Programmieren IV
(Entwurf)

SS 04

Sven Eric Panitz

TFH Berlin

Version 24. Juni 2004

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 SS 04 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.

Die Vorlesung setzt die Kerninhalte der Vorgängervorlesungen [Pan03a], [Pan03b] und [Pan03c] voraus.

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. Der XML-Quelltext verweist direkt auf ein XSLT-Skript, das eine HTML-Darstellung erzeugt, so daß ein entsprechender Browser mit XSLT-Prozessor die XML-Datei direkt als HTML-Seite darstellen kann. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Einführung

Ziel der Vorlesung

Mit den Grundtechniken der Programmierung am Beispiel von Java wurde in der Vorgängervorlesungen vertraut gemacht. Im ersten Semester haben wir die Grundzüge der Programmierung anhand von Java erlernt, im zweiten Semester den Umgang mit Javas Bibliotheken. Nach unserem Exkurs in die Welt von C++ im dritten Semester wenden wir uns jetzt wieder Java zu. Ziel ist es komplexe Anwendungsbeispiele zu entwerfen. Hierbei kümmern wir uns nicht um die softwaretechnische Modellierung einer solchen Anwendung, sondern werden algorithmische Tricks und Entwurfsmuster im Vordergrund stellen. Hierbei wird uns ab und an ein Blick über den Tellerrand zu anderen Programmiersprachen helfen. Als Beispielanwendungsfall wird im Zentrum die Programmierung von XML-Strukturen stehen, da an der baumartigen Struktur von XML sich viele allgemeine Konzepte der Informatik durchspielen lassen.

Kapitel 1

Erweiterungen in Java 1.5

1.1 Einführung

Mit Java 1.5 werden einige neue Konzepte in Java eingeführt, die viele Programmierer bisher schmerzlich vermisst haben. Obwohl die ursprünglichen Designer über diese neuen Konstrukte von Anfang an nachgedacht haben und sie auch gerne in die Sprache integriert hätten, schaffen diese Konstrukte es erst jetzt nach vielen Jahren in die Sprache Java. Hierfür kann man zwei große Gründe angeben:

- Beim Entwurf und der Entwicklung von Java waren die Entwickler bei Sun unter Zeitdruck.
- Die Sprache Java wurde in ihren programmiersprachlichen Konstrukten sehr konservativ entworfen. Die Syntax wurde von C übernommen. Die Sprache sollte möglichst wenige aber mächtige Eigenschaften haben und kein Sammelsurium verschiedenster Techniken sein.

Seitdem Java eine solch starke Bedeutung als Programmiersprache erlangt hat, gibt es einen definierten Prozess, wie neue Eigenschaften der Sprache hinzugefügt werden, den *Java community process (JPC)*. Hier können fundierte Verbesserungs- und Erweiterungsvorschläge gemacht werden. Wenn solche von genügend Leuten unterstützt werden, kann ein sogenannter *Java specification request (JSR)* aufgemacht werden. Hier wird eine Expertenrunde gegründet, die die Spezifikation und einen Prototypen für die Javaerweiterung erstellt. Die Spezifikation und prototypische Implementierung werden schließlich öffentlich zur Diskussion gestellt. Wenn es keine Einwände von irgendwelcher Seite mehr gibt, wird die neue Eigenschaft in eine der nächsten Javaversionen integriert.

Mit Java 1.5 findet das Ergebnis einer Vielzahl JSRs Einzug in die Sprache. Die Programmiersprache wird in einer Weise erweitert, wie es schon lange nicht mehr der Fall war. Den größten Einschnitt stellt sicherlich die Erweiterung des Typsystems auf generische Typen dar. Aber auch die anderen neuen Kosntrukte, die verbesserte `for`-Schleife, Aufzählungstypen, statisches Importieren und automatisches Boxen, sind keine esoterischen Eigenschaften, sondern werden das alltägliche Arbeiten mit Java beeinflussen. In den folgenden Abschnitten werfen wir einen Blick auf die neuen Javaeigenschaften im Einzelnen.

