Algebraische Typen und Besucher

Eine Übung in Java 1.5

(Entwurf)

Sven Eric Panitz

TFH Berlin

Version 28. April 2004

Mit Java 1.5 werden generische Typen in die Sprache eingeführt. Funktionale Sprachen kennen algebraische Typen als deklarative Typbeschreibung. Funktionen über algebraische Typen lassen sich über *pattern matching* in funktionalen Sprachen definieren.

In diesem Papier wird aufgezeigt, wie sich die deklarative Programmiertechnik der algebraischen Typen in Java realisieren läßt. Dabei findet das sogenannte Besuchsmuster Anwendung. Es wird eine Syntax für algebraische Klassen in Java vorgeschlagen und ein Programm zur Generierung von Klassen für algebraische Typen vorgestellt.

Der Quelltext dieses Papiers ist eine XML-Datei, die über XQuery- und XSLT-Skripte in entsprechende Druckformate konvertiert wurde. Die Beispielprogramme können aus dem XML Quelltext extrahiert und übersetzt werden.

Inhaltsverzeichnis

1 Einführung

Eine Hauptunterscheidung des objektorientierten zum funktionalen Programmierparadigma besteht in der Organisationsstruktur des Programms. In der Objektorientierung werden bestimmte Daten mit auf diesen Daten anwendbaren Funktionsteilen organisatorisch in Klassen gebündelt. Die Methoden einer Klassen können als die Teildefinition des Gesamtalgorithmus betrachtet werden. Der gesamte Algorithmus ist die Zusammenfassung aller überschriebenen Methodendefinitionen in unterschiedlichen Klassen.

In der funktionalen Programmierung werden organisatorisch die Funktionsdefinitionen gebündelt. Alle unterschiedlichen Anwendungsfälle finden sich untereinander geschrieben. Die unterschiedlichen Fälle für verschiedene Daten werden durch Fallunterscheidungen definiert.

Beide Programmierparadigmen haben ihre Stärken und Schwächen. Die objektorientierte Sicht ist insbesondere sehr flexibel in der Modularisierung von Programmen. Neue Klassen können geschrieben werden, für die für alle Algorithmen der Spezialfall für diese neue Art von Daten in einer neuen Methodendefinition überschrieben werden kann. Die andere Klassen brauchen hierzu nicht geändert zu werden.

In der funktionalen Sicht lassen sich einfacher neue Algorithmen auf bestehende feste Datenstrukturen hinzufügen. Während in der objektorientierten Sicht in jeder Klasse der entsprechende Fall hinzugefügt werden muß, kann eine gebündelte Funktionsdefinition geschrieben werden.

In objektorientierten Sprachen werden in der Regel Objekte als Parameter übergeben. In der funktionalen Programmierung werden oft nicht nur Daten, sondern auch Funktionen als Parameter übergeben.

Die beiden Programmierparadigmen schließen sich nicht gegenseitig aus: in modernen funktionalen Programmiersprachen läßt sich auch nach der objektorientierten Sichtweise programmieren und umgekehrt erlaubt eine objektorientierte Sprache auch eine funktionale Programmierweise. Im folgenden werden wir untersuchen, wie sich in Java funktional programmieren läßt.

1.1 Funktionale Programmierung

Bevor wir uns der Javaprogrammierung zuwenden, werfen wir einmal einen Blick über den Tellerand auf die funktionale Programmiersprache Haskell.

1.1.1 Fallunterscheidungen

Der Hauptbildungsblock in funktionalen Sprachen stellt die Funktionsdefinition dar. Naturgemäß finden sich in einer komplexen Funktionsdefinition viele unterschiedliche Fälle, die zu unterscheiden sind. Daher bieten moderne funktionale Programmiersprachen wie Haskell[?], Clean[PvE95], ML[MTH90][Ler97] viele Möglichkeiten, Fallunterscheidungen auszudrücken an. Hierzu gehören *pattern matching* und *guards*. Man betrachte z.B. das folgende Haskellprogramm, das die Fakultät berechnet:

Fakul1.hs
```
fak 0 = 1
fak n = n*fak (n-1)

main = print (fak 5)
```

Hier wird die Fallunterscheidung durch zwei Funktionsgleichungen ausgedrückt, die auf bestimmte Daten passen. Die erste Gleichung kommt zur Anwendung, wenn als Argument ein Ausdruck mit dem Wert `0` übergeben wird, die zweite Gleichung andernfalls.

Folgende Variante des Programms in Haskell, benutzt statt des *pattern matchings* sogenannte *guards*.

Fakul2.hs
```
fak n
  |n==0      = 1
  |otherwise = n*fak (n-1)

main = print (fak 5)
```

Beide Ausdrucksmittel zur Fallunterscheidung können in funktionalen Sprachen auch gemischt benutzt werden. Damit lassen sich komplexe Fallunterscheidungen elegant und übersichtlich untereinander schreiben. Das klassische verschachtelte `if`-Konstrukt kommt in funktionalen Programmen nicht zur Anwendung.[1]

1.1.2 Generische Typen

Generische Typen sind ein integraler Bestandteil des in funktionalen Programmiersprachen zur Anwendung kommenden Milner Typsystems[Mil78]. Funktionen lassen sich auf naive Weise generisch schreiben.

Hierzu betrachte man folgenes kleines Haskellprogramm, in dem von einer Liste das letzte Element zurückgegeben wird.

Last.hs
```
lastOfList [x] = x
lastOfList (_:xs) = lastOfList xs

main = do
  print (lastOfList [1,2,3,4,5])
  print (lastOfList ["du","freches","lüderliches","weib"])
```

Die Funktion `lastOfList` ist generisch über den Elementtypen des Listenparameters.

[1] Zumindest nicht, wenn der Programmierer einigermaßen gesund im Kopf ist.

1.1.3 Objektorientierte Programmierung in Haskell

In vielen Problemfällen ist die objektorientierte Sichtweise von Vorteil. Hierfür gibt es in Haskell Typklassen, die in etwa den Schnittstellen von Java entsprechen.

Zunächst definieren wir hierzu eine Typklasse, die in unserem Fall genau eine Funktion enthalten soll:

DoubleThis.hs

```
class DoubleThis a where
  doubleThis :: a -> a
```

Jetzt können wir Listen mit beliebigen Elementtypen zur Instanz dieser Typklasse machen. In Javaterminologie würden wir sagen: Listen implementieren die Schnittstelle `DoubleThis`.

DoubleThis.hs

```
instance DoubleThis [a] where
  doubleThis s =  s ++  s
```

Ebenso können wir ganze Zahlen zur Instanz der Typklasse machen:

DoubleThis.hs

```
instance DoubleThis Integer where
  doubleThis x = 2*x
```

Jetzt existiert die Funktion `doubleThis` für die zwei Typen und kann überladen angewendet werden.

DoubleThis.hs

```
main = do
  print (doubleThis "hallo")
  print (doubleThis [1,2,3,4])
  print (doubleThis (21::Integer))
```

Soweit unser erster Exkurs in die funktionale Programmierung.

1.2 Java

Java ist zunächst als rein objektorientierte Sprache entworfen worden. Es wurde insbesondere auch auf reine Funktionsdaten verzichtet. Es können nicht nackte Funktionen wie in Haskell als Parameter übergeben werden, noch gibt es das Konzept des Funktionszeigers wie z.B. in C. Erst recht kein Konstrukt, das dem *pattern matching* oder den *guards* aus funktionalen Sprachen ähnelt, ist bekannt.

1.2.1 Überladung: pattern matching des armen Mannes

Es gibt auf dem ersten Blick eine Möglichkeit in Java, die einem *pattern match* äußerlich sehr ähnlich sieht: überladene Methoden.

Beispiel:
In folgender Klasse ist die Methode `doubleThis` überladen, wie im entsprechenden Haskellprogramm oben. Einmal für `Integer` und einmal für Listen.

JavaDoubleThis.java

```
package example;
import java.util.*;
public class JavaDoubleThis {
  public static Integer doubleThis(Integer x){return 2*x;}
  public static List<Integer> doubleThis(List<Integer> xs){
    List<Integer> result = new ArrayList<Integer>();
    result.addAll(xs);
    result.addAll(xs);
    return result;
  }
  public static void main(String _){
    System.out.println(doubleThis(21));
    List<Integer> xs = new ArrayList<Integer>();
    xs.add(1);
    xs.add(2);
    xs.add(3);
    xs.add(4);
    System.out.println(doubleThis(xs));
  }
}
```

Das Überladen von Methoden in Java birgt aber eine Gefahr. Es wird während der Übersetzungtszeit statisch ausgewertet, welche der überladenen Methodendefinition anzuwenden ist. Es findet hier also keine späte Bindung statt. Wir werden in den nachfolgenden Abschnitten sehen, wie dieses umgangen werden kann.

1.2.2 Generische Typen

Mit der Version Java 1.5 wird das Typsystem von Java auf generische Typen erweitert. Damit lassen sich typische Containerklasse variabel über den in Ihnen enthaltenen Elementtypen schreiben, aber darüberhinaus lassen sich allgemeine Klassen und Schnittstellen zur Darstellung von Abbildungen und Funktionen schreiben. Erst mit der statischen Typsicherheit der generischen Typen lassen sich komplexe Programmiermuster auch in Java ohne fehleranfällige dynamische Typzusicherungen schreiben.

Schon in den Anfangsjahren von Java gab es Vorschläge Java um viele aus der funktionalen Programmierung bekannte Konstrukte, insbesondere generische Typen, zu erweitern. In Pizza[OW97] wurden neben den mitlerweile realisierten generischen Typen, algebraische Typen mit *pattern matching* und auch Funktionsobjekte implementiert.

