Compilerbau

(Entwurf)

WS 04/05

Sven Eric Panitz

FH Wiesbaden

Version 28. Januar 2005

Die vorliegende Fassung des Skriptes ist ein roher Entwurf für die Vorlesung des kommenden Semesters und wird im Laufe der Vorlesung erst seine endgültige Form finden. Kapitel können dabei umgestellt, vollkommen revidiert werden oder gar ganz wegfallen.

Dieses Skript entsteht begleitend zur Vorlesung des WS 04/05 vollkommen neu. Es stellt somit eine Mitschrift der Vorlesung dar. Naturgemäß ist es in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler werden sich im Eifer des Gefechtes nicht vermeiden lassen und wahrscheinlich in nicht geringem Maße auftreten. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Kapitel 1

Einführung

There's nothing you can do
that can't be done
Nothing you can sing
that can't be sung
Nothing you can say
but you can learn
how to play the game.
It's easy.

Lennon/McCartney

Compilerbau ist wahrscheinlich eine der ältesten Disziplinen der Informatik. Ein Compiler ist unbedingte Voraussetzung, um in einer halbwegs interessanten Programmiersprache Software zu entwickeln. Daher war eine der ersten und wichtigsten Aufgaben der Informatik der Compilerbau und daher resultierte auch ein reges Interesse am Compilerbau in den 60er und 70er Jahren des letzten Jahrhunderts. Compilerbau stellt somit in seinen Techniken auch eine sehr reife und gut verstandene Disziplin der Informatik dar. Eine Reihe von Lehrbüchern zum Compilerbau sind trotz ihres Alters in vielen Aspekten noch relevant und haben sich teilweise als Klassiker etabliert und eine Reihe von Neuauflagen erfahren [AU77][WG84]. Eine ausführliche Literaturliste finden sich den Skripten von Herrn Weber [Web02] und Herrn Grude [GKS01].

Heutzutage wird nur ein verschwindend geringer Teil der Informatiker während seiner beruflichen Laufbahn in die Verlegenheit kommen, einen Compiler zu entwickeln. Eine Vielzahl interessanter und ausgereifter Compiler steht zur Verfügung und der Compilerbau stellt eher eine Spezialdisziplin dar. Aus verschiedenen Gründen lohnt es sich für jeden Informatiker, sich ausgiebig mit Compilerbau zu beschäftigen:

- um eine profundes Kenntnis von Programmiersprachen zu erlangen, ist eine Beschäftigung mit ihrer Implementierung unerläßlich.
- viele Grundtechniken der Informatik und elementare Datenstrukturen wie Keller, Listen, Abbildungen, Bäume, Graphen, Automaten etc. finden im Compilerbau Anwendung.
- aufgrund seiner Reife gibt es hervorragende Beispiele von formaler Spezifikation im Compilerbau.
- mit dem Gebiet der formalen Sprachen berührt der Compilerbau interessante Aspekte moderner Linguistik.

- die Unterscheidung von Syntax und Semantik ist eine grundlegende Technik fast aller formalen Systeme.

1.1 Was ist ein Compiler

Die adequate Übersetzung des englischen *compiler* ins deutsche ist *Übersetzer*.[1] Ein Übersetzer übersetzt einen Satz von einer Sprache in eine andere Sprache. Man spricht von Quell- und Zielsprache. Dabei ist die wichtigste Eigenschaft eines Übersetzers, den Inhalt nicht zu verfälschen. Man geht davon aus, daß die Sätze von Quell- und Zielsprache jeweils eine Bedeutung haben, man spricht von der Semantik.

In der Regel übersetzt ein Compiler ein Programm einer Programmiersprache in eine Codefolge einer Maschine. Es gibt zwar auch Compiler, die wieder Code einer Hochsprache erzeugen, allerdings dürften zum überwiegenden Teil diese Compiler die Hochsprache als eine Art Pseudoassembler benutzen.

Wir können also davon ausgehen, daß die Zielsprache eines Compilers eine Maschinensprache ist. Als Maschinensprache verstehen wir dabei eine Sprache, deren Programme aus einer Folge einfacher Befehle bestehen. Damit ist die Syntax einer Maschinensprache trivial. Die Semantik einer Maschinensprache wird als Zustandsmaschine beschrieben. Die Maschine hat einen Zustand, der durch die Ausführung eines Befehls auf deterministische Weise manipuliert wird.

Es ist dabei unerheblich, ob die Maschine als konkrete Hardware existiert, oder nur eine gedachte abstrakte, eine virtuelle Maschine ist, die über Software realisiert ist.

1.2 Aufbau eines Compilers

Ein Compiler 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 Compilers ermöglicht Flexibilität in zwei Richtungen:

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

[1] Anders als z.B. im Französischen das Wort *compilateur* hat sich im Deutschen der Begriff *Kompilator* nicht durchgesetzt.

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 Compilers 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 Compiler berechnet werden ....

1.2.1 Kleines Beispiel

Wir betrachten im folgenden an einem kleinem Programmausschnitt die einzelnen Phasen eines Compilers. Ausgangspunkt ist das Programm in textueller Form einer Zeichenkette.

```
result := 1;
for (i in 1 to 5) result:=result*i;
```

Dieses Programm berechnet die Fakultät von 5.

Lexikalische Analyse

In der lexikalischen Analyse wird die Zeichenkette in Wörter zerteilt. Man spricht von Token.

$\boxed{\textbf{ident}\ \texttt{result}}$ $\boxed{\textbf{assign}}$ $\boxed{\textbf{Num}\ \texttt{1}}$ $\boxed{\textbf{Semicolon}}$
$\boxed{\textbf{for}}$ $\boxed{\textbf{Lpar}}$ $\boxed{\textbf{ident}\ \texttt{i}}$ $\boxed{\textbf{Num}\ \texttt{1}}$ $\boxed{\textbf{in}}$ $\boxed{\textbf{Num}\ \texttt{5}}$ $\boxed{\textbf{Rpar}}$ $\boxed{\textbf{ident}\ \texttt{result}}$ $\boxed{\textbf{assign}}$ $\boxed{\textbf{ident}\ \texttt{result}}$
$\boxed{\textbf{OpMult}}$ $\boxed{\textbf{ident}\ \texttt{i}}$ $\boxed{\textbf{Semicolon}}$

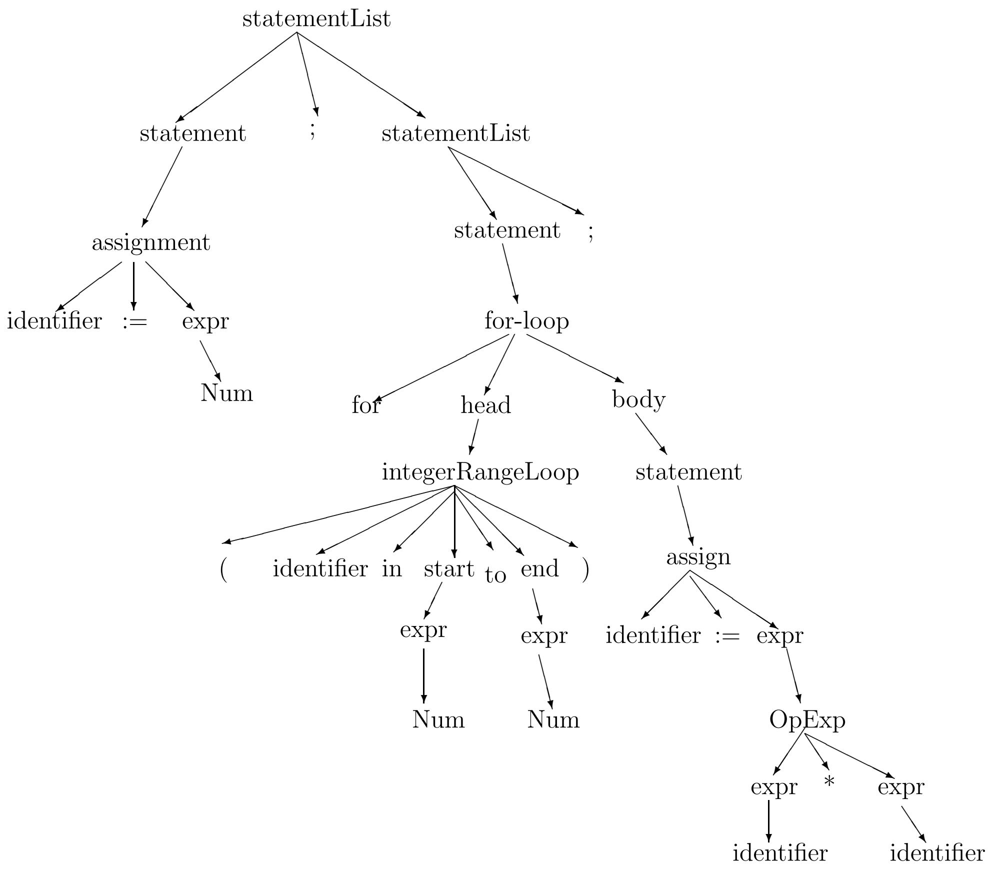


Abbildung 1.1: Syntaxbaum.

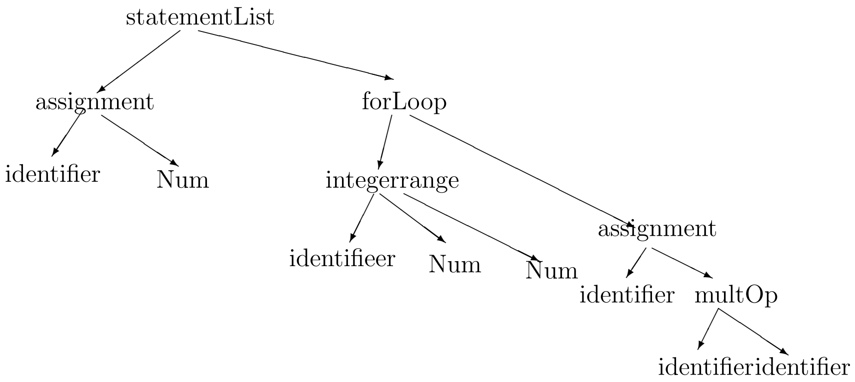


Abbildung 1.2: Strukturbaum

Syntaktische Analyse

Die Folge der Token ist die Eingabe der syntaktischen Analyse. Ein erster Syntaxbaum wird erzeugt. Der Syntaxbaum unseres Beispiels findet sich in Abbildung 1.1.

Ein Syntaxbaum ist aufgrund dessen, daß er direkt die grammatischen Regeln, die zur Erzeugung des Programms notwendig sind, darstellt, recht komplex. Viele Knoten eines Syntaxbaums sind zur weiteren Analyse des Programms nicht mehr notwendig. Dieses sind Knoten mit den Trennzeichen (z.B. das Semikolon) und Schlüsselwörtern der Sprache, sowie oft auch innere Knoten des Baumes. Daher wird ein Syntaxbaum in der Regeln vereinfacht. Ein sogenannte Strukturbaum wird erzeugt. Oft geschieht dieses direkt als Ergebnis der syntaktischen Analyse, ohne daß physisch der Syntaxbaum jemals erzeugt wurde. Für unser Beispiel könnte ein Syntaxbaum wie in Abbildung 1.2 dargestellt aussehen. Ein Strukturbaum ist in der Regel wesentlich kompakter als ein vollständiger Syntaxbaum.

Semantische Analyse

Über den Syntaxbaum können jetzt verschiedene semantische Analyse geschickt werden. Eine erste Analyse wird dabei überprüfen, ob alle Bezeichner von Variablen vor ihrer Benutzung bereits deklariert wurden. Eine weitere wichtige semantische Analyse ist der Typcheck oder gegebenenfalls die Typinferrenz.

- Beim Typcheck werden die Typannotation, die der Programmier im Programm gemacht hat, auf Konsistenz überprüft. Es wird z.B. überprüft, daß einer Variablen, die vom Typ String deklariert wurde, auch nur Ausdrücke vom Typ String zugewiesen wurden.
- Bei der Typinferrenz stehen keine expliziten Typinformationen im Programmtext. Der Compiler inferriert diese aus den Kontext der Benutzung. Für eine Variable wird geguckt, was für Typen die Ausdrücke haben, die ihr zugewiesen werden, und somit auf den Typ für die Variable geschlossen. Sprachen wie C, Java, Pascal benutzen fast durchgenend nur einen Typcheck, aber in C++ gibt es ein Konstrukt, für das eine Typinferrenz notwendig

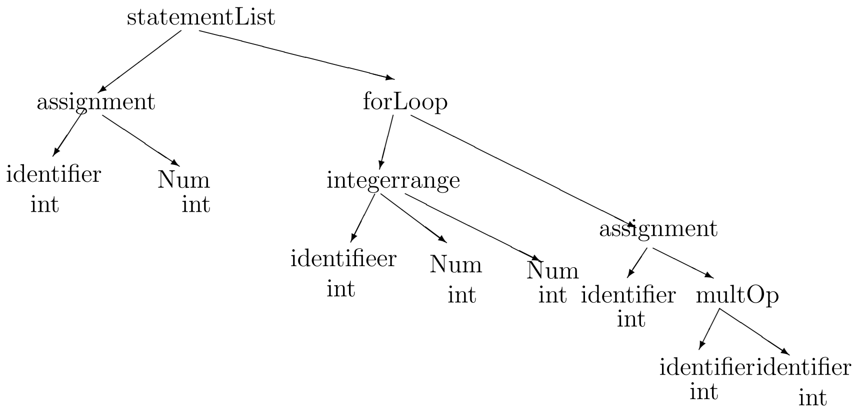


Abbildung 1.3: getypter Strukturbaum

ist: die Anwendung von Schablonen- (template) Funktionen. Sprachen, die komplett auf eine Typinferrenz basieren sind: ML[MTH90], Haskell [PHA+97], Clean [Pv97].

In unserem Beispiel gab es keinerlei Typannotationen im Programmtext. Der Compiler muß die Typinformation inferrieren. Die inferrierten Typen können als Attribute an den Baum notiert werden. Wir erhalten den getypten Strukturbaum aus Abbildung 1.3.

In einen nächsten Schritt kann der Compiler komplexe hochsprachliche Konstrukte durch einfachere ersetzen. Hierzu wird oft eine Kernsprache definiert, die eine Untermenge der Programmiersprache ist. In unseren Fall haben wir eine for-Schleife über einen Zahlenbereich als Sprachkonstrukt benutzt. Ein solches Konstrukt wird in der Regel durch ein äquivalentes einfacheres ersetzt. In unseren Fall könnte dies durch eine while-Schleife der folgenden Art geschehen:

```
i := 1; while (i<5){result:=result*i;i:=i+1;}
```

Eine solche Programmtransformation wird auf dem Baum gemacht. Wir erhalten den Baum aus Abbildung 1.4.

Es sind auf den Strukturbaum beliebig viele und beliebig komplexe Analysen und Transformationen denkbar. Hier finden sich interessanteste Forschungen im Bereich des Compilerbaus.

Codegenerierung

Wenn alle diese Analysen abgeschlossen sind, kann schließlich die Befehlssequenz der Zielmaschine generiert werden. Für unser Beispiel könnte eine solche Befehlssequenz wie folgt aussehen:

```
PushInt 1
Store result
PushInt 1
```

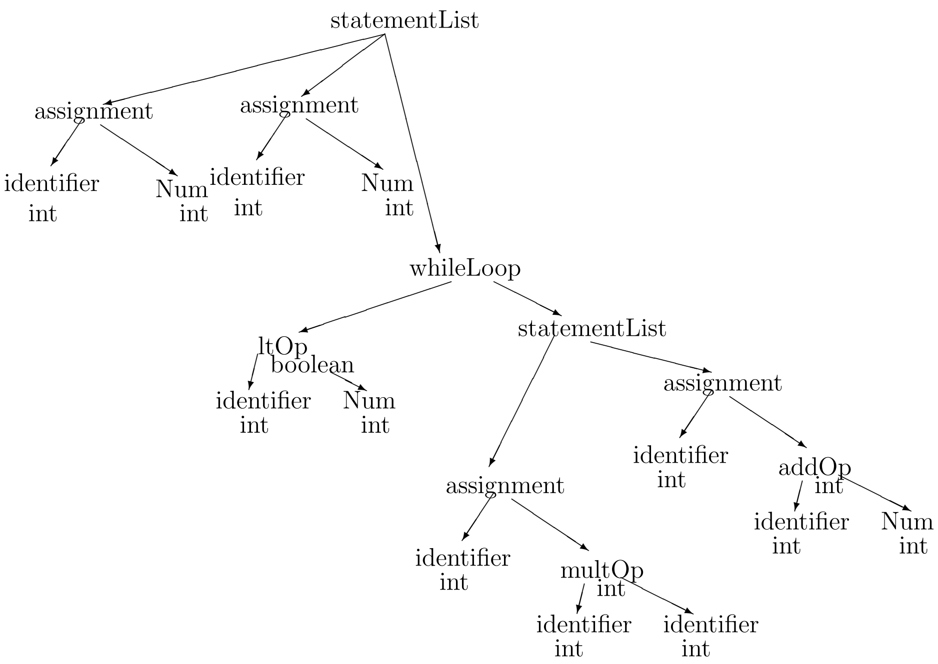


Abbildung 1.4: In Kernsprache transformierter Strukturbaum.


```
Store i
Label: whileStart
Push i
PushInt 5
Lt
JumpTrue bodyStart
Jump whileEnd
label: bodyStart
Push result
Push i
Mult
Store result
Push i
PushInt 1
Add
Store i
Jump whileStart
Label: whileEnd
```

1.3 Programmiersprachen

In den letzten 50 Jahren sind unzählige Programmiersprachen entwickelt worden. Programmiersprachen lassen sich nach vielen unterschiedlichen Eigenschaften charakterisieren. Je nach den Eigenschaften einer Programmiersprache, werden ganz unterschiedliche Anforderungen an einen Compiler gestellt.

1.3.1 Klassifizierung von Programmiersprachen

Es gibt mittlerweile mehr Programmiersprachen als natürliche Sprachen.[2] Die meisten Sprachen führen entsprechend nur ein Schattendasein und die Mehrzahl der Programme konzentriert sich auf einige wenige Sprachen. Programmiersprachen lassen sich nach den unterschiedlichsten Kriterien klassifizieren.

Hauptklassen

Everybody seems to think
I'm lazy
I don't mind
I thing they're crazy
Running everywhere
at such a speed
Till they find
there's no need.
Lennon/McCartney

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

[2] Wer Interesse hat, kann im Netz einmal suchen, ob er eine Liste von Programmiersprachen findet.

- **imperativ** (C, Pascal, Fortran, Cobol): das Hauptkonstrukt dieser Sprachen sind Befehle, die den Speicher manipulieren.
- **objektorientiert** (Java, C++, C#, Eiffel, Smalltalk): Daten werden in Form von Objekten organisiert. Diese Objekte bündeln mit den Daten auch die auf diesen Daten anwendbaren Methoden.
- **funktional** (Lisp, ML, Haskell, Scheme, Erlang, Clean): Programme werden als mathematische Funktionen verstanden und auch Funktionen können Daten sein. Dieses Programmierparadigma versucht, sich möglichst weit von der Architektur des Computers zu lösen. Veränderbare Speicherzellen gibt es in rein funktionalen Sprachen nicht und erst recht keine Zuweisungsbefehle. Man kann zwei Arten der Auswertung funktionaler Sprachen unterscheiden:
  - **strikt**: bei einem Funktionsaufruf werden erst die Argumente zu ihrem Ergebnis ausgewertet, bevor der Funktionsrumpf mit den Werten der Argumente instanziiert wird.
  - **nicht-strikt**: die Argumente werden erst dann und nur soweit ausgewertet, wie sie unbedingt notwendig sind, um das Ergebnis des Funktionsaufrufs zu errechnen. Um z.B. die Länge einer Liste zu berechnen, brauchen die eigentlichen Elemente der Liste nicht betrachtet zu werden.
- **Skriptsprachen** (Perl, AWK, PHP): solche Sprachen sind dazu entworfen, einfache kleine Programme schnell zu erzeugen. Sie haben meist kein Typsystem und nur eine begrenzte Zahl an Strukturierungsmöglichkeiten, oft aber eine mächtige Bibliothek, um Zeichenketten zu manipulieren.
- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

funktionale Sprachen

Im Laufe der Vorlesung werden wir einen Compiler für eine rein funktionale Sprache entwickeln. Die Wahl fiel auf eine funktionale Sprache, weil sich in darin mit wenig Syntax und wenig Spezialkonstrukten bereits komplexe Algorithmen schreiben lassen, so daß sich der Compiler der funktionalen Sprache in dieser Sprache selbst implementieren läßt.

Programme einer funktionalen Sprache bestehen aus einer Menge von Funktionsdefinitionen und Datenkonstruktordeklarationen. Es gibt nur Ausdrücke, keine Befehle (wie z.B. Schleifen). Insbesondere gibt es keinen Zuweisungsbefehl. Variablen enthalten keine veränderbaren Werte. Sie haben einen festen Wert.

In der Funktion `main` eines funktionalen Programms ist ein Ausdruck definiert, dessen Wert zu errechnen ist. Dieses geschieht durch Anwendung der Funktionsdefinitionen. Man spricht von Reduktion.

Beispiel:
In unserem ersten kleinen Testprogramm, definieren wir zwei Konstruktoren, die die Wahrheitswerte darstellen sollen. Die Konstruktoren sind nullstellig.

Wir gehen davon aus, daß die Standardarithmetik in gewohnter Weise eingebaut ist und Vergleichsoperatoren die definierten Wahrheitswerte als Ergebnis haben.

Zusätzlich wird die Funktion `wenn` definiert. Sie hat drei Argumente. Je nach dem Wert des ersten Arguments hat sie das zweite oder das dritte Argument als Ergebnis.

Zusätzlich definieren wir eine einstellige Funktion `f`, die nicht terminiert.

First.fab4
```
constr True 0;
constr False 0;

wenn pred alt1 alt2
  = case pred of
      True  -> alt1;
      False -> alt2;

f x = f (x+1);

main = wenn (2*7 = 14) (2*(17+4)) (f 1)
```

Durch Reduktion, d.h. Anwendung dieser Definitionen, kann der Wert des Ausdrucks `main` berechnet werden:

```
  main
→ wenn (2*7 == 14) (2*(17+4)) (f 1)
→ case (2*7 == 14) of True  -> (2*(17+4));False -> (f 1);
→ case (14 == 14) of True  -> (2*(17+4));False -> (f 1);
→ case (True) of True  -> (2*(17+4));False -> (f 1);
→ (2*(17+4))
→ (2*21)
→ (42)
```

Auswertungsreihenfolge Im letzten Beispiel haben wir uns entschieden, die Reduktion immer an der ersten Stelle des Ausdrucks vorzunehmen. Es ist nicht notwendiger Weise gesagt, daß so vorzugehen ist. Es sind eine Reihe von alternativen Auswertungsreihenfolgen denkbar:

```
  main
→ wenn (2*7 == 14) (2*(17+4)) (f 1)
→ wenn (14 == 14) (2*(17+4)) (f 1)
→ wenn (14 == 14) (2*(21)) (f 1)
→ wenn (14 == 14) (2*(21)) (f (1+1))
→ wenn (14 == 14) (2*(21)) (f 2)
→ wenn (True) (2*(21)) (f 2)
→ wenn (True) (2*(21)) (f (2+1))
→ case (True) of True  -> (2*21);False -> (f (2+1));
→ case (True) of True  -> (42);False -> (f (2+1));
```

```
→ case (True) of True  -> (42);False -> (f 3);
→ (42)
```

Wie man sieht, ergibt die Auswertung das gleiche Ergebnis, allerdings werden ein paar zusätzliche Auswertungsschritte benötigt. Bei ungeschickter Wahl der reduzierten Unterausdrücke kann es sogar passieren, daß die Auswertung nicht terminiert:

```
  main
→ wenn (2*7 == 14) (2*(17+4)) (f 1)
→ wenn (2*7 == 14) (2*(17+4)) (f (1+1))
→ wenn (2*7 == 14) (2*(17+4)) (f 2)
→ wenn (2*7 == 14) (2*(17+4)) (f (2+1))
→ wenn (2*7 == 14) (2*(17+4)) (f 3)
→ wenn (2*7 == 14) (2*(17+4)) (f (3+1))
→ wenn (2*7 == 14) (2*(17+4)) (f 4)
→ wenn (2*7 == 14) (2*(17+4)) (f (4+1))
→ wenn (2*7 == 14) (2*(17+4)) (f 5)
→ wenn (2*7 == 14) (2*(17+4)) (f (5+1))
→ ...
```

In der Literatur unterscheidet man zwei Hauptreihenfolgen der Reduktion:

- normale Reihenfolge: immer der äußerste reduzierbare Ausdruck im Gesamtausdruck wird reduziert.
- applikative Reihenfolge: ein Ausdruck wird erst dann reduziert, wenn er keine reduzierbaren Unterausdrücke enthält.

Sprachen, die die applikative Reihenfolge der Auswertung benutzen, sind die strikten Sprachen, Srachen mit normaler Auswertungsreihenfolge sind nicht-strikte Sprachen oder auch Sprachen mit verzögerter Auswertung.

Aufgabe 1 Schreiben Sie ein dem obigen funktionalen Programm äquivalentes Programm in C++ (oder Java) und lassen dieses laufen. Was stellen Sie fest? Was schließen daraus über die Auswertungsreihenfolge von C?

Graphreduktion Die normale Ausführungsreihenfolge hat leider auch einen Nachteil. Hierzu batrachten einmal folgendes Programm:

Square.fab4

```
square x = x*x;
double x = x+x;
main=square (double 2)
```

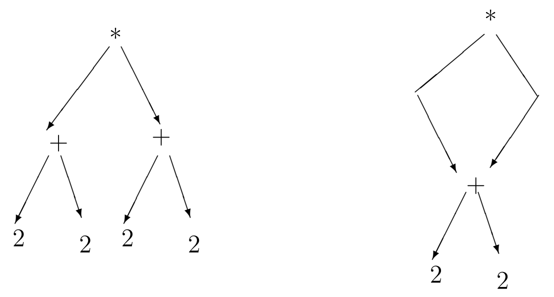



Abbildung 1.5: Term und Graphdarstellung eines Ausdrucks.

In der Applikativen Reihenfolge ergibt sich folgende Reduktion:

```
  main
→ square (double 2)
→ square (2+2)
→ square 4
→ 4*4
→ 16
```

Die normale Auswertungsreihenfolge hat folgende Reduktion:

```
  main
→ square (double 2)
→ (double 2) * (double 2)
→ (2+2) * (double 2)
→ (4) * (double 2)
→ 4*(2+2)
→ 4*4
→ 16
```

Wie man sieht wird ein Schritt mehr benötigt. Die Berechnung der Summe von `2+2` wird zweimal durchgeführt. Der Grund liegt darin, daß in der rechten Seite der Funktionsdefinition von `square` eine Argumentvariable zweimal erscheint, sozusagen das Argument verdoppelt wird. Diesen Nachteil der normalen Auswertungsreihenfolge kann man umgehen, indem nicht auf Ausdrücken sondern auf Graphen, die diese Ausdrücke darstellen, gerechnet wird. In Abbildung 1.5 findet sich der Ausdruck `(2+2) * (2+2)`, wie er aus der Reduktion von `square (2+2)` entsteht einmal als Term und einmal als Graph.

Reduziert man also auf einer Graphdarstellung von Termen, umgeht man die Verdoppelung von Unterausdrücken in der normalen Auswertungsreihenfolge. Man spricht in diesem Fall davon, daß die Programmiersprache *lazy* sei.

Ausführungsmodelle

Der Programmierer schreibt den lesbaren Quelltext seines Programmes. Um ein Programm auf einem Computer laufen zu lassen, muß es erst in einen Programmcode übersetzt werden, den der Computer versteht. Für diesen Schritt gibt es auch unterschiedliche Modelle:

- **kompiliert** (C, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muß stets der Interpreter zur Verfügung stehen, um das Programmm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.
- **abstrakte Maschine über** *byte code* (Java, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, daß durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

Es gibt Programmiersprachen, für die sowohl Interpreter als auch Übersetzer zur Verfügung stehen. In diesem Fall wird der Interpreter gerne zur Programmentwicklung benutzt und der Übersetzer erst, wenn das Programm fertig entwickelt ist.

1.4 Ziel der Vorlesung

Ziel der Vorlesung ist, einen kompletten Compiler zu entwickeln. Die kompilierte Sprache soll dabei ausdrucksstark genug sein, daß der Compiler in ihr selbst implementiert werden kann. Wir werden diese Implementierung in der Vorlesung entwickeln (und dabei feststellen, daß sie wesentlich kürzer ausfällt als die zunächst in C++ entwickelte Version des Compilers).

Naturgemäß kann eine zweistündige Vorlesung nicht alle Aspekte des Compilerbaus erschöpfend behandeln und es sind Abstriche zu machen. In unserem Fall wird die Beschäftigung mit dem Compiler etwas backend-lastig sein. Zur Kompilierung der Quellsprache werden nur die nötigsten Schritte vollzogen, insbesondere natürlich das Parsen. Wir werden leider keinen statischen Typcheck oder andere statische Analysen betrachten können. Auch die lexikalische Analyse wird nur marginal behandelt werden.

1.5 Aufbau der Vorlesung

What goes on.
Lennon/McCartney/Starkey

In dieser Vorlesung werden wir einen kompletten Compiler entwickeln. Hierbei werden wir uns von einer abstrakten Maschine ausgehend hocharbeiten. Das entspricht nicht der Reihenfolge

des Kompiliervorgangs, in dem die abstrakte Maschine ganz hinten in der Kette steht, aber es entspricht der Reihenfolge, in der ein Compiler in der Regel entwickelt wird. Immer weiter aufbauend von einer einfachen Maschinensprache. Erst wenn wir eine Maschine und Ihre Befehle kennen, können wir für diese auch Code erzeugen. Und wenn die Maschine zuerst implementiert ist, können wir diesen Code dann auch gleich laufen lassen. Daher werden wir zuerst die Zielmaschine implementieren.

Um geistig rege zu bleiben, werden wir Programme in 4 Sprachen schreiben:

- C++ zur Entwicklung des Compilers.
- Java für die Implementierung eines Interpreters der G-Maschine.
- Die Sprache *Fab4*, die in diesem Skript definiert und für die ein Compiler entwickelt wird.
- G-Code für die Programme einer abstrakten Graphenreduktionsmaschine (G-Maschine).

Mengenmäßig wird der C++-Code überwiegen.

Wir sind mit vier Sprachen recht unterschiedlicher Eigenschaften konfrontiert:

- **C++**: imperative prozedurale Sprache (mit objektorientierten Erweiterungen). Explizite Speicherverwaltung. Kompiliert.
- **Java**: objektorientierte Sprache. Implizite Speicherverwaltung (garbage collection). Realisierung über virtuelle Maschine.
- **Fab4**: rein funktionale verzögert ausgewertete Sprache. Implizite Speicherverwaltung (garbage collection). Realisierung über virtuelle Maschine.
- **G-Code**: Maschinensprache einer Kellermaschine zur Manipulation von Graphenstrukturen.

Wir werden in Kapitel 2 eine abstrakte Maschine definieren. Für die Maschine wird ein Interpreter entwickelt. Dieser Interpreter wird in Java geschrieben. In Kapitel 3 werden die aus der theoretischen Informatik bekannten Konzepte formaler Sprachen wiederholt. Kapitel 4 beschäftigt sich mit der Implementierung eines Parsers. Hierzu wird eine Parserbibliothek in C++ entwickelt. In Kapitel 5 wird die Sprache Fab4 spezifiziert und in Kapitel 6 ein Compiler von Fab4 nach G-Code in C++ entwickelt. Schließlich wird in Kapitel 7 der Fab4-Compiler reimplementiert, diesmal in Fab4, so daß anschließend die C++ Implementierung des Fab4-Compilers weggeworfen werden kann.

Abbildung 1.6 gibt einen Schematischen Überblick über den in diesem Skript entwickelten Compiler.

Quelltext wird in diesem Skript jeweils in einem Kasten durchnumerierter Zeilen dargestellt. Der Kasten enthält eine Überschrift als Markierung, in welcher Quelltextdatei der entsprechende Code steht. Wenn eine Quelltextdatei auf mehrere Kästen verteilt ist, läßt sich dieses leicht an der fortlaufenden Zeilennumerierung erkennen.

Da dieses Skript den kompletten Quelltext eines Compilers und eines Interpreters enthält, lassen sich Vorwärtsverweise nicht vermeiden. Es werden mitunter Codezeilen in einem früheren Kapitel im Text auftauchen, auf deren Sinn und Erklärung der Leser auf ein späteres Kapitel vertröstet wird.

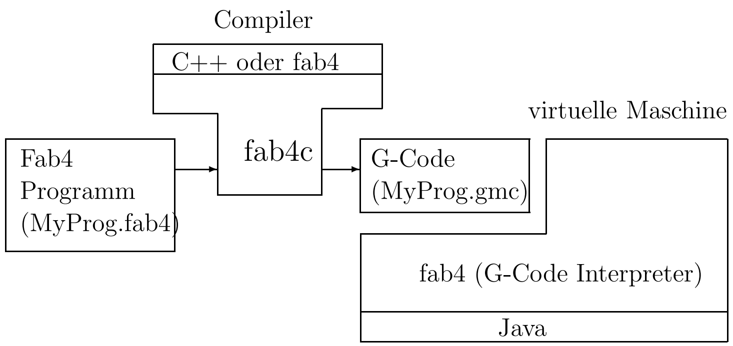


Abbildung 1.6: Überblick über fab4.

1.5.1 Terminologie

In Laufe dieses Skripts wird versucht, weitgehendst die Terminologie aus der objektorientierten Programmierung zu benutzen. Um grobe Mißverständnisse zu vermeiden, folgt hier eine kurze Liste der wichtigsten Begriffe:

Klasse: eine Klasse beschreibt eine Bauvorschrift für Objekte. Die Klasse hat eine Menge von Eigenschaften in Form von Feldern und Methoden.

Feld: ein Feld bezeichnet eine Referenz in einer Klasse. Achtung: mit Feld ist in diesem Kontext kein *array* gemeint.

Methoden: eine Methode bezeichnet ein Unterprogramm innerhalb einer Klasse. Man unterscheidet statische Methoden und Objektmethoden.

Polymorphie: Gleichnamige Methoden können in unterschiedlichen Klassen auftauchen. Sind dieses Objektmethoden (in C++ als virtuell markiert), wird dynamisch bei einem Aufruf das Objekt zur Ausführung seiner speziellen Methodendefinition aufgefordert. Achtung: in der funktionalen Programmierung hat der Bergriff *Polymorphie* eine leicht andere Bedeutung.

Überladung: verschiedene Methodendefinitionen für eine Methode gleichen Namens nennt man Überladung. Überladungen werden statisch zur Übersetzungszeit aufgelöst.

Überschreiben: wird in einer Klasse eine Methode, die in einer Oberklasse bereits existiert, mit gleicher Signatur neu definiert, spricht man vom *Überschreiben* der Methode. Durch das sogenannte *late-binding* werden in Java überschriebene Methoden (in C++ als virtuell markierte Methoden) dynamisch während der Laufzeit aufgelöst.

Generizität: bei einer Methodendefinition die für verschiedene Typen gleichartig funktioniert, spricht man von Generizität. In C++ wird Generizität über Schablonen *templates* ausgedrückt. In Java seit Java 1.5 mit generischen Typen.

Reihung: als Deutscher Ausdruck für *array* wird die Bezeichnung *Reihung* verwendet. Damit ensteht keine Mehrdeutungkeit mit dem in der Objektorientierung bereits anderweitig belegten Begriff *Feld*.

Kapitel 2

Die G-Maschine

Nothing is real.
Lennon/McCartney

2.1 Einführung

Die Zielmaschine unseres Compilers ist die sogenannte G-Maschine. Das *G* steht dabei für *graph*. Die G-Maschine wurde als abstrakte Maschine zur Manipulation von Graphen im Rahmen der Übersetzung funktionaler Sprachen in den 80er Jahren entwickelt. Die ursprünglichen Arbeiten basieren auf einen Compiler für eine verzögert ausgewerteten Variante der Sprache ML [Aug84].[1] Eine gute Präsentation der G-Maschine findet sich in [PJ87].[2] Eine Optimierung der G-Maschine ist die sogenannte STG-Maschine [PJ92][PJS89], die die Grundlage des Haskellcompilers GHC [PJHH+92] darstellt. Eine weitere Variante ist die ABC-Maschine [KvEN+90], die die Grundlage des Compilers der Sprache *Clean* bildet [Pv97].

Im Laufe dieses Skripts folgen wir in weiten Teilen der Präsentation der G-Maschine aus [PJL91]. Dieses Buch steht glücklicher Weise als Onlineversion zur Verfügung (`research.microsoft.com/Users/simonpj/Papers/pj-lester-book/`) .

Die G-Maschine besitzt einen Keller, auf dem Adressen des Heaps verwaltet werden. In dieser Hinsicht unterscheidet sich die G-Maschine kaum von den meisten üblichen abstrakten Maschinenmodellen. Hingegen besonders an der G-Maschine ist, auf welche Weise Funktionsanwendungen aufgerufen werden.

Wir implementieren in diesem Kapitel einen Interpreter der G-Maschine in Java. Die Wahl fiel dabei auf Java, weil Javas Reflektionsmechanismus es auf einfache Art ermöglicht, aus der G-Maschine beliebigen Javacode aufzurufen. Damit lassen sich insbesondere Ein- und Ausgabeoperationen in Java implementieren und innerhalb des G-Codes aufrufen. Wir werden dabei mit den typischen Problemen der *foreign function calls* konfrontiert werden. Leider ist es nicht gelungen, den notwendigen Code für die Aufrufe von Javacode aus der G-Maschine komplett in einen eigenen Kapitel zu bündeln. So wird leider unser Quelltext immer wieder durch Zeilen verschmutzt, die notwendig für die *foreign function calls* sind.

[1] Der Autor Lennart Augustsson hat sich nebenbei auch als mehrfacher Sieger des obfuscated C Wettbewerbs hervor getan.

[2] Simon Peyton-Jones ist nach seiner Zeit an der Glasgower Universität zu Microsoft Research gegangen. Was ihn aber nicht davon abgehalten hat, auf seiner Webseite bei Microsoft Hinweise zu geben, wie man Windows NT möglichst gut fast wie Unix benutzen kann.

2.2 Java in drei Minuten

Da zu den unbedingten Voraussetzungen dieser Vorlesung nicht die Kenntnis vom Umgang mit Java gehört, wollen wir wenigstens eine schnelle Starthilfe geben.

2.2.1 Übersetzung und Ausführung von Javaprogrammen

Da Java sich einer abstrakten Maschine bedient, sind, um zur Ausführung zu gelangen, sowohl ein Übersetzer als auch ein Interpreter notwendig. Der zu übersetzende Quelltext steht in Dateien mit der Endung `.java`, der erzeugte *byte code* in Dateien mit der Endung `.class`. In einer Javaquelltextdatei steht immer nur genau eine Klasse.[3] Klassennamen beginnen mit einem Großbuchstaben.

Der Javaübersetzer kann von der Kommandozeile mit dem Befehl `javac` aufgerufen werden. Um eine Programmdatei `Test.java` zu übersetzen, kann folgendes Kommando eingegeben werden:

```
javac Test.java
```

Im Falle einer fehlerfreien Übersetzung wird eine Datei `Test.class` im Dateisystem erzeugt. Dieses erzeugte Programm wird allgemein durch folgendes Kommando im Javainterpreter ausgeführt:

```
java Test
```

2.2.2 Hello Java world

I don't know why
You say goodbye
I say hello.
Lennon/McCartney

Wir geben das übliche helloword-Programm an, das eine einfache Ausgabe auf den Bildschirm macht:

FirstProgram.java

```
class FirstProgram{
  public static void main(String [] args){
    System.out.println("hello world");
  }
}
```

Aufgabe 2 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. Führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

[3] Es gibt Ausnahmen, aber davon lassen wir vorerst die Finger.

Wir können in der Folge diesen Programmrumpf benutzen, um beliebige Objekte auf den Bildschirm auszugeben. Hierzu werden wir das `"hello world"` durch andere Ausdrücke ersetzen.

Wollen Sie z.B. eine Zahl auf dem Bildschirm ausgeben, so ersetzen Sie den in Anführungszeichen eingeschlossenen Ausdruck durch diese Zahl:

Answer.java
```
class Answer {
  public static void main(String [] args){
    System.out.println(42);
  }
}
```

2.2.3 Pakete, Klassenpfad

Eine kleine Schwierigkeit in Java, stellt oft die Übersetzung und Ausführung von Javaprogrammen, die eine Paketangabe haben, dar. Pakete entsprechen logisch dabei in etwa den Namensräumen in C++. Die folgende Klasse liegt in einem Paket `paket`, das ein Unterpaket des Pakets `panitz` ist, das ein Unterpaket des Pakets `name` ist.

FirstPackage.java
```
package name.panitz.paket;

class FirstPackage{

  public static void main(String [] args){
    System.out.println("hello package");
  }
}
```

Bei der Übersetzung dieses Programms, sollte man den Javacompiler dazu bringen die Datei `FirstPackage.class` in einer Ordnerhierarchie entsprechend der Pakethierarchie zu speichern. Dieses kann man mit der Option `-d zielordnername` des Javacompilers. Ausgehend vom Ordner `zielordnername` wird eine Ordnerhierarchie für die Pakete generiert. Angenommen ein Ordner `klassen` existiert im Dateisystem, läßt sich obiges Programm also mit dem Befehl:

```
javac -d klassen FirstPackage.java
```

übersetzen.

Zum Ausführen einer Klasse mit einer Hauptmethode, die in einem Paket liegt, ist dem Javainterpreter die Klasse vollqualifiziert mit Ihrer Paketangabe zu nennen. Zusätzlich ist in einer Pfadangabe anzugeben, von welchen Ordnern aus sich Klassen und Paketstrukturen befinden. Dieses kann durch Setzen der Umgebungsvariablen `CLASSPATH` oder durch die Kommandozeilenoption `-classpath` geschehen. Unser Beispielprgramm läßt sich also ausführen mit dem Kommando:

```
java -classpath klassen name.panitz.paket.FirstPackage
```


2.3 Hilfsklassen

With a little help from my friends.
Lennon/McCartney

Bevor wir anfangen die G-Maschine zu spezifizieren und zu implementieren, schreiben wir zwei kleine Hilfklassen. Wir benutzen dabei die aktuelle Javaversion mit generischen Typen [BCK+01]. C++-Programmierern sind generische Typen halbwegs durch die Schablonen (templates) in C++ bekannt.

2.3.1 Allgemeine Tupelklasse

Unsere erste Hilfsklasse ist eine einfache Tupelklasse, die es erlaubt zwei Objekte zu einem Objektpaar zusammenzufassen. Diese Tupelklasse ist generisch über die Typen der beiden Objekte:

Pair.java
```
package name.panitz.util;
public class Pair<a,b>{
```

Wir haben zwei Typvariablen `a` und `b` eingeführt. In C++ hätten wir hierfür eine Schablone geschrieben:

```
template <typename a,typename b>
class Pair{
```

Die beiden Typvariablen sind als allquantifiziert zu verstehen; sie repräsentieren zwei beliebige aber wenn einmal gewählt dann feste Typen. Wir können den Klassenanfang lesen als: für alle Typen `a` und `b` definieren wir eine Klasse `Pair<a,b>`.

Die zwei Objekte die in einem Paar zusammengefasst werden sollen, brauchen jeweils ein Feld in der Klasse `Pair`, um sie dort abzuspeichern. Wir brauchen also ein Feld vom Typ `a` und eines vom Type `b`. Wir benutzen für diese beiden Felder die Bezeichner `e1` und `e2`:

Pair.java
```
  public a e1;
  public b e2;
```

Wir sehen einen Konstruktor für die Klasse vor, in dem die beiden Felder mit übergebenen Objekten initialisiert werden:

Pair.java
```
  public Pair(a e1,b e2){this.e1=e1;this.e2=e2;}
```

Schließlich überschreiben wir noch die Methode `toString` um eine aussagekräftige Repräsentation des Paarobjekts zu geben:

Pair.java
```
  public String toString(){return "("+e1+","+e2+")";}
}
```

2.3.2 einfach verkettete Listen

You know my name
look up the number
Lennon/McCartney

Die zweite kleine Hilfsklasse soll einfach verkette Listen realisieren. Ein Listenobjekt ist im Prinzip auch ein Paar. Es enthält das erste Listenelement und als zweites Objekt die Restliste der übrigen Elemente. In der Literatur haben sich für diese beiden Bestandteile die Begriffe *head* und *tail* durchgesetzt.[4] Eine besondere Form von Listen, sind leere Listen. Für diese existieren Kopf und Schwanz der Liste nicht.

Die implementierte Listenklasse soll generisch über den Typ der Listenelemente sein.

Li.java
```
package name.panitz.util;
public class Li<a>{
```

Wir benötigen Felder zum Speichern von Kopf und Schwanz.

Li.java
```
  private a head;
  private Li<a> tail;
```

Ein zusätzliches bool'sches Feld soll als Flag dienen, um anzugeben, ob das Listenobjekt eine leere Liste darstellt. Standardmäßig sei dieses auf `false` gesetzt:

Li.java
```
  private boolean isEmpty=false;
```

Zwei Konstruktoren seien vorgesehen:

- einer zum Erzeugen einer neuen Liste, durch Anfügen eines zusätzlichen Elements an eine bestehende Liste.
- einer zum Erzeugen von leeren Listen.

Li.java
```
  public Li(a hd,Li<a> tl){head=hd;tail=tl;}
  public Li(){isEmpty=true;}
```

Die Felder der Klassen waren privat gesetzt, so daß auf sie nicht von außerhalb der Klasse zugegriffen werden kann. Daher sehen wir drei öffentliche Zugriffsmethoden vor:

Li.java
```
  boolean isEmpty(){return isEmpty;}
  public a head(){return head;}
  public Li<a> tail(){return tail;};
```

[4] Lispprogrammierer kennen sicherlich `car` und `cdr` von Listen.

Schließlich sei noch eine Methode `toString` definiert, die für eine Liste eine gut lesbare Stringdarstellung erstellt:

Li.java
```
  public String toString(){
    StringBuffer result= new StringBuffer("[");
    Li<a> run=this;
    boolean first=true;
    while (!run.isEmpty){
      if (first) first=false; else result.append(",");
      result.append(run.head);run=run.tail;
    }
    result.append("]");
    return result.toString();
  }
}
```

Aufgabe 3 Sollten Sie mit Java noch nicht vertraut sein, so kompilieren Sie die beiden obigen Klassen. Schreiben Sie eine minimale Testklasse, die ein Paar und eine Liste erzeugt und auf dem Bildschirm ausgibt. (Hinweis: in der nachfolgenden Aufgabe finden Sie ein Beispiel für eine `main`-Methode und den Ausgabebefehl `println`.)

Lösung

In dieser Aufgabe soll hauptsächlich der Umgang mit Javacompiler und Interpreter geübt werden. Eine kleine Testklasse könnte dann wie folgt aussehen:

TestLi.java
```
package name.panitz.util;
public class TestLi {
  public static void main(String [] args){
    System.out.println(new Pair<String,Integer>("hallo",42));
    System.out.println(new Li<String>());
    System.out.println(
          new Li<String>("hallo"
         ,new Li<String>("welt"
         ,new Li<String>())));
  }
}
```

Aufgabe 4 Mit den zwei oben vorgestellten Klassen, lassen sich Abbildungen realisieren. Eine endliche Abbildung läßt sich als Liste von Schlüssel/Wert-Paaren darstellen. Schreiben Sie eine in einer Klasse eine statische Methode mit folgender Signatur:

```
public static <keyType,valueType>
valueType lookUp(keyType key,Li<Pair<keyType,valueType>> map)
```

`lookUp` soll für ein Schlüsselobjekt einen entsprechend damit gepaarten Wert in einer Liste von Paaren nachschlagen.

Testen Sie Ihre Methode mit folgender Testklasse:

TestStaticMap.java
```
package name.panitz.util;
public class TestStaticMap {
 public static void main(String [] args){

  Li<Pair<String,Integer>> notenliste =
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("john",4),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("paul",3),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("george",1),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("ringo",4),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("stu",2),
    new Li<Pair<String,Integer>>()))))));

  System.out.println(notenliste);
  System.out.println(StaticMap.lookUp("george",notenliste));
  System.out.println(StaticMap.lookUp("stu",notenliste));
  System.out.println(StaticMap.lookUp("pete",notenliste));
 }
}
```

Lösung

Der Rumpf der Methode hat tatsächlich nur drei Zeilen.

- Teste, ob die Liste leer ist. Für leere Liste gebe `null` aus. (Alternativ ließe sich hier auch eine Ausnahme werfen.)
- Bei nichtleeren Liste prüfe das erste Listenelement, im Erfolgsfall gib dessen Wert aus.
- Ansonsten rekursiver Aufruf auf die Restliste.

Wer bisher keine Erfahrung in Java hatte, wird wahrscheinlich statt des Aufrufs der Methode `equals` den Operator `==` verwendet haben.

StaticMap.java
```
package name.panitz.util;
public class StaticMap {
  public static <keyType,valueType>
  valueType lookUp(keyType key,Li<Pair<keyType,valueType>> map){
    if (map.isEmpty()) return null;
    if (map.head().e1.equals(key)) return map.head().e2;
```



```
    return lookUp(key,map.tail());
  }
}
```

Aufgabe 5 Schreiben Sie jetzt eine generische Klasse `MyMap<keyType,valueType>`, die den Typ `Li<Pair<keyType,valueType>>` erweitert. Implementieren Sie in der Klasse `MyMap` die Methode
`public valueType lookUp(keyType key)`,
die für ein Schlüsselobjekt das mit diesem Schlüssel gepaarte Wertobjekt zurückgibt.

Testen Sie Ihre Implementierung mit folgender Klasse:

TestYourMap.java
```
package name.panitz.util;
public class TestYourMap {
  public static void main(String [] args){
    MyMap<String,Integer> notenliste
     =  new MyMap<String,Integer>("john",4,
        new MyMap<String,Integer>("paul",1,
        new MyMap<String,Integer>("george",1,
        new MyMap<String,Integer>("ringo",5,
        new MyMap<String,Integer>("stu",2,
        new MyMap<String,Integer>()))))));

    System.out.println(notenliste.lookUp("john"));
    System.out.println(notenliste.lookUp("george"));
    System.out.println(notenliste.lookUp("pete"));
  }
}
```

2.4 Der Heap

Als *heap* wird der Speicherbereich bezeichnet, in dem die eigentlichen Anwendungsdaten eines Programmes liegen und manipuliert werden.[5] Der Heap besteht aus einzelnen durchnumerierten Speicherzellen. In diesen Speicherzellen werden prinzipiell zwei verschiedene Arten von Daten gespeichert:

- primitive Werte, Zahlen oder Zeichen.
- Verweise auf andere Heapzellen.

Und damit wird klar in welcher Weise im Heap Graphen dargestellt werden: die Verweise auf andere Heapzellen stellen gerichtete Kanten des Graphen dar. In C läßt sich diese Unterscheidung sehr direkt auf der Sprachebene nachvollziehen. Zeiger sind expliziter Sprachbestandteil. Ein Zeiger wird meistens durch eine gerichtete Kante eines Graphen visualisiert.

[5] Leider hat sich in der deutschen Fachsprache keiner der Begriffe *Haufen* oder *Halde* etablieren können.

2.4.1 HeapKnoten

Wir werden 7 verschiedene Arten von Heapknoten in der G-Maschine vorsehen (einer begegnet uns allerdings erst in Kapitel 2.9). Für jede Art von Heapknoten werden wir eine Javaklasse schreiben. Als gemeinsamen Obertypen für Heapknoten definieren wir zunächst die allgemeine Schnittstelle `Node`:

Node.java
```java
package name.panitz.gm;

public interface Node{
```

In dieser Schnittstelle sehen wir ein Methode vor, die wir benutzen werden, um das Besuchsmuster für Heapknoten zu realisieren.

Node.java
```java
  public <a> a visit(NodeVisitor<a> v) throws Exception;
}
```

Die Idee des Besuchersmusters werden wir uns erst später genauer anschauen. Vorerst soll es genügen zu wissen, daß für alle Klassen, die die Schnittstelle `Node` implementieren, die Methode `visit` auf diesselbe Weise implementiert wird, nämlich:

```java
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
```

Die Klasse `NodeVisitor` lernen wir auf Seite 2.4.1 kennen. In ihr werden alle Knotenklassen referenziert. Damit entsteht eine zyklische Abhängigkeit der Klassen beim kompilieren. Eine Knotenklasse leitet von der Klasse `Node` ab. In der Klasse `Node` wird die Schnittstelle `NodeVisitor` benutzt. In dieser Schnittstelle werden alle Knotenklassen benutzt. Solange eine Knotenklasse, aus den folgenden Aufgaben noch nicht implementiert ist, ist die entsprechende Zeile, in der diese Klasse in der Schnittstelle `NodeVisitor` benutzt wird, auszukommentieren. Ansonsten kommt es zu Übersetzungsfehlern.

primitive Typen

Die ersten Art von Heapknoten sollen Daten primitiver Typen darstellen. In der G-Maschine (und später auch in fab4) werden wir nur zwei primitive Typen vorsehen:

- `int`
- `char`

Für beide dieser primitiven Typen sehen wir eine Klasse vor, die den entsprechenden Heapknoten ausdrückt. Beginnen wir mit dem Typ `int`. Wir definieren eine Knotenklassen `NNum`, die die Schnittstelle `Node` implementiert:

NNum.java
```java
package name.panitz.gm;

public class NNum implements Node{
```

Die Knoten `NNum` sollen ganze Zahlen speichern, also sehen wir ein entsprechendes Feld in der Klasse vor und einen Konstruktor, um dieses zu initialisieren:

NNum.java
```java
  int n;
  public NNum(int n){this.n=n;}
```

Zusätzlich definieren wir noch die Gleichheit auf dieser Klasse

NNum.java
```java
  public boolean equals(Object other){
    return other instanceof NNum
          && n==((NNum)other).n;
  }
```

Wie versprochen, wird die Methode `visit` auf die oben gezeigte Art implementiert.

NNum.java
```java
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
```

Schließlich implementieren wir noch die Methode `toString`, so daß sie die gespeicherte Zahl als String zurückgibt.

NNum.java
```java
  public String toString(){return ""+n;}
}
```

Aufgabe 6 Implementieren Sie eine zu `NNum` analoge Klasse `NChar`, die primitive Zeichen (char) im Heap darstellen kann.

Indirektionen

Im vorherigen Abschnitt haben wir Heapknoten definiert, die Daten primitiver Typen darstellen können. Betrachten wir die Daten im Heap als einen Graph, dann stellen diese Knoten entsprechend Knoten in diesem Graph dar. Die übrigen Knoten des Heaps werden jeweils Kanten des Graphen enthalten. Als einfachste Form von Kanten definieren wir Indirektionsknoten. Diese kann man in gewisser Weise mit Zeigern in C identifizieren. Es sind Heapknoten, die eine weitere Adresse im Heap enthalten.

Die Implementierung sieht wieder der Implementierung von `NNum` und `NChar` sehr ähnlich. In einem Feld wird die Heapadresse gespeichert, auf die der Indirektionsknoten verweist. Im Konstruktor wird dieses Feld initialisiert, die Methoden `visit`, `toString` und `equals` sind entsprechend zu implementieren.

Aufgabe 7 Implementieren Sie die Klasse `NInd`, die Indirektionsknoten im Heap darstellt.

Konstruktorknoten

Bisher können wir noch nicht wirklich Graphen mit unseren drei Heapknotenklassen darstellen. Hierzu bedarf es noch der Möglichkeit, von einem Heapknoten mehrere Verweise auf andere Heapknoten ausgehen lassen zu können. Hierzu definieren wir Konstruktorknoten. Ein Konstruktorknoten fasst mehrere Verweise auf andere Heapknoten zusammen. Man kann einen Konstruktorknoten mit einem Record von Zeigern in C vergleichen. Ebenso wie ein Recordtyp einen Namen hat, hat ein Konstruktorknoten auch einen Konstruktornamen. Dieser soll im Heapknoten als ganze Zahl gespeichert sein.

Ein Konstruktorknoten hat also zwei Bestandteile:

- eine ganze Zahl, die den Konstruktornamen codiert.
- einer Reihung (array) von Zahlen, die die Verweise auf weitere Heapadressen darstellt.

Die Verweise, die von einem Konstruktorknoten ausgehen, bezeichnen wir als die Argumente des Konstruktors.

Damit läßt sich auch die entsprechende Heapknotenklasse direkt umsetzen:

NConstr.java
```
package name.panitz.gm;

public class NConstr implements Node{
  int name;
  int[]args;
```

Die Felder `name` und `args` enthalten die relevanten Informationen für einen Konstruktorknoten. Wir sehen Konstruktoren vor diese zu initialsieren. Werden keine Argumente bei der Konstruktion eines Konstruktorknotens mit übergeben, so wird für die Argumente eine Reihung der Länge 0 erzeugt.

NConstr.java
```
  public NConstr(int n,int[]as){name=n;args=as;}
  public NConstr(int n){name=n;args=new int[0];}
```

Schließlich werden wieder die Methoden `visit` und `toString` implementiert:

NConstr.java
```
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
  public String toString(){
    StringBuffer result=new StringBuffer("(Constr "+name+" [");
    for (int i:args) result.append(" "+i);
    result.append("])");
    return result.toString();
  }
}
```

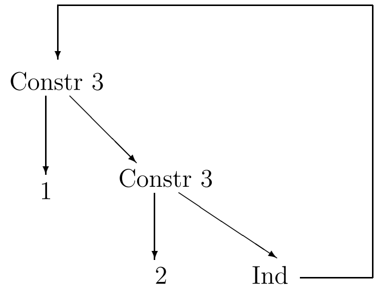



Abbildung 2.1: Graph einer zyklischen Liste.

Jetzt haben wir tatsächlich genügend Arten von Heapknoten, um Graphen darzustellen:

Beispiel:
In Abbildung 2.1 ist ein Graph zu sehen, in dem es zwei Knoten mit primitiven Werten gibt, zwei Konstruktorknoten und einen Indirektionskoten. Der Graph stellt eine zyklische Liste dar, in der sich die Werte `1` und `2` wiederholen.

Folgendes Testprogramm erzeugt die Heapstruktur für den Graph:

EinsZwei.java
```
package name.panitz.gm.example;
import name.panitz.gm.*;

public class EinsZwei {
  public static void main(String [] _){
    Node[] einszwei = new Node[5];
    int[] args1 = {1,2};
    einszwei[0] = new NConstr(3,args1);
    einszwei[1] = new NNum(1);
    int[] args2 = {3,4};
    einszwei[2] = new NConstr(3,args2);
    einszwei[3] = new NNum(2);
    einszwei[4] = new NInd(0);

    for (int i=0;i<einszwei.length;i++)
      System.out.println(i+": "+einszwei[i]);
  }
}
```

Dieses Testprogramm hat folgende Ausgabe:

```
sep@linux:~/fh/compiler/tutor> java -classpath classes/ name.panitz.gm.example.EinsZwei
0: (Constr 3 [ 1 2])
1: 1
2: (Constr 3 [ 3 4])
3: 2
4: NInd -> 0
sep@linux:~/fh/compiler/tutor>
```

Vordefinierte Konstruktornamen 5 Konstruktoren werden eine ausgezeichnete Rolle in unserer G-Maschine spielen:

- zwei nullstellige Konstruktoren um die bool'schen Werte `True` und `False` darzustellen.
- zwei Konstruktoren zu Listenrepräsentation entsprechend unserer Javaklasse `Li`. Einer davon um leere Listen darzustellen und einer davon um eine Liste mit Kopf- und Schwanzbestandteil darzustellen.
- ein Konstruktor um Paare entsprechend der Javaklasse `Pair` im Heap zu speichern.

Für diese 5 Konstruktoren vergeben wir hier ein für alle mal feste Nummern, die wir als Konstanten in einer Javaklasse definieren:

ConstructorConstants.java

```java
package name.panitz.gm;

public class ConstructorConstants{
  public static final int TRUE =0;
  public static final int FALSE=1;
  public static final int NIL  =2;
  public static final int CONS =3;
  public static final int PAIR =4;
```

Für diese 5 Konstruktoren sehen wir auch Namen im Klartext vor, die ebenfalls als statische Konstanten in dieser Klasse festgehalten sind:

ConstructorConstants.java

```java
  public static final String TRUE_S ="True";
  public static final String FALSE_S="False";
  public static final String NIL_S  ="Nil";
  public static final String CONS_S ="Cons";
  public static final String PAIR_S ="Pair";
}
```

Wenn wir in Zukunft den Heap als Graph skizzieren, werden wir für Konstruktoren nicht die Nummer des Konstruktors benutzen, sondern einen Klartextnamen.

Strings Wir werden in der G-Maschine und in Fab4 keinen besonderen Typ zur Darstellung von Zeichenketten vorsehen. Ein String sei einfach modelliert über eine verkettete Liste von Zeichen.

> **Beispiel:**
> In Abbildung 2.2 ist der Graph für den String `"hallo"` zu bewundern.

Aufgabe 8 Schreiben Sie ähnlich der Klasse `EinsZwei` eine kleine Testmethode, die den Graph aus Abbildung 2.2 erzeugt.

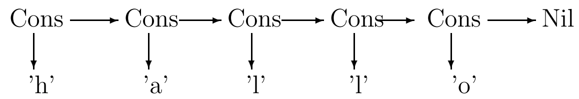


Abbildung 2.2: Graph eines Strings.

Funktionen

Bisher können wir beliebige Datenstrukturen im Heap speichern. Jetzt werden wir Heapknoten definieren, die noch nicht ausgewertete Ausdrücke darstellen. Wir führen das Konzept von Funktionen ein. Funktionen werden auf bestimmten Code verweisen. Funktionen haben eine endliche Anzahl von Parametern (die auch 0 sein darf).

Verweise auf globale Funktionen Zunächst sehen wir einen Knotenart vor, die eine Referenz auf eine global definierte Funktion darstellt. In diesem Knoten sind zwei Informationen zu speichern:

- die Argumentanzahl der Funktion.
- eine Zahl, die auf den Funktionscode verweisen soll.

Damit ergibt sich die Knotenklasse zur Darstellung von Verweisen auf global definierte Funktion relativ naheliegend:

NGlobal.java

```java
package name.panitz.gm;

public class NGlobal implements Node{
  int argc;
  int i;
  NGlobal(int a,int c){this.argc=a;this.i=c;}

  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
  public String toString(){return "(NGlobal "+argc+" "+i+")";}
}
```

Currying Wir können nun Verweise auf globale Funktionen im Heap abspeichern. Jetzt wollen wir Anwendungsknoten im Heap anlegen können, die ausdrücken, daß eine bestimmte Funktion auf Argumente anzuwenden ist. Hierbei werden wir uns eines Tricks aus der Mathematik bedienen, der unter der Bezeichnung *Currying* bekannt ist. Diesen Prinzip ist nicht nach einem indischen Reisgericht sondern nach dem Mathematiker Haskell B. Curry (1900-1982) benannt[6].

Beim Currying wird davon ausgegangen, daß es nur noch einstellige Funktionen gibt. Eine n-stellige Funktion mit $n > 1$ wird dann modelliert als eine einstellige Funktion, deren Ergebnis eine $n-1$-stellige Funktion ist.

[6]Obwohl die Idee hierfür wohl ursprünglich von Moses Schönfinkel stammt, aber *Schönfinkeling* wäre wohl eine wenig attraktive Bezeichnung gewesen.

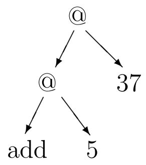


Abbildung 2.3: Darstellung der Anwendung einer zweistelligen Funktion im Heap.

Beispiel:
Die zweistellige Funktion $f(x, y) = x + y$ wird als einstellige Funktion betrachtet. Die Anwendung von f auf nur ein Argument hat eine neue Funktion als Ergebnis. So ist das Ergebis von $f(5)$ eine Funktion mit folgender Definition: $f2(y) = 5 + y$. Diese kann dann gegebenenfalls auf ein zweites Argument angewendet werden. Es läßt sich also statt $f(5, 37)$ schreiben: $(f(5))(37)$. Dieses läßt sich lesen als: wende die Funktion, die durch Anwendung der Funktion f auf die Zahl 5 entsteht, schließlich auf die Zahl 37 an.

Funktionsanwendungen Nach den Überlegungen aus dem letzten Abschnitt reicht es aus nur einstellige Funktionsanwendungen als Heapknoten vorzusehen. Eine einstellige Funktionsanwendung braucht zwei Verweise auf andere Heapknoten:

- der Heapknoten, der in irgendeiner Weise eine Funktion darstellt.
- der Heapknoten, auf den diese Funktion angwendet werden soll.

es ergibt sich die folgende naheliegende Klasse für Funktionsanwendungsknoten im Heap:

NAp.java
```
package name.panitz.gm;

public class NAp implements Node{
  int n1;
  int n2;
  NAp(int n1,int n2){this.n1=n1;this.n2=n2;}
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
  public String toString(){return "NAp("+n1+" "+n2+")";}
}
```

Wenn wir den Graph eines Heaps zeichnen, so benutzen wir für die Markierung von Verweisen auf Funktionen einen Klartextnamen und für Applikationsknoten das Klammeraffensymbol.

Beispiel:
In Abbildung 2.3 ist die Anwendung einer zweistelligen Funktion als Graph dargestellt.

Soweit die verschiedene Knotentypen im Heap. Einen siebten Knotentypen werden wir in Kapitel 2.9 kennenlernen.

Eine Besucherschnittstelle für Heapknoten

Wir haben bisher immer brav die Methode `visit` in den Klassen für Heapknoten implementiert und dabei auch die generische Schnittstelle `NodeVisitor` benutzt. Es ist an der Zeit, zu erklären, warum wir dieses gemacht haben. Wir wollen das Entwurfsmuster eines Besuchers realisieren [GHJV95].