1.2 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert[Pan00]. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun bald vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza [OW97] existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

1.2.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muß daher den Typ `Object` erhalten:

OldBox.java

```
class OldBox {
  Object contents;
  OldBox(Object contents){this.contents=contents;}
}
```

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

```
class UseOldBox{
  public static void main(String [] _){
    OldBox b = new OldBox("hello");
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
    System.out.println(((String) s).toUpperCase());
  }
}
```

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

UseOldBoxError.java
```java
class UseOldBoxError{
  public static void main(String [] _){
    OldBox b = new OldBox(new Integer(42));
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, daß diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

Box.java
```java
class Box<elementType> {
  elementType contents;
  Box(elementType contents){this.contents=contents;}
}
```

Die Typvariable `elementType` ist als allquantifiziert zu verstehen. Für jeden Typ `elementType` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansonsten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java
```java
class UseBox{
  public static void main(String [] _){
    Box<String> b1 = new Box<String>("hello");
    String s = b1.contents;
    System.out.println(s.toUpperCase());
    System.out.println(b1.contents.toUpperCase());

    Box<Integer> b2 = new Box<Integer>(new Integer(42));

    System.out.println(b2.contents.intValue());
  }
}
```

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

```java
class UseBoxError{
  public static void main(String [] _){
    Box<String> b = new Box<String>(new Integer(42));
    String s = b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                    ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so daß wir zwei Objekte speichern können:

Pair.java
```
class Pair<at,bt> extends Box<at>{
  Pair(at x,bt y){
    super(x);
    snd = y;
  }

  bt snd;

  public String toString(){
    return "("+contents+","+snd+")";
  }
}
```

Die Klasse `Pair` hat zwei Typvariablen. Instanzen von `Pair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

UsePair.java
```
class UsePair{
  public static void main(String [] _){
    Pair<String,Integer> p
     = new Pair<String,Integer>("hallo",new  Integer(40));

    System.out.println(p);
    System.out.println(p.contents.toUpperCase());
    System.out.println(p.snd.intValue()+2);
  }
}
```

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

UniPair.java
```
class UniPair<at> extends Pair<at,at>{
  UniPair(at x,at y){super(x,y);}
  void swap(){
    final at z = snd;
    snd = contents;
    contents = z;
  }
}
```


Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

UseUniPair.java
```
class UseUniPair{
  public static void main(String [] _){
    UniPair<String> p
     = new UniPair<String>("welt","hallo");

    System.out.println(p);
    p.swap();
    System.out.println(p);
  }
}
```

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java
```
class StringBox extends Box<String>{
  StringBox(String x){super(x);}
}
```

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java
```
class UseStringBox{
  public static void main(String [] _){
    StringBox b = new StringBox("hallo");
    System.out.println(b.contents.length());
  }
}
```

Einschränken der Typvariablen

Bisher standen in allen Beispielen die Typvariablen einer generischen Klasse für jeden beliebigen Objekttypen. Hier erlaubt Java uns, Einschränkungen zu machen. Es kann eingeschränkt werden, daß eine Typvariable nicht für alle Typen ersetzt werden darf, sondern nur für bestimmte Typen.

Versuchen wir einmal, eine Klasse zu schreiben, die auch wieder der Klasse `Box` entspricht, zusätzlich aber eine `set`-Methode hat und nur den neuen Wert in das entsprechende Objekt speichert, wenn es größer ist als das bereits gespeicherte Objekt. Hierzu müssen die zu speichernden Objekte in einer Ordnungsrelation vergleichbar sein, was in Java über die Implementierung der Schnittstelle `Comparable` ausgedrückt wird. Im herkömmlichen Java würden wir die Klasse wie folgt schreiben:

CollectMaxOld.java
```java
class CollectMaxOld{
  private Comparable value;

  CollectMaxOld(Comparable x){value=x;}

  void setValue(Comparable x){
    if (value.compareTo(x)<0) value=x;
  }

  Comparable getValue(){return value;}
}
```

Die Klasse `CollectMaxOld` ist in der Lage, beliebige Objekte, die die Schnittstelle `Comparable` implementieren, zu speichern. Wir haben wieder dasselbe Problem wie in der Klasse `OldBox`: Greifen wir auf das gespeicherte Objekt mit der Methode `getValue` erneut zu, wissen wir nicht mehr den genauen Typ dieses Objekts und müssen eventuell eine dynamische Typzusicherung durchführen, die zu Laufzeitfehlern führen kann.