2 Algebraische Typen

Wir haben in den Vorgängervorlesungen Datenstrukturen für Listen und Bäume bereits auf algebraische Weise definiert. Dabei sind wir von einer Menge unterschiedlicher Konstruktoren ausgegangen. Selektormethoden ergeben sich naturgemäß daraus, daß die Argumente der

Konstruktoren aus dem entstandenen Objekt wieder zu selektieren sind. Ebenso natürlich ergeben sich Testmethoden, die prüfen, mit welchem Konstruktor die Daten konstruiert wurden. Prinzipiell bedarf es nur die Menge der Konstruktoren mit ihren Parametertypen anzugeben, um einen algebraischen Typen zu definieren.

2.1 Algebraische Typen in funktionalen Sprachen

In Haskell (mit leichten Syntaxabweichungen ebenso in Clean und ML) können algebraische Typen direkt über die Aufzählung ihrer Konstruktoren definniert werden. Hierzu dient das Schlüsselwort `data` gefolgt von dem Namen des zu definierenden Typen. Auf der linken Seite eines Gleichheitszeichens folgt die Liste der Konstruktoren. Die Liste ist durch vertikale Striche `|` getrennt.

Algebarische Typen können dabei generisch über einen Elementtypen sein.

Beispiel:
Wir definieren einen algebraischen Typen für Binärbäume in Haskell. Der Typ heiße `Tree` und sei generisch über die im Baum gespeicherten Elemente. Hierfür steht die Typvariabel `a`. Es gibt zwei Konstruktoren:

- `Empty`: zur Erzeugung leerer Bäume
- `Branch`: zur Erzeugung einer Baumverzweigung. `Branch` hat drei Argumente: einen linken und einen rechten Teilbaum, sowie ein Element an der Verzweigung.

HsTree.hs
```
data Tree a =  Branch (Tree a) a (Tree a)
              |Empty
```

Über *pattern matching* lassen sich jetzt Funktionen auf Binärbäumen in Form von Funktionsgleichungen definieren.

Die Funktion `size` berechnet die Anzahl der in einem Binärbaum gespeicherten Elemente:

HsTree.hs
```
size Empty = 0
size (Branch l _ r) = 1+size l+size r
```

Eine weitere Funktion transformiere einen Binärbaum in eine Liste. Hierfür schreiben wir drei Funktionsgleichungen. Man beachte, daß *pattern matching* beliebig tief verschachtelt auftreten kann.

HsTree.hs
```
flatten Empty = []
flatten (Branch Empty x xs) = (x: (flatten xs))
flatten (Branch (Branch l y r) x xs)
  = flatten (Branch l  y (Branch r x xs))
```

(Der Infixkonstruktor des Doppelpunkts `(:)` ist in Haskell der Konstruktor `Cons` für Listen, das Klammernpaar `[]` konstruiert eine leere Liste.)

Zur Illustration eine kleine Testanwendung der beiden Funktionen:

HsTree.hs
```
main =  do
  print (size (Branch Empty "hallo" Empty))
  print (flatten (Branch (Branch Empty "freches" Empty)
                         "lüderliches"
                         (Branch Empty "Weib" Empty )
                 )
        )
```

Hiermit wollen wir vorerst den zweiten kleinen Exkurs in die funktionale Programmierung verlassen und untersuchen, wie wir algebraische Typen in Java umsetzen können.

2.2 Implementierungen in Java

Wir wollen bei dem obigen Haskellbeispiel für Binärbäume bleiben und auf unterschiedliche Arten die entsprechende Spezifikation in Java umsetzen.

2.2.1 Objektmethoden

Die natürliche objektorientierte Vorgehensweise ist, für jeden Konstruktor eines algebraischen Typen eine eigene Klasse vorzusehen und die unterschiedlichen Funktionsgleichungen auf die unterschiedlichen Klassen zu verteilen. Hierzu können wir für die Binärbäume eine gemeinsame abstrakte Oberklasse für alle Binärbäume vorsehen, in der die zu implementierenden Methoden abstrakt sind:

Tree.java
```
package example;
public abstract class Tree<a>{
  public abstract int size();
  public abstract java.util.List<a> flatten();
}
```

Jetzt lassen sich für beide Konstruktoren jeweils eine Klasse schreiben, in der die entsprechenden Funktionsgleichungen implementiert werden. Dieses ist für den parameterlosen Konstruktor `Empty` sehr einfach:

Empty.java
```
package example;
public class Empty<a> extends Tree<a>{
  public int size(){return 0;}
  public java.util.List<a> flatten(){
    return new java.util.ArrayList<a>() ;
  }
}
```

Für den zweiten Konstruktor entsteht eine ungleich komplexere Klasse,

Branch.java
```
package example;
public class Branch<a> extends Tree<a>{
```

Zunächst sehen wir drei interne Felder vor, um die dem Konstruktor übergebenen Objekte zu speichern:

Branch.java
```
  private a element;
  private Tree<a> left;
  private Tree<a> right;
```

Für jedes dieser Felder läßt sich eine Selektormethode schreiben:

Branch.java
```
  public a getElement(){return element;}
  public Tree<a> getLeft(){return left;}
  public Tree<a> getRight(){return right;}
```

Und natürlich benötigen wir auch einen eigentlichen Konstruktor der Klasse:

Branch.java
```
  public Branch(Tree<a> l,a e,Tree<a> r){
    left=l;element=e;right=r;
  }
```

Schließlich sind noch die entsprechenden Funktionsgleichungen umzusetzen. Im Falle der Funktion `size` ist dieses noch relativ einfach.

Branch.java
```
  public int size(){return 1+getLeft().size()+getRight().size();}
```

Für die Methode `flatten` wird dieses schon sehr komplex. Der innerer des verschachtelten *pattern matches* aus der Haskellimplementierung kann nur noch durch eine `if`-Abfrage durchgeführt werden. Es entstehen zwei Fälle. Zunächst der Fall, daß der linke Teilbaum leer ist:

Branch.java
```
  public java.util.List<a> flatten(){
    if (getLeft() instanceof Empty){
      java.util.List<a> result = new java.util.ArrayList<a>();
      result.add(getElement());
      result.addAll(getRight().flatten());
      return result;
    }
```

Und anschließend der Fall, in dem der linke Teilbaum nicht leer ist:

Branch.java
```
    Branch<a> theLeft = (Branch<a>)getLeft();
    return new Branch<a>(theLeft.getLeft()
                        ,theLeft.getElement()
                        ,new Branch<a>(theLeft.getRight()
                                      ,getElement()
                                      ,getRight())
                        ).flatten() ;
  }
```

Die entsprechende Testmethode aus der Haskellimplementierung sieht in Java wie folgt aus.

Branch.java
```
  public static void main(String []_){
    System.out.println(
      new Branch<String>
        (new Empty<String>(),"hallo",new Empty<String>())
        .size());
    System.out.println(
      new Branch<String>
       (new Branch<String>
          (new Empty<String>(),"freches",new Empty<String>())
       ,"lüderliches"
       ,new Branch<String>
          (new Empty<String>(),"Weib",new Empty<String>())
       ).flatten());
  }
}
```

2.2.2 Fallunterscheidung in einer Methode

Im letzten Abschnitt haben wir die Funktionen auf Binärbäumen auf verschiedene Unterklassen verteilt. Alternativ können wir natürlich eine Methode schreiben, die alle Fälle enthält und in der die Fallunterscheidung vollständig vorgenommen wird. Im Falle der Funktion `size` erhalten wir folgende Methode:

Size.java
```
package example;
class Size{
  static public <a> int size(Tree<a> t){
    if (t instanceof Empty) return 0;
    if (t instanceof Branch<a>) {
      Branch<a> dies = (Branch<a>) t;
      return
         1+size(dies.getLeft())
          +size(dies.getRight());
    }
    throw new RuntimeException("unmatched pattern: "+t);
  };
```

```
  public static void main(String [] _){
    System.out.println(size(
     new Branch<String>(new Empty<String>()
                       ,"hallo"
                       ,new Empty<String>())));
  }
}
```

Man sieht, daß die Unterscheidung über `if`-Bedingungen und `instanceof`-Ausdrücken mit Typzusicherungen recht komplex werden kann. Hiervon kann man sich insbesondere überzeugen, wenn man die Funktion `flatten` auf diese Weise schreibt:

Flatten.java
```
package example;
import java.util.*;
class Flatten{
  static public <a> List<a> flatten(Tree<a> t){
    if (t instanceof Empty) return new ArrayList<a>();
    Branch<a> dies = (Branch<a>) t;
    if (dies.getLeft() instanceof Empty){
      List<a> result = new ArrayList<a>();
      result.add(dies.getElement());
      result.addAll(flatten(dies.getRight()));
      return result;
    }
    Branch<a> theLeft = (Branch<a>)dies.getLeft();
    return flatten(
           new Branch<a>(theLeft.getLeft()
                        ,theLeft.getElement()
                        ,new Branch<a>(theLeft.getRight()
                                      ,dies.getElement()
                                      ,dies.getRight())
                        )) ;
  }
```

Wahrscheinlich ist die Funktion `flatten` auf diese Weise geschrieben schon kaum noch nachzuvollziehbar.

Zumindest in einen kleinem Test, wollen wir uns davon versichern, daß diese Methode wunschgemäß funktioniert:

Flatten.java
```
  public static void main(String []_){
    System.out.println(flatten(
      new Branch<String>
       (new Branch<String>
          (new Empty<String>(),"freches",new Empty<String>())
       ,"lüderliches"
       ,new Branch<String>
          (new Empty<String>(),"Weib",new Empty<String>())
        )));
  }
}
```


3 Visitor

Wir haben oben zwei Techniken kennengelernt, wie man in Java Funktionen über baumartige Strukturen schreiben kann. In der einen finden sich die Funktionen sehr verteilt auf unterschiedliche Klassen, in der anderen erhalten wir eine sehr komplexe Funktion mit vielen schwer zu verstehenden Fallunterscheidungen. Mit dem Besuchsmuster[GHJV95] lassen sich Funktionen über baumartige Strukturen schreiben, so daß die Funktionsdefinition in einer einer Klasse gebündelt auftritt und trotzdem nicht ein großes Methodenkonglomerat mit vielen Fallunterscheidungen ensteht.