Die Idee des Besuchsusters ist im Prinzip für eine objektorientiert definierte Datenstruktur gerade nicht objektorientiert zu programmieren. Hierzu betrachten wir zunächst einmal, worin die Unterschiede zwischen Objektorientierung und funktionaler Programmierung liegen:

- Bei der Objektorientierung steht das Objekt im Vordergrund. Die Algorithmen sind verteilt auf mehrere Klassen. Dieses wird in objektorientierten Sprachen als Polymorphie bezeichnet. Eine Methode mit gleichen Namen findet sich in unterschiedlichen Klassen. Die Vereinigung aller dieser Methoden gleichen Namens kann als der Gesamtalgorithmus bezeichnet werden. Jede Klasse beschreibt den Spezialfall, wie für seine Objektart der Algorithmus funktioniert.
- Bei einer funktionalen Programmierung steht der Algorithmus im Vordergrund. Es gibt dabei nur eine Funktionsdefinition, die eine Fallunterscheidung auf ihr Argument macht.

Die verschiedenen Knotenarten im Heap haben wir objektorientiert modelliert. In jeder dieser Klassen haben wir die Methode `toString` überladen. Damit haben wir den Algorithmus, der für Heapknoten eine textuelle Darstellung erzeugt auf die unterschiedlichen Klassen verteilt. Wir können natürlich den Algorithmus `toString` auch nicht objektorientiert als statische Funktion realisieren:

NodeToString.java
```
package name.panitz.gm;
public class NodeToString {
  static public String toString(Node n) throws Exception{
    if (n instanceof NNum) return "(NNum "+((NNum)n).n+")";
    if (n instanceof NChar)return "(NChar \'"+((NChar)n).c+"\')";
    if (n instanceof NConstr ){
      String args = "[";
      for (int i:((NConstr)n).args) args=args+i+" ";
      return "(NConstr "+((NConstr)n).name+args+"])";
    }
    if (n instanceof NInd)return "(NInd "+((NInd)n).adr+")";
    if (n instanceof NAp)
      return "(NAp "+((NAp)n).n1+","+((NAp)n).n2+")";
    if (n instanceof NGlobal)
      return "(NGlobal "+((NGlobal)n).argc+","+((NGlobal)n).i+")";
    throw new Exception("unmatched node type: "+n);
  }
}
```

In dieser Umsetzung wird eine Fallunterscheidung auf das Argument gemacht. Dieses widerspricht fundamental einem objektorientierten Entwurf. In C++ hat man daher ganz bewußt auf ein Konstrukt entsprechend Javas `instanceof` verzichtet. Es gibt in C++ kein Konstrukt ein Objekt darauf zu prüfen, ob es von einer bestimmten Klasse ist.

Trotzdem und gerade im Compilerbau ist es oft sinnvoll einen Algorithmus nicht verteilt auf die einzelnen Klassen, sondern gebündelt in genau einer Klasse zu entwerfen. Dann ist aber eine Umsetzung wie in obiger Klasse wenig elegant. Das Besuchsmuster erlaubt auf elegante Weise, einen Algorithmus funktional in einer Klasse gebündelt zu schreiben, ohne dabei das Konstrukt **`instanceof`** und eine **`if`**-Abfrage zu benötigen. Hierzu definiert man eine allgemeine Schnittstelle, in der für jede Klasse, über den der Algorithmus definiert werden soll, eine Methode `eval` existiert. Das Rückgabeergebnis dieser Methode kann dabei generisch gehalten werden.

Für unsere Heapknoten sieht die entsprechende Schnittstelle wie folgt aus:

NodeVisitor.java

```
package name.panitz.gm;

public interface NodeVisitor<a> {
  public a eval(NNum n)throws Exception;
  public a eval(NChar n)throws Exception;
  public a eval(NConstr n)throws Exception;
  public a eval(NInd n)throws Exception;
  public a eval(NAp n)throws Exception;
  public a eval(NGlobal n)throws Exception;
  public a eval(NJavaObject n)throws Exception;
  public a eval(Node n)throws Exception;
}
```

Möchte man jetzt einen Algorithmus über eine bestimmte Datenstruktur schreiben, so kann man die entsprechende Besucherschnittstelle implementieren. Wie man sieht, wird die Fallunterscheidung jetzt durch eine überladene Methode ausgedrückt.

Beispiel:
Realisieren wir nun also die `toString`-Methode für Heapknoten mitteld eines Besuchers:

ShowNode.java

```
package name.panitz.gm;
public class ShowNode implements NodeVisitor<String> {
  public String eval(NNum n){return "(NNum "+n.n+")";}
  public String eval(NChar n){return "(NChar \'"+n.c+"\')";}
  public String eval(NConstr n){
    String args = "[";
    for (int i:n.args) args=args+i+" ";
    return "(NConstr "+n.name+args+"])";
  }
  public String eval(NInd n){return "(NInd "+n.adr+")";}
  public String eval(NAp n){return "(NAp "+n.n1+","+n.n2+")";}
  public String eval(NGlobal n){
    return "(NGlobal "+n.argc+","+n.i+")";}
  public String eval(NJavaObject n){return "JavaObject";}
  public String eval(Node n)throws Exception{
    throw new Exception("unmatched node type: "+n);
  }
}
```


Aufgrund dessen, daß überladene Methoden nicht wie überschriebene Methoden (in C++ als virtuell markierte Methoden) durch späte Bindung zur Laufzeit aufgelöst werden, sondern statisch bereits zur Übersetzungszeit, wird die Methode `eval` des Besuchsobjekts nicht direkt auf das Objekt angewendet, sondern in den einzelnen Klassen die Methode `visit` (als virtuelle) Methode überschrieben. Um den Algorithmus auszuführen, wird das Besucherobjekt dem Argumentobjekt an die `visit`-Methode übergeben, die dann wiederum die korrekte überladene Version der Methode `eval` des Besuchers ausführt.

Zugegeben, das Besuchsmuster mutet etwas wie schwarze Magie an. Die gute Nachricht ist, daß man sich nicht zu tief mit ihm beschäftigen muß, um es anzuwenden.

Beispiel:
Im folgenen ein kleines Testprogramm für die Anwendung der Besucherklasse `ShowNode`.

UseShowNode.java

```
package name.panitz.gm;
public class UseShowNode{
  static ShowNode visitor = new ShowNode();
  static public String show(Node n)throws Exception{
    return n.visit(visitor);}
  static public void main(String [] _)throws Exception{
    System.out.println(show(new NAp(17,4)));
    System.out.println(show(new NChar('c')));
  }
}
```

Die Anwendung des Besuchsmusters empfiehlt sich immer dann, wenn im Laufe des Entwicklungsprozesses die Datenstrukturen relativ stabil ist, d.h. nicht zu erwarten ist, daß neue Klassen weitere Objektausprägungen definieren werden, hingegen über diese feste Datenstruktur eine Vielzahl komplexer Algorithmen entworfen werden. Das ist in einem Compiler der Fall. Hier ist der Datentyp für den Syntaxbaum als relativ stabil zu betrachten. Über die Knoten dieses Baums werden aber unter Umständen viele verschiedene komplexe Algorithmen zum Typcheck, semantischer Analyse, Optimierungen und ähnlichen geschrieben. So ist insbesondere auch im Javacompiler `javac` selbst das Besuchsmuster in der Implementierung benutzt worden.

Im Laufe des Skriptes werden wir noch eine Reihe unterschiedlicher Besucher auf den Heapknoten definieren.

2.4.2 Heapimplementierung

Nachdem wir nun die Arten von Heapknoten definiert haben, gilt es eine Klasse zur Darstellung des gesammten Heaps zu definieren. Im Prinzip ist der Heap eine Reihung von Knoten. So ist das Hauptcharakteristikum des Heaps eine solche Reihung:

GmHeap.java

```
package name.panitz.gm;
public class GmHeap{
  Node[] heap;
```

Ein Heap hat in der Regel noch freie Speicherplätze. Den index des nächsten freien Speicherplatzes halten wir in einer Variablen `next`, die Gesamtgröße in einer Variable `size`[7]:

GmHeap.java
```
  int size;
  int next=0;
```

Zwei unterschiedliche Konstruktoren werden definiert, einer mit einer Standardgröße des Heaps, der andere bekommt diese als Argument übergeben:

GmHeap.java
```
  public GmHeap(int s){size=s;heap=new Node[size];};
  public GmHeap( ){this(909);};
```

Eine der wichtigsten Operationen auf einem Heap ist, einen neuen Knoten im Heap anzuspeichern. Hierzu definieren wir die Methode `alloc`. Auf den nächsten freien Speicherplatz des Heaps wird der übergebene Heapknoten gespeichert. Die Methode gibt die Position dieses Speicherplatzes als Ergebnis zurück:

GmHeap.java
```
  public int alloc(Node n){
    heap[next]=n;next=next+1;return next-1;
  }
```

Wir sehen zwei Methhoden vor, um den an einer bestimmten Position gespeicherten Knoten zu erfragen. Die zweite Version hiervon löst Indirektionsknoten auf:

GmHeap.java
```
  public Node get(int i){return heap[i];}
  public Node getNode(int i){
    Node n = heap[i];
    while (n instanceof NInd){
      n=get(((NInd)n).adr);
    }
    return n;
  }
```

Schließlich wollen wir noch eine Methode vorsehen, mit der bestimmte Heapknoten verändert werden können:

GmHeap.java
```
  public void update(int i,Node n){heap[i]=n;}
```

Aufgabe 9 Implementieren Sie in der Klasse `GmHeap` eine aussagekräftige Methode `toString`.

Hiermit können wir die Klasse `GmHeap` beenden.

GmHeap.java
```
}
```

[7] Eine in Java unnöltige Maßnahme. weil man eine Reihung anders als in C nach ihrer Größe fragen kann.

Damit haben wir eine Heapstruktur implementiert. Neue Daten werden mit Hilfe der Methode `alloc` im Heap angelegt.

Beispiel:
Folgendes Programm legt im Heap den String `hallo` als Liste an. Je nachdem, wie Sie in der letzten die Methode `toString` in der Klasse `GmHeap` implementiert haben, wird die Ausgabe des Programms ausfallen.

FirstHeap.java
```
package name.panitz.gm.example;
import  name.panitz.gm.*;
import static name.panitz.gm.ConstructorConstants.*;

public class FirstHeap{
  public static void main(String [] _){
    GmHeap heap=new GmHeap();

    int nil = heap.alloc(new NConstr(NIL));
    int o = heap.alloc(new NChar('o'));
    int[] args1 = {o,nil};
    int co = heap.alloc(new NConstr(CONS,args1));
    int l = heap.alloc(new NChar('l'));
    int[] args2 = {l,co};
    int clo = heap.alloc(new NConstr(CONS,args2));
    int[] args3 = {l,clo};
    int cllo = heap.alloc(new NConstr(CONS,args3));
    int a = heap.alloc(new NChar('a'));
    int[] args4 = {a,cllo};
    int callo = heap.alloc(new NConstr(CONS,args4));
    int h = heap.alloc(new NChar('h'));
    int[] args5 = {h,callo};
    heap.alloc(new NConstr(CONS,args5));

    System.out.println(heap);
  }
}
```

2.5 Der G-Maschinenzustand

Die G-Maschine wird definiert als eine Zustandsübergangsmaschine. Die Maschine hat einen Zustand. Jeder Zustand hat maximal einen Nachfolgezustand, der durch Ausführen des aktuellen Befehls erreicht wird. Hat ein Zustand keinen Nachfolgezustand, so hat die Maschine terminiert, der Finalzustand ist erreicht.

Die G-Maschine ist ein Tupel aus mehreren Komponenten. Eine dieser Komponenten ist der Heap, für dessen Implementierung wir bereits eine Klasse egschrieben haben.

Bei einem Zustandsübergang werden bestimmte Komponente der Maschine manipuliert, andere werden nicht verändert.

Die veränderlichen Bestandteile der G-Maschine sind:

- **Heap**: so wie im lertzten Abschnitt bereits entwickelt.
- **Stack (Keller)**: hier liegen Heapadressen. Die Befehle beziehen sich auf die Referenzen des Stacks.
- **Dump**: zum zwischenzeitlichen Abspeichern von Stacks.
- **Programmzähler**: gibt an welcher Befehl für den nächsten Zustansübergang auszuführen ist.

Darüber hinaus gibt es noch Bestandteile der Maschine, die während der Ausführung konstant bleiben:

- **Code**: der auszuführende Code der Maschine.
- **Globale Definitionen**: die Konstruktoren und Funktionen des Programms.
- **Garbage Collector**: zur Freigabe nicht mehr benötigter Heapzellen.

2.5.1 Komponenten der G-Maschine

Wir definieren in diesem Abschnitt die Klasse `GmState`, die einen G-Maschinenzustand vollständig implementiert.

Mit der Klasse `GmHeap` haben wir einen Typ für die Daten, die die G-Maschine manipulieren soll, implementiert. Der Heap stellt die umfangreichste Komponente des Zustands der G-Maschine dar:

GmState.java
```java
package name.panitz.gm;
import java.io.*;
import java.util.*;
import name.panitz.util.*;

public class GmState{
  GmHeap heap;
```

Die G-Maschine benötigt ein Garbage-Collector, der unbenutzte Heapzellen wieder freigibt. Daher sehen wir ein Feld vor, in dem eine Referenz auf einen Garbage-Collector gespeichert ist. Den hier instanziierten Garbage-Collector werden wir in Abschnitt 2.8 entwickeln.

GmState.java
```java
  GC gc;
```

Die G-Maschine wird auf einem Stack basieren. Zusätzlich braucht sie einen Dump, auf dem Stackzustände für spätere Rücksprünge zwischengespeichert werden können. Mit dem Heap sind dieses die wichtigsten und die dynamischen von Befehlen manipulierten Bestandteile des Maschinenzustands.

GmState.java
```java
  GmStack stack=new GmStack();
  GmDump dump=new GmDump();
```


Ein Fab4 Programm wird aus einer Menge von globalen Funktionsdefinitionen und aus einer Menge von Konstrukordefinitionen bestehen. Beides ist Bestandteil des Zustands der G-Maschine. Beide werden nicht mehr von der Maschine verändert. Für globale Funktionen werden wir eine eigene Klasse definieren. Für die Konstruktoren sehen wir einfach eine Reihung von Namen vor. Der Index in dieser Reihung entspricht der Konstruktornummer:

GmState.java
```java
  GmGlobals globals;
  final String[] constructors;
```

Eine der spannensten Komponenten der G-Maschine ist der Code. Wir sehen eine Reihung von Instruktionen vor, die den Maschinencode darstellt und eine Index in diese Reihung, die den aktuell abzuarbeiten Befehl anzeigt.

GmState.java
```java
  public final Instruction[] code;
  int pc=0;
```

Wir sehen ein Feld vor, in dem ein Programmname gespeichert werden kann.[8] Ein weiteres Feld enthält einen Ausgabestrom, auf dem die Maschine Ausgaben tätigen soll. Dieser kann umdefiniert werden. Standardmäßig setzen wir diesen auf den Ausgabestrom des Bildschirms.

GmState.java
```java
  String name="GM Code";
  Writer output=new PrintWriter(System.out);
```

Der Zustand enthält ein Heapobjekt. Der Bequemlichkeit halber definieren wir die Methode `alloc` auf diesen Zustand, die dann entsprechend die `alloc`-Methode des Heapobjekts aufruft:

GmState.java
```java
  public int alloc(Node n){return heap.alloc(n);}
```

2.5.2 Zustandsübergänge

Zunächst sehen wir eine Testmethode vor, die prüft ob es einen Nachfolgezustand gibt. Diese Abfrage prüft, ob es noch auszuführenden Code gibt, oder ob die Ausführung beendet ist und der Finalzustand erreicht wurde. Die Maschine soll im Finalzustand sein, wenn der Programmzähler auf einem illegalen Wert steht.

GmState.java
```java
  public boolean finalState(){return pc<0||pc>=code.length;}
```

Ein Zustandsübergang bedeutet, daß der Befehl, auf den der Befehlszähler zeigt, ausgeführt und der Befehlszähler um eins erhöht wird.

[8] Tatsächlich wird dieses bisher in der noch folgenden Implementierung nicht mehr benutzt.

GmState.java
```
  public void evalStep()throws Exception{
    Instruction i = code[pc];
    pc=pc+1;
    i.process(this);
  }
```

Die gesammte Ausführung der Maschine besteht aus einer Schleife, in der solange nicht der Finalzustand erreicht wurde, der nächste Schritt gemacht wird.

GmState.java
```
  public void eval()throws Exception{
    while (!finalState()) evalStep();
  }
```

2.5.3 Konstruktor und toString-Methode

Wie für jede anständige Klasse sehen wir eine Methode `toString` und sinnvolle Konstruktoren vor.

Die Methode `toString` soll zunächst Stack, Heap und Code der Maschine als Stringdarstellung erzeugen:

GmState.java
```
  public String toString(){
    StringBuffer result=new StringBuffer("Stack:\n[");
    boolean first=true;

    for (Integer i:stack){
      if (first) first=false;else result.append(",");
      result.append(i+" -> "+heap.getNode(i).toString());
    }
    result.append("]");
    result.append("\n\nHeap\n");

    result.append(heap.toString());

    result.append("\n\nCode:\n[");
    for (Instruction c:code) result.append(c+";\n");
    result.append("]");

    return result.toString();
  }
```

Zwei Konstruktoren werden uns nützlich sein: Im konstruktor wird auch die garbage collection instanziiert, allerdings nicht direkt sondern über eine Frabrikmethode. Eine statische Methode liefert uns ein garbage collection Objekt, ohne daß wir wissen, von welcher konkreten Klasse dieses sein wird.

GmState.java
```
  public GmState(Instruction[] code,GmHeap heap
                ,GmGlobals glob,String[] constructors){
    this.code=code;
    this.heap=heap;
    this.globals=glob;
    this.constructors=constructors;
    gc=GCFactory.getGCInstance(this);
  }

  public GmState(Instruction[] code){
    this(code,new GmHeap(),new GmGlobals(),new String[0]);
  }
```

2.5.4 Extras für Aufrufe externer Funktionen

Die Methoden in diesem Abschnitt werden benötigt zur Realisierung des Aufrufs externer Javamethoden. Wir werden diese vorerst nicht weiter hinterfragen.

GmState.java
```
  String[] progArgs;
  public String[] getProgArgs(){return progArgs;}
```

GmState.java
```
  Integer javaCallAddress=null;
  int getJavaCallAddress(){
    if (javaCallAddress==null)
      javaCallAddress
        =((NGlobal)heap.get((globals.get("javaCall")))).i;
    return  javaCallAddress;
  }

  Integer staticJavaCallAddress=null;
  int getStaticJavaCallAddress(){
    if (staticJavaCallAddress==null)
      staticJavaCallAddress
        =((NGlobal)heap.get((globals.get("staticJavaCall")))).i;
    return  staticJavaCallAddress;
  }
```

2.5.5 Hilfsmethode für Fehlerausgabe

Sollte in der Maschinenausführung ein Fehler auftreten, so wollen wir wenigstens versuchen auszugeben, bei der Ausführung welcher globalen Funktion dieser Fehler auftrat. Diese Informtion ist nicht direkt in der Maschine enthalten und muß mühsam berechnet werden.

GmState.java
```
  public String getCurrentFunction(){
    final Map<Integer,String> functionEntries
```

```
      = new HashMap<Integer,String>();
    for (Map.Entry<String,Integer> entry:globals.entrySet()){
      final int codeAdr =((NGlobal)heap.get(entry.getValue())).i;
      functionEntries.put(codeAdr,entry.getKey());
    }
    Object [] values = functionEntries.keySet().toArray();
    Arrays.sort(values);

    int jumper=0;

    for (Object val:values){
      if (((Integer)val)>=pc)break;
      jumper=(Integer)val;
    }
    return functionEntries.get(jumper);
  }
}
```

Damit haben wir eine Klasse für den G-Maschinenzustand vollständig definiert. Jetzt müssen wir uns noch Klassen für die einzelnen Komponenten des Zustands definieren.

2.5.6 Der Stack

Der Stack soll eine dynamisch wachsene Sammlung von Heapadressen sein. Hierzu bietet sich an auf eine der Stansardlistenimplementierungen von Java zurückzugreifen. Unser Stack soll eine Spezialisierung der Java-Klasse `ArrayList` mit dem Elementtyp `Integer` sein:

GmStack.java
```
package name.panitz.gm;
import java.util.*;

public class GmStack extends ArrayList<Integer>{
```

Für einen Stack benötigen wir die Methode `push`, die ein neues Element oben auf den Stack ablegt. Hierzu benutzen wir einfach die Methode `add` der Klasse `ArrayList`, die ein Element ans Ende der Liste anhängt.

GmStack.java
```
  public void push(int i){add(i);}
```

Die inverse Operation zu `push` ist `pop`. Sie gibt das letzte Element der Liste (oberste Element des Stacks) zurück und löscht es aus der Liste.

GmStack.java
```
  public int pop(){
    int result=get(size()-1);
    remove(size()-1);
    return result;
  }
```


Zwei weitere kleine Methoden erlauben es, Elemente von oben des Stack ausgehend zu erfragen, ohne den Stack zu verändern.

GmStack.java
```
  public int peek(int n){
    return get(size()-1-n);
  }
  public int last(){return get(size()-1);}
}
```

2.5.7 Der Dump

Der Dump wird benötigt, um einen Programmzähler zusammen mit einen Stack zwischenzuspeichern, um später wieder auf diese aufzusetzen. Auch hierfür können wir uns wieder der Klasse `ArrayList` bedienen. Die Elemente sind jetzt vom Typ `Pair<Integer,GmStack>`, um Paare von Programmzählern und Stacks abspeichern zu können.

GmDump.java
```
package name.panitz.gm;
import java.util.*;
import name.panitz.util.*;

public class GmDump extends ArrayList<Pair<Integer,GmStack>>{}
```

2.5.8 Auflistung der globalen Funktionen

Für globale Funktionsnamen brauchen wir eine Abbildung, die die Klartextnamen der Funktion auf eine Heapadresse abbildet. Eine Abbildung wird in der Informatik durch eine `Map` modelliert. In Java können wir hierfür die Standardklasse `HashMap` mit entsprechenden Elementtypen benutzen.

GmGlobals.java
```
package name.panitz.gm;
import java.util.*;

public class GmGlobals extends HashMap<String,Integer>{}
```

2.6 Der Befehlssatz

Alle Bestandteile, die einen Maschinenzustand ausmachen sind somit definiert. Es folgt der Befehlssatz der Maschine. Den Befehlssatz modellieren wir objektorientiert. Ein Befehl hat eine Methode `process`, in der das Befehlsobjekt bestimmt, wie durch seine Ausführung ein G-Maschinenzustand manipuliert wird. Wir modellieren dieses über eine Schnittstelle, deren Existenz wir oben bereits angenommen haben.

Instruction.java
```
package name.panitz.gm;
public interface  Instruction{
```

```
  public GmState process(GmState st)throws Exception;
}
```

Die einzelnen Befehle der Maschine müssen diese Schnittstelle implementieren.

Unser erster einfacher Befehl hat überhaupt keine Wirkung und heißt daher einfach `Nop`. In seiner Methode `process` gibt er den Maschinenzustand unverändert zurück.

Nop.java
```
package name.panitz.gm;
import name.panitz.util.*;

public class Nop implements Instruction{
  public GmState process(GmState st){return st;}
  public String toString(){return "NOP ";}
}
```

Unser nächster einfacher Befehl zwingt die Maschine zu terminieren, indem sie den Maschinenzustand auf einen Finalzustand setzt. Der Befehl soll dementsprechend `End` heißen. Wird der Befehl auf einem Zustand ausgeführt, so wird der Programmzähler auf den illegalen Wert `-1` gesetzt:

End.java
```
package name.panitz.gm;
import name.panitz.util.*;

public class End implements Instruction{
  public GmState process(GmState st){
    st.pc=-1;
    return st;
  }
```

Zusätzlich implementieren wir textuelle Darstellung dieser Befehlsobjekte.

End.java
```
  public String toString(){return "END ";}
}
```

In der Folge werden wir alle Befehle der G-Maschine definieren und implementieren. Um eine Folge von Befehlen auszuprobieren, definieren wir uns einer kleine Hilfsmethode, die eine Befehlsfolge auf einem Standardmaschinenzustand zur Ausführung bringt und den Finalzustand schließlich auf der Konsole ausgibt.

RunTest.java
```
package name.panitz.gm.test;
import name.panitz.gm.*;
public class RunTest{
  public static void run(Instruction [] code)throws Exception{
    GmState st = new GmState(code);
    st.eval();
    System.out.println(st);
  }
}
```

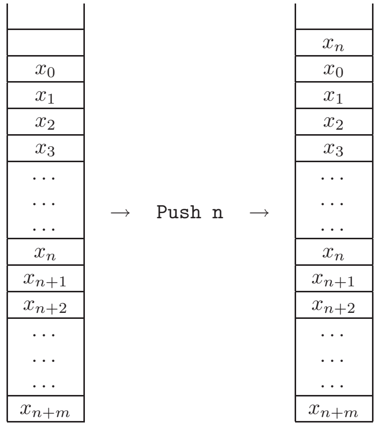

2.6.1 Stackmanipulatoren

Eine erste Gruppe von Befehlen der G-Maschine manipuliert den Stack. Diese Gruppe besteht aus den Befehlen: `Push`, `Pop`, `PushInt`, `PushChar` und `Slide`. Diese werden in diesem Abschnitt definiert und implementiert. Eine Gemeinheit ist, daß die Befehle `Pop` und `Push` nicht identisch sind, mit den Operationen `push` und `pop` auf dem Stack.

Push

Der Befehl `Push` hat eine Zahl n als Parameter. Er bewirkt, daß das n-te Element von der Stackspitze ausgehend ein weiteres Mal auf den Stack abgelegt wird.

Somit manipuliert die Ausführung des Befehls `Push n` den Stack auf fogende Weise.



Wir schreiben eine Klasse, die diesen Befehl darstellt. Der Befehl hat eine ganze Zahl als Argument. Diese wird in einem Feld der Klasse gespeichert und im Konstruktor initialisiert.

Push.java
```
package name.panitz.gm;
public class Push implements Instruction{
  int n;
  public Push(int n){this.n=n;}
```

Bei der Ausführung wird auf dem Stack des Maschinenzustands zunächst das n-te Element erfragt und es dann auf dem Stack oben abgelegt.

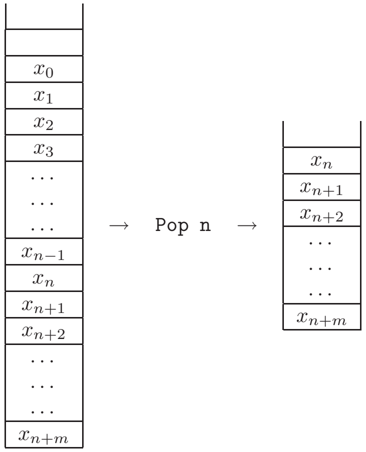
Push.java
```
  public GmState process(GmState st){
    int np1 = st.stack.peek(n);
    st.stack.push(np1);
    return st;}
```

Schließlich sei noch textuelle Darstellung des Befehls implementiert.

Push.java
```
  public String toString(){return "PUSH "+n;}
}
```

Pop

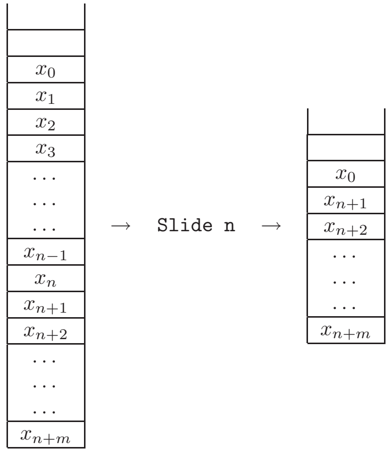
Auch der Befehl `Pop` der G-Maschine hat eine Zahl als Argument. Sie gibt an, wie viele Elemente vom Stack zu entfernen sind:



Aufgabe 10 Implementieren Sie eine Klasse `Pop`, die den G-Maschinenbefehl `Pop` umsetzt.

Slide

Auch der Befehl `Slide` hat ein Argument. Er funktioniert ähnlich wie der Befehl `Pop` mit dem Unterschied, daß das oberste Stackelement erhalten bleibt:



Aufgabe 11 Implementieren Sie eine Klasse `Slide`, die den G-Maschinenbefehl `Slide` umsetzt.

Neue Heapzellen für primitive Werte

Die nächsten zwei Befehle legen neue Adressen auf dem Stack ab. Die neue Adresse bezeichnet dabei eine Heapzelle, in der neue primitive Wert abgelegt ist. Da wir zwei primitive Werte auf dem Heap kennen, gibt es zwei Varianten dieses `Push`-Befehls.

PushInt Zur Ausführung des Befehls `PushInt n` ist für die Zahl n ein `NNum` Heapknoten zu erzeugen. Dieser ist mit `alloc` im Heap zu speichern und die dabei erhaltene Heapadresse auf dem Stack abzulegen. Bevor wir mit `alloc` den Knoten in eine neue Heapzelle schreiben ist mit dem Objekt zur garbage collection zu testen, ob der Heap nicht voll ist.

PushInt.java
```
package name.panitz.gm;
public class PushInt implements Instruction{
  int n;
  public PushInt(int n){this.n=n;}

  public GmState process(GmState st){
    st.gc.check();
    int adr=st.alloc(new NNum(n));
    st.stack.add(adr);
    return st;
  }

  public String toString(){return "PUSHINT "+n;}
}
```

PushChar Analog zu `PushInt` gibt es den Befehl `PushChar`.

Aufgabe 12 Implementieren Sie eine Klasse `PushChar`, die den G-Maschinenbefehl `PushChar` umsetzt.

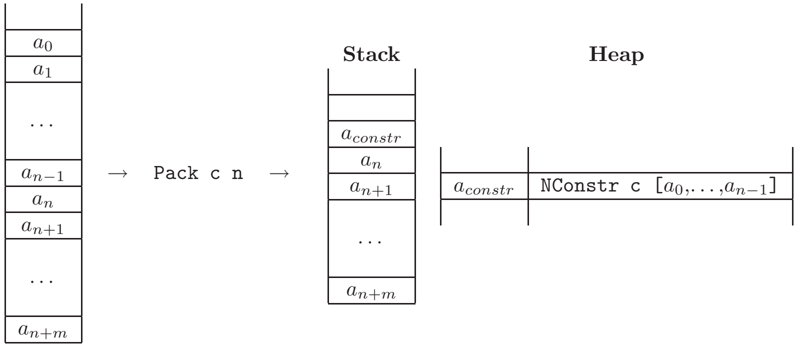
Wir haben bereits eine Methode zum Testen von Codesequenzen bereitgestellt. Sie können jetzt Ihre Lösungen zu obigen Aufgaben mit kleinen Klasse der folgenden Art testen:

StackExample.java
```
package name.panitz.gm.test;
import name.panitz.gm.*;
public class StackExample {
  public static void main(String [] args)throws Exception{
    Instruction [] code
      = {new PushInt(42)
        ,new Push(0)
        ,new PushChar('g')
        ,new Slide(1)
        ,new Pop(1)
        ,new End()
        };
    RunTest.run(code);
  }
}
```

2.6.2 Konstruktoren

Mit den bisherigen Befehlen lassen sich Knoten mit primitiven Werten im Heap anlegen. Jetzt werden wir einen Befehl kennenlernen der es ermöglicht, einen Konstruktorknoten `NConstr` im Heap anzulegen. Hierzu definieren wir den G-Maschinenbefehl `Pack`. Er hat zwei Argumente: den Namen (als seine Nummer, nicht im Klartext) des Konstruktors, für den ein Heapknoten angelegt werden soll, und die Stelligkeit des Konstruktors.

Der Befehl `Pack c n` nimmt die obersten n Elemente vom Stack, um sie als Argumente des Konstruktorknotens zu benutzen. Mit diesen Argumenten wird ein neuer Konstruktorknoten mit Namen c auf dem Heap abgelegt und dessen Adresse schließlich auf den Stack gelegt.



Wir können die entsprechnde Klasse definieren:

Pack.java
```
package name.panitz.gm;
public class Pack implements Instruction{
  int name;
  int argc;
  public Pack(int n,int c){name=n;argc=c;}

  public String toString(){return "PACK "+name+" "+argc;}

  public GmState process(GmState st){
```

Aufgabe 13 Implementieren Sie den Rumpf Methode `process` nach entsprechender Spezifikation. Beachten Sie auch hier wieder, zunächst mit dem garbage collector zu checken, ob es freie Heapzellen gibt.

Nachfolgender Code ist das Ende der Klasse `Pack`. Am Ende der Methode wird der geänderte Zustand zurückgegeben und alle Klammern noch geschlossen

Pack.java
```
    return st;
  }
}
```

Wir können jetzt ersten Code ausführen, der tatsächliche Graphstrukturen auf dem Heap hinterläßt. Folgendes kleines Beispiel erzeugt die Liste des Strings `"hallo"` auf dem Heap.

ConstrExample.java
```
package name.panitz.gm.test;
import name.panitz.gm.*;
import static name.panitz.gm.ConstructorConstants.*;

public class ConstrExample {
  public static void main(String [] args)throws Exception{
    Instruction [] code
      = {new Pack(NIL,0)
        ,new PushChar('o')
        ,new Pack(CONS,2)
        ,new PushChar('l')
        ,new Pack(CONS,2)
        ,new PushChar('l')
        ,new Pack(CONS,2)
        ,new PushChar('e')
        ,new Pack(CONS,2)
        ,new PushChar('h')
        ,new Pack(CONS,2)
        ,new End()
        };
    RunTest.run(code);
  }
}
```

2.6.3 Operatoren

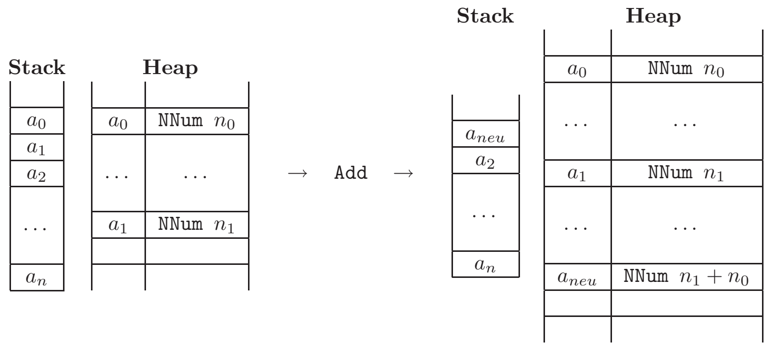
Wir wollen selbstverständlich auch rechnen. Hierzu sieht die G-Maschine Operatoren der gängigen Grundrechenarten und Vergleiche zur Verfügung.

Arithmetische Operatoren

One and one and one is three.
Lennon/McCartney

Bis auf die eigentliche Operation, funktionieren alle arithmetischen Operatoren der G-Maschine gleich. Hole aus den oberen beiden Adresse auf dem Stack die Operanden. Führe mit diesen die Operation durch, lege für das Ergebnis eine neue Heapadresse an und lege diese auf dem Stack ab.

Für den Additionsbefehl `Add` ergibt sich folgendes Bild:



Wir können eine gemeinsame abstrakte Oberklasse für die arithmetischen Operatoren definieren, in der die eigentliche Operation abstrakt gehalten wird. Erst in den vier Unterklassen `Add`, `Sub`, `Mult` und `Div` wird die eigentliche Operation `op` implementiert:

ArithOp.java

```java
package name.panitz.gm;
public abstract class ArithOp implements Instruction{
  abstract public int op(int x,int y);

  public GmState process(GmState st){
    Node n1=st.heap.getNode(st.stack.pop());
    Node n2=st.heap.getNode(st.stack.pop());
    int i1=0;
    int i2=0;

    if (n1 instanceof NChar)i1=(int)((NChar)n1).c;
    else i1=((NNum)n1).n;

    if (n2 instanceof NChar)i2=(int)((NChar)n2).c;
    else i2=((NNum)n2).n;
```



```java

    st.gc.check();
    int a =st.alloc(new NNum(op(i1,i2)));
    st.stack.push(a);
    return st;}
}
```

Wir haben zusätzlich die implizite Konversion von `char` Werten auf `int`-Werte in dieser Klasse vorgenommen.

Die eigentlichen Operationsbefehle sind jetzt Unterklassen der Klasse `ArithOp`.

Add.java

```java
package name.panitz.gm;

public class Add extends ArithOp {
  public int op(int x,int y){return x+y;}

  public String toString(){return "ADD";}
}
```

Aufgabe 14 Implementieren Sie die Klassen `Sub`, `Mult` und `Div`.

Es läßt sich jetzt mit der G-Maschine auf Zahlen rechnen. Beachten Sie, daß nicht auf dem Stack gerechnet wird. Auf dem Stack stehen nur Heapadressen, in denen die Operanden zu finden sind.

ArithExample.java

```java
package name.panitz.gm.test;
import name.panitz.gm.*;
import static name.panitz.gm.ConstructorConstants.*;

public class ArithExample {
  public static void main(String [] args)throws Exception{
    Instruction [] code
      = {new PushInt(21)
        ,new PushInt(2)
        ,new PushInt(17)
        ,new PushInt(4)
        ,new Add()
        ,new PushInt(6)
        ,new Mult()
        ,new Div()
        ,new Sub()
        ,new End()
        };
    RunTest.run(code);
  }
}
```

Vergleichsoperatoren

Die zweite Gruppe von Operatoren sind die Vergleichsoperatoren. Sie haben einen bool'schen Wert als Ergebnis. Wir kennen keine primitiven bool'schen Werte sondern benutzen für dies nullstellige Konstruktoren.

Auch hier gehen wir so vor, daß wir eine abstrakte allgemeine Oberklasse für Vergleichsoperatoren definieren:

CompareOp.java
```
package name.panitz.gm;
import static name.panitz.gm.ConstructorConstants.*;

abstract public class CompareOp implements Instruction{
  abstract boolean op(int x,int y);
  public GmState process(GmState st){
    Node n1=st.heap.getNode(st.stack.pop());
    Node n2=st.heap.getNode(st.stack.pop());
    int i1=0;
    int i2=0;

    if (n1 instanceof NChar)i1=(int)((NChar)n1).c;
    else i1=((NNum)n1).n;

    if (n2 instanceof NChar)i2=(int)((NChar)n2).c;
    else i2=((NNum)n2).n;

    st.gc.check();
    int a =  op(i1,i2)
             ?st.alloc(new NConstr(TRUE))
             :st.alloc(new NConstr(FALSE));
    st.stack.push(a);
    return st;}
}
```

Und die eigentlichen Operatoren werden als Unterklassen derselben implementiert.

Lt.java
```
package name.panitz.gm;

public class Lt extends CompareOp{
  public boolean op(int x,int y){return x<y;}
  public String toString(){return "Lt";}
}
```

Aufgabe 15 Schreiben Sie die Klassen für die übrigen Vergleichsoperatoren: `Eq`, `Gt`, `Ge` und `Le`.

2.6.4 Funktionsaufrufe

Die G-Maschine kennt fünf Befehle zur Behandlung und Auswertung von Funktionsanwendungen: `PushGlobal`, `MkAp`, `UnwindNode`, `Eval` und `Update`.

Funktionsadressen auf dem Stack legen

Um eine Funktionsawendung in der G-Maschine auszudrücken, ist zunächst die Adresse des Funktionsknotens im Heap auf den Stack abzulegen. Hierzu dient der Befehl `PushGlobal name`. Er hat ein Stringargument, das den Klartextnamen der globalen Funktion darstellt. Bei der Auswertung dieses Befehls wird für den Namen der Funktion in der globalen Abbildung die entsprechende Adresse des Funktionsknotens im Heap nachgeschlagen `int adr = st.globals.get(name)`. Diese wird dann auf den Stack abgelegt `st.stack.push(adr);`.

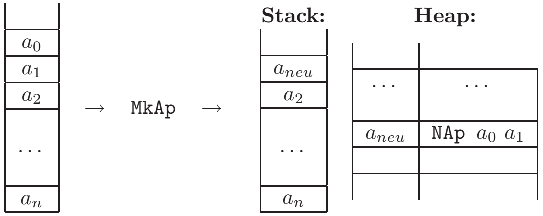
Wir erhalten folgende kurze Klasse:

PushGlobal.java
```
package name.panitz.gm;

public class PushGlobal implements Instruction{
  String name;
  PushGlobal(String n){name=n;}

  public GmState process(GmState st){
    int adr = st.globals.get(name);
    st.stack.push(adr);
    return st;
  }

  public String toString(){return "PUSHGLOBAL "+name;}
}
```

Applikationsknoten erzeugen

Mit dem vorherigen Befahl haben wir jetzt eine Möglichkeit Adressen auf globale Funktionsobjekte auf dem Stack zu speichern. Funktionen sind erst dann interessant, wenn wir sie auf bestimmte Argumente anwenden können. Wir haben uns in Abschnitt 2.4.1 bereits davon überzeugt, daß es reicht, nur einstellige Funktionsanwendungen vorzusehen. Es gibt einen expliziten Heapknoten, der ausdrückt, daß ein bestimmter Knoten auf einen anderen anzuwenden ist. Der G-Maschinenbefehl `MkAp` erlaubt es mit den oberen beiden Stackadressen einen solchen Knoten `NAp` zu erzeugen.

Die operationale Semantik des Befehls `MkAp` läßt sich wie folgt illustrieren:



Aufgabe 16 Implementieren Sie die Klasse `MkAp`, die den entsprechenden G-Maschinenbefehl realisiert. Denken Sie wieder daran, zunächst einen Check vom garbage collection Objekt durchführen zu lassen.

Funktionsauswertung anstoßen

Der Befehl `Unwind` ist der komplexeste der G-Machine. Er bereitet die Auswertung einer Funktionsanwendung vor. Mit den bisherigen Befehlen haben wir nur die Möglichkeit auf dem Heap einen Graph zu erzeugen, der Funktionsanwendungen darstellt. Jetzt soll tatsächlich die Funktion für bestimmte Argumente ausgewertet werden.

Der Befehl `Unwind` unterscheidet nach der Art des Heapknotens, der mit dem obersten Stackelement bezeichnet wird. Für die verschiedenen Arten von Heapknoten verhält sich der Befehl `Unwind` unterschiedlich. Deshalb bietet es sich an, den Befehl `Unwind` über eine Besucherklasse für Knoten zu realisieren. Dazu werden wir den Knotenbesucher `UnwindNode` definieren. Bei der Ausführung des Befehls `Unwind` wird entsprechend der durch den obersten Stackknoten referenzierte Knoten mit diesem Besucher behandelt.

Es ergibt sich folgende Befehlsklasse:

Unwind.java
```
package name.panitz.gm;

public class Unwind implements Instruction{
  public GmState process(GmState st)throws Exception{
   return st.heap.get(st.stack.peek(0)).visit(new UnwindNode(st));
  }

  public String toString(){return "UNWIND";}
}
```

Besucherklasse für Unwind

Der Befehl `Unwind` wird durch eine Besucherklasse ausgedrückt. Da das Ergebnis einer Befehlsausführung ein neuer Maschinenzustand ist, definieren wir einen Knotenbesucher mit dem Ergebnistyp `GmState`. Beim Instanziieren der Besucherklasse übergeben wir den Maschinenzustand, auf dem der Befehl ausgeführt werden soll:

UnwindNode.java
```
package name.panitz.gm;
import name.panitz.util.Pair;
```



```

public class UnwindNode implements NodeVisitor<GmState>{
  GmState st;
  public UnwindNode(GmState st){this.st=st;}
```

Jetzt können wir für die verschiedenen Arten von Heapknoten definieren, wie der Befehl `Unwind` für die wirkt.

Unwind für ausgewertete Knoten

Handelt es sich bei den Heapknoten um ein bereits als Daten vorliegenden Knoten, so ist keine Auswertung für diesen Knoten anzustoßen. Solche Knoten sind Knoten die primitive Werte darstellen, also `NNum` und `NChar`, Konstruktorknoten `NConstr` sowie der Heapknoten, den wir erst später kennenlernen, zur Speicherung von externen Javaobkjekten `NJavaObject`. Da für diese drei Knotenarten der Befehl `Unwind` identisch operiert, verweisen wir in den entsprechenden drei überladenen Methodenvarianten auf eine gemeinsame Hilfsmethode `evalConstant`.

UnwindNode.java
```
  public GmState eval(NNum n){return evalConstant(n);}
  public GmState eval(NChar n){return evalConstant(n);}
  public GmState eval(NConstr n){return evalConstant(n);}
  public GmState eval(NJavaObject n){
    return evalConstant(n);
  }
```

Für diese Datenknoten für den Befehl `Unwind` keine Aktion durchzuführen; aber er stößt in diesem Fall an, daß eventuell ein alter Stack und Programmzähler vom Dump reaktiviert wird. Sollte es auf dem Dump einen geparkten Zustand geben, so wird dieser vom Dump genommen. Programmzähler und Stack werden auf die des Dumps gesetzt. Das oberste Element des ursprünglichen Stacks auf den neuen Stack abgelegt.

UnwindNode.java
```
  private GmState  evalConstant(Node n){
   if (st.dump.size()>0){
      Pair<Integer,GmStack> d=st.dump.get(0);
      st.dump.remove(0);
      d.e2.push(st.stack.pop());
      st.stack=d.e2;
      st.pc=d.e1;
    }
    return st;
  }
```

Unwind für Indirektionsknoten

Ist der Befehl `Unwind` auf einen Indirektionsknoten auszuführen, so wird dem Indirektionsknoten gefolgt und der Befehl auf den Knoten, auf den die Indirektion zeigt augdgeführt. Die Befehlsausführung hat dann drei Schritte:

- entferne das oberste Stackelement.
- bestimme die Adresse, auf die die Indirektion zeigt und lege diese auf den Stack.
- besuche den Knoten, auf den die neue Adresse zeigt mit demselben `Unwind`-besucher.

UnwindNode.java
```
public GmState eval(NInd n)throws Exception{
  st.stack.pop();
  st.stack.push(n.adr);
  return st.heap.get(n.adr).visit(this);}
```

Unwind für Applikationsknoten

Stößt der `Unwind`-Befehl auf einen Applikationsknoten. So ist die Adresse des angewendeten Knotens auf den Stack zu legen, und der Knoten auf den dieser zeigt schließlich mit dem Besucher zu bearbeiten.

UnwindNode.java
```
public GmState eval(NAp n)throws Exception{
  st.stack.push(n.n1);
  return st.heap.get(n.n1).visit(this);}
```

Unwind von Funktionsknoten

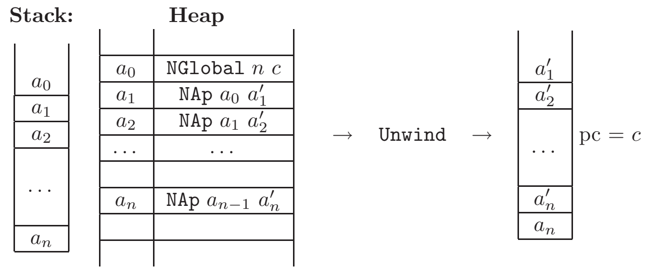
Am komplexesten ist der Befehl `Unwind`, wenn er auf eine Funktionsdefinition stößt. Hier sind drei Unterfälle zu berücksichtigen:

- es sind nicht ausreichend genug Argumente für die Funktion auf dem Stack.
- es ist die benötigte Anzahl von Argumenten auf dem Stack.

zu wenig Argumente wenn zu wenig Argumente auf dem Stack für die abzuarbeitende Funktion liegen, dann liegt ein Fehler vor. Wir können in diesem Fall die Maschine ein entsprechendes Ausnahmeobjekt werfen lassen. Dabei können wir insbesondere auf die Methode `getCurrentFunction` zurückgreifen, die bei der Diagnose des Fehlers eine große Hilfe ist.

UnwindNode.java
```
public GmState eval(NGlobal n){
  if (st.stack.size()<=n.argc)
    throw new RuntimeException
       ("Unwinding to few Arguments to "+n+" in function: "
        +st.getCurrentFunction()
        +"\n found "+st.stack.size()
        + " required: "+n.argc+" arguments");
```


passende Argumenteanzahl Ist die Argumentanzahl auf dem Stack passend zu Definition der globalen Funktions, so wird der Stack umorganisiert. Es wird davon ausgegangen, daß die Argumente über Applikationsknoten auf dem Stack referenziert werden. Statt dieser Applikationsknoten sind die Adressen der eigentlichen Argumente auf den Stack abzulegen. Schließlich ist der Programmzähler auf den den Codeeinstieg der entsprechenden Globalen Funktion zu setzen.



Es folgt der Code für diesen Zustandsübergang. Zusätzlich zu dem Fall, daß genau richtig viele Argumente auf dem Stack liegen, werden die Fälle behandelt, daß es sich bei der globalen Funktion um die eingebauten Spezialfunktionen für Aufrufe externer Javamethoden handelt.

UnwindNode.java
```
    st.stack.pop();
    GmStack newArgs = new GmStack();
    for (int i=0;i<n.argc;i++){
      int a = i==n.argc-1?st.stack.peek(0):st.stack.pop();
      newArgs.add(0,((NAp)st.heap.get(a)).n2);
    }
    st.stack.addAll(newArgs);
    st.pc=n.i;
    return st;
  }
```

Die Arbeitsweise den Unwind-Befehls auf einen Funktionsapplikationsknoten ist in Abbildung 2.4 noch einmal mit graphischer Darstellung des Heaps zu sehen. Der linke Stack ist der Zustand des Stacks vor der Befehlsausführung, der rechte derselbe danach.

Jetzt ergibt sich die Arbeitsweise der G-Maschine für Funktionsaufrufe. Zunächst wird der Ausdruck der Funktionsanwendung als Graph gebaut. Hierzu werden soviele Applikationsknoten `NAp` im Heap erzeugt, wie die Funktionsanwendung Knoten Argumente hat. Ist dieser Graph im Heap gebaut, kann ein `Unwind`-Befehl den Sprung zum Code dieser Funktion generieren. Bevor der Programmzähler allerdings auf den Code der angewendeten Funktion gesetzt wird, wird der Stack umarrangiert. Statt auf dem Stack die Adressen der Anwendungsknoten `NAp` liegen zu haben, werden die eigentlichen Argumentadressen auf dem Stack gelegt.

Man beachte, daß der unterste Applikationsknoten auf dem Stack erhalten bleibt. Diesen werden wir im Befehl `Update` benutzen, um den Anwendungsknoten im Heap zu überschreiben.

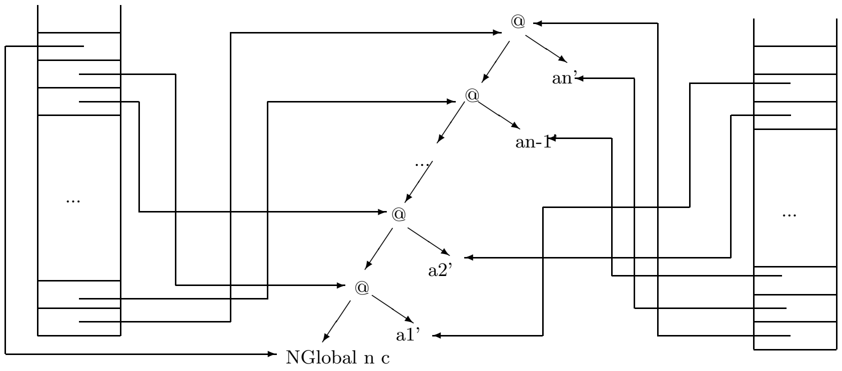


Abbildung 2.4: Unwind auf eine Funktionsapplikation.

nicht abgedeckte Fälle

Entsprechend der Idee von Besuchsklassen, sehen wir noch die allgemeine Überladung der Methode `eval` vor, die anzeigt, daß für alle übrigen Fälle keine eigene Methode überladen war.

UnwindNode.java
```
  public GmState eval(Node n)throws Exception{
    throw new Exception("unmatched pattern: "+n);
  }
}
```

Auswertung Anstoßen

Der Befehl `Unwind` sorgt dafür, daß direkt zum Code einer bestimmten Funktion gesprungen wird. Die G-Maschine kennt einen weiteren Befehl, der die Auswertung eines Ausdrucks anstößt: `Eval`. Für den Befehl `Eval` wird der aktuelle Stack und Programmzähler der Maschine auf den Dump abgelegt. Ein neuer Stack mit der obersten Stackadresse wird erzeugt. Der Befehl `Unwind` wird für den Knoten, auf den der oberste Stacknoten zeigt, ausgeführt.

Die folgende Klasse soll uns als Spezifikation des Befehls `Eval` genügen:

Eval.java
```
package name.panitz.gm;
import name.panitz.util.*;
public class Eval implements Instruction{
  public GmState process(GmState st)throws Exception{
    GmStack stack = new GmStack();
    int a=st.stack.pop();
    stack.add(a);
    st.dump.add(0,new Pair<Integer,GmStack>(st.pc,st.stack));
    st.stack=stack;
    return st.heap.get(a).visit(new UnwindNode(st));
```



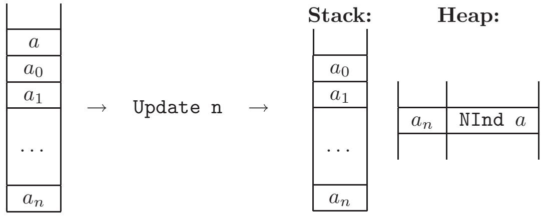
```
  }

  public String toString(){return "EVAL";}
}
```

Wir definieren uns eine Hilfsmethode `whnf`. Diese stößt den Befehl `Eval` an. Wann immer wir in irgendeinen Kontext den obersten Knoten des Stacks ausgewertet haben wollen, können wir diese statische Methode auf den Maschinenzustand anwenden.

WHNF.java
```
package name.panitz.gm;
public class WHNF{
  static void whnf(GmState st)throws Exception{
    int i = st.pc;
    st.pc=-1;
    new Eval().process(st);
    st.eval();
    st.pc=i;
  }
}
```

Funktionsergebnis Überschreiben

Mit den letzten Befehlen können wir endlich Funktionensanwendungen erzeugen und deren Auswertung anstoßen. Die Auswertung einer Funktionsanwendung erzeugt in der Regel einen neuen Heapknoten mit dem Ergebnis der Funktionsanwendung (oder liefert eine bereits eine im Heap bekannte Adresse als Ergebnis). Wenn wir mehrere Zeiger auf die gleiche Funktionsanwenung haben, so kann es dazu führen, daß diese Anwendung auch mehrfach ausgeführt wird. Wie wir im Einführungskapitel bereits gesehen haben, kann dieses bei normaler Auswertungsreihenfolge sehr schnell passieren. Dieses wollen wir möglichst verhindern. Hierzu überschreiben wir den Anwendungsknoten im Heap mit einem Indirektionsknoten auf das Ergebnis dieser Anwendung. Für diese Operation stellt die G-Maschine den Befehl `Update` zur Verfügung. Er hat eine ganze Zahl n als Argument. Der Befehl `Uodate n` baut für die oberste Stackadresse einen Indirektionsknoten und überschreibt die n-te Adresse auf dem Stack im Heap mit diesem Indirektionsknoten:



Aufgabe 17 Implementieren Sie die Klasse `Update`.

Beispielcode

Betrachten wir einmal, wie wir mit den bisherigen G-Maschinenbefehlen Funktionen und deren Anwendungen ausdrücken können. Nehmen wir hierzu das kleine Programm aus der Einführung, mit dem wir illustriert haben, wie die Graphenreduktion zur lazy Auswertung eines Programms führt, d.h. verhindert das trotz der normalen Auswertungsreihenfolge Funktionsanwendungen nicht mehrfach ausgewertet werden.

```
square x = x*x;
main=square (2+2)
```

Schauen wir uns zunächst den G-MAschinencode an, der die beiden Funktionsdefinitionen operational umsetzt. Beim Eintritt in eine Funktion gehen wir nach der Spezifikation von `Unwind` davon aus, daß die oberen n Stackeinträge die Adressen der Argumente des Funktionsanwenung sind, und darunter noch die Adresse des Anwendungsknoten im Heap zu finden ist

Code für square Mit folgenden Code, kann die Funktion `square` in der G-Maschine realisiert werden:

```
[PUSH 0
,EVAL
,PUSH 1
,EVAL
,MULT
,UPDATE 1
,POP 1
,UNWIND]
```

Das Argument wird oben auf den Stack gelegt und ausgewertet `PUSH 0, EVAL`. Dann ist das Argument ein weiteres Mal auf den Stack zu legen und auszuwerten `PUSH 1, EVAL`. Hier könnte der `EVAL`-Befehl auch fehlen, weil wir ja bereits dafür gesorgt haben, daß der Wert des Arguments ausgewertet wurde. Wenn unser Compiler später den Code erzeugt, müßte er aber dieses erst beweisen und solange er das nicht konnte, muß der zweite `EVAL`-Befehl eingefügt sein. Mit `MULT` wird schließlich die Multiplikation durchgeführt. Dann der Applikationsknoten überschrieben mit einer Indirektion auf das Ergebnis der Multiplikation `UPDATE 1`. Schließlich wird das Argument vom Stack entfernt `POP 1`.

Mit `UNWIND` lassen wir jede Funktion enden. Damit wird sichergestellt, daß die Maschine eventuell vom Dump alte Zustände reaktiviert.

Code für double Der Code für die Funktion `double` ist bis auf die Multiplikation mit dem der Funktion `square` identisch.

```
[PUSH 0
,EVAL
,PUSH 1
,EVAL
,ADD
```



```
,UPDATE 1
,POP 1
,UNWIND]
```

Code für main Für die Funktion `main` wird erst die Applikation der Funktion `double` auf die `2` und schließlich die Applikation der Funktion `square` hierauf erzeugt. `UNWIND` stößt in diesen Fall die Auswertung an.

```
[PUSHINT 2
,PUSHGLOBAL double
,MKAP
,PUSHGLOBAL square
,MKAP
,UNWIND]
```

Code zum Start der Maschine Die Maschine startet damit, die Funktion `main` auf den Stack zu legen und diese auszuwerten. Anschließend endet die Maschine.

```
[PushGlobal main
,Eval
,End]
```

Wir initialisieren die Maschinen mit einem Heap, auf dem die globalen Funktionsknoten abgelegt sind. Den Programmzähler setzen wir auf die Befehl zum Starte der Maschine durch laden der Funktion `main`:

Stack: **Heap:** **Dump:**

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	
4	
5	
6	
7	

pc=

pc = 22
`PushGlobal main`

Die Adresse der Funktion `main` wird auf den Stack gelegt:

Stack: **Heap:** **Dump:**

2

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	
4	
5	
6	
7	

pc=

pc = 23
`Eval`

Die Auswertung der Funktion `main` wird angestoßen. Hierbei wird der aktuelle Programmzähler auf den Dump abgelegt[9]:

Stack:

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	
4	
5	
6	
7	

Dump: pc=24

pc = 16
`Pushint 2`

Ein Knoten mit der Zahl 2 wird im dem Heap angelegt und dessen Adresse auf dem Stack abgelegt:

Stack: 3

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	
5	
6	
7	

Dump: pc=24

pc = 17
`PushGlobal double`

Der globale Knoten der Funktion `double` wird auf den Stack gelegt:

Stack: 1, 3

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	
5	
6	
7	

Dump: pc=24

pc = 18
`MkAp`

Ein neuer Applikationsknoten wird aus den oberen beiden Stackadressen erzeugt:

Stack: 4

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	NAp 1 3
5	
6	
7	

Dump: pc=24

pc = 19
`PushGlobal square`

[9]Eigentlich wird noch ein `Unwind` durchgeführt, das aber bei nullstelligen Funktionen keinen Effekt hat.

Der Funktionsknoten der globalen Funktion `square` wird auf den Stack gelegt:

Stack: 0, 4

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	NAp 1 3
5	
6	
7	

Dump: pc=24

pc = 20
`MkAp`

Ein weiterer Applikationsknoten wird erzeugt:

Stack: 5

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	NAp 1 3
5	NAp 0 4
6	
7	

Dump: pc=24

pc = 21
`Unwind`

`Unwind` auf einen Applikationsknoten:

Stack: 0, 5

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	NAp 1 3
5	NAp 0 4
6	
7	

Dump: pc=24

pc = 21
`Unwind`

`Unwind` auf einen Funktionsknoten. Der Programmzähler wird auf den Startcode der Funktion gesetzt:

Stack: 4, 5

Heap:

0	NGlobal 1 0 (square)
1	NGlobal 1 8 (double)
2	NGlobal 0 16 (main)
3	NNum 2
4	NAp 1 3
5	NAp 0 4
6	
7	

Dump: pc=24

pc = 0
`Push 0`

Das oberste Stackelement wird nochmals auf den Stack gelegt:

Stack: 4 | 4 | 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [ ] pc=24

pc = 1
`Eval`

Der aktuelle Kontext wird auf den Dump gelegt:

Stack: 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4 | 4 | 5] pc=2; [ ] pc=24

pc = 1
`Unwind`

`Unwind` eines Applikationsknotens:

Stack: 1 | 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4 | 5] pc=2; [ ] pc=24

pc = 1
`Unwind`

`Unwind` eines globalen Funktionsknotens:

Stack: 3 | 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	
8	
9	

Dump: [4 | 5] pc=2; [ ] pc=24

pc = 8
`Push 0`

Das oberste Stackelement wird nochmals auf den Stack gelegt:

Stack: 3 | 3 | 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4 | 5] pc=2; [ ] pc=24

pc = 9
`Eval`

Aufgabe 18 Führen Sie die manuelle Ausführung des Programm `square` in der G-Maschine fort. Zeichnen Sie für jeden Ausführungsschritt Heap und Stack der Maschine.

Lösung

Zunächst wurde die Auswertung mit `Eval` wieder angestoßen. Ein neuer Kontext wird auf dem Dump Zwischengespeichert:

Stack: 3

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [3 | 4] pc=9; [4 | 5] pc=2; [ ] pc=24

pc = 9
`Unwind`

Unwind auf einen Datenknoten bewirkt die Reaktivierung des oberen Dumpkontexts.

Stack: 3 | 3 | 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4 | 5] pc=2; [ ] pc=24

pc = 10
`Push 1`

Das Argument wird erneut auf den Stack gelegt:

Stack: 3, 3, 3, 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 11 `Eval`

Der Befehl `Eval` bewirkt das Speichern des aktuellen Kontext auf den Dump:

Stack: 3

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [3, 3, 4] pc=12; [4, 5] pc=2; [ ] pc=24

pc = 11 `Unwind`

Das `Unwind` auf einen Datenknoten bewirkt das Reaktivieren des obersten Dumpkontextes.

Stack: 3, 3, 3, 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 12 `Add`

Der Befehl `Add` bewirkt die Erzeugung eines neuen Heapknotens mit dem Ergebnis der Addition.

Stack: 6, 3, 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NAp 1 3`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 13 `Update 1`

Das `Update` ersetzt im Heap einen Applikationsknoten durch eine Indirektion:

Stack: 3, 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 14 `Pop 1`

Ein Stackelement wird entfernt.

Stack: 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 15 `Unwind`

`Unwind` auf einer Indriktion stößt ein `Unwind` auf den referenzierten Knoten an.

Stack: 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [4, 5] pc=2; [ ] pc=24

pc = 15 `Unwind`

`Unwind` auf Datenknoten führt zur Reaktivierung eines Dumpkontextes. Wir sind wieder in der Methode `square`.

Stack: 4, 4, 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [ ] pc=24

pc = 2 `Push 1`

Das Argument wird nochmals auf den Stack gelegt.

Stack: 4, 4, 4, 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [ ] pc=24

pc = 3
`Eval`

`Eval` führt dazu den Stack im Dump abzulegen.

Stack: 4

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump:
[4, 4, 5] pc=4
[ ] pc=24

pc = 4
`Unwind`

`Unwind` auf einer Indirektion stößt ein `Unwind` auf den referenzierten Knoten an.

Stack: 6

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump:
[4, 4, 5] pc=4
[ ] pc=24

pc = 4
`Unwind`

`Unwind` auf Datenknoten führt zur Reaktivierung eines Dumpkontextes.

Stack: 6, 4, 4, 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	

Dump: [ ] pc=24

pc = 4
`Mult`

Ein neuer Heapknoten mit dem Ergebnis der Multiplikation wird erzeugt.

Stack: 7, 4, 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NAp 0 4`
6	`NNum 4`
7	`NNum 16`
8	

Dump: [ ] pc=24

pc = 5
`Update 1`

Ein Applikationsknoten wird mit einen Indirektionsknoten überschrieben.

Stack: 4, 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NInd 7`
6	`NNum 4`
7	`NNum 16`

Dump: [ ] pc=24

pc = 6
`Pop 1`

Unötige Knoten werden vom Stack entfernt.

Stack: 5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NInd 7`
6	`NNum 4`
7	`NNum 16`

Dump: [ ] pc=24

pc = 7
`Unwind`

Unwind auf Indirektionsknoten:

Stack: 7

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NInd 7`
6	`NNum 4`
7	`NNum 16`

Dump: [ ] pc=24

pc = 7
`Unwind`

Reaktivierung von Kontext auf dem Stack.

Stack:

5

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NInd 7`
6	`NNum 4`
7	`NNum 16`

Dump:

pc=

pc = 24
`End`

Maschine terminiert:

Stack:

7

Heap:

0	`NGlobal 1 0 (square)`
1	`NGlobal 1 8 (double)`
2	`NGlobal 0 16 (main)`
3	`NNum 2`
4	`NInd 6`
5	`NInd 7`
6	`NNum 4`
7	`NNum 16`

Dump:

pc=

pc = -1

Aufgabe 19 Schreiben Sie ein kleines Testprogramm, das den Code für das Programm `square` in der G-Maschine ausführt.

Lösung

Die Reihung mit dem Code ist anzulegen. Ein GMstate, in dem die Zeiger auf die Einstiegsadressen für die globalen Funktionen korrekt gesetzt sind. Und schon kann es losgehen. Wir legen zum späteren Gebrauch die Reihung mit dem Code in einer eigenen Klasse als statisches Feld an.