Javas generische Typen können dieses Problem beheben. In gleicher Weise, wie wir die Klasse `Box` aus der Klasse `OldBox` erhalten haben, indem wir den allgemeinen Typ `Object` durch eine Typvariable ersetzt haben, ersetzen wir jetzt den Typ `Comparable` durch eine Typvariable, geben aber zusätzlich an, daß diese Variable für alle Typen steht, die die Untertypen der Schnittstelle `Comparable` sind. Dieses wird durch eine zusätzliche `extends`-Klausel für die Typvariable angegeben. Wir erhalten somit eine generische Klasse `CollectMax`:

CollectMax.java
```java
class CollectMax <elementType extends Comparable>{
  private elementType value;

  CollectMax(elementType x){value=x;}

  void setValue(elementType x){
    if (value.compareTo(x)<0) value=x;
  }

  elementType getValue(){return value;}
}
```

Für die Benutzung diese Klasse ist jetzt für jede konkrete Instanz der konkrete Typ des gespeicherten Objekts anzugeben. Die Methode `getValue` liefert als Rückgabetyp nicht ein allgemeines Objekt des Typs `Comparable`, sondern exakt ein Objekt des Instanztyps.

UseCollectMax.java
```java
class UseCollectMax {
  public static void main(String [] _){
    CollectMax<String> cm = new CollectMax<String>("Brecht");
    cm.setValue("Calderon");
    cm.setValue("Horvath");
    cm.setValue("Shakespeare");
    cm.setValue("Schimmelpfennig");
    System.out.println(cm.getValue().toUpperCase());
  }
}
```

Wie man in der letzten Zeile sieht, entfällt wieder die dynamische Typzusicherung.

1.2.2 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er ist gleichbedeutend damit, daß keine Information über einen konkreten Typ während der Übersetzungszeit zur Verfügung steht. Im herkömmlichen Java ist in APIs von Bibliotheken der Typ `Object` allgegenwärtig. Sogar in der Klasse `Object` selbst begegnet er uns in Signaturen. Die Methode `equals` hat einen Parameter vom Typ `Object`, d.h. prinzipiell kann ein Objekt mit Objekten jeden beliebigen Typs verglichen werden. Zumeist will man aber nur gleiche Typen miteinander vergleichen. In diesem Abschnitt werden wir sehen, daß generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objekte gleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleich zu generischen Klassen ist nichts Neues zu lernen. Syntax und Benutzung funktionieren auf die gleiche Weise.

Äpfel mit Birnen vergleichen

Um zu realisieren, daß nur noch Objekte gleichen Typs miteinander verglichen werden können, definieren wir eine Gleichheitsschnitstelle. In ihr wird eine Methode spezifiziert, die für die Gleichheit stehen soll. Die Schnittstelle ist generisch über den Typen, mit dem vergleichen werden soll.

EQ.java
```java
interface EQ<otherType> {
  public boolean eq(otherType other);
}
```

Jetzt können wir für jede Klasse nicht nur bestimmen, daß sie die Gleichheit implementieren soll, sondern auch, mit welchen Typen Objekte unserer Klasse verglichen werden sollen. Schreiben wir hierzu eine Klasse `Apfel`. Die Klasse Apfel soll die Gleichheit auf sich selbst implementieren. Wir wollen nur Äpfel mit Äpfeln vergleichen können. Daher definieren wir in der `implements`-Klausel, daß wir `EQ<Apfel>` implementieren wollen. Dann müssen wir auch die Methode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java
```java
class Apfel implements EQ<Apfel>{
  String typ;

  Apfel(String typ){
    this.typ=typ;}

  public boolean eq(Apfel other){
    return this.typ.equals(other.typ);
  }
}
```

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java
```java
class TestEq{
  public static void main(String []_){
    Apfel a1 = new Apfel("Golden Delicious");
    Apfel a2 = new Apfel("Macintosh");
    System.out.println(a1.eq(a2));
    System.out.println(a1.eq(a1));
  }
}
```

Schreiben wir als nächstes eine Klasse die Birnen darstellen soll. Auch diese implementiere die Schnittstelle `EQ`, und zwar dieses Mal für Birnen:

Birne.java
```java
class Birne implements EQ<Birne>{
  String typ;

  Birne(String typ){
    this.typ=typ;}

  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

Während des statischen Typchecks wird überprüft, ob wir nur Äpfel mit Äpfeln und Birnen mit Birnen vergleichen. Der Versuch, Äpfel mit Birnen zu vergleichen, führt zu einem Typfehler:

```java
class TesteEqError{
  public static void main(String []_){
    Apfel a = new Apfel("Golden Delicious");
    Birne b = new Birne("williams");
    System.out.println(a.equals(b));
    System.out.println(a.eq(b));
```



```java
  }
}
```

Wir bekommen die verständliche Fehlermeldung, daß die Gleichheit auf Äpfel nicht für einen Birnenparameter aufgerufen werden kann.