3.1 Besucherobjekte als Funktionen über algebraische Typen

Ziel ist es, Funktionen über baumartige Strukturen zu schreiben, die in einer Klasse gebündelt sind. Diese Klasse stellt dann die Funktion dar. Wir bedienen uns daher einer Schnittstelle, die von unseren Funktionsklassen implementiert werden soll. In [Pan04] haben wir bereits eine Schnittstelle zur Darstellung einstelliger Funktionen dargestellt. Diese werden wir fortan unter den Namen `Visitor` benutzen.

Visitor.java
```
package name.panitz.crempel.util;

public interface Visitor<arg,result>
                      extends UnaryFunction<arg,result>{
}
```

Diese Schnittstelle ist generisch. Die Typvariable `arg` steht für den Typ der baumartigen Struktur (der algebraische Typ) über die wir eine Funktion schreiben wollen; die Typvariable `result` für den Ergebnistyp der zu realisierenden Funktion.

In einem Besucher für einen bestimmten algebraischen Typen werden wir verlangen, daß für jeden Konstruktorfall die Auswertungsmethode `eval` überladen ist. Für die Binärbäume, wie wir sie bisher definiert haben, erhalten wir die folgende Schnittstelle:

TreeVisitor.java
```
package example;
import name.panitz.crempel.util.Visitor;

public interface TreeVisitor<a,result>
                     extends Visitor<Tree<a>,result>{
  public result eval(Tree<a> t);
  public result eval(Branch<a> t);
  public result eval(Empty<a> t);
}
```

Diese Schnittstelle läßt sich jetzt für jede zu realisierende Funktion auf Bäumen implementieren. Für die Funktion `size` erhalten wir dann folgende Klasse.

TreeSizeVisitor.java

```
package example;

public class TreeSizeVisitor<a> implements TreeVisitor<a,Integer>{
  public Integer eval(Branch<a> t){
    return 1+eval(t.getLeft())+eval(t.getRight());
  }

  public Integer eval(Empty<a> t){return 0;}

  public Integer eval(Tree<a> t){
    throw new RuntimeException("unmatched pattern: "+t);
  }
}
```

Hierbei gibt es die zwei Fälle der Funktionsgleichungen aus Haskell als zwei getrennte Methodendefinitionen. Zusätzlich haben wir noch eine Methodendefinition für die gemeinsame Oberklasse `Tree` geschrieben, in der abgefangen wird, ob es noch weitere Unterklassen gibt, für die wir keinen eigenen Fall geschrieben haben.

Leider funktioniert die Implementierung über die Klasse `TreeSizeVisitor` allein nicht wie gewünscht.

TreeSizeVisitorNonWorking.java

```
package example;
public class TreeSizeVisitorNonWorking{
  public static void main(String [] _){
    Tree<String> t = new Branch<String>(new Empty<String>()
                                        ,"hallo"
                                        ,new Empty<String>());
    System.out.println(new TreeSizeVisitor<String>().eval(t));
  }
}
```

Starten wir den kleinen Test, so stellen wir fest, daß die allgemeine Version für `eval` auf der Oberklasse `Tree` ausgeführt wird:

```
sep@linux:~/fh/adt/examples> java example.TreeSizeVisitorNonWorking
Exception in thread "main" java.lang.RuntimeException: unmatched pattern: example.Branch@1a46e30
        at example.TreeSizeVisitor.eval(TreeSizeVisitor.java:12)
        at example.TreeSizeVisitorNonWorking.main(TreeSizeVisitorNonWorking.java:8)
sep@linux:~/fh/adt/examples>
```

Der Grund hierfür ist, daß während der Übersetzungszeit nicht aufgelöst werden kann, ob es sich bei dem Argument der Funktion `eval` um ein Objekt der Klasse `Empty` oder `Branch` handelt und daher ein Methodenaufruf zur Methode `eval` auf `Tree` generiert wird.

3.2 Besucherobjekte und Späte-Bindung

Im letzten Abschnitt hat unsere Implementierung über eine Besuchsfunktion nicht den gewünschten Effekt gehabt, weil überladene Funktionen nicht dynamisch auf dem Objekttypen aufgelöst werden, sondern statisch während der Übersetzungszeit. Das Ziel ist eine dynamische Methodenauflösung. Dieses ist in Java nur über späte Bindung für überschriebene Methoden möglich. Damit wir effektiv mit dem Besuchsobjekt arbeiten können, brauchen wir eine Methode in den Binärbäumen, die in den einzelnen Unterklassen überschrieben wird. Wir nennen diese Methode `visit`. Sie bekommt ein Besucherobjekt als Argument und wendet dieses auf das eigentliche Baumobjekt an.

Wir schreiben also eine neue Baumklasse, die vorsieht, daß sie ein Besucherobjekt bekommt.

VTree.java

```
package example;
import name.panitz.crempel.util.Visitor;

public abstract class VTree<a>{
  public <result> result visit(VTreeVisitor<a,result> v){
    return v.eval(this);
  }
}
```

Entsprechend brauchen wir für diese Baumklasse einen Besucher:

VTreeVisitor.java

```
package example;
import name.panitz.crempel.util.Visitor;

public interface VTreeVisitor<a,result>
                      extends Visitor<VTree<a>,result>{
  public result eval(VTree<a> t);
  public result eval(VBranch<a> t);
  public result eval(VEmpty<a> t);
}
```

Und definieren entsprechend der Konstruktoren wieder die Unterklassen.

Einmal für leere Bäume:

VEmpty.java

```
package example;
public class VEmpty<a> extends VTree<a>{
  public <result> result visit(VTreeVisitor<a,result> v){
    return v.eval(this);
  }
}
```

Und einmal für Baumverzweigungen:

VBranch.java

```java
package example;
public class VBranch<a> extends VTree<a>{
  private a element;
  private VTree<a> left;
  private VTree<a> right;
  public a getElement(){return element;}
  public VTree<a> getLeft(){return left;}
  public VTree<a> getRight(){return right;}
  public VBranch(VTree<a> l,a e,VTree<a> r){
    left=l;element=e;right=r;
  }
  public <result> result visit(VTreeVisitor<a,result> v){
    return v.eval(this);
  }
}
```

Jetzt können wir wieder einen Besucher definieren, der die Anzahl der Baumknoten berechnen soll. Es ist jetzt zu beachten, daß niemals der Besucher als Funktion auf Baumobjekte angewendet wird, sondern, daß der Besucher über die Methode `visit` auf Bäumen aufgerufen wird.

VTreeSizeVisitor.java

```java
package example;

public class VTreeSizeVisitor<a>
            implements VTreeVisitor<a,Integer> {
  public Integer eval(VBranch<a> t){
    return 1+t.getLeft().visit(this)+t.getRight().visit(this);
  }
  public Integer eval(VEmpty<a> t){return 0;}
  public Integer eval(VTree<a> t){
    throw new RuntimeException("unmatched pattern: "+t);
  }

}
```

Jetzt funktioniert der Besucher als Funktion wie gewünscht:

TreeSizeVisitorWorking.java

```java
package example;
public class TreeSizeVisitorWorking{
  public static void main(String [] _){
    VTree<String> t = new VBranch<String>(new VEmpty<String>()
                                        ,"hallo"
                                        ,new VEmpty<String>());
    System.out.println(t.visit(new VTreeSizeVisitor<String>()));
  }
}
```


Entsprechend läßt sich jetzt auch die Funktion `flatten` realisieren. Um das verschachtelte *pattern matching* der ursprünglichen Haskellimplementierung aufzulösen, ist es notwendig tatsächlich zwei Besuchsklassen zu schreiben: eine für den äußeren *match* und eine für den Innerern.

So schreiben wir einen Besucher für die entsprechende Funktion:

VFlattenVisitor.java

```java
package example;
import java.util.*;

public class VFlattenVisitor<a>
                   implements VTreeVisitor<a,List<a>>{
```

Der Fall eines leeren Baumes entspricht der ersten der drei Funkionsgleichungen:

VFlattenVisitor.java

```java
  public List<a> eval(VEmpty<a> t){return new ArrayList<a>();}
```

Die übrigen zwei Funktionsgleichungen, die angewendet werden, wenn es sich nicht um einen leeren Baum handelt, lassen wir von einem inneren verschachtelten Besucher erledigen. Dieser innere Besucher benötigt das Element, den rechten Teilbaum und den äußeren Besucher:

VFlattenVisitor.java

```java
  public List<a> eval(VBranch<a> t){
    return t.getLeft().visit(
      new InnerFlattenVisitor(t.getElement(),t.getRight(),this)
    );
  }

  public List<a> eval(VTree<a> t){
    throw new RuntimeException("unmatched pattern: "+t);
  }
```

Den inneren Besucher realisisren wir über eine innere Klasse. Diese braucht Felder für die drei mitgegebenen Objekte und einen entsprechenden Konstruktor:

VFlattenVisitor.java

```java
  public class InnerFlattenVisitor<a>
            implements VTreeVisitor<a,List<a>> {

    final a element;
    final VTree<a> right;
    final VFlattenVisitor<a> outer;

    InnerFlattenVisitor(a e,VTree<a> r,VFlattenVisitor<a> o){
     element=e;right=r;outer=o;}
```

Für den inneren Besucher gibt es zwei Methodendefinitionen: für die zweite und dritte Funktionsgleichung je eine. Zunächst für den Fall das der ursprüngliche linke Teilbaum leer ist:

VFlattenVisitor.java
```
    public List<a> eval(VEmpty<a> t){
      List<a> result = new ArrayList<a>();
      result.add(element);
      result.addAll(right.visit(outer));
      return result;
    }
```

Und schließlich der Fall, daß der ursprünglich linke Teilbaum nicht leer war.