SquareCode.java

```
package name.panitz.gm;
public class SquareCode{
  static Instruction [] code
   = {new Push(0)
     ,new Eval()
     ,new Push( 1)
     ,new Eval()
     ,new Mult()
     ,new Update( 1)
     ,new Pop( 1)
     ,new Unwind()
     ,new Push( 0)
     ,new Eval()
     ,new Push( 1)
     ,new Eval()
     ,new Add()
```



```
     ,new Update( 1)
     ,new Pop( 1)
     ,new Unwind()
     ,new PushInt( 2)
     ,new PushGlobal("double")
     ,new MkAp()
     ,new PushGlobal("square")
     ,new MkAp()
     ,new Unwind()
     ,new PushGlobal("main")
     ,new Eval()
     ,new End()
     };
}
```

RunSquare.java

```
package name.panitz.gm;
public class RunSquare{
   static GmState st = new GmState(SquareCode.code);
   public static void main(String[] _)throws Exception{
     st.globals.put("square",st.alloc(new NGlobal(1,0)));
     st.globals.put("double",st.alloc(new NGlobal(1,8)));
     st.globals.put("main",st.alloc(new NGlobal(0,16)));
     st.pc=22;
     st.eval();
     System.out.println(st);
   }
}
```

Aufgabe 20 Es war ein mühsames Geschäft, die G-Maschine von Hand auszuführen. Schön wäre eine spezielle G-Maschine die Schrittweise alle Zustände, die sie bei der Ausführung durchläuft, loggt.

In objektorientierten Sprachen läßt sich dieses leicht implementieren, indem eine Unterklasse von `GmState` geschrieben wird, die nur die Methode `evalStep` überschreibt, so daß in dieser Methode vor dem Aufruf der gleichen Methode aus der Oberklasse, der Maschinenzustand auf die Konsole ausgegeben wird.

```
{System.out.println(this);super.evalStep();}
```

Schreiben Sie eine solche Unterklasse `GmStateLog` und führen Sie mit dieser den Code aus der letzten Aufgabe aus.

2.6.5 Ausgaben

Die G-Maschine ist bisher noch etwas autistisch. Sie reduziert bis ein Finalzustand erreicht ist. Das oberste Stackelement des Finalzustands stellt dann in der Regel das Ergebnis der Berechnung dar. Wir sehen in diesem Kapitel einen Befehl definieren, mit dem die G-Maschine dazu gebracht wird, das oberste Stackelement als String auf einem Ausgabestrom auszugeben. Für diesen Zweck haben wir in der Klasse `GmState` bereits ein Feld `output` für den Ausgabestrom vorgesehen.

Drei Befehle für die Ausgabe auf dem Strom `output` stellen wir zur Verfügung:

- `Output`: zum Ausdrucken von Strinkonstanten.
- `Print`: zum Drucken des obersten Stackelements.
- `PrintString`: zum Drucken des obersten Stackelements als Stringwert.

Output

Der Befehl `Output` hat ein Stringargument, welches bei Ausführung des Befehls auf dem Ausgabestrom ausgegeben wird.

Output.java
```java
package name.panitz.gm;

public class Output implements Instruction{
  String s;
  Output(String s){this.s=s;}
  public GmState process(GmState st)throws Exception{
    st.output.write(s);
    st.output.flush();
    return st;
  }

  public String toString(){return "OUTPUT "+s;}
}
```

Dieser Befehl ist hauptsächlich als kleiner Hilfsbefehl zu verstehen.

Print

Der Befehl `Print` holt den Knoten des obersten Stackelements und wendet auf ihn den neuen Besucher zum Ausdrucken an.

Wir definieren eine Befehlsklasse für den Befehl `Print`:

Print.java
```java
package name.panitz.gm;

public class Print implements Instruction{
  public GmState process(GmState st)throws Exception{
    Node n = st.heap.get(st.stack.peek(0));
    return n.visit(new PrintNode(st));
  }

  public String toString(){return "PRINT";}
}
```


Die entsprechende Besucherklasse geht davon aus, daß es sich bei den auszudruckenden Knoten bereits um einen ausgewerteteten Datenknoten (`NNum`, `NChar`, `NJavaObject` oder `NConstr`) handelt.

PrintNode.java
```java
package name.panitz.gm;

public class PrintNode implements NodeVisitor<GmState>{
  GmState st;
  public PrintNode(GmState st){this.st=st;}

  public GmState eval(NNum n)throws Exception{
    st.output.write(""+n.n);
    st.output.flush();
    st.stack.pop();

    return st;
  }
  public GmState eval(NChar n)throws Exception{
    st.output.write("'"+n.c+"'");
    st.output.flush();
    st.stack.pop();

    return st;
  }
  public GmState eval(NJavaObject n)throws Exception{
    st.output.write(""+n.o);
    st.stack.pop();

    return st;
  }
```

Konstruktoren sollen derart ausgegeben werden, daß in Klammer eingeschlossen, dem Konstruktornamen die Argumente folgen.

PrintNode.java
```java
  public GmState eval(NConstr constr)throws Exception{
    st.output.write("("+st.constructors[constr.name]);
    st.output.flush();
    st.stack.pop();
```

Damit sind öffnende Klammer und Konstruktornamen ausgegeben. Jetzt sind die Argumente des Konstruktors auszugeben. Wir iterieren über diese:

PrintNode.java
```java
    for (Integer a:constr.args) {
```

Der Befehl `Print` geht nicht davon aus, daß die Argumente von dem Konstruktorknoten schon ausgewertet vorliegen. Daher müüsen wir, bevor wir diese rekursiv drucken können, zunächst zur Auswertung bringen. Dazu lege wir diese auf den Stack, stoßen mit der Methode `whnf` die Auswertung an, um dann diese auf den Ausgabestrom auszugeben:

PrintNode.java
```
      st.stack.push(a);
      WHNF.whnf(st);
      st=new Output(" ").process(st);
      st=new Print().process(st);
    }
```

Nach den Argumenten bleibt nur noch die schließende Klammer auszugeben.

PrintNode.java
```
      st=new Output(")").process(st);
    st.output.flush();
    return st;
  }
```

Für die übrigen Knotenfälle verhält sich der Besucher uninteressant:

PrintNode.java
```
  public GmState eval(NInd n) throws Exception{
    st.stack.push(n.adr);
    return st.heap.get(n.adr).visit(this);
  }

  public GmState eval(NAp n)throws Exception{
    st.stack.pop();
    st.output.flush();
    return st;
  }

  public GmState eval(NGlobal n)throws Exception{
    st.stack.pop();
    st.output.flush();
    return st;
  }

  public GmState eval(Node n)throws Exception{
    throw new Exception("unmatched pattern: "+n);
  }
}
```

PrintString

In der G-Maschine kennen wir keinen eigenen Datentypen für Strings. Strings werden durch eine verkettete Liste von primitiven Char-Knoten dargestellt. Würden wir für den String `hello` den obigen Befehl `Print` zur Ausgabe nutzen, so erhielten wir:

```
(Cons 'h' (Cons 'e' (Cons 'l' (Cons 'l' (Cons 'o' (Nil))))))
```


Um in diesen Fällen die Listenkonstruktoren und Klammerungen nicht auszugeben, wird ein zweiter Ausgabebefehl `PrintString` definiert, der nur die primitiven Werte ausgibt. Die Implementierung läuft analog.

PrintString.java
```
package name.panitz.gm;

public class PrintString implements Instruction{
  public GmState process(GmState st)throws Exception{
    Node n = st.heap.get(st.stack.peek(0));
    return n.visit(new PrintStringNode(st));
  }
  public String toString(){return "PRINTSTRING";}
}
```

Der Besucher `PrintStringNode` läßt sich als Unterklasse des Besuchers `PrintNode` definieren. Nur für den Fall von Konstruktorknoten, verhält sie sich anders.

PrintStringNode.java
```
package name.panitz.gm;
public class PrintStringNode extends PrintNode{
  public PrintStringNode(GmState st){super(st);}

  public GmState eval(NConstr constr)throws Exception{
    int[]args=(((NConstr)st.heap.getNode(st.stack.peek(0)))).args;
    int i=0;

    while (i<args.length) {
      st.stack.push(((NConstr)st.heap.getNode(st.stack.peek(0))).args[i]);
      WHNF.whnf(st);
      Node n = st.heap.getNode(st.stack.peek(0));
      if (n instanceof NChar){
        st.output.write(((NChar)n).c+"");
        st.output.flush();
        st.stack.pop();
      }else if(n instanceof NConstr){
        i=0;

        final int tmp=st.stack.pop();
        st.stack.pop();
        st.stack.push(tmp);
        args=((NConstr)n).args;
        continue;
      }
      i=i+1;
    }
    st.stack.pop();
    st.output.flush();
    return st;
  }
```

```

  public GmState eval(NChar n)throws Exception{
    st.output.write(((NChar)n).c+"");
    st.output.flush();
    st.stack.pop();
    return st;
  }
}
```

Testbeispiel

Als kleines Testbeispiel erzeugen wir den String `hallo` im Heap und geben diesen mit den beiden `Print`-Befehlen aus:

TestOutput.java

```
package name.panitz.gm;
import static name.panitz.gm.ConstructorConstants.*;

public class TestOutput{
  static Instruction [] code
      = {new Output("jetzt gehts los Freunde!\n")
        ,new Pack(NIL,0)
        ,new PushChar('o')
        ,new Pack(CONS,2)
        ,new PushChar('l')
        ,new Pack(CONS,2)
        ,new PushChar('l')
        ,new Pack(CONS,2)
        ,new PushChar('e')
        ,new Pack(CONS,2)
        ,new PushChar('h')
        ,new Pack(CONS,2)
        ,new Push(0)
        ,new Output("\nAusgabe als Konstruktorknoten\n")
        ,new Print()
        ,new Output("\nAusgabe als Stringwert\n")
        ,new PrintString()
        ,new Output("\nund tschüß!\n")
        ,new End()
        };

  public static void main(String [] _)throws Exception{
    String [] constrs= {TRUE_S,FALSE_S,NIL_S,CONS_S,PAIR_S};
    new GmState(code,new GmHeap(),new GmGlobals(),constrs).eval();
  }
}
```


2.6.6 Verzweigungen und Sprünge

Wir sind bei der letzten Gruppe von G-Maschinenbefehlen angelangt. Den bedingten und unbedingten Sprüngen.

Unbedingte Sprünge

Für unbedingte Sprünge sehen wir in der G-Maschine den Befehl `Jump` vor. Er hat eine Zahl als Argument, die angibt, um wieviel der Befehlszähler bei Ausführung des Befehls erhöht werden soll.

Aufgabe 21 Implementieren Sie die Klasse `Jump`.

Bedingte Sprünge

Der bedingte Sprungbefehl `CaseJump` ist wesentlich komplexer. Er geht davon aus, daß der von der obersten Stackadresse adressierte Knoten ein Konstruktorknoten ist. Der Befehl `CaseJump` ist eine Abbildung von Konstruktornummern auf Sprungentfernungen. Wenn für den auf den Stack adressierten Konstruktorknoten in dieser Abbildung ein Eintrag existiert, so wird entsprechend dieses Eintrags der Befehlszähler erhöht. Ansonsten bricht die Ausführung mit einen Fehler ab.

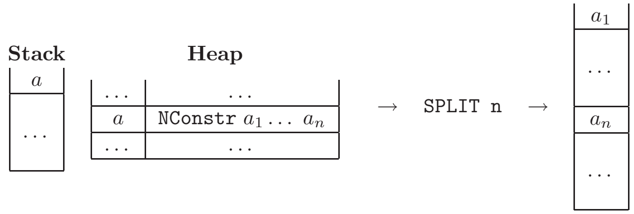
Da der Befehl `CaseJump` eine Abbildung darstellt, empfiehlt es sich bei der Implementierung auf eine der Abbildungsklasse des Javapakets `java.util` zurückzugreifen. Wir leiten die Klasse `CaseJump` von der Klasse `java.util.HashMap` ab. Damit können mit der geerbten Methode `put` dem Befehl Sprungweiten für bestimmte Konstruktornummern hinzufügen.

CaseJump.java

```
package name.panitz.gm;
import java.util.*;

public class CaseJump extends HashMap<Integer,Integer>
                                    implements Instruction {
  public GmState process(GmState st)throws Exception{
    NConstr c = (NConstr)st.heap.getNode(st.stack.peek(0));
    Integer jump = get(c.name);
    if (null==jump) //kein Eintrag in Sprungtabelle
      throw new Exception
               ("unmatched pattern: "+st.constructors[c.name]
               +" in function: "+st.getCurrentFunction()+" ");
    st.pc=st.pc+jump;
    return st;
  }
  public String toString(){return "CASEJUMP "+super.toString();}
}
```

Zugriff auf Konstruktorargumente

Der letzte Befehl, den wir kennenlernen, sorgt dafür, daß für einen Konstruktorknoten die Adressen der Argumente auf den Stack gelegt werden. Er geht davon aus, daß der oberste Knoten auf dem Stack auf einen ausgewerteten Konstruktorknoten zeigt. Dessen Adressen wird vom Stack entfernt, dafür die Argumente des Konstruktors, wie sie auf dem Heap zu finden, auf dem Stack abgelegt.



Wir definieren den Befehl `Split`:

Split.java
```java
package name.panitz.gm;

public class Split implements Instruction{
  int argc;
  public Split(int c){argc=c;}
  public GmState process(GmState st){
    int[] args =((NConstr)st.heap.getNode(st.stack.pop())).args;
    for (int i=args.length-1;i>=0;i--) st.stack.push(args[i]);
    return st;
  }
  public String toString(){return "SPLIT "+argc;}
}
```

Mit den bedingten Sprungbefehl sind wir nun in der Lage alle berechenbaren Funktionen in der G-Maschine zu programmieren.

Beispiel: Fakultät

Betrachten wir als erstes Beispiel die Fakultätsfunktion. In Fab4 wird sie folgende Definition haben.

Fak.fab4
```
constr True 0;
constr False 0;

main = f 3;

f x = case (x=0) of
  True -> 1;
  False-> x * (f(x-1))
```


zunächst schreiben wir von Hand G-Code für die zwei Funktionen.

Code für main In der Hauptfunktion legen wir das Argument zum Funktionsaufruf der Funktion `f` auf den Stack. Dann die globale Funktion `f`. Ein Applikationen wird erzeugt und mit dem Befehl `Unwind` dessen Auswertung angestoßen.

Fak.gmc
```
[PUSHINT 3
,PUSHGLOBAL f
,MKAP
,UNWIND]
```

Code für f Im Code der rekursiven Funktion `f` werden zunächst die Argumente für den Test des Arguments auf 0 auf den Stack gelegt. Der Vergleich wird ausgeführt. Abhängig von dessen Ergebnis wird zu zwei verschiedenen Adressen gesprungen. Im Falle, daß das Argument `0` ist, wird die `1` als Ergebnis auf den Stack gelegt. Andernfalls werden die Argumente der Subtraktion auf den Stack gelegt. Diese wird gleich ausgeführt. Für das Ergebnis ein Applikationsknoten mit `f` erzeugt. Dieses wird schließlich mit dem Argument multipliziert.

Fak.gmc
```
[PUSHINT 0
,PUSH 1
,EVAL
,EQ
,CASEJUMP {1=4, 0=0}
,SPLIT 0
,PUSHINT 1
,SLIDE 0
,JUMP 12
,SPLIT 0
,PUSHINT 1
,PUSH 1
,EVAL
,SUB
,PUSHGLOBAL f
,MKAP
,EVAL
,PUSH 1
,EVAL
,MULT
,SLIDE 0
,UPDATE 1
,POP 1
,UNWIND]
```

Code zum Starten der Maschine Die Maschine starten wir, indem die Funktion `main` ausgewertet wird und das Ergebnis ausgegeben wird.

Fak.gmc
```
[PUSHGLOBAL main
,EVAL
,PRINT
,OUTPUT '\n'
,END]
```

Implementierung Für den obigen Code können wir direkt eine kleine Testklassen schreiben.

RunFak.java
```
package name.panitz.gm;
public class RunFak{
  static CaseJump jump=new CaseJump();

  static Instruction [] code
   ={new PushInt( 3)
    ,new PushGlobal( "f")
    ,new MkAp()
    ,new Unwind()
    ,new PushInt( 0)
    ,new Push(1)
    ,new Eval()
    ,new Eq()
    ,jump
    ,new Split(0)
    ,new PushInt( 1)
    ,new Slide( 0)
    ,new Jump( 12)
    ,new Split(0)
    ,new PushInt( 1)
    ,new Push(1)
    ,new Eval()
    ,new Sub()
    ,new PushGlobal( "f")
    ,new MkAp()
    ,new Eval()
    ,new Push(1)
    ,new Eval()
    ,new Mult()
    ,new Slide( 0)
    ,new Update( 1)
    ,new Pop( 1)
    ,new Unwind()
    ,new PushGlobal("main")
    ,new Eval()
    ,new Print()
    ,new Output("\n")
    ,new End()};

```



```
  static GmState st = new GmStateLog(code);

  public static void main(String[] _)throws Exception{
    jump.put(1,4);
    jump.put(0,0);
    st.globals.put("main",st.alloc(new NGlobal(0,0)));
    st.globals.put("f",st.alloc(new NGlobal(1,4)));
    st.pc=28;
    st.eval();
  }
}
```

Ein vollständiger Log des Ablaufs dieses G-Maschinenprogramms findet sich in Anhang A.

2.7 Bytecode

Mit Ausnahme der für die externen Aufrufe von Javamethoden aus der G-Maschine benötigten Befehle `JavaCall`, `StaticJavaCall` und `GetRuntime` haben wir alle Befehle spezifiziert und implementiert. Ein G-Maschinenprogramm soll als Bytecode abgespeichert und geladen werden. Hierzu definieren wir jetzt ein binäres Format zur Speicherung von G-Maschinenprogrammen. Dabei wird zunächst jedem der 32 Befehle eine Zahl zugeordnet. Wir fassen diese Befehle als Konstantendeklaration zusammen.

CodeConstants.java
```
package name.panitz.gm;
public class CodeConstants {
  static final byte NOP          = 0;
  static final byte POP          = 1;
  static final byte END          = 2;
  static final byte JUMP         = 3;
  static final byte PUSH         = 4;
  static final byte UNWIND       = 5;
  static final byte EVAL         = 6;
  static final byte PRINT        = 7;
  static final byte PRINTSTRING  = 9;
  static final byte OUTPUT       = 10;
  static final byte MKAP         = 11;
  static final byte JAVACALL     = 12;
  static final byte PUSHGLOBAL   = 13;
  static final byte PUSHINT      = 14;
  static final byte PUSHCHAR     = 15;
  static final byte SLIDE        = 16;
  static final byte UPDATE       = 17;
  static final byte PACK         = 18;
  static final byte CASEJUMP     = 19;
  static final byte SPLIT        = 20;
  static final byte ADD          = 21;
  static final byte SUB          = 22;
```

```
  static final byte MULT          = 23;
  static final byte DIV           = 24;
  static final byte LE            = 25;
  static final byte GE            = 26;
  static final byte LT            = 27;
  static final byte GT            = 28;
  static final byte EQ            = 29;
  static final byte STATICJAVACALL= 30;
  static final byte GETRUNTIME    = 31;

  static final byte FA            = (byte)250;
  static final byte B4            = (byte)180;
}
```

Zusätzlich sind noch zwei Bytewerte definiert, um Anfang und Ende einer G-Maschinenen-Bytecode-Datei zu markieren. Hier haben wir die Werte 250 und 180 gewählt, die zusammen in hexadezimaler Darstellung den Wert `FAB4` ergeben.[10]

2.7.1 Byte Code Format

Wir kodieren ein G-Maschinenprogramm binär. Zwischenstartmarkierung und Endmarkierung, finden sich die Funktionsdefinitionen, Konstruktornamen und schließlich der Code des Programms.

Die Bytecode-Dateien `.gmc` der G-Maschine haben das in Abbildung 2.5 angegebene Format.

Eine entsprechende binäre Datei im Hex-Mode des Emacs erscheint wie folgt:

```
00000000: fab4 0000 0007 0047 004d 0020 0043 006f  .......G.M. .C.o
00000010: 0064 0065 0000 0005 0000 000e 0073 0074  .d.e.........s.t
00000020: 0061 0074 0069 0063 004a 0061 0076 0061  .a.t.i.c.J.a.v.a
00000030: 0043 0061 006c 006c 0000 0000 0000 0002  .C.a.l.l........
00000040: 0000 0008 006a 0061 0076 0061 0043 0061  .....j.a.v.a.C.a
00000050: 006c 006c 0000 0006 0000 0003 0000 000a  .l.l............
00000060: 0067 0065 0074 0052 0075 006e 0074 0069  .g.e.t.R.u.n.t.i
00000070: 006d 0065 0000 000e 0000 0000 0000 0004  .m.e............
00000080: 006d 0061 0069 006e 0000 0010 0000 0000  .m.a.i.n........
00000090: 0000 0001 0066 0000 0014 0000 0001 0000  .....f..........
000000a0: 0002 0000 0004 0054 0072 0075 0065 0000  .......T.r.u.e..
000000b0: 0005 0046 0061 006c 0073 0065 0000 0030  ...F.a.l.s.e...0
000000c0: 0400 0000 0106 0400 0000 0106 1e05 0400  ................
000000d0: 0000 0206 0400 0000 0206 0400 0000 0206  ................
000000e0: 0c05 1f05 0e00 0000 030d 0000 0001 0066  ...............f
000000f0: 0b05 0e00 0000 0006 0400 0000 0106 1d06  ................
00000100: 1300 0000 0200 0000 0000 0000 0000 0000  ................
00000110: 0100 0000 0414 0000 0000 0e00 0000 0110  ................
00000120: 0000 0000 0300 0000 0e14 0000 0000 0e00  ................
00000130: 0000 0106 0400 0000 0106 160d 0000 0001  ................
```

[10] Auf ähnliche Weise beginnen Java Klassendateien mit den Hexadezimalwerten `CAFEBABE`.

Startmarkierung		FA B4
Länge des Modulname		n_0 n_1 n_2 n_3
Modulname		$c_{1_1}c_{1_2}$ ... $c_{1_n}c_{1_n}$
Anzahl der Funktionen		n_0n_1 n_2n_3
Funktion$_1$	Länge des Funktionsnamen	n_0 n_1 n_2 n_3
	Funktionsname	$c_{1_1}c_{1_2}$ ... $c_{1_n}c_{1_n}$
	Code Adresse	
	Anzahl der Funktionsargumente	n_0n_1 n_2n_3
...	...	
Funktion$_n$	Länge des Funktionsnamen	n_0 n_1 n_2 n_3
	Funktionsname	$c_{1_1}c_{1_2}$ ... $c_{1_n}c_{1_n}$
	Code Adresse	
	Anzahl der Funktionsargumente	n_0n_1 n_2n_3
Anzahl der Konstruktoren		n_0n_1 n_2n_3
Konstruktor$_1$	Länge des Konstruktornamens	n_0 n_1 n_2 n_3
	Konstruktornamen	$c_{1_1}c_{1_2}$ ... $c_{1_n}c_{1_n}$
...	...	
Konstruktor$_n$	Länge des Konstruktornamens	n_0 n_1 n_2 n_3
	Konstruktornamen	$c_{1_1}c_{1_2}$ ... $c_{1_n}c_{1_n}$
Anzahl der Befehle		n_0n_1 n_2n_3
Befehl$_1$	Befehlsbyte	Befehlsargumente
...	...	
Befehl$_n$	Befehlsbyte	Befehlsargumente
Endemarkierung		FA B4

Abbildung 2.5: Bytecode-Format der G-Maschine

```
00000140: 0066 0b06 0400 0000 0106 1710 0000 0000  .f..............
00000150: 0300 0000 0011 0000 0001 0100 0000 0105  ................
00000160: fab4                                     ..
```

2.7.2 Schreiben des Bytecodes

In diesem Abschnitt wird eine Methode entwickelt, die Bytecodedateien für einen G-Maschinenzustand schreiben. Hierzu wird eine Klasse `WriteGm` geschrieben. Javaklassen für IO werden benötigt und das entsprechende Javapaket importiert:

WriteGm.java
```
package name.panitz.gm;

import java.io.*;
import java.util.*;
import static name.panitz.gm.CodeConstants.*;

public class WriteGm{
```

Zum Schreiben von Bytecode wird eine statische Methode definiert. Sie hat zwei Argumente:

- den binären Ausgabestrom, in den der Bytecode geschrieben werden soll. In der Regel wird es sich um eine Datei handeln.
- Den G-Maschinenzustand, für den Bytecode geschrieben werden soll.

Da bei IO-Operationen jede Menge denkbarer Ausnahmen auftreten können, um die wir uns nicht kümmern wollen, muß dieses im Methodenkopf vermerkt werden.

WriteGm.java
```
  static void writeGMCode(DataOutputStream out, GmState st)
                                    throws Exception{
```

Wir benutzen als Ausgebestrom die Javaklasse `DataOutputStream`. Auf dieser gibt es Methoden `writeInt`, `writeChars` und `write` zum Schreiben von Zahlen, Strings oder Bytes.

Im Bytecodeformat der G-Maschine sind Strings derart kodiert, daß erst die Länge des String geschrieben wird und dann die einzelnen Zeichen. Dieses kapseln wir in einer Hilfsmethode `write`, die weiter unten definiert wird.

Wir schreiben nach und nach die einzelne Teile des Bytecodes:

Startmarkierung Wir schreiben die zwei Byte der Startmarkierung:

WriteGm.java
```
    out.write(FA);out.write(B4);        //FAB4
```

Modulname Anschließend schreiben wir den namen des Moduls.[11] Hierzu benutzen wir eine Methode `write`, die weiter unten definiert wird und in einen Ausgabestrom zunächst die Länge eines zu schreibenen Strings und dann dessen einzelnen Zeichen als Byte codiert schreibt.

WriteGm.java
```
    write(out,st.name);                  //module name
```

Globale Funtionen Zum Schreiben der Funktionen wird erst deren Anzahl geschrieben:

WriteGm.java
```
    out.writeInt(
         st.globals.entrySet().size()); //number of functions
```

Und dann über die Abbildung aller Funktionseinträge iteriert. Für jede einzelne Funktion wird der name geschrieben, dann die Adresse des Codes und schließlich die Argumentanzahl. Diese beiden Informationen sind im Heap gespeichert.

[11] Eine Information, die bisher nirgends weiter in der Implementierung benutzt wird.

WriteGm.java
```
    for (Map.Entry<String,Integer> entry
                   :st.globals.entrySet()){
      write(out,entry.getKey());        //function name
      final NGlobal glob = (NGlobal)st.heap.get(entry.getValue());
      out.writeInt(glob.i);             //code pointer of function
      out.writeInt(glob.argc);          //number of function args
    }
```

Konstruktornamen Auch für die Konstruktoren wird zunächst die Anzahl und dann jeder einzelnen Konstruktorname geschrieben. Die Konstruktornummer ergibt sich aus dessen Position.

WriteGm.java
```
    out.writeInt(st.constructors.length);//number of constructors
    for (String constr:st.constructors)
      write(out,constr);                //constructor name
```

Befehle Endlich schreiben wir die Befehle. Zunächst die Anzahl und dann jeden einzelnen Befehl. Zum schreiben der Befehle nutzen wir eine eigene Methode.

WriteGm.java
```
    out.writeInt(st.code.length);
    for (Instruction i:st.code) writeInstruction(out,i);
```

Endemarkierung Die Bytecode-Datei soll wieder mit der FAB4-Markierung enden:

WriteGm.java
```
    out.write(FA);out.write(B4);        //FAB4
    out.close();
  }
```

Und schließlich konnten wir den Ausgabestrom chließen.

Hilfsmethode für Strings Die erste Hilfsmethode, die wir verwendet haben, schreibt einen String in einen Bytestrom, indem erst die Anzahl der Zeichen und dann alle Zeichen geschrieben werden.

WriteGm.java
```
  static private void write(DataOutputStream out,String s)
                                              throws Exception{
    out.writeInt(s.length());      //length of string
    out.writeChars(s);             //characters of string
  }
```

Schreiben der Befehle Es verbeibt die Methode zum Schreiben der einzelnen Befehle. Wir haben hier keinen objektorientierten Entwurf gewählt. Die Befehlsklassen haben keine Methode, um sich in eine Bytecode-Datei zu schreiben. Deshalb enden wir in einer großen Fallunterscheidung. Der Befehl, für den Bytecode zu schreiben ist, wird per `instanceof` befragt, von welcher Klasse er ist. Entsprechend wird dann seine Bytekonstante geschrieben.

Wir beginnen mit den Befehlen,die keine Argumente haben. Für diese ist lediglich ein Byte zu schreiben:

WriteGm.java
```java
  static void writeInstruction(DataOutputStream out,Instruction i)
                                          throws Exception{
   if (i instanceof End){out.write(END);
   }else if (i instanceof Unwind){out.write(UNWIND);
   }else if (i instanceof Eval){out.write(EVAL);
   }else if (i instanceof Print){out.write(PRINT);
   }else if (i instanceof PrintString){out.write(PRINTSTRING);
   }else if (i instanceof MkAp){out.write(MKAP);
   }else if (i instanceof JavaCall){out.write(JAVACALL);
   }else if (i instanceof StaticJavaCall){out.write(STATICJAVACALL);
   }else if (i instanceof GetRuntime){out.write(GETRUNTIME);
   }else if (i instanceof Add){out.write(ADD);
   }else if (i instanceof Sub){out.write(SUB);
   }else if (i instanceof Mult){out.write(MULT);
   }else if (i instanceof Div){out.write(DIV);
   }else if (i instanceof Lt){out.write(LT);
   }else if (i instanceof Gt){out.write(GT);
   }else if (i instanceof Le){out.write(LE);
   }else if (i instanceof Ge){out.write(GE);
   }else if (i instanceof Eq){out.write(EQ);
```

Die nächsten Befehle haben ganze Zahlen als Argument. Dem Befehlsbyte folgt die ganze Zahl:

WriteGm.java
```java
   }else if (i instanceof Push){
     out.write(PUSH);out.writeInt(((Push)i).n);
   }else if (i instanceof Pop){
     out.write(POP);out.writeInt(((Pop)i).n);
   }else if (i instanceof Jump){
     out.write(JUMP);out.writeInt(((Jump)i).n);
   }else if (i instanceof PushInt){
     out.write(PUSHINT);out.writeInt(((PushInt)i).n);
   }else if (i instanceof Slide){
     out.write(SLIDE);out.writeInt(((Slide)i).n);
   }else if (i instanceof Update){
     out.write(UPDATE);out.writeInt(((Update)i).n);
   }else if (i instanceof Split){
     out.write(SPLIT);out.writeInt(((Split)i).argc);
```

Es verbleiben Befehle mit unterschiedlichen Argumenten:

WriteGm.java
```java
   }else if (i instanceof Output){
     out.write(OUTPUT);
     final String s = ((Output)i).s;
     out.writeInt(s.length());
     out.writeChars(s);
   }else if (i instanceof PushGlobal){out.write(PUSHGLOBAL);
     final String s = ((PushGlobal)i).name;
     out.writeInt(s.length());
     out.writeChars(s);
   }else if (i instanceof PushChar){out.write(PUSHCHAR);
     final char[] c={((PushChar)i).n};
     out.writeChars(new String(c));
   }else if (i instanceof Pack){out.write(PACK);
     final Pack p=(Pack)i;
     out.writeInt(p.name);
     out.writeInt(p.argc);
   }else if (i instanceof CaseJump){out.write(CASEJUMP);
     final CaseJump jumps = (CaseJump)i;
     out.writeInt(jumps.entrySet().size());
     for (Map.Entry<Integer,Integer> entry:jumps.entrySet()){
       out.writeInt(entry.getKey());
       out.writeInt(entry.getValue());
     }
   }
  }
}
```

2.7.3 Lesen des Bytecode

Nachdem wir jetzt Bytecodedateien erzeugen können, was eigentlich von der G-Maschine nicht benötigt wird, denn die Bytecode-Dateien soll ja schließlich der Compiler erzeugen, müssen wir uns um den inversen Vorgang kümmern: dem Lesen einer Bytecode-Datei. Entsprechend des Vorgehens im letzten Abschnitt implementieren wir genau die inversen Operationen.

Zunächst einmal wieder die Hilfsmethode zum Lesen eines Strings aus dem Bytecode:

ReadGm.java
```java
package name.panitz.gm;

import java.io.*;
import java.util.*;
import static name.panitz.gm.CodeConstants.*;

public class ReadGm{
  static String readString(DataInputStream in)throws Exception{
    final int size = in.readInt();
    String result="";
    for (int i=0;i<size;i++)result=result+in.readChar();
```

```
    return result;
  }
```

Das Lesen einer Bytecode-Datei erzeugt einen Maschinenzustand. Wir haben mitlerweile schon zwei verschiedene Maschinen implementiert. `GmState` als die Standardmaschine und die spezialisierte Klasse `GmStateLog`. Daher werden wir zwei Methoden **`readGMCode`** schreiben: eine zum Instanziieren der Maschine mit `GmState` und eine zum Instanziieren von `GmStateLog`.

ReadGm.java

```
  static GmState readGMCode(DataInputStream in) throws Exception{
    return readGMCode(in,false);}

  static GmState readGMCodeLog(DataInputStream in) throws Exception{
    return readGMCode(in,true);}
```

Jetzt läßt sich nach und nach eine Bytecode-Datei einlesen. Wenn die Bytes nicht dem spezifizierten Format entsprechen, wird eine Ausnahme geworfen.

ReadGm.java

```
  static GmState readGMCode(DataInputStream in,boolean log)
                                                throws Exception{
    final byte[]fab4= new byte[2];
    in.read(fab4);
    if (fab4[0]!=FA||fab4[1]!=B4) throw new Exception("no gm code");
```

Nach den Startbytes wird der Name des Modules gelesen:

ReadGm.java

```
    final String name =readString(in);
```

Die ersten Komponenten der Maschine werden erzeugt:

ReadGm.java

```
    final GmHeap heap=new GmHeap();
    final GmGlobals glob=new GmGlobals();
```

Die Funktionen können eingelesen werden und für sie Knoten im Heap erzeugt werden. Der Funktionsname wird auf die entsprechende Heapadresse abgebildet:

ReadGm.java

```
    final int fs=in.readInt();
    for (int i=0;i<fs;i++){
      final String n = readString(in);
      final int pc   = in.readInt();
      final int argc = in.readInt();
      int adr = heap.alloc(new NGlobal(argc,pc));
      glob.put(n,adr);
    }
```


Es folgen die Konstruktoren. Nachdem ihre Anzahl bekannt ist, können sie in der entsprechenden Reihung eingetragen werden:

ReadGm.java

```
    final int cs=in.readInt();
    String [] constructors = new String[cs];
    for (int i=0;i<cs;i++){
      constructors[i]=readString(in);
    }
```

Und zu guter Letzt ist der Code zu lesen. Den Code zum Starten der Maschine erwarten wir nicht in der Bytecode-datei, sondern fügen die fünf notwendigen Befehle hier ein:

ReadGm.java

```
    final int codeLength=in.readInt()+5;

    Instruction [] code=new Instruction[codeLength];
    for (int i=0;i<codeLength-5;i++)code[i]=readInstruction(in);
    code[codeLength-5]=new PushGlobal("main");
    code[codeLength-4]=new Eval();
    code[codeLength-3]=new Print();
    code[codeLength-2]=new Output("\n");
    code[codeLength-1]=new End();
```

Alle Komponenten der Maschine sind gelesen. Je nachdem ob Logging gewünscht ist oder nicht wird eine maschine instanziiert.

ReadGm.java

```
    GmState result ;
    if (log) result = new GmStateLog(code,heap,glob,constructors);
    else result = new GmState(code,heap,glob,constructors);

    result.pc=codeLength-5;
    result.name=name;
    return result;
  }
```

Es ist schließlich noch die Methode zum Einlesen der Befehle notwendig.

ReadGm.java

```
  static Instruction readInstruction(DataInputStream in)
                                                throws Exception{
```

Wir müssen den Befehlsbyte lesen und abhängig von diesem einen Befehl zurückgeben:

ReadGm.java

```
    final byte b=in.readByte();
    switch (b){
      case NOP:        return new Nop();
```

Aufgabe 22 Jetzt sind Sie dran. Ergänzen Sie die Switschanweisung für alle weiteren Befehle der G-Maschine. In der Klasse `DataInputStream` stehen Ihnen Methoden `ReadInt` und `readChar` zur verfügung. Sollte kein der Switchanweisung zugetroffen haben, liegt wohl ein Fehler in der Bytecode-Datei vor:

ReadGm.java
```
    }
    throw new Exception("Illegal GM Code: "+b);
  }
}
```

2.7.4 Ausführen der G-Maschine

Wir haben zunächst alles beieinander, um eine G-Maschinen-Bytecodedatei zu lesen und auszuführen. Wir fassen dieses in zwei Hauptklassen zum Ausführen einer Bytecode-Datei zusammen. Einmal ohne Logging:

Gm.java
```
package name.panitz.gm;

import java.io.*;

public class Gm{
  public static void main(String[] args)throws Exception{
    runModule(args);
  }

  static public void runModule(String[] args)throws Exception{
    GmState st
     = ReadGm.readGMCode
         (new DataInputStream
             (new FileInputStream(args[0]+".gmc")));
    st.progArgs=args;
    st.eval();
  }
}
```

Und dasselbe mit Logging:

GmLog.java
```
package name.panitz.gm;

import java.io.*;

public class GmLog{
  public static void main(String[] args)throws Exception{
    runModule(args);
  }

  static public void runModule(String[] args)throws Exception{
```



```
    GmState st
     = ReadGm.readGMCode
         (new DataInputStream
             (new FileInputStream(args[0]+".gmc")),true);
    st.progArgs=args;
    st.eval();
  }
}
```

Wenn alle Aufgaben bisher korrekt gelöst wurden, steht eine funktionierende G-Maschine zur Verfügung. Allerdings wird relativ schnell der Heap dieser Maschine bei der Ausführung des Codes an seine Kapazitätsgrenzen stoßen.

Die Maschine sollte in der Lage sein kleine Programme auszuführen. Hierzu stehen G-Maschinen Bytecodedateien (`http://www.panitz.name/student/gmc/index.html`) im Netz zur Verfügung.

2.8 Garbage Collection

Cleanup time.

Lennon

Die G-Maschine in ihrer Art hat eine unangenehme Eigenart. Sie kennt nur Befehle, um im Heap neue Speicherplätze anzufordern und zu belegen. Es gibt keinerlei Befehle, um nicht mehr benötigten Speicherplatz wieder freizugeben. Die G-Maschine ist in dieser Hinsicht recht unordentlich.[12]

In der Komponente `GmState` haben wir schon eine Müllentsorgung vorgesehen, von der wir auch stets die Methode `check` aufgerufen haben, bevor wir neuen Heapspeicher allokiert haben. Hier nun die volle Klasse der Müllentsorgung. Sie ist als abstrakte Klasse realisiert, in der es eine abstrakte Methode gibt, die checkt ob noch eine bestimmte Anzahl von Heapzellen frei ist. Wenn nicht, soll der Aufruf von `check` dafür sorgen, daß die im Heap benötigte Anzahl von Speicherzellen freigegeben wird. Die Spezialversion der Methode `check` ohne Parameter, setzt den Parameter auf den Standardwert 1.

GC.java
```
package name.panitz.gm;

public abstract class  GC {
  public abstract  void check(int i);
  public void check(){check(1);}
}
```

Solange wir noch keine wirkliche Müllentsorgung geschrieben haben, benutzen wir eine Klasse, die nichts macht, beim Aufruf der Methode `check`.

FakeGC.java
```
package name.panitz.gm;
public class FakeGC extends GC{
  public void check(int i){};
}
```

[12]Und spricht damit vielleicht einigen von uns aus dem Herzen: lazy (faul) und räumt selbst nicht auf.

Wir wollen etwas flexibel bleiben. Welche Klasse als Müllentsorgung instanziiert wird. Zunächst schauen wir, ob eine Klasse als Property für die Müllentsorgung gesetzt ist, und versuchen diese zu instanziieren. Ansonsten versuchen wir es mit einer speziellen Klasse, der `TwoSpaceCopyGC`, die in der nächsten Aufgabe implementiert werden soll, und wenn alles schief geht nehmen wir die `FakeGC`.

GCFactory.java

```java
package name.panitz.gm;
public class GCFactory {

  public static String STANDARD_GC_CLASS
   = "name.panitz.gm.TwoSpaceCopyGC";

  public static GC getGCInstance(GmState st){
    try{
      String gmClassName = System.getProperty("g-machine.gc.class");
      gmClassName  =   null==gmClassName
                     ? STANDARD_GC_CLASS
                     : gmClassName;
      return (GC)Class.forName(gmClassName)
                      .getDeclaredConstructor(st.getClass())
                      .newInstance(st);
    }catch(Exception _){return new FakeGC();}
  }
}
```

Dieses Vorgehen ermöglicht es uns, sogar noch dynamisch von außen zu bestimmen, mit welcher Müllentsorgung unsere Maschine laufen soll. Beim Starten des Javainterpreters kann mit der Option `-D` eine System-Property gesetzt werden. Soll z.B. die Klasse `FakeGC` als garbage collection benutzt werden, so rufen Sie die G-maschine wie folgt auf:

```
sep@pc:~/> java -Dg-machine.gc.class=name.panitz.gm.FakeGC name.panitz.gm.Gm TestGM
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 909
        at name.panitz.gm.GmHeap.alloc(GmHeap.java:12)
        at name.panitz.gm.GmState.alloc(GmState.java:23)
        at name.panitz.gm.Pack.process(Pack.java:16)
        at name.panitz.gm.GmState.evalStep(GmState.java:30)
        at name.panitz.gm.GmState.eval(GmState.java:34)
        at name.panitz.gm.Gm.runModule(Gm.java:16)
        at name.panitz.gm.Gm.main(Gm.java:7)
sep@pc216-5:~/fh/compiler/tutor>
```

Da die `FakeGC` keine wirkliche garbage collection durchführt bricht die Maschine wegen mangelnden Speicherplatz sehr schnell ab.

Ist hingegen eine echte garbage collection vorhanden, kann sie ihr Ergebnis berechnen:

```
sep@pc:~/> java  name.panitz.gm.Gm TestGM
42
sep@pc216-5:~/fh/compiler/tutor>
```


2.8.1 Two-Space-Copy-Müllentsorgung

Ein gängiges Verfahren zur Implementierung einer Müllentsorgung, wird als *two space copy* bezeichnet[FY69]. Die Idee ist in etwa, nicht seinen Müll zu sammeln und dann wegzuschmeißen, sondern eher wie bei einem Umzug vorzugehen. Man hat eine schöne neue, meistens auch größere Wohnung als die alte. Alles was man in der alten Wohnung hat und noch benötigt, transferriert man in die neue Wohnung. In der alten Wohnung liegt dann aller möglicher Krempel herum, den man nicht mehr benötigt. Das kann dann geschlossen weggeworfen werden.

Sozusagen als neue Wohnung wird beim *two space copy* ein zweiter frischer Heap instanziiert. Dieser hat in der Regel auch mehr Speicherzellen als der alte, muß aber nicht. In diesen werden alle Heapknoten aus dem alten Heap kopiert, die noch gebraucht werden.

Wie kann man feststellen, welche Knoten noch gebraucht werden? Dreh und Angelpunkt der G-Maschine ist der Stack. Hier befinden sich Heapadressen und dieses sind auch noch genau die Heapadressen, mit denen wir noch arbeiten wollen. Wir brauchen also schon einmal alle die Heapzellen, die auf dem Stack angesprochen sind. Zusätzlich können in diesen Heapadressen Knoten stehen, die auf andere Heapknoten verweisen; zum einen durch Indirektionsknoten, zum anderen in Applikationsknoten und Konstruktorknoten. Wir müssen also den Verweisen von gebrauchten Knoten folgen. Was wir benötigen ist ein transitiver Abschluss. So lassen sich alle Heapknoten ermitteln, die noch benötigt werden.

Es reicht nicht aus die benötigten Heapknoten in den neuen Heap zu kopieren. Das wäre so, wie wenn wir beim Umzug alles irgendwie ungeordnet in die neue Wohnung stellten. Die Dinge werden sich im neuen Heap an anderer Stelle finden als im alten. Während des Umzugs in die neue Wohnung ist daher Sorge zu tragen, daß jeweils im alten Heap vermerkt ist, ob sich ein Heapknoten schon im neuen Heap befindet. Wenn der Knoten kopiert wurde, heißt das, das im alten Heap vermerkt ist, welche Adresse dieser Knoten nun im neuen Heap bekommen hat. Hierzu sehen wir einen neuen Heapnotentyp vor. Er wird nur während der Müllentsorgung benutzt. Daher soll er auch nicht im Besucher für Heapknoten auftreten. Die Besucher werden für ihn Ausnahmen werfen.

Wie definieren die Klasse `ForwardNode`. Sie kennzeichnet, daß das gesuchte Objekt schon in einen neuen Heap umgezogen ist, und gibt die entsprechende neue Adresse an. Sie ist so etwas wie ein Nachsendeauftrag.

ForwardNode.java

```java
package name.panitz.gm;

public class ForwardNode implements Node{
  int adr;
  public ForwardNode(int a){adr=a;}
  public String toString(){return "(ForwardNode "+adr+")";}
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
}
```

2.8.2 Beispiel

Betrachten wir die garbage collection am Beispiel. Abbildung 2.6 zeigt einen Maschinenzustand während der Ausführung des Programms zur Primzahlengenerierung nach Ertosthenes.

Stack:

34
36
34
38

Heap:

0	(NGlobal 2 0)
1	(NGlobal 3 6)
2	(NGlobal 0 14)
3	(NGlobal 1 16)
4	(NGlobal 1 30)
5	(NGlobal 2 42)
6	(NGlobal 2 78)
7	(NGlobal 1 100)
8	(NGlobal 0 120)
9	2
10	NInd -> 15
11	NInd -> 20
12	1
13	3
14	NInd -> 24
15	(Constr 3 [9 14])
16	NAp(6 9)
17	NAp(5 16)
18	NInd -> 35
19	NInd -> 40
20	(Constr 3 [9 19])
21	1
22	4
23	NAp(4 22)
24	(Constr 3 [13 23])
25	NInd -> 31
26	0
27	1
28	2
29	1
30	(Constr 1 [])
31	NInd -> 32
32	(Constr 0 [])
33	NAp(5 16)
34	NAp(33 23)
35	(Constr 3 [13 34])
36	NAp(6 13)
37	NAp(5 36)
38	NAp(37 34)
39	NAp(7 38)
40	(Constr 3 [13 39])

Dump:

39 38	pc=102
	pc=-1

pc = 43
EVAL

Abbildung 2.6: Heap während der Ausführung der Primzahlengenerierung

Kopieren von globalen Funktionsknoten Stack und Dump

Im ersten Schritt der Garbage Collection sind die globalen Knoten (im `GMGlobals`-Objekt aufgelisteten Knoten) zu kopieren, sowie die auf dem Stack und den in auf den gedumpten Stacks referenzierten Knoten in einen neuen Heap zu kopieren. Im alten Heap ist zu vermerken, daß diese Knoten kopiert wurden und auf welcher Adresse sie im neuen Heap stehen. Nach diesem Schritt ergibt sich die Belegung der maschine wie in Abbildung 2.7 dargestellt

Stack:

10
11
10
9

Old Heap:

0	Forward 0
1	Forward 1
2	Forward 2
3	Forward 3
4	Forward 4
5	Forward 5
6	Forward 6
7	Forward 7
8	Forward 8
9	2
10	NInd -> 15
11	NInd -> 20
12	1
13	3
14	NInd -> 24
15	(Constr 3 [9 14])
16	NAp(6 9)
17	NAp(5 16)
18	NInd -> 35
19	NInd -> 40
20	(Constr 3 [9 19])
21	1
22	4
23	NAp(4 22)
24	(Constr 3 [13 23])
25	NInd -> 31
26	0
27	1
28	2
29	1
30	(Constr 1 [])
31	NInd -> 32
32	(Constr 0 [])
33	NAp(5 16)
34	Forward 10
35	(Constr 3 [13 34])
36	Forward 11
37	NAp(5 36)
38	Forward 9
39	Forward 12
40	(Constr 3 [13 39])

New Heap:

0	(NGlobal 2 0)
1	(NGlobal 3 6)
2	(NGlobal 0 14)
3	(NGlobal 1 16)
4	(NGlobal 1 30)
5	(NGlobal 2 42)
6	(NGlobal 2 78)
7	(NGlobal 1 100)
8	(NGlobal 0 120)
9	NAp(37,34)
10	NAp(33 23)
11	NAp(6 13)
12	NAp(7 38)
13	

Dump:

12 9	pc=102
	pc=-1

Abbildung 2.7: Zustand nach Kopie von Stack, Dump und Globals

erste Iteration

Im ersten Schritt haben wir die direkt nootwendigen Knoten kopiert und 13 Knoten in den neuen Heap angelegt. In diesen Knoten des neuen Heaps befinden sich Referenzen auf Zellen im alten Heap. Jetzt sind diese Knoten aus dem alten Heap zu kopieren, oder falls sie schon kopiert wurden, deren neue Adresse zu übernehmen. Folgt man den Referenzen der Knoten in Zellen 0 bis 12 des neuen Heaps, so erhält man die in Abbildung 2.8 dargestellte Situation.

Stack:

10
11
10
9

Old Heap:

0	Forward 0
1	Forward 1
2	Forward 2
3	Forward 3
4	Forward 4
5	Forward 5
6	Forward 6
7	Forward 7
8	Forward 8
9	2
10	NInd -> 15
11	NInd -> 20
12	1
13	Forward 16
14	NInd -> 24
15	(Constr 3 [9 14])
16	NAp(6 9)
17	NAp(5 16)
18	NInd -> 35
19	NInd -> 40
20	(Constr 3 [9 19])
21	1
22	4
23	Forward 15
24	(Constr 3 [13 23])
25	NInd -> 31
26	0
27	1
28	2
29	1
30	(Constr 1 [])
31	NInd -> 32
32	(Constr 0 [])
33	Forward 14
34	Forward 10
35	(Constr 3 [13 34])
36	Forward 11
37	Forward 13
38	Forward 9
39	Forward 12
40	(Constr 3 [13 39])

New Heap:

0	(NGlobal 2 0)
1	(NGlobal 3 6)
2	(NGlobal 0 14)
3	(NGlobal 1 16)
4	(NGlobal 1 30)
5	(NGlobal 2 42)
6	(NGlobal 2 78)
7	(NGlobal 1 100)
8	(NGlobal 0 120)
9	NAp(13,10)
10	NAp(14 15)
11	NAp(6 16)
12	NAp(7 9)
13	NAp(5 36)
14	NAp(5 16)
15	NAp(4 22)
16	3

Dump:

12 9	pc=102
	pc=-1

Abbildung 2.8: Zustand nach Kopieren der in Zellen 0 bis 12 referenziereten Knoten

zweite und dritte Iteration

Die erste Iteration hat wieder neue Knoten im neuen Heap angelegt, sie jetzt wiederum Referenzen in den alten Heap enthalten. Abermals ist doiesen zu folgen. Folgen wir diesen wird wieder ein neuer Knoten im neuen Heap angelegt, der auf Adressen im alten Heap verweist. Abermals ist zu kopieren. Nach zwei weiteren Iterationen erhält man das Bild aus Abbildung 2.9.

Jetzt wurde kein neuer Knoten mit Referenzen auf den alten Heap angelegt. Der alte Heap kann komplett gelöscht werden. Der neue Heap ist weniger als halb so groß, wie der alte. Über die Hälfte der Knoten hat sich als nicht mehr benötigter Müll herausgestellt.

Stack:

10
11
10
9

Old Heap:

0	Forward 0
1	Forward 1
2	Forward 2
3	Forward 3
4	Forward 4
5	Forward 5
6	Forward 6
7	Forward 7
8	Forward 8
9	Forward 19
10	NInd -> 15
11	NInd -> 20
12	1
13	Forward 16
14	NInd -> 24
15	(Constr 3 [9 14])
16	Forward 17
17	NAp(5 16)
18	NInd -> 35
19	NInd -> 40
20	(Constr 3 [9 19])
21	1
22	Forward 18
23	Forward 15
24	(Constr 3 [13 23])
25	NInd -> 31
26	0
27	1
28	2
29	1
30	(Constr 1 [])
31	NInd -> 32
32	(Constr 0 [])
33	Forward 14
34	Forward 10
35	(Constr 3 [13 34])
36	Forward 11
37	Forward 13
38	Forward 9
39	Forward 12
40	(Constr 3 [13 39])

New Heap:

0	(NGlobal 2 0)
1	(NGlobal 3 6)
2	(NGlobal 0 14)
3	(NGlobal 1 16)
4	(NGlobal 1 30)
5	(NGlobal 2 42)
6	(NGlobal 2 78)
7	(NGlobal 1 100)
8	(NGlobal 0 120)
9	NAp(13,10)
10	NAp(14 15)
11	NAp(6 16)
12	NAp(7 9)
13	NAp(5 11)
14	NAp(5 17)
15	NAp(4 18)
16	3
17	NAp(6 19)
18	4
19	2

Dump:

12 9	pc=102
	pc=-1

Abbildung 2.9: Zustand nach Kopieren der in Zellen 13 bis 16 und schließlich in 17 referenzierten Knoten

Aufgabe 23 Mit der obigen Klasse `ForwardNode` läßt sich eine Garbage Collecion für die G-Maschine umsetzen. Schreiben Sie eine Klasse `TwoSpaceCopyGC`, die die Klasse `GC` erweitert. Sie hat einen Konstruktor, in dem die Maschinenzustandsobjekt übergeben wird:

```
public TwoSpaceCopyGC(GmState st)
```

Beim Aufruf von `check(i)` sollen, falls keine `i` Speicherplätze mehr im Heap sind, alle noch gebrauchten Heapknoten in einen neuen Heap copiert werden. Der neue Heap wird schließlich der neue Heap im Maschinenzustand. Denken Sie daran auf Stack und den im Dump gespeicherten Stacks abgelegten Adressen auf die Adressen im neuen Heap abgelegt werden müssen. Vergessen Sie nicht auch die globalen Funktionsknoten des Heaps zu kopieren.

Als Test für Ihre garbage collection verwenden wir folgendes kleines fab4-Programm:

TestGC.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
constr Pair 2;

last xs = case xs of
  Cons y ys -> case ys of
                Nil       -> y;
                Cons z zs -> last ys;

take i xs = case (i=0) of
  True      -> (Nil);
  False     -> case xs of
    Nil       -> (Nil);
    Cons y ys -> (Cons y (take (i-1) ys));

repeat x = (Cons x (repeat x));

main = last (take 1000 (repeat 42))
```

Dieses Prgramm definiert eine unendliche Liste der Zahl 42, nimmt davon die Teilliste der ersten 1000 Elemente und gibt von dieser das letzte Element als Wert aus. Da Sie noch keinen Fab4-Compiler haben, steht Ihnen der Bytecode für dieses Programm (`http://www.panitz.name/student/gmc/TestGC.gmc`) im Netz zur Verfügung.

2.9 Java Calls

Dieses Kapitel beschäftigt sich mit einem sehr unangenehmen Kapitel. Deshalb soll es auch nicht Bestand der Vorlesung sein. Es geht darum aus einer Sprache heraus eine andere Sprache aufzurufen. In unserem Fall wollen wir in der Lage sein in der G-maschine mit Java zu kommunizieren. Javamethoden sollen aufrufbar sein, Java-Objekte sollen im Heap abgelegt werden können.

2.9.1 Ein weiterer Heapknoten

NJavaObject.java

```
package name.panitz.gm;

public class NJavaObject implements Node{
  Object o;
  NJavaObject(Object o){this.o=o;}

```



```
  public <a> a visit(NodeVisitor<a> v)throws Exception{
    return v.eval(this);
  }
  public String toString(){return "(JavaObject "+o+")";}
}
```

2.9.2 Javamethoden aufrufen

JavaCall.java

```
package name.panitz.gm;
import java.io.*;
import java.lang.reflect.Method;
import static name.panitz.gm.MarshallingNode.*;

public class JavaCall implements Instruction{
  public GmState process(GmState st) throws Exception{
    Node n=st.heap.getNode(st.stack.pop());
    int arity=0;
    if (n instanceof NChar)arity=(int)((NChar)n).c;
    else arity=((NNum)n).n;

    Writer originalWriter = st.output;
    st.output=new StringWriter();
    st=new PrintString().process(st);

    String methodName = st.output.toString();

    Object dies = st.heap.get(st.stack.pop())
                     .visit(new MarshallingNode(st));

    st.stack.pop();
    st.stack.pop();
    st.stack.pop();
    st.stack.pop();

    GmStack newArgs = new GmStack();
    for (int i=0;i<arity;i++){
      int a = st.stack.pop();
      newArgs.add(0,((NAp)st.heap.getNode(a)).n2);
    }

    st.stack.addAll(newArgs);
    Object [] javaArgs = new Object[arity];
    for (int i=0;i<arity;i++){
      WHNF.whnf(st);
      javaArgs[i]=st.heap.getNode(st.stack.pop())
                  .visit(new MarshallingNode(st));
    }

```

```
    st.output=originalWriter;

    Class klasse = dies.getClass();
    Class[] argType = new Class[arity];
    for (int i=0;i<arity;i++)argType[i]=javaArgs[i].getClass();
    Method m = klasse.getMethod(methodName,argType);

    final Object o=m.invoke(dies,javaArgs);
    final int neededSpace= MarshallingNode.neededHeapSpace(o);
    st.gc.check(neededSpace);

    int adr=demarshalling(o,st);
    st.stack.push(adr);

    return st;
  }
  public String toString(){return "JavaCall";}
}
```

StaticJavaCall.java

```
package name.panitz.gm;
import java.io.*;

public class StaticJavaCall implements Instruction{
  public GmState process(GmState st) throws Exception{
    Node n=st.heap.getNode(st.stack.pop());
    int arity=0;
    if (n instanceof NChar)arity=(int)((NChar)n).c;
    else arity=((NNum)n).n;

    Writer originalWriter = st.output;
    st.output=new StringWriter();
    st=new PrintString().process(st);
    st.stack.pop();
    st.stack.pop();
    st.stack.pop();

    String methodName = st.output.toString();

    GmStack newArgs = new GmStack();
    for (int i=0;i<arity;i++){
      int a = st.stack.pop();
      newArgs.add(0,((NAp)st.heap.getNode(a)).n2);
    }
    st.stack.addAll(newArgs);
    Object [] javaArgs = new Object[arity];
    for (int i=0;i<arity;i++){
      WHNF.whnf(st);
      javaArgs[i]=st.heap.getNode(st.stack.pop())
                      .visit(new MarshallingNode(st));
```



```
    }

    st.output=originalWriter;
    String className
      = methodName.substring(0,methodName.lastIndexOf('.'));
    String mName
      = methodName.substring(methodName.lastIndexOf('.')+1);

    Class klasse = Class.forName(className);
    Class[] argType = new Class[arity];
    for (int i=0;i<arity;i++)argType[i]=javaArgs[i].getClass();
    java.lang.reflect.Method m = klasse.getMethod(mName,argType);

    final Object o=m.invoke(null,javaArgs);
    final int neededSpace= MarshallingNode.neededHeapSpace(o);
    st.gc.check(neededSpace);
    st.stack.push(
       MarshallingNode.demarshalling(o,st));
    return st;
  }
  public String toString(){return "StaticJavaCall";}
}
```

2.9.3 Datenkonversion

MarshallingNode.java

```
package name.panitz.gm;
import name.panitz.util.Pair;

import static name.panitz.gm.ConstructorConstants.*;
import java.io.*;

public class MarshallingNode implements NodeVisitor<Object>{
  GmState st;
  public MarshallingNode(GmState st){this.st=st;}
  public Object eval(Node n)throws Exception{
    throw new Exception("unmatched pattern: "+n);
  }
  public Object eval(NInd n)throws Exception{
    st.stack.pop();
    st.stack.push(n.adr);
    return st.heap.get(n.adr).visit(this);}

  public Object eval(NAp n)throws Exception{
    throw new Exception("unmatched pattern: "+n);
  }

  public Object eval(NGlobal n)throws Exception{
    throw new Exception("unmatched pattern: "+n);
```

```
  }

  public Object eval(NJavaObject n){
    return n.o;
  }
  public Object eval(NNum n){return new Integer(n.n);}
  public Object eval(NChar n){return new Character(n.c);}

  public Object eval(NConstr constr)throws Exception{
    Writer newOut=new StringWriter();
    Writer orgOut=st.output;
    st.output=newOut;
    st.stack.push(st.alloc(constr));
    constr.visit(new PrintStringNode(st));
    st.output=orgOut;
    return newOut.toString();
  }

  static public int demarshalling(Object o,GmState st){
    final int neededSpace= neededHeapSpace(o);
    return demarshallingAux(o,st);
  }

  static  int neededHeapSpace(Object o){
    if (o instanceof String)return 1+2*((String)o).length();
    if (o instanceof Object[]) {
      Object[] xs=(Object[])o;
      int result = xs.length+1;
      for(Object x:xs)result=result+neededHeapSpace(x);
      return result;
    }
    if (o instanceof Pair)
      return 1+neededHeapSpace(((Pair)o).e1)
              +neededHeapSpace(((Pair)o).e2);
    return 1;
  }

  static private int demarshallingAux(Object o,GmState st){
    if (o instanceof Boolean){
      int [] args={};
      if ((Boolean)o) {return st.alloc(new NConstr(TRUE,args));}
      return st.alloc(new NConstr(FALSE,args));
    }
    if (o instanceof Integer) {
      return st.alloc(new NNum((Integer)o));}
    if (o instanceof Character){
      return st.alloc(new NChar((Character)o));}
    if (o instanceof Pair) {
      int [] args
        ={demarshallingAux(((Pair)o).e1,st)
```



```
         ,demarshallingAux(((Pair)o).e2,st)};
      return st.alloc(new NConstr(PAIR,args));

    }
    if (o instanceof Object[]) {
      Object[] xs=(Object[])o;
      int aTail = st.alloc(new NConstr(NIL));
      for (int i=xs.length-1;i>=0;i--){
        int aHead=demarshallingAux(xs[i],st);
        final int[] args={aHead,aTail};

        aTail=st.alloc(new NConstr(CONS,args));
      }
      return aTail;
    }
    if (o instanceof String){
      String s=(String)o;
      final int[] nilargs={};
      int aTail = st.alloc(new NConstr(NIL,nilargs));
      for (int i=s.length()-1;i>=0;i--){
        int aHead=st.alloc(new NChar(s.charAt(i)));

        final int[] args={aHead,aTail};

        aTail=st.alloc(new NConstr(CONS,args));
      }
      return aTail;
    }
    return st.alloc(new NJavaObject(o));
  }
}
```

2.9.4 Zugriff auf die Laufzeitumgebung

GetRuntime.java