```
./TestEQError.java:6: eq(Apfel) in Apfel cannot be applied to (Birne)
    System.out.println(a.eq(b));
                        ^
1 error
```

Wahrscheinlich ist es jedem erfahrenden Javaprogrammierer schon einmal passiert, daß er zwei Objekte verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn die Methode `equals` läßt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, daß eine generische Schnittstelle mit der korrekten Signatur implementiert wird. Der Versuch, eine Birneklasse zu schreiben, die eine Gleichheit mit Äpfeln implementieren soll, dann aber die Methode `eq` mit dem Parametertyp `Birne` zu implementieren, führt ebenfalls zu einer Fehlermeldung:

```java
class BirneError implements EQ<Apfel>{
  String typ;

  BirneError(String typ){
    this.typ=typ;}

  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

Wir bekommen folgende Fehlermeldung:

```
sep@linux:~/fh/java1.5/examples/src> javac BirneError.java
BirneError.java:1: BirneError is not abstract and does not override abstract method eq(Apfel) in EQ
class BirneError implements EQ<Apfel>{
^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

1.2.3 Kovarianz gegen Kontravarianz

Gegeben seien zwei Typen `A` und `B`. Der Typ `A` soll Untertyp des Typs `B` sein, also entweder ist `A` eine Unterklasse der Klasse `B` oder `A` implementiert die Schnittstelle `B` oder die Schnittstelle `A` erweitert die Schnittstelle `B`. Für diese Subtyprelation schreiben wir das Relationssymbol $\sqsubseteq$. Es gelte also $A \sqsubseteq B$. Gilt damit auch für einen generischen Typ `C`: `C<A>`$\sqsubseteq$`C<B>`?

Man mag geneigt sein, zu sagen ja. Probieren wir dieses einmal aus:

```
class Kontra{
  public static void main(String []_){
    Box<Object> b  = new Box<String>("hello");
  }
}
```

Der Javaübersetzer weist dieses Programm zurück:

```
sep@linux:~/fh/java1.5/examples/src> javac Kontra.java
Kontra.java:4: incompatible types
found   : Box<java.lang.String>
required: Box<java.lang.Object>
    Box<Object> b  = new Box<String>("hello");
                     ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Eine `Box<String>` ist keine `Box<Object>`. Der Grund für diese Entwurfsentscheidung liegt darin, daß bestimmte Laufzeitfehler vermieden werden sollen. Betrachtet man ein Objekt des Typs `Box<String>` über eine Referenz des Typs `Box<Object>`, dann können in dem Feld `contents` beliebige Objekte gespeichert werden. Die Referenz über den Typ `Box<String>` geht aber davon aus, daß in `contents` nur Stringobjekte gespeichert werden.

Man vergegenwärtige sich nochmals, daß Reihungen in Java sich hier anders verhalten. Bei Reihungen ist die entsprechende Zuweisung erlaubt. Eine Reihung von Stringobjekten darf einer Reihung beliebiger Objekte zugewiesen werden. Dann kann es bei der Benutzung der Reihung von Objekten zu einen Laufzeitfehler kommen.

Ko.java

```
class Ko{
  public static void main(String []_){
    String [] x = {"hello"};
    Object [] b  = x;
    b[0]=new Integer(42);
    x[0].toUpperCase();
  }
}
```

Das obige Programm führt zu folgendem Laufzeitfehler:

```
sep@linux:~/fh/java1.5/examples/classes> java Ko
Exception in thread "main" java.lang.ArrayStoreException
        at Ko.main(Ko.java:5)
sep@linux:~/fh/java1.5/examples/classes>
```

Für generische Typen wurde ein solcher Fehler durch die Strenge des statischen Typchecks bereits ausgeschlossen.

1.2.4 Sammlungsklassen

Die Paradeanwendung für generische Typen sind natürlich Sammlungsklassen, also die Klassen für Listen und Mengen, wie sie im Paket `java.util` definiert sind. Mit der Version 1.5 von Java finden sich generische Versionen der bekannten Sammlungsklassen. Jetzt kann man angeben, was für einen Typ die Elemente einer Sammlung genau haben sollen.