VFlattenVisitor.java
```
    public List<a> eval(VBranch<a> t){
      return new VBranch<a>(t.getLeft()
                           ,t.getElement()
                           ,new VBranch<a>(t.getRight()
                                          ,element
                                          ,right)
                           ).visit(outer) ;
    }

    public List<a> eval(VTree<a> t){
      throw new RuntimeException("unmatched pattern: "+t);
    }
  }
}
```

Wie man sieht, lassen sich einstufige *pattern matches* elegant über Besucherklassen implementieren. Für verschachtelte Fälle entsteht aber doch ein recht unübersichtlicher Überbau.

4 Generierung von Klassen für algebraische Typen

Betrachten wir noch einmal die Haskellimplementierung. Mit wenigen Zeilen ließ sich sehr knapp und trotzdem genau ein generischer algebraischer Typ umsetzen. Für die Javaimplementierung war eine umständliche Codierung von Hand notwendig. Das Wesen eines Programmiermusters ist es, daß die Codierung relativ mechanisch von Hand auszuführen ist. Solch mechanische Umsetzungen lassen sich natürlich automatisieren. Wir wollen jetzt ein Programm schreiben, das für die Spezifikation eines algebraischen Typs entsprechende Javaklassen generiert.

4.1 Eine Syntax für algebraische Typen

Zunächst müssen wir eine Syntax definieren, in der wir algebraische Typen definieren wollen. Wir könnten uns der aus Haskell bekannten Syntax bedienen, aber wir ziehen es vor eine Syntax, die mehr in den Javarahmen paßt, zu definieren.

Eine algebraische Typspezifikation soll einer Klassendefinition sehr ähnlich sehen. Paket- und Implortdeklarationen werden gefolgt von einer Klassendeklaration. Diese enthält als

zusätzliches Attribut das Schlüsselwort `data`. Der Rumpf der Klasse soll nur einer Auflistung der Konstruktoren des algebraischen Typs bestehen. Diese Konstruktoren werden in der Syntax wie abstrakte Javamethoden deklariert.

Beispiel:
In der solcherhand vorgeschlagenen Syntax lassen sich Binärbäume wie folgt deklarieren:

T.adt
```
package example.tree;
data class T<a> {
  Branch(T<a> left,a element,T<a> right);
  Empty();
}
```

Einen Parser für unsere Syntax der algebraischen Typen in `javacc`-Notation befindet sich im Anhang.

4.1.1 Generierte Klassen

Aus einer Deklaration für einen algebraischen Typ wollen wir jetzt entsprechende Javaklassen generieren. Wir betrachten die generierten Klassen am Beispiel der oben deklarierten Binärbäume. Zunächst wird eine abstrakte Klasse für den algenraischen Typen generiert. In dieser Klasse soll eine Methode `visit` existieren:

```
package example.tree;

public abstract class T<a> implements TVisitable<a>{
  abstract public <b_> b_ visit(TVisitor<a,b_> visitor);
}
```

Doppelt gemoppelt können wir dieses auch noch über eine implementierte Schnittstelle ausdrücken.

```
package example.tree;

public interface TVisitable<a>{
  public <_b> _b visit(TVisitor<a,_b> visitor);}
```

Die Besucherklasse soll als abstrakte Klasse generiert werden: hier wird bereits eine Ausnahme geworfen, falls ein Fall nicht durch eine spezielle Methodenimplementierung abgedeckt ist.

```
package example.tree;
import name.panitz.crempel.util.Visitor;

public abstract class TVisitor<a,result>
  implements Visitor<T<a>,result>{
```

```
  public abstract result eval(Branch<a> _);
  public abstract result eval(Empty<a> _);
  public result eval(T<a> xs){
    throw new RuntimeException("unmatched pattern: "+xs.getClass());
  }}
```

Und schließlich sollen noch Klassen für die definierten Konstruktoren generiert werden. Diese Klassen müssen die Methode `visit` implementieren. Zusätzlich lassen wir noch sinnvolle Methoden `toString` und `equals` generieren.

In unserem Beispiel erhalten wir für den Konstruktor ohne Argumente folgende Klasse.

```
package example.tree;
public class Empty<a> extends T<a>{
  public Empty(){}

  public <_b> _b visit(TVisitor<a,_b> visitor){
    return visitor.eval(this);
  }
  public String toString(){
    return "Empty("+")";
  }
  public boolean equals(Object other){
    if (!(other instanceof Empty)) return false;
    final Empty o= (Empty) other;
    return true  ;
  }
}
```

Für die Konstruktoren mit Argumenten werden die Argumentnamen als interne Feldnamen und für die Namen der `get`-Methoden verwendet. Für den Konstruktor `Branch` wird somit folgende Klasse generiert.

```
package example.tree;

public class Branch<a> extends T<a>{
  private T<a> left;
  private a element;
  private T<a> right;

  public Branch(T<a> left,a element,T<a> right){
    this.left = left;
    this.element = element;
    this.right = right;
  }

  public T<a> getLeft(){return left;}
  public a getElement(){return element;}
  public T<a> getRight(){return right;}
```



```
  public <_b> _b visit(TVisitor<a,_b> visitor){
    return visitor.eval(this);
  }
  public String toString(){
    return "Branch("+left+","+element+","+right+")";
  }
  public boolean equals(Object other){
    if (!(other instanceof Branch)) return false;
    final Branch o= (Branch) other;
    return true  &&left.equals(o.left)
                 &&element.equals(o.element)
                 &&right.equals(o.right);
  }
}
```

4.1.2 Schreiben von Funktionen auf algebraische Typen

Wenn wir uns die Klassen generiert haben lassen, so lassen sich jetzt auf die im letztem Abschnitt vorgestellte Weise für den algebraischen Typ Besucher schreiben. Nachfolgend der hinlänglich bekannte Besucher zum Zählen der Knotenanzahl:

TSize.java

```
package example.tree;
public class TSize<a> extends TVisitor<a,Integer>{
  public Integer eval(Branch<a> x){
    return 1+size(x.getLeft())+size(x.getRight());
  }
  public Integer eval(Empty<a> _){return 0;}

  public int size(T<a> t){
    return t.visit(this);}
}
```

Es empfielt sich in einem Besucher eine allgemeine Methode, die die Funktion realisiert zu schreiben. Der Aufruf `.visit(this)` ist wenig sprechend. So haben wir in obiger Besuchsklasse die Methode `size` definiert und in rekursiven Aufrufen benutzt.

4.1.3 Verschachtelte algebraische Typen

Algebraische Typen lassen sich wie gewöhnliche Typen benutzen. Das bedeutet insbesondere, daß sie Argumenttypen von Konstruktoren anderer algebraischer Typen sein können. Hierzu definieren wir eine weitere Baumstruktur. Zunächst definieren wir einfach verkettete Listen:

Li.adt

```
package example.baum;
data class Li<a> {
  Cons(a head ,Li<a> tail);
  Empty();
}
```

Ein Baum sein nun entweder leer oder habe eine Elementmarkierung und eine Liste von Kindbäumen:

Baum.adt
```
package example.baum;
data class Baum<a> {
  Zweig(Li<Baum<a>> children,a element);
  Leer();
}
```

Im Konstruktor `Zweig` benutzen wir den algebraischen Typ `Li` der einfach verketteten Listen.

Wollen wir für diese Klasse eine Funktion schreiben, so brauchen wir zwei Besucherklassen.

Beispiel:
Für die Bäume mit beliebig großer Kinderanzahl ergeben sich folgende Besucher für das Zählen der Elemente:

BaumSize.java
```
package example.baum;
public class BaumSize<a> extends BaumVisitor<a,Integer>{
  final BaumVisitor<a,Integer> dies = this;
  final LiBaumSize inner = new LiBaumSize();

  public Integer size(Baum<a> t){return t.visit(this);}
  public Integer size(Li<Baum<a>> xs){return xs.visit(inner);}

  class LiBaumSize  extends LiVisitor<Baum<a>,Integer>{
    public Integer eval(Empty<Baum<a>> _){return 0;}
    public Integer eval(Cons<Baum<a>> xs){
      return size(xs.getHead()) + size(xs.getTail());}
  }
    public Integer eval(Zweig<a> x){
      return 1+size(x.getChildren());}
    public Integer eval(Leer<a> _){return 0;}
}
```

Wir haben die zwei Klassen mit Hilfe einer inneren Klasse realisiert. Ein kleiner Test, der diesen Besucher illustriert:

BaumSizeTest.java
```
package example.baum;
public class BaumSizeTest{
  public static void main(String [] _){
    Baum<String> b
      = new Zweig<String>
         (new Cons<Baum<String>>
           (new Leer<String>()
           ,new Cons<Baum<String>>
              (new Zweig<String>
                    (new Empty<Baum<String>>(),"welt")
              ,new Empty<Baum<String>>()))
```



```
           ,"hallo");
    System.out.println(new BaumSize<String>().size(b));
  }
}
```

Die Umsetzung der obigen Funktion über eine Klasse, die gleichzeitig eine Besucherimplementierung für Listen von Bäumen wie auch für Bäume ist gelingt nicht:

```
package example.baum;
import name.panitz.crempel.util.Visitor;

public class BaumSizeError<a>
                 implements Visitor<Baum<a>,Integer>
                          , Visitor<Li<Baum<a>>,Integer>{
  public Integer size(Baum<a> t){return t.visit(this);}
  public Integer size(Li<Baum<a>> xs){return xs.visit(this);}

  public Integer eval(Empty<Baum<a>> _){return 0;}
  public Integer eval(Cons<Baum<a>> xs){
    return size(xs.getHead())+size(xs.getTail());}
  public Integer eval(Zweig<a> x){return size(x.getChildren());}
  public Integer eval(Leer<a> _){return 0;}

  public Integer eval(Li<Baum<a>> t){
    throw new RuntimeException("unsupported pattern: "+t);}
  public Integer eval(Baum<a> t){
    throw new RuntimeException("unsupported pattern: "+t);}
}
```

Es kommt zu folgender Fehlermeldung während der Übersetzungszeit:

```
BaumSizeError.java:5: name.panitz.crempel.util.Visitor cannot
be inherited with different arguments:
    <example.baum.Baum<a>,java.lang.Integer>
and <example.baum.Li<example.baum.Baum<a>>,java.lang.Integer>
```

Der Grund liegt in der internen Umsetzung von generischen Klassen in Java 1.5. Es wird eine homogene Umsetzung gemacht. Dabei wird eine Klasse für alle möglichen Instanzen der Typvariablen erzeugt. Der Typ der Typvariablen wird dabei durch den allgemeinsten Typ, den diese erhalten kann, ersetzt (in den meisten Fällen also durch `Object`). Auf Klassendateiebene kann daher Java nicht zwischen verschiedenen Instanzen der Typvariablen unterscheiden. Der Javaübersetzer muß daher Programme, die auf einer solchen Unterscheidung basieren zurückweisen.