```
package name.panitz.gm;
import java.io.*;

public class GetRuntime implements Instruction{
  public GmState process(GmState st) throws Exception{
    int adr=MarshallingNode.demarshalling(st,st);
    st.stack.push(adr);

    return st;
  }
  public String toString(){return "GetRuntime";}
}
```

Kapitel 3

Formale Sprachen, Grammatiken, Parser

Sprachen sind ein fundamentales Konzept nicht nur der Informatik. In der Informatik begegnen uns als auffälligste Form der Sprache die *Programmiersprachen*. Es gibt gewisse Regeln, nach denen die Sätze einer Sprache aus einer Menge von Wörtern geformt werden können. Die Regeln nennen wir im allgemeinen eine *Grammatik*. Ein Satz einer Programmiersprache nennen wir Programm. Die Wörter sind, Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen.

Wir werden in diesem Kapitel die wichtigsten Grundkenntnisse hierzu betrachten, wie sie zum Handwerkszeug eines jeden Informatikers gehören.

3.1 formale Sprachen

Eine der bahnbrechenden Erfindungen des 20. Jahrhunderts geht auf den Sprachwissenschaftler Noam Chomsky [Cho56][1] zurück. Er präsentierte als erster ein formales Regelsystem, mit dem die Grammatik einer Sprache beschrieben werden kann. Dieses Regelsystem ist in seiner Idee verblüffend einfach. Es bietet Regeln an, mit denen mechanisch die Sätze einer Sprache generiert werden können.

Systematisch wurden Chomsky Ideen zum erstenmal für die Beschreibung der Syntax der Programmiersprache Algol angewendet [NB60].

Definition Sei $\mathcal{A}$ eine Menge von Symbolen. Dann bezeichne $\mathcal{A}^*$ die Menge aller endlichen Zeichenketten $a_1 \ldots a_n$ mit $a_i \in\mathcal{A}$, $n \geq 0$.
$\mathcal{A}^+$ bezeichne die Menge aller endlichen Zeichenketten $a_1 \ldots a_n$ mit $a_i \in\mathcal{A}$, $n \geq 1$.

Für $\alpha \in\mathcal{A}^*$ bezeichne $\mid \alpha \mid$ die Länge der Zeichenkette α.

[1] Chomsky gilt als der am häufigsten zitierte Wissenschaftler des 20. Jahrhunderts. Heutzutage tritt Chomsky weniger durch seine wissenschaftlichen Arbeiten als vielmehr durch seinen Einsatz für Menschenrechte und bedrohte Völker in Erscheinung.

Definition (Grammatik) Eine Grammatik $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{R})$ besteht aus:

- einer Menge $\mathcal{T}$ von Wörtern, den *Terminalsymbolen*.
- einer Menge $\mathcal{N}$ von Nichtterminalsymbolen mit $\mathcal{N}\cap\mathcal{T}= \emptyset$.
- ein ausgezeichnetes Startsymbol $S \in\mathcal{N}$.
- einer endlichen Menge $\mathcal{R}$ von Regeln (Produktionen) der Form:
  α`::=`β, $\alpha \in (\mathcal{N}\cup\mathcal{T})^+$, $\beta \in (\mathcal{N}\cup\mathcal{T})^*$.

Grammatiken beschreiben Regeln zum Erzeugen von Sprachen. Dabei wird mit dem ausgezeichneten Startsymbol angefangen die Regeln anzuwenden. Bei einer Regelanwendung wird innerhalb einer Zeichenkette die linke Regelseite durch die rechte Regelseite ersetzt. Die Untermenge von $\mathcal{T}^*$ aller Zeichenketten die durch Regelanwendung aus S erzeugt werden kann, ist die durch die Grammatik erzeugte Sprache.

Kann durch eine Regelanwendung ein Wort α in ein Wort β abgelitten werden, so schreiben wir $\alpha \rightarrow \beta$. Für den transitiven Abschluß der Relation $\rightarrow$ benutzen wir das Symbol $\Rightarrow$: $\alpha \Rightarrow \beta$ bedeutet also, es gibt ein $n \geq= 0$, so daß es eine Ableitung der folgenden Form gibt: $\alpha \rightarrow_1 \ldots \rightarrow_n \beta$.

3.1.1 kontextfreie Grammatik

Im Compilerbau spielt eine spezielle Form von Grammatiken eine besondere Rolle. Dieses sind die kontextfreien Grammatiken. Die Syntax fast jeder Programmiersprache ist durch eine kontextfreie Grammatik beschrieben.

Definition (kontextfreie Grammatik) Eine Grammatik heißt kontextfrei, wenn jede Produktion der folgenden Form ist:
nt`::=` β, wobei $nt \in\mathcal{N}$, $\beta \in (\mathcal{N}\cup\mathcal{T})^*$.

Mit den Regeln einer kontextfreien Grammatik werden Sätze gebildet, indem ausgehend vom Startsymbol Regel angewendet werden. Bei einer Regelanwendung wird ein Nichtterminalzeichen t durch die Rechte Seite einer Regel, die t auf der linken Seite hat, ersetzt.

Beispiel:
Wir geben eine Grammatik an, die einfache Sätze über unser Sonnensystem auf Englisch bilden kann:

- $\mathcal{T}$= {mars,mercury,deimos,phoebus,orbits,is,a,moon,planet}
- $\mathcal{N}$= {*start*,*noun-phrase*,*verb-phrase*,*noun*,*verb*,*article*}
- S =*start*
- *start* ::= *noun-phrase verb-phrase*
  noun-phrase ::= *noun*
  noun-phrase ::= *article noun*
  verb-phrase ::= *verb noun-phrase*
  noun ::= planet
  noun ::= moon
  noun ::= mars

noun ::= deimos
noun ::= phoebus
verb ::= orbits
verb ::= is
article ::= a

Wir können mit dieser Grammatik Sätze in der folgenden Art bilden:

- *start*
  →*noun-phrase verb-phrase*
  →*article noun verb-phrase*
  →*article noun verb noun-phrase*
  → a *noun verb noun-phrase*
  → a moon *verb noun-phrase*
  → a moon orbits *noun-phrase*
  → a moon orbits *noun*
  → a moon orbits mars

- *start*
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → mercury *verb-phrase*
  → mercury *verb noun-phrase*
  → mercury is *noun-phrase*
  → mercury is *article noun*
  → mercury is a *noun*
  → mercury is a planet

- Mit dieser einfachen Grammatik lassen sich auch Sätze bilden, die weder korrektes Englisch sind, noch eine vernünftige inhaltliche Aussage machen:
  start
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → planet *verb-phrase*
  → planet *verb noun-phrase*
  → planet orbits *noun-phrase*
  → planet orbits *article noun*
  → planet orbits a *noun*
  → planet orbits a phoebus

Eine Grammatik beschreibt die Syntax einer Sprache; im Gegensatz zur Semantik, der Bedeutung, einer Sprache.

Rekursive Grammatiken

Die Grammatik aus dem letzten Beispiel kann nur endlich viele Sätze generieren. Will man mit einer Grammatik unendlich viele Sätze beschreiben, so wie eine Programmiersprache unendlich viele Programme hat, so kann man sich des Tricks der Rekursion bedienen. Eine Grammatik

kann rekursive Regeln enthalten; das sind Regeln, in denen auf der rechten Seite das Nichtterminalsymbol der linken Seite wieder auftaucht.

Beispiel:
Die folgende Grammatik erlaubt es, arithmetische Ausdrücke zu generieren:

- $\mathcal{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$
- $\mathcal{N} = \{start, expr, op, integer, digit\}$
- S =*start*
- *start* ::= *expr*
  expr ::= *integer*
  expr ::= *integer op expr*
  integer ::= *digit*
  integer ::= *digit integer*
  op ::= +
  op ::= -
  op ::= *
  op ::= /
  digit ::= 0
  digit ::= 1
  digit ::= 2
  digit ::= 3
  digit ::= 4
  digit ::= 5
  digit ::= 6
  digit ::= 7
  digit ::= 8
  digit ::= 9

Diese Grammatik hat zwei rekursive Regeln: eine für das Nichtterminal *expr* und eine für das Nichtterminal *integer*.

Folgende Ableitung generiert einen arithmetischen Ausdruck mit dieser Grammatik:

- *start*
  →*expr*
  →*integer op expr*
  →*integer op integer op expr*
  →*integer op integer op integer op expr*
  →*integer op integer op integer op integer*
  →*integer* + *integer op integer op integer*
  →*integer* + *integer* $*$ *integer op integer*
  →*integer* + *integer* $*$ *integer* −*integer*
  →*digit integer* + *integer* $*$ *integer* − *integer*
  → 1 *integer* + *integer* $*$ *integer* − *integer*
  → 1 *digit integer* + *integer* $*$*integer* − *integer*
  → 12 *integer* + *integer* $*$*integer* − *integer*
  → 12 *digit* + *integer* $*$ *integer* − *integer*
  → 129+ *integer* $*$ *integer* −*integer*
  → 129+ *digit* $*$ *integer* −*integer*

$\rightarrow 129 + 4 * \mathit{integer} - \mathit{integer}$
$\rightarrow 129 + 4 * \mathit{digit}\ \mathit{integer} - \mathit{integer}$
$\rightarrow 129 + 4 * 5\mathit{integer} - \mathit{integer}$
$\rightarrow 129 + 4 * 5\mathit{digit} - \mathit{integer}$
$\rightarrow 129 + 4 * 53 - \mathit{integer}$
$\rightarrow 129 + 4 * 53 - \mathit{digit}\ \mathit{integer}$
$\rightarrow 129 + 4 * 53 - 8\ \mathit{integer}$
$\rightarrow 129 + 4 * 53 - 8\ \mathit{digit}$
$\rightarrow 129 + 4 * 53 - 87$

Aufgabe 24 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbole: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Grenzen kontextfreier Grammatiken

Kontextfreie Grammatiken sind ein einfaches und dennoch mächtiges Beschreibungsmittel für Sprachen. Dennoch gibt es viele Sprachen, die nicht durch eine kontextfreie Grammatik beschrieben werden können.

syntaktische Grenzen Es gibt syntaktisch recht einfache Sprachen, die sich nicht durch eine kontextfreie Grammatik beschreiben lassen. Eine sehr einfache solche Sprache besteht aus drei Wörtern: $\mathcal{T}=$ {a,b,c}. Die Sätze dieser Sprache sollen so gebildet sein, daß für eine Zahl n eine Folge von n mal dem Zeichen a, n mal das Zeichen b und schließlich n mal das Zeichen c folgt, also

$$\{a^n b^n c^n | n \in I\!N\}.$$

Die Sätze dieser Sprache lassen sich aufzählen:

abc
aabbcc
aaabbbccc
aaaabbbbcccc
aaaaabbbbbccccc
aaaaaabbbbbbcccccc
...

Es gibt formale Beweise, daß derartige Sprachen sich nicht mit kontextfreie Grammatiken bilden lassen. Versuchen Sie einmal das Unmögliche: eine Grammatik aufzustellen, die diese Sprache erzeugt.

Eine weitere einfache Sprache, die nicht durch eine kontextfreie Grammatik auszudrücken ist, hat zwei Terminalsymbole und verlangt, daß in jedem Satz die beiden Symbole gleich oft vorkommen, die Reihenfolge jedoch beliebig sein kann.

semantische Grenzen Über die Syntax hinaus, haben Sprachen noch weitere Einschränkungen, die sich nicht in der Grammatik ausdrücken lassen. Die meisten syntaktisch korrekten Javaprogramme werden trotzdem vom Javaübersetzer als inkorrekt zurückgewiesen. Diese Programme verstoßen gegen semantische Beschränkungen, wie z.B. gegen die Zuweisungskompatibilität. Das Programm:

```
class SemanticalIncorrect{int i = "1";}
```

ist syntaktisch nach den Regeln der Javagrammatik korrekt gebildet, verletzt aber die Beschränkung, daß einem Feld vom Typ `int` kein Objekt des Typs `String` zugewiesen werden darf.

Aus diesen Grund besteht ein Übersetzer aus zwei großen Teilen. Der syntaktischen Analyse, die prüft, ob der Satz mit den Regeln der Grammatik erzeugt werden kann und der semantischen Analyse, die anschließend zusätzliche semantische Bedingungen prüft.

Das leere Wort

Manchmal will man in einer Grammatik ausdrücken, daß in Nichtterminalsymbol auch zu einem leeren Folge von Symbolen reduzieren soll. Hierzu könnte man die Regel

t::=

mit leerer rechter Seite schreiben. Es ist eine Konvention ein spezielles Zeichen für das leere Wort zu benutzen. Hierzu bedient man sich des griechischen Buchstabens ϵ. Obige Regel würde man also schreiben als:

t::=ϵ

3.1.2 kontextsensitive Grammatiken

Die im vorherigen Abschnitt vorgestellten Grammatiken heißen *kontextfrei*, weil eine Regel für ein Nichtterminalzeichen angewendet wird, ohne dabei zu betrachten, was vor oder nach dem Zeichen für ein weiteres Zeichen steht, der Kontext also nicht betrachtet wird. Läßt man auch Regeln zu, die auf der linken Seite nicht ein Nichtterminalzeichen stehen haben, so kann man mächtigere Sprachen beschreiben, als mit einer kontextfreien Grammatik.

Definition (kontextsensitive Grammatik) Sind alle Produktionen einer Grammatik der Form:

- α`::=`β mit $|\ \alpha\ | \leq |\ \beta\ |$ oder
- $\sigma A \tau$`::=`$\sigma\gamma\tau$ mit $\sigma, \tau \in (\mathcal{N} \cup \mathcal{T})^*$, A$\in \mathcal{N}^+, \gamma \in (\mathcal{N} \cup \mathcal{T})^+$

so heißt die Grammatik kontextsensitiv.

Beispiel:
Wir können mit der folgenden nicht-kontextfreien Grammatik die Sprache beschreiben, in der jeder Satz gleich oft die beiden Terminalsymbole, aber in beliebiger Reihenfolge enthält.

- $\mathcal{T}$= {a,b}
- $\mathcal{N}$= $\{start, A, B\}$
- $S = start$
- $start$ ::= $ABstart$
  start ::= AB
  AB ::= BA
  BA ::= AB
  A ::= a
  B ::= b

$start$
$\rightarrow ABstart$
$\rightarrow ABABstart$
$\rightarrow ABABABstart$
$\rightarrow ABABABABstart$
$\rightarrow ABABABABAB$
$\rightarrow AABBABABAB$
$\rightarrow AABBBAABAB$
$\rightarrow AABBBABAAB$
$\rightarrow AABBBABABA$
$\rightarrow AABBBABBAA$
$\rightarrow AABBBBABAA$
$\rightarrow AABBBBBAAA$
$\rightarrow$ a$ABBBBBAAA$
$\rightarrow$ aa$BBBBBAAA$
$\rightarrow$ aab$BBBBAAA$
$\rightarrow$ aabb$BBBAAA$
$\rightarrow$ aabbb$BBAAA$
$\rightarrow$ aabbbb$BAAA$
$\rightarrow$ aabbbbbAAA
$\rightarrow$ aabbbbbaAA
$\rightarrow$ aabbbbbaa$A$
$\rightarrow$ aabbbbbaaa

Beispiel:
Auch die nicht durch eine kontextfreie Grammatik darstellbare Sprache:

$\{a^n b^n c^n | n \in I\!N\}$.

läßt sich mit einer solchen Grammatik generieren:

S::=ab$T$c
T::=AbTc|Abc
bA::=Ab
aA::=aa

Eine Ableitung mit dieser Grammatik sieht wie folgt aus:

S
$\rightarrow$ ab$T$c
$\rightarrow$ ab$A$bTcc
$\rightarrow$ ab$A$bAbTccc
$\rightarrow$ ab$A$bAbAbTcccc
$\rightarrow$ ab$A$bAbAbAbccccc
$\rightarrow$ abAAbb$A$bAbccccc
$\rightarrow$ abAAbAbb$A$bccccc
$\rightarrow$ abAAAbbb$A$bccccc
$\rightarrow$ abAAAbb$A$bbccccc
$\rightarrow$ abAAAbAbbbccccc
$\rightarrow$ ab$AAAA$bbbbccccc
$\rightarrow$ aAbAAAbbbbccccc
$\rightarrow$ aAAbAAbbbbccccc
$\rightarrow$ aAAAbAbbbbccccc
$\rightarrow$ a$AAAA$bbbbbccccc
$\rightarrow$ aaAAAbbbbbccccc
$\rightarrow$ aaaAAbbbbbccccc
$\rightarrow$ aaaa$A$bbbbbccccc
$\rightarrow$ aaaaabbbbbccccc

Kontextsensitive Sprachen spielen im praktischen Compilerbau kaum eine Rolle.

3.1.3 reguläre Grammatiken

Schränkt man die Regeln einer kontextfreien Grammatik noch weiter ein so erhält man sogenannte reguläre Sprachen.

Definition (reguläre Gramatik) Haben alle Produktionen einer kontextfreien Grammatik eine der beiden Formen:

- A`::=`a, mit $a \in \mathcal{T}$
- A`::=`αB mit $\alpha \in \mathcal{T}^*$, $B \in \mathcal{N}$

so nennt man die Grammatik rechtslinear.

Analog für Regeln der Form A`::=`$B\alpha$ mit $\alpha \in \mathcal{T}^*$, $B \in \mathcal{N}$ wird eine Grammatik linkslinear bezeichnet.

Eine Grammatik, die linkslinear oder rechtslinear ist, wird als regulär bezeichnet. Die entsprechende Sprache wird auch als regulär bezeichnet.

Beispiel:

- $\mathcal{T}$= $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}$
- $\mathcal{N}$= $\{S, A, B\}$

- Produktionen:
  S::=$+A \mid -A \mid B$
  A::=$0B \mid 1B \mid 2B \mid 3B \mid 4B \mid 5B \mid 6B \mid 7B \mid 8B \mid 9B$
  B::=$0B \mid 1B \mid 2B \mid 3B \mid 4B \mid 5B \mid 6B \mid 7B \mid 8B \mid 9B \mid \epsilon$

reguläre Ausdrücke

Wie dem obigen Beispiel zu entnehmen war, sehen reguläre Grammatiken nicht sehr schön aus. Es gibt einen weiteren Beschreibungsformalismus für reguläre Sprachen erzeugt. Die sogenannten regulären Ausdrücke.

Definition (Syntax regulärer Ausdrücke) Reguläre Ausdrücke über ein Alphabet Σ sind wie folgt induktiv definiert:

- für alle $\alpha \in \sigma$ gilt: α ist ein regulärer Ausdruck.
- Der griechische Buchstabe Λ ist ein regulärer Ausdruck.
- Sind α und β reguläre Ausdrücke, so sind auch:
  - $(\alpha\beta)$ (Produkt)
  - $(\alpha \mid \beta)$ (Summe)
  - α^* (Wiederholung)

  reguläre Ausdrücke.

Reguläre Ausdrücke beschreiben auch Sprachen

Definition (Semantik regulärer Ausdrücke) Für jeden regulären Ausdruck α über ein Alphabet Σ wird die durch ihn beschriebene Sprache $L(\alpha)$ wie folgt induktiv definiert:

- $L(a) = a$ für $a \in \Sigma$
- $L(\Lambda) = \emptyset$.
- $L(\alpha\beta) = \{ab \mid a \in L(\alpha), b \in L(\beta)$.
- $L(\alpha \mid \beta) = L(\alpha) \cup L(\beta)$.
- $L(\alpha^*) = L(\alpha)^*$.

Aus der theoretischen Informatik ist bekannt, daß die Sprachen, die durch reguläre Ausdrücke beschrieben werden können, genau die Sprachen sind, die durch reguläre Grammatiken beschrieben werden können.

3.2 Grammatiken und Bäume

Eine Grammatik stellt in naheliegender Weise nicht nur ein Beschreibung einer Sprache dar, sondern jede Generierung eines Satzes dieser Sprache entspricht einem Baum, dem Ableitungsbaum. Die Knoten des Baumes sind mit Terminal- und Nichtterminalzeichen markiert, wobei Blätter mit Terminalzeichen markiert sind. Die Kinder eines Knotens sind die Knoten, die mit der rechten Seite einer Regelanwendung markiert sind.

Liest man die Blätter eines solchen Baumes von links nach rechts, so ergibt sich der generierte Satz.

> **Beispiel:**
> Die Ableitungen der Sätze unserer ersten Grammatik haben folgende Baumdarstellung:

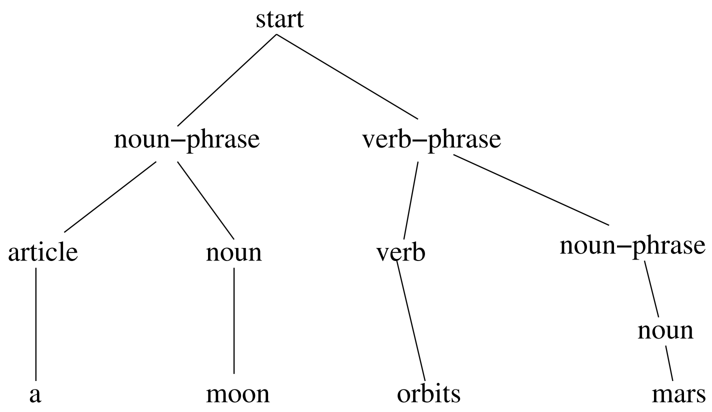


Abbildung 3.1: Ableitungsbaum für a moon orbits mars.

Gegenüber unserer bisherigen Darstellung der Ableitung eines Wortes mit den Regeln einer Grammatik, ist die Reihenfolge, in der die Regeln angewendet werden in der Baumdarstellung nicht mehr ersichtlich. Daher spricht man häufiger auch vom Syntaxbaum des Satzes.

Aufgabe 25 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzten Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

3.3 Erweiterte Backus-Naur-Form

Die Ausdrucksmöglichkeit einer kontextfreien Grammatik ist auf wenige Konstrukte beschränkt. Das macht das Konzept einfach. Im Kontext von Programmiersprachen gibt es häufig sprachliche Konstrukte, wie die mehrfache Wiederholung oder eine Liste von bestimmten Teilen, die

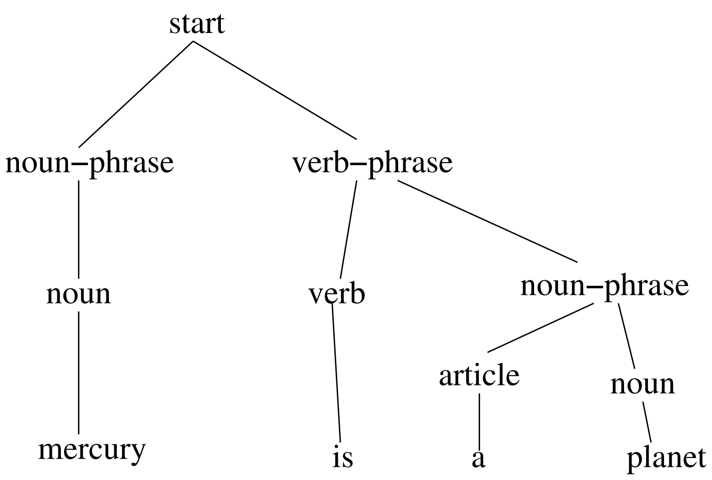


Abbildung 3.2: Ableitungsbaum für mercury is a planet.

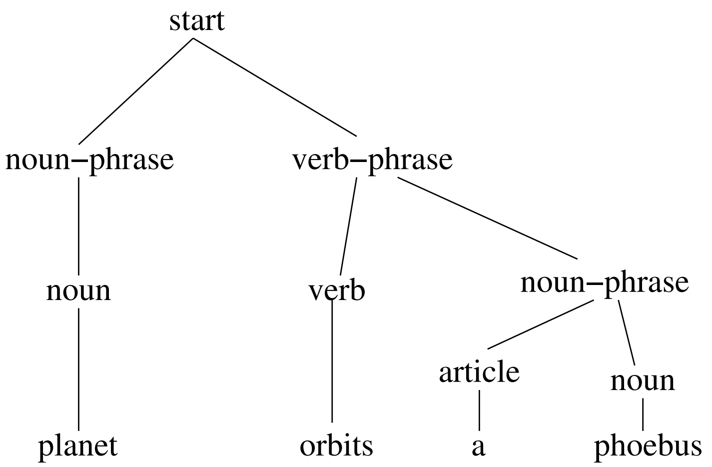


Abbildung 3.3: Ableitungsbaum für planet orbits a phoebus.

zwar mit einer kontextfreien Grammatik darstellbar ist, für die aber spezielles zusätzliche Ausdrucksmittel in der Grammatik eingeführt werden. In den nächsten Abschnitten werden wir diese Erweiterungen kennenlernen, allerdings werden wir in unseren Algorithmen für Sprachen und Grammatiken diese Erweiterungen nicht berücksichtigen.

Alternativen

Wenn es für ein Nichtterminalzeichen mehrere Regeln gibt, so werden diese Regeln zu einer Regel umgestaltet. Die unterschiedlichen rechten Seiten werden dann durch einen vertikalen Strich | gestrennt.

Durch diese Darstellung verliert man leider den direkten Zusammenhang zwischen den Regeln und einen Ableitungsbaum.

Beispiel:
Die Regeln unserer ersten Grammatik können damit wie folgt geschrieben werden:

start ::= *noun-phrase verb-phrase*
noun-phrase ::= *noun*|*article noun*
verb-phrase ::= *verb noun-phrase*
noun ::= planet|moon|mars|deimos|phoebus
verb ::= orbits|is
article ::= a

Gruppierung

Bestimmte Teile der rechten Seite einer Grammatik können durch Klammern gruppiert werden. In diesen Klammern können wieder durch einen vertikalen Strich getrennte Alternativen stehen.

Beispiel:
Die einfache Grammatik für arithmetische Ausdrücke läßt sich damit ohne das Nichtterminalzeichen *op* schreiben:

start ::= *expr*
expr ::= *integer*|*integer* (+|−|∗|/) *expr*
integer ::= *digit*|*digit integer*
digit ::= 0|1|2|3|4|5|6|7|8|9

Wiederholungen

Ein typische Konstrukt in Programmiersprachen ist, daß bestimmte Kontrukte wiederholt werdn können. So stehen z.B. in Java im Rumpf einer Methode mehrere Anweisungen. Solche Sprachkonstrukte lassen sich mit einer kontextfreien Grammatik ausdrücken.

Beispiel:
Eine Zahl besteht aus einer Folge von n Ziffern ($n > 0$). Dieses läßt sich durch folgende Regel ausdrücken:

Zahl ::= *Ziffer Zahl* | *Ziffer*

1 bis n-fach Im obigen Beispiel handelt es sich um eine 1 bis n-fache Wiederholung des Zeichens *Ziffer*. Hierzu gibt es eine abkürzende Schreibweise. Dem zu wiederholenden Teil wird das Zeichen `+` nachgestellt.

> **Beispiel:**
> Obige Regel für das Nichtterminal *Zahl* läßt sich mit dieser abkürzenden Schreibweise schreiben als:
>
> *Zahl* ::= *Ziffer*`+`

0 bis n-fach Soll ein Teil in einer Wiederholung auch keinmal vorkommen, so wird statt des Zeichens `+` das Zeichen `*` genommen.

> **Beispiel:**
> Folgende Regel drückt aus, daß ein Ausdruck eine durch Operatoren getrennte Liste von Zahlen ist.
>
> *expr* ::= *Zahl* | (*Op Zahl*)*

Option Ein weiterer Spezialfall der Wiederholung ist die, in der der entsprechende Teil keinmal oder einmal vorkommen darf, d.h. der Teil ist optional. Optionale Teile werden in der erweiterten Form in eckige Klammern gesetzt.

3.4 Mehrdeutigkeiten

Betrachten wir noch einmal die erste Grammatik für arithmetische Ausdrücke.

- $\mathcal{T}= \{number, +, -, *, /\}$
- $\mathcal{N}= \{expr, op\}$
- $S = expr$
- *expr* ::= *expr op expr*
  |`zahl`
  op ::= `+`|`-`|`*`|`/`

Wollen wir mit dieser Grammatik den Ausdruck `17+4*2` ableiten, so sind verschiedene Ableitungen denkbar. Betrachten wir zwei solche:

- *expr*
  →*expr op expr*
  →`zahl` *op expr*
  →`zahl + ` *expr*
  →`zahl + ` *expr op expr*
  →`zahl + zahl` *op expr*
  →`zahl + zahl * ` *expr*
  →`zahl + zahl * zahl`

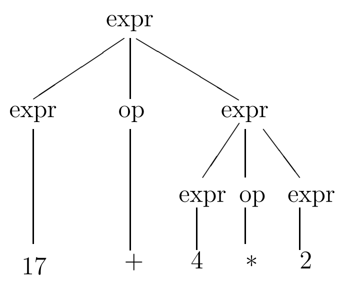


Abbildung 3.4: Syntaxbaum zu 17+4*2.

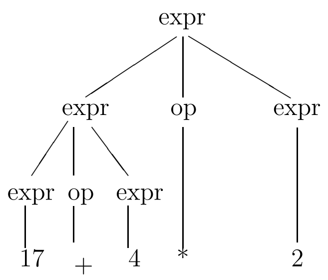


Abbildung 3.5: Syntaxbaum zu 17+4*2.

- *expr*
  →*expr op expr*
  →*expr* `*` *expr*
  →*expr op expr* `*` *expr*
  →*expr op expr* `* zahl`
  →*expr op* `zahl * zahl`
  →*expr* `+ zahl * zahl`
  →`zahl + zahl * zahl`

Die Bäume dieser beiden Ableitungen sind in Abbildung 3.4 bzw. 3.5 gegeben.

Sie unterscheiden sich darin, ob für das erste oder für das zweite Auftreten des Nichtterminals *expr* die Operatorregel angewendet wird.

Definition (eindeutige Grammatik) Eine kontextfreie Grammatik heißt eindeutig, wenn es zu jedem Satz der durch die Grammatik erzeugten Sprache genau einen Ableitungsbaum gibt. Ansonsten heißt die Grammatik mehrdeutig.

Abstrahieren wir die obigen Syntaxbäume zu Bäumen, die als Berechnungsvorschrift für den arithmetischen Ausdruck betrachtet werden können, so erhalten wir die Bäume aus Abbildung 3.6. Die beiden Bäume klammern die Ausdrücke unterschiedlich. Benutzt man sie für die Be-

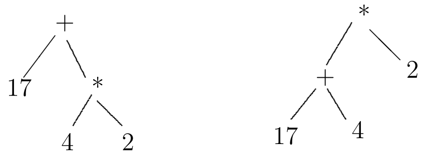


Abbildung 3.6: Berechnungsbäume zu den zwei Ableitungen von 17+4*2.

rechnung de Ausdrucks, so erhält man einmal das Ergebnis `42` das andere Mal das Ergebnis `25`.

Verknüpft man an den Ableitungsbaum also auch eine Semantik, so kann ein Satz unterschiedliche Bedeutungen haben. Auch die Grammatik der deutsche Sprache ist mehrdeutig. man betrachte einmal den Satz:

> Gestern ist eine Mutter von fünf Kindern ermodet worden.

Er kann zwei sehr unterschiedliche Bedeutungen haben, je nachdem wie wir ihn mit den Regeln unserer Grammatik ableiten.[2]

In Programmiersprachen sind mehrdeutige Programme natürlich nicht wünschenswert. Eine eindeutige Semantik für Programme sollte gegeben sein. Dieses kann erreicht werden, indem die Grammatik so modelliert wird, daß sie eindeutig ist, oder Mehrdeutigkeiten gesondert aufgelöst werden.

Betrachten wir noch einmal unser Beispiel der arithmetischen Ausdrücke. Die Grammatik läßt sich auf einfache Weise in eine eindeutige Grammatik, die dieselbe Sprache erzeugt, umschreiben.

- $\mathcal{T}=\{number, +, -, *, /\}$
- $\mathcal{N}=\{expr, op\}$
- S =*expr*
- *expr* ::= `zahl` *op expr*
  |`zahl`
  op ::= `+`|`-`|`*`|`/`

In dieser Version der Grammatik haben wir festgelegt, daß wir die Ausdrücke als von rechts nach links geklammert verstehen wollen, entsprechend den Ableitungsbaum aus Abbildung 3.4. Allerdings entspricht diese Regel nicht unserer intendierten Semantik für arithmetische Ausdrücke. In der Mathematik benutzen wir die implizite Regel, daß die Punktrechnung vor der Strichrechnung auszuführen ist[3]. Die obige Grammtik würde für den Ausdruck `2*17+4` einen Ableitungsbaum erzeugen, der der Klammerung `(2*(17+4))` entspricht. Dieses ist nicht die intendierte Semantik des Ausdrucks. Ebenso würde für einen Ausdruck der Form `44-1-1` die Klammerung `(44-(1-1))` benutzt werden. Auch das widerspricht der intendierten Semantik.

[2] In natürlichen Sprachen bedienen wir uns zum Auflösen von Mehrdeutigkeiten des Kontextes aber auch zusätzlicher Sprachattribute wie Sprachmelodie, Betonung und Pausen.

[3] So wie weitere Regeln, daß Vergleichsoperatoren nach den Strichoperatoren auszuwerten sind

Man kann die relativ komplexe Semantik arithmetischer Ausdrücke in einer Grammatik zu codieren. Hierzu benutzt man statt einer Regel, in der ein Operatorausdruck mit zwei Operandenausdrücken abgeleitet wird, eine Staffel von Regeln. Für jede Klasse von Operatoren mit einer Bindungspräzedenz wird eine solche Regel definiert. In den Regeln für Operatoren die schwächer binden, tauchen die Nichtterminale der Regeln, die stärker binden auf der rechten Seite auf. Arithmetische Ausdrücke bekommen damit die komplexere Grammatik:

- $\mathcal{T}=\{number, +, -, *, /\}$
- $\mathcal{N}=\{addExpr, multExpr, addOp, multOp\}$
- S =*addExpr*
- *addExpr* ::= *addExpr addOp multExpr*
  |*multExpr*

  multExpr ::= *multExpr multOp* `zahl`
  |`zahl`

  addOp ::= `+`|`-`
  multOp ::= `*`|`/`

Diese Grammatik ist eindeutig und erzeugt immer einen Ableitungsbaum, der der Standardsemantik arithmetischer Ausdrücke entspricht. Punktrechnung geht vor Strichrechnung und gleichwertige arithmetische Ausdrücke sind links geklammert. Hierzu zwei Beispiele:

- *addExpr*
  →*addExpr addOp multExpr*
  →*addExpr addOp multExpr multOp* `2`
  →*addExpr addOp multExpr* `* 2`
  →*addExpr addOp* `4 * 2`
  →*addExpr* `+ 4 * 2`
  →`17 + 4 * 2`

- *addExpr*
  →*addExpr addOp multExpr*
  →*addExpr addOp multExpr addOp multExpr*
  →*multExpr addOp multExpr addOp multExpr*
  →`44` *addOp multExpr addOp multExpr*
  →`44 -` *multExpr addOp multExpr*
  →`44 - 1` *addOp multExpr*
  →`44 - 1 -` *multExpr*
  →`44 - 1 - 1`

3.5 Parser

Grammatiken geben an, wie Sätze einer Sprache gebildet werden können. In der Informatik interessiert der umgekehrte Fall. Ein Satz ist vorgegeben und es soll geprüft werden, ob dieser Satz mit einer bestimmten Grammatik erzeugt werden kann. Ein Javaübersetzer prüft z.B. ob ein vorgegebenes Programm syntaktisch zur durch die Javagrammatik beschriebenen Sprache gehört. Hierzu ist ein Prüfalgorithmus anzugeben, der testet, ob die Grammatik einen bestimmten Satz generieren kann. Ein solches Programm wird *Parser* genannt. Ein Parser[4] zerteilt einen Satz in seine syntaktischen Bestandteile.

3.5.1 Parsstrategien

Man kann unterschiedliche Algorithmen benutzen, um zu testen, daß ein zu der Sprache einer Grammatik gehört. Hierbei unterscheidet man zwei grundlegende Vorgehensweisen:

- *top-down-Verfahren*: ausgehend von dem Startsymbol der Grammatik wird versucht ein Ableitungsbaum zu erzeugen. Man fängt also an der Wurzel des Ableitungsbeumes an, zu probieren, ob ein Ableitungsbaum für den Satz gebildet werden kann. Diese Strategie geht primär von der Grammatik aus.
- *bottom-up-Verfahren*: Ausgehend von den abzuleitenden Satz wird versucht rückwärts die Regeln der Grammatik anzuwenden. Es wird also von den Blättern des Ableitungsbaumes versucht auszugehen. Diese Strategie geht primär von den zu prüfenden Satz aus.

Wir werden uns in dieser Vorlesung hauptsächlich mit einem*top-down-Verfahren* beschäftigen.

Rekursiv absteigend

Eine einfache Idee zum Schreiben eines Parsers ist es, einfach nacheinander auszuprobieren, ob die Regelanwendung nach und nach einen Satz bilden kann. Hierzu startet man mit dem Startsymbol und wendet nacheinander die Regeln an. Dabei wendet man immer eine Regel auf das linkeste Nichtterminalzeichen an, so lange, bis der Satzanfang abgelitten wurde, oder aber ein falscher Satzanfang abgelitten wurde. Im letzteren Fall hat man offensichtlich bei einer Regel die falsche Alternative gewählt. Man ist in eine Sackgasse geraten. Nun geht man in seiner Ableitung zurück bis zu der letzten Regelanwendung, an der eine andere Alternative hätte gewählt werden können. Man spricht dann von *backtracking*; auf deutsch kann man treffend von Zurückverfolgen sprechen.

Auf diese Weise gelangt man entweder zu einer Ableitung des Satzes, oder stellt irgendwann fest, daß man alle Alternativen ausprobiert hat und immer in eine Sackgasse geraten ist. Dann ist der Satz nicht in der Sprache.

Diese Strategie ist eine Suche. Es wird systematisch aus allen möglichen Ableitungen in der Grammatik nach der Ableitung gesucht, die den gewünschten Satz findet.

Beispiel:
Wir suchen mit dieser Strategie die Ableitung des Satzes *a moon orbits mars* in dem Sonnensystembeispiel. Sackgassen sind durch einen Punkt • markiert.

(0)		*start*	
(1) aus 0	→	*noun-phrase verb-phrase*	
(2a) aus 1	→	*noun verb-phrase*	
(2a3a) aus 2a	→	planet *verb-phrase*	•
(2a3b) aus 2a	→	moon *verb-phrase*	•
(2a3c) aus 2a	→	mars *verb-phrase*	•
(2a3d) aus 2a	→	deimos *verb-phrase*	•
(2a3e) aus 2a	→	phobus *verb-phrase*	•
(2b) aus 1	→	*article noun verb-phrase*	
(3) aus 2b	→	a *noun verb-phrase*	
(4a) aus 3	→	a planet *verb-phrase*	•
(4b) aus 3	→	a moon *verb-phrase*	
(5) aus 4b	→	a moon *verb noun-phrase*	
(6) aus 5	→	a moon orbits *noun-phrase*	
(7) aus 6	→	a moon orbits *noun*	
(8a) aus 7	→	a moon orbits planet	•
(8b) aus 7	→	a moon orbits moon	•
(8c) aus 7	→	a moon orbits mars	

[4] Lateinisch Pars bedeutet Teil.

Alternativ könnte man diese Strategie auch symetrisch nicht von links sondern von der rechten Seite an ausführen, also immer eine Regel für das rechtestes Nichtterminalsymbol anwenden.

Linksrekursion Die Strategie des rekursiven Abstiegs auf der linken Seite funktioniert für eine bestimmte Art von Regeln nicht. Dieses sind Regeln, die in einer Alternative als erstes Symbol wieder das Nichtterminalsymbol stehen haben, das auch auf der linken Seite steht. Solche Regeln heißen linksrekursiv. Unsere Strategie terminiert in diesen Fall nicht.

Beispiel:
Gegeben sei eine Grammatik mit einer linksrekursiven Regel:

expr::=*expr*+*zahl*|*zahl*

Der Versuch den Satz

zahl+zahl

mit einem links rekursiv absteigenden Parser abzuleiten, führt zu einem nicht terminierenden rekursiven Abstieg. Wir gelangen nie in eine Sackgasse:

$$\begin{array}{ll} & expr \\ \rightarrow & expr+zahl \\ \rightarrow & expr+zahl+zahl \\ \rightarrow & expr+zahl+zahl+zahl \\ \rightarrow & expr+zahl+zahl+zahl+zahl \\ & \ldots \end{array}$$

Es läßt sich also nicht für alle Grammatiken mit dem Verfahren des rekursiven Abstiegs entscheiden, ob ein Satz mit der Grammatik erzeugt werden kann; aber es ist möglich, eine Grammatik mit linksrekursiven Regeln so umzuschreiben, daß sie die gleiche Sprache generiert, jedoch nicht mehr linksrekursiv ist. Hierzu gibt es ein einfaches Schema:

Eine Regel nach dem Schema:

$A ::= A\ rest|alt2$

ist zu ersetzen durch die zwei Regeln:

$A ::= alt2\ R$
$R ::= rest\ R|\epsilon$

wobei R ein neues Nichtterminalsymbol ist.

Beispiel:
Wir können uns wieder der Grammatik der arithmetischen Ausdrücke aus dem letzten Abschnitt zuwenden. Sie hat zwei linksrekursive Regeln. Elemination der linksrekursiven Regeln ergint folgende neue äquivalente Grammatik:

- $\mathcal{T} = \{number, +, -, *, /\}$
- $\mathcal{N} = \{addExpr1, addExpr2, multExpr1, multExpr2, addOp, multOp\}$
- $S = addExpr1$
- *addExpr1* ::= *multExpr1 addExpr2*
  addExpr2 ::= *addOp multExpr1 addExpr2*
  $|\epsilon$
  multExpr1 ::= `zahl` *multExpr2*
  mutlExpr2 ::= *multOp* `zahl` *multExpr2*
  $|\epsilon$
  addOp ::= `+`|`-`
  multOp ::= `*`|`/`

Linkseindeutigkeit Die rekursiv absteigende Strategie hat einen weiteren Nachteil. Wenn sie streng schematisch angewendet wird, führt sie dazu, daß bestimmte Prüfungen mehrfach durchgeführt werden. Diese mehrfache Ausführung kann sich bei wiederholter Regelanwendung multipizieren und zu einem sehr ineffizienten Parser führen. Grund dafür sind Regeln, die zwei Alternativen mit gleichem Anfang haben:

$S ::= AB|A$

Beide Alternativen für das Nichtterminalzeichen S starten mit dem Zeichen A. Für unseren Parser bedeutet das soweit: versuche nach der ersten Alternative zu parsen. Hierzu parse erst nach dem Symbol A. Das kann eine sehr komplexe Berechnung sein. Wenn sie gelingt, dann versuche anschließend weiter nach dem Symbol B zu parsen. Wenn das fehlschlägt, dann verwerfe die Regelalternative und versuche nach der zweiten Regelalternative zu parsen. Jetzt ist wieder nach dem Symbol A zu parsen, was wir bereits gemacht haben.

Regeln der obigen Art wirken sich auf unsere Parsstrategie ungünstig aus. Wir können aber ein Schema angegeben, wie man solche Regeln aus der Grammatik eleminiert, ohne die erzeugte Sprache zu ändern:

Regeln der Form

$S ::= AB|A$

sind zu ersetzen durch

$S ::= AT$
$T ::= B|\epsilon$

wobei T ein neues Nichtterminalzeichen ist.

Diese Transformation auf Grammatiken nennt man: Linksfaktorierung. Sie funktioniert analog zu der Umwandlung in der Arithmetik: $x * y + x * z = x * (y + z)$.

Vorausschau Im letzten Abschnitt haben wir gesehen, wie wir die Grammatik umschreiben können, so daß nach dem Zurücksetzen in unserem Algorithmus es nicht vorkommt, bereits ausgeführte Regeln ein weiteres Mal zu durchlaufen. Schöner noch wäre es, wenn wir auf das Zurücksetzen ganz verzichten könnten. Dieses läßt sich allgemein nicht erreichen, aber es gibt Grammatiken, in denen man durch Betrachtung des nächsten zu parsenden Zeichens erkennen kann, welche der Regelalternativen als einzige Alternative in betracht kommt. Hierzu kann man für eine Grammatik für jedes Nichtterminalzeichen in jeder Regelalternative berechnen, welche Terminalzeichen als linkestes Zeichen in einem mit dieser Regel abgelittenen Satz auftreten kann. Wenn die Regelalternativen disjunkte solche Menge des ersten Zeichens haben, so ist eindeutig bei Betrachtung des ersten Zeichens eines zu parsenden Satzes erkennbar, ob und mit welcher Alternative dieser Satz nur parsbar sein kann.

Die gängigsten Parser benutzen diese Entscheidung, nach dem erstem Zeichen. Diese Parser sind darauf angewiesen, daß die Menge der ersten Zeichen der verschiedenen Regelalternativen disjunkt sind.

Der Vorteil an diesem Verfahren ist, daß die Token nach und nach von links nach rechts stückweise konsumiert werden. Sie können durch einen Datenstrom, relisiert werden. Wurden sie einmal konsumiert, so werden sie nicht mehr zum Parsen benötigt, weil es kein Zurücksetzen gibt.

Definition (First-Menge) Für $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{R})$ und $\sigma \in (\mathcal{T} \cup \mathcal{N})^*$ sei $First(\sigma)$ definiert durch:

- $First(\sigma) = \{\mathrm{t} \in \mathcal{T} \mid \sigma \Rightarrow \mathrm{t}\alpha\} \cup \{\epsilon\}, (\alpha \in (\mathcal{T} \cup \mathcal{N})^*)$ falls $\sigma \Rightarrow \epsilon$
- $First(\sigma) = \{\mathrm{t} \in \mathcal{T} \mid \sigma \Rightarrow \mathrm{t}\alpha\}, (\alpha \in (\mathcal{T} \cup \mathcal{N})^*)$ andernfalls.

First-Mengen sind enthalten die möglichen Satzanfänge, die aus einer Zeichenkette abgelitten werden kann. First-Mengen lassen sich algorithmisch bestimmen:

Berechnung von $First(X)$ für $X \in \mathcal{N} \cup \mathcal{T}$:
die folgenden Regeln sind solange anzuwenden, bis zu keiner *First*-Menge mehr ein neues Terminal oderϵ hinzukommt:

- für $X \in \mathcal{T}$ ist $First(X) = \{X\}$.
- gilt $X \rightarrow \epsilon$ so füge ϵ zu $First(X)$ hinzu.
- für $X \in \mathcal{N}$ und $X ::= Y_1 \ldots Y_k \in \mathcal{R}$ dann füge wie folgt Symbole zu $First(X)$ hinzu:

  - falls $a \in \mathit{First}(Y_i)$ und für alle $1 \le j < i$ gilt $\epsilon \in \mathit{First}(Y_j)$ dann füge a zu $\mathit{First}(X)$ hinzu.
  - wenn für alle $1 \le i \le k$ gilt $\epsilon \in \mathit{First}(Y_i)$ dann füge ϵ zu $\mathit{First}(X)$ hinzu.

Dieser Algorithmus spielt quasi abstrakt die möglichen Ableitungen der Grammatik durch.

Auf naheliegende Weise läßt sich ein Algorithmus zur Berechnung der First-Menge eines Wortes erweitern.

Um gutartige Grammatiken zu definieren, für die insbesondere Sackgassen vermieden werden können, benötigen wir noch einen zweiten Schritt.

Definition (Follow-Menge) Für $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{R})$ und $A \in \mathcal{N}$ sei $\mathit{Follow}(A)$ definiert durch: $\mathit{Follow}(A) = \{\mathrm{t} \in \mathcal{T} \mid S \Rightarrow \alpha A \mathrm{t} \beta\}$

Die Follow-Menge beschreibt alle Terminalsymbole die in einer Zeichenfolge einer Ableitung einem bestimmten Nichtterminal folgen können. Auch Follow-Mengen lassen sich algorithmisch bestimmen:

Follow(A) wird für alle Nichtterminale A berechnet, indem die folgenden Regeln solange angewendet werden, bis keine Follow-Menge mehr vergrößert werden kann:

- für eine Produktion $B\texttt{::=}\alpha A \beta$ werden alle $\texttt{t}\in\mathit{First}(\beta)$ mit $\texttt{t}\neq \epsilon$ in *Follow(A)* aufgenommen.
- für eine Produktionen $B\texttt{::=}\alpha A$ füge *Follow(B)* zu der Menge *Follow(A)* hinzu.
- für eine Produktionen $B\texttt{::=}\alpha A \beta$ mit $\epsilon \in\mathit{First}(\beta)$ füge *Follow(B)* zu der Menge *Follow(A)* hinzu.

Mit Hilfe der beiden Mengen *First* und *Follow* wird eine besondere Klasse kontextfreier Grammatiken definiet:

Definition (LL(1) Grammatik) Eine kontextfreie Grammatik $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \mathcal{R})$ heißt LL(1)-Grammatik wenn die folgenden Bedingungen erfüllt sind:

- Für jedes $A \in \mathcal{N}$ mit $A \rightarrow \alpha$ und $A \rightarrow \beta$ mit $\alpha \neq \beta$ gilt: $\mathit{First}(\alpha) \cap \mathit{First}(\beta) = \emptyset$.
- für alle $A \in \mathcal{N}$ mit $A \Rightarrow \epsilon$ gilt: $\mathit{First}(A) \cap \mathit{Follow}(A) = \emptyset$.

Betrachten wir nochmal die Grammatik für arithmetische Ausdrücke mit den linksrekursiven Regeln:

- $\mathcal{T} = \{\mathit{number}, +, -, *, /\}$
- $\mathcal{N} = \{$*addExpr, multExpr, addOp, multOp*$\}$
- $S =$*addExpr*

- *addExpr* ::= *addExpr addOp multExpr*
  |*multExpr*

  multExpr ::= *multExpr multOp* `zahl`
  |`zahl`

  addOp ::= `+`|`-`
  multOp ::= `*`|`/`

Es lassen sich die *First*- und *Follow*-Mengen berechnen:

	First	*Follow*
addOp	`+`,`-`	`zahl`
multOp	`*`,`/`	`zahl`
addExpr	`zahl`	`+`,`-`
multExpr	`zahl`	`*`,`/`

Diese Grammatik ist keine LL(1)-Grammatik. Es gilt:
addExpr→*addExpr addOp multExpr*
addExpr→*multExpr*

Die First-Mengen der beiden rechten Seiten sind nicht disjunkt:*First(addExpr addOp multExpr)*= {`zahl`}
First(multExpr)= {`zahl`}.

Die Grammatik ist nicht LL(1).

LL(1)-Grammatiken haben eine Reihe schöner Eigenschaften:

- LL(1)-Grammatiken erlauben es bei einem rekursiv absteigenden Parser Sackgassen und Backtracking zu vermeiden.
- LL(1)-Grammatiken enthalten keine linksrekursiven Regeln.

Schieben und Reduzieren

Der rekursiv absteigende Parser geht vom Startsymbol aus und versucht durch Regelanwendung den in Frage stehenden Satz abzuleiten. Eine andere Strategie ist die des Schiebens-und-Reduzierens (shift-reduce). Diese geht vom im Frage stehenden Satz aus und versucht die Regeln rückwärts anzuwenden, bis das Startsymbol erreicht wurde. Hierzu wird der Satz von links nach rechts (oder symmetrisch von rechts nach links) betrachtet und versucht, rechte Seiten von Regeln zu finden und durch ihre linke Seite zu ersetzen. Dazu benötigt man einen Marker, der angibt, bis zu welchen Teil man den Satz betrachtet. Wenn links des Markers keine linke Seite einer Regel steht, so wird der Marker ein Zeichen weiter nach rechts verschoben.

Wir leiten im folgenden unserer allseits bekannten Beispielsatz aus dem Sonnensystem durch Schieben und Reduzieren ab. Als Marker benutzen wir einen Punkt.

		. a moon orbits mars
(shift)	→	a . moon orbits mars
(reduce)	→	*article* . moon orbits mars
(shift)	→	*article* moon . orbits mars
(reduce)	→	*article noun* . orbits mars

(reduce)	$\rightarrow$	*noun-phrase* . orbits mars
(shift)	$\rightarrow$	*noun-phrase* orbits . mars
(reduce)	$\rightarrow$	*noun-phrase verb* . mars
(shift)	$\rightarrow$	*noun-phrase verb* mars .
(reduce)	$\rightarrow$	*noun-phrase verb noun* .
(reduce)	$\rightarrow$	*noun-phrase verb noun-phrase* .
(reduce)	$\rightarrow$	*noun-phrase verb-phrase* .
(reduce)	$\rightarrow$	*start*.

Wir werden im Laufe dieser Vorlesung diese Parserstrategie nicht weiter verfolgen.

3.6 Zusammenfassung der Chomsky-Hierarchie

Wir haben in diesem Kapitel drei grundlegende verschiedene Arten von Grammatiken betrachtet, die drei unterschiedliche Klassen von Sprachen generieren. Die drei Arten von Grammatiken unterschieden sich in der syntaktischen Form der Regeln. Abbildung 3.7 ist noch einmal die sogenannte Chomsky-Hierarchie wie aus der theoretischen Informatik bekannt, zusammengefasst zu sehen.

Typ	Name	Produktionen	Spracherkennung
0	nicht eingeschränkt (Semi-Thue-System)	α`::=`β $\alpha,\beta \in (\mathcal{N} \cup \mathcal{T})^*$ $\alpha \neq \epsilon$	Turing-maschine
1	kontextsensitiv	α`::=`β mit $\mid \alpha \mid \leq \mid \beta \mid$ oder $\sigma A \tau$`::=`$\sigma\gamma\tau$ mit $\sigma,\tau \in (\mathcal{N} \cup \mathcal{T})^*$, A$\in \mathcal{N}^+, \gamma \in (\mathcal{N} \cup \mathcal{T})^+$	linear beschränkter Automat
2	kontextfrei	**kontextfreie Regeln:** A`::=`α $\alpha \in (\mathrm{T} \cup \mathrm{N})^*$	Kellerautomat, rekursiv absteigender Parser.
3	regulär	**lineare Produktionen:** A`::=a`B A`::=a`	endlicher Automat

Abbildung 3.7: Chomsky-Hierarchie

Kapitel 4

Implementierung von Parsern

In der Praxis werden größere Parserprojekte nicht vollständig von Hand kodiert sondern Tools zur Generierung des Parsers benutzt. Der wohl am weitesten verbreitete Parsergenerator ist Yacc[Joh75]. 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*. Die lexikalische Analyse steht im im Zusammenspiel mit Yacc und Bison jeweils auch ein Generatorprogramm zur Verfügung, das einen Tokenizer generiert. Dieses Programm heißt `lex`. Weitere modernere und auf Java zugeschnittene Parsergeneratoren sind: `yavacc`, `antlr` und `gentle`[Gru01]. Im Prinzip kann man davon ausgehen, daß es für fast alle Sprachen einen Parsergenerator gibt, so z.B. auch *Happy*[GM95] 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.

Wir werden uns in diesem Kapitel keines Parsergenerators bedienen, sondern eine C++-Bibliothek zur Konstruktion von rekursiv absteigenden Parsern entwickeln. Die dabei entstehenden Parser werden Backtracking durchführen, d.h. bei mehreren Regelalternativen für ein Nichtterminalzeichen, werden diese so lange ausprobiert bis eine Alternative nicht mehr in eine Sackgasse führt. Das hat den Vorteil, daß die Grammatik nicht die LL(1)-Eigenschaft zu haben braucht und wir auch nicht die First- und Follow-Mengen berechnen müssen. Das hat den Nachteil, daß der Parser Irrwege gehen wird und nicht direkt zum Parsergebnis führt.

4.1 Parserbibliothek

Wir entwickeln eine Parserbliblithek im Baukastenprinzip. Ein Parser wird ein Objekt sein. Für jedes Nichtterminalsymbol einer Grammatik wird ein Parserobjekt erzeugt. Ein Parserobjekt soll dabei direkt aus der Produktion der Grammatik abgeleitet werden. Eine Grammatikregel wird dabei als Bauvorschrift für den Parser betrachtet. Hierzu vergegenwärtigen wir uns zunächst der Syntax für Grammatikregeln. Es ist möglich eine kontextfreie Grammatik für die Sprache der Grammatikregeln anzugeben. Eine solche Grammatik ist in Abbildung 4.1 angegeben.

$$\begin{array}{rcll}
\mathit{rule} & \rightarrow & \mathit{lhs}\ \texttt{::=}\ \mathit{rhs} & (4.1)\\
\mathit{lhs} & \rightarrow & \texttt{name} & (4.2)\\
\mathit{rhs} & \rightarrow & \mathit{atom}\ \mid\ \mathit{seq}\ \mid\ \mathit{alt}\ \mid\ \texttt{(}\ \mathit{rhs}\ \texttt{)} & (4.3)\\
\mathit{atom} & \rightarrow & \texttt{name} & (4.4)\\
\mathit{seq} & \rightarrow & \mathit{rhs}\ \mathit{rhs} & (4.5)\\
\mathit{alt} & \rightarrow & \mathit{rhs}\ \texttt{'|'}\ \mathit{rhs} & (4.6)
\end{array}$$

Abbildung 4.1: Eine Grammatik für Grammtikregeln

Abgesehen von geklammerten Ausdrücken, erkennt man, daß es drei Möglichkeiten für die Definition der rechten Seite einer Produktion gibt. Wir werden diese drei Möglichkeiten mit drei Arten einen Parser zu definieren identifizieren:

- atomare Parser, die direkt den Parser für ein anderes Nichtterminalzeichen aufrufen, oder die ein Terminalzeichen erkennen.
- die Sequenz zweier Parser.
- die Alternative aus zwei Parsern.

Wir werden Methoden vorsehen, die es erlauben, zwei Parserobjekte zu einem neuen Parserobjekt zu kombinieren. Man spricht von sogenannten Parserkombinatoren. Hauptsächlich zwei Kombinatoren für Parser werden benötigt, um für eine Grammatik einen Parser im Baukastenprinzip zu definieren:

- ein Kombinator für die Sequenz: aus zwei Parserobjekten wird ein neues Parserobjekt erzeugt, das die nacheinander Ausführung der beiden Parser darstellt. Man spricht auch von der horizontalen Kombination. In der Grammatik wird die Sequenz durch horizontales Nacheinanderschreiben von zwei Symbolen dargstellt.
- ein Kombinator für die Alternative: aus zwei Parserobjekten wird ein neues Parserobjekt erzeugt, das erst den ersten Parser ausprobiert und wenn dieser Parser scheitert, es mit dem zweiten versucht. Hier wird also das Backtracking realisiert. Man spricht auch von vertikaler Kombination, weil die Regelalternativen für ein Nichtterminalzeichen in der Grammatik untereinander stehen.

Die Idee der Parserkombination stammt aus der funktionalen Programmierung. Ein frühes beeindruckendes Beispiel für Parserkombinatoren findet sich in [FL89], in dem ausführlich eine Grammatik für englischsprachige Sätze über das Sonnensystem modelliert und in der Sprache Miranda[Tur85] implementiert ist.

Wir bedienen uns Parserkombinatoren aus zwei Gründen:

- es lassen sich dabei viele interessante C++ Eigenschaften erlernen.
- unsere Sprache fab4, in der wir einen eigenen Compiler schreiben wollen ist funktional. In ihr läßt sich verblüffend einfach die Bibliothek der Parserkombinatoren implementieren.

4.1.1 Allgemeine Hilfsfunktionen zur Stringdarstellung

Bevor wir mit der Definition der eigentlichen Parserbibliothek beginnen, definieren wir ein kleines Hilfmodul, das uns erlaubt in C++ ähnlich wie in Java mit der `toString`-Methode zu arbeiten. Hierzu definieren wir ein Modul `Show`, in der es eine mehrfach überladene Methode `show` gibt, die eine textuelle Darstellung von Daten zurückgibt. Primitive Typen werden dabei direkt in Strings umgewandelt, Objekttypen werden aufgefordert ihre Methode `toString` zu benutzen.

Show.h

```
#ifndef __SHOW_H
#define __SHOW_H ;

#include <string>
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;
namespace name{namespace panitz{namespace util{

static string show(int x ){
  stringstream ss;
  string result;

  ss << x;
  ss >> result;
  return result;
}

static string show(char x ){
  stringstream ss;
  string result;

  ss << x;
  ss >> result;
  return result;
}

static string show(string  x){return "\""+x+"\"";}
static string show(string* x ){return show(*x);}

template <typename a>
std::string show(vector<a*>  xs){
  string result="[";
  for (int i=0;i<xs.size();i++) {
    if (i>0) result=result+",";
    result=result+show(xs[i]);
  }
  return result+"]";
}
```

```

template <typename a>
std::string show(vector<a>  xs){
  string result="[";
  for (int i=0;i<xs.size();i++) {
    if (i>0) result=result+",";
    result=result+show(xs[i]);
  }
  return result+"]";
}

template <typename X>
std::string show(vector<X>* x ){return show(*x);}

template <typename X>
std::string show(X* x ){return x->toString();}

template <typename X>
std::string show(X x ){return x.toString();}
}}}//end of namespace
#endif

```

Als Reminizenz an unseren Javacode haben wir einen hierarchischen Namensraum entsprechend der Pakete in Java benutzt.

4.1.2 Der Typ eines Parser

In diesem Abschnitt wird die Bibliothek `ParsLib` entwickelt. Die Bibliothek wird möglichst generisch gehalten und besteht hauptsächlich aus Schablonenklassen *(templates)*. Daher schreiben wir die komplette Bibliothek in eine Headerdatei. Nur so können wir sicher gehen, daß der benötigte konkrete Code für die instanziierten Schablonentypen auch vom Compiler generiert wird.

Wir inkludieren eine Reihe von Dateien aus unserem Namensraum `util`, die wir weiter unten definieren werden. Zusätzlich inkludieren wir ein paar Klassen aus der Standard Template Library:

ParsLib.h

```
#ifndef __PARSLIB_H
#define __PARSLIB_H

#include <string>
#include <sstream>
#include <iostream>
#include <vector>
#include "../util/Show.h"
#include "../util/Tuple.h"
#include "../util/Either.h"

```



```
using namespace std;
using namespace name::panitz::util;

namespace name {namespace panitz{ namespace parser{
```

Cache Objekte für konstante Funktionen

Bevor wir eine Klasse für Parser definieren, schreiben wir noch eine weitere Hilfsklasse: die Klasse `CAF`. `CAF` steht dabei für*constant applicative form*. Ein `CAF`-Objekt stellt eine nullstellige Funktion dar. Wenn dieses Funktionsobjekt das erste Mal aufgerufen wird, so speichert sich das Objekt das Ergebnis zwischen, um bei einem zweiten Aufruf den Wert nicht noch einmal berechnen zu müssen. `CAF`-Objekte haben also eine Art *cache* für eine nullstellige Funktion.

Wir halten die Klasse `CAF` generisch über den Typ des Funktionsergebnisses.

ParsLib.h

```
template <class resultType>
class CAF {
```

Als privates Attribut enthält die Klasse einen Funktionszeiger, auf die nullstellige Funktion.

ParsLib.h

```
  private:
    resultType(*f)();
```

Ein weiteres Feld wird vorgesehen, um das Ergebnis dieser Funktion zu cachen. der Anwendungsoperator `()` wird so überladen, daß, falls im Feld `result` schon ein Ergebnis zwischengespeichert ist, dieses direkt ausgegeben wird, ansonsten die Funktion `f` zunächst aufgerufen wird, ihr Ergebnis im Feld `result` gespeichert und ausgegeben wird.

ParsLib.h

```
  public:
    resultType result;

    resultType operator()(){
      if (result==NULL) result=f();
      return result;
    }
```

Wir sehen drei Konstruktoren und eine einfache Methode `toString` vor.

ParsLib.h

```
    CAF(){}
    CAF(resultType x):result(x){}
    CAF(resultType(*f)()){this->f=f;result=NULL;}

    virtual string toString(){return "CAF(..)";}
};
```

Um bequemer `CAF`-Objekte erzeugen zu können, sehen wir zwei entsprechende generische Methoden vor.

ParsLib.h
```
template <class resultType>
CAF<resultType> caf(resultType(*f)()){return CAF<resultType>(f);}

template <class resultType>
CAF<resultType> caf(resultType x){return CAF<resultType>(x);}
```

Ergebnis eines Parsvorgangs

Ein Parser ist eine Funktion, die versucht eine Folge von Token gemäß der Regeln für ein bestimmtes Zeichen zu zerlegen und für eine erfolgreiche Zerlegung ein bestimmtes Ergebnis zu erzeugen. Wir werden jetzt eine Klasse definieren, deren Objekte alle nötigen Informationen für das Ergebnis eines Parsvorgangs enthalten. Dieses wollen wir möglichst allgemein halten und greifen daher wieder auf eine Schablone zurück. Wir legen uns nicht fest, von welchem Typ die Token sind und von welchem Typ das Ergebnis ist.

ParsLib.h
```
template <typename resultType,typename tokenType>
class ParsResult{
```

Das Ergebnis eines Parsvorgangs enthält zwei entscheidene Informationen:

- zum einen das eigentliche Ergebnis des Parsvorgangs.
- zum anderen die Liste der Token, die noch nicht durch den Parser konsumiert worden, sprich die, die noch weiter zu parsen sind.

Für diese beiden Informationen sehen wir jeweils ein privates Feld vor. Für eine Tokenliste benutzen wir die Klasse `vector` aus der *standard template library*.

ParsLib.h
```
  private:
    resultType result;
    vector<tokenType*> furtherToken;
```

Wir sehen entsprechende Konstruktoren und Destruktoren vor:

ParsLib.h
```
  public:
    ParsResult(resultType result,vector<tokenType*>& furtherToken) {
      this->result=result;
      this->furtherToken=furtherToken;
      ParsResultCount=ParsResultCount+1;
    }
    ParsResult(vector<tokenType*>& furtherToken) {
      this->furtherToken=furtherToken;
      ParsResultCount=ParsResultCount+1;
    }
    ~ParsResult(){ParsResultCount=ParsResultCount-1;}
```


Für die beiden privaten Felder gibt es je eine `get`-Methode. Wir verzichten auf die `set`-Methoden. Unsere Objekte sind damit unveränderbar.

ParsLib.h
```
    resultType getResult(){return result;}
    vector<tokenType*>& getFurtherToken(){return furtherToken;}
```

Schließlich sehen wir noch eine Methode vor, die als bool'sches Flag angibt, ob der Parsvorgang, dessen Ergebnis das Objekt darstellt mißlungen ist. Standardmäßig ist das in dieser Klasse auf falsch gesetzt.

ParsLib.h
```
    virtual bool failed(){return false;}
```

Und schließlich bieten wir noch in Javamanier eine `toString`-Methode an.

ParsLib.h
```
    virtual string toString(){
      return "ParsResult("+show(result)
                       +","+show(getFurtherToken())+")";}
};

static int ParsResultCount=0;
```

Soweit die Klasse, die Ergebnisse von Parsvorgängen beschreibt. Wir sehen eine spezialisierte Form dieser Klasse vor, die einen mißlungenen Parsvorgang beschreibt. Darin werden die Methoden `failed` und `toString` entsprechend überschrieben.

ParsLib.h
```
template <typename resultType,typename tokenType>
class Fail:public ParsResult<resultType,tokenType>{
  public:
    Fail(vector<tokenType*>& furtherToken)
        :ParsResult<resultType,tokenType>(furtherToken){
    }
    virtual bool failed(){return true;}
    virtual string toString(){
     return "Fail("+show(getFurtherToken())+")";}
};
```

Da wir in der Oberklasse die Methoden als `virtual` markiert haben, erhalten wir das in der objektorientierten Programmierung gewünschte Verhalten der späten Bindung.

Die allgemeine Parser Oberklasse

Ein Parser läßt sich jetzt relativ einfach berschreiben: ein Parser ist ein Objekt, das eine Methode zum Parsen hat. Diese Methode konsumiert eine Tokenliste und erzeugt ein Parsergebnis-Objekt. Wir lassen wieder allgemein, von welchem Typ die Token sind und von welchem Typ das eigentlichen Ergebnis ist. Damit entsteht folgende kleine Schablonenklasse:

ParsLib.h
```
template <typename resultType,typename tokenType>
class Parser{
  public:
    virtual ParsResult<resultType,tokenType>*
                  parse(vector<tokenType*>& xs)=0;

    virtual ~Parser(){};
    virtual std::string toString()=0;
};
```

Parser zum Erkennen von Token

Wir werden jetzt verschiedene Arten von Parsern definieren. Diese werden dann alle Unterklassen der Klasse `Parser` sein. Einer der elementarsten Parser testet, ob das nächste Token in der Tokenliste ein bestimmtes Terminalzeichen ist. Hierfür definieren wir die Klasse `GetToken`. Objekte dieser Klasse sind Parser, deren Ergebnis ein erfolgreicher Parsvorgang ist, wenn das nächste Token dem Token, nach dem dieser Parser sucht, entspricht. Im Erfolgsfall ist das eigentliche Parsergebnis das erkannte Token.

Wir definieren eine Unterklasse von `Parser`:

ParsLib.h
```
template <typename tokenType>
class GetToken: public Parser<tokenType*,tokenType>{
```

Objekte dieser Klasse enthalten zwei Informationen in privaten Feldern:

- das Token, nach dem der Parser sucht.
- die Gleichheitsfunktion, mit der dieses Token verglichen wird.

ParsLib.h
```
  private:
     tokenType* token;
     bool(*eq)(tokenType*,tokenType*);
```

Ein Konstruktor steht zur Verfügung, der dieses Felder initialisiert:

ParsLib.h
```
  public:
    GetToken(tokenType* token,bool(*eq)(tokenType*,tokenType*))
     {this->token=token;this->eq=eq;}
```

Am interessantesten ist natürlich die Methode `parse`. Zunächst wird in dieser getestet, daß die Tokenliste noch nicht leer ist. Dann wird das nächste Token der Eingabeliste mit dem des Parsers verglichen. Hierzu wird die Vergleichsmethode des Parsers benutzt. Falls das alles erfolgreiche Test waren wird ein `ParsResult`-Objekt erzeugt. Die Liste der restlichen Token entsteht aus der ursprünglichen Liste durch Abtrennen des ersten Elements. Das erste Element wird dabei als eigentliches Parsergebnis benutzt. Schlägt einer der Tests fehl, so wird ein `Fail`-Objekt erzeugt. Da kein Token konsumiert wurde, bleibt die Eingabeliste unverändert.

ParsLib.h
```
    ParsResult<tokenType*,tokenType>* parse(vector<tokenType*>& xs){
      if (!xs.empty() && eq(token,xs[0])){
        tokenType* tok = xs[0];
        vector<tokenType*> restToken
         = vector<tokenType*>(xs.begin()+1,xs.end());
        ParsResult<tokenType*,tokenType>*result
          =new ParsResult<tokenType*,tokenType>(tok,restToken) ;
        return result;
      }
      return  new Fail<tokenType*,tokenType>(xs);
    }
```

Schließlich realisieren wir noch die Methode `toString`:

ParsLib.h
```
    virtual std::string toString(){
      return "GetToken("+show(token)+")";
    }
};
```

Objekte generische Klassen zu konstruieren ist etwas umständlich, weil in C++ (ebenso wie in Java) verlangt wird, daß die konkreten Typen der Schablonenvariablen im Konstruktornamen mit eckigen Klammern angegeben werden. Um uns hier das Tippen zu vereinfachen, schreiben wir noch eine generische Methode, die den Konstruktor von `GetToken` aufruft. Für generische Methoden inferiert der Typchecker die Typen der Schablonenvariablen:

ParsLib.h
```
template <typename tokenType>
Parser<tokenType*,tokenType>*
       getToken(tokenType* token,bool(*eq)(tokenType*,tokenType*)){
  return new GetToken<tokenType>(token,eq);}
```

Die ersten simplen Parser können damit geschrieben werden, nähmlich Parser, die genau ein Terminalzeichen erkennen. Zeit dieses in einem kleinen Test auszuprobieren. Hierzu definieren wir einen Parser über eine Liste von Strings. Wir versuchen als kleinen Test aus der Eingabeliste `[hallo,welt]` einmal das Token `hallo` und einmal das Token `welt` zu parsen.

TestTokenParser.cpp
```
#include "ParsLib.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

bool stringEq(string* x,string* y){return *x == *y;}

int main(){
```

```
  string hallo = "hallo";
  string welt  = "welt";
  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);

  Parser<string*,string>* getHallo = getToken(&hallo,stringEq);
  cout << "show(getHallo): " << show(getHallo) << endl;

  ParsResult<string*,string>* res1 = getHallo->parse(xs);
  cout << "show(res1): " << show(res1) << endl;

  Parser<string*,string>* getWelt = getToken(&welt,stringEq);
  cout << "show(getWelt): " << show(getWelt) << endl;

  ParsResult<string*,string>* res2 = getWelt->parse(xs);
  cout << "show(res2): " << show(res2) << endl;
}

```

Das Programm sollte folgende Ausgabe erzeugen:

```
sep@pc216-5:~/name/panitz/parser> g++ -o TestTokenParser TestTokenParser.cpp
sep@pc216-5:~/name/panitz/parser> ./TestTokenParser
show(getHallo): GetToken(hallo)
show(res1): ParsResult(hallo,[welt])
show(getWelt): GetToken(welt)
show(res2): Fail([hallo,welt])
sep@pc216-5:~/name/panitz/parser>
```

4.1.3 Sequenzen

Bisher haben wir nur Parser, die genau ein Token erkennen und zurückgeben können. In diesem Abschnitt definieren wir eine Klasse, die es ermöglicht aus zwei Parsern einen neuen zu kombinieren, der die Nacheinanderausführung der zwei Parser darstellt. Die Frage stellt sich dabei, was das Ergebnis des Sequenzparsers sein soll. Wir haben zwei Teilergebnisse. Am naheliegendsten ist es, zunächst ein Paar aus diesen zwei Teilergebnissen zu konstruieren. daher definieren wir uns zunächst eine allgemeine Klasse `Pair` zur Darstellung beliebiger Paare von Objekten. Die Klasse `Pair` entspricht dabei der gleichnamigen Javaklasse.[1]

Tuple.h

```
#ifndef __TUPLE_H
#define __TUPLE_H
#include <string>
#include "Show.h"
namespace name {namespace panitz{ namespace util{

```

[1] Das statische Feld `PairCount` benutzen wir nur, um später testen zu können, ob wir auch alle neu angelegten Paarobjekte aus dem Speicher wieder löschen.


```
static int PairCount = 0;

template <typename fstType,typename sndType>
class Pair{
  public:
    fstType fst;
    sndType snd;
    Pair( fstType c, sndType snd):fst(c),snd(snd){
      PairCount=PairCount+1;}

    virtual ~Pair(){PairCount=PairCount-1;};
    std::string toString(){
      return "("+show(fst)+","+show(snd)+")";
    }
};
}}}
#endif

```

Damit haben wir jetzt eine Klasse, die das Ergebnis zweier nacheinander ausgeführter Parser adequat darstellen kann. Wir können nun die Parserklasse der Sequenz definieren. Wir werden allerdings nicht direkt zwei Parser zu einen neuen Parser kombinieren, sonder zwei `CAF`-Objekte, die einen Parser zurückgeben zu einen neuen Parser kompinieren. Den Grund hierfür werden wir erkennen, wenn wir wir versuchen Parser für rekursive Grammatiken zu konstruieren.

Natürlich soll die Klasse der Sequenz auch möglichst allgemein gehalten werden. Diesmal gibt es drei Typen, die generisch sind:

- der Ergebnistyp des ersten Parsers.
- der Ergebnistyp des zweiten Parsers.
- der Tokentyp für beide Parser.

ParsLib.h

```
template
  <typename resultType1,typename resultType2,typename tokenType>
```

Der Sequenzparser ist ein Parser, dessen Ergebnis ein Paar aus den beiden Argumentparsern ist:

ParsLib.h

```
class Seq:public Parser<Pair<resultType1,resultType2>*,tokenType >{
```

Für die beiden zu kombinierenden Parser wird in dem `Seq`-Objekt je ein privates Feld bereit gehalten. Dabei ist zu beachten, daß nicht direkt die beiden Parser kombiniert werden, sondern `CAF`-Objekte, die diese zurückgeben:

ParsLib.h

```
  private:
     CAF<Parser<resultType1,tokenType>*> p1;
     CAF<Parser<resultType2,tokenType>*> p2;
```

Wie üblich sehen wir einen Konstruktor vor:

ParsLib.h
```
 public:
   Seq(CAF<Parser<resultType1,tokenType>*> p1
      ,CAF<Parser<resultType2,tokenType>*> p2){
     this->p1=p1;
     this->p2=p2;
   }
```

Schließlich ist die Methoden zum Parsen mit dem Sequenzobjekt zu implementieren. Das Ergebnis wird im Erfolgsfall ein Paar der beiden Teilergebnisse zurückgeben:

ParsLib.h
```
   virtual ParsResult<Pair<resultType1,resultType2>*,tokenType>*
                                   parse(vector<tokenType*>& xs){
```

Als erstes wenden wir einmal den ersten Parser auf die Tokenliste an:

ParsLib.h
```
     ParsResult<resultType1,tokenType>* res1 = p1()->parse(xs);
```

Nur wenn der erste Parser nicht gescheitert ist, wird mit dem zweiten Parser weitergearbeitet. Wir sichern uns vom ersten Parsergebnis das eigentliche Ergebnis und die Liste der Token, mit denen weitergearbeitet wird. Auf diese wird der zweite Parser angewendet:

ParsLib.h
```
    if (!res1->failed()) {
       vector<tokenType*> further = res1->getFurtherToken();
       resultType1 r1 = res1->getResult();

       ParsResult<resultType2,tokenType>* res2
                                  = p2()->parse(further);
```

Auch der zweite Parser muß erfolgreich angewendet worden sein:

ParsLib.h
```
       if (!res2->failed()){
         resultType2 r2 = res2->getResult();
         vector<tokenType*> further = res2->getFurtherToken();
```

Jetzt haben wir die zwei Teilergebnisse, mit denen wir das neue Gesamtergebnis erzeugen können. Um nicht in Müll zu ersticken sind die temporären Teilergebnisse aus dem Speicher zu löschen.

ParsLib.h
```
         delete res1;
         delete res2;
         return
         new ParsResult<Pair<resultType1,resultType2>*,tokenType>
```



```
             (new Pair<resultType1,resultType2>(r1,r2),further);
         }
         delete res2;
       }
```

Sollte einer der beiden Parser fehlgeschlagen sein, so ist als Gesamtergebnis auch ein entsprechendes `ParsResult`-Objekt des Typs `Fail` zu generieren:

ParsLib.h
```
       delete res1;
       return new Fail<Pair<resultType1,resultType2>*,tokenType>(xs);
     }
```

Schließlich bieten wir noch eine einfache Methode `toString` in der Klasse `Seq` an

ParsLib.h
```
    virtual std::string toString(){
        return "Seq(..)";
    }
};
```

Ebenso wie für die anderen generischen Klassen bieten wir auch eine statische Methode, die lediglich den Konstruktor aufruft an.

ParsLib.h
```
template <typename a,typename b,typename c>
Parser<Pair<a,b>*,c>* seq(CAF<Parser<a,c>*>p1,CAF<Parser<b,c>*> p2){
  return ((Parser<Pair<a,b>*,c>*)new Seq<a,b,c>(p1,p2));
}
```

Eine attraktive Eigenschaft von C++ ist, die Möglichkeit Operatoren zu überladen. Hiervon machen wir Gebrauch. Wir überladen die Operatoren `-` und den Kommaoperator `,`, so daß diese jeweils zwei Parser zu einen Sequenzparser kombinieren:

ParsLib.h
```
template <typename a,typename b,typename c>
Parser<Pair<a,b>*,c>*
    operator-(CAF<Parser<a,c>*> p1,CAF<Parser<b,c>*> p2){
  return seq(p1,p2);
}

template <typename a,typename b,typename c>
CAF<Parser<Pair<a,b>*,c>*>
  operator,(CAF<Parser<a,c>*> p1,CAF<Parser<b,c>*> p2){
  return caf(seq(p1,p2));
}
```

Zeit für einen kleinen Test. Wir wollen einen Parser für die folgende kleine Grammatik definieren:

S::= `hallo welt`

Es soll also genau die Sequenz aus zwei Token erkannt werden. Hierzu definieren wir uns zunächst die beiden Tokenparser:

TestSeq.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

bool stringEq(string* x,string* y){return *x == *y;}

int main(){
  string hallo = "hallo";
  string welt  = "welt";

  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);

  Parser<string*,string>* getHallo = getToken(&hallo,stringEq);
  Parser<string*,string>* getWelt = getToken(&welt,stringEq);
```

Und schließlich den Sequenzparser, den wir einemal erfolgreich und einmal vergeblich auf eine Tokenliste anwenden:

TestSeq.cpp

```
  Parser<Pair<string*,string*>*,string>* getHW
   = caf(getHallo)-caf(getWelt);

  cout << "show(getHW): " << show(getHW) << endl;

  ParsResult<Pair<string*,string*>*,string>* res1 =getHW->parse(xs);
  cout << "show(res1): " << show(res1) << endl;

  vector<string*> ys;
  ys.insert(ys.end(),&hallo);
  ys.insert(ys.end(),&hallo);

  ParsResult<Pair<string*,string*>*,string>* res2 =getHW->parse(ys);
  cout << "show(res2): " << show(res2) << endl;
}

```

Das Programm sollte die folgende Ausgabe erzeugen:


```
sep@pc216-5:~/fh/compiler/tutor/bin> ./TestSeq
show(getHW): Seq(..)
show(res1): ParsResult((hallo,welt),[])
show(res2): Fail([hallo,hallo])
sep@pc216-5:~/fh/compiler/tutor/bin>
```

4.1.4 Alternativen

Die zweite Kombination von Parsern stellt die Altenative dar. Dabei werden zwei Parser kombiniert zu einen neuen Parser, der erst den einen und dann den Parser versucht anzuwenden. In der Grammatik drücken sich Alternativen durch mhrfache Regeln für ein Nichtterminal bzw. durch einen vertikalen Strich `|` aus.

Ähnlich wie für die Sequenz, brauchen wir auch für die Alternative eine Klasse, deren Objekte das Ergebnis der Alternative darstellen. Bei der Sequenz war dieses eine Paar mit den beiden Teilergebnissen. Bei der Alternative, brauchen wir die Möglichkeit auszudrücken, daß wir entweder das Ergebnis des einen Parsers oder des anderen Parsers haben. Genau eine solche Situation läßt sich gut in objektorientierter Weise darstellen. Hierzu benutzen wir die Klasse `Either`. Die Klasse `Either` hat zwei Unterklassen: die eine um das Ergebnis des ersten, die andere um das Ergebnis des zweiten Parsers darzustellen. Die beiden Unterklassen heißen `Left` und `Right`.

Natürlich halten wir auch die Klasse `Either` generisch über die beiden Typen der Teilergebnisse:

Either.h

```
#ifndef __EITHER_H
#define __EITHER_H

#include <string>
#include "Show.h"

using namespace std;

namespace name {namespace panitz{ namespace util{

template <typename leftType,typename rightType>
class Either{
```

Wir sehen für die Klasse lediglich die Methode `toString` und eine Methode, die testet ob es sich um ein `Left`-Objekt handelt vor:

Either.h

```
  public:
    virtual string toString(){return "Either()";}
    virtual bool isLeft(){return false;}
};
```

Die Unterklasse `Left` enthält ein Feld das linke Teilergebnis zu speichern:

Either.h

```
template <typename leftType,typename rightType>
class Left:public Either<leftType,rightType>{
```

```
  public:
    leftType left;
    Left( leftType left){this->left=left;}
    bool isLeft(){return true;}
    string toString(){return "Left("+show(left)+")";}
};
```

Entsprechend sei die zweite Unterklasse, `Right` definiert:

Either.h
```
template <typename leftType,typename rightType>
class Right:public Either<leftType,rightType>{
  public:
    rightType right;
    Right( rightType right){this->right=right;}
    string toString(){return "Right("+show(right)+")";}
};
```

Wenn der linke und der rechte Typ identisch sind, dann ist in der Regel auch egal, ob es sich um das linke oder das rechte Objekt handelt. Dann können wir es aus dem `Either`-Objekt extrahieren. Für diesen Zweck sei die folgende kleine generische methode definiert:

Either.h
```
template <typename a>
a getLeftRight(Either<a,a>* either){
  a result;
  if (either->isLeft()){result=((Left<a,a>*)either)->left;}
  else {result=((Right<a,a>*)either)->right;}
  delete either;
  return result;
}
}}}
#endif

```

Jetzt wo wir eine Klasse zum Ausdrücken des Ergebnisse zweier Regelalternativen haben, können wir ähnlich der Klasse `Seq` die Klasse `Alt` zur Darstellung der Regelalternativen schreiben. Diese stellt einen Parser mit einem `Either`-Objekt als Ergebnis dar:

ParsLib.h
```
template
 <typename resultType1,typename resultType2,typename tokenType>
class Alt:public Parser<Either<resultType1,resultType2>*,tokenType>{
```

Wie in `Seq` haben wir auch zwei private Felder, in denen die zu kombinierenden Teilparser gespeichert sind. Wie in `Seq` werden dieses nicht die Parser direkt, sondern `CAF`-Objekte sein, die diese Parser als Ergebnis liefern:

ParsLib.h
```
  private:
     CAF<Parser<resultType1,tokenType>*> p1;
     CAF<Parser<resultType2,tokenType>*> p2;
```

Ein normaler Konstruktor zur Initialisierung steht zur Verfügung:

ParsLib.h
```
  public:
    Alt( CAF<Parser<resultType1,tokenType>*> p1
       , CAF<Parser<resultType2,tokenType>*> p2){
      this->p1=p1;
      this->p2=p2;
    }
```

Es verbleibt das Kernstück, die Methode zum Parsen:

ParsLib.h
```
    virtual ParsResult<Either<resultType1,resultType2>*,tokenType>*
                                     parse(vector<tokenType*>& xs){
```

Zunächst einmal versuchen wir den linken der beiden Parser anzuwenden:

ParsLib.h
```
      ParsResult<resultType1,tokenType>* res1 = p1()->parse(xs);
```

Wenn der erste Parser erfolgreich war, so sind wir schon zufrieden und erzeugen ein Ergebnisobjekt für diesen. Hierzu nehmen wir das Ergebnis des ersten Parsers und bauen für das Ergebnis des `Alt`-Parsers ein `Left`-Objekt. Das zwischenzeitliche `ParsResult`-Objekt muß im Speichern wieder entfernt werden. Wir werden es nicht mehr brauchen.

ParsLib.h
```
      if (!res1->failed()) {
        vector<tokenType*> further = res1->getFurtherToken();
        resultType1 r1 = res1->getResult();
        delete res1;
        return
        new ParsResult<Either<resultType1,resultType2>*,tokenType>
           (new Left<resultType1,resultType2>(r1),further);
      }
```

Der zweite Fall ist, daß der erste Parser fehlgeschlagen ist. Dann löschen wir dieses `Fail`-Objekt und wenden die zweiten (rechten) Parser an. Sollte dieser erfolgreich engewendet werden können, so wird als Gesamtergebnis ein `Right`-Objekt erzeugt. An dieser Stelle findet das *backtracking* statt. Wir ignorieren die von dem ersten Parser eventuell bereits konsumierten Token, sondern setzen den zweiten Parser frisch auf die Eingabetokenliste an.

ParsLib.h

```
      delete res1;
      ParsResult<resultType2,tokenType>* res2 = p2()->parse(xs);
      if (!res2->failed()) {
        resultType2 r2 = res2->getResult();
        vector<tokenType*> further = res2->getFurtherToken();
        delete res2;
        return
        new ParsResult<Either<resultType1,resultType2>*,tokenType>
           (new Right<resultType1,resultType2>(r2),further);
      }
```

Wenn keiner der beiden Parser zum Erfolg geführt hat, so sird als Gesamtergebnis ein `Fail`-Objekt erzeugt. man beachte, daß auch in diesem `Fail`-Objekt in den Typen enthalten ist, was denn das Ergebnis gewesen wäre, wenn es kein `Fail` gewesen wäre.

ParsLib.h

```
      delete res2;
      return
       new Fail<Either<resultType1,resultType2>*,tokenType>(xs);
    }
```

Eine kleine wenig aussagekräftige `toString`-Methode, schließe die Klasse ab:

ParsLib.h

```
    virtual std::string toString(){
        return "Alt(..)";
    }
};
```

Auch für die Klass `Alt` sei eine generische Methode vorgesehen, die wir bequemer statt des Konstruktors benutzen können:

ParsLib.h

```
template <typename a,typename b,typename c>
Parser<Either<a,b>*,c>*
  alt(CAF<Parser<a,c>*>p1,CAF<Parser<b,c>*> p2){
  return ((Parser<Either<a,b>*,c>*)new Alt<a,b,c>(p1,p2));
}
```

Und auch hier bedienen wir uns der netten Eigenschaft, daß Operatoren in C++ überlden werden können. Der Oder-Operator `|`, den wir auch in der Grammatik benutzen, wird zu diesen Zweck überladen:

ParsLib.h

```
template <typename a,typename b,typename c>
Parser<Either<a,b>*,c>*
    operator|(CAF<Parser<a,c>*> p1,CAF<Parser<b,c>*> p2){
  return alt(p1,p2);
}
```


Es wird mal wieder Zeit für einen kleinen Test. Diesmal definieren wir eine Parser, der entweder das Wort `hallo` oder das Wort `welt` erkennt, also für die Grammatik: S`::= hallo | welt`.

TestAlt.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

bool stringEq(string* x,string* y){return *x == *y;}

int main(){
  string hallo  = "hallo";
  string welt   = "welt";
  string world  = "world";

  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);

  Parser<string*,string>* getHallo = getToken(&hallo,stringEq);
  Parser<string*,string>* getWelt = getToken(&welt,stringEq);
```

Der Parser der Alternative läßt sich jetzt syntaktisch fast genauso Definieren, wie wir es aus der Grammatik gewohnt sind:

TestAlt.cpp

```
  Parser<Either<string*,string*>*,string>* getHW
   = caf(getHallo)|caf(getWelt);

  cout << "show(getHW): " << show(getHW) << endl;

  typedef  ParsResult<Either<string*,string*>*,string>* Result;
  Result res1 = getHW->parse(xs);
  cout << "show(res1): " << show(res1) << endl;

  vector<string*> ys;
  ys.insert(ys.end(),&world);

  Result res2 = getHW->parse(ys);
  cout << "show(res2): " << show(res2) << endl;
}
```

Um lange und komplizierte Instanzen von generischen Typen besser zu benutzen, haben wir uns der Möglichkeit von Typsynonymen bedient und eine entsprechende `typedef`-Definition eingeführt. Das Programm sollte die folgende Ausgabe haben:

```
sep@pc216-5:~/fh/compiler/tutor/bin> ./TestAlt
```

```
show(getHW): Alt(..)
show(res1): ParsResult(Left(hallo),[welt])
show(res2): Fail([world])
sep@pc216-5:~/fh/compiler/tutor/bin>
```

4.1.5 Ergebnis Manipulieren

Theoretische können wir jetzt alles was in einer kontextfreien Grammatik beschrieben wird mit unserer Parserbibliothek ausdrücken. Es gibt im Prinzip nur die zwei Bildungsprinzipien der Sequenz und der Alternative, um die definierende rechte Seite für ein Nichtterminalzeichen zu beschreiben. Allerdings sind unsere allgemeinen Ergebnisse für diese Bildungsprinzipien recht unhandlich. Wir bekommen Paar- oder `Either`-Objekte. Im konkreten Fall wollen wir diese gerne zu eine für unsere Anwendung aussagekräftigen Ergebis verändern. Hierzu sehen wir eine weitere Klasse vor, die einen Parser kapselt, die Klasse `Map`. In dieser werden ein Parser und eine Funktion zusammengefasst. `Map`-Objekte sind wieder Parser. Sie wenden erst den inneren Parser an, um anschließend auf das darin erhaltene Ergebnis die Funktion anzuwenden, um ein neues Ergebnisobjekt zu erhalten. Die Klasse `Map` ist natürlich wieder generisch gehalten. Dieses Mal über den Ergebnistyp des inneren Parsers und über den Ergebnistyp des entstehenden Parsers (sowie natürlich auch wieer über den Tokentyp).

ParsLib.h
```
template <typename X,typename Y,typename tokenType>
class Map:public Parser<Y,tokenType>{
```

`Map`-Objekte haben ein Feld zum Speichern des inneren Parsers, sowie ein Feld, um den Funktionszeiger, der auf das Ergebnis anzuwendenen Funktion zu speichern:

ParsLib.h
```
  private:
    Parser<X,tokenType>* p;
    Y(*f)(X);
```

Entsprechend sei auch der Konstruktor definiert

ParsLib.h
```
  public:
    Map(Y(*f)(X),Parser<X,tokenType>* p):f(f),p(p){}
```

Die Methode zum Parsen ist jetzt relativ simpel.

ParsLib.h
```
    virtual ParsResult<Y,tokenType>* parse(vector<tokenType*>& xs){
```

Wende zunächst den inneren Parser an:

ParsLib.h
```
     ParsResult<X,tokenType>* res1 = p->parse(xs);
```

Im Erfolgsfall hole dessen Ergebisbestandteile:

ParsLib.h
```
      if (!res1->failed()){
        X r1 = res1->getResult();
        vector<tokenType*> further = res1->getFurtherToken();
        delete res1;
```

Und erzeuge das Gesamtergebnis, um auf dem inneren Ergebis die Funktion anzuwenden:

ParsLib.h
```
        return new ParsResult<Y,tokenType>((*f)(r1),further);
      }
```

Ansonsten wird ein adequates `Fail`-Objekt für das Gesamtergebnis erzeugt:

ParsLib.h
```
      delete res1;
      return new Fail<Y,tokenType>(xs);
    }
```

Wie üblich sei eine primitive Version der Methode `toString` überschrieben.

ParsLib.h
```
    virtual std::string toString(){
        return "Map(..)";
    }
};
```

Auch für die Klasse `Map` schreiben wir eine Funktion, die den Konstruktor aufruft.

ParsLib.h
```
template <typename a,typename b,typename c>
Parser<b,c>* map(CAF<Parser<a,c>*> p,  b(*f)(a)){
  return ((Parser<b,c>*)new Map<a,b,c>(f,p()));
}
```

Und auch in diesen Fall können wir Operatoren überladen. Wir benutzen jetzt die Operatoren `<` und `<<`, um auszudrücken, daß auf das Parsergebnis noch eine Funktion anzuwenden ist. Im Gegensatz zu `<` kapselt `<<` den entstehenden Parser noch als ein `CAF`-Objekt.

ParsLib.h
```
template <typename a,typename b,typename c>
Parser<b,c>* operator<(CAF<Parser<a,c>*> p,  b(*f)(a)){
  return map(p,f);
}

template <typename a,typename b,typename c>
CAF<Parser<b,c>*> operator<<(CAF<Parser<a,c>*> p,  b(*f)(a)){
  return caf(map(p,f));
}
```

Mit den `Map`-Parsern können wir jetzt auch eine Alternativkombination von Parsern, die beide den gleichen Ergebnistyp haben definieren, in der wir das `Either`-Objekt bereits ausgepackt haben. Hierzu definieren wir die Funktion `ualt` (für *uniforme Alternative*). In dieser wird zunächst eine Alternative aus zwei parser gebildet und dann mit der Funktion `getLeftRight` das in einem `Left`- oder `Right`-Objekt gekapselte Ergebnis extrahiert.

ParsLib.h

```
template <typename a,typename c>
Parser<a,c>* ualt(CAF<Parser<a,c>*> p1,CAF<Parser<a,c>*> p2){
  return map<Either<a,a>*,a,c>(alt(p1,p2),getLeftRight<a>);
}
```

Als Operator für die Funktion `ualt` benutzen wir den doppelten vertikalen Strich `||`.

ParsLib.h

```
template <typename a,typename c>
CAF<Parser<a,c>*>
     operator||(CAF<Parser<a,c>*> p1,CAF<Parser<a,c>*> p2){
  return caf(ualt(p1,p2));
}
```

Wieder einmal Zeit für einen ersten Test. Im folgenden Test wird das Wort `hallo` erkannt, und auf das Ergebnis die Funktion `length` angewendet, die Länge des Ergebnisstrings zurückgiebt.

TestMap.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>

int length(string* x){return x->length();}
bool stringEq(string* x,string* y){return *x == *y;}

int main(){
  using namespace name::panitz::parser;
  using namespace std;

  string hallo = "hallo";
  string welt  = "welt";

  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);

  CAF<Parser<string*,string>*> getHallo
   = caf(getToken(&hallo,stringEq));

  Parser<int,string>* countHello  = getHallo < length;

  cout<<"show(countHello): "<< show(countHello) <<endl;

```



```
  ParsResult<int,string>* res1 = countHello->parse(xs);
  cout<<"show(res1): "<< show(res1) <<endl;

  ParsResult<int,string>* res2
    = countHello->parse(res1->getFurtherToken());
  cout<<"show(res2): "<< show(res2) <<endl;
}

```

Das Programm sollte folgende Ausgabe haben:

```
sep@pc216-5:~/fh/compiler/tutor> bin/TestMap
show(countHello): Map(..)
show(res1): ParsResult(5,[welt])
show(res2): Fail([welt])
sep@pc216-5:~/fh/compiler/tutor>
```

4.1.6 Rekursive Parser

Grammatiken sind erst interessant, wenn sie rekursiv sind. Auf naive Weise können wir unsere Parserkombinatoren in C++ leider nicht benutzen, um rekursive Regeln einer Grammatik umzusetzen. Betrachten wir hierzu die folgende kleine Grammatik:

A`::= hallo` A`|welt`

Es soll also erst eine Folge des Terminals `hallo` und anschließend das Terminal `welt` erkannt werden. Die linke Seite der Produktion besteht lediglich aus Sequenz und Alternative, eigentlich zwei Bildungskonstrukte, für die wir Parser konstruieren können. Eine naive Umsetzung dieser Grammatik sieht demnach wie folgt aus:

zunächst definieren wir Funktionen zum Zusammensetzen der Teilergebnisse und Stringvariablen für die zu parsenden Token:

NaiveRecursion.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

bool stringEq(string* x,string* y){return *x == *y;}
vector<string*>* singleton(string* x){
  vector<string*>* result= new vector<string*>();
  result->insert(result->end(),x);
  return result;
}

vector<string*>* insert(Pair<string*,vector<string*>*>* p){
  vector<string*>* result = p->snd;
  result->insert(result->begin(),p->fst);
  delete p;
```

```
  return result;
}

string hallo = "hallo";
string welt  = "welt";
```

Wir versuchen damit die Grammatik direkt umzusetzen. Dazu werden zwei Parser zum Erkennen der Token und einer für die rekursive Regel definiert:

NaiveRecursion.cpp
```
CAF<Parser<string*,string>*>getHallo=caf(getToken(&hallo,stringEq));
CAF<Parser<string*,string>*>getWelt =caf(getToken(&welt,stringEq));

CAF<Parser<vector<string*>*,string>*> getHallos
    =    (getHallo,getHallos)    << insert
       || getWelt                << singleton;
```

Betrachten wir schließlich einen Test für diesen Parser.

NaiveRecursion.cpp
```
int main(){
  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);

  cout<<"show(getHallos): "<< show(getHallos) <<endl;

  ParsResult<vector<string*>*,string>* res1 =getHallos()->parse(xs);
  cout<<"show(res1): "<< show(res1) <<endl;

  ParsResult<vector<string*>*,string>* res2
    = getHallos()->parse(res1->getFurtherToken());
  cout<<"show(res2): "<< show(res2) <<endl;
}

```

Dieser Parser funktioniert leider nicht und bricht mit einem Fehler ab.

```
sep@pc216-5:~/fh/compiler/tutor> bin/NaiveRecursion
show(getHallos): CAF(..)
Speicherzugriffsfehler
sep@pc216-5:~/fh/compiler/tutor>
```


Der Grund liegt natürlich darin, daß in der direkten Umsetzung der Regel:

A`::= hallo` A
`|welt`

In den Code

```
CAF<Parser<vector<string*>*,string>*> getHallos
    =    (getHallo,getHallos)    << insert
       || getWelt                << singleton;
```

Die Variable `getHallos` auf der rechten Seite der Zuweisung bereits benutzt wird, während sie noch undefiniert ist. Die Lösung dieses Problems geht über Funktionsobjekte. Jetzt erkennen wir, warum wir die Klasse `CAF` definiert machen. Mit ihr können wir nullstellige memorisierende Funktionsobjekte realisieren. Die rekursive Regel läßt sich dann durch folgenden Code ausdrücken:

```
Parser<vector<string*>*,string>* getHallosFunction();
CAF<Parser<vector<string*>*,string>*> getHallos
 = caf(getHallosFunction);

Parser<vector<string*>*,string>* getHallosFunction() {
     return
         (   (getHallo,getHallos)             <<insertThis
           || getWelt                          <<singleton)();
}
```

Insgesamt erhalten wir für die kleine rekursive Grammatik folgende Parserdefinition:

CorrectRecursion.cpp
```
#include "ParsLib.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

bool stringEq(string* x,string* y){return *x == *y;}

vector<string*>* results= new vector<string*>();

vector<string*>* singleton(string* x){
  results->insert(results->end(),x);
  return results;
}

vector<string*>* insertThis(Pair<string*,vector<string*>*>* p){
  vector<string*>* result = p->snd;
  result->insert(result->begin(),p->fst);
  delete p;
```

```
  return result;
}

string hallo = "hallo";
string welt  = "welt";
vector<string*> xs;

CAF<Parser<string*,string>*>getHallo=caf(getToken(&hallo,stringEq));
CAF<Parser<string*,string>*>getWelt =caf(getToken(&welt,stringEq));


Parser<vector<string*>*,string>* getHallosFunction();
CAF<Parser<vector<string*>*,string>*> getHallos
 = caf(getHallosFunction);

Parser<vector<string*>*,string>* getHallosFunction() {
     return
         (   (getHallo,getHallos)             <<insertThis
           || getWelt                         <<singleton)();
}

int main(){
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);

  cout<<"show(getHallos): "<< show(getHallos()) <<endl;

  ParsResult<vector<string*>*,string>* res1
     =getHallos()->parse(xs);
  cout <<show(res1)<<endl;;

  ParsResult<vector<string*>*,string>* res2
    = getHallos()->parse(res1->getFurtherToken());
  cout<<"show(res2): "<< show(res2) <<endl;
}
```

Und tatsächlich funktioniert der Parser wie gewünscht.

```
sep@pc216-5:~/fh/compiler/tutor> bin/CorrectRecursion
show(getHallos): Map(..)
ParsResult([hallo,hallo,hallo,hallo,hallo,hallo,hallo,welt],[hallo])
show(res2): Fail([hallo])
```



```
sep@pc216-5:~/fh/compiler/tutor>
```

4.1.7 Wiederholungen

Eigentlich könnnten wir mit den bisher vorgestellten Parserkombinatoren zufrieden sein. Wie schon im letzten Beispiel zu sehen war, lassen sich damit Parser für eine Wieerholung von Zeichen durch eine rekursive Regel ausdrücken. In Programmiersprachen sind solche Konstrukte sehr häufig. Daher lohnt es sich eigene Parserklassen für die n-fache Wiederholung eines Parser bereit zu stellen. im folgenden wird die Klasse `Repetition0`, die die 0 bis n-fache Wiederholung eines Parsers darstellt, definiert. Ihr Ergebnistyp ist ein Vektor des Ergebnistyps des Eingabeparsers:

ParsLib.h

```
template <typename resultType,typename tokenType>
class Repetition0:public Parser<vector<resultType>*,tokenType>{
  private:
    CAF<Parser<resultType,tokenType>*> p;

  public:
    Repetition0(CAF<Parser<resultType,tokenType>*> p):p(p){}
```

In der `parse`-Methode wird in einer `while`-Schleife sol lange der übergebene innere Parser auf die Tokenliste angewendet, bis er schließlich mit einem `Fail`-Objekt endet.

ParsLib.h

```
    virtual ParsResult<vector<resultType>*,tokenType>*
                                   parse(vector<tokenType*>& xs){
      vector<tokenType*>ys= xs;
      vector<resultType>* results = new vector<resultType>();
      ParsResult<resultType,tokenType>* res = p()->parse(ys);

      while (!res->failed()){
        results->insert(results->end(),res->getResult());
        ys = res->getFurtherToken();
        delete res;
        res=p()->parse(ys);
      }
      vector<tokenType*> further=res->getFurtherToken();
      delete res;

      return new ParsResult<vector<resultType>*,tokenType>
                                          (results,further);
    }

    CAF<Parser<resultType,tokenType>*> getP(){return p;}

    virtual std::string toString(){
      return "Repetition0("+show(p)+")";
```

```
    }
};
```

Aus dem Parser der 0 bis n-fachen Wiederholung läßt sich relativ einfach, über den Sequenzoperator ein Parser der 1 bis n-fachen Wiederholung ableiten.

ParsLib.h

```
template <typename resultType>
vector<resultType>*
          insertFirst(Pair<resultType,vector<resultType>*>* p){
  vector<resultType>* result = p->snd;
  result->insert(result->begin(),p->fst);
  delete p;
  return result;
}

template <typename resultType,typename tokenType>
class Repetition1:public Repetition0<resultType,tokenType>{
  public:
    Repetition1(CAF<Parser<resultType,tokenType>*> p)
      :Repetition0<resultType,tokenType>(p){}

    virtual ParsResult<vector<resultType>*,tokenType>*
                                parse(vector<tokenType*>& xs){
      Parser<vector<resultType>*,tokenType>* rep0
        = new Repetition0<resultType,tokenType>(getP());

      Parser<vector<resultType>*,tokenType>* repP
       =  (getP(),caf(rep0)) < (insertFirst<resultType>);
      return repP->parse(xs);
    }

    virtual std::string toString(){
      return "Repetition1("+show(getP())+")";
    }
};
```

Zur bequemeren Benutzbarkeit seien auch wieder Funktionen, die den Konstruktor aufrufen, definiert.

ParsLib.h

```
template <typename a,typename b>
CAF<Parser<vector<a>*,b>*> rep0(CAF<Parser<a,b>*> p){
  return caf((Parser<vector<a>*,b>*)new Repetition0<a,b>(p));
}

template <typename a,typename b>
CAF<Parser<vector<a>*,b>*> rep1(CAF<Parser<a,b>*> p){
  return caf((Parser<vector<a>*,b>*)new Repetition1<a,b>(p));
}
```


Auch hierfür sei ein kleiner erster Test gegeben:

TestRepetition.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>
bool stringEq(string* x,string* y){return *x == *y;}

int main(){
  using namespace name::panitz::parser;
  using namespace std;

  string hallo = "hallo";
  string welt  = "welt";

  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);

  Parser<string*,string>* getHallo = getToken(&hallo,stringEq);
  Parser<string*,string>* getWelt  = getToken(&welt,stringEq);

  Parser<vector<string*>*,string>* repHallo
   = new Repetition1<string*,string>(getHallo);

  ParsResult<vector<string*>*,string>* res1 = repHallo->parse(xs);
  cout<<"show(res1):" << show(res1)<<endl;

  vector<string*> further = res1->getFurtherToken();
  ParsResult<string*,string>* res2 = getWelt->parse(further);
  cout<<"show(res2):" << show(res2)<<endl;

  ParsResult<string*,string>* res3 = getHallo->parse(further);
  cout<<"show(res3):" << show(res3)<<endl;
}
```

Das Programm hat die folgende zu erwartende Ausgabe:

```
sep@pc216-5:~/fh/compiler> tutor/bin/TestRepetition
show(res1):ParsResult([hallo,hallo,hallo,hallo,hallo,hallo,hallo],[welt])
show(res2):ParsResult(welt,[])
show(res3):Fail([welt])
sep@pc216-5:~/fh/compiler>
```

4.1.8 Separierte Listen

Ein weiteres in Programmiersprachen sehr häufig vorkommendes Schema sind seperierte Listen. Dieses sind Wiederholungen bestimmter grammatischer Elemente, zwischen denen sich ein Separator befindet. Oft trägt der Separator nicht zur Semantik bei, sondern dient lediglich zum Trennen zwischen bestimmten Teilen. Ein typisches Beispiel hierfür sind die Argumentlisten von Funktionen in C, die durch Kommas getrennt sind. Ähnlich wie für Wiederholungen können wir für separarierte Liste einen entsprechenden Parser vorsehen. Auch hier wieder in zwei Varianten, einmal wenn die nullfache Wiederholung erlaubt ist, einmal wenn mindestens ein Element der Liste vorhanden sein muß.

Die entsprechenden Parserklassen haben einen weiteren Typ, über den sie generisch gehalten sind: den Typ des Separators.

ParsLib.h

```
template <typename resultType
         ,typename seperatorType
         ,typename tokenType>
class SeperatedBy:public Parser<vector<resultType>*,tokenType>{
  private:
    CAF<Parser<resultType,tokenType>*> p;
    CAF<Parser<seperatorType,tokenType>*> sep;
  public:
    SeperatedBy(CAF<Parser<resultType,tokenType>*> p
               ,CAF<Parser<seperatorType,tokenType>*> sep
               ):p(p),sep(sep){}
    virtual ParsResult<vector<resultType>*,tokenType>*
                                   parse(vector<tokenType*>& xs){
      vector<tokenType*> ys=xs;
      vector<resultType>* results = new vector<resultType>();
      ParsResult<resultType,tokenType>* res = p()->parse(ys);
      if (res->failed()) {
        delete res;
        return new Fail<vector<resultType>*,tokenType>(ys);
      }

      ys = res->getFurtherToken();
      results->insert(results->end(),res->getResult());

      delete res;
      Parser<Pair<seperatorType,resultType>*,tokenType> * sepPars
       = new Seq<seperatorType,resultType,tokenType>(sep,p);

      ParsResult<Pair<seperatorType,resultType>*,tokenType>* resPair
        = sepPars->parse(ys);

      while (!resPair->failed()){
         results->insert(results->end(),resPair->getResult()->snd);
         ys = resPair->getFurtherToken();
         delete resPair->getResult();
         delete resPair;
         resPair= sepPars->parse(ys);
      }
      vector<tokenType*> further=resPair->getFurtherToken();
      delete resPair;
      delete sepPars;

      return
      new ParsResult<vector<resultType>*,tokenType>(results,further);
    }

    virtual std::string toString(){
      return "SeperatedBy("+show(p)+","+show(sep)+")";
    }
};

template <typename a,typename b,typename c>
CAF<Parser<vector<a>*,c>*>
      sepBy(CAF<Parser<a,c>*> p,CAF<Parser<b,c>*> sep){
  return caf((Parser<vector<a>*,c>*) new SeperatedBy<a,b,c>(p,sep));
}
```


Auch für die separierten Listen, wollen wir uns einen Test gönnen. Wir definieren und testen den Parser über Strings, der eine durch das Wort `welt` separierte Wiederholung des Worts `hallo` erkennt.

TestSeperated.cpp

```
#include "ParsLib.h"
#include <string>
#include <iostream>

bool stringEq(string* x,string* y){return *x == *y;}

int main(){
  using namespace name::panitz::parser;
  using namespace std;

  string hallo = "hallo";
  string welt  = "welt";

  vector<string*> xs;
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
```

```
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&welt);
  xs.insert(xs.end(),&hallo);
  xs.insert(xs.end(),&hallo);

  Parser<string*,string>* getHallo=getToken<string>(&hallo,stringEq);
  Parser<string*,string>* getWelt =getToken<string>(&welt,stringEq);

  Parser<vector<string*>*,string>* sepHallo
   = new SeperatedBy<string*,string*,string>(getHallo,getWelt);

  ParsResult<vector<string*>*,string>* res1 = sepHallo->parse(xs);
  cout<<"show(res1):" << show(res1)<<endl;

  vector<string*> further = res1->getFurtherToken();
  ParsResult<string*,string>* res2 = getWelt->parse(further);
  cout<<"show(res2):" << show(res2)<<endl;

  ParsResult<string*,string>* res3 = getHallo->parse(further);
  cout<<"show(res3):" << show(res3)<<endl;
}
```

4.1.9 Leerer Parser mit Ergebnis

In Grammatiken haben wir die Möglichkeit des ϵ, um auszudrücken, daß eine leere Wortfolge zu parsen ist. Als letzte Parserklasse sehen wir auch dafür eine Implementierung vor. Die Klasse `Result` stellt einen Parser vor, der die leere Sequenz von Wörtern erkennt und ein vorgegebenes Ergebnis für die zurückgibt.

ParsLib.h

```
template <typename resultType,typename tokenType>
class Result:public Parser<resultType,tokenType>{
  private:
    resultType result;
  public:
    Result(resultType result):result(result){}

    ParsResult<resultType,tokenType>* parse(vector<tokenType*>& xs){
     return new ParsResult<resultType,tokenType>(result,xs);
    }
    string toString(){return "Result";}
};

template <typename resultType,typename tokenType>
Parser<resultType,tokenType>* result(resultType res){
  return new Result<resultType,tokenType>(res);
}
}}}//namespace end

#endif
```


4.1.10 Beispiel: Parsen und Auswertung arithmetischer Ausdrücke

Bevor wir uns im nächsten Kapitel unserer eigentlichen Programmiersprache *Fab4* widmen, wollen wir in diesem Abschnitt einen kleinen Minitaschenrechner für arithmetische Ausdrücke schreiben.

Tokenklassen

Alle unsere Testparser bisher haben eine Liste von Strings als Eingabe gehabt. Für die arithmetischen Ausdrücke definieren wir uns eigene Objekte für die Token. Es soll dabei zwei verschiedene Arten von Token geben: Zahlen und Operatoren. Wir drücken dieses damit aus, daß wir eine gemeinsame Oberklasse `Token` definieren, die die beiden spezialisierten Klassen `Number` und `Operator` hat.

Zunächst die gemeinsame Oberklasse:

ArithToken.h

```
#ifndef __ARITHTOKEN_H
#define __ARITHTOKEN_H ;

#include "../name/panitz/util/Show.h"
using namespace name::panitz::util;
namespace arith {

class Token{
 public:
  string name;
  Token::Token(string name);
  virtual bool Token::eq(Token* t);
  virtual bool Token::isNumber();
  virtual bool isOperator();
  virtual string Token::toString();
};
```

von der die beiden spezialsierten eigentlichen Tokenklassen ableiten:

ArithToken.h

```
class Number:public Token{
 public:
  int val;
  Number(int val);
  virtual bool isNumber();
  virtual bool eq(Token* t);
  virtual string toString();
```

```
};

class Operator:public Token{
  public:
    int(*op)(int,int);
    Operator(string n,int(*op)(int,int));
    virtual bool eq(Token* t);
    virtual bool isOperator();
};
```

Für die Operatoren definieren wir eigene Funktionen:

ArithToken.h
```
int addOp(int x,int y);
int mulOp(int x,int y);
int subOp(int x,int y);
int divOp(int x,int y);
```

Und für jedes Token legen wir eine Konstante an:

ArithToken.h
```
static Operator addTok = Operator("+",addOp);
static Operator mulTok = Operator("*",mulOp);
static Operator subTok = Operator("-",subOp);
static Operator divTok = Operator("/",divOp);
static Number numTok = Number(42);
}
#endif

```

Die Implementierung dieser Kopfdatei ist sehr geradeheraus:

ArithToken.cpp
```
#include "ArithToken.h"
using namespace name::panitz::util;
namespace arith {

Token::Token(string name):name(name){};
bool Token::eq(Token* t){
  return name==t->name;}
bool Token::isNumber(){return false;}
bool Token::isOperator(){return false;}
string Token::toString(){return name;}

Number::Number(int val):Token("Number"),val(val){};
bool Number::isNumber(){return true;}
bool Number::eq(Token* t){
  return t->isNumber();}
string Number::toString(){return show(val);}
```



```

Operator::Operator(string n,int(*op)(int,int)):Token(n),op(op){};
bool Operator::eq(Token* t){
  return t->isOperator()&&name==name;}
bool Operator::isOperator(){return true;}

int addOp(int x,int y){return x+y;};
int mulOp(int x,int y){return x*y;};
int subOp(int x,int y){return x-y;};
int divOp(int x,int y){return x/y;};

}// end namespace

```

Parser und Interpreter

Nachdem wir uns über den Typ der Token verständigt haben, sind wir in der Lage den Parser zu schreiben. Das Ergebnis eines erfolgreichen Parsvorgangs soll in unseren Fall nicht ein Baum sein, sondern bereits das errechnete Ergebnis für den arithmetischen Ausdruck. Damit bekommen wir einen Parser für den Tokentyp `Token` und den Ergebnistyp `int`. Dieses drückt folgende Kopfdatei aus:

ArithParser.h
```
#ifndef __ARITHPARSER_H
#define __ARITHPARSER_H ;

#include "../name/panitz/parser/ParsLib.h"
#include "../name/panitz/util/Show.h"
#include "ArithToken.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

namespace arith {
 typedef Parser<int,Token>* ArithParser;
 ArithParser getAddExpr();
}
#endif

```

Das Startsymbol unserer Grammatik sei demnach *addExpr*. Die Grammatik für die arithmetischen Ausdrücke dieses Abschnitts ist in Abbildung 4.2 angegeben.

Somit können wir jetzt also den Parser implementieren. Beginnen wir zunächst mit Parsern für die Terminalsymbole. Als erstes mit einen Parser, der das Token `Number` erkennt. Hierzu bedienen wir uns natürlich der Funktion `getToken` aus der Parserbibliothek:

$$\begin{aligned}
addExpr &\rightarrow multExpr\ addOp\ addExpr \mid multExpr && (4.7)\\
multExpr &\rightarrow \texttt{Number}\ multOp\ multExpr \mid \texttt{Number} && (4.8)\\
addOp &\rightarrow \texttt{+} \mid \texttt{-} && (4.9)\\
multOp &\rightarrow \texttt{*} \mid \texttt{/} && (4.10)
\end{aligned}$$

Abbildung 4.2: Einfache Grammatik arithmetischer Ausdrücke.

ArithParser.cpp

```cpp
#include "ArithParser.h"

using namespace name::panitz::parser;
using namespace std;

namespace arith {
int getVal(Number* n){return n->val;}

template <typename t>
bool eq(t* t1,t* t2){return t1->eq(t2);}

CAF<Parser<int,Token>*> numberP
 = caf((Parser<Number*,Token>*)getToken(&numTok,eq)) << getVal;
```

In gleicher Weise definieren wir Parser für die vier Grundrechenarten:

ArithParser.cpp

```cpp
Parser<Operator*,Token>* getOp(Operator* op){
  return (Parser<Operator*,Token>*)getToken(op,eq);}

CAF<Parser<Operator*,Token>*> addP = caf(getOp(&addTok));
CAF<Parser<Operator*,Token>*> mulP = caf(getOp(&mulTok));
CAF<Parser<Operator*,Token>*> subP = caf(getOp(&subTok));
CAF<Parser<Operator*,Token>*> divP = caf(getOp(&divTok));
```

Jetzt lassen sich schon ganz bequem die letzten beiden Regeln der Grammatik umsetzen.

ArithParser.cpp

```cpp
CAF<Parser<Operator*,Token>*>  addOpP = addP || subP;
CAF<Parser<Operator*,Token>*>  mulOpP = mulP || divP;
```

Jetzt bleiben noch die ersten beiden Regeln umzusetzen. Diese haben jeweils zwei Alternativen. Die erste dieser zwei Alternativen ist jeweils eine Sequenz aus drei Bestandteilen: zwei arithmetischen Ausdrücken und einem Operator. Um aus diesen drei Teilen ein Gesamtergebnis zu errechnen, stellen wir eine entsprechende Methode zur Verfügung.

ArithParser.cpp

```cpp
int calculate(Pair<Pair<int,Operator*>*,int>* pp){
  int i2 = pp->snd;
```



```cpp
  Pair<int,Operator*>* p = pp->fst;
  Operator* op= p->snd;
  int i1=p->fst;
  delete p;
  delete pp;
  return op->op(i1,i2);
}
```

Man beachte, daß das temporär durch die Sequenzkombination angelegte Paarobjekt in dieser Methode weiterverarbeitet und schließlich gelöscht wird.

Nun lassen sich die beiden rekursiven Regeln der Grammatik direkt umsetzen:

ArithParser.cpp

```cpp
ArithParser getMultExpr();
CAF<ArithParser> multExpr=caf(getMultExpr);

ArithParser getMultExpr(){
  return
    ( (numberP,mulOpP,multExpr) << calculate
    || numberP
    )();
}

ArithParser getAddExpr();
CAF<ArithParser> addExpr=caf(getAddExpr);
ArithParser getAddExpr(){
  return
    ( (multExpr,addOpP,addExpr) << calculate
    || multExpr
    )();
}

}//end namespace

```

Und schließlich ein kleiner Test für unseren Parser

ArithParserExample.cpp

```cpp
#include "ArithParser.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;
using namespace arith;

int main(){
  vector<Token*> xs;
  xs.insert(xs.end(),(Token*)new Number(17));
```

```
  xs.insert(xs.end(),(Token*)&addTok);
  xs.insert(xs.end(),(Token*)new Number(16));
  xs.insert(xs.end(),(Token*)&addTok);
  xs.insert(xs.end(),(Token*)new Number(4));
  xs.insert(xs.end(),(Token*)&mulTok);
  xs.insert(xs.end(),(Token*)new Number(2));
  xs.insert(xs.end(),(Token*)&addTok);
  xs.insert(xs.end(),(Token*)new Number(1));

  xs.insert(xs.end(),(Token*)new Number(-1));
  cout << show(getAddExpr()->parse(xs))<<endl;
}

```

Und tatsächlich berechnet uns der Parser das Ergebnis eines arithmetischen Ausdrucks.

```
sep@pc216-5:~/fh/compiler/tutor> bin/ArithParserExample
ParsResult(45,[-1])
sep@pc216-5:~/fh/compiler/tutor>
```

Wie man sieht, wurde das letzte Token von dem Parser nicht verbraucht.

4.2 lex zur Generierung von Scannern

Im Gegensatz zu unseren Parser, den wir mit Hilfe der entwickelten Parserbibliothek von Hand codiert haben, soll der Lexer durch ein Standardtool generiert werden. Hierzu bedienen wir uns des Proramms `lex` (bzw. `flex`). `flex` ist ein Programm, das für ein reguläre Ausdrücke einen Automaten generiert, der diese erkennt. Der erzeugte Code ist C-Code. Es gibt `flex`-ähnliche Programme, die Code für andere Programmiersprachen erzeugen.

Wir geben in diesem Abschnitt eine Kurzeinführung in `lex`. Für eine detaillierte Einführung sind die Hilfedateien von `lex` zu konsultieren.

Eine `flex`-Eingabedatei hat drei Blöcke, die mit der Zeichenfolge `%%` getrennt sind:

```
Definitionen
%%
Regeln
%%
Benutzer spezifischer Code
```

Wir werden im folgenden die drei Blöcke am Beispiel eines einfachen Lexers für die Token der arithmetischen Ausdrücke betrachten:

4.2.1 Definitionsblock

Im Definitionsblock finden sich zum einen Definitionen von Token, die dann in den Regeln benutzt werden können zum anderen aber auch C Code, der direkt in das generierte Programm übernommen wird.

Einrückung in diesem Block ist relevant:

- Definitionen beginnen immer in der ersten Spalte und haben die Form: `name definition`
- C-Code der direkt in die Ausgabe übernommen werden soll, beginnt in einer höheren Spalte oder ist mit `%{` und `%}` geklammert.

Zusätzlich können in diesem Teil auch noch Optionen für `lex` gesetzt werden.

Die Definitionen können Namen für reguläre Ausdrücke definieren. Lex hat eine recht komplexe und vieseitige Syntax, um reguläre Ausdrücke zu beschreiben. In unserem Beipiel definieren wir Zahlen und Leerraum:

ArithLexer.lex

```
%option noyywrap
%{
#include <vector>
#include <string>
#include <sstream>
#include "../name/panitz/parser/ParsLib.h"
#include "ArithToken.h"

%}
     using namespace std;
     using namespace arith;
     vector<Token*> result;
     string strLit;

NUMBER     [0-9]+
WHITE      [\n\t\ ]
```

4.2.2 Regeln

Im zweiten Block der Lex-Eingabe werden Regeln f+r verschiedene Token aufgestellt. Diese Regeln beschreiben, was zu tun ist, wenn ein bestimmtes Token erkannt wurde. Token, für die es keine definierte Regel gibt, werden von dem von `lex` erzeugten Programm auf der Standardausgabe ausgegeben. In unseren Beispiel sehen wir sechs Regeln vor: vier für die arithmetischen Operatoren, eine für Leerraum und schließlich eine für Zahlentoken.

ArithLexer.lex

```
%%

"+"  result.insert(result.end(),&addTok);
"*"  result.insert(result.end(),&mulTok);
"-"  result.insert(result.end(),&subTok);
"/"  result.insert(result.end(),&divTok);

{WHITE}    ;
{NUMBER}  {result.insert(result.end(),new Number(atoi(yytext)));}
```

4.2.3 Benutzerspezifischer Code

In einem letzten Abschnitt ermöglicht es lex noch verbatim benutzerspezifischen Code zu schreiben, der ein zu eins in das generierte Programm übernommen wird. Wir schreiben hier eine Funktion, die eine Datei öffnet und diese nach den Regeln scannt:

ArithLexer.lex
```
%%
  namespace arith{
     vector<Token*> tokenize(string fileName){
       char fi [fileName.size()];
       fi [fileName.size()]='\0';
       fileName.copy(fi,fileName.size());
       yyin = fopen(fi, "r" );
       yylex();

       return result;
     }
  }//end namespace

```

Für den generierten Lexer sei eine passende Kopfdatei definiert:

ArithLexer.h
```
#include <string>
#include "ArithToken.h"
namespace arith {
vector<Token*> tokenize(string fileName);
}

```

Nun können wir alles zusammenpacken und erhalten ein kleines Taschenrechnerprogramm.

ArithMain.cpp
```
#include "ArithLexer.h"
#include "ArithParser.h"

#include <string>
#include <iostream>

using namespace std;
using namespace arith;

int main(int argc,char* args[]){
  vector<Token*> inToken=tokenize(args[1]);
  cout<< show(inToken)<<endl;;
  cout<< show(getAddExpr()->parse(inToken))<<endl;
}
```

Der Leser sei darauf hingewiesen, daß unser Programm einen kleine Fehler in Hinblick auf Assoziativität enthält.

4.3 Datentypen für Sprachen

Im letzten Abschnitt haben wir einen Parser entwickelt, der gar keinen Baum aufbaut, sondern gleich die geparsten Bestandteile zum Berechnen einer Zahl weiterverarbeitet. Damit wurde unser Parser automatisch zu einem Interpreter. In der Regel will man als Ergebnis des Parsens einen abstrakten Syntaxbaum erhalten, der dann für weitere Pässe zur Weiterverarbeitung benutzt wird (z.B. Typcheck oder Codegenerierung). Wir haben bereits im Zusammenhang mit den Knoten des Heaps das Standardmuster kennengelernt, wie in einer objektorientierten Modellierung über eine Datenstruktur Algorithmen in gewohnter Weise geschrieben werden: das Besuchsmuster. Das Besuchsmuster findet insbesondere über abstrakte Syntaxbäume in Compilern häufige Anwendung. So werden auch wir es in unserem Parser anwenden. Hierzu definieren wir uns allgemein in generischer abstrakter Form, wie ein Besucher auszusehen hat.

Visitor.h
```
#ifndef __VISITOR_H
#define __VISITOR_H ;

namespace name{namespace panitz{namespace util{

template <typename argType>
class GeneralVisitor{
 public:
    virtual void eval(argType arg)=0;
};

}}}//end namespace
#endif

```

Um die Sache ein wenig zu vereinfachen, verzichten wir auf einen Ergebnistyp für die Methode `eval`. Besucherobjekte werden in der Regel ein Ergebnis erzeugen, daß in einem Feld des Objekts gespeichert ist und abgerufen werden kann.

Kapitel 5

Fab4

when we was fab

Harrison

Wir sind nun endlich gerüstet, um unsere eigene erste Programmiersprache zu implementieren. Wir haben eine funktionierende abstrakte Maschine realisiert, wir haben die wichtigsten Fakten über Grammatiken wiederholt und uns eine kleine Parserbibliothek gebastelt. Zusätzlich mit den Generieren eines Scanners durch *lex* sollte es nun möglich sein, dieses alles zu einem Compiler zusammenzuführen.

5.1 Grammatik

Beginnen wir also mit der Syntax der Sprache *Fab4*. Wir sehen dabei eine Reihe von Terminalsymbolen vor:

- einfache Literale natürlicher Zahlen mit 0.
- Stringliterale
- Charakterliterale
- `CName`: Bezeichner, die mit einen Großbuchstaben beginnen.
- `Name`: Bezeichner, die mit einen Kleinbuchstaben beginnen.
- Schlüsselwörter: `constr`, `case`, `of`
- Symbole: `;`, `+`, `-`, `*`, `/`,`=`, `>`, `>=`, `<`, `<=`, `->`, `(`, `)`

Die komplette Grammatik der Sprache *fab4* ist in Abbildung 5.1 gegeben.

5.2 Fab4 Beispielprogramme

Bevor die Sprache *Fab4* endlich implementieren, ein paar kleine Beispielprogramme.

$$\begin{array}{rcll}
\mathit{fab4} & \to & \mathit{def}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{def}_n & ,n>0 \quad (5.1)\\
\mathit{def} & \to & \mathit{constrDef}\ \mid\ \mathit{funDef} & (5.2)\\
\mathit{constrDef} & \to & \texttt{constr CName Number} & (5.3)\\
\mathit{funDef} & \to & \texttt{Name}\ \mathit{vars}\ \texttt{=}\ \mathit{expr} & (5.4)\\
\mathit{vars} & \to & \mathit{name}_1\ \ldots\ \mathit{name}_n & ,n\geq 0 \quad (5.5)\\
\mathit{expr} & \to & \mathit{conCall}\ \mid\ \mathit{caseExpr}\ \mid\ \mathit{funCall} & (5.6)\\
\mathit{conCall} & \to & \texttt{Cname}\ \mathit{atomExpr}_1\ \ldots\ \mathit{atomExpr}_n & ,n\geq 0 \quad (5.7)\\
\mathit{caseExpr} & \to & \texttt{case}\ \mathit{expr}\ \texttt{of}\ \mathit{caseAlts} & (5.8)\\
\mathit{caseAlts} & \to & \mathit{caseAlt}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \mathit{caseAlt}_n & ,n>0 \quad (5.9)\\
\mathit{caseAlt} & \to & \texttt{CName}\ \mathit{vars}\ \texttt{->}\ \mathit{expr} & (5.10)\\
\mathit{atomExpr} & \to & \texttt{Number}\ \mid\ \texttt{CharLit}\ \mid\ \texttt{StringLit}\ \mid\ \texttt{Name}\ \mid\ \mathit{parExpr} & (5.11)\\
\mathit{parExpr} & \to & \texttt{(}\ \mathit{expr}\ \texttt{)} & (5.12)\\
\mathit{funCall} & \to & \mathit{compExpr}_1\ \ldots\ \mathit{compExpr}_n & ,n\geq 1 \quad (5.13)\\
\mathit{compExpr} & \to & \mathit{addExpr}\ \mathit{compOp}_1\ \mathit{addExpr}_1\ \ldots\ \mathit{compOp}_n\ \mathit{addExpr}_n & ,n\geq 0 \quad (5.14)\\
\mathit{addExpr} & \to & \mathit{multExpr}\ \mathit{addOp}_1\ \mathit{multExpr}_1\ \ldots\ \mathit{addOp}_n\ \mathit{multExpr}_n & ,n\geq 0 \quad (5.15)\\
\mathit{multExpr} & \to & \mathit{atomExpr}\ \mathit{multOp}_1\ \mathit{atomExpr}_1\ \ldots\ \mathit{multOp}_n\ \mathit{atomExpr}_n & ,n\geq 0 \quad (5.16)\\
\mathit{compOp} & \to & \texttt{<}\ \mid\ \texttt{<=}\ \mid\ \texttt{>}\ \mid\ \texttt{>=}\ \mid\ \texttt{=} & (5.17)\\
\mathit{addOp} & \to & \texttt{+}\ \mid\ \texttt{-} & (5.18)\\
\mathit{multOp} & \to & \texttt{*}\ \mid\ \texttt{/} & (5.19)
\end{array}$$

Abbildung 5.1: Fab4 Grammatik

5.2.1 Erste Listenverarbeitung

Zunächst ein kleines Programm, das ein wenig banal mit Listen hantiert. Wir definieren darin die Funktion `last`, die für eine nichtleere Liste das letzte Element zurückgibt. Als zweite Funktion wird `append` zum Aneinanderfügen zweier Listen definiert. Da Zeichenketten in *Fab4* als verkettete Liste modelliert sind, können wir die beiden Listenfunktionen für Strings verwenden.

List1.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
constr Pair 2;

last xs = case xs of
  Cons y ys -> case ys of
                Nil      -> y;
                Cons z zs -> last ys;

append xs ys = case xs of
  Nil       -> ys;
```

```
  Cons z zs -> Cons z (append zs ys);

main=  (Pair (append "hello" (Cons ' ' "world")) (last "jjj!"))
```

5.2.2 Parser in Fab4 schreiben

Erstaunlicher Weise ist unsere Sprache *Fab4* so mächtig, daß die Parserbibliothek `ParsLib` in ihr auf sehr elegante und knappe Weise ausgedrückt werden kann. Alles was in der C++ Implementierung durch Objekte ausgedrückt wurde, wird entweder durch eine Funktion oder Konstruktordaten dargestellt.

Zunächst definieren wir die Standardkonstruktoren:

Arith.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
constr Pair 2;
```

Zwei Konstruktoren werden zur Darstellung des Parserergebnisses benötigt:

Arith.fab4

```
constr ParsResult 2;
constr Fail 1;
```

Der atomare Parser ist eine Funktion, die eine Gleichheitsfunktion für das zu parsende Token bekommt, ein Token, das zu parsen ist und natürlich die Tokenliste:

Arith.fab4

```
parsToken tokenEq tok xs
  = case xs of
      Nil        -> Fail xs;
      Cons y ys  -> case (tokenEq y tok) of
                       True  -> ParsResult y ys;
                       False -> Fail xs;
```

Die Sequenz wird implmentiert als eine Funktion, die zwei Parserfunktionen benutzt um diese nacheinander auf die Tokenliste anzuwenden. Das Ergebnis wird im Erfolgsfall durch ein Paar repräsentiert:

Arith.fab4

```
seq p1 p2 xs
  = case (p1 xs) of
      Fail ys    -> (Fail xs);
      ParsResult res further
                 -> (case (p2 further) of
                      Fail zs -> (Fail xs);
                      ParsResult res2 fs2
                              -> (ParsResult (Pair res res2) fs2));
```


Ähnlich einfach sieht es mit der Alternativkombination aus. Wir hatten in C++ eine Klasse `Either` benutzt, für das Alternativergebnis. Die zwei Ausprägungen von Either drücken wir in *Fab4* durch die zwei Konstruktoren `Left` und `Right` aus. Der Alternativkombinator versucht erst den ersten Parser anzuwenden. Wenn das mißglückt, wird der zweite Parser angewendet. Mit `Left` oder `Right` wird markiert welcher Parser erfolgreich war.

Ebenso wie in C++ stelle wir auch den Operator `ualt` für typuniforme Parseralternativen bereit. Ein `ualt` Parser verpackt das Ergebnis nicht in `Left` bzw. `Right` Konstruktoren.

Arith.fab4

```
constr Left  1;
constr Right 1;

alt p1 p2 xs
  = case (p1 xs) of
      Fail ys
          -> (case  (p2 xs) of
                Fail zs  -> (Fail xs);
                ParsResult l fs1 ->  (ParsResult (Right l) fs1));
      ParsResult res further -> (ParsResult (Left res) further);

ualt p1 p2 xs
  = case (p1 xs) of
      Fail ys  -> (case  (p2 xs) of
                     Fail zs  -> (Fail xs);
                     ParsResult l fs1 ->  (ParsResult l fs1));
      ParsResult res further -> (ParsResult res further);
```

Das Anwenden einer Funktion auf das Parsergebnis mit einem `Map`-Parser ist in *Fab4* eine einfache Funktion höherer Ordnung.

Arith.fab4

```
map f p xs
  =  case (p xs) of
       Fail ys -> (Fail xs);
       ParsResult res further -> (ParsResult (f res) further);
```

Auch die mehrfache Wiederholung wie aus unserer C++-Bibliothek bekannt, läßt sich relativ einfach umsetzen.

Arith.fab4

```
rep0 p1 xs
  = case (p1 xs) of
       Fail fs1 -> (ParsResult (Nil) xs);
       ParsResult res fs2
                -> (case (rep0 p1 fs2) of
                      ParsResult res2 fs3
                        -> (ParsResult (Cons res res2) fs3));

consPair pa  = case pa of Pair x xs ->(Cons x xs);

rep1 p1 xs = map consPair (seq p1 (rep0 p1)) xs;
```

Wir wollen jetzt das altbekannte Beispiel einfacher arithmetischer Ausdrücke in *Fab4* implementieren. Als Tokentyp nehmen wir `char`, parsen also eine Liste von Zeichen. Hierzu folgt eine eine besondere Instanz zum Erkennen von einzelnen Zeichen.

Arith.fab4
```
charEq x y = x=y;
parsCharToken tok xs = parsToken charEq tok xs;
```

Die folgenden Funktionen ermöglichen es natürliche Zahlen zu parsen:

Arith.fab4
```
numberP xs   = map toNum (rep1 digitP) xs;
digitP xs    = oneOfTheseChars ("0123456789") xs;

oneOfTheseChars str xs
 = case str of
    Cons s tr -> (case tr of
        Nil    -> parsCharToken s xs;
        Cons t r -> ualt (parsCharToken s) (oneOfTheseChars tr) xs);

toNum xs = toNumAux 0 xs;
toNumAux n xs
 = case xs of
     Nil -> n;
     Cons c cs -> toNumAux (n*10 +(c+0-48)) cs;
```

Es folgen die Funktionen zum Parsen der Operatoren:

Arith.fab4
```
addF x y = x+y;
subF x y = x-y;
mulF x y = x*y;
divF x y = x/y;

operatorParser ch o xs = map (toOperator o)(parsCharToken ch) xs;

toOperator  op x = op;

addP xs = operatorParser '+' addF xs;
subP xs = operatorParser '-' subF xs;
mulP xs = operatorParser '*' mulF xs;
divP xs = operatorParser '/' divF xs;
```

Und schließlich lassen sich die Regeln der Grammatik direkt mit unseren Kombinatoren niederschreiben:

Arith.fab4
```
addExpr xs
 = ualt (map calc (seq multExpr (seq addOp addExpr)))
        multExpr
```



```
        xs;

multExpr xs
 = ualt (map calc (seq numberP (seq multOp multExpr)))
        numberP
        xs;

addOp  xs = ualt addP subP xs;
multOp xs = ualt mulP divP xs;

calc pp
 = case pp of Pair oe p -> (case p of Pair opF oz -> (opF oe oz));
```

Als Hauptfunktion ein kleiner Test:

Arith.fab4
```
main=addExpr ("8+2*17ende")
```

5.3 Ein Datentyp für Fab4

Wir wollen jetzt den Compiler für Fab4 in C++ implementieren. Hierzu benötigen wir einen Tokenizer, dessen ergebnis als Eingabe für den Parser dient. Der Parser wird als Ergebnis einen abstrakten Syntaxbaum generieren. Auf diesen abstrakten Syntaxbaum können dann verschiedene Compilerpässe angewendet werden. Wir werden in Rahmen der Vorlesung genau einen Pass schreiben, die Codegenerierung. Andere wichtige Pässe eines Compilers werden semantische Überprüfungen sein, eine Typinferenz, Baumtransformationen aber auch Dinge wie ein *Pretty Printer* für die Sprache.

In diesem Abschnitt definieren wir Klassen zur Definition des abstrakten Syntaxbaums. Wir geben dabei die Kopfdatei an. Die sehr naheliegende und einfache Implementierung dür diese Kopfdatei findet sich im AnhangC.2

Fab4.h
```
#ifndef __FAB4_H
#define __FAB4_H

#include <vector>
#include <string>
#include "../name/panitz/util/Visitor.h"
#include "Fab4Token.h"

using namespace std;
using namespace name::panitz::util;

namespace fab4 {
```

Das gängige Muster im objektorientierten Compilerbau für die Implementierung des abstrakten Syntaxbaums und der Pässe über diesen Baum, ist das Besuchsmuster, das wir bereits schon für die Knotenklassen des Heaps in Java kennengelernt haben.

Zentrales Element eines Fab4-Programms sind Ausdrücke. Anders als im imperativen Sprachen gibt es keine Befehle sondern nur Ausdrücke, die einen Wert berechnen. Eine Funktionsdefinition hat genau einen Ausdruck auf der linken Seite. Daher sehen wir eine zentrale Oberklasse `Expr` für alle Arten von Ausdrücken vor. Für die Menge aller Unterklassen von `Expr` sehen wir Besuchsklassen vor:

Fab4.h
```
class Expr;
class ExprVisitor;
```

Audrücke sollen für uns zwei allgemeine Eigenschaften haben:

- eine `toString`-Methode.
- eine Besuchsmethode.

Fab4.h
```
class Expr{
  public:
    virtual std::string toString()=0;
    virtual void visit(ExprVisitor* vis)=0;
};
```

5.3.1 Klassen für Ausdrücke

Aus der Grammatik lassen sich 7 unterschiedliche Arten von Ausdrücken abstrahieren:

- Variablennamen
- Zeichenliterale
- Zahlenliterale
- Operatorausdrücke
- Funktionsanwendungen
- Konstruktoranwendungen
- case-Ausdrücke

Die in der Grammatik auftretenen geklammerten Ausdrücke, lassen sich durch die Hierarchie im Baum ausdrücken. Stringliterale werden in Listen von Zeichen, also durch Folgen von Konstruktoraufrufen dargestellt. Für jede dieser sieben Arten von Ausdrücken sehen wir nun eine spezialisierte Klasse vor:

Variablen

Variablen tragen als einzige Information den als String kodierten Namen der Variabel.

Fab4.h
```
class Var:public Expr{
  public:
    string name;
    Var(string n);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Zeichenkonstanten

Eine Zeichenkonstante enthält als einzige spezifische Information des Zeichenwertes, der durch die konstante dargestellt ist.

Fab4.h
```
class CharExpr:public Expr{
  public:
    char c;
    CharExpr(char c);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Numerische Konstanten

Analog enthält die Klasse für numerische Literale exakt den numerischen Wert, der durch das Literal dargestellt wird.

Fab4.h
```
class NumExpr:public Expr{
  public:
    int n;
    NumExpr(int n);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Operatorusdrücke

Operatorausdrücke enthalten drei Informationen. Die zwei Operanden und den operator:

Fab4.h
```
class OpExpr:public Expr{
  public:
    Expr* e1;
```

```
    Expr* e2;
    Operator* op;
    OpExpr(Expr* e1,Operator* op,Expr* e2);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Funktionsanwendungen

In Fab4 haben wir Funktionen immer als einstellig dargestellt. Die entsprechende Klasse enthält daher genau zwei Informationen: der Ausdruck der angewendet werden soll (also eine Funktion), und der Ausdruck, auf dem diese Funktion angewendet werden soll:

Fab4.h
```
class App:public Expr{
  public:
    Expr* p1;
    Expr* p2;
    App(Expr* p1,Expr* p2);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Konstruktorausdrücke

Konstruktoranwendungen, anders als Funktionsanwendung, sind immer gesättigt, d. h. haben soviel Argumente, wie die Stelligkeit des Konstruktors definiert wurde. Die Argumente eines Konstruktoraufrufs werden daher in einem Vektor gespeichert. Der angewendete Konstruktor selbst wird durch einen als String gespeicherten Namen ausgedrückt:

Fab4.h
```
class Constr:public Expr{
  public:
    string name;
    vector<Expr*>* args;
    Constr(string n,vector<Expr*>* a);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

Caseausdrücke

Die einzig wirklich komplexe Form von Ausdrücken sind die case-Ausdrücke. Diese bestehen aus einer Folge von Alternativen. Daher sehen wir zunächst eine Klasse vor, die in der Lage ist, die Alternativen auszudrücken. Eine case-Alternative enthält drei Informationen:

- den Namen des Konstruktors, für den diese Alternative steht.

- die Namen der Variablen, die für die Argumente des Konstruktors.
- und der Ausdruck, der für diese Alternative auszuwerten ist (in der Fab4-Syntax der Teil nach dem Pfeilzeichen `->`).

Fab4.h
```
class CaseAlt{
  public:
    string constr;
    vector<string> vars;
    Expr* alt;
    CaseAlt(string constr,vector<string> vars,Expr* alt);
    virtual string toString();
};
```

Ein case-Ausdruck beinhaltet einen Ausdruck, für den die Fallunterscheidung getroffen wird und einen Vektor der verschiedenen Alternativen.

Fab4.h
```
class CaseExpr:public Expr{
  public:
    Expr* e1;
    vector<CaseAlt*>* alts;
    CaseExpr(Expr* e1,vector<CaseAlt*>* alts);
    virtual string toString();
    virtual void visit(ExprVisitor* vis);
};
```

5.3.2 Allgemeiner Besucher für Ausdrücke

Gemäß dem Besuchsmuster definieren wir allgemein die Schnittstelle für Besucher über Ausdrücke.

Fab4.h
```
class ExprVisitor:public GeneralVisitor<Expr*>{
 public:
  virtual void eval(Expr* arg)=0;
  virtual void eval(Var* arg)=0;
  virtual void eval(Constr* arg)=0;
  virtual void eval(App* arg)=0;
  virtual void eval(CaseExpr* arg)=0;
  virtual void eval(NumExpr* arg)=0;
  virtual void eval(CharExpr* arg)=0;
  virtual void eval(OpExpr* arg)=0;
};
```

5.3.3 Globale Definitionen

Ein Fab4-Programm besteht nicht allein aus Ausdrücken, sondern den globalen Funktions- und Konstruktordefinitionen. Auch für diese sind Klasse vorzusehen.

Funktionsdefinitionen

Eine Funktion hat einen Namen, eine Parameterliste und auf der rechten Seite einen Ausdruck, zu dem Funktionsanwendungen dieser Funktion zu reduzieren sind. Es läßt sich direkt eine entsprechende Klasse definieren:

Fab4.h

```
class FunDef{
  public:
   string name;
   vector<string> args;
   Expr* expr;
   FunDef(string name,vector<string> args,Expr* expr);

   virtual ~FunDef();
   virtual std::string toString();
};
```

Konstruktordefinitionen

Eine Konstruktordefinition besteht aus einen Konstruktornamen zusammen mit einer Stelligkeit dieses Konstruktors. Auch dieses mündet direkt in die folgende Klasse:

Fab4.h

```
class ConstrDef{
  public:
   string name;
   int arity;
   ConstrDef(string name,int arity);
   virtual std::string toString();
};
```

Komplettes Fab4 Programm

Ein Fab4-Programm ist eine Menge von konstruktor- und Funktionsdefinitionen. Zusätzlich sei noch ein Programmname vorgesehen. Diese Information wird allerdings bisher weder im Programm definiert noch benutzt.

Fab4.h

```
class Fab4 {
  public:
    string name;
    vector<ConstrDef*> constructors;
    vector<FunDef*> functions;

    Fab4(vector<ConstrDef*> cs,vector<FunDef*> fs,string n);
    virtual ~Fab4();
    virtual std::string toString();
};
}
```



```
#endif

```

Damit haben wir den Datentyp des Ergebnisses eines Fab4-Parser komplett spezifiziert.

5.4 Fab4 Syntaxanalyse

5.4.1 Fab4 Token

Um den Parser zu implementieren, müssen wir uns zunächst auf einen Tokentyp für Fab4 einigen. Dieses kann analog zum Beispiel für arithmetische Ausdrücke geschehen. Neben den unterschiedlichen Schlüsselwörtern und Sybolen gibt es folgende besonderen Token:

- Konstruktorbezeichner: dieses sind Bezeichner, die mit Großbuchstaben beginnen.
- Variablen- und Funktionsbezeichner, dieses sind Bezeichner, die mit einem Kleinbuchstaben beginnen.
- Zahlenliterale.
- Zeichenliterale.
- Stringliterale.
- Operatoren.

Zahlenliterale und Operatoren haben wir schon in `ArithToken` kennengelernt.

Zunächst sei eine allgemeine Oberklasse für Token definiert:

Fab4Token.h

```
#ifndef __FAB4_TOKEN_H
#define __FAB4_TOKEN_H ;

#include <string>

using namespace std;

namespace fab4 {

class Token{
 public:
  string name;
  Token(string name);
  virtual bool eq(Token* t);
  virtual bool isCName();
  virtual bool isName();
  virtual bool isNumber();
  virtual bool isCharLiteral();
  virtual bool isStringLiteral();
```

```
  virtual bool isOperator();
  virtual string toString();
};
```

Jetzt lassen sich, wie wir das bereits bei `ArithToken` gesehen haben, Unterklassen für spezielle Token definieren:

Fab4Token.h
```
class CName:public Token{
 public:
  CName(string name);
  virtual bool isCName();
  virtual bool eq(Token* t);
  virtual string toString();
};

class Name:public Token{
 public:
  Name(string name);
  virtual bool isName();
  virtual bool eq(Token* t);
  virtual string toString();
};

class Number:public Token{
 public:
  int val;
  Number(int val);
  virtual bool isNumber();
  virtual bool eq(Token* t);
  virtual string toString();
};

class StringLiteral:public Token{
 public:
  StringLiteral(string s);
  virtual bool eq(Token* t);
  virtual bool isStringLiteral();
};

class CharLiteral:public Token{
 public:
  char c;
  CharLiteral(char c);
  virtual bool eq(Token* t);
  virtual bool isCharLiteral();
};

class Operator:public Token{
  public:
```



```
  string op;
  Operator(string op);
  virtual bool isOperator();
  string toString();
};
```

Für alle Tokenarten, die es in Fab4 gibt, sei eine Prototyp angegeben. Diese werden mit der Gleichheitsfunktion auf Token benutzt, um die entsprechenden Tokenparser im Parser zu definieren:

Fab4Token.h
```
static Token* ARROW_T     = new Token("->");
static Operator* EQ_T     = new Operator("=");
static Operator* ADD_T    = new Operator("+");
static Operator* SUB_T    = new Operator("-");
static Operator* MULT_T   = new Operator("*");
static Operator* DIV_T    = new Operator("/");
static Operator* LT_T     = new Operator("<");
static Operator* GT_T     = new Operator(">");
static Operator* GE_T     = new Operator(">=");
static Operator* LE_T     = new Operator("<=");
static Token* CONSTR_T    = new Token("constr");
static Token* CASE_T      = new Token("case");
static Token* OF_T        = new Token("of");
static Token* LPAR_T      = new Token("(");
static Token* RPAR_T      = new Token(")");
static Token* SEMICOLON_T = new Token(";");
static Number* NUMBER_T   = new Number(0);
static StringLiteral* STRING_T   = new StringLiteral("");
static CharLiteral* CHAR_T   = new CharLiteral(' ');
static Token* NAME_T      = new Name("NAME_T");
static Token* CNAME_T     = new CName("CNAME_T");

bool tokenEq(Token* x,Token* y);

}
#endif
```

Die naheliegende Implementierung dieser Kopfdatei ist im Anhang C.1 angegeben.

5.4.2 Fab4 Lexer

Der Fab4-Lexer sei mittels Lex generiert. Die im Lexer enthaltene Funktion `tokenize` soll eine Datei einlesen und den Vektor der darin gefundenen Token als Rückgabe haben. Wir drücken dieses durch die folgende Kopfdatei aus:

Fab4Lexer.h
```
#include <string>
#include "Fab4Token.h"
```

```
namespace fab4{
vector<Token*> tokenize(string fileName);
}

```

In der Eingabespezifikation für Lex erlauben wir uns den Luxus, zwei Zustände zu haben:

- den Standardzustand, in dem nach Token jeder Art gescannt wird.
- einen besonderen, wenn ein Stringliteral gescannt wird.

Das Erkennen von Stringliteralen verkompliziert den Lexer dabei enorm:

Fab4Lexer.lex

```
%option noyywrap

%{
#include <vector>
#include <string>
#include <sstream>
#include "../name/panitz/parser/ParsLib.h"
#include "../name/panitz/util/Show.h"
#include "Fab4Token.h"
%}

     using namespace std;
     using namespace fab4;
     int num_lines = 0, num_chars = 0;
     vector<Token*> result;
     string strLit;

CHAR        [\'].[\']
DIGIT       [0-9]
NAME        [a-z][a-zA-Z0-9]*
CNAME       [A-Z][a-zA-Z0-9]*
QUOTE       [\"]
WHITE       [\n\t\ ]

%x str
%%
\"         {strLit="";} BEGIN(str);

<str>\"    { /* saw closing quote - all done */
             BEGIN(INITIAL);
             result.insert(result.end(),new StringLiteral(strLit));
           }

<str>\n    {
             /* error - unterminated string constant */
             /* generate error message */
```



```
             throw "unterminated String constant";
           }

<str>\\[0-7]{1,3} {
             /* octal escape sequence */
              int result;
              (void) sscanf( yytext + 1, "%o", &result );
              if ( result > 0xff )
                 throw "error, constant is out-of-bounds";
              strLit=strLit+show(result);
           }

<str>\\[0-9]+ {
              /* generate error - bad escape sequence; something
               * like '\48' or '\0777777'
               */
             throw "bad escape sequence";
           }

<str>\\n  strLit=strLit+ '\n';
<str>\\t  strLit=strLit+ '\t';
<str>\\r  strLit=strLit+ '\r';
<str>\\b  strLit=strLit+ '\b';
<str>\\f  strLit=strLit+ '\f';

<str>\\(.|\n)  strLit=strLit+ yytext[1];

<str>[^\\\n\"]+        {
       char *yptr = yytext;

       while ( *yptr ){strLit=strLit+ (*yptr++);}
     }

"constr"  result.insert(result.end(),CONSTR_T);
"case"    result.insert(result.end(),CASE_T);
"of"      result.insert(result.end(),OF_T);
"->"      result.insert(result.end(),ARROW_T);
";"       result.insert(result.end(),SEMICOLON_T);
"("       result.insert(result.end(),LPAR_T);
")"       result.insert(result.end(),RPAR_T);
"+"       result.insert(result.end(),ADD_T);
"-"       result.insert(result.end(),SUB_T);
"*"       result.insert(result.end(),MULT_T);
"/"       result.insert(result.end(),DIV_T);
"<="      result.insert(result.end(),LE_T);
">="      result.insert(result.end(),GE_T);
"<"       result.insert(result.end(),LT_T);
">"       result.insert(result.end(),GT_T);
"="       result.insert(result.end(),EQ_T);

```

```
{WHITE}    ;
{CHAR}     result.insert(result.end(),new CharLiteral(yytext[1] ));
{DIGIT}+   result.insert(result.end(),new Number(atoi( yytext ) ));
{NAME}     {result.insert(result.end(),new Name(yytext ) );}
{CNAME}    {result.insert(result.end(),new CName(yytext ) );}

%%
  namespace fab4 {

     vector<Token*> tokenize(string fileName){
       char fi [fileName.size()];
       fi [fileName.size()]='\0';
       fileName.copy(fi,fileName.size());
       yyin = fopen(fi, "r" );
       yylex();

       return result;}
     }

```

Mit folgenden Befehlen läßt sich der Lexer schließlich generieren und übersetzen:

```
sep@pc216-5:~/fh/compiler/student/src/fab4> lex -oFab4Lexer.c Fab4Lexer.lex
sep@pc216-5:~/fh/compiler/student/src/fab4> g++ -c Fab4Lexer.c
sep@pc216-5:~/fh/compiler/student/src/fab4>
```

5.4.3 Parser

Nun der Eingabetyp für den Parser ebenso wie der Ergebnistyp vollständig spezifiziert. Mit der Parserbibliothek kann die Grammatik direkt in C++-Code umgesetzt werden. Es ist nur noch zu implementieren, wie mit den Parserzwischenergebnissen der abstrakte Syntaxbaum umgesetzt wird. Dieses soll in einen seperaten Modul geschehen, in dem sich Funktionen befinden, die aus den durch die Grammatikregeln im Parser entstehenden Zwischenergebnissse Teile des abstrakten Syntaxbaums generiert werden. Die Funktionen dieses Moduls sind in der folgenden Kopfdatei aufgelistet:

Fab4MkFunctions.h

```
#ifndef __FAB4MKFUNCTIONS_H
#define __FAB4MKFUNCTIONS_H

#include "../name/panitz/parser/ParsLib.h"
#include "Fab4.h"
#include "Fab4Lexer.h"

using namespace name::panitz::parser;
using namespace std;

namespace fab4 {

```



```
typedef CAF<Parser<Expr*,Token>*>          PExpr;

typedef
  Pair<Pair<Pair<Token*,vector<Token*>*>*,Token*>*,Expr*>*
  caseAltResult;

typedef
  Pair<Pair<Pair<Token*,Expr*>*,Token*>*,vector<CaseAlt*>*>*
  caseParsResult;

typedef  Either<ConstrDef*,FunDef*> ConstrOrFun;

template <typename a, typename b>
a fst(Pair<a,b>* p){a result = p->fst;delete p;return result;}

template <typename a, typename b>
b snd(Pair<a,b>* p){b result = p->snd;delete p;return result;}

Parser<Token*,Token>*    pTok(Token* t);
Parser<Operator*,Token>* pOp(Token* t);
Parser<Number*,Token>*   pNum(Token* t);

Expr* mkNumExpr(Number* num);
Expr* mkCharExpr(CharLiteral* cl);
Expr* mkStringExpr(StringLiteral* sl);
Expr* mkVar(Token* t);
Expr* mkOpExpr(Pair<Expr*,vector<Pair<Operator*,Expr*>*>*>* p);
Expr* mkFunApps(vector<Expr*>* exprs);
Expr* mkConstr(Pair<Token*,vector<Expr*>*>* p);
CaseAlt* mkCaseAlt(caseAltResult caseAlt);
Expr* mkCaseExpr(caseParsResult casePars);
FunDef* mkFunDef
 (Pair<Pair<Pair<Token*,vector<Token*>*>*,Operator*>*,Expr*>* p);
ConstrDef* mkConstr(Pair<Token*,Number*>* p);
Fab4* mkFab4(vector<ConstrOrFun*>* constrOrFuns);

}
#endif

```

Die Funktionen dieses Moduls können benutzt werden, um an die Regeln der Grammatik in der C++-Implementierung die Semantik mit den Operator `<<` anzuheften. So läßt sich jetzt die Regel der Grammatik für Vergleichsoperatoren wie folgt in C++ umsetzen:

```
PExpr compExprP = (addExprP,rep0((compOpP,addExprP)))<< mkOpExpr;
```

Die Implementierung von `Fab4MkFunctions` befindet sich im Anhang. Damit ist alles vorhanden, um den Parser umzusetzen.

Aufgabe 26 Implementieren Sie einen Parser für die Sprache Fab4. Hierzu ist für die folgende Header-Datei eine Implementierung zu erstellen:

Fab4Parser.h

```
#ifndef __FAB4PARSER_H
#define __FAB4PARSER_H

#include "../name/panitz/parser/ParsLib.h"
#include "Fab4.h"
#include "Fab4Lexer.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

namespace fab4 {
ParsResult<Fab4*,Token>* parsFab4(string fileName);
}
#endif
```

Hinweis: Ihr Programm wird etwas übersichtlicher und leichter zu lesen, wenn sie sich der Möglichkeit von Typsynonymen bedienen. Folgende Typsynonyme sind dabei sinnvoll:

```
typedef CAF<Parser<Token*,Token>*>         PToken;
typedef CAF<Parser<Operator*,Token>*>      POperator;
typedef CAF<Parser<Number*,Token>*>        PNumber;
typedef CAF<Parser<StringLiteral*,Token>*> PStringLit;
typedef CAF<Parser<CharLiteral*,Token>*>   PCharLit;
```

Aufgabe 27 Testen Sie Ihren Parser mit folgendem kleinen Hauptprogramm anhand einer Reihe unterschiedlicher Fab4-Programme.

TestFab4Parser.cpp

```
#include "Fab4Parser.h"
using namespace fab4;
using namespace name::panitz::parser;

int main(int argc,char* args[]){
  vector<Token*> xs;

  cout<<"PairCount: "<<PairCount<<endl;
  cout<<"ParsResultCount: "<<ParsResultCount<<endl;

  ParsResult<Fab4*,Token>* res = parsFab4(args[1]);
  cout<<show(res)<<endl;

  cout<<"PairCount: "<<PairCount<<endl;
```



```
  cout<<"ParsResultCount: "<<ParsResultCount<<endl;
}
```

Kapitel 6

Codegenerierung

In einem letzten Implementierungsschritt gilt es nun aus dem abstrakten Syntaxbaum Code für die G-Maschine zu erzeugen.

6.1 Alte Bekannte: G-Maschinen Befehlsklassen

Wir definieren für die G-Maschinenbefehle Konstanten, so wie wir es in Java bereits getan haben:

Instruction.h

```
#ifndef __INSTRUCTION_H
#define __INSTRUCTION_H

#include <map>
#include <vector>
#include <string>
#include <fstream>

#include "../name/panitz/util/Show.h"
#include "Fab4Parser.h"
#include "Instruction.h"

using namespace std;
using namespace name::panitz::util;

namespace fab4 {
typedef char byte;

#define NOP           (byte)0
#define POP           (byte)1
#define END           (byte)2
#define JUMP          (byte)3
#define PUSH          (byte)4
#define UNWIND        (byte)5
```



```
#define EVAL          (byte)6
#define PRINT         (byte)7
#define PRINTSTRING   (byte)9
#define OUTPUT        (byte)10
#define MKAP          (byte)11
#define JAVACALL      (byte)12
#define PUSHGLOBAL    (byte)13
#define PUSHINT       (byte)14
#define PUSHCHAR      (byte)15
#define SLIDE         (byte)16
#define UPDATE        (byte)17
#define PACK          (byte)18
#define CASEJUMP      (byte)19
#define SPLIT         (byte)20
#define ADD           (byte)21
#define SUB           (byte)22
#define MULT          (byte)23
#define DIV           (byte)24
#define LE            (byte)25
#define GE            (byte)26
#define LT            (byte)27
#define GT            (byte)28
#define EQ            (byte)29
#define STATICJAVACALL (byte)30
#define GETRUNTIME    (byte)31

#define FA            (byte)250
#define B4            (byte)180
```

Für jeden Befehl können wir wieder eine eigene Klasse vorsehen. Dazu gibt es eine gemeinsame Oberklasse für Befehle. Für die Befehlsklassen sind zwei Mehtoden vorgesehen: ein `toString` und eine Methode, die den Befehl in einen Ausgabestrom für den Bytecode schreibt.

Instruction.h

```
class  Instruction{
 public:
  virtual string toString()=0;
  virtual void write(ostream& out)=0;
};
```

Es folgen die Klassen für die einzelnen Befehle. Zunächst Befehle ohne Parameter:

Instruction.h

```
class End:public Instruction{
 public: string toString();  virtual void write(ostream& out);};

class Unwind:public Instruction{
 public: string toString();  virtual void write(ostream& out);};

class Eval:public Instruction{
```

```
 public:  string toString();  virtual void write(ostream& out);};

class Print:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class PrintString:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class MkAp:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class JavaCall:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class GetRuntime:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class StaticJavaCall:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Add:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Sub:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Mult:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Div:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Le:public Instruction{
 public:  string toString();  virtual void write(ostream& out);};

class Ge:public Instruction{
 public: string toString();  virtual void write(ostream& out);};

class Lt:public Instruction{
 public: string toString();  virtual void write(ostream& out);};

class Gt:public Instruction{
 public: string toString();  virtual void write(ostream& out);};

class Eq:public Instruction{
 public: string toString();  virtual void write(ostream& out);};
```

Nun die Befehle mit einem `int` oder `char` Parameter:

Instruction.h

```
class Jump:public Instruction{
 public:
  int n;
  Jump(int n);
  string toString();  virtual void write(ostream& out);};

class Push:public Instruction{
 public:
  int n;
  Push(int n);
  string toString();  virtual void write(ostream& out);};

class Pop:public Instruction{
 public:
  int n;
  Pop(int n);
  string toString();  virtual void write(ostream& out);};

class PushInt:public Instruction{
 public:
  int n;
  PushInt(int n);
  string toString();  virtual void write(ostream& out);};

class Slide:public Instruction{
 public:
  int n;
  Slide(int n);
  string toString();  virtual void write(ostream& out);};

class Update:public Instruction{
 public:
  int n;
  Update(int n);
  string toString();  virtual void write(ostream& out);};

class Split:public Instruction{
 public:
  int argc;
  Split(int c);
  string toString();  virtual void write(ostream& out);};

class PushChar:public Instruction{
 public:
  char n;
  PushChar(char n);
  string toString();  virtual void write(ostream& out);};
```

Der Befehl `Pack` mit seinen zwei Parametern stellt eine kleine Besonderheit dar:

Instruction.h
```
class Pack:public Instruction{
 public:
  int name;
  int argc;
  Pack(int n,int c);
  string toString();  virtual void write(ostream& out);};
```

Zwei Befehle haben einen `string` Parameter:

Instruction.h
```
class Output:public Instruction{
 public:
  string s;
  Output(string s);
  string toString();  virtual void write(ostream& out);};

class PushGlobal:public Instruction{
 public:
  string name;
  PushGlobal(string n);
  string toString();  virtual void write(ostream& out);};
```

Wie bereits in der Javaimplementierung erweitert der Befehl `CaseJump` eine Klasse `map` für Abbildungen:

Instruction.h
```
class CaseJump:  public std::map<int,int>
                ,public Instruction {
 public:
  string toString();  virtual void write(ostream& out);};
```

Die folgenden Methoden sind Hilfreich bei der Codegenerierung und seien hier bereits definiert.

Instruction.h
```
Instruction* getOp(Operator* opT);

template <typename InputIterator,typename Element>
bool contains(Element el,InputIterator begin,InputIterator end){
  for (InputIterator it = begin+1;it<end;it++){
    if (el == (*it)){ return true;}
  }
  return false;
}
template <typename InputIterator,typename Element>
bool contains(Element el,InputIterator begin,InputIterator end
             ,bool(*eq)(Element,Element)){
  for (InputIterator it = begin+1;it<end;it++){
```



```
    if (eq(el,(*it))){return true;}
  }
  return false;
}

void writeInt(int i,ostream& s);
void writeString(string str,ostream& s);
}
#endif

```

Die vollständige Implementierung dieser Kopfdatei ist wieder im Anhang angegeben. Sie enthält nur naheliegenden Code.

6.2 Erzeugung von Fab4 Code

Als ein letztes Modul schreiben wir nun endlich die Codegenerierung. Das Modul `GenCode` soll in unserem Fall der Einfachheit halber die Hauptmethode des Compilers enthalten. Daher definieren wir für dieses Modul keine Kopfdatei, sondern implementieren es gleich vollständig:

GenCode.cpp
```
#include <map>
#include <vector>
#include <string>
#include <fstream>

#include "../name/panitz/util/Show.h"
#include "Fab4Parser.h"
#include "Instruction.h"

using namespace std;
using namespace name::panitz::util;

namespace fab4 {
```

6.2.1 Eine Besucherklasse zur Codegenerierung

Die wichtigste Aufgabe der Codegenerierung besteht im Erzeugen von Code für Ausdrücke, den rechten Seiten der Funktionsdefinitionen. Die Baumklassen für Ausdrücke sind Klassen, für die Besucher geschrieben werden können. Wir schreiben einen Besucher, der Code generiert:

GenCode.cpp
```
class GenExprCode:public ExprVisitor{
  public:
```

Das Ergebnis dieses Besuchers soll ein Vektor mit dem erzeugten Code sein:

GenCode.cpp
```
    vector<Instruction*>* result;
```

Um Code korrekt zu generieren, brauchen wir in dem Besucher die Namen der Funktionen des Fab4 Programms, eine Abbildung von Konstruktornamen auf Ihre Nummern, und eine Abbildung von Konstruktornamen auf Ihre Wertigkeit:

GenCode.cpp
```
    vector<string> functions;
    std::map<string,int> constructors;
    std::map<string,int> constructorArity;
```

In fab4 gibt es Variablen. Zum einen die Parameternamen der Funktionsparameter, zum anderen die Variablen, die in Case-Alternativen eingeführt werden. Betrachten wir hierzu noch einmal die Fab4 Funktion zum konkatenieren von Listen:

```
append xs ys = case xs of
  Nil      -> ys;
  Cons z zs -> Cons z (append zs ys);
```

Hier geivt es die lokalen Variablen `xs` und `ys`, nämlich die Parameter der Funktione; desweiteren führt die zweite Case-Alternative noch die Variablen `z` und `zs` ein. Innerhalb der G-Maschine liegen diese Variablen irgendwo auf dem Stack. Während der Codegenerierung müssen wir darüber Buch führen, auf welcher Position im Stack diese Variablen liegen. Daher braucht der Besucher zur Codegenerierung eine Umgebung, die Variablennamen auf Stackpositionen Abbildung. Wir brauchen also einen `Map`, zur Abbildung von `string`-Werten (den Variablennamen) auf `int`-Werte (den Positionen dieser Variablen auf dem aktuellen Stack).

Wir sehen eine solche Abbildung im Besucher zur Codegenerierung vor. Statt auf eine `map`-Klasse aus der STL zurückzugreifen, modellieren wir diese zur Übung noch einmal selbst, durch eine Liste von Paaren, so wie es in einer der ersten Javaübungen verlangt war:

GenCode.cpp
```
    vector<Pair<string,int> > env;

    int lookUp(string n){
      for (int i=0;i<env.size();i++)
        if (env[i].fst==n) return env[i].snd;
      throw ("unknown variable: "+n);
    }
```

Ein Konstruktor wird angeboten, die obigen Felder zu initialisieren:

GenCode.cpp
```
    GenExprCode
      (vector<Instruction*>* r
      ,vector<Pair<string,int> >& e
      ,vector<string>& fs
      ,std::map<string,int>& cs,std::map<string,int>&  arity){
        result=r;
```



```
        env=e;
        functions=fs;
        constructors=cs;
        constructorArity=arity;
    }
```

In gewohnter Weise ist nun der Besucher zu implementieren. Im Fall, daß `eval` für die allgemeine Klasse `Expr` aufgerufen wird, ist ein Fehler zu werfen:

GenCode.cpp
```
    virtual void eval(Expr* arg){
      throw ("unmatched visitor pattern for: "+show(arg));
    }
```

Code für Zahlenkonstanten

Der Code für eine Zahlenkonstante ist, das mag nicht überaschend kommen, trivial. Es gibt den G-Maschinenbefehl `PushInt`. Dieser ist genau einmal in den Befehlsvektor hinten anzufügen (mit `insert(result->end(),...)`).

Aufgabe 28 Ergänzen Sie den Rumpf der Methode `eval` für `NumExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(NumExpr* arg){
```

GenCode.cpp
```
    }
```

Ihre Implementierung sollte jetzt in der Lage sein, für das erste kleine einfache Programm eine ausführbare Byte-Code-Datei zu erzeugen.

```
main = 42
```

Hierzu ist natürlich alles zu übersetzen und linken:

```
sep@pc216-5:~/fh/compiler/student/src/fab4> lex -oFab4Lexer.cpp Fab4Lexer.lex
sep@pc216-5:~/fh/compiler/student/src/fab4> g++ -c *.cpp
sep@pc216-5:~/fh/compiler/student/src/fab4> rm TestFab4Parser.o
sep@pc216-5:~/fh/compiler/student/src/fab4> g++ -o fab4c *.o
sep@pc216-5:~/fh/compiler/student/src/fab4> ./fab4c ../Prog0
[PUSH 1,EVAL,PUSH 1,EVAL,STATICJAVACALL,UNWIND,PUSH 2,EVAL,PUSH 2,EVAL,PUSH 2,EVAL,JAVACALL
,UNWIND,GETRUNTIME,UNWIND,PUSHINT 42,UNWIND]
sep@pc216-5:~/fh/compiler/student/src/fab4> java -cp ~/fh/compiler/tutor/classes/ name.panitz.gm.Gm ../Prog0
42
sep@pc216-5:~/fh/compiler/student/src/fab4>
```

Code für Variablen

es gibt die Klasse `Var` in unserem abstrakten Syntaxbaum. Sie drückt aus, daß im Quelltext ein Bezeicher stehen kann. Dieser Bezeichner kann zweierlei bedeuten:

- es kann ein Funktionsname einer global definierten Funktion sein.
- es kann der Name einer lokalen Variablen, für die in der Umgebung `env` eine Stackposition vermerkt ist sein.

Für diese beiden Fälle ist unterschiedlicher Code zu generieren:

- für Funktionsnamen `name` ein `PushGlobal(name)`.
- für sonstige Variablennamen `name` ein `Push(lookuo(name))`.

Aufgabe 29 Ergänzen Sie den Rumpf der Methode `eval` für Variablennamen, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

Um zu testen, ob der Variablenname ein Funktionsname ist benutzen Sie die Funktion `contains` aus Modul `Instruction` mit folgender Gleichheitsfunktion:

GenCode.cpp
```
    static bool stringEq(string s1,string s2){return s1==s2;}
```

GenCode.cpp
```
    virtual void eval(Var* arg){
```

GenCode.cpp
```
    }
```

Jetzt sollte Ihre Implementierung in der Lage sein, das folgende erste Fab4-Programm zu übersetzen und korrekten ausführbaren Code dafür erzeugen:

```
zweiundvierzig = 42;
main = zweiundvierzig
```

Code für Funktionsapplikationen

Als nächstes wollen wir Funktionsanwendungen übersetzen können. Hierzu brauchen wir mit Sicherheit den G-Maschinenbefehl `MkAp`. Dieser Befehl allein reicht allerdings nicht aus. Zuvor müssen wir dafür sorgen, daß Code für die beiden an der Funktionsanwendung beteiligten Ausdrücken erzeugt wird. Dieser Code muß dafür Sorgen, daß auf dem Stack die Referenzen dieser beiden Ausdrücke liegen.

Für einen Ausdruck der Form `App`, der die Unterausdrücke `p2` und `p1` enthält, ist auf folgender Art Code zu erzeugen:

- erzeuge Code für `p2`, indem der Besucher zur Codegenerierung auf `p2` angewendet wird.
- wird der für `p2` generierte Code ausgeführt, dann verschieben sich die Referenzen auf dem

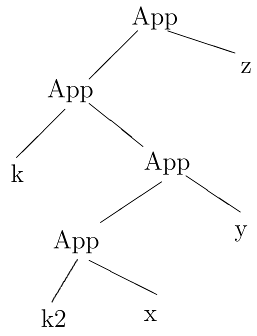


Abbildung 6.1: AST für kk.

Stack. Deshalb ist in einem zweiten Schritt die Umgebung zu verändern. Jede Zahl, die in der Umgebung steht, ist um eins zu erhöhen.

- dann erzeuge Code für `p1`, indem der Besucher zur Codegenerierung auf `p2` angewendet wird.
- schließlich füge den Befehl `MkAp` an die Befehlsliste an.
- setze die Umgebung wieder auf die ursprüngliche Umgebung zurück.

Beispiel:
Ein kleines Beispiel soll dieses vergegenwärtigen. Betrachten wir dazu folgendes Programm:

```
k x y = x;
k2 x y = y;
kk x y z = k (k2 x y) z;
main = kk 17 42 4
```

Uns interessiert die Coegenerierung für die Funktion `kk`. Der Parser sorgt dafür das über *currying* die Funktionsanwendungen der zweistelligen Funktionen, durch einstellige Funktionen dargestellt ist. Für den Audruck `k (k2 x y) z` ist folgender abstrakte Syntaxbaum erzeugt worden:

Beim Einstieg in die Funktion `kk` liegen die Argumente auf dem Stack. Die Umgebung ist also folgende Abbildung:
$\{x \to 0, y \to 1, z \to 2\}$

Zunächst ist Code für die Variable z zu erzeugen. Es wird der Befehl `Push 2`, erzeugt. Die Umgebung wird verändert zu:
$\{x \to 1, y \to 2, z \to 3\}$

Jetzt ist mit dieser neuen Umgebung Code zu erzeugen für das erste Argument der Anwendung von `k`. Dieses ist die Anwendung von `k2`. Zunächst das zweite

Argument. Dieses ist die Variable y. In der aktuellen Umgebung ist die auf Stackposition zwei zu finden, also wird `Push 2` generiert. Wieder ist die Umgebung um eins zu erhöhen. Wir erhalten jetzt die Umgebung:
$\{x \rightarrow 2, y \rightarrow 3, z \rightarrow 4\}$
Mit dieser Umgebung ist Code für das erste Argument der Anwenung von `k2` zu generieren. Dieses ist die Variable x, die auf Stackposition zwei zu finden ist. Es wird `Push 2` generiert. Nochmals ist die Umgebung um eins zu erhöhen. Wir erhalten jetzt die Umgebung:
$\{x \rightarrow 3, y \rightarrow 4, z \rightarrow 5\}$
Nun ist Code für den Funktionsaufruf von `k2` zu generieren, also der Befehl `PushGlobal k2`.
Anschließend wird die Umgebung wieder um eins erniedrigt (alte zwischengespeicherte Umgebung reaktiviert). Jetzt haben wir die Umgebung:
$\{x \rightarrow 2, y \rightarrow 3, z \rightarrow 4\}$

Nun kann der erste Applikationsknoten erzeugt werden mit `MkApp`.
Anschließend wird die Umgebung wieder um eins erniedrigt (alte zwischengespeicherte Umgebung reaktiviert). Jetzt haben wir die Umgebung:
$\{x \rightarrow 1, y \rightarrow 2, z \rightarrow 3\}$

Nun kann der zweite Applikationsknoten erzeugt werden mit `MkApp`.
Jetzt haben wir Code für `k2 x y` auf dem Stack, und darunter Code für z. Der erste Applikationsknoten der Anwendung von `k` kann generiert werden. Also nocheinmal `MkAp`.

Anschließend wird die Umgebung wieder um eins erhöht (alte zwischengespeicherte Umgebung reaktiviert). Jetzt haben wir die Umgebung:
$\{x \rightarrow 2, y \rightarrow 3, z \rightarrow 4\}$
Nun ist Code für den Funktionsaufruf von `k` zu generieren, also der Befehl `PushGlobal k`.
Anschließend wird die Umgebung wieder um eins erniedrigt (alte zwischengespeicherte Umgebung reaktiviert). Jetzt haben wir die Umgebung:
$\{x \rightarrow 1, y \rightarrow 2, z \rightarrow 3\}$
Ein letztes Mal wird die Umgebung wieder um eins erniedrigt; nämlich nach Erzeugung des Codes für das erste Kind des obersten Applikationsknotens. Jetzt haben wir die Umgebung:
$\{x \rightarrow 0, y \rightarrow 1, z \rightarrow 2\}$

Insgesamt haben wir folgenden Code generiert:
`PUSH 2,PUSH 2,PUSH 2,PUSHGLOBAL k2,MKAP,MKAP,PUSHGLOBAL k,MKAP,MKAP`

Aufgabe 30 Ergänzen Sie den Rumpf der Methode `eval` für Funktionsapplikationen `App`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
  virtual void eval(App* arg){
```

GenCode.cpp
```
  }
```


Jetzt sollte Ihre Implementierung in der Lage sein, das folgende Fab4-Programm zu übersetzen und korrekten ausführbaren Code dafür zu erzeugen:

```
k x y = x;
main = k 42 15
```

Natürlich sollte jetzt auch für das Programm `kk` korrekter Code erzeugt werden.

Code für Operatorausdrücke

Die Codegenerierung für Operatorausdrücke ist sehr ähnlich der Codegenerierung für Applikationsknoten. Auch hier ist wieder zu bedenken, daß nach Generierung des Codes für den zweiten Operanden, die Umgebung um eins zu erhöhen ist. Mit dieser neuen Umgebung kann dann Code für den ersten Operanden erzeugt werden.

Im Gegensatz zu den Applikationsknoten sind die eingebauten Operatoren allerdings strikt. Sie erwarten, daß ihre Argumente bereits ausgewertet vorliegen. Deshalb ist nach Generierung des Codes für einen Operator stets noch der efehl `Eval` zu generieren.

Insgesamt ist der Code für Operatorausdrücken in folgenden Schritten

- erzeuge Code für zweiten Operanden.
- generiere Befehl `Eval`.
- erhöhe die Umgebung um eins.
- erzeuge Code für zweiten Operanden.
- generiere Befehl `Eval`.
- erniedrige die Umgebung um eins.
- geneiere passenden Befehl für den Operator. `Add`,`Sub`...

Aufgabe 31 Ergänzen Sie den Rumpf der Methode `eval` für Operatorausdrücke `OpExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
   virtual void eval(OpExpr* arg){
```

GenCode.cpp
```
   }
```

Jetzt sollte Ihre Implementierung in der Lage sein, das folgende Fab4-Programm zu übersetzen und korrekten ausführbaren Code dafür zu erzeugen:

```
double x  = x+x;
square x = x*x;

main = square (double 2)
```


Code für Zeichenkonstanten

Der Code für Zeichenkonstanten ist vollkomen analog für den Code für numerische Konstanten zu generieren.

Aufgabe 32 Ergänzen Sie den Rumpf der Methode `eval` für Zeichenkonstanten `CharExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
   virtual void eval(CharExpr* arg){
```

GenCode.cpp
```
   }
```

Jetzt sollte Ihre Implementierung in der Lage sein, das folgendes Fab4-Programm zu übersetzen und korrekten ausführbaren Code dafür zu erzeugen:

```
main = 'a'
```

Code für Konstruktoraufrufe

Für Konstruktoren ist mit Sicherheit ein Befehl `Pack` zu generieren. Zuvor ist aber auch noch der Code zu generieren, der dafür sorgt, daß die Argumente des Konstruktoraufrufs auf den Stack gelegt wurden. Auch hier gilt wieder, daß nach Generierung des Codes für ein Argument die Umgebung jeweils um eins zu erhöhen ist. Die Codegenerierung für Konstruktoren hat also folgende Struktur:

- iteriere vom letzten bis zum ersten Argument des konstruktoraufrufs.
- in jedem Iterationsschritt, generiere für das aktuelle Argument Code und erhöhe anschließend die Umgebung.
- reaktiviere die ursprüngliche Umgebung.
- generiere den Befehl `Pack`.

Aufgabe 33 Ergänzen Sie den Rumpf der Methode `eval` für Konstruktoraufrufe `Constr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
   virtual void eval(Constr* arg){
```

GenCode.cpp
```
   }
```

Jetzt sollte Ihre Implementierung in der Lage sein, das folgendes Fab4-Programm zu übersetzen und korrekten ausführbaren Code dafür zu erzeugen:

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;

repeat x = (Cons x (repeat x));
main = repeat 42
```

Auch für Stringliterale sollte Ihr Compiler jetzt korrekten Code erzeugen. Es wird dafür eine Liste von Zeichen erzeugt. Dieses geschieht bereits in der Funktion `mkStringExpr`, die im Parser aufgerufen wird:

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;

main = "hallo welt!"
```

Code für Case-Ausdrücke

Es fehlt nur noch Code für die letzte Form von Ausdrücken zu erzeugen, den Case-Ausdrücken. Und wie auch schon bei anderen Operationen, ist dieses wieder der schwerste Fall.

Zwei Dinge sind hier zu berücksichtigen, die bei den anderen Ausdrücken noch nicht vorkamen:

- Die Alternativen eines Case-Ausdrucks können neue Variablen einführen, die auf dem Stack abgelegt werden. Die Umgebung ist für jede Alternative um diese variablen zu erweitern.
- Eine Sprungtabelle ist aufzubauen. Hierzu ist zu zählen, wieviele Befehle jede der Alternativen beinhaltet.

Für einen Ausdruck der Form:
`case` e `of`
alt_1
...
alt_n

Erzeuge Code in folgender Form:

- erzeuge Code für e.
- erzeuge den Befehl `Eval`.
- erzeuge Code für jede case Alternative. Dieser beginnt mit einem Split Befehl (lege die Argumente des gefundenen Konstruktors auf den Stack) und endet mit einem `Slide` (entferne die Argumente des Konstruktors aus dem Stack) und schließlich einen `Jump`-Befehl (überspringe die nächsten Case-Alternativen).
- baue dabei den korrekten `CaseJump` auf.

Aufgabe 34 Ergänzen Sie den Rumpf der Methode `eval` für case-Ausdrücke `caseExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp

```
  virtual void eval(CaseExpr* arg){
```

Lösung

Diese Lösung ist weit von einer Musterlösung entfernt. Insbesondere der neue Besucher, der mit `new` erzeugt wird, sollte auch wieder aus dem Speicher gelöscht werden. Auch sonst idt die Lösung unübersichtlich und kaum mehr zu verstehen.

GenCode.cpp

```

    arg->e1->visit(this);
    result->insert(result->end(),new Eval());
    vector<CaseAlt*>* alts = arg->alts;
    vector<int>* jumpPointToEnd= new vector<int>();
    vector<Instruction*>* code=new vector<Instruction*>();

    CaseJump* jump=new CaseJump();
    result->insert(result->end(),jump);
    int startAddress = 0;
    for (int i=0;i<alts->size();i++){
      CaseAlt* alt=(*alts)[i];
      (*jump)[constructors[alt->constr]]=startAddress;

      vector<Instruction*>* altCode=new vector<Instruction*>();
      int n= alt->vars.size();
      altCode->insert(altCode->end(),new Split(n));
      vector<Pair<string,int> > newEnv=env;
      for (int j=0;j<env.size();j++)
        newEnv[j].snd=newEnv[j].snd+n;
      for (int j=0;j<n;j++){
        newEnv.insert(newEnv.begin(),Pair<string,int>(alt->vars[j],j));
      }
      alt->alt->visit(new GenExprCode
       (altCode,newEnv,functions,constructors,constructorArity));
      altCode->insert(altCode->end(),new Slide(n));
      altCode->insert(altCode->end(),new Jump(0));
      code->insert(code->end(),altCode->begin(),altCode->end());;
      startAddress=code->size();
      jumpPointToEnd->insert(jumpPointToEnd->end(),startAddress-1);
   }
   for (int i=0;i<jumpPointToEnd->size();i++){
    int jumpPoint=(*jumpPointToEnd)[i];
    (*code)[jumpPoint]=new Jump(startAddress-jumpPoint-1);
   }
   result->insert(result->end(),code->begin(),code->end());;
```

GenCode.cpp
```
    }
```

Damit sollten jetzt alle Fab4-Programme mit Ihren Compiler korrekt übersetzt werden. Der Besucher zur Codegenerierung ist komplett implementiert.

GenCode.cpp
```
};
```

6.2.2 Code für Funktionsdefinitionen

Bei der Generierung des Codes für Funktionsdefinitionen ist zu beachten: beim Einstieg in den Code, liegen die Parameter der Funktion auf dem Stack. Unter den Parametern liegt der Applikationsknoten für die Funktionsanwendung. Es ist Code für den Ausdruck auf der rechten Seite der Funktionsdefinition zu generieren. Dieser Code sorgt nach Abarbeitung dafür, daß das Ergebnis der Funktionsanwendung auf dem Stack liegt. Es ist schließlich bei einer Funktion mit n Parametern noch folgende Codesequenz zu generieren:

```
[Update n,Pop n,Unwind]
```

Die Funktionsapplikation ist durch das Ergebnis zu überschreiben. Dann sind die Argumente vom Stack zu poppen. Schließlich wird die G-Maschine mit dem `Unwind` zu prüfen, was weiter mit dem Ergebnis zu geschehen hat.

Folgender Code realisiert die Generierung des G-Maschinencodes für Funktionen:

GenCode.cpp
```
vector<Instruction*>* genCode
  (FunDef* fun
  ,vector<Instruction*>* result
  ,vector<string> fs
  ,std::map<string,int>& constr
  ,std::map<string,int>& ar){

    std::vector<Pair<string,int> > env;
    for (int i=0;i<fun->args.size();i++)
      env.insert(env.end(),Pair<string,int>(fun->args[i],i));
    GenExprCode* genCoder = new GenExprCode(result,env,fs,constr,ar);
    fun->expr->visit(genCoder);
    if(fun->args.size()>0){
      result->insert(result->end(),new Update(fun->args.size()));
      result->insert(result->end(),new Pop(fun->args.size()));
    }
    result->insert(result->end(),new Unwind());

    return result;
};
```


6.2.3 Code für komplettes Modul

Ein Fab4 Programm besteht aus einer Menge von Funktions- und Konstruktordefinitionen. Für jede Funktion ist Code zu generieren. Die Konstruktoren sind aufzusammeln mit ihren Namen und ihrer Stelligkeit.

GenCode.cpp
```
vector<Instruction*>* genCode (Fab4* fab4){
  vector<string> fs;
  vector<Instruction*>* result=new vector<Instruction*>();
  std::map<string,int> constr;
  std::map<string,int> arity;

  for (int i=0;i<fab4->constructors.size();i++){
    constr[fab4->constructors[i]->name]=i;
    arity[fab4->constructors[i]->name]=fab4->constructors[i]->arity;
  }

  for (int i=0;i<fab4->functions.size();i++)
    fs.insert(fs.end(),fab4->functions[i]->name);
  for (int i=0;i<fab4->functions.size();i++)
    genCode(fab4->functions[i],result,fs,constr,arity);
  return result;
}
}
```

6.2.4 Compiler Hauptmethode

Es folgt eine zugegebener Maßen überfrachtete Hauptfunktion. Sie ruft den Parser auf, läßt Code generieren und schreibt für diesen eine Bytecodedatei. Zusätzlich werden noch drei fest verdrahtete Standardfunktionen generiert: `javaCall`, `staticJavaCall`, `getRuntime`. Diese erlauben es mit der Außenwelt zu kommunizieren.

GenCode.cpp
```
using namespace fab4;

int main(int argc,char* args[]){
 try {
  string a1 = args[1];
  ParsResult<Fab4*,Token>* res = parsFab4(a1+".fab4");
  string a1gmc=a1+".gmc";
  char fi [a1gmc.size()];
  fi [a1gmc.size()]='\0';
  a1gmc.copy(fi,a1gmc.size());
  ofstream out(fi,ios::out);

  out<<FA<<B4;

  cout << show(res->getFurtherToken())<<endl;

```

```
  Fab4* fab4=res->getResult();
  writeString(fab4->name,out);

  std::map<string,int> constr;
  std::map<string,int> arity;
  for (int i=0;i<fab4->constructors.size();i++){
    constr[fab4->constructors[i]->name]=i;
    arity[fab4->constructors[i]->name]=fab4->constructors[i]->arity;
  }

  vector<string> fs;
  vector<Instruction*>* code=new vector<Instruction*>();

  for (int i=0;i<fab4->functions.size();i++)
    fs.insert(fs.end(),fab4->functions[i]->name);

  // insert standard functions into machine
  fs.insert(fs.end(),"staticJavaCall");
  fs.insert(fs.end(),"javaCall");
  fs.insert(fs.end(),"getRuntime");

  writeInt(fs.size(),out);

  writeString("staticJavaCall",out);
  writeInt(0,out);
  writeInt(2,out);
  code->insert(code->end(),new Push(1));
  code->insert(code->end(),new Eval());
  code->insert(code->end(),new Push(1));
  code->insert(code->end(),new Eval());
  code->insert(code->end(),new StaticJavaCall());
  code->insert(code->end(),new Unwind());

  writeString("javaCall",out);
  writeInt(code->size(),out);
  writeInt(3,out);
  code->insert(code->end(),new Push(2));
  code->insert(code->end(),new Eval());
  code->insert(code->end(),new Push(2));
  code->insert(code->end(),new Eval());
  code->insert(code->end(),new Push(2));
  code->insert(code->end(),new Eval());
  code->insert(code->end(),new JavaCall());
  code->insert(code->end(),new Unwind());

  writeString("getRuntime",out);
  writeInt(code->size(),out);
  writeInt(0,out);
  code->insert(code->end(),new GetRuntime());
  code->insert(code->end(),new Unwind());
```



```

  // write function table and generate code
  for (int i=0;i<fab4->functions.size();i++){
    writeString(fab4->functions[i]->name,out);
    writeInt(code->size(),out);
    writeInt(fab4->functions[i]->args.size(),out);

    genCode(fab4->functions[i],code,fs,constr,arity);
  }

  // write byte code for constructor table
  writeInt(fab4->constructors.size(),out);
  for (int i =0;i<fab4->constructors.size();i++)
    writeString(fab4->constructors[i]->name,out);

  //write byte code for functions
  writeInt(code->size(),out);
  for (int i=0;i<code->size();i++){
    ((*code)[i])->write(out);
  }

  out<<FA<<B4<<flush;
  cout<<show(code)<<endl;
 }catch (string s){
  cout << "error: "<<s<<endl;
 }
}
```

6.3 Fab4 Beispiele

See how they run.
Lennon/McCartney

6.3.1 Primzahlengenerierung

Ein Verfahren zur Generierung aller Primzahlen ist unter den Namen *Sieb des Eratosthenes* bekannt. Ausgegangen wird von der aufsteigenden Folge aller natürlichen Zahlen größer 1: $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, \ldots\}$

Nun werden aus dieser Folge nach und nach die Zahlen ausgesiebt, die durch die erste Zahl teilbar sind. Zunächst werden also alle gerade Zahlen ausgesiebt: $\{2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33 \ldots\}$

Dann werden aus dieser Folge die Vielfachen der nächsten Zahl, also der 3, ausgesiebt: $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47 \ldots\}$

Dieser Prozeß wird jeweils rekursiv fortgesetzt. Wir erhalten nach und nach die Folgen: $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59 \ldots\}$

$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59 \ldots\}$
...

Dieses Verfahren läßt sich relativ einfach in Fab4 implementieren. Zunächst brauchen wir wieder bool'sche Werte und Listen als Datenstrukturen:

Eratosthenes.fab4
```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
```

Für die Negation haben wir keinen eingebauten Operator und schreiben hierfür eine Funtion:

Eratosthenes.fab4
```
not x = case x of True->(False);False->(True);
```

Die Funktion `from` erzeugt die aufsteigende Liste aller natürlichen Zahlen ausgehend von einer Startzahl:

Eratosthenes.fab4
```
from x = Cons x (from (x+1));
```

Die Funktion `filter` ermöglicht es aus einer bestehenden Liste über ein Prädikat bestimmte Elemente herauszufiltern:

Eratosthenes.fab4
```
filter p xs
  = case xs of
     Nil -> (Nil);
     Cons y ys -> case (p y) of False -> filter p ys;
                               True-> (Cons y (filter p ys));
```

Wir haben keine Modulooperator in Fab4 und implementieren eine Funktion, die testet, ob die zweite Zahl Vielfaches der ersten ist:

Eratosthenes.fab4
```
notteilbar y x  = not ((x - ((x/y)*y)) = 0);
```

Damit haben wir alles, um das Sieb des Eratosthenes zu implementieren:

Eratosthenes.fab4
```
sieb xs
 = case xs of
    Cons y ys -> (Cons y (sieb (filter (notteilbar y) ys)));
```

Um die die Liste aller Primzahlen zu erlangen, wenden wir das Sieb auf die Liste aller natürlichen Zahlen größer 1 an:

Eratosthenes.fab4
```
main=sieb(from 2)
```


Und tatsächlich, wenn wir das Programm durch die G-Maschine ausführen lassen, wird eine Liste von Primzahlen auf der Konsole ausgegeben:

```
sep@linux:fh/compiler/tutor> fab4.sh src/Eratosthenes 2> out.txt
(Cons 2 (Cons 3 (Cons 5 (Cons 7 (Cons 11 (Cons 13 (Cons 17 (Cons 19 (Cons
23 (Cons 29 (Cons 31 (Cons 37 (Cons 41 (Cons 43 (Cons 47 (Cons 53 (Cons 5
9 (Cons 61 (Cons 67 (Cons 71 (Cons 73 (Cons 79 (Cons 83 (Cons 89 (Cons 97
 (Cons 101 (Cons 103 (Cons 107 (Cons 109 (Cons 113 (Cons 127 (Cons 131 (C
ons 137 (Cons 139 (Cons 149 (Cons 151 (Cons 157 (Cons 163 (Cons 167 (Cons
 173 (Cons 179 (Cons 181 (Cons 191 (Cons 193 (Cons 197 (Cons 199 (Cons 211
 (Cons 223 (Cons 227 (Cons 229 (Cons 233 (Cons 239 (Cons 241 (Cons 251 (Co
ns 257 (Cons 263 (Cons 269 (Cons 271 (Cons 277 (Cons 281 (Cons 283 (Cons 2
93 (Cons 307 (Cons 311 (Cons 313 (Cons 317 (Cons 331 (Cons 337 (Cons 347 (
Cons 349 (Cons 353 (Cons 359 (Cons 367 (Cons 373 (Cons 379 (Cons 383 (Cons
 389 (Cons 397 (Cons 401 (Cons 409 (Cons 419 (Cons 421 (Cons 431 (Cons 433
 (Cons 439 (Cons 443 (Cons 449 (Cons 457 (Cons 461 (Cons 463 (Cons 467 (Co
ns 479 (Cons 487 (Cons 491 (Cons 499 (Cons 503 (Cons 509 (Cons 521 (Cons 5
23 (Cons 541 (Cons 547 (Cons 557 (Cons 563 (Cons 569 (Cons 571 (Cons 577 (
Cons 587 (Cons 593 (Cons 599 (Cons 601 (Cons 607 (Cons 613 (Cons 617 (Cons
 619 (Cons 631 (Cons 641 (Cons 643 (Cons 647 (Cons 653 (Cons 659 (Cons 661
 (Cons 673 (Cons 677 (Cons 683 (Cons 691 (Cons 701 (Cons 709 (Cons 719 (Co
ns 727 (Cons 733 (Cons 739 (Cons 743 (Cons 751 (Cons 757 (Cons 761 (Cons 7
69 (Cons 773 (Cons 787 (Cons 797 (Cons 809 (Cons 811 (Cons 821 (Cons 823 (
Cons 827 (Cons 829 (Cons 839 (Cons 853 (Cons 857 (Cons 859 (Cons 863 (Cons
 877 (Cons 881 (Cons 883 (Cons 887 (Cons 907 (Cons 911 (Cons 919 (Cons 929
 (Cons 937 (Cons 941 (Cons 947 (Cons 953 (Cons 967 (Cons 971 (Cons 977 (Co
ns 983 (Cons 991 (Cons 997 (Cons 1009 (Cons 1013 (Cons 1019 (Cons 1021 (Co
ns 1031 (Cons 1033 (Cons 1039
sep@linux:fh/compiler/tutor>
```

Kapitel 7

Das Münchhausenprinzip: Bootstrapping

Just like starting Over.
Lennon

In diesem Kapitel wollen wir nocheinmal ganz von vorne anfangen. Ein erster Compiler der Sprache Fab4 ist geschrieben. Jetzt wollen wir einen Compiler der Sprache Fab4 in Fab4 implementieren. Damit erhalten wir einen Compiler, der mit sich selbst übersetzt werden kann. Man spricht von einem *boot strap*. Der Begriff ist das englische Bild dafür, sich selbst hochzuziehen, an den Schnürsenkeln der Stiefel sich emporzuheben. Im Deutschen mag man da eher an die Geschichte des Baron Münchhausen denken, der sich an den eigenen Haaren aus den Sumpf gezogen haben will.

Die meisten Compilerbauer, wollen sich möglichst schnell unabhängig von anderen Sprachen machen, und am liebsten nur noch in ihrer Sprache entwickeln und nur noch ihren eigenen Compiler enutzen. Die meisten allgemeinen Programmiersprachen haben einen Comiler, der in der Sprache selbst entwickelt wurde.

7.1 Java Calls in Fab4

Um einen Compiler in Fab4 zu implementieren, müssen wir ein minimales Quantum an I/O-Operationen durchführen; zumindest ist ja eine Quelltextdatei zu lesen und eine Bytecodedatei zu schreiben. Fab4 kommt mit keinerlei eigenen I/O-Bibliotheken daher; aber wir haben in Fab4 eine Möglichkeit vorgesehen, externen Javaprogramme aufzurufen. Bevor wir im folgenden Kapitel beginnen, den Fab4 Compiler zu implemenmtieren, betrachten wir die Möglichkeit der *foreign function calls* in Fab4.

Fab4 hat zwei fest eingebaute Funktionen, um Aufrufe von Javacode zu generieren:

- `staticJavaCall`,
- `javaCall`,

Die beiden Funktionen unterscheiden sich darin, ob die zu rufenden Methode eine statische Javamethode oder eine Objektmethode ist.

Diese beiden Funktionen haben einen ersten Parameter vom Typ `int`, der die Parameteranzahl der zu rufenden Javamethode angibt. Dann folgt ein zweiter Parameter, der als String codiert den Namen der zu rufenden Javamethode. Für statische Methoden ist deser Name vollqualifiziert mit Paket und Klassenangabe zu spezifizieren. Für Objektmethoden nur der einfache Name der methode. Anschließend folgen die Argumente für den Aufruf der Javamethode. Handelt es sich bei der Javamethode um eine Objektmethode ist das erster dieser Argumente das `this`-Objekt.

7.1.1 Typen

Prinzipiell können beim Aufruf von von Javamethoden alle Objekttypen Javas verwendet werden. Für ein paar bestimmte Java-Typen findet eine Typkonvertierung statt. Sie werden nicht im Fab4 Heap als Javaobjekte gespeichert, sondern in Fab4 eigenen Datenstrukturen übersetzt. Man spricht hierbei auch vom sogenannten *marshalling*.

Folgende Typumwandlung von Java nach Fab4 und zurück werden vorgenommen:

- Objekte des Javatyps `String` werden als Liste von `char` im Fab4-Heap dargestellt. Hierzu werden die Konstruktoren `Nil` und `Cons` verwendet.
- Auch Javas Arrays werden als Liste von im Fab4-Heap dargestellt.
- Java Objekte des Typs `Boolean` werden als Konstruktorknoten im Fab4 Heap dargestellt. Hierzu werden die Werte `true` und `false` durch die Fab4 Konstruktoren`True` und `False` repräsentiert.
- Objekte der Javaklasse `name.panitz.util.Pair` werden durch den Konstruktorknoten `Pair` im Fab4 Heap dargestellt.
- Daten von Typ `Integer` und `Character` können mit `NNum` bzw. `NChar` im Fab4 Heap dargestellt werden.

Im folgenden seinen verschiedene Beispiele zum Aufruf von Javamethoden gegeben.

7.1.2 Rückgabe

Da Fab4 eine rein funktionale Sprache ist, wird von jedem Ausdruck verlangt, daß er einen Rückgabetyp hat. Daher müssen auch die Aufrufe von `javCall` und `staticJavaCall` ein Ergebnis liefern. Es folgt, daß keine `void` Methoden von Java aufgerufen werden können. Möchte man trotzdem eine `void`-Methode von Java nutzen, ist diese in einer anderen Javamehtode zu wrappen, die zumindest irgendein Ergebnis als Rückgabe vorsieht.

Ansonsten können alle Javamethoden, die keinen primitiven Typen als Rückgabe haben, gerufen werden. Betrachten wir ein erstes Beispiel. Hierzu benutzen wir folgende statische parameterlose Javamethode die einen `String` als Rückgabe liefert.

GetHello.java
```
package hell;
public class GetHello{
  public static String getHello(){return "hello";}
}
```

Das folgende Fab4-Programm ruft die Javamethode `getHello` auf.

GetHello.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;

main = staticJavaCall 0 "hell.GetHello.getHello"
```

Es ist darauf zu achten, daß im Fab4 Proramm die Standardkonstruktoren für die in Java benutzen Typen auch definiert wurden.

tatsächlich läßt sich dieses Programm mit unserem Fab4 Interpreter ausführen:

```
sep@pc216-5:~/fh/compiler/tutor/src> javac -d . GetHello.java
sep@pc216-5:~/fh/compiler/tutor/src> fab4c GetHello
[PUSH 1,EVAL,PUSH 1,EVAL,STATICJAVACALL,UNWIND,PUSH 2,EVAL,PUSH 2,EVAL
,PUSH 2,EVAL,JAVACALL,UNWIND,GETRUNTIME,UNWIND,PACK 2 0,PUSHCHAR o
,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR e,PACK 3 2,PUSHCHAR H
,PACK 3 2,PUSHCHAR t,PACK 3 2,PUSHCHAR e,PACK 3 2,PUSHCHAR g,PACK 3 2,PUSHCHAR .
,PACK 3 2,PUSHCHAR o,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR e
,PACK 3 2,PUSHCHAR H,PACK 3 2,PUSHCHAR t,PACK 3 2,PUSHCHAR e,PACK 3 2,PUSHCHAR G
,PACK 3 2,PUSHCHAR .,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR l,PACK 3 2,PUSHCHAR e
,PACK 3 2,PUSHCHAR h,PACK 3 2,PUSHINT 0,PUSHGLOBAL staticJavaCall,MKAP,MKAP,UNWIND]
sep@pc216-5:~/fh/compiler/tutor/src> fab4 GetHello
(Cons 'h' (Cons 'e' (Cons 'l' (Cons 'l' (Cons 'o' (Nil))))))
sep@pc216-5:~/fh/compiler/tutor/src>
```

Folgendes Programm können wir benutzen, um alle verschiedenen Typvarianten einer Javarückgabe einmal zu testen.

GetMore.java

```
package hell;
import name.panitz.util.Pair;
public class GetMore{
  public static Integer getInt(){return 42;}
  public static Boolean getBool(){return true;}
  public static Pair<String,Integer> getPair(){
    return new Pair<String,Integer>("hallo",42);}
  public static Integer[] ints = {1,2,3,4,5,6,7,8,9,10};
  public static Integer[] getInts(){return ints;}
  public static Object getObject(){
    return new javax.swing.JFrame("fenster");}
}
```

Das folgende Javaprogramm ruft alle diese Funktionen einmal auf:

GetMore.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
```



```
constr Pair 2;

main =  Cons (staticJavaCall 0 "hell.GetMore.getInt")
       (Cons (staticJavaCall 0 "hell.GetMore.getBool")
       (Cons (staticJavaCall 0 "hell.GetMore.getPair")
       (Cons (staticJavaCall 0 "hell.GetMore.getInts")
       (Cons (staticJavaCall 0 "hell.GetMore.getObject")
       (Nil)))))
```

Dieses Programm führt zu folgender Ausgabe:

```
sep@pc216-5:~/fh/compiler/tutor> javac -cp classes -d classes/ src/hell/GetMore.java
sep@pc216-5:~/fh/compiler/tutor> fab4 src/GetMore
(Cons 42 (Cons (True) (Cons (Pair (Cons 'h' (Cons 'a' (Cons 'l' (Cons 'l' (Cons 'o' (Nil)))))
 42) (Cons (Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 (Cons 6 (Cons 7 (Cons 8 (Cons 9 (Cons 10 (Nil)))))))))))
 (Cons javax.swing.JFrame[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,title=fenster
,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.JRootPane[,0,0,0x0,invalid
,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=16777673
,maximumSize=,minimumSize=,preferredSize=],rootPaneCheckingEnabled=true] (Nil))))))
sep@pc216-5:~/fh/compiler/tutor>
```

7.1.3 Aufruf mit Parametern

Schließlich können wir natürlich auch Javametoden aufrufen, die Parameter enthalten. Als kleines Beispiel sehen wir eine statische Methode vor, die ein Fenster mit einem als Parameter übergebenen String als Fenstertitel öffnet.

GetJFrame.java

```
package hell;
import javax.swing.*;

public class GetJFrame {
  static public JFrame getOpenFrame(String title){
    JFrame result = new JFrame(title);
    result.setSize(200,100);
    result.setVisible(true);
    return result;
  }
}
```

Das folgende kleine Fab4-Programm, öffnet ein Fenster, ruft auf diesen Fensterobjekt die Methode `getTitle` auf, um auf das erhaltene Stringobjekt die Objektmethode `toUpperCase` aufzurufen:

GetJFrame.fab4

```
constr True 0;
constr False 0;
constr Nil 0;
constr Cons 2;
constr Pair 2;

```

```
main =
  javaCall 0 "toUpperCase"
  (javaCall 0   "getTitle"
  (staticJavaCall 1 "hell.GetJFrame.getOpenFrame" "hello world"))
```

Startet man dieses Programm, wird tatsächlich ein Fenster geöffnet und der Titel des Fenster als Fab4-Liste nach Großbuchstaben konvertiert auf der Kommandozeile ausgegeben.

7.1.4 Java Hilfsklassen für I/O im Fab4 Compiler

Jetzt, da wir damit vertraut sind, wie Javafunktionen in Fab4 aufgerufen werden können, legen wir uns Javaklassen an, die das für uns notwendige I/O übernehmen.

Lesen des Quelltextes

Zunächst eine Klasse zum Lesen der Quelltextdatei `GmFileReader`. Wir sehen drei Methoden vor:

- die statische Methode `getInstance`, die für einen Dateinamen einen `GmFileReader` erzeugt.
- eine Objektmethode `next`, die ein Paar zurückgiebt: das nächste Zeichen im Datenstrom, und das `GmFileReader`-Objekt selbst.
- eine Objektmethode `hasNext`, die ein Paar zurückgiebt: der bool'schen Wert, ob in der Datei ein weiteres Zeichen steht, und das `GmFileReader`-Objekt selbst.

Die Funktionen geben nicht nur den Wert zurück, sondern auch noch das `GmFileReader`-Objekt selbst. Dieses ist notwendig, um sicherzustellen, daß wir in Fab4 wirklich sequentiell nacheinander alle Zeichen bekommen. Die *lazy* Semantik könnte uns Aufrufe auf das`GmFileReader`-Objektwegoptimieren. Auch haben wir keine Reihenfolge definiert, in der Ausdrücke auszuwerten sind. Diese können wir erzwingen, indem wir das `GmFileReader`-Objekt durch die Funktionsaufrufe von `next` durchreichen. Man beachte, daß der Aufruf von `next` ja jeweils dieses Objekt verändert.

GmFileReader.java

```
package name.panitz.gm;

import java.io.*;
import java.util.*;

import name.panitz.util.Pair;

public class GmFileReader {
  Reader in=null;
  int next;
  private  GmFileReader(){};

  public static  GmFileReader getInstance(String fileName)
```



```
                                           throws Exception{
    GmFileReader result = new GmFileReader();
    result.in=new FileReader(fileName);
    result.next=result.in.read();
    return result;
  }

  public Pair<Boolean,GmFileReader> hasNext(){
    return new Pair<Boolean,GmFileReader>(next>=0,this);
  }

  public Pair<Character,GmFileReader> next()throws Exception{
    final char c=(char)next;
    next=in.read();
    return new Pair<Character,GmFileReader>(c,this);
  }
}
```

Schreiben der Bytecodedatei

Zum Schreiben der Bytecodedatei verfahren wir ähnlich. Alle Schreiboperationen geben den Strom, auf dem geschrieben wurde wieder zurück, damit auf diesem weitere Operationen ausgeführt werden können:

GmWriter.java

```
package name.panitz.gm;

import java.io.*;
import java.util.*;

public class GmWriter {
  DataOutputStream out=null;
  private GmWriter(){};

  public static GmWriter getInstance(String fileName)
                                         throws Exception{
    GmWriter result = new GmWriter();
    result.out
     =new DataOutputStream(new FileOutputStream(fileName));
    return result;
  }

  public GmWriter write(Integer i)throws Exception{
    out.write((byte)i.intValue());return this;}

  public GmWriter write(Character i)throws Exception{
    out.writeChars(""+i.charValue());return this;}

  public GmWriter writeInt(Integer i)throws Exception{
```

```
    out.writeInt(i);return this;}

  public GmWriter writeChars(String i)throws Exception{
    out.writeChars(i);return this;}

  public static String error(String message)throws Exception{
    throw new Exception(message);}

  public static Boolean print(String message)throws Exception{
    System.out.println(message);
    return true;
  }

  public static Object trace(String message,Object o)throws Exception{
    System.out.println(message+o);
    return o;
  }
  public static Object trace(String message,String o)throws Exception{
    System.out.println(message+o);
    return o;
  }
```

Die folgenden statischen Methoden werden in einem anderen Kontext benutzt werden. Sie erlauben es in eine Textdatei zu schreiben. Sie werden für die Variante des Fab4 Compilers benutzt, der C Code, statt Bytecode generiert. Hier wird als Ergebnistyp der Dateiname benutzt, der dann bei weiteren Aufrufen als Argument eingesetzt wetrden kann.

GmWriter.java

```
  public static String writeFile(String fileName,String content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName);
    f.write(content);
    f.close();
    return fileName;
  }

  public static String appendFile(String fileName,String content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content);
    f.close();
    return fileName;
  }
  public static String appendFile(String fileName,int content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content+"");
    f.close();
    return fileName;
  }
```



```
  public static String appendFile(String fileName,char content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content+"");
    f.close();
    return fileName;
  }
  public static String appendFile(String fileName,Object content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content+"");
    f.close();
    return fileName;
  }
  public static String appendFile(String fileName,Integer content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content+"");
    f.close();
    return fileName;
  }
  public static String appendFile(String fileName,Character content)
                        throws Exception{
    FileWriter f= new FileWriter(fileName,true);
    f.write(content+"");
    f.close();
    return fileName;
  }
}
```

7.2 Fab4C

Starten wir mit der Implementierung von Fab4 in Fab4.

7.2.1 Bool'sche Werte

Zunächst sehen wir Bool'sche Werte vor. Dabei benötigen wir die einfachsten Funktionen auf diesen: ein `and`, `or`, `not` und `if`.

fab4c.fab4

```
constr True 0;
constr False 0;

if cond a1 a2 = case (cond) of True -> a1;False->a2;

and x y = case (x) of False -> (False); True -> y;
or x y = case x of True -> (True); False -> y;
not x = case x of True -> (False); False -> (True);
```

7.2.2 Listen

Wir werden vielfach einfach verkettete Listen benötigen. Es folgen eine Reihe von Standardfunktionen auf diesen, wie `last`, `length`, `append`, `concat`...

fab4c.fab4

```
constr Nil 0;
constr Cons 2;

head xs = case xs of Cons y ys -> y;
tail xs = case xs of Cons y ys -> ys;
isEmpty xs = case xs of Nil ->(True);Cons -> (False);

last xs = case xs of
  Cons y ys -> (case ys of
    Cons z zs -> last ys;
    Nil -> y);

length xs = case xs of Nil -> 0;Cons y ys-> 1+ (length ys);

append xs ys = case xs of
  Nil -> ys;
  Cons z zs -> (Cons z (append zs ys));

concat xss = case xss of
  Nil -> (Nil);
  Cons ys yss -> append ys (concat yss);

reverse xs = case xs of
  Nil ->(Nil);
  Cons y ys -> append (reverse ys) (Cons y (Nil));

contains comp x xs = case xs of
  Nil -> (False);
  Cons y ys -> if (comp x y) (True) (contains comp x ys);

from n = (Cons n (from (n+1)));

listMap f xs = case xs of
  Nil ->(Nil);
  Cons y ys->(Cons (f y) (listMap f ys));
```

7.2.3 Paare

Wir haben in Java schon gesehen, wie nützlich eine allgemeine Möglichkeit Daten zu Paaren zusammenzufassen sind. Insbesondere läßt sich über eine Liste von Paaren eine Abbildung realisieren. Wir definieren uns daher Paare und ein paar nützliche Funktionen für diese.

fab4c.fab4

```
constr Pair 2;
fst p = case p of Pair e1 e2 -> e1;
```



```
snd p = case p of Pair e1 e2 -> e2;

stringEq xs ys = case xs of
  Nil -> (case ys of Nil -> (True);
                   Cons y1 ys1 -> (False));
  Cons x1 xs1 -> (case ys of Nil -> (False);
                   Cons y1 ys1 -> and (y1=x1) (stringEq xs1 ys1));

containsStr x xs = contains stringEq x xs;

lookUp x ps = case ps of
  Cons p ys -> (case p of
    Pair key value -> (case (stringEq ( key) x) of
      True -> value;
      False-> lookUp x ys));
  Nil -> abort (append (append ("not in environment: ") x) ("!"));

zip xs ys = case xs of
  Nil -> (Nil);
  Cons x1 xs1 ->(case ys of
    Nil -> (Nil);
    Cons y1 ys1 -> (Cons (Pair x1 y1)(zip xs1 ys1)) );

getPosition i str xs = case xs of
  Cons y ys -> (case (stringEq str y) of
    True -> i;
    False-> getPosition (i+1) str ys);
```

7.2.4 Either

Die Unterklassen `Left` und `Right` haben in der `ParsLib` den Typ `Either` gebildet. Auch diesen wollen wir in Fab4 implementieren. Wir brauchen die zwei Konstruktoren, sowie eine Selektorfunktion:

fab4c.fab4

```
constr Left  1;
constr Right 1;

getLeftRight lr = case lr of
  Left l  -> l;
  Right r -> r;
```

7.2.5 Parser Bibliothek

Eine Fab4 Implementierung unserer Parserbibliothek haben wir bereits im Beispiel `Arith` gesehen.

fab4c.fab4

```
constr ParsResult 2;
constr Fail 1;
```

```

parsCharToken tok xs = case xs of
  Nil        -> (Fail xs);
  Cons y ys  -> if (y = tok) (ParsResult y ys)(Fail xs);

parsStringToken tok xs = case xs of
  Nil        -> (Fail xs);
  Cons y ys  -> if (stringEq y tok)(ParsResult y ys)(Fail xs);

getToken xs
  = case xs of Nil -> (Fail xs);Cons y ys ->(ParsResult y ys);

seq p1 p2 xs = case (p1 xs) of
  Fail ys    -> (Fail xs);
  ParsResult res further -> (case (p2 further) of
    Fail zs -> (Fail xs);
    ParsResult res2 fs2 -> (ParsResult (Pair res res2) fs2));

alt p1 p2 xs = case (p1 xs) of
  Fail ys  -> (case  (p2 xs) of
    Fail zs  -> (Fail xs);
    ParsResult l fs1 ->  (ParsResult (Right l) fs1));
  ParsResult res further -> (ParsResult (Left res) further);

ualt p1 p2 xs = case (p1 xs) of
  Fail ys -> (case  (p2 xs) of
    Fail zs  -> (Fail xs);
    ParsResult l fs1 ->  (ParsResult l fs1));
  ParsResult res further -> (ParsResult res further);

map f p xs =  case (p xs) of
  Fail ys -> (Fail xs);
  ParsResult res further -> (ParsResult (f res) further);

rep0 p1 xs = case (p1 xs) of
  Fail fs1 -> (ParsResult (Nil) xs);
  ParsResult res fs2 -> (case (rep0 p1 fs2) of
    ParsResult res2 fs3 -> (ParsResult (Cons res res2) fs3));

consPair pa = case pa of Pair x xs ->(Cons x xs);

rep1 p1 xs = map consPair (seq p1 (rep0 p1)) xs;

return x xs = (ParsResult x xs);

sepBy p sep xs=map consPair (seq p (rep0 (map snd (seq sep p)))) xs;
sepBy0 p sep xs= ualt  (sepBy p sep) (return (Nil)) xs;
```


Wir werden keinen eigenen Tokenizer schreiben, sondern auch den Tokenizer über die Grammatik ausdrücken. Unser Parser wird als Tokentyp nur den Typ `char` benutzen. Daher erweitern wir die Bibliothek noch um ein paar einfache Funktionen.

Zunächst zum Parsen eines bestimmten Schlüsselwortes:

fab4c.fab4

```
pString str xs
 = case str of
     Nil -> (ParsResult str xs);
     Cons s tr -> (case (parsCharToken s xs) of
         Fail f -> (Fail xs);
         ParsResult c further -> (case (pString tr further) of
              Fail f2 -> (Fail xs);
              ParsResult cs further2
                      -> (ParsResult (Cons c cs) further2)));
```

Zusätzlich müssen wir auch die Möglichkeit haben Leerzeichen zu überspringen:

fab4c.fab4

```
whiteChars = (" \n\t");

oneOfTheseChars str xs
 = case str of
    Cons s tr -> (case tr of
        Nil    -> parsCharToken s xs;
        Cons t r -> ualt (parsCharToken s) (oneOfTheseChars tr) xs);

whiteChar xs = oneOfTheseChars whiteChars xs;

whiteSpace xs  = rep0 whiteChar xs;

deleteWhite p xs = map snd (seq whiteSpace p) xs;
```

7.2.6 Fab4 Parser

Fab4 Konstruktoren

fab4c.fab4

```
constr Fab4 2;
constr ConstrDef 2;
constr FunDef 3;

constr VarExpr 1;
constr ConstructorCall 2;
constr NumExpr 1;
constr CharExpr 1;
constr AppExpr 2;
constr CaseExpr 2;
constr OpExpr 3;

```

```
constr CaseAlt 3;

nilName = ("Nil");
consName = ("Cons");
```

Tokenizer

Es folgend die Tokenizerfunktionen für die Fab4 Grammatik. zunächst Schlüsselworte, Zahlen und Variablennamen:

fab4c.fab4

```
keywords = (Cons ("constr") (Cons ("case") (Cons ("of") (Nil))));

constrP xs   = deleteWhite (pString ("constr")) xs;
caseP xs     = deleteWhite (pString ("case"))   xs;
ofP xs       = deleteWhite (pString ("of"))     xs;
arrowP xs    = deleteWhite (pString ("->"))     xs;
semicolonP xs= deleteWhite (pString (";"))      xs;
commaP xs    = deleteWhite (pString (","))      xs;
lparP xs     = deleteWhite (pString ("("))      xs;
rparP xs     = deleteWhite (pString (")"))      xs;
lbraP xs     = deleteWhite (pString ("["))      xs;
rbraP xs     = deleteWhite (pString ("]"))      xs;
addP xs      = deleteWhite (pString ("+"))      xs;
subP xs      = deleteWhite (pString ("-"))      xs;
multP xs     = deleteWhite (pString ("*"))      xs;
divP xs      = deleteWhite (pString ("/"))      xs;
leP xs       = deleteWhite (pString ("<="))     xs;
geP xs       = deleteWhite (pString (">="))     xs;
ltP xs       = deleteWhite (pString ("<"))      xs;
gtP xs       = deleteWhite (pString (">"))      xs;
eqP xs       = deleteWhite (pString ("="))      xs;

cNameP xs = deleteWhite
 (map consPair (seq upperCaseLetterP (rep0 symbolP))) xs;

nameP xs = case
  (deleteWhite (map consPair (seq lowerCaseLetterP (rep0 symbolP))) xs)
 of Fail f -> Fail xs;
    ParsResult y ys
           -> if (containsStr y keywords)(Fail xs)(ParsResult y ys);

numberP xs   = map mkNum (deleteWhite (rep1 digitP)) xs;
mkNum x = (NumExpr (toNum x));
toNum xs = toNumAux 0 xs;

toNumAux n xs = case xs of
  Nil -> n;
  Cons c cs -> toNumAux (n*10 +(c+0-48)) cs;

```



```
upperCaseLetterP xs=oneOfTheseChars("ABCDEFGHIJKLMNOPQRSTUVWXYZ")xs;
lowerCaseLetterP xs=oneOfTheseChars("abcdefghijklmnopqrstuvwxyz")xs;
letterP xs         =ualt upperCaseLetterP lowerCaseLetterP xs;
digitP xs          =oneOfTheseChars ("0123456789") xs;
symbolP xs         =ualt digitP letterP xs;
```

Schließlich noch Stringliterale und Zeichenliterale:

fab4c.fab4

```
stringLiteralP xs = map mkStringLiteral stringLitP xs;
mkStringLiteral str = case str of
  Nil -> (ConstructorCall nilName (Nil) );
  Cons c cs
      -> (ConstructorCall consName
             (Cons (CharExpr c) (Cons (mkStringLiteral cs) (Nil))));

stringLitP xs = case (pString ("\"") xs) of
  Fail f -> (Fail xs);
  ParsResult c cs -> (case (getStringLit cs) of
    Fail f2 -> (Fail xs);
    ParsResult str further ->(ParsResult str further));

getStringLit xs = case (getToken xs) of
  Fail rest -> (Fail xs);
  ParsResult y ys  -> (case (y = (head ("\\"))) of
    True -> (case (getEscapeChar ys) of
      Fail f ->(Fail xs);
      ParsResult c cs -> (ParsResult c cs) );
    False-> (case (y=(head ("\""))) of
      True -> (ParsResult (Nil) ys);
      False-> getFurtherChars y ys xs));

getFurtherChars c cs xs = case (getStringLit cs) of
  Fail rest -> (Fail xs);
  ParsResult y ys -> (ParsResult (Cons c y) ys);

getEscapeChar xs = case (getEscape xs) of
  Fail ys -> (Fail xs);
  ParsResult  c cs -> getFurtherChars c cs xs;
```

Ebenso Zeichenliterale. Auch für diese werden die Escapesequenzen berücksichtigt:

fab4c.fab4

```
charLiteralP xs = case (pString ("'") xs) of
  Fail f -> (Fail xs);
  ParsResult c1 cs1 -> (case (getCharLit cs1) of
    Fail f2 -> (Fail xs);
    ParsResult c2 cs2 -> (case (pString ("'") cs2) of
      Fail f3 ->(Fail xs);
      ParsResult c3 cs3 -> (ParsResult (CharExpr c2) cs3)));
```

```

getEscape xs = case (getToken xs) of
  Fail ys -> (Fail xs);
  ParsResult  c cs
  -> if (c=('n')) (ParsResult (head ("\n")) cs)
     (if (c=('t')) (ParsResult (head ("\t")) cs)
     (if (c=(head("\""))) (ParsResult (head ("\"")) cs)
     (if (c=(head("\\"))) (ParsResult (head ("\\")) cs) (Fail xs))));

getCharLit xs =  case (getToken xs) of
  Fail f -> (Fail xs);
  ParsResult c1 cs1
    -> if (c1=(head ("\\")))(getEscape cs1)(ParsResult c1 cs1);
```

Wir erweitern unsere Sprache gegenüber der C Implementierung und sehen ein weiteres Literal vor. Eine kompakte Schreibweise für Listen. Wir benutzen die aus den Sprachen *Clean* und *Haskell* begräuchliche Syntax für Listen. Ein Listliteral ist eine durch Kommas getrennte Liste von Ausdrücken innerhalb eines Paares eckiger Klammern. Wir können damit statt

```
main = (Cons 1 (Cons 2 (Cons 3 (Nil))))
```

einfach

```
main = [1,2,3]
```

schreiben. Leider können wir diese neue Syntax in dieser Implementierung noch nicht verwenden, da wir diesen Bootstrapcompiler erst mit den in C++ geschriebenen Compiler übersetzen müssen. Dieser kennt noch keine Listliterale.

fab4c.fab4

```
listLiteralP xs
 = map mkListLiteral
   (map snd (seq lbraP
              (map fst (seq (sepBy0 exprP commaP) rbraP) ))) xs;

mkListLiteral xs = case xs of
  Nil -> (ConstructorCall nilName (Nil) );
  Cons y ys -> (ConstructorCall consName
                 (Cons y (Cons (mkListLiteral ys) (Nil))));
```

Grammatikregeln

Mit der Kombinatorparserbibliothek lassen sich die Regeln für die Grammatik direkt umsetzen. Allerdings haben wir in Fab4 (noch) nihcht die Möglichkeit Operatoren zu überladen. Daher sehen die Umsetzungen der Regeln nicht ganz so elegant wie in C++ aus.

Zunächst die Regel 5.1 für ein fab4-Modul:

fab4c.fab4

```
fab4P xs = map mkFab4 (sepBy defP semicolonP) xs;

mkFab4 xs = case xs of
  Nil -> (Fab4 (Nil) (Nil));
  Cons y ys -> (case (mkFab4 ys) of
    Fab4 constrs funs -> (case y of
      Left fun -> (Fab4 constrs (Cons ( fun) funs));
      Right con-> (Fab4 (Cons con constrs) funs))) ;
```

Die Regel 5.2 für die Definitionsliste:

fab4c.fab4

```
defP xs = map fst (seq (alt funDefP constrDefP) whiteSpace) xs;
```

Die Regel 5.3 für Konstruktordefinitionen:

fab4c.fab4

```
constrDefP xs
 = map mkConstr (map snd (seq constrP (seq cNameP numberP))) xs;

mkConstr p = case p of Pair n i -> ConstrDef n i;
```

Die Regel 5.4 für ein Funktionsdefinitionen:

fab4c.fab4

```
funDefP xs
 = map mkFun (seq nameP (seq varsP (map snd (seq eqP exprP)))) xs;

mkFun p  = case p of
  Pair n varex -> (case varex of Pair var ex->FunDef n var ex);
```

Die Regel 5.5 für Variablenlisten:

fab4c.fab4

```
varsP xs = rep0 nameP xs;
```

Die Regel 5.6 für Ausdrücke:

fab4c.fab4

```
exprP xs = ualt constructorCallP (ualt caseExprP funCallP) xs;
```

Die Regel 5.7 für Konstruktoraufrufe:

fab4c.fab4

```
constructorCallP xs
 = map mkConstructorCall (seq cNameP (rep0 atomExprP)) xs;

mkConstructorCall p = case p of
  Pair n args -> (ConstructorCall n args);
```

Die Regel 5.8 für ein Case-Ausdrücke:

fab4c.fab4
```
caseExprP xs = map mkCaseExpr
   (map snd (seq caseP
   (seq exprP (map snd (seq ofP caseAltsP))))) xs;

mkCaseExpr p = case p of
  Pair expr alts -> (CaseExpr expr alts);
```

Die Regel 5.9 für ein Case-Alternativen:

fab4c.fab4
```
caseAltsP xs = sepBy caseAltP semicolonP xs;
```

Die Regel 5.10 für eine Case-Alternative:

fab4c.fab4
```
caseAltP xs = map mkCaseAlt
  (seq cNameP (seq varsP (map snd (seq arrowP exprP)))) xs;

mkCaseAlt p = case p of
  Pair c argsAlt -> (case argsAlt of
    Pair as e -> (CaseAlt c as e));
```

Die Regel 5.11 für einen atomaren Audruck. Hier schon ergänzt um das neue Konstrukt für Listliterale:

fab4c.fab4
```
atomExprP xs = ualt numberP
  (ualt charLiteralP
  (ualt stringLiteralP
  (ualt listLiteralP
  (ualt (map mkVar nameP) parExprP)))) xs;

mkVar x = (VarExpr x);
```

Die Regel 5.12 für geklammerte Ausdrücke :

fab4c.fab4
```
parExprP xs = map snd (seq lparP (map fst (seq exprP rparP))) xs;
```

Die Regel 5.13 für Funktionsanwendungen:

fab4c.fab4
```
funCallP xs = map mkFunCall (rep1 compExprP) xs;

mkFunCall xs =  case xs of Cons y ys -> mkFunCall2 y ys;

mkFunCall2 x xs =  case xs of
 Nil -> x;
 Cons y ys -> mkFunCall2 (AppExpr x y) ys;
```


Die Regel 5.14 für Vergleichsausdrücke:

fab4c.fab4
```
compExprP xs
 = map mkOp (seq addExprP (rep0 (seq compOpP addExprP))) xs;
```

Die Regel 5.15 für Additionsausdrücke:

fab4c.fab4
```
addExprP xs
 = map mkOp (seq multExprP (rep0 (seq addOpP multExprP))) xs;
```

Die Regel 5.16 für Multiplikationsausdrücke:

fab4c.fab4
```
multExprP xs
 = map mkOp (seq atomExprP (rep0 (seq multOpP atomExprP))) xs;
```

Die für die Operatorausdrücke benutzte Funktion zum Erzeugen des Syntaxbaums:

fab4c.fab4
```
mkOp p = case p of
  Pair e1 opes -> mkOpsAux e1 opes;

mkOpsAux e1 ps = case ps of
  Nil -> e1;
  Cons p ys ->(case p of Pair o e2 -> mkOpsAux (OpExpr e1 o e2) ys);
```

Die Regel 5.17 für Vergleichsoperatoren:

fab4c.fab4
```
compOpP xs = ualt eqP (ualt leP (ualt geP (ualt ltP gtP))) xs;
```

Die Regel 5.18 für Additionsoperatoren:

fab4c.fab4
```
addOpP xs = ualt addP subP xs;
```

Die Regel 5.19 für Multiplikationsoperatoren:

fab4c.fab4
```
multOpP xs = ualt multP divP xs;
```

7.2.7 Code Generierung

Der Fab4 Parser damit implementiert. Er erzeugt einen abstrakten Syntaxbaum. Für den kann jetzt die Codegenerierung implementiert werden.

Byte Code Konstruktoren

Zunächst definieren wir Konstruktoren für die Maschinenbefehle. Die meisten Konstruktoren sind nullstellig.

fab4c.fab4
```
constr End 0;
constr Jump 1;
constr Unwind 0;
constr Eval 0;
constr Print 0;
constr PrintString 0;
constr Output 1;
constr MkAp 0;
constr Add 0;
constr Sub 0;
constr Mult 0;
constr Div 0;
constr Le 0;
constr Ge 0;
constr Lt 0;
constr Gt 0;
constr Eq 0;
constr PushGlobal 1;
constr Push 1;
constr Pop 1;
constr PushInt 1;
constr PushChar 1;
constr Slide 1;
constr Update 1;
constr Pack 2;
constr CaseJump 1;
constr Split 1;
constr JavaCall 0;
constr StaticJavaCall 0;
constr GetRuntime 0;
```

Byte Code Konstanten

Konstante Funktionen benutzen wir, um die tatsächlichen Bytecodewerte der einzelnen Befehle festzulegen:

fab4c.fab4
```
bNOP         = 0;
bPOP         = 1;
bEND         = 2;
bJUMP        = 3;
bPUSH        = 4;
bUNWIND      = 5;
bEVAL        = 6;
```



```
bPRINT       = 7;
bPRINTSTRING = 9;
bOUTPUT      = 10;
bMKAP        = 11;
bJAVACALL    = 12;
bPUSHGLOBAL  = 13;
bPUSHINT     = 14;
bPUSHCHAR    = 15;
bSLIDE       = 16;
bUPDATE      = 17;
bPACK        = 18;
bCASEJUMP    = 19;
bSPLIT       = 20;
bADD         = 21;
bSUB         = 22;
bMULT        = 23;
bDIV         = 24;
bLE          = 25;
bGE          = 26;
bLT          = 27;
bGT          = 28;
bEQ          = 29;
bSTATICJAVACALL= 30;
bGETRUNTIME  = 31;

bFA          = 250;
bB4          = 180;
```

Code für Fab4

Auch in der C++-Implementierung war die entscheidene Information während der Codegenerierung, wo sich auf den Stack die einzelnen Variablen befinden. Hierzu gab es eine Umgebung, die Variablennamen auf einen Stackoffset abgebildet hat. Die Umgebung war als Map über eine Liste von Paaren implementiert. Auch in diesem Compiler werden wir entsprechend vorgehen. Die dazu benötigte Funktion `lookup` haben wir bereits implementiert.

Die folgende kleine Funktion erlaubt es, aus einer Liste von Variablennamen eine Umgebung zu erzeugen:

fab4c.fab4
```
mkEnv i as  = case as of
  Nil -> (Nil);
  Cons a ys -> (Cons (Pair  a i) (mkEnv (i+1) ys));
```

Wie wir in der C++-Implementierung gesehen haben, muß die Umgebung häufig um eins erhöt werden. Hierfür sehen wir die folgende Funktion vor.

fab4c.fab4
```
addOneToSnd p = addNToSnd (1) p;

addNToSnd n p = case (p) of Pair e1 e2 ->(Pair e1 (e2+n));
```

Es kann losgehen. Statt eines Besuchers wie in der C++ Implementierung haben wir eine Funktion, die einen Baumknoten untersuhct und die Fallunterscheidung über einen großen Case-Ausdruck bewerkstelligt. Die Funktion zur Codegenerierung von Ausdrucksknoten erhält drei zusätzliche Argumente:

- eine Liste der globalen Funktionsnamen.
- eine Liste der Konstruktornamen.
- und die Umgebung, die Verzeichnet, wo auf dem Stack lokale Variablen angelegt sind.

fab4c.fab4

```
genExprCode functions constructors env expr = case expr of
```

Das Ergebnis wird eine Liste mit Befehlen sein. Diese Liste wird die Befehle in umgekehrter Reihenfolge generiert und erst zum Shcluß umgedreht.

Variablen Beginnen wir mit der Codeerzeugung für Variablenknoten. Wenn der Variablenname eine Funktion ist wird ein `PushGlobal` erzeugt, ansonsten wird in der Umgebung nachgeschaut, wo sich die Variable auf dem Stack befindet, und für diesen Offset ein `Push` generiert.

fab4c.fab4

```
  VarExpr v-> if (contains stringEq v functions)
                 (Cons (PushGlobal v) (Nil))
                 (Cons (Push (lookUp v env)) (Nil));
```

Zahlen- und Zeichenkonstanten Für die zwei Konstantenarten werden die entsprechenden Befehle `PushInt` bzw. `PushChar` generiert.

fab4c.fab4

```
  NumExpr n -> (Cons (PushInt n) (Nil));
  CharExpr c-> (Cons (PushChar c) (Nil));
```

Applikationsknoten Wie schon in der C++ Implementierung sind nacheinander für die beiden zu applizierenden Unterausdrücke Code zu erzeugen, der dafür sorgt, daß auf dem Stack eine Referenz auf das entsprechende Ergebnis gelegt wird. Daher ist nach dem Code für das Argument zur Erzeugung des zu applizierenden Ausdrucks die Umgebung um eins zu erhöhen. Schließlich wird noch der Befehl `MkAp` an die Ergebnisliste angefügt.

fab4c.fab4

```
  AppExpr e1 e2 ->
   (Cons (MkAp)
    (append
     (genExprCode functions constructors
                                 (listMap addOneToSnd env) e1)
     (genExprCode functions constructors env e2)));
```


Operatorausdrücke Operatorausdrücke funktionieren entsprechend analog, mit der Ausnahme, daß der `Eval` Befehl für beide Operanden einzufügen ist, und der entsprechende Befehl für den Operator.

fab4c.fab4

```
  OpExpr e1 op e2->
    (Cons (getOpInstruction op)
     (append
      (Cons (Eval) (genExprCode functions constructors
                                 (listMap addOneToSnd env) e1))
      (Cons (Eval)(genExprCode functions constructors env e2))));
```

Konstruktoraufrufe Auch Konstruktoraufrufe funktionieren ähnlich. Hier ist allerdings notwendig über die Liste der Argumente zu iterieren, was wir mit der Funktion `genConstrCallArgs` machen, die im Anschluß an die derzeitige Funktion definiert wird.

fab4c.fab4

```
     ConstructorCall c args->
     (Cons (Pack (getPosition 0 c constructors) (length args))
     (genConstrCallArgs functions constructors env (reverse args)));
```

Case Ausdrücke Case-Ausdrücke sind natürlich wieder die weitaus nervenaufreibendsten. Wir lassen zunächst Code für alle Case-Alternativen erzeugen. Anschließend sind die Sprünge korrkt zu berechnen. Zum einen natürlich die Liste im Befehl `CaseJump` aber auch die Prünge am Ende jeder Alternative.

Die Funktion `genCaseAltsCode` sorgt dabei für die Generierung korrekter Sprünge. Die Funktion `genAltCode` erzeugt noch den inkorrkten Sprungwert `-1`.

fab4c.fab4

```
  CaseExpr e alts
   ->(case (listMap (genAltCode functions constructors env) alts) of
       Cons cs css -> (case
            (reverse (genCaseAltsCode 0
              (length (concat (listMap snd (Cons cs css))))
              (Cons cs css))) of
         Cons pis iss ->
           append
            (concat (listMap snd (Cons pis iss)))
            (Cons (CaseJump (listMap fst (Cons pis iss)))
            (Cons (Eval)
            (genExprCode functions constructors env e) ))
         )
       );

genAltCode functions constructors env caseAlt= case (caseAlt) of
  CaseAlt n vars e ->
   (Pair (getPosition 0 n constructors)
      (Cons (Jump (0-1))
      (Cons (Slide (length vars))
```

```
       (append
        (genExprCode functions constructors
          (append (zip vars (from 0))
                  (listMap (addNToSnd (length vars)) env))
         e)
        (Cons (Split (length vars)) (Nil))
       ))));

genCaseAltsCode start total  alts = case (alts) of
  Nil -> (Nil);
  Cons na alts2 ->(case na of
    Pair n a -> (case a of
      Cons jump is ->(Cons (Pair (Pair n start)
                         (Cons (Jump ((total-start)-(length a))) is))
                (genCaseAltsCode (start+(length a)) total alts2)
               ))
        );
```

Wir sind noch ein paar Hilfsfunktionen schuldig. Das erzeugen des Codes für die Argumente eines Konstruktoraufrufs:

fab4c.fab4

```
genConstrCallArgs functions constructors env args = case (args) of
  Nil -> (Nil);
  Cons arg further -> (append
       (genConstrCallArgs
            functions constructors (listMap addOneToSnd env) further)
       (genExprCode functions constructors env arg)
     );
```

Die Tabelle aller Operatorausdrücke mit ihrem korrekten Befehl:

fab4c.fab4

```
opInstructions
 = (Cons (Pair ("+") (Add))
   (Cons (Pair ("-") (Sub))
   (Cons (Pair ("*") (Mult))
   (Cons (Pair ("/") (Div))
   (Cons (Pair ("<") (Lt))
   (Cons (Pair (">") (Gt))
   (Cons (Pair ("<=") (Le))
   (Cons (Pair (">=") (Ge))
   (Cons (Pair ("=") (Eq)) (Nil))))))))));

getOpInstruction op = lookUp op opInstructions;
```

Code für ganzes Modul Zur Erzeugung des Codes für ein ganzes Modul wird das Ergebnis des Parsers betrachtet. Es wird nur für ein erfolgreiches `ParsResult` mit leerer Liste übriggebliebener Token zur Codegenerierung verwendet. Ansonsten brechen wir mit einer Fehlermeldung ab.

fab4c.fab4

```
genCode w pa = case pa of
  Fail f -> (abort (append ("parse error: could not parse: ") f));
  ParsResult fab4 rest -> if (isEmpty rest)(genFab4Code w fab4)
     (abort (append ("parse error: could not parse:") rest));
```

Damit haben wir einen abstrakten Syntaxbaum für Fab4 vorliegen. Zunächst lassen wir uns aus diesem die Namen aller definierten Konstruktoren und aller definierten Funktionen geben:

fab4c.fab4

```
genFab4Code w fab4 = case fab4 of
  Fab4 constrs functions
   -> genFab4Code2 w (listMap getConstrName constrs)
             (listMap getFunctionName ( functions)) ( functions);

getConstrName c = case c of ConstrDef n i -> n;
getFunctionName f = case f of FunDef n v e -> n;
```

Für die rechte Seite der Funktion ist dann Code zu erzeugen. Der Code für Funktionen endet immer mit einem `Unwind`. Wenn die Funktion Argumente hat, so sind diese noch vom Stack zu räumen und auch ein `Update` einzufügen:

fab4c.fab4

```
genFunctionCode conNames funNames fun = case fun of
 FunDef n as e ->addFunEndCode (length as)
                   (genExprCode funNames conNames (mkEnv 0 as) e);

addFunEndCode n cs = if (n>0)
  (reverse (Cons (Unwind) (Cons (Pop n) (Cons (Update n) cs))))
  (reverse (Cons (Unwind) cs));
```

Eingebaute Funktionen Drei Funktionen sind in Fab4 fest eingebaut. Diese drei Funktionen beschäftigen sich alle mit Kommunikation mit der Außenwelt. Es folgt der Code dieser eingebauten Funktionen, der schließlich einfach der Funktionsliste zugefügt wird.

fab4c.fab4

```
buildInNames
 = (Cons ("staticJavaCall")
   (Cons ("javaCall")(Cons ("getRuntime")(Nil))));

buildInFunctions = (Cons staticJavaCallFun
                   (Cons javaCallFun
                   (Cons getRuntimeFun
                   ((Nil)))));

staticJavaCallFun = (Pair
    (FunDef ("staticJavaCall")
            (Cons ("x1") (Cons ("x2") (Nil)))
            (VarExpr ("x1")))
```

```
    staticJavaCallCode
   );
javaCallFun = (Pair
    (FunDef ("javaCall")
            (Cons ("x1") (Cons ("x2") (Cons ("x3") (Nil))))
            (VarExpr ("x1")))
    javaCallCode
   );
getRuntimeFun = (Pair
    (FunDef ("getRuntime") (Nil) (NumExpr 42))
    getRuntimeCode
   );

staticJavaCallCode = (Cons (Push 1) (Cons (Eval) (Cons (Push 1)
    (Cons (Eval) (Cons (StaticJavaCall) (Cons (Unwind) (Nil)))))));

javaCallCode = (Cons (Push 2) (Cons (Eval) (Cons (Push 2)
    (Cons (Eval) (Cons (Push 2) (Cons (Eval) (Cons (JavaCall)
    (Cons (Unwind) (Nil)))))))));

getRuntimeCode = (Cons (GetRuntime)  (Cons (Unwind) (Nil)));
```

Das I und O des Compilers

Wir sind jetzt in der Lage, eine Befehlsliste für ein Fab4-Programm zu generieren. Jetzt ist noch der Bytecode für diese Befehle zu schreiben. Dieses wird über Javamethoden implementiert.

Dazu haben wir zwei Javaklassen implementiert:

fab4c.fab4

```
getReaderMethod="name.panitz.gm.GmFileReader.getInstance";
getWriterMethod="name.panitz.gm.GmWriter.getInstance";
```

Lesen einer Textdatei Zum Lesen einer Textdatei wird eine Instanz des Readers erzeugt und aus diesem so oft ausgelesen, bis kein weiteres Zeichen mehr im Reader ist:

fab4c.fab4

```
getReader fileName= staticJavaCall 1 getReaderMethod fileName;

readFile fileName = fst (read (getReader fileName));

read reader = case (javaCall 0 ("hasNext") reader) of
  Pair b reader2 -> (case (b) of
    True ->  (case (javaCall 0 ("next") reader2) of
      Pair c reader3 -> (case (read reader3)of
        Pair cs reader4 ->(Pair (Cons c cs) reader4) ));
    False -> (Pair (Nil) reader2));
```


Fehlerbehandlung und Programmargumente Ein paar einfache Funktionen zum Tracen sollen zur Verfügung stehen. Ebenso müssen wir eine Möglichkeit realisieren, auf die Programmargumente zuzugreifen.

fab4c.fab4

```
getProgArgs = tail(javaCall 0 ("getProgArgs") getRuntime);
abort x = (staticJavaCall 1 ("name.panitz.gm.GmWriter.error") x);
trace x
 = if (staticJavaCall 1 ("name.panitz.gm.GmWriter.print") x) x x;
trace2 x y = staticJavaCall 1 ("name.panitz.gm.GmWriter.trace") x y;
```

Schreiben der Bytecodedatei Zum Schreiben der Bytecodedatei sehen wir Funktionen vor, die die unterschiedlichen Typen korrekt in den Bytecodestrom schreiben.

fab4c.fab4

```
getWriter fileName = (staticJavaCall 1 getWriterMethod fileName);
write w n = javaCall 1 ("write")  w n;
writeChars w n = javaCall 1 ("writeChars")  w n;
writeInt w n = javaCall 1 ("writeInt")  w n;
writeString w n=javaCall 1 ("writeChars") (writeInt w (length n)) n;
```

Schreiben des Bytecodes

Damit steht alles zur Verfügung um den Bytecode zu schreiben. Zunächst sehen wir eine Funktion zum Schreiben der Befehlsliste in die Bytecode-Datei vor:

fab4c.fab4

```
writeIs writer is = case (is) of
  Cons j js -> writeIs (writeI writer j) js;
  Nil -> writer;
```

Eine große Fallunterscheidung betrachtet jeden einzelnen Befehl und schreibt für diesen dann den entsprechenden Code:

fab4c.fab4

```
writeI w i = case (i) of
 End  -> write w bEND ;
 Jump n -> writeInt (write w bJUMP) n;
 Unwind -> write w bUNWIND;
 Eval ->write w bEVAL;
 Print -> write w bPRINT;
 PrintString -> write w bPRINTSTRING;
 Output s-> write w bOUTPUT;
 MkAp -> write w bMKAP;
 Add -> write w bADD;
 Sub -> write w bSUB;
 Mult-> write w bMULT;
 Div -> write w bDIV;
 Le -> write w bLE;
```

```
Ge -> write w bGE;
Lt -> write w bLT;
Gt -> write w bGT;
Eq -> write w bEQ;
PushGlobal n-> writeString (write w bPUSHGLOBAL) n;
Push n-> writeInt (write w bPUSH) n;
Pop n-> writeInt (write w bPOP) n;
PushInt n-> writeInt (write w bPUSHINT)n;
PushChar n-> write (write w bPUSHCHAR)n;
Slide n-> writeInt (write w bSLIDE) n;
Update n-> writeInt (write w bUPDATE) n;
Pack n1 n2 -> writeInt (writeInt (write w bPACK) n1) n2;
CaseJump n
 -> writeCaseJumps (writeInt (write w bCASEJUMP) (length n)) n;
Split n-> writeInt (write w bSPLIT) n;
JavaCall -> write w bJAVACALL;
StaticJavaCall -> write w bSTATICJAVACALL;
GetRuntime -> write w bGETRUNTIME;
```

Die Paare von Konstruktornummern und Sprungweiten für einen `CaseJump`-Befehl schreibt folgende Funktion in den Bytecode:

fab4c.fab4

```
writeCaseJumps w js = case (js) of
  Nil -> w;
  Cons i is -> (case (i) of
    Pair co adr ->  writeCaseJumps
                        (writeInt
                        (writeInt w
                         co)
                         adr)
                         is );
```

Zum Schreiben eines ganzen Moduls sind entsprechend der Spezifikation die Bytemarkeierer am Anfang und Ende `FAB4` zu schreiben, eine Konstruktortabelle, eine Funktionstabelle und natürlich der Code:

fab4c.fab4

```
writeFab4Code w conNames functions is =
         write
         (write
         (writeCode
         (writeConstructors
         (writeFunctionTable
         (writeInt
         (writeString
         (write
         (write w
          bFA)
          bB4)
```



```
          ("Fab4 Programe"))
          (((length ( functions))+(length buildInFunctions))))
          0 is)
          conNames)
          is)
          bFA)
          bB4;
```

Schreiben der Konstruktortabelle

fab4c.fab4

```
writeConstructors w cs
  = writeConstructors2 (writeInt w (length cs)) cs;

writeConstructors2 w cs = case cs of
  Nil -> w;
  Cons c cs2 -> writeConstructors2 (writeString w c) cs2;
```

Schreiben der Funktionstabelle

fab4c.fab4

```
writeFunctionTable w n funs = case funs of
  Nil -> w;
  Cons f funs2 -> (case f of
    Pair fDef is -> (case fDef of
      FunDef name args code -> writeFunctionTable
                                (writeInt
                                (writeInt
                                (writeString w name)
                                 n)
                                 (length args))
                                 (n+(length is))
                                 funs2));
```

Schreiben des Codes

fab4c.fab4

```
writeCode w funs = case (concat (listMap snd funs)) of
  Cons c ode -> writeIs (writeInt w (1+(length ode))) (Cons c ode);
```

Aufruffunktion für die Codegenerierung

fab4c.fab4

```
genFab4Code2 w conNames funNames functions
 = writeFab4Code w conNames functions
      (append buildInFunctions
         (zip (functions)
           (listMap (genFunctionCode conNames
                                (append (buildInNames) funNames))
                    (functions))));
```

Die Mainfunktion

Alles ist beieinander. Die Mainfunktion holt sich den Modulnamen aus den Programmargumenten, liest die entsprechende Fab4 Quelltextdatei, wirft den Parser an und generiert den Code.

fab4c.fab4

```
main = case getProgArgs of
  Cons arg args1 -> genCode (getWriter (append  arg (".gmc")))
                    (fab4P (readFile (append  arg (".fab4"))))
```

DAmit haben wir jetzt den Bootstrapcompiler definiert. Wir haben tatsächlich einen Compiler, der mit sich selbst übersetzt werden kann. Den ursprünglich in C++ entwickelten Compiler können wir noch genau einmal benutzen, um den in Fab4 geschriebenen Compiler zu übersetzen. Von da an können wir nun vollständig in unserer eigenen Sprache und mit unserem eigenen Compiler arbeiten.

Folgendes Auszug einer Shellsitzung zeigt den Bootstrap:

- Übersetzen des Fab4-Compilers mit dem in C++ geschriebenen Compilers:

```
sep@pc216-5:~/fh/compiler/tutor> bin/fab4c src/fab4c
[]
[PUSH 1,EVAL,PUSH 1,EVAL,STATICJAVACALL,UNWIND,PUSH 2,EVAL,PUSH 2,EVAL,PUSH
2,EVAL,JAVACALL,UNWIND,GETRUNTIME,UNWIND,PUSH 0,EVAL,CASEJUMP [],SPLIT 0,PUSH
1,SLIDE 0,JUMP 4,SPLIT 0,PUSH 2,SLIDE 0,JUMP 0,UPDATE 3,POP 3,UNWIND,PUSH
0,EVAL,CASEJUMP [],SPLIT 0,PACK 1 0,SLIDE 0,JUMP 4,SPLIT 0,PUSH 1,SLIDE 0,JUMP
0,UPDATE 2,POP 2,UNWIND,PUSH 0,EVAL,CASEJUMP [],SPLIT 0,PACK 0 0,SLIDE 0,JUMP
4,SPLIT 0,PUSH 1,SLIDE 0,JUMP 0,UPDATE 2,POP 2,UN.....
.....BAL getProgArgs,EVAL,CASEJUMP [],SPLIT 2,PACK 2 0,PUSHCHAR 4,PACK 3
2,PUSHCHAR b,PACK 3 2,PUSHCHAR a,PACK 3 2,PUSHCHAR f,PACK 3 2,PUSHCHAR .,PACK
3 2,PUSH 1,PUSHGLOBAL append,MKAP,MKAP,PUSHGLOBAL readFile,MKAP,PUSHGLOBAL
fab4P,MKAP,PACK 2 0,PUSHCHAR c,PACK 3 2,PUSHCHAR m,PACK 3 2,PUSHCHAR g,PACK 3
2,PUSHCHAR .,PACK 3 2,PUSH 2,PUSHGLOBAL append,MKAP,MKAP,PUSHGLOBAL
getWriter,MKAP,PUSHGLOBAL genCode,MKAP,MKAP,SLIDE 2,JUMP 0,UNWIND]
```

- Kompilieren des `fab4c` mit sich selbst:

```
sep@pc216-5:~/fh/compiler/tutor> fab4 src/fab4c src/fab4c
gc:  old heap: 909 new heap: 14547 old used: 898 new used: 373
gc:  old heap: 14547 new heap: 11867 old used: 14536 new used: 2194
gc:  old heap: 11867 new heap: 20972 old used: 11856 new used: 3444
gc:  old heap: 20972 new heap: 27222 old used: 20961 new used: 5675
gc:  old heap: 27222 new heap: 38377 old used: 27211 new used: 8443
gc:  old heap: 38377 new heap: 52217 old used: 38366 new used: 12274
gc:  old heap: 52217 new heap: 71372 old used: 52206 new used: 17390
gc:  old heap: 71372 new heap: 96952 old used: 71361 new used: 24319
gc:  old heap: 96952 new heap: 131601 old used: 96939 new used: 33618
gc:  old heap: 131601 new heap: 178092 old used: 131590 new used: 46181
gc:  old heap: 178092 new heap: 240907 old used: 178081 new used: 63093
    .
    .
    .
gc:  old heap: 368982 new heap: 368372 old used: 368971 new used: 74024
gc:  old heap: 368372 new heap: 380122 old used: 368361 new used: 74192
gc:  old heap: 380122 new heap: 380962 old used: 380111 new used: 72694
```



```
gc:  old heap: 380962 new heap: 373472 old used: 380951 new used: 248439
gc:  old heap: 373472 new heap: 1252197 old used: 373461 new used: 243892
name.panitz.gm.GmWriter@10b4199
sep@pc216-5:~/fh/compiler/tutor>
```

Kapitel 8

Schlußkapitel

And in the end
The love you take
is equal to the love you make
Lennon/McCartney

Der Konzeption dieser Vorlesung lagen zwei hauptsächliche Entwurfsentscheidungen zugrunde:

- es soll ein Bootstrap in der implementierten Sprache möglich sein.
- möglichst viele Programmiertechniken (Besuchsmuster, Operatorüberladung, Funktionen höherer Ordnung...) sollten zum Einsatz kommen.

Der erste Punkt war nur mit einigen Klimmzügen zu erreichen und mir ist kein vergleichbarer Kurs bekannt, der dieses bewerkstelligt. Hierzu war zunächst die Wahl, eine *lazy* ausgewertete Programmiersprache zu implementieren, entscheidend. Solche Sprachen sind aufgrund des *Currying* und die sehr abstrakte Behandlung von Datentypen über Konstruktoren und Caseausdrücken in der Lage, mit minimalen syntaktischen Konstrukten mächtige Programme auszudrücken. Insbesondere die Behandlung der Strings als einfach verkettete Lsten hat es ermöglicht Dateien einzulesen und relativ einfach zu parsen.

Die Umsetzung der abstrakten Maschine hat hoffentlich ein Paar erhellende Lichter auf Fragen des Speichermanagements und der Semantik von Programmiersprachen geworfen, die auch hilfreich und interessant als Hintergrund sein können, wenn man in imperativen Programmiersprachen wie C codiert.

Leider lasen sich in einer 2-stündigen Vorlesungsreihe nicht mehr Fragen des Compilerbaus behandeln. Jetzt, wo es erst so richtig interessant werden würde müssen wir leider aufhören. Die nächste Schritte die in einer Nachfolgevorlesung behandelt werden könnten, sind:

- **syntaktische Erweiterungen**: Unsere Sprache ist bisher minimalst. Die nächsten wichtigen syntaktischen Erweiterungen, die unsere Sprache näher an die vergleichbaren Sprachen Haskell, Clean oder ML brächten, wären:
  - lokale Variablen und Funktionen mittels eines `let`-Konstruktes:

```
let f x = x+x in square (f 4)
```


  - anonyme Funktion mittels eines Lambda-Ausdrucks:

```
listMap (\x -> x+5) xs
```

  - Listliterale:

```
[1,4,76,34,432]
```

  - mehrstufiges Patternmatching als bequemere Variante der Caseausdrücke:

```
last (Cons x Nil) = x
last (Cons x xs) = last xs
```

  - Mengenausdrücke für Listen:

```
[x*x| x<-[1,2,..],(mod x 2)=1]
```

    (berechnet die Liste aller Quadratzahlen ungerader Zahlen)

- Fehlerbehandlung, Fehlerausgabe mit Zeilennummern. Dieses ist aufgrund der Generizität unserer Implementierung relativ einfach zu erreichen. Statt einen Parser über einfache Token zu implementieren, agiert der Parser auf eine Liste von Paaren, aus Positionsangabe und eigentlichen Token.
- Implementierung einer Typinferenz: Sichere das interessanteste anstehende Thema.
- Optimierung der Laufzeitmaschine. Der erste Schritt hierzu ist bereits gemacht durch die Umsetzung der G-Maschine in C, wie sie im Anhang zu finden ist. Aber auch in dieser Maschine gibt es viel Raum zum Optimieren. Hierzu ist auch in der Codegenerierung entsprechend zu verbessern:
  - optimierte Behandlung von Primitiven. Wenn es geht sollten Zwischenergebnisse arithmetischer Ausdrücke nicht im Heap gelegt werden, sondern direkt mit den Werten im Stack gerechenet werden
  - Tail Call Optimierung: der Aufruf einer rekursiven Funktionen braucht nicht unbedingt einen neuen leeren Stack, d.h. er braucht nicht in bestimmten Situationen den Stack im Dump zu zwischenzulagern.
  - Optimierteres Maschinenmodell: Verzicht auf die Ketten von Applikationsknoten bei mehrstelligen Funktionen. Ein optimiertes G-Maschinenmodell ist die sogenannte stg-Maschine.
  - Benutzung von `C--` als Zielsprache. `C--` ist speziell dafür konzipiert als Sprache zur Implementierung eines Backends für einen Compiler zu implementieren. Eine optimale Abbildung der abstrakten Maschine auf die konkrete Maschine kann über den `C--` Compiler erreicht werden.
- Einführung eines Modulsystems.

Es gäbe also viel zu tun, unseren kleinen Compiler zu einen mächtigen Compiler zu erweitern.

Ich hoffe die Vorlesung hat soweit Spaß gemacht und einige erhellende Erkenntnisse nicht nur über Compilerbau gebracht.

Anhang A

G-Maschinen-Log der Fakultät von 3

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 28
	1	(NGlobal 1 4)		PUSHGLOBAL main

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 29
0	1	(NGlobal 1 4)		EVAL

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 0
	1	(NGlobal 1 4)	pc=30	PUSHINT 3

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 1
2	1	(NGlobal 1 4)	pc=30	PUSHGLOBAL f
	2	3		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 2
1	1	(NGlobal 1 4)	pc=30	MKAP
2	2	3		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 3
	1	(NGlobal 1 4)	pc=30	UNWIND
3	2	3		
	3	NAp(1 2)		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 4
2	1	(NGlobal 1 4)	pc=30	PUSHINT 0
3	2	3		
	3	NAp(1 2)		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 5
4	1	(NGlobal 1 4)	pc=30	PUSH 1
2	2	3		
3	3	NAp(1 2)		
	4	0		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 6
2	1	(NGlobal 1 4)	pc=30	EVAL
4	2	3		
2	3	NAp(1 2)		
3	4	0		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 7
2	1	(NGlobal 1 4)	pc=30	Eq
4	2	3		
2	3	NAp(1 2)		
3	4	0		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 8
	1	(NGlobal 1 4)	pc=30	CASEJUMP {1=4, 0=0}
5	2	3		
2	3	NAp(1 2)		
3	4	0		
	5	(Constr 1 [])		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 13
	1	(NGlobal 1 4)	pc=30	SPLIT 0
5	2	3		
2	3	NAp(1 2)		
3	4	0		
	5	(Constr 1 [])		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 14
	1	(NGlobal 1 4)	pc=30	PUSHINT 1
2	2	3		
3	3	NAp(1 2)		
	4	0		
	5	(Constr 1 [])		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 15
	1	(NGlobal 1 4)	pc=30	PUSH 1
6	2	3		
2	3	NAp(1 2)		
3	4	0		
	5	(Constr 1 [])		
	6	1		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 16
	1	(NGlobal 1 4)	pc=30	EVAL
2	2	3		
6	3	NAp(1 2)		
2	4	0		
3	5	(Constr 1 [])		
	6	1		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 17
	1	(NGlobal 1 4)	pc=30	SUB
2	2	3		
6	3	NAp(1 2)		
2	4	0		
3	5	(Constr 1 [])		
	6	1		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 18
	1	(NGlobal 1 4)	pc=30	PUSHGLOBAL f
7	2	3		
2	3	NAp(1 2)		
3	4	0		
	5	(Constr 1 [])		
	6	1		
	7	2		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 19
	1	(NGlobal 1 4)	pc=30	MKAP
1	2	3		
7	3	NAp(1 2)		
2	4	0		
3	5	(Constr 1 [])		
	6	1		
	7	2		

Stack:	Heap:		Dump:	
	0	(NGlobal 0 0)		pc = 20
	1	(NGlobal 1 4)	pc=30	EVAL
	2	3		
8	3	NAp(1 2)		
2	4	0		
3	5	(Constr 1 [])		
	6	1		
	7	2		
	8	NAp(1 7)		

Stack:	Heap:		Dump:		
	0	(NGlobal 0 0)			pc = 4
	1	(NGlobal 1 4)	3	pc=21	PUSHINT 0
	2	3	2		
7	3	NAp(1 2)		pc=30	
8	4	0			
	5	(Constr 1 [])			
	6	1			
	7	2			
	8	NAp(1 7)			

Stack:	Heap:		Dump:		
	0	(NGlobal 0 0)			pc = 5
	1	(NGlobal 1 4)	3	pc=21	PUSH 1
	2	3	2		
9	3	NAp(1 2)		pc=30	
7	4	0			
8	5	(Constr 1 [])			
	6	1			
	7	2			
	8	NAp(1 7)			
	9	0			

Stack:

7
9
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0

Dump:

3 2	pc=21
	pc=30

pc = 6
EVAL

Stack:

7
9
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0

Dump:

3 2	pc=21
	pc=30

pc = 7
Eq

Stack:

10
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])

Dump:

3 2	pc=21
	pc=30

pc = 8
CASEJUMP {1=4, 0=0}

Stack:

10
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])

Dump:

3 2	pc=21
	pc=30

pc = 13
SPLIT 0

Stack:

7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])

Dump:

3 2	pc=21
	pc=30

pc = 14
PUSHINT 1

Stack:

11
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1

Dump:

3 2	pc=21
	pc=30

pc = 15
PUSH 1

Stack:

7
11
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1

Dump:

3 2	pc=21
	pc=30

pc = 16
EVAL

Stack:

7
11
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1

Dump:

3 2	pc=21
	pc=30

pc = 17
SUB

Stack:

12
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1

Dump:

3 2	pc=21
	pc=30

pc = 18
PUSHGLOBAL f

Stack:

1
12
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1

Dump:

3 2	pc=21
	pc=30

pc = 19
MKAP

Stack:

13
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)

Dump:

3 2	pc=21
	pc=30

pc = 20
EVAL

Stack:

12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)

Dump:

8 7	pc=21
3 2	pc=21
	pc=30

pc = 4
PUSHINT 0

Stack:

14
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0

Dump:

8 7	pc=21
3 2	pc=21
	pc=30

pc = 5
PUSH 1

Stack: 12, 14, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 6
`EVAL`

Stack: 12, 14, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 7
`Eq`

Stack: 15, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 8
`CASEJUMP {1=4, 0=0}`

Stack: 15, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 13
`SPLIT 0`

Stack: 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 14
`PUSHINT 1`

Stack: 16, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 15
`PUSH 1`

Stack: 12, 16, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 16
`EVAL`

Stack: 12, 16, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 17
`SUB`

Stack: 17, 12, 13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 18
`PUSHGLOBAL f`

Stack:

1
17
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0

Dump:

8 7	pc=21
3 2	pc=21
	pc=30

pc = 19
MKAP

Stack:

18
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)

Dump:

8 7	pc=21
3 2	pc=21
	pc=30

pc = 20
EVAL

Stack:

17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 4
PUSHINT 0

Stack:

19
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 5
PUSH 1

Stack:

17
19
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 6
EVAL

Stack:

17
19
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 7
Eq

Stack:

20
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 8
CASEJUMP {1=4, 0=0}

Stack:

20
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])

Dump:

13 12	pc=21
8 7	pc=21
3 2	pc=21
	pc=30

pc = 9
SPLIT 0

Stack:

17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 10
PUSHINT 1

Stack:

21
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])
21	1

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 11
SLIDE 0

Stack:

21
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])
21	1

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 12
JUMP 12

Stack:

21
17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NAp(1 17)
19	0
20	(Constr 0 [])
21	1

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 25
UPDATE 1

Stack:

17
18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 26
POP 1

Stack:

18

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1

Dump:

13, 12	pc=21
8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 27
UNWIND

Stack:

21
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 21
PUSH 1

Stack:

12
21
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 22
EVAL

Stack:

12
21
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 23
`MULT`

Stack:

22
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 24
`SLIDE 0`

Stack:

22
12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NAp(1 12)
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 25
`UPDATE 1`

Stack:

12
13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 26
`POP 1`

Stack:

13

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

8, 7	pc=21
3, 2	pc=21
	pc=30

pc = 27
`UNWIND`

Stack:

22
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

3, 2	pc=21
	pc=30

pc = 21
`PUSH 1`

Stack:

7
22
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

3 2	pc=21
	pc=30

pc = 22
EVAL

Stack:

7
22
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1

Dump:

3 2	pc=21
	pc=30

pc = 23
MULT

Stack:

23
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

3 2	pc=21
	pc=30

pc = 24
SLIDE 0

Stack:

23
7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NAp(1 7)
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

3 2	pc=21
	pc=30

pc = 25
UPDATE 1

Stack:

7
8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

3 2	pc=21
	pc=30

pc = 26
POP 1

Stack:

8

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

3 2	pc=21
	pc=30

pc = 27
UNWIND

Stack:

23
2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

pc=30

pc = 21
PUSH 1

Stack:

2
23
2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

pc=30

pc = 22
EVAL

Stack:

2
23
2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2

Dump:

pc=30

pc = 23
MULT

Stack:

24
2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc=30

pc = 24
SLIDE 0

Stack:

24
2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NAp(1 2)
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc=30

pc = 25
UPDATE 1

Stack:

2
3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NInd -> 24
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc=30

pc = 26
POP 1

Stack: 3

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NInd -> 24
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump: pc=30

pc = 27
`UNWIND`

Stack: 24

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NInd -> 24
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc = 30
`PRINT`

6

Stack:

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NInd -> 24
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc = 31
`OUTPUT`

Stack:

Heap:

0	(NGlobal 0 0)
1	(NGlobal 1 4)
2	3
3	NInd -> 24
4	0
5	(Constr 1 [])
6	1
7	2
8	NInd -> 23
9	0
10	(Constr 1 [])
11	1
12	1
13	NInd -> 22
14	0
15	(Constr 1 [])
16	1
17	0
18	NInd -> 21
19	0
20	(Constr 0 [])
21	1
22	1
23	2
24	6

Dump:

pc = 32
`END`

Anhang B

C Implementierung der G-Maschine

In diesem Abschnitt folgt vorerst unkommentiert eine Implementierung der G-Maschine in C. Die C-Version ist nicht in der Lage externe Aufrufe an javamethoden vorzunehmen, dafür aber kann sie mit externen C-Funktionen interagieren. Die C-Version ist nicht auf Klarheit sondern ein wenig mehr auf Effizienz ausgerichtet.Tatsächlich laufen in einfachen Tests Fab4-Programme mit der in C implementierten G-Maschine etwa 20 mal schneller als mit der in Java implementierten Maschine.

constants.h

```
#ifndef _CONSTANTS_H
#define _CONSTANTS_H

#define NOP          0
#define POP          1
#define END          2
#define JUMP         3
#define PUSH         4
#define UNWIND       5
#define EVAL         6
#define PRINT        7
#define PRINTSTRING  9
#define OUTPUT       10
#define MKAP         11
#define JAVACALL     12
#define PUSHGLOBAL   13
#define PUSHINT      14
#define PUSHCHAR     15
#define SLIDE        16
#define UPDATE       17
#define PACK         18
#define CASEJUMP     19
#define SPLIT        20
#define ADD          21
```



```
#define SUB          22
#define MULT         23
#define DIV          24
#define LE           25
#define GE           26
#define LT           27
#define GT           28
#define EQ           29
#define STATICJAVACALL 30
#define GETRUNTIME   31
#define CCALL        32

//Heapknoten
#define NUM 1
#define CHR 2
#define IND 3
#define APP 4
#define CON 5
#define GLB 6
#define FWD 7
#define IDN 8

//Konstruktoren
#define TRUE   0
#define FALSE  1
#define NIL    2
#define CONS   3
#define PAIR   4

#endif /* _CONSTANTS_H */

```

gm.h

```
#ifndef _GM_H
#define _GM_H

#define INITIALHeapSize 909
#define INITIALStackSize 100000

void eval();

int nglb(int argc,int code);
int nchr(char n);
int nnum(int n);
int ncon(int n,int argc,int argv []);

void initGM();

void whnf();
int pop();
```

```c

void check(int i);

int pc;
int* numberOfGlobals;
int* heap;

static int code[];
static char* constructors[] ;

 char** progArgsV;
 int progArgsC;

#endif /* _GM_H */

```

gm.c

```c
#include <stdio.h>
#include <string.h>
#include "gm.h"
#include "constants.h"

int nextHeap=0;
int heapSize=INITIALHeapSize;
int lastUsed=INITIALHeapSize;
int lastNeeded=5;

int* heap;
int* fromSpace;

int doTrace = 1;
void initGM(){
  heap =(int*)malloc(sizeof(int[INITIALHeapSize]));
}

int getAddress(int a){
  int data=heap[a];
  switch  (data){
    case IND: return getAddress(heap[a+1]);
    case IDN: return getAddress(heap[a+1]);
  }
  return a;
}

int nextStack=0;
const int maxStackSize=INITIALStackSize;
int stack [INITIALStackSize];
int stackBottom=0;

int stackSize(){
```



```c
  return nextStack-stackBottom;
}



int pop(){
  nextStack=nextStack-1;
  return stack[nextStack];
}
void popn(int n){
  nextStack=nextStack-n;
}

int peek(int n){
  return stack[nextStack-1-n];
}


void push(int n){
  stack[nextStack]=n;
  nextStack=nextStack+1;
}

void dump(){
  int sz=stackSize();
  push(pc);
  push(stackBottom);
  stackBottom=nextStack;

}

void restore(){
  nextStack=stackBottom;
  stackBottom=pop();
  pc=pop();
}


void showStack(){
    int i=0;//stackBottom;
 printf("stack bottom %i\n:",stackBottom);
  for (;i<nextStack;i++)
    printf("%i -> %i\n",i,stack[i]);
}


void showHeap(){
  printf("heapSize %i\n",nextHeap);
  printf("206851 %i %c\n",heap[206851],heap[206852]);
  int i=0;
```

```c
  for (;i<nextHeap;){
    printf("%i -> ",i);
    i=i+printNode(i);
    printf(" %i nextH %i",i,nextHeap);
    printf(" special %i %c",heap[206851],heap[206852]);
    fflush(stdout);
    printf("ok\n");
    fflush(stdout);
  }
    printf("heap gedruckt\n");
  fflush(stdout);
}

int printNode(int a){
  int nodeType = heap[a];
  int a1= heap[a+1];
  switch (nodeType){
    case IND: {printf("Ind %i",a1);return 2;}
    case IDN: {printf("Idn %i",a1);return heap[a+2];}
    case NUM: {printf("Num %i",a1);return 2;}
    case CHR: {printf("Chr %c",a1);return 2;}
    case CON:{
      printf("(");
      printf(constructors[a1]);

      int i=0;
      int argc = heap[a+2];
      for (;i<heap[a+2];i++){
        printf(" %i",getAddress(heap[a+3+i]));
      }
      printf(")");
      return argc+3;
    }
    case APP: {printf("App %i %i",a1,heap[a+2]);return 3;}
    case GLB: {printf("Glb %i %i",a1,heap[a+2]);return 3;}
  }
  printf("was ist das %i\n",nodeType);
  return 2;
}


void checkHeapBounds(int i,char* where){
  if (nextHeap+i>heapSize){
    fprintf(stderr,"Error: Heap size exceeded in %s\n",where);
    fflush(stderr);
    exit(1);
  }
}

int nnum(int n){
```



```c
    checkHeapBounds(2,"nnum");
  int result = nextHeap;
  heap[nextHeap]=NUM;
  heap[nextHeap+1]=n;
  nextHeap=nextHeap+2;
  return result;
}

int nchr(char n){
    checkHeapBounds(2,"nchr");
  int result = nextHeap;
  heap[nextHeap]=CHR;
  heap[nextHeap+1]=n;
  nextHeap=nextHeap+2;
  return result;
}

int nind(int n){
    checkHeapBounds(2,"nind");
  int result = nextHeap;
  heap[nextHeap]=IND;
  heap[nextHeap+1]=n;
  nextHeap=nextHeap+2;
  return result;
}

int nidn(int n,int l){
    checkHeapBounds(l,"nidn");
  int result = nextHeap;
  heap[nextHeap]=IDN;
  heap[nextHeap+1]=n;
  heap[nextHeap+2]=l;
  nextHeap=nextHeap+l;
  return result;
}

int napp(int n1,int n2){
    checkHeapBounds(3,"napp");
  int result = nextHeap;
  heap[nextHeap]=APP;
  heap[nextHeap+1]=n1;
  heap[nextHeap+2]=n2;
  nextHeap=nextHeap+3;
  return result;
}

int ncon(int n,int argc,int argv []){
  checkHeapBounds(argc+2,"ncon");
  int result = nextHeap;
  heap[nextHeap]=CON;
```

```
  heap[nextHeap+1]=n;
  heap[nextHeap+2]=argc;
  int i=0;
  for (;i<argc;i++){heap[nextHeap+3+i]=argv[i];}
  nextHeap=nextHeap+3+argc;
  return result;
}

int nglb(int argc,int code){
    checkHeapBounds(3,"nglb");
  int result = nextHeap;
  heap[nextHeap]=GLB;
  heap[nextHeap+1]=argc;
  heap[nextHeap+2]=code;
  nextHeap=nextHeap+3;
  return result;
}

void eval();
void unwind();

void check(int i);

void whnf(){
  int i = pc;
  pc=-1000;

  int a=getAddress(pop());
  dump();
  push(a);
  unwind();
  eval();
  pc=i;
}

void print(){
  int a = getAddress(pop());
  int nodeType = heap[a];
  switch (nodeType){
    case IND: {push(heap[a+1]);break;}
    case IDN: {push(heap[a+1]);break;}
    case NUM: {printf("%i",heap[a+1]);break;}
    case CHR: {printf("%c",heap[a+1]);break;}
    case CON:
    {
      printf("(");
      printf(constructors[heap[a+1]]);

      int i=0;
      for (;i<heap[a+2];i++){
```



```
        push(getAddress(heap[a+3+i]));
        printf(" ");
        whnf();
        print();
      }
      printf(")");
      break;
    }
    default: printf("kann man nicht drucken %i\n",nodeType);
  }
}

void unwind(){
  int a = getAddress(pop());
  int nodeType = heap[a];
  switch (nodeType){
    case IND:{push(heap[a+1]);unwind();break;}
    case IDN:{push(heap[a+1]);unwind();break;}
    case APP:{
      push(a);
      push(heap[a+1]);
      unwind();
      break;
    }
    case GLB:{
      int argc = heap[a+1];
      int cp = heap[a+2];
      int sSize=stackSize()+1;
      if (sSize<=argc) {
        fprintf(stderr,"args stimmen nicht %i %i für %i\n",sSize,argc,cp);
        exit(1);}
      else if (sSize==argc+1){
        int args[argc];
        int tmp;
        int i=argc-1;
        for (;i>=0;i--){
          tmp=pop();
          args[i]=heap[tmp+2];
        }
        if (argc>0)push(tmp);
        for (i=0;i<argc;i++)
          push(args[i]);
        pc=cp;

      }else{
        restore();
        push(a);
        unwind();
      }
      break;
```

```
    }
    default:{
      if (stackBottom>0) {
        restore();
        push(a);
        pc=pc+1;
      }
    }
  }
}

int add(int x, int y){return x+y;}
int sub(int x, int y){return x-y;}
int mult(int x, int y){return x*y;}
int div(int x, int y){return x/y;}
int le(int x, int y){return x<=y;}
int ge(int x, int y){return x>=y;}
int lt(int x, int y){return x<y;}
int gt(int x, int y){return x>y;}
int eq(int x, int y){return x==y;}

void arithOp(int (*f)(int,int)){
  check(2);
  int x1=getAddress(pop());
  int x2=getAddress(pop());
  push(nnum(f(heap[x1+1],heap[x2+1])));
  pc=pc+1;
}

void compOp(int (*f)(int,int)){
  check(3);
  int x1=getAddress(pop());
  int x2=getAddress(pop());
  int argv[0];
  int name=f(heap[x1+1],heap[x2+1])?0:1;
  push(ncon(name,0,argv));
  pc=pc+1;
}

void step(){
  switch (code[pc]){
    case PUSHINT:    {
      check(2);
      push(nnum(code[pc+1]));
      pc=pc+2;
      break;
    }
    case PUSHCHAR:    {
      check(2);
      push(nchr(code[pc+1]));
```



```
      pc=pc+2;
      break;
    }
    case PUSH:    {
      push(peek(code[pc+1]));
      pc=pc+2;
      break;
    }
    case POP:    {
      popn(code[pc+1]);
      pc=pc+2;
      break;
    }
    case SLIDE:    {
      int top=pop();
      popn(code[pc+1]);
      push(top);
      pc=pc+2;
      break;
    }
    case PACK:    {
      int name=code[pc+1];
      int argc=code[pc+2];
      check(3+argc);
      int argv [argc];
      int i=0;
      for (;i<argc;i++){
        argv[i]=pop();
      }
      int top=ncon(name,argc,argv);
      push(top);
      pc=pc+3;
      break;
    }
    case ADD:  {arithOp(add);break;}
    case SUB:  {arithOp(sub);break;}
    case MULT: {arithOp(mult);break;}
    case DIV:  {arithOp(div);break;}
    case LT:   {compOp(lt);break;}
    case GT:   {compOp(gt);break;}
    case LE:   {compOp(le);break;}
    case GE:   {compOp(ge);break;}
    case EQ:   {compOp(eq);break;}
    case PUSHGLOBAL:{push(code[pc+1]);pc=pc+2;break;}
    case MKAP:    {
      check(3);
      int a0=getAddress(pop());
      int a1=getAddress(pop());
      push(napp(a0,a1));
      pc=pc+1;
```

```
      break;
    }
    case UNWIND: {unwind();break;}
    case EVAL:    {
      int a=getAddress(pop());
      dump();
      push(a);
      unwind();
      break;
    }
    case UPDATE:    {
      int a = getAddress(pop());
      int n = peek(code[pc+1]);

      int size=getSize(n);
      heap[n]=(size==2)?IND:IDN;
      heap[n+1]=a;

      if (size>2) heap[n+2]=size;

      pc=pc+2;
      break;
    }
    case JUMP:    {pc=pc+code[pc+1];break;}
    case CASEJUMP:{
      int name = heap[getAddress(peek(0))+1];
      int jumpc = code[pc+1];
      int i=0;
      int found=1;

      for (;i<jumpc;i++){
        if (code[pc+2+2*i]==name){
          pc=pc+code[pc+2+2*i+1];
          found=0;
          break;
        }
      }
      if (found!=0){
        fprintf(stderr,"unmatched pattern %i",name);
        fprintf(stderr,constructors[name]);printf("\n");
        exit(1);
      }
      break;
    }
    case SPLIT:    {
      int n = code[pc+1];
      int a = getAddress(pop());
      int i=n;

      for (;i>0;i--)
```



```
       push(heap[a+2+i]);
      pc=pc+2;
      break;
    }
    case PRINT:    {print();pc=pc+1;break;}
    case CCALL:    {
      void(*f)();
      f=(void(*)())code[pc+1];
      f();
      pc=pc+2;
      break;
    }
    case END:    {pc=-1;break;}
    case NOP:    {pc=pc+1;break;}
    case GETRUNTIME:    {
      pc=pc+1;
      int args[]={};
      int args2[]={0,0};
      int a= ncon(NIL,0,args);
      push(a);
      int i=progArgsC-1;
      for (;i>0;i--){
        args2[0]=pop(pushCString(progArgsV[i]));
        args2[1]=pop();
        a=ncon(CONS,2,args2);
        push(a);
      }
      break;
    }
  }
}

void eval(){while (pc>=0){step();}}


/*---------garbage collection---------------------------------------*/
void copyGlobals();
void copyStack();
void copyHeap();
int copy(int adr);
void follow(int adr);

int* toSpace;

int from;
int to;

void gc(int needs){
  fprintf(stderr,"gc: old heap %i, cells %i, ",nextHeap,needs);

```

```
  fromSpace = heap;
  to=nextHeap;

  heapSize=lastUsed>heapSize/2?5*lastUsed+4*needs:heapSize+4*needs;
  fprintf(stderr,"new size %i, ",heapSize);
  fflush(stderr);

  toSpace=(int*)malloc(sizeof(int[heapSize]));
  heap=toSpace;

  fprintf(stderr," A");
  fflush(stderr);

  nextHeap=0;
  copyGlobals();
  fprintf(stderr,"G");

  fflush(stderr);
  copyStack();
  fprintf(stderr,"S");
  fflush(stderr);
  from=0;

  copyHeap();
  fprintf(stderr,"H ");
  fflush(stderr);

  free(fromSpace);

  fprintf(stderr,"new heap %i\n",nextHeap);
  fflush(stderr);

  lastUsed=nextHeap;
  lastNeeded=needs;
}

void copyGlobals(){
  int i=0;
  for(;i<(*numberOfGlobals);i++){copy(i*3);}
}

void copyStack(){
  int currentBottom=stackBottom;
  int currentNext=nextStack;
  int i=currentBottom;
  while(currentBottom>=0){
    for(;i<currentNext;i++){
      stack[i]=copy(stack[i]);
    }

```



```
    if (currentBottom==0)break;

    currentNext=currentBottom-2;
    currentBottom=stack[currentBottom-1];

    i=currentBottom;
  }
}


void copyHeap(){
  to=nextHeap;
  while (from<to){
    while (from<to){
      follow(from);
    }
    from=to;
    to=nextHeap;
  }
}

int copy(int adr){
  int n=(fromSpace)[adr];
  int result;
  switch (n) {
    case FWD:{return (fromSpace)[adr+1];}
    case IND:{result=nind((fromSpace)[adr+1]);break;}
    case IDN:{result=nind((fromSpace)[adr+1]);break;}
    case NUM:{result=nnum((fromSpace)[adr+1]);break;}
    case CHR:{result=nchr((fromSpace)[adr+1]);break;}
    case APP:{result=napp((fromSpace)[adr+1],(fromSpace)[adr+2]);break;}
    case CON:{
      int name=(fromSpace)[adr+1];
      int argc=(fromSpace)[adr+2];
      int argv[argc];
      int i=0;
      for (;i<argc;i++){argv[i]=(fromSpace)[adr+3+i];}
      result=ncon(name,argc,argv);
      break;
    }
    case GLB:{result= nglb((fromSpace)[adr+1],(fromSpace)[adr+2]);break;}
  }
  (fromSpace)[adr]=FWD;
  (fromSpace)[adr+1]=result;
  return result;
}

int getSize(int a){
  int n=heap[a];
  switch (n) {
```

```
    case APP:{return 3;}
    case GLB:{return 3;}
    case CON:{return 3+heap[a+2];}
    case IDN:{return heap[a+2];}
  }
  return 2;
}

void follow(int adr){
  int n=(toSpace)[adr];
  switch (n) {
    case IND:{(toSpace)[adr+1]=copy((toSpace)[adr+1])
             ;from=from+2;break;}
    case IDN:{(toSpace)[adr+1]=copy((toSpace)[adr+1])
             ;from=from+(toSpace[adr+2]);break;}
    case APP:{
      (toSpace)[adr+1]=copy((toSpace)[adr+1]);
      (toSpace)[adr+2]=copy((toSpace)[adr+2]);
      from=from+3;
      break;}
    case CON:{
      int argc=(toSpace)[adr+2];
      from=from+3+argc;
      for(;argc>0;argc--){
        (toSpace)[adr+2+argc]=copy((toSpace)[adr+2+argc]);
      }
      break;}
    case GLB:{from=from+3;break;}
    default:{from=from+2;break;}
  }
}

void check(int i){if (heapSize<nextHeap+i) gc(i);}

```

B.1 GM-Bytecode nach C konvertieren

WriteC.java

```
package name.panitz.gm;
import java.io.*;
import java.util.*;

import name.panitz.util.*;

public class WriteC {

  static public void writeC(GmState st,String fileName)
                                                   throws Exception{
    writeC(st,new FileWriter(fileName+".c"));
```



```
  }

  static public void writeC(GmState st,Writer out) throws Exception{
    Map<String,Integer> globs=new HashMap<String,Integer>();

    int pc=1;
    List<Pair<Integer,Instruction>> code
      = new ArrayList<Pair<Integer,Instruction>>();

    for (Instruction i:st.code){
      code.add(new Pair<Integer,Instruction>(pc,i));
      pc=pc+size(i);
    }

    out.write("#include \"constants.h\"\n");
    out.write("#include \"gm.h\"\n");
    out.write("#include <stdio.h>\n\n");


    out.write("char* constructors [] \n ={");
    boolean first = true;
    for (String c : st.constructors){
      if (first){first=false;}else {out.write("\n  ,");};
      out.write("\""+c+"\"");
    }

    out.write("\n  };\n\n");
    out.write("void init(){");

    GmGlobals globals = st.globals;
    int i =0;
    for (Map.Entry<String,Integer> glob:st.globals.entrySet()){
      out.write("\n  nglb("
      +((NGlobal)st.heap.get(glob.getValue())).argc+","
      +code.get(((NGlobal)st.heap.get(glob.getValue())).i).e1+");");
      globs.put(glob.getKey(),i);
      i=i+3;
    }
    out.write("\n  int i="+globs.size()+";");
    out.write("\n  numberOfGlobals=&i;");
    out.write("\n}\n\n");

    out.write("int code [] =\n  {NOP");
    int nr=0;
    for (Pair<Integer,Instruction> p:code){
      writeInstruction(p,code,globs,nr,out);
      nr=nr+1;
    }

    out.write("\n  };\n\n");
```

```
    out.write("\nint main()");
    out.write("\n{");
    out.write("\n    initGM();");
    out.write("\n    init();");
    out.write("\n    pc="+code.get(st.pc).e1+";");
    out.write("\n    eval();");
    out.write("\n    return (0);");
    out.write("\n}\n");

    out.flush();out.close();
  }

  static int size(Instruction i){
    if (i instanceof End)return 1;
    if (i instanceof Unwind)return 1;
    if (i instanceof Eval)return 1;
    if (i instanceof Print)return 1;
    if (i instanceof PrintString)return 1;
    if (i instanceof MkAp)return 1;
    if (i instanceof JavaCall)return 1;
    if (i instanceof StaticJavaCall)return 1;
    if (i instanceof GetRuntime)return 1;
    if (i instanceof Add)return 1;
    if (i instanceof Sub)return 1;
    if (i instanceof Mult)return 1;
    if (i instanceof Div)return 1;
    if (i instanceof Lt)return 1;
    if (i instanceof Gt)return 1;
    if (i instanceof Le)return 1;
    if (i instanceof Ge)return 1;
    if (i instanceof Eq)return 1;
    if (i instanceof Output)return 1;

    if (i instanceof Push)return 2;
    if (i instanceof Pop)return 2;
    if (i instanceof Jump)return 2;
    if (i instanceof PushInt)return 2;
    if (i instanceof Slide)return 2;
    if (i instanceof Update)return 2;
    if (i instanceof Split)return 2;
    if (i instanceof PushGlobal)return 2;
    if (i instanceof PushChar)return 2;
    if (i instanceof Pack)return 3;
    if (i instanceof CaseJump)
      return 2+((CaseJump)i).entrySet().size()*2;
    return 1;
  }

  static void writeInstruction
              (Pair<Integer,Instruction> pi
```



```
              ,List<Pair<Integer,Instruction>> is
              ,Map<String,Integer> globs
              ,int nr
              ,Writer out)throws Exception{
    out.write("\n  ,");
    Instruction i=pi.e2;
    if (i instanceof End){out.write("END");}
    else if (i instanceof Output){out.write("NOP");}
    else if (i instanceof Unwind){out.write("UNWIND");}
    else if (i instanceof Eval){out.write("EVAL");}
    else if (i instanceof Print){out.write("PRINT");}
    else if (i instanceof PrintString){out.write("PRINTSTRING");}
    else if (i instanceof MkAp){out.write("MKAP");}
    else if (i instanceof JavaCall){out.write("JAVACALL");}
    else if (i instanceof StaticJavaCall){
      out.write("STATICJAVACALL");}
    else if (i instanceof GetRuntime){out.write("GETRUNTIME");}
    else if (i instanceof Add){out.write("ADD");}
    else if (i instanceof Sub){out.write("SUB");}
    else if (i instanceof Mult){out.write("MULT");}
    else if (i instanceof Div){out.write("DIV");}
    else if (i instanceof Lt){out.write("LT");}
    else if (i instanceof Gt){out.write("GT");}
    else if (i instanceof Le){out.write("LE");}
    else if (i instanceof Ge){out.write("GE");}
    else if (i instanceof Eq){out.write("EQ");}

    else if (i instanceof Push){
      out.write("PUSH,");out.write(""+((Push)i).n);}
    else if (i instanceof Pop){
      out.write("POP,");out.write(""+((Pop)i).n);}
    else if (i instanceof Jump){//to do
      out.write("JUMP,");
      out.write(""+newJump(is,nr,((Jump)i).n));
    }
    else if (i instanceof PushInt){
      out.write("PUSHINT,");
      out.write(""+((PushInt)i).n);
    }
    else if (i instanceof Slide){
      out.write("SLIDE,");out.write(""+((Slide)i).n);}
    else if (i instanceof Update){
      out.write("UPDATE,");out.write(""+((Update)i).n);}
    else if (i instanceof Split){
      out.write("SPLIT,");out.write(""+((Split)i).argc);}
    else if (i instanceof PushGlobal){
      out.write("PUSHGLOBAL");
      out.write(","+globs.get(((PushGlobal)i).name));
    }
    else if (i instanceof PushChar){
```

```
      out.write("PUSHCHAR,");
      out.write("\'"+((PushChar)i).n+"\'");
    }

    else if (i instanceof Pack){
      out.write("PACK");
      out.write(","+((Pack)i).name);
      out.write(","+((Pack)i).argc);
    }
    else if (i instanceof CaseJump){
      out.write("CASEJUMP");
      out.write(","+((CaseJump)i).size());
      for (Map.Entry<Integer,Integer> jump:((CaseJump)i).entrySet()){
        out.write(","+jump.getKey());
        out.write(","+newJump(is,nr,jump.getValue()));
      }
    }
    out.write(" // "+pi.e1);
  }

  static int newJump
        (List<Pair<Integer,Instruction>> code,int from, int step){
    Integer fr = code.get(from).e1;
    Integer to = code.get(from+step+1).e1;
    return to-fr;
  }

  public static void main(String[] args)throws Exception{
    GmState st
     = ReadGm.readGMCode
         (new DataInputStream
             (new FileInputStream(args[0]+".gmc")));
    writeC(st,args[0]);
  }
}
```

B.2 Aufruf von C aus der G-Maschine

ccall.h

```
#ifndef _CCALLS_H
#define _CCALLS_H

typedef  char*   string;

string creadF(string fileName,string result);
string cwriteF(string fileName,string content,string result);
string cappendF(string fileName,string content,string result);

int getInt();
```



```
char getChr();
string getString(string result);

void pushCCar(char c);
void pushCInt(int  n);
void pushCString(string  str);

#endif /* _CCALLS_H */

```

ccall.c

```
#include <stdio.h>
#include "ccall.h"
#include "gm.h"
#include "constants.h"

struct CharList{char c; struct CharList* next;} ;

int getInt(){
  whnf();
  return heap[getAddress(pop())+1];
}

char getChr(){
  whnf();
  return (char)heap[getAddress(pop())+1];
}

string getString(string result){
  int i = 0;
  struct CharList* current
    = (struct CharList*)malloc(sizeof(struct CharList));
  struct CharList* newCurrent;

  whnf();//achtung gc!
  int a=getAddress(peek(0));

  while (heap[a+1]==CONS){
    i=i+1;
    push(heap[a+3]);
    whnf();
    char c=(char)heap[getAddress(pop())+1];
    a=getAddress(pop());
    (*current).c=c;
    newCurrent
      = (struct CharList*)malloc(sizeof(struct CharList));
    (*newCurrent).next=current;
    current=newCurrent;
    push(heap[a+4]);
    whnf();
```

```
    a=getAddress(peek(0));
  }
  pop();

  result = (string)malloc (sizeof(char[i+1]));

  result[i]='\0';

  i=i-1;
   current=(*current).next;
  for (;i>=0;i--){
    result[i]=(*current).c;
    struct CharList* oldCurrent=current;
    current=(*current).next;
    free(oldCurrent);
  }
  return result;
}

void pushCChr(char c){check(2);push(nchr(c));}
void pushCInt(int  n){check(2);push(nnum(n));}
void pushCString(string  str){
  int i=0;
  int args[]={};
  int args2[]={0,0};
  for (;!(str[i]=='\0');i++){}
  check(i*2+i*5+3);
  int a=ncon(NIL,0,args);
  i=i-1;
  for (;i>=0;i--){
    args2[0]=nchr(str[i]);
    args2[1]=a;
    a=ncon(CONS,2,args2);
  }
  push(a);
}

```

fileIO.c

```
#include "stdio.h"
#include "ccall.h"


string creadF(string fileName,string result){
  FILE *fp;
  int c;
  int size=0;
  fp = fopen(fileName, "r");

  c = getc(fp);
```



```
  while (c != EOF) {
    size = size + 1;
    c = getc(fp);
  }
  size=size;
  fclose(fp);
  result = (string)malloc(1+size * sizeof(char));
  fp = fopen(fileName, "r");
  int i=0;
  for (;i<size;i++){
    result[i]=(char)getc(fp);
  }
  result[size]='\0';
  fclose(fp);

  for (i=0;i<size-1;i++){}
  return result;
}

string cwriteF(string fileName,string content,string result){
  FILE *fp;
  fp = fopen(fileName, "w");
  int i=0;
  char c = content[i];
  while (c!='\0'){
    putc(c,fp);
    i=i+1;
    c=content[i];
  }
  fclose(fp);
  return fileName;
}

string cappendF(string fileName,string content,string result){
  FILE *fp;
  fp = fopen(fileName, "a");
  int i=0;
  char c = content[i];
  while (c!='\0'){
    putc(c,fp);
    i=i+1;
    c=content[i];
  }
  fclose(fp);
  return fileName;
}

```

B.3 Beispielcode: Primzahlen

Eratosthenes.c

```
#include "constants.h"
#include "gm.h"
#include <stdio.h>

char* constructors[]
 ={"True"
  ,"False"
  ,"Nil"
  ,"Cons"
  };

void init(){
  nglb(2,74);
  nglb(0,215);
  nglb(1,178);
  nglb(2,1);
  nglb(1,54);
  nglb(1,22);
  nglb(3,9);
  nglb(0,20);
  nglb(2,148);
  int i=9;
  numberOfGlobals=&i;
}


int code []=
  {NOP
  ,PUSH,1 // 1
  ,EVAL // 3
  ,PUSH,1 // 4
  ,EVAL // 6
  ,STATICJAVACALL // 7
  ,UNWIND // 8
  ,PUSH,2 // 9
  ,EVAL // 11
  ,PUSH,2 // 12
  ,EVAL // 14
  ,PUSH,2 // 15
  ,EVAL // 17
  ,JAVACALL // 18
  ,UNWIND // 19
  ,GETRUNTIME // 20
  ,UNWIND // 21
  ,PUSH,0 // 22
  ,EVAL // 24
  ,CASEJUMP,2,1,15,0,6 // 25
```



```
  ,SPLIT,0 // 31
  ,PACK,1,0 // 33
  ,SLIDE,0 // 36
  ,JUMP,11 // 38
  ,SPLIT,0 // 40
  ,PACK,0,0 // 42
  ,SLIDE,0 // 45
  ,JUMP,2 // 47
  ,UPDATE,1 // 49
  ,POP,1 // 51
  ,UNWIND // 53
  ,PUSHINT,1 // 54
  ,EVAL // 56
  ,PUSH,1 // 57
  ,EVAL // 59
  ,ADD // 60
  ,PUSHGLOBAL,12 // 61
  ,MKAP // 63
  ,PUSH,1 // 64
  ,PACK,3,2 // 66
  ,UPDATE,1 // 69
  ,POP,1 // 71
  ,UNWIND // 73
  ,PUSH,1 // 74
  ,EVAL // 76
  ,CASEJUMP,2,2,6,3,15 // 77
  ,SPLIT,0 // 83
  ,PACK,2,0 // 85
  ,SLIDE,0 // 88
  ,JUMP,53 // 90
  ,SPLIT,2 // 92
  ,PUSH,0 // 94
  ,PUSH,3 // 96
  ,MKAP // 98
  ,EVAL // 99
  ,CASEJUMP,2,1,6,0,20 // 100
  ,SPLIT,0 // 106
  ,PUSH,1 // 108
  ,PUSH,3 // 110
  ,PUSHGLOBAL,0 // 112
  ,MKAP // 114
  ,MKAP // 115
  ,SLIDE,0 // 116
  ,JUMP,21 // 118
  ,SPLIT,0 // 120
  ,PUSH,1 // 122
  ,PUSH,3 // 124
  ,PUSHGLOBAL,0 // 126
  ,MKAP // 128
  ,MKAP // 129
```

```
  ,PUSH,1 // 130
  ,PACK,3,2 // 132
  ,SLIDE,0 // 135
  ,JUMP,2 // 137
  ,SLIDE,2 // 139
  ,JUMP,2 // 141
  ,UPDATE,2 // 143
  ,POP,2 // 145
  ,UNWIND // 147
  ,PUSHINT,0 // 148
  ,EVAL // 150
  ,PUSH,1 // 151
  ,EVAL // 153
  ,PUSH,2 // 154
  ,EVAL // 156
  ,PUSH,4 // 157
  ,EVAL // 159
  ,DIV // 160
  ,EVAL // 161
  ,MULT // 162
  ,EVAL // 163
  ,PUSH,3 // 164
  ,EVAL // 166
  ,SUB // 167
  ,EVAL // 168
  ,EQ // 169
  ,PUSHGLOBAL,15 // 170
  ,MKAP // 172
  ,UPDATE,2 // 173
  ,POP,2 // 175
  ,UNWIND // 177
  ,PUSH,0 // 178
  ,EVAL // 180
  ,CASEJUMP,1,3,4 // 181
  ,SPLIT,2 // 185
  ,PUSH,1 // 187
  ,PUSH,1 // 189
  ,PUSHGLOBAL,24 // 191
  ,MKAP // 193
  ,PUSHGLOBAL,0 // 194
  ,MKAP // 196
  ,MKAP // 197
  ,PUSHGLOBAL,6 // 198
  ,MKAP // 200
  ,PUSH,1 // 201
  ,PACK,3,2 // 203
  ,SLIDE,2 // 206
  ,JUMP,2 // 208
  ,UPDATE,1 // 210
  ,POP,1 // 212
```



```
  ,UNWIND // 214
  ,PUSHINT,2 // 215
  ,PUSHGLOBAL,12 // 217
  ,MKAP // 219
  ,PUSHGLOBAL,6 // 220
  ,MKAP // 222
  ,UNWIND // 223
  ,PUSHGLOBAL,3 // 224
  ,EVAL // 226
  ,PRINT // 227
  ,NOP // 228
  ,END // 229
  };

int main(){
    initGM();
    init();
    pc=224;
    eval();
    return (0);
}
```

Anhang C

Implementierungen

C.1 Token für Fab4

Fab4Token.cpp

```
#include <string>
#include "../name/panitz/util/Show.h"
#include "Fab4Token.h"

using namespace std;
using namespace name::panitz::util;

namespace fab4 {

Token::Token(string name):name(name){};
 bool Token::eq(Token* t){
   return name==t->name;
}

bool Token::isCName(){return false;}
bool Token::isName(){return false;}
bool Token::isNumber(){return false;}
bool Token::isStringLiteral(){return false;}
bool Token::isCharLiteral(){return false;}
bool Token::isOperator(){return false;}

string Token::toString(){return name;}

CName::CName(string name):Token(name){};
 bool CName::eq(Token* t){
  return t->isCName();
}

 bool CName::isCName(){return true;}
 string CName::toString(){return name;}

```



```

Name::Name(string name):Token(name){};
 bool Name::eq(Token* t){
  return t->isName();
}

bool Name::isName(){return true;}
string Name::toString(){return name;}

Number::Number(int val):Token("Number"),val(val){};
bool Number::eq(Token* t){
  return t->isNumber();
}
bool Number::isNumber(){return true;}
string Number::toString(){return show(val);}

StringLiteral::StringLiteral(string str):Token(str){};
bool StringLiteral::isStringLiteral(){return true;}
bool StringLiteral::eq(Token* t){
  return t->isStringLiteral();
}

CharLiteral::CharLiteral(char c):Token("CharLiteral"){this->c=c;};
bool CharLiteral::isCharLiteral(){return true;}
bool CharLiteral::eq(Token* t){
  return t->isCharLiteral();
}

bool Operator::isOperator(){return true;}
string Operator::toString(){return op;}
Operator::Operator(string op):Token(op){this->op=op;}

bool tokenEq(Token* x,Token* y){return x->eq(y);}

vector<Token*> tokenize(char* fileName);
}

```

C.2 Datentyp für Fab4

Fab4.cpp

```
#include <vector>
#include <string>

#include "Fab4.h"
#include "../name/panitz/util/Show.h"

using namespace std;
using namespace name::panitz::util;
```

```
namespace fab4 {

Var::Var(string n):name(n){};
string Var::toString(){return name;}

Constr::Constr(string n,vector<Expr*>* a):name(n),args(a){};
string Constr::toString(){return name+" "+show(args);}

App::App(Expr* p1,Expr* p2):p1(p1),p2(p2){}
string App::toString(){return "("+show(p1)+" "+show(p2)+")";}

void Expr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void Var::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void Constr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void App::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void CaseExpr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void NumExpr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void CharExpr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}
void OpExpr::visit(ExprVisitor* vis){
  ((ExprVisitor*)vis)->eval(this);}

OpExpr::OpExpr(Expr* e1,Operator* op,Expr* e2):op(op){
    this->op=op;this->e1=e1;this->e2=e2;};
string OpExpr::toString(){return "("+show(e1)+show(op)+show(e2)+")";}

NumExpr::NumExpr(int n):n(n){};
string NumExpr::toString(){return show(n);}

CharExpr::CharExpr(char c):c(c){};
string CharExpr::toString(){return show(c);}

FunDef::FunDef(string name,vector<string> args,Expr* expr):
  name(name),args(args),expr(expr){}

FunDef::~FunDef(){
  delete expr;}

string FunDef::toString(){
  string result = name;
  for (int i=0;i<args.size();i=i+1)
    result = result+" "+args[i];
  result = result+" = ";
```



```
  result = result+show(expr);
  return result;
}

CaseAlt::CaseAlt(string constr,vector<string> vars,Expr* alt):
    constr(constr),vars(vars),alt(alt){};
string CaseAlt::toString(){
  return show(constr)+" "+show(vars)+" "+show(alt);}

CaseExpr::CaseExpr(Expr* e1,vector<CaseAlt*>* alts):
  e1(e1),alts(alts){};
string CaseExpr::toString(){
  return "case "+show(e1)+ " of "+show(alts);}

ConstrDef::ConstrDef(string name,int arity):name(name),arity(arity){}
std::string ConstrDef::toString(){
  return "constr "+name+" "+show(arity);
}

Fab4::Fab4(vector<ConstrDef*> cs,vector<FunDef*> fs,string n):
  constructors(cs),functions(fs),name(n){}

Fab4::~Fab4(){
  for (int i = 0;i<constructors.size();i=i+1)
    delete constructors[i];
  for (int i = 0;i<functions.size();i=i+1)
    delete functions[i];
}

std::string Fab4::toString(){
  string result="";
  for (int i = 0;i<constructors.size();i=i+1){
    if (i>0) result=result+";\n\n";
    result=result+show(constructors[i]);
  }
  for (int i = 0;i<functions.size();i=i+1){
    if (i>0||constructors.size()>0) result=result+";\n\n";
    result=result+show(functions[i]);
  }

  return result;
}
}
```

C.3 Funktionen zum Bauen des Fab4 Parsergebnisses

Fab4MkFunctions.cpp

```
#include "../name/panitz/parser/ParsLib.h"
#include "Fab4MkFunctions.h"

using namespace name::panitz::parser;
using namespace std;

namespace fab4 {
Parser<Token*,Token>* pTok(Token* t){
  return getToken(t,tokenEq);
}

Parser<Operator*,Token>* pOp(Token* t){
  return ((Parser<Operator*,Token>*)getToken(t,tokenEq));
}

Parser<Number*,Token>* pNum(Token* t){
  return ((Parser<Number*,Token>*)getToken(t,tokenEq));
}


Expr* mkNumExpr(Number* num){return new NumExpr(num->val);}
Expr* mkCharExpr(CharLiteral* cl){return new CharExpr(cl->c);}
Expr* mkStringExpr(StringLiteral* sl){
  vector<Expr*>* args=new vector<Expr*>();
  Expr* result=new Constr("Nil",args);
  for (int i=sl->name.size()-1;i>=0;i--){
      args=new vector<Expr*>();
    args->insert(args->end(),new CharExpr(sl->name[i]));
    args->insert(args->end(),result);
    result=new Constr("Cons",args);
  }
  return result;
}

Expr* mkVar(Token* t){return new Var(t->name);}

Expr* mkOpExpr(Pair<Expr*,vector<Pair<Operator*,Expr*>*>*>* p){
  Expr* result=p->fst;
  vector<Pair<Operator*,Expr*>*>* opEs = p->snd;
  for (int i=0;i<opEs->size();i++){
    Operator* op=(*opEs)[i]->fst;
    Expr* e2 = (*opEs)[i]->snd;
    result = new OpExpr(result,op,e2);
  }
  return result;
}

```



```
Expr* mkFunApps(vector<Expr*>* exprs){
  Expr* result = (*exprs)[0];
  for (int i=1;i<exprs->size();i++)
    result=new App(result,(*exprs)[i]);
  delete exprs;
  return result;
}

Expr* mkConstr(Pair<Token*,vector<Expr*>*>* p){
  return new Constr(p->fst->name,p->snd);}

CaseAlt* mkCaseAlt(caseAltResult caseAlt){
  string cname = show(caseAlt->fst->fst->fst);
  vector<Token*>* varToks = caseAlt->fst->fst->snd;
  Expr* expr = caseAlt->snd;

  vector<string> vars;
  for (int i=0;i<varToks->size();i++)
      vars.insert(vars.end(),show((*varToks)[i]));

  delete caseAlt->fst->fst;
  delete caseAlt->fst;
  delete caseAlt;
  CaseAlt* result = new CaseAlt(cname,vars,expr);
  return result;
}

Expr* mkCaseExpr(caseParsResult casePars){
  Expr* e = casePars->fst->fst->snd;
  vector<CaseAlt*>* alts = casePars->snd;
  delete casePars->fst->fst;
  delete casePars->fst;
  delete casePars;
  return new CaseExpr(e,alts);
}

FunDef* mkFunDef
     (Pair<Pair<Pair<Token*,vector<Token*>*>*,Operator*>*,Expr*>* p){
  Token* name = p->fst->fst->fst;
  vector<Token*>* varTs = p->fst->fst->snd;
  vector<string> vars;
  for (int i=0;i<varTs->size();i=i+1)
      vars.insert(vars.end(),((*varTs)[i])->name);

  Expr* expr = p->snd;

  delete varTs;
  delete p->fst->fst;
  delete p->fst;
  delete p;
```

```
  return new FunDef(name->name,vars,expr);
}

ConstrDef* mkConstr(Pair<Token*,Number*>* p){
 string name = p->fst->name;
 int num = p->snd->val;
 delete p;
 return new ConstrDef(name,num);}

Fab4* mkFab4(vector<ConstrOrFun*>* constrOrFuns){
  vector<ConstrDef*> constrDefs;
  vector<FunDef*> funDefs;

  for (int i=0;i< constrOrFuns->size();i++) {
    if ((*constrOrFuns)[i] ->isLeft())
      constrDefs.insert
       (constrDefs.end()
       ,((Left<ConstrDef*,FunDef*>*)((*constrOrFuns)[i]))->left);
    else
      funDefs.insert
       (funDefs.end()
       ,((Right<ConstrDef*,FunDef*>*)((*constrOrFuns)[i]))->right);
    delete (*constrOrFuns)[i];
  }
  delete constrOrFuns;
  return new Fab4(constrDefs,funDefs,"GM Code");
}
}

```

C.4 G-Maschinenbefehle

Instruction.cpp

```
#include <map>
#include <vector>
#include <string>
#include <fstream>

#include "../name/panitz/util/Show.h"
#include "Fab4Parser.h"
#include "Instruction.h"

using namespace std;
using namespace name::panitz::util;

namespace fab4 {

void writeInt(int i,ostream& s){
```



```
   byte b1 = i/256/256/256%256;
   byte b2 = i/256/256%256;
   byte b3 = i/256%256;
   byte b4 = (byte)(i%256);
   s<<b1;
   s<<b2;
   s<<b3;
   s<<b4;
}

void writeString(string str,ostream& s){
  writeInt(str.size(),s);
  char cs [str.size()];

  for (int i=0;i<str.size();i++){
      s<<'\0';
      s<<str[i];
  };
}

string End::toString(){return "END";}
void End::write(ostream& out){out<<END;}

Jump::Jump(int n):n(n){}
string Jump::toString(){return "JUMP "+show(n);}
void Jump::write(ostream& out){out<<JUMP;writeInt(n,out);}

string Unwind::toString(){return "UNWIND";}
void Unwind::write(ostream& out){out<<UNWIND;}

string Eval::toString(){return "EVAL";}
void Eval::write(ostream& out){out<<EVAL;}

string Print::toString(){return "PRINT";}
void Print::write(ostream& out){out<<PRINT;}

string PrintString::toString(){return "PRINTSTRING";}
void PrintString::write(ostream& out){out<<PRINTSTRING;}

Output::Output(string s){this->s=s;}
string Output::toString(){return "OUTPUT "+s;}
void Output::write(ostream& out){out<<OUTPUT;writeString(s,out);}

string MkAp::toString(){return "MKAP";}
void MkAp::write(ostream& out){out<<MKAP;}

string JavaCall::toString(){return "JAVACALL";}
void JavaCall::write(ostream& out){out<<JAVACALL;}

string GetRuntime::toString(){return "GETRUNTIME";}
```

```
void GetRuntime::write(ostream& out){out<<GETRUNTIME;}

string StaticJavaCall::toString(){return "STATICJAVACALL";}
void StaticJavaCall::write(ostream& out){out<<STATICJAVACALL;}

string Add::toString(){return "ADD";}
void Add::write(ostream& out){out<<ADD;}

string Sub::toString(){return "SUB";}
void Sub::write(ostream& out){out<<SUB;}

string Mult::toString(){return "MULT";}
void Mult::write(ostream& out){out<<MULT;}

string Div::toString(){return "DIV";}
void Div::write(ostream& out){out<<DIV;}

string Le::toString(){return "LE";}
void Le::write(ostream& out){out<<LE;}

string Ge::toString(){return "GE";}
void Ge::write(ostream& out){out<<GE;}

string Lt::toString(){return "LT";}
void Lt::write(ostream& out){out<<LT;}

string Gt::toString(){return "GT";}
void Gt::write(ostream& out){out<<GT;}

string Eq::toString(){return "EQ";}
void Eq::write(ostream& out){out<<EQ;}

PushGlobal::PushGlobal(string n){name=n;}
string PushGlobal::toString(){return "PUSHGLOBAL "+name;}
void PushGlobal::write(ostream& out){
  out<<PUSHGLOBAL;writeString(name,out);
}

Push::Push(int n){this->n=n;}
string Push::toString(){return "PUSH "+show(n);}
void Push::write(ostream& out){out<<PUSH;writeInt(n,out);}

Pop::Pop(int n){this->n=n;}
string Pop::toString(){return "POP "+show(n);}
void Pop::write(ostream& out){out<<POP;writeInt(n,out);}

PushInt::PushInt(int n){this->n=n;}
string PushInt::toString(){return "PUSHINT "+show(n);}
void PushInt::write(ostream& out){out<<PUSHINT;writeInt(n,out);}

```

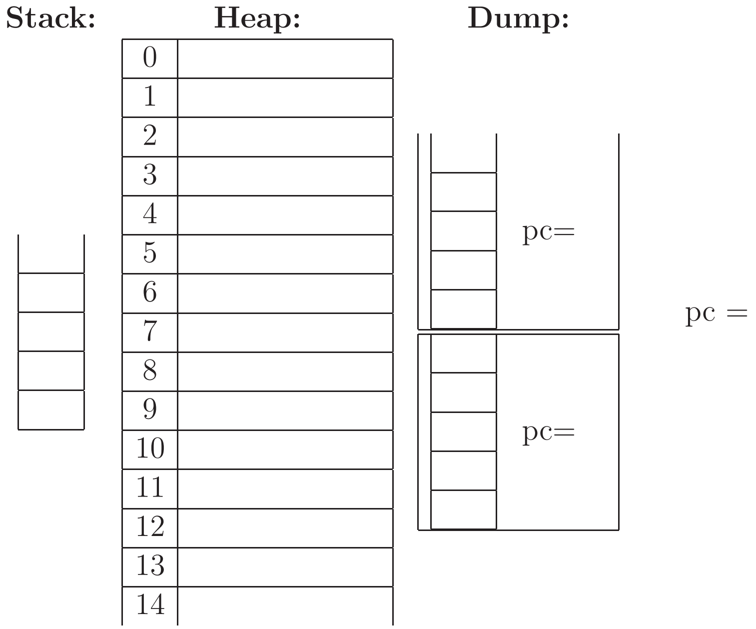


```
PushChar::PushChar(char n){this->n=n;}
string PushChar::toString(){return "PUSHCHAR "+show(n);}
void PushChar::write(ostream& out){out<<PUSHCHAR<<'\0'<<n;}

Slide::Slide(int n){this->n=n;}
string Slide::toString(){return "SLIDE "+show(n);}
void Slide::write(ostream& out){out<<SLIDE;writeInt(n,out);}

Update::Update(int n){this->n=n;}
string Update::toString(){return "UPDATE "+show(n);}
void Update::write(ostream& out){out<<UPDATE;writeInt(n,out);}

Pack::Pack(int n,int c){name=n;argc=c;}
string Pack::toString(){return "PACK "+show(name)+" "+show(argc);}
void Pack::write(ostream& out){
  out<<PACK;writeInt(name,out);writeInt(argc,out);}

struct ltstr
{
  bool operator()(int i1,int i2) const
  {
      return i1<i2;
  }
};

string CaseJump::toString(){return "CASEJUMP []"; }
template<class Iterator>
static void writeJumps(Iterator begin,Iterator end,ostream& out){
   for (Iterator it=begin;it!=end;it++){
     writeInt((*it).first,out);
     writeInt((*it).second,out);
   }
}
void CaseJump::write(ostream& out){
  out<<CASEJUMP;
  writeInt(this->size(),out);
  writeJumps(this->begin(),this->end(),out);}

Split::Split(int c){argc=c;}
string Split::toString(){return "SPLIT "+show(argc);}
void Split::write(ostream& out){out<<SPLIT;writeInt(argc,out);}

Instruction* getOp(Operator* opT){
  string op = opT->toString();
  if (op=="+")  return new Add();
  if (op=="-")  return new Sub();
  if (op=="*")  return new Mult();
  if (op=="/")  return new Div();
  if (op=="<")  return new Lt();
  if (op==">")  return new Gt();
```

```
  if (op=="<=") return new Le();
  if (op==">=") return new Ge();
  if (op=="=")  return new Eq();
  throw "unknown operator";
}
}
```

Anhang D

Formularblatt für G-Maschinenzustand



Anhang E

Gesammelte Aufgaben

Aufgabe 1 Schreiben Sie ein dem obigen funktionalen Programm äquivalentes Programm in C++ (oder Java) und lassen dieses laufen. Was stellen Sie fest? Was schließen daraus über die Auswertungsreihenfolge von C?

Aufgabe 2 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. Führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

Aufgabe 3 Sollten Sie mit Java noch nicht vertraut sein, so kompilieren Sie die beiden obigen Klassen. Schreiben Sie eine minimale Testklasse, die ein Paar und eine Liste erzeugt und auf dem Bildschirm ausgibt. (Hinweis: in der nachfolgenden Aufgabe finden Sie ein Beispiel für eine `main`-Methode und den Ausgabebefehl `println`.)

Lösung

In dieser Aufgabe soll hauptsächlich der Umgang mit Javacompiler und Interpreter geübt werden. Eine kleine Testklasse könnte dann wie folgt aussehen:

TestLi.java

```java
package name.panitz.util;
public class TestLi {
  public static void main(String [] args){
    System.out.println(new Pair<String,Integer>("hallo",42));
    System.out.println(new Li<String>());
    System.out.println(
          new Li<String>("hallo"
         ,new Li<String>("welt"
         ,new Li<String>())));
  }
}
```


Aufgabe 4 Mit den zwei oben vorgestellten Klassen, lassen sich Abbildungen realisieren. Eine endliche Abbildung läßt sich als Liste von Schlüssel/Wert-Paaren darstellen. Schreiben Sie eine in einer Klasse eine statische Methode mit folgender Signatur:

```java
public static <keyType,valueType>
valueType lookUp(keyType key,Li<Pair<keyType,valueType>> map)
```

`lookUp` soll für ein Schlüsselobjekt einen entsprechend damit gepaarten Wert in einer Liste von Paaren nachschlagen.

Testen Sie Ihre Methode mit folgender Testklasse:

TestStaticMap.java

```java
package name.panitz.util;
public class TestStaticMap {
 public static void main(String [] args){

  Li<Pair<String,Integer>> notenliste =
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("john",4),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("paul",3),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("george",1),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("ringo",4),
    new Li<Pair<String,Integer>>(
       new Pair<String,Integer>("stu",2),
    new Li<Pair<String,Integer>>())))));

  System.out.println(notenliste);
  System.out.println(StaticMap.lookUp("george",notenliste));
  System.out.println(StaticMap.lookUp("stu",notenliste));
  System.out.println(StaticMap.lookUp("pete",notenliste));
 }
}
```

Lösung

Der Rumpf der Methode hat tatsächlich nur drei Zeilen.

- Teste, ob die Liste leer ist. Für leere Liste gebe `null` aus. (Alternativ ließe sich hier auch eine Ausnahme werfen.)
- Bei nichtleeren Liste prüfe das erste Listenelement, im Erfolgsfall gib dessen Wert aus.
- Ansonsten rekursiver Aufruf auf die Restliste.

Wer bisher keine Erfahrung in Java hatte, wird wahrscheinlich statt des Aufrufs der Methode `equals` den Operator `==` verwendet haben.

StaticMap.java

```java
package name.panitz.util;
public class StaticMap {
  public static <keyType,valueType>
  valueType lookUp(keyType key,Li<Pair<keyType,valueType>> map){
    if (map.isEmpty()) return null;
    if (map.head().e1.equals(key)) return map.head().e2;
    return lookUp(key,map.tail());
  }
}
```

Aufgabe 5 Schreiben Sie jetzt eine generische Klasse `MyMap<keyType,valueType>`, die den Typ `Li<Pair<keyType,valueType>>` erweitert. Implementieren Sie in der Klasse `MyMap` die Methode
`public valueType lookUp(keyType key)`,
die für ein Schlüsselobjekt das mit diesem Schlüssel gepaarte Wertobjekt zurückgibt.

Testen Sie Ihre Implementierung mit folgender Klasse:

TestYourMap.java

```java
package name.panitz.util;
public class TestYourMap {
  public static void main(String [] args){
    MyMap<String,Integer> notenliste
     =  new MyMap<String,Integer>("john",4,
        new MyMap<String,Integer>("paul",1,
        new MyMap<String,Integer>("george",1,
        new MyMap<String,Integer>("ringo",5,
        new MyMap<String,Integer>("stu",2,
        new MyMap<String,Integer>()))))));

    System.out.println(notenliste.lookUp("john"));
    System.out.println(notenliste.lookUp("george"));
    System.out.println(notenliste.lookUp("pete"));
  }
}
```

Aufgabe 6 Implementieren Sie eine zu `NNum` analoge Klasse `NChar`, die primitive Zeichen (char) im Heap darstellen kann.

Aufgabe 7 Implementieren Sie die Klasse `NInd`, die Indirektionsknoten im Heap darstellt.

Aufgabe 8 Schreiben Sie ähnlich der Klasse `EinsZwei` eine kleine Testmethode, die den Graph aus Abbildung 2.2 erzeugt.

Aufgabe 9 Implementieren Sie in der Klasse `GmHeap` eine aussagekräftige Methode `toString`.

Aufgabe 10 Implementieren Sie eine Klasse `Pop`, die den G-Maschinenbefehl `Pop` umsetzt.

Aufgabe 11 Implementieren Sie eine Klasse `Slide`, die den G-Maschinenbefehl `Slide` umsetzt.

Aufgabe 12 Implementieren Sie eine Klasse `PushChar`, die den G-Maschinenbefehl `PushChar` umsetzt.

Aufgabe 13 Implementieren Sie den Rumpf Methode `process` nach entsprechender Spezifikation. Beachten Sie auch hier wieder, zunächst mit dem garbage collector zu checken, ob es freie Heapzellen gibt.

Aufgabe 14 Implementieren Sie die Klassen `Sub`, `Mult` und `Div`.

Aufgabe 15 Schreiben Sie die Klassen für die übrigen Vergleichsoperatoren: `Eq`, `Gt`, `Ge` und `Le`.

Aufgabe 16 Implementieren Sie die Klasse `MkAp`, die den entsprechenden G-Maschinenbefehl realisiert. Denken Sie wieder daran, zunächst einen Check vom garbage collection Objekt durchführen zu lassen.

Aufgabe 17 Implementieren Sie die Klasse `Update`.

Aufgabe 18 Führen Sie die manuelle Ausführung des Programm `square` in der G-Maschine fort. Zeichnen Sie für jeden Ausführungsschritt Heap und Stack der Maschine.

Aufgabe 19 Schreiben Sie ein kleines Testprogramm, das den Code für das Programm `square` in der G-Maschine ausführt.

Aufgabe 20 Es war ein mühsames Geschäft, die G-Maschine von Hand auszuführen. Schön wäre eine spezielle G-Maschine die Schrittweise alle Zustände, die sie bei der Ausführung durchläuft, loggt.

In objektorientierten Sprachen läßt sich dieses leicht implementieren, indem eine Unterklasse von `GmState` geschrieben wird, die nur die Methode `evalStep` überschreibt, so daß in dieser Methode vor dem Aufruf der gleichen Methode aus der Oberklasse, der Maschinenzustand auf die Konsole ausgegeben wird.

```java
{System.out.println(this);super.evalStep();}
```

Schreiben Sie eine solche Unterklasse `GmStateLog` und führen Sie mit dieser den Code aus der letzten Aufgabe aus.

Aufgabe 21 Implementieren Sie die Klasse `Jump`.

Aufgabe 22 Jetzt sind Sie dran. Ergänzen Sie die Switschanweisung für alle weiteren Befehle der G-Maschine. In der Klasse `DataInputStream` stehen Ihnen Methoden `ReadInt` und `readChar` zur verfügung.

Aufgabe 23 Mit der obigen Klasse `ForwardNode` läßt sich eine Garbage Collecion für die G-Maschine umsetzen. Schreiben Sie eine Klasse `TwoSpaceCopyGC`, die die Klasse `GC` erweitert. Sie hat einen Konstruktor, in dem die Maschinenzustandsobjekt übergeben wird:

```
  public TwoSpaceCopyGC(GmState st)
```

Beim Aufruf von `check(i)` sollen, falls keine `i` Speicherplätze mehr im Heap sind, alle noch gebrauchten Heapknoten in einen neuen Heap copiert werden. Der neue Heap wird schließlich der neue Heap im Maschinenzustand. Denken Sie daran auf Stack und den im Dump gespeicherten Stacks abgelegten Adressen auf die Adressen im neuen Heap abgelegt werden müssen. Vergessen Sie nicht auch die globalen Funktionsknoten des Heaps zu kopieren.

Aufgabe 24 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbole: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Aufgabe 25 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzten Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

Aufgabe 26 Implementieren Sie einen Parser für die Sprache Fab4. Hierzu ist für die folgende Header-Datei eine Implementierung zu erstellen:

Fab4Parser.h

```
#ifndef __FAB4PARSER_H
#define __FAB4PARSER_H

#include "../name/panitz/parser/ParsLib.h"
#include "Fab4.h"
#include "Fab4Lexer.h"
#include <string>
#include <iostream>

using namespace name::panitz::parser;
using namespace std;

namespace fab4 {
ParsResult<Fab4*,Token>* parsFab4(string fileName);
}
#endif

```


Hinweis: Ihr Programm wird etwas übersichtlicher und leichter zu lesen, wenn sie sich der Möglichkeit von Typsynonymen bedienen. Folgende Typsynonyme sind dabei sinnvoll:

```
typedef CAF<Parser<Token*,Token>*>         PToken;
typedef CAF<Parser<Operator*,Token>*>      POperator;
typedef CAF<Parser<Number*,Token>*>        PNumber;
typedef CAF<Parser<StringLiteral*,Token>*> PStringLit;
typedef CAF<Parser<CharLiteral*,Token>*>   PCharLit;
```

Aufgabe 27 Testen Sie Ihren Parser mit folgendem kleinen Hauptprogramm anhand einer Reihe unterschiedlicher Fab4-Programme.

TestFab4Parser.cpp

```
#include "Fab4Parser.h"
using namespace fab4;
using namespace name::panitz::parser;

int main(int argc,char* args[]){
  vector<Token*> xs;

  cout<<"PairCount: "<<PairCount<<endl;
  cout<<"ParsResultCount: "<<ParsResultCount<<endl;

  ParsResult<Fab4*,Token>* res = parsFab4(args[1]);
  cout<<show(res)<<endl;

  cout<<"PairCount: "<<PairCount<<endl;
  cout<<"ParsResultCount: "<<ParsResultCount<<endl;
}

```

Aufgabe 28 Ergänzen Sie den Rumpf der Methode `eval` für `NumExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp

```
    virtual void eval(NumExpr* arg){
```

GenCode.cpp

```
    }
```

Aufgabe 29 Ergänzen Sie den Rumpf der Methode `eval` für Variablennamen, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

Um zu testen, ob der Variablenname ein Funktionsname ist benutzen Sie die Funktion `contains` aus Modul `Instruction` mit folgender Gleichheitsfunktion:

GenCode.cpp

```
    static bool stringEq(string s1,string s2){return s1==s2;}
```

GenCode.cpp
```
    virtual void eval(Var* arg){
```

GenCode.cpp
```
    }
```

Aufgabe 30 Ergänzen Sie den Rumpf der Methode `eval` für Funktionsapplikationen `App`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(App* arg){
```

GenCode.cpp
```
    }
```

Aufgabe 31 Ergänzen Sie den Rumpf der Methode `eval` für Operatorausdrücke `OpExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(OpExpr* arg){
```

GenCode.cpp
```
    }
```

Aufgabe 32 Ergänzen Sie den Rumpf der Methode `eval` für Zeichenkonstanten `CharExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(CharExpr* arg){
```

GenCode.cpp
```
    }
```

Aufgabe 33 Ergänzen Sie den Rumpf der Methode `eval` für Konstruktoraufrufe `Constr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(Constr* arg){
```

GenCode.cpp
```
    }
```

Aufgabe 34 Ergänzen Sie den Rumpf der Methode `eval` für case-Ausdrücke `caseExpr`, so daß der korrekte G-Maschinencode dem Ergebnisvektor `result` zugefügt wird.

GenCode.cpp
```
    virtual void eval(CaseExpr* arg){
```


Lösung

Diese Lösung ist weit von einer Musterlösung entfernt. Insbesondere der neue Besucher, der mit `new` erzeugt wird, sollte auch wieder aus dem Speicher gelöscht werden. Auch sonst idt die Lösung unübersichtlich und kaum mehr zu verstehen.

GenCode.cpp
```

      arg->e1->visit(this);
      result->insert(result->end(),new Eval());
      vector<CaseAlt*>* alts = arg->alts;
      vector<int>* jumpPointToEnd= new vector<int>();
      vector<Instruction*>* code=new vector<Instruction*>();

      CaseJump* jump=new CaseJump();
      result->insert(result->end(),jump);
      int startAddress = 0;
      for (int i=0;i<alts->size();i++){
        CaseAlt* alt=(*alts)[i];
        (*jump)[constructors[alt->constr]]=startAddress;

        vector<Instruction*>* altCode=new vector<Instruction*>();
        int n= alt->vars.size();
        altCode->insert(altCode->end(),new Split(n));
        vector<Pair<string,int> > newEnv=env;
        for (int j=0;j<env.size();j++)
          newEnv[j].snd=newEnv[j].snd+n;
        for (int j=0;j<n;j++){
          newEnv.insert(newEnv.begin(),Pair<string,int>(alt->vars[j],j));
        }
        alt->alt->visit(new GenExprCode
         (altCode,newEnv,functions,constructors,constructorArity));
        altCode->insert(altCode->end(),new Slide(n));
        altCode->insert(altCode->end(),new Jump(0));
        code->insert(code->end(),altCode->begin(),altCode->end());;
        startAddress=code->size();
        jumpPointToEnd->insert(jumpPointToEnd->end(),startAddress-1);
     }
     for (int i=0;i<jumpPointToEnd->size();i++){
      int jumpPoint=(*jumpPointToEnd)[i];
      (*code)[jumpPoint]=new Jump(startAddress-jumpPoint-1);
     }
     result->insert(result->end(),code->begin(),code->end());;

```

GenCode.cpp
```
    }
```

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[AU77] A.V. Aho and J.D Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[Aug84] Lennart Augustsson. A compiler for Lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 218–227, 1984.

[BCK+01] G. Bracha, N. Cohen, Chr. Kemper, St. Marx, S. E. Panitz, D. Stoutamire, K. Thorup, and Ph. Wadler. Adding generics to the java programming language: Participant draft specification. Java Community Process 14, 4 2001.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions of Information Theory*, 2:113–124, 1956.

[FL89] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[FY69] R. R. Fenichel and J. C. Yochelson. A lsip garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GKS01] Ulrich Grude, Chritsoph Knabe, and Andreas Solymosi. Compilerbau. Skript zur Vorlesung, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptCompilerbau.pdf`.

[GM95] Andy Gill and Simon Marlow. Happy: the parser generator for Haskell. Technical report, University of Glasgow, 1995.

[Gru01] Ulrich Grude. Einführung in gentle. Skript, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptGentle.pdf`.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler compiler. Technical Report 39, Bell Laboratories, 1975.

[KvEN+90] Pieter Koopman, Marko van Eekelen, Eric Nöcker, Sjaak Smetsers, and Rinus Plasmeijer. The abc-machine: A sequential stack-based abstract machine for graph rewriting. In *Proc. of the Sec. Intern. Workshop on Implementation of Functional Languages on Parallel Architectures*, number 90–16 in Technical Report, pages 297–321. University of Nijmegen, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

[PHA+97] John Peterson [ed.], Kevin Hammond [ed.], Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell: A non-strict, purely functional language, Version 1.4, 1997.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[PJHH+92] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1992.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[PJS89] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Functional Programming Languages and Computer Architecture*, pages 184–201. ACM Press, 1989.

[Pv97] M. J. Plasmeijer and M van Eekelen. *Concurrent Clean Language Report 1.2*. University of Nijmegen, 1997.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer, 1985.

[Web02] Helmut Weber. Compiler. Skript zur Vorlesung, FH Wiesbaden, 2002.

[WG84] William M. Waite and Garhard Goos. *Compiler Construction*. Springer-Verlag, 1984.