Nicht als Aufgabe zu realisieren sind Funktionsaufrufe. Hierbei wird Code für die Argumente generiert. Diese werden zunächst in die Argumentregister geladen, wenn es weitere Argumente sind, werden sie auf den Stack gelegt. Als letztes wird die Anweisung `call` mit dem entsprechenden Funktionsnamen generiert

Java: AST

```
  ,_FunCall   (fc ->
    {for (int i=0;i<Math.min(5, fc.args.size());i++) {
      fc.args.get(i).asm(fs, e, r);
      r.lnwrite("movq %rax, "+rs[i]);
    }
    ;fc.args.stream().skip(rs.length).forEach(arg -> {
      arg.asm(fs, e, r);
      r.lnwrite("pushq %rax");
    });
    ;r.lnwrite("call "+fc.name)
    ;})
  );
}
```

7 Eine main-Methode für Compiler oder Interpreter

Um alles zu testen, den Compiler, den Pretty-Printer und den Interpreter, dient diese `main`-Methode,

7.1 Hilfestellungen

Es gibt in der Interpreterschleife eine Hilfestellung.

Java: AST

```
static String[] help =
  {"<expr>          evaluate <expr>"
  ,"':q'            quits the interpreter"
  ,"':defs'         System.out.println();shows defined function names"
  ,"':s <funname>'  prints function definition of <funname>"
  ,"':?             this help"};

  static void printHelp(){
    for (var h:help)System.out.println(h);
  }
```

7.2 Anwender Frontend des Systems

Die Methode kennt zwei Modi. Die Compilierung und die Interpreterschleife. Beide lesen eine Datei mit Funktionsdefinitionen ein.

Java: AST

```
public static void main(String[] args) throws Exception {
  if (args.length==0){
    System.out.println
      ("usage: java name.panitz.longStack.AST [-i] filename");
    System.out.println
      ("  where -i starts interpreter "
      +"otherwise assembler code is generated");
    return;
  }
```

Zunächst wird geprüft, ob Assembler generiert werden soll oder eine Interpreter-Session gestartet werden soll.

Java: AST

```
var interpreter = args[0].equals("-i");
var funDefs =
↪  LongStackParser.parseFunDefs(args[0].equals("-i")?args[1]:args[0]);
```

Zum Start des Interpreters wird auf die Hilfe verwiesen.

Java: AST

```
if (interpreter){
  var in =  new BufferedReader(new InputStreamReader(System.in));
  System.out.println("TUGS (Tiny Usable Great System) :? for help");
```

Die eigentliche Lesen-Evaluieren-Drucken-Schleife (*read eval print loop*) kurz *REPL*).

Die nächste Zeile wird gelesen:

Java: AST

```
while (true){
  System.out.print("> ");
  var ln = in.readLine();
```

Soll das Programm beendet werden?

Java: AST

```
if (ln.equals(":q")) break;
```

Braucht jemand Hilfe?

Java: AST

```
if (ln.equals(":?")){printHelp();continue;}
```

Sollen die Funktionsdefinitionen aufgelistet werden?

Java: AST

```
if (ln.equals(":defs")){
  funDefs.parallelStream()
   .forEach(fd->System.out.println(fd.name));
  continue;
}
var showFunction = ln.startsWith(":s ");
try{
```

Soll eine Funktion angezeigt werden?

Java: AST

```
if (showFunction){
  var funname = ln.substring(2).trim();
  Fun fundef = funDefs.stream()
    .reduce
      (null
      ,(r,fd)->(r==null&&fd.name.equals(funname))?fd:r
      ,(fd1,fd2)->fd1==null?fd2:fd1);
  if (fundef!=null)
    System.out.println(show(fundef));
  else System.out.println("unknown function: "+funname);
```

Es soll also schließlich ein Ausdruck ausgewertet werden:

Java: AST

```
    }else{
      System.out.println
       (LongStackParser.parseExpr(ln).ev(funDefs));
    }
  }catch(Exception e){
     System.out.println(e);
  }
}
```

Wenn keine REPL gestartet wurde, sondern Assembler generiert werden soll:

Java: AST

```
    }else{
      var out = new FileWriter
          (args[0].substring(0,args[0].lastIndexOf('.'))+".s");
      var o = new ExWriter(out);
      for (var fun:funDefs) asm(fun,o);
      o.nl();
      out.close();
    }
  }
}
```

Literatur

[BG18] G. Bierman and B. Goetz, "Pattern matching for java," 9 2018, [accessed 25-May-2019]. [Online]. Available: https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html

[DW19] D. Dietrich and R. Winkler, "Vavr user guide," 2 2019, [accessed 25-May-2019]. [Online]. Available: https://www.vavr.io/vavr-docs/

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[Goe19] B. Goetz, "Data classes and sealed types for java," 2 2019, [accessed 25-May-2019]. [Online]. Available: https://cr.openjdk.java.net/~briangoetz/amber/datum.html

[Par18] N. Parlog, "Definitive guide to switch expressions in java 12," 11 2018, [accessed 25-May-2019]. [Online]. Available: https://blog.codefx.org/java/switch-expressions/