4.2 Java Implementierung

Nun ist es an der Zeit, das Javaprogramm zur Generierung der Klassen für eine algebraische Typspezifikation zu schreiben. Wer an die Details dieses Programmes nicht interessiert ist, sondern es lediglich zur Programmierung mit algebraischen Typen benutzen will, kann diesen Abschnitt schadlos überspringen.

Über Parser und Parsergeneratoren haben wir uns bereits im zweiten Semester unterhalten [Pan03]. Wir benutzen jetzt einen in `javacc` spezifizierten Parser für unsere Syntax für algebraische Typen. Dieser Parser generiert Objekte, die einen algebraischen Typen darstellen. Diese Objekte enthalten Methoden zur Generierung der gewünschten Javaklassen.

4.2.1 Abstrakte Typbeschreibung

Beginnen wir mit einer Klasse zur Beschreibung algebraischer Typen in Java:

AbstractDataType.java
```
package name.panitz.crempel.util.adt;

import java.util.List;
import java.util.ArrayList;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;
import java.io.IOException;

public class AbstractDataType {
```

Eine algebraische Typspezifikation hat

- einen Typnamen für den den algebraischen Typen
- eventuell eine Paketspezifikation
- eine Liste Imports
- eine Liste von Typvariablen
- eine Liste von Konstrurtordefinitionen

Für die entsprechenden Informationen halten wir jeweils ein Feld vor:

AbstractDataType.java
```
  String name;
  String thePackage;
  List<String> typeVars;
  List<String> imports;
  public List<Constructor> constructors;
```

Verschiedene Konstruktoren dienen dazu, diese Felder zu initialisieren:

AbstractDataType.java
```
  public AbstractDataType(String p,String n,List tvs,List ps){
    this(p,n,tvs,ps,new ArrayList<String>());
  }

```



```
  public AbstractDataType
               (String p,String n,List tvs,List ps,List im){
    thePackage = p;
    name=n;
    typeVars=tvs;
    constructors=ps;
    imports=im;
  }
```

Wir überschreiben aus Informationsgründen und für Debuggingzwecke die Methode `toString`, so daß sie wieder eine parsebare textuelle Darstellung der algebraischen Typspezifikation erzeugt:

AbstractDataType.java
```
  public String toString(){
    StringBuffer result = new StringBuffer("data class "+name);
    if (!typeVars.isEmpty()){
      result.append("<");
      for (String tv:typeVars){result.append("\u0020"+tv);}
      result.append(">");
    }
    result.append("{\n");
    for (Constructor c:constructors){result.append(c.toString());}
    result.append("}\n");
    return result.toString();
  }
```

Wir sehen ein paar Methoden vor, um bestimmte Information über einen algebraischen Typen zu erfragen; wie den Namen mit und ohne Typparametern oder die Liste der Typparameter:

AbstractDataType.java
```
  public String getFullName(){return name+getParamList();}

  public String getName(){return name;}

  String commaSepParams(){
    String paramList = "";
      boolean first=true;
      for (String tv:typeVars){
        if (!first)paramList=paramList+",";
        paramList=paramList+tv;
        first=false;
      }
      return paramList;
  }

  public String getParamList(){
    if (!typeVars.isEmpty()){return '<'+commaSepParams()+'>';}
    return "";
```

```
  }

  String getPackageDef(){
    return  thePackage.length()==0
            ?""
            :"package "+thePackage+";\n\n";
  }
```

Unsere eigentliche und Hauptaufgabe ist das Generieren der Javaklassen. Hierzu brauchen wir eine Klasse für den algebraischen Typ und für jeden Konstruktor eine Unterklasse. Hinzu kommt die abstrakte Besucherklasse und die Schnittstelle `Visitable`. Wir sehen jeweils eigene Methoden hierfür vor. Als Argument wird zusätzlich als `String` übergeben, in welchen Dateipfad die Klasse generiert werden sollen.

AbstractDataType.java

```
  public void generateClasses(String path){
    try{
      generateVisitorInterface(path);
      generateVisitableInterface(path);
      generateClass(path);
    }catch (IOException _){}
}
```

Beginnen wir mit der eigentlichen Klasse für den algebraischen Typen. Hier gibt es einen Sonderfall, den wir extra behandeln: wenn der algebraische Typ nur einen Konstruktor enthält, so wird auf die Generierung der Typhierarchie verzichtet.

AbstractDataType.java

```
  public void generateClass(String path) throws IOException{
    String fullName = getFullName();
```

Zunächst also der Normalfall mit mehreren Konstruktoren: Wir schreiben eine Datei mit den Quelltext einer abstrakten Javaklasse. Anschließend lassen wir die Klassen für die Konstruktoren generieren.

AbstractDataType.java

```
    if (constructors.size()!=1){
      FileWriter out = new FileWriter(path+"/"+name+".java");
      out.write( getPackageDef());
      writeImports(out);
      out.write("public abstract class ");
      out.write(fullName);
      out.write(" implements "+name+"Visitable"+getParamList());
      out.write("{\n");
      out.write("  abstract public <b_> b_ visit("
              +name+"Visitor<"+commaSepParams()
                              +(typeVars.isEmpty()?"":",")
                              +"b_> visitor);\n");
      out.write("}");
      out.close();
      for (Constructor c:constructors){c.generateClass(path,this);}
```


Andernfalls wird nur eine Klasse für den einzigen Konstruktor generiert. Eine bool'sches Argument markiert, daß es sich hierbei um den Spezialfall eines einzelnen Konstruktors handelt.

AbstractDataType.java

```
    }else{constructors.get(0).generateClass(path,this,true);}
  }
```

Wir haben eine kleine Hilfsmethode benutzt, die auf einem Ausgabestrom die Importdeklarationen schreibt:

AbstractDataType.java

```
  void writeImports(Writer out) throws IOException{
    for (String imp:imports){
      out.write("\nimport ");
      out.write(imp);
    }
    out.write("\n\n");
  }
```

Es verbleiben die Besucherklassen. Zunächst läßt sich relativ einfach die Schnittstelle `Visitable` für den algebraischen Typen generieren:

AbstractDataType.java

```
  public void generateVisitableInterface(String path){
    try{
      final String interfaceName = name+"Visitable";
      final String fullName = interfaceName+getParamList();
      FileWriter out
        = new FileWriter(path+"/"+interfaceName+".java");
      out.write( getPackageDef());
      writeImports(out);
      out.write("public interface ");
      out.write(fullName+"{\n");
      out.write("  public <_b> _b visit("
                +name+"Visitor<"+commaSepParams()
                                +(typeVars.isEmpty()?"":",")
                                +"_b> visitor);");
      out.write("}");
      out.close();
    }catch (Exception _){}
  }
```

Etwas komplexer gestaltet sich die Methode zur Generierung der abstrakten Besucherklasse. Hier wird für jeden Konstruktor des algebraischen Typens eine abstrakte Methode `eval` überladen. Zusätzlich gibt es noch die Standardmethode `eval`, die für die gemeinsame Oberklasse der Konstruktorklassen überladen ist. In dieser Methode wird eine Ausnahme für unbekannte Konstruktorklassen geworfen.

AbstractDataType.java
```
  public void generateVisitorInterface(String path){
    try{
      final String interfaceName = name+"Visitor";
      final String fullName
       = interfaceName+"<"
                      +commaSepParams()
                      +(typeVars.isEmpty()?"":",")
                      +"result>";
      FileWriter out
       = new FileWriter(path+"/"+interfaceName+".java");
      out.write( getPackageDef());
      out.write( "\n");
      out.write("import name.panitz.crempel.util.Visitor;\n");
      writeImports(out);
      out.write("public abstract class ");
      out.write(fullName+" implements Visitor<");
      out.write(getFullName()+",result>{\n");
      if (constructors.size()!=1){
        for (Constructor c:constructors)
          out.write("  "+c.makeEvalMethod(this)+"\n");
      }

      out.write("  public result eval("+getFullName() +" xs){\n");
      out.write("    throw new RuntimeException(");
      out.write("\"unmatched pattern: \"+xs.getClass());\n");
      out.write("  }");

      out.write("}");
      out.close();
    }catch (Exception _){}
  }
}
```

Soweit die Klasse zur Generierung von Quelltext für eine algebraische Typspezifikation.

4.2.2 Konstruktordarstellung

Die wesentlich komplexere Informationen zu einen algebraischen Typen enthalten die Konstruktoren. Hierfür schreiben wir eine eigene Klasse:

Constructor.java
```
package name.panitz.crempel.util.adt;

import name.panitz.crempel.util.Tuple2;

import java.util.List;
import java.util.ArrayList;
import java.io.FileWriter;
```



```
import java.io.Writer;
import java.io.IOException;

public class Constructor {
```

Ein Konstruktor zeichnet sich durch eine Liste von Parametern aus. Wir beschreiben Parameter über ein Paar aus einem Typen und dem Parameternamen.

Constructor.java
```
  String name;
  List<Tuple2<Type,String>> params;
```

Wir sehen einen Konstruktor zur Initialisierung vor:

Constructor.java
```
  public Constructor(String n,List ps){name=n;params=ps;}
  public Constructor(String n){this(n,new ArrayList());}
```

Auch in dieser Klasse soll die Methode `toString` so überschrieben sein, daß die ursprüngliche textuelle Darstellung wieder vorhanden ist.

Constructor.java
```
  public String toString(){
    StringBuffer result = new StringBuffer("  ");
    result.append(name);
    result.append("(");
    boolean first = true;
    for (Tuple2<Type,String> t:params){
        if (first) {first=false;}
        else {result.append(",");}
      result.append(t.e1+"\u0020"+t.e2);
    }
    result.append(");\n");
    return result.toString();
  }
```

Wir kommen zur Generierung des Quelltextes für eine Javaklasse.

Die Argumente sind:

- das Objekt, das den algebraischen Typen darstellt
- und der Pfad zum Order des Dateisystems.

Die Methode unterscheidet, ob es sich um eine einzigen Konstruktor handelt, oder ob der algebraische Typ mehr als einen Konstruktor definiert hat. Hierzu gibt es ein bool'schen Parameter, der beim Fehlen standardmäßig auf `false` gesetzt wird.

Constructor.java
```
  public void generateClass(String path,AbstractDataType theType){
    generateClass(path,theType,false);
  }
```

Wir generieren die Klasse für den Konstruktor. Ob von einer Klasse abzuleiten ist und welche Schnittstelle zu implementieren ist, hängt davon ab, ob es noch weitere Konstruktoren gibt:

Constructor.java

```java
public void generateClass
      (String path,AbstractDataType theType,boolean standalone){
  try{
    if (standalone) name=theType.getFullName();
    FileWriter out = new FileWriter(path+"/"+name+".java");
    out.write( theType.getPackageDef());
    theType.writeImports(out);

    out.write("public class ");
    out.write(name);
    out.write(theType.getParamList());
    if (!standalone){
      out.write(" extends ");
      out.write(theType.getFullName());
    } else{
      out.write(" implements "+theType.name+"Visitable");
    }
    out.write("{\n");
```

Im Rumpf der Klasse sind zu generieren:

- Felder für die Argumente des Konstruktors
- `Get`-Methoden für diese Felder
- die Methode `visit`
- die Methode `equals`
- die Methode `toString`

Hierfür haben wir eigene Methoden entworfen:

Constructor.java

```java
    writeFields(out);
    writeConstructor(out);
    writeGetterMethods( out);
    writeVisitMethod(theType, out);
    writeToStringMethod(out);
    writeEqualsMethod(out);
    out.write("}\n");
    out.close();
  }catch (Exception _){}
}
```


Felder Wir generieren für jedes Argument des Konstruktors ein privates Feld:

Constructor.java

```java
private void writeFields(Writer out)throws IOException{
  for (Tuple2<Type,String> pair:params){
   out.write("  private final ");
   out.write(pair.e1.toString());
   out.write("\u0020");
   out.write(pair.e2);
   out.write(";\n");
  }
}
```

Kostruktor Wir generieren einen Konstrukor, der die privaten Felder initialisiert.

Constructor.java

```java
private void writeConstructor(Writer out)throws IOException{
  out.write("\n  public "+name+"(");
  boolean first= true;
  for (Tuple2<Type,String> pair:params){
    if (!first){out.write(",");}
    out.write(pair.e1.toString());
    out.write("\u0020");
    out.write(pair.e2);
    first=false;
  }
  out.write("){\n");
  for (Tuple2<Type,String> pair:params){
    out.write("    this."+pair.e2);
    out.write(" = ");
    out.write(pair.e2);
    out.write(";\n");
  }
  out.write("  }\n\n");
}
```

Get-Methoden Für jedes Feld wird eine öffentliche Get-Methode generiert:

Constructor.java

```java
private void writeGetterMethods(Writer out)throws IOException{
  for (Tuple2<Type,String> pair:params){
    out.write("  public ");
    out.write(pair.e1.toString());
    out.write(" get");
    out.write(Character.toUpperCase(pair.e2.charAt(0)));
    out.write(pair.e2.substring(1));
    out.write("(){return "+pair.e2 +";}\n");
  }
}
```

visit Die generierte Methode `visit` ruft die Methode `eval` des Besucherobjekts auf:

Constructor.java
```
  private void writeVisitMethod
                  (AbstractDataType theType,Writer out)
         throws IOException{
      out.write("  public <_b> _b visit("
               +theType.name+"Visitor<"+theType.commaSepParams()
                               +(theType.typeVars.isEmpty()?"":",")
                               +"_b> visitor){"
               +"\n    return visitor.eval(this);\n  }\n");
  }
```

toString Die generierte Methode `toString` erzeugt eine Zeile für den Konstruktor in der algebraischen Typspezifikation.

Constructor.java
```
  private void writeToStringMethod(Writer out) throws IOException{
    out.write("  public String toString(){\n");
    out.write("    return \""+name+"(\"");
    boolean first=true;
    for (Tuple2<Type,String> p:params){
     if (first){first=false;}
     else out.write("+\",\"");
     out.write("+"+p.e2);
    }
    out.write("+\")\";\n  }\n");
  }
```

equals Die generierte Methode `equals` vergleicht zunächst die Instanzen nach ihrem Typ und vergleicht anschließend Feldweise:

Constructor.java
```
  private void writeEqualsMethod(Writer out) throws IOException{
    out.write("  public boolean equals(Object other){\n");
    out.write("    if (!(other instanceof "+name+")) ");
    out.write("return false;\n");
    out.write("    final "+name+" o= ("+name+") other;\n");
    out.write("    return true  ");
    for (Tuple2<Type,String> p:params){
      out.write("&& "+p.e2+".equals(o."+p.e2+")");
    }
    out.write(";\n  }\n");
  }
```

die Eval-Methode In dem abstrakten Besucher für die algebraische Typspezifikation findet sich für jeden Konstruktor eine Methode `eval`, die mit folgender Methode generiert werden kann.

Constructor.java
```
  public String makeEvalMethod(AbstractDataType theType){
    return "public abstract result eval("
               +name+theType.getParamList()+" _);";
  }
}
```

4.2.3 Parametertypen

Für die Typen der Parameter eines Konstruktors haben wir eine kleine Klasse benutzt, in der die Typnamen und die Typparameter getrennt abgelegt sind.

Type.java
```
package name.panitz.crempel.util.adt;

import java.util.List;
import java.util.ArrayList;

public class Type {

    private String name;
    private List<Type> params;
    public String getName(){return name;}
    public List<Type> getParams(){return params;}

    public Type(String n,List ps){name=n;params=ps;}
    public Type(String n){this(n,new ArrayList());}

    public String toString(){
      StringBuffer result = new StringBuffer(name);
      if (!params.isEmpty()){
        result.append("<");
        boolean first=true;
        for (Type t:params){
          if (!first) result.append(',');
          result.append(t);
          first=false;
        }
        result.append('>');
      }
      return result.toString();
    }
}
```

4.2.4 Hauptgenerierungsprogramm

Damit sind wir mit dem kleinem Generierungsprogrammm fertig. Es bleibt nur, eine Hauptmethode vorzusehen, mit der für algebraische Typspezifikationen die entsprechenden Javaklassen generiert werden können. Algebraische Typspezifikationen seien in Dateien mit der Endung `.adt` gespeichert.

ADTMain.java
```java
package name.panitz.crempel.util.adt;

import name.panitz.crempel.util.adt.parser.ADT;
import java.util.List;
import java.util.ArrayList;
import java.io.FileReader;
import java.io.File;

public class ADTMain {
  public static void main(String args[])  {
    try{
      List<String> fileNames = new ArrayList<String>();

      if (args.length==1 && args[0].equals("*.adt")){
        for (String arg:new File(".").list()){
          if (arg.endsWith(".adt")) fileNames.add(arg);
        }
      }else for (String arg:args) fileNames.add(arg);

      for (String arg:fileNames){
        File f = new File(arg);
        ADT parser = new ADT(new FileReader(f));
        AbstractDataType adt = parser.adt();
        System.out.println(adt);
        final String path
          = f.getParentFile()==null?".":f.getParentFile().getPath();
        adt.generateClasses(path);
      }
    }catch (Exception _){_.printStackTrace();}
  }
}
```

5 Beispiel: eine kleine imperative Programmiersprache

Wir haben in den letzten Kapiteln ein relativ mächtiges Instrumentarium entwickelt. Nun wollen wir einmal sehen, ob algebraische Klassen mit Besuchern tatsächlich die Programmierarbeit erleichtern und Programme übersichtlicher machen.

Übersetzer von Komputerprogrammen sind ein Gebiet, in dem algebraische Typen gut angewendet werden können. Ein algebraischer Typ stellt den Programmtext als hierarchische Baumstruktur da. Die Verschiedenen Übersetzerschritte lassen sich als Besucher realisieren. Auch der Javaübersetzer `javac` ist mit dieser Technik umgesetzt worden.

5.1 Algebraischer Typ für Klip

Als Beispiel schreiben wir einen einfachen Interpreter für eine kleine imperative Programmiersprache, fortan *Klip* bezeichnet. In Klip soll es eine Arithmetik auf ganzen Zahlen geben, Zuweisung auf Variablen sowie ein Schleifenkonstrukt. Wir entwerfen einen algebraischen Typen für alle vorgesehenen Konstrukte von Klip.

Klip.adt
```java
package name.panitz.crempel.util.adt.examples;

import java.util.List;

data class Klip{
  Num(Integer i);
  Add(Klip e1,Klip e2);
  Mult(Klip e1,Klip e2);
  Sub(Klip e1,Klip e2);
  Div(Klip e1,Klip e2);
  Var(String name);
  Assign(String var,Klip e);
  While(Klip cond,Klip body);
  Block(List stats);
}
```

Wir sehen als Befehle arithmetische Ausdrücke mit den vier Grundrechenarten und Zahlen und Variablen als Operanden vor. Ein Zuweisungsbefehl, eine Schleife und eine Sequenz von Befehlen.

5.2 Besucher zur textuellen Darstellung

Als erste Funktion über den Datentyp `Klip` schreiben wir einen Besucher, der eine textuelle Repräsentation für den Datentyp erzeugt. Hierbei soll Zuweisungsoperator `:=` benutzt werden. Ansonsten sei die Syntax sehr ähnlich zu Java. Befehle einer Sequenz enden jeweils mit einem Semikolon, die Operatoren sind in Infixschreibweise und die While-Schleife hat die aus C und Java bekannte Syntax. Arithmetische Ausdrücke können geklammert sein.

Beispiel:
Ein kleines Beispiel für ein Klipprogramm zur Berechnung der Fakultät von 5.

fak.klip
```
x := 5;
y:=1;
while (x){y:=y*x;x:=x-1;};
y;
```

Den entsprechenden Besucher zu schreiben ist eine triviale Aufgabe.

ShowKlip.java
```java
package name.panitz.crempel.util.adt.examples;
import name.panitz.crempel.util.*;
```

```
import java.util.*;

public class ShowKlip extends KlipVisitor<String> {
  public String show(Klip a){return a.visit(this);}

  public String eval(Num x){return x.getI().toString();}
  public String eval(Add x){
    return "("+show(x.getE1())+" + "+show(x.getE2())+")";}
  public String eval(Sub x){
    return "("+show(x.getE1())+" - "+show(x.getE2())+")";}
  public String eval(Div x){
    return "("+show(x.getE1())+" / "+show(x.getE2())+")";}
  public String eval(Mult x){
    return "("+show(x.getE1())+" * "+show(x.getE2())+")";}
  public String eval(Var v){return v.getName();}
  public String eval(Assign x){
    return x.getVar()+" := "+show(x.getE());}
  public String eval(Block b){
    StringBuffer result=new StringBuffer();
    for (Klip x:(List<Klip>)b.getStats())
      result.append(show(x)+";\n");
    return result.toString();
  }
  public String eval(While w){
    StringBuffer result=new StringBuffer("while (");
    result.append(show(w.getCond()));
    result.append("){\n");
    result.append(show(w.getBody()));
    result.append("\n}");
    return result.toString();
  }
}
```

5.3 Besucher zur Interpretation eines Klip Programms

Jetzt wollen wir Klip Programme auch ausführen. Auch hierzu schreiben wir eine Besucherklasse, die einmal alle Knoten eines Klip-Programms besucht. Hierbei wird direkt das Ergebnis des Programms berechnet. Um darüber Buch zu führen, welcher Wert in den einzelnen Variablen gespeichert ist, enthält der Besucher eine Abbildung von Variablennamen auf ganzzahlige Werte. Ansonsten ist die Auswertung ohne große Tricks umgesetzt. Alle Werte ungleich 0 werden als wahr interpretiert.

EvalKlip.java

```
package name.panitz.crempel.util.adt.examples;
import name.panitz.crempel.util.*;
import java.util.*;

public class EvalKlip extends KlipVisitor<Integer> {
  Map<String,Integer> env = new HashMap<String,Integer>();
```



```
  public Integer val(Klip x){return x.visit(this);}

 public Integer eval(Num x){return x.getI();}
 public Integer eval(Add x){return val(x.getE1())+val(x.getE2());}
 public Integer eval(Sub x){return val(x.getE1())-val(x.getE2());}
 public Integer eval(Div x){return val(x.getE1())/val(x.getE2());}
 public Integer eval(Mult x){return val(x.getE1())*val(x.getE2());}
 public Integer eval(Var v){return env.get(v.getName());}
 public Integer eval(Assign ass){
   Integer i = val(ass.getE());
   env.put(ass.getVar(),i);
   return i;
 }
 public Integer eval(Block b){
    Integer result = 0;
    for (Klip x:(List<Klip>)b.getStats()) result=val(x);
    return result;
 }
 public Integer eval(While w){
    Integer result = 0;
    while (w.getCond().visit(this)!=0){
      System.out.println(env); //this is a trace output
      result = w.getBody().visit(this);
    }
    return result;
 }
}
```

Soweit unsere zwei Besucher. Es lassen sich beliebige weitere Besucher schreiben. Eine interessante Aufgabe wäre zum Beispiel ein Besucher, der ein Assemblerprogramm für ein Klipprogramm erzeugt.

5.4 javacc Parser für Klip

Schließlich, um Klipprogramme ausführen zu können, benötigen wir einen Parser, der die textuelle Darstellung eines Klipprogramms in die Baumstruktur umwandelt. Wir schreiben einen solchen Parser mit Hilfe des Parsergenerators `javacc`.

Der Parser soll zunächst eine Hauptmethode enthalten, die ein Klipprogramm parst und die beiden Besucher auf ihn anwendet:

KlipParser.jj

```
options {
   STATIC=false;
}

PARSER_BEGIN(KlipParser)
package name.panitz.crempel.util.adt.examples;

import name.panitz.crempel.util.Tuple2;
```

```
import java.util.List;
import java.util.ArrayList;
import java.io.FileReader;

public class KlipParser  {
  public static void main(String [] args)throws Exception{
    Klip klip = new KlipParser(new FileReader(args[0]))
                .statementList();

    System.out.println(klip.visit(new ShowKlip()));
    System.out.println(klip.visit(new EvalKlip()));
  }
}
PARSER_END(KlipParser)
```

5.4.1 Scanner

In einer `javacc`-Grammatik wird zunächst die Menge der Terminalsymbole spezifiziert.

KlipParser.jj

```
TOKEN :
{<WHILE: "while">
|<#ALPHA:        ["a"-"z","A"-"Z","_","."]        >
|<NUM:                ["0"-"9"]                >
|<#ALPHANUM:        <ALPHA> | <NUM>                >
|<NAME: <ALPHA> ( <ALPHANUM> )*>
|<ASS: ":=">
|<LPAR: "(">
|<RPAR: ")">
|<LBRACKET: "{">
|<RBRACKET: "}">
|<SEMICOLON: ";">
|<STAR: "*">
|<PLUS: "+">
|<SUB: "-">
|<DIV: "/">
}
```

Zusätzlich läßt sich spezifizieren, welche Zeichen als Leerzeichen anzusehen sind:

KlipParser.jj

```
SKIP :
{ "\u0020"
| "\t"
| "\n"
| "\r"
}
```


5.4.2 Parser

Es folgen die Regeln der Klip-Grammatik. Ein Klip Programm ist zunächst eine Sequenz von Befehlen:

KlipParser.jj

```
Klip statementList() :
{  List stats = new ArrayList();
   Klip stat;}
{
  (stat=statement() {stats.add(stat);} <SEMICOLON>)*
  {return new Block(stats);}
}
```

Ein Befehl kann zunächst ein arithmetischer Ausdruck in Punktrechnung sein.

KlipParser.jj

```
Klip statement():
{Klip e2;Klip result;boolean sub=false;}
{
  result=multExpr()
  [ (<PLUS>|<SUB>{sub=true;}) e2=statement()
   {result = sub?new Sub(result,e2):new Add(result,e2);}]
  {return result;}
}
```

Die Operanden der Punktrechnung sind arithmetische Ausdruck in Strichrechnung. Auf diese Weise realisiert der Parser einen Klip-Baum, in dem Punktrechnung stärker bindet als Strichrechnung.

KlipParser.jj

```
Klip multExpr():
{Klip e2;Klip result;boolean div= false;}
{
  result=atomicExpr()
  [ (<STAR>|<DIV>{div=true;})
    e2=multExpr()
    {result = div?new Div(result,e2):new Mult(result,e2);}]
  {return result;}
}
```

Die Operanden der Punktrechnung sind entweder Literale, Variablen, Zuweisungen, Schleifen oder geklammerte Ausdrücke.

KlipParser.jj

```
Klip atomicExpr():
{Klip result;}
{
   (result=integerLiteral()
   |result=varOrAssign()
```

```
    |result=whileStat()
    |result=parenthesesExpr()
   )
  {return result;}
}
```

Ein Literal ist eine Sequenz von Ziffern.

KlipParser.jj
```
Klip integerLiteral():
{ int result = 0;
  Token n;
  boolean m=false;}
{  [<SUB> {m = true;}]
   (n=<NUM>
    {result=result*10+n.toString().charAt(0)-48;})+
   {return new Num(new Integer(m?-result:result));}
}
```

Geklammerte Ausdrücke klammern beliebige Befehle.

KlipParser.jj
```
Klip parenthesesExpr():
{Klip result;}
{ <LPAR> result = statement() <RPAR>
{return result;}}
```

Variablen können einzeln oder auf der linken Seite einer Zuweisung auftreten.

KlipParser.jj
```
Klip varOrAssign():
{ Token n;Klip result;Klip stat;}
{ n=<NAME>{result=new Var(n.toString());}
  [<ASS> stat=statement()
    {result = new Assign(n.toString(),stat);}
  ]
  {return result;}
}
```

Und schließlich noch die Regel für die `while`-Schleife.

KlipParser.jj
```
Klip whileStat():{
  Klip cond; Klip body;}
{ <WHILE> <LPAR>cond=statement()<RPAR>
  <LBRACKET> body=statementList()<RBRACKET>
  {return new While(cond,body);}
}
```


5.4.3 Klip-Beispiele

Unser Klip-Interpreter ist fertig. Wir können Klip-Programme ausführen lassen.

Zunächst mal zwei Programme, die die Arithmetik demonstrieren:

arith1.klip
```
2*7+14*2;
```

arith2.klip
```
2*(7+9)*2;
```

```
sep@linux:~> java name.panitz.crempel.util.adt.examples.KlipParser arith1.klip
((2 * 7) + (14 * 2));

42
sep@linux:~> java name.panitz.crempel.util.adt.examples.KlipParser arith2.klip
(2 * ((7 + 9) * 2));

64
sep@linux:~>
```

Auch unser erstes Fakultätsprogramm in Klip läßt sich ausführen:

```
sep@linux:~> java name.panitz.crempel.util.adt.examples.KlipParser fak.klip
x := 5;
y := 1;
while (x){
y := (y * x);
x := (x - 1);

};
y;

{y=1, x=5}
{y=5, x=4}
{y=20, x=3}
{y=60, x=2}
{y=120, x=1}
120
sep@linux:~>
```

Wie man sieht bekommen wir auch eine Traceausgabe über die Umgebung während der Auswertung.

A Javacc Definition für ATD Parser

Es folgt in diesem Abschnitt unkommentiert die `javacc` Grammatik für algebraische Datentypen. Die Grammatik ist absichtlich sehr einfach gehalten. Unglücklicher Weise weist `javacc` bisher noch Java 1.5 Syntax zurück, so daß die Übersetzung des entstehenden Parser Warnungen bezüglich nicht überprüfter generischer Typen gibt.

adt.jj

```
options {
   STATIC=false;
}

PARSER_BEGIN(ADT)
package name.panitz.crempel.util.adt.parser;

import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.FileReader;

public class ADT {

}
PARSER_END(ADT)

TOKEN :
{<DATA: "data">
|<CLASS: "class">
|<DOT: ".">
|<#ALPHA:        ["a"-"z","A"-"Z","_","."]        >
|<#NUM:                ["0"-"9"]                >
|<#ALPHANUM:        <ALPHA> | <NUM>                >
|<PAKET: "package">
|<IMPORT: "import">
|<NAME: <ALPHA> ( <ALPHANUM> )*>
|<EQ: "=">
|<BAR: "|">
|<LPAR: "(">
|<RPAR: ")">
|<LBRACKET: "{">
|<RBRACKET: "}">
|<LE: "<">
|<GE: ">">
|<SEMICOLON: ";">
|<COMMA: ",">
|<STAR: "*">
}

SKIP :
{ "\u0020"
| "\t"
| "\n"
| "\r"
}

AbstractDataType adt() :
```



```
{ Token nameT;
  String name;
  String paket;
  List typeVars = new ArrayList();
  List constructors;
  List imports;
}
{
   paket=packageDef()
   imports=importDefs()

   <DATA> <CLASS> nameT=<NAME>{name=nameT.toString();}

   [ <LE>
       (nameT=<NAME> {typeVars.add(nameT.toString());})
       (<COMMA> nameT=<NAME> {typeVars.add(nameT.toString());})*
     <GE>]
    <LBRACKET>
   constructors=defs()
    <RBRACKET>
{return
  new AbstractDataType(paket,name,typeVars,constructors,imports);}
}

String packageDef():
{StringBuffer result=new StringBuffer();Token n;}
{
  [<PAKET> n=<NAME>{result.append(n.toString());}
    (<DOT> n=<NAME>{result.append("."+n.toString());})*
    <SEMICOLON>
  ]
{return result.toString();}
}


List importDefs():{
  List result=new ArrayList();Token n;StringBuffer current;}
{
  ({current = new StringBuffer();}
    <IMPORT> n=<NAME>{current.append(n.toString());}
    (<DOT> n=<NAME>{current.append("."+n.toString());})*
    [<DOT><STAR>{current.append(".*");}]
    <SEMICOLON>
    {current.append(";");result.add(current.toString());}
  )*
{return result;}
}


List defs() :
```

```
{
  Constructor def ;
  ArrayList result=new ArrayList();
}
{
   def=def(){result.add(def);} (def=def() {result.add(def);} )*
   {return result;}
}

Constructor def() :
{ Token n;
  Type param ;
  String name;
  ArrayList params=new ArrayList();
}
{
  n=<NAME> {name=n.toString();}
     <LPAR>[(param=type() n=<NAME>
                {params.add(new Tuple2(param,n.toString()));}  )
         (<COMMA> param=type() n=<NAME>
            {params.add(new Tuple2(param,n.toString()));}  )*
     ]<RPAR>
  <SEMICOLON>
  {return new Constructor(name,params);}
}

Type type():
{ Type result;
  Token n;
  Type typeParam;
  ArrayList params=new ArrayList();
}
{
  ( n=<NAME>
   ([<LE>
        typeParam=type() {params.add(typeParam);}
       (<COMMA> typeParam=type() {params.add(typeParam);}   )*
       {result = new Type(n.toString(),params);}
     <GE>])
 )
{
{result = new Type(n.toString(),params);}
return result;
}
}
```


B Aufgaben

Aufgabe 1 Gegeben sei folgende algebraische Datenstruktur[2].

HLi.hs

```
data HLi a
  =   Empty
    | Cons a (HLi a)
```

Auf dieser Struktur sei die Methode `odds` durch folgende Gleichungen spezifiziert:

HLi.hs

```

odds(Cons x (Cons y ys)) = (Cons x (odds(ys)))
odds(Cons x Empty)       = (Cons x Empty)
odds(Empty)              = Empty
```

Reduzieren Sie schrittweise den Ausdruck:
`odds(Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Empty)))))`

Lösung

```
    odds(Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 Empty)))))
→   Cons 1 (odds(Cons 3 (Cons 4 (Cons 5 Empty))))
→   Cons 1 (Cons 3 (odds(Cons 5 Empty)))
→   Cons 1 (Cons 3 (Cons 5 Empty))
```

Aufgabe 2 Gegeben sei folgende algebraische Datenstruktur.

HBT.hs

```
data HBT a
  =   T a
    | E
    | B (HBT a) a (HBT a)
```

Auf dieser Struktur sei die Methode `addLeft` durch folgende Gleichungen spezifiziert:

HBT.hs

```

addLeft (T a)     = a
addLeft (E)       = 0
addLeft (B l x r) = x+addLeft(l)
```

[2] in Haskellnotation

Reduzieren Sie schrittweise den Ausdruck:
`addLeft(Branch(Branch (Branch (T 4) 3 (E)) 2 E) 1 (T 2))`

Lösung

```
   addLeft(Branch(Branch (Branch (T 4) 3 (E)) 2 E) 1 (T 2))
→ 1+addLeft(Branch (Branch (T 4) 3 (E)) 2 E)
→ 1+2+addLeft(Branch (T 4) 3 (E)))
→ 1+2+3+addLeft(T 4)
→ 1+2+3+4
→ 10
```

Aufgabe 3 Gegeben sei folgende algebraische Typspezifikation für Binärbäume:

BT.adt
```
package name.panitz.aufgaben;
data class BT<at>{
  E();
  Branch(BT<at> left,at mark,BT<at> right);
}
```

a) Schreiben Sie einen Besucher `HTMLTree<at>`, der `BTVisitor<at,StringBuffer>` erweitert. Er soll in einem StringBuffer einen String erzeugen, der HTML-Code darstellt. Die Kinder eines Knotens sollen dabei mit einem `<ul>`-Tag gruppiert werden und jedes Kind als `<li>` Eintrag in dieser Gruppe auftreten.

Beispiel: Für folgenden Baum:

HTMLTreeExample.java
```
package name.panitz.aufgaben;
public class HTMLTreeExample{
  public static void main(String [] _){
    BT<String> bt
     =new Branch<String>
       (new Branch<String>(new E<String>()
                          ,"brecht"
                          ,new E<String>())
       ,"horvath"
       ,new Branch<String>
         (new Branch<String>(new E<String>()
                            ,"ionesco"
                            ,new E<String>())
         ,"shakespeare"
         ,new E<String>()));
    System.out.println(bt.visit(new HTMLTree()));
  }
}
```


Wird folgenden HTML Code erzeugt:

```
horvath
<ul>
<li>brecht
<ul>
<li>E()</li>
<li>E()</li></ul></li>
<li>shakespeare
<ul>
<li>ionesco
<ul>
<li>E()</li>
<li>E()</li></ul></li>
<li>E()</li></ul></li></ul>
```

Lösung

HTMLTree.java
```
package name.panitz.aufgaben;
import java.util.List;
import java.util.ArrayList;
public class HTMLTree<at> extends  BTVisitor<at,StringBuffer>{
  StringBuffer result=new StringBuffer();
  public StringBuffer eval(E<at> _){
    result.append("E()");return result;}

  public StringBuffer eval(Branch<at> n){
    result.append(n.getMark());
    result.append("\n<ul>");
    result.append("\n<li>");
    n.getLeft().visit(this);
    result.append("</li>");
    result.append("\n<li>");
    n.getRight().visit(this);
    result.append("</li>");
    result.append("</ul>");
    return result;
  }
}
```

b) Schreiben Sie einen Besucher `FlattenTree<at>`, der `BTVisitor<at,List<at>>` erweitert. Er soll alle Knotenmarkierungen des Baumes in seiner Ergebnisliste sammeln.

Lösung

FlattenTree.java
```
package name.panitz.aufgaben;
import java.util.List;
import java.util.ArrayList;
```

```
public class FlattenTree<at> extends  BTVisitor<at,List<at>>{
  List<at> result = new ArrayList<at>();

  public List<at> eval(E<at> _){return result;}
  public List<at> eval(Branch<at> n){
    n.getLeft().visit(this);
    result.add(n.getMark());
    n.getRight().visit(this);
    return result;
  }
}
```

Literatur

[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.

[Ler97] Xavier Leroy. The Caml Light system release 0.73. Institut National de Recherche en Informatique et Automatique, 1 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J.Comp.Sys.Sci*, 17:348–375, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan03] Sven Eric Panitz. Programmieren II. Skript zur Vorlesung, TFH Berlin, 2003. `www.panitz.name/prog2/index.html`.

[Pan04] Sven Eric Panitz. Erweiterungen in Java 1.5. Skript zur Vorlesung, TFH Berlin, 2004. `www.panitz.name/java1.5/index.html`.

[PvE95] R. Plasmeijer and M. van Eekelen. Concurrent clean: Version 1.0. Technical report, Dept. of Computer Science, University of Nijmegen, 1995. draft.