ListTest.java

```
import java.util.*;
import java.util.List;
class ListTest{
  public static void main(String [] _){
    List<String> xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = xs.get(1);
    System.out.println(xs);
  }
}
```

Aus Kompatibilitätsgründen mit bestehendem Code können generische Klassen auch weiterhin ohne konkrete Angabe des Typparameters benutzt werden. Während der Übersetzung wird in diesen Fällen eine Warnung ausgegeben.

WarnList.java

```
import java.util.*;
import java.util.List;
class WarnTest{
  public static void main(String [] _){
    List xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = (String)xs.get(1);
    System.out.println(xs);
  }
}
```

Obiges Programm übersetzt mit folgender Warnung:

```
sep@linux:~/fh/java1.5/examples/src> javac WarnList.java
Note: WarnList.java uses unchecked or unsafe operations.
Note: Recompile with -warnunchecked for details.
sep@linux:~/fh/java1.5/examples/src>
```

1.2.5 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientierten Kontext gebunden, sondern basieren ganz im Gegenteil auf dem Milner-Typsystem, das funktionale Sprachen, die nicht objektorientiert sind, benutzen. In Java verläßt man den objektorientierten Kontext in statischen Methoden. Statische Methoden sind nicht an ein Objekt gebunden. Auch statische Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache statische generische Methode ist eine *trace*-Methode, die ein beliebiges Objekt erhält, dieses Objekt auf der Konsole ausgibt und als Ergebnis genau das erhaltene Objekt unverändert wieder zurückgibt. Diese Methode `trace` hat für alle Typen den gleichen Code und kann daher entsprechend generisch geschrieben werden:

Trace.java

```java
class Trace {

  static <elementType>  elementType trace(elementType x){
    System.out.println(x);
    return x;
  }

  public static void main(String [] _){
    String x = trace ((trace ("hallo")
                  +trace( " welt")).toUpperCase());

    Integer y = trace (new Integer(40+2));
  }
}
```

In diesem Beispiel ist zu erkennen, daß der Typchecker eine kleine Typinferenz vornimmt. Bei der Anwendung der Methode `trace` ist nicht anzugeben, mit welchen Typ die Typvariable `elementType` zu instanziieren ist. Diese Information inferriert der Typchecker automatisch aus dem Typ des Arguments.

1.3 Iteration

Typischer Weise wird in einem Programm über die Elemente eines Sammlungstyp iteriert oder über alle Elemente einer Reihung. Hierzu kennt Java verschiedene Schleifenkonstrukte. Leider kannte Java bisher kein eigenes Schleifenkonstrukt, das bequem eine Iteration über die Elemente einer Sammlung ausdrücken konnte. Die Schleifensteuerung mußte bisher immer explizit ausprogrammiert werden. Hierbei können Programmierfehler auftreten, die insbesondere dazu führen können, daß eine Schleife nicht terminiert. Ein Schleifenkonstrukt, das garantiert terminiert, kannte Java bisher nicht.

Beispiel:
In diesen Beispiel finden sich die zwei wahrscheinlich am häufigsten programmierten Schleifentypen. Einmal iterieren wir über alle Elemente einer Reihung

und einmal iterieren wir mittels eines Iteratorobjekts über alle Elemente eines Sammlungsobjekts:

OldIteration.java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

class OldIteration{

  public static void main(String [] _){
    String [] ar
      = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List xs = new ArrayList();

    for (int i= 0;i<ar.length;i++){
      final String s = ar[i];
      xs.add(s);
    }

    for (Iterator it=xs.iterator();it.hasNext();){
      final String s = (String)it.next();
      System.out.println(s.toUpperCase());
    }
  }
}
```

Die Codemenge zur Schleifensteuerung ist gewaltig und übersteigt hier sogar die eigentliche Anwendungslogik.

Mit Java 1.5 gibt es endlich eine Möglichkeit, zu sagen, mache für alle Elemente im nachfolgenden Sammlungsobjekt etwas. Eine solche Syntax ist jetzt in Java integriert. Sie hat die Form:

`for (`*Type identifier* : *expr*`){`*body*`}`

Zu lesen ist dieses Kosntrukt als: für jedes *identifier* des Typs *Type* in *expr* führe *body* aus.[1]

Beispiel:
Damit lassen sich jetzt die Iterationen der letzten beiden Schleifen wesentlich eleganter ausdrücken.

NewIteration.java

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

class NewIteration{

```

[1] Ein noch sprechenderes Konstrukt wäre gewesen, wenn man statt des Doppelpunkts das Schlüsselwort `in` benutzt hätte. Aus Aufwärtskompatibilitätsgründen wird jedoch darauf verzichtet, neue Schlüsselwörter in Java einzuführen.

```
  public static void main(String [] _){
    String [] ar
     = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List<String> xs = new ArrayList<String>();

    for (String s:ar) xs.add(s);
    for (String s:xs) System.out.println(s.toUpperCase());
  }
}
```

Der gesamte Code zur Schleifensteuerung ist entfallen. Zusätzlich ist garantiert, daß für endliche Sammlungsiteratoren auch die Schleife terminiert.

Wie man sieht, ergänzen sich generische Typen und die neue `for`-Schleife.

1.3.1 Die neuen Schnittstellen Iterable

Beispiel:

ReaderIterator.java

```
package sep.util.io;

import java.io.Reader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Iterator;


public class ReaderIterator
              implements Iterable<Character>
                        ,Iterator<Character>{
  private Reader reader;
  private int n;
  public ReaderIterator(Reader r){
    reader=new BufferedReader(r);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
  }
  public Character next(){
    Character result = new Character((char)n);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
    return result;
  }

  public boolean hasNext(){
    return n!=-1;
  }

```



```
  public void remove(){
    throw new UnsupportedOperationException();
  }

  public Iterator<Character> iterator(){return this;}
}
```

TestReaderIterator.java

```
import sep.util.io.ReaderIterator;
import java.io.FileReader;

class TestReaderIterator {
  public static void main(String [] args) throws Exception{
      Iterable<Character> it
        =new ReaderIterator(new FileReader(args[0]));
      for (Character c:it){
      System.out.print(c);
    }
  }
}
```

Beispiel:
Ein abschließendes kleines Beispiel für generische Sammlungsklassen und die neue `for`-Schleife. Die folgende Klasse stellt Methoden zur Verfügung, um einen String in eine Liste von Wörtern zu spalten und umgekehrt aus einer Liste von Wörtern wieder einen String zu bilden:

TextUtils.java

```
import java.util.*;
import java.util.List;

class TextUtils {

  static List<String> words (String s){
    final List<String> result = new ArrayList<String>();

    StringBuffer currentWord = new StringBuffer();

    for (char c:s.toCharArray()){
      if (Character.isWhitespace(c)){
        final String newWord = currentWord.toString().trim();
        if(newWord.length()>0){
          result.add(newWord);
          currentWord=new StringBuffer();
        }
      }else{currentWord.append(c);}
    }
    return result;
  }

  static String unwords(List<String> xs){
```

```
    StringBuffer result=new StringBuffer();
    for (String x:xs) result.append(" "+x);
    return result.toString().trim();
  }

  public static void main(String []_){
    List<String> xs = words("  the   world  is my Oyster  ");

    for (String x:xs) System.out.println(x);

    System.out.println(unwords(xs));
  }
}
```

1.4 Automatisches Boxen

Javas Typsystem ist etwas zweigeteilt. Es gibt Objekttypen und primitive Typen. Die Daten der primitiven Typen stellen keine Objekte dar. Für jeden primitiven Typen gibt es allerdings im Paket `java.lang` eine Klasse, die es erlaubt, Daten eines primitiven Typs als Objekt zu speichern. Dieses sind die Klassen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` und `Character`. Objekte dieser Klassen sind nicht modifizierbar. Einmal ein Integer-Objekt mit einer bestimmten Zahl erzeugt, läßt sich die in diesem Objekt erzeugte Zahl nicht mehr verändern.

Wollte man in bisherigen Klassen ein Datum eines primitiven Typen in einer Variablen speichern, die nur Objekte speichern kann, so mußte man es in einem Objekt der entsprechenden Klasse kapseln. Man spricht von *boxing*. Es kam zu Konstruktoraufrufen dieser Klassen. Sollte später mit Operatoren auf den Zahlen, die durch solche gekapselten Objekte ausgedrückt wurden, gerechnet werden, so war der primitive Wer mit einem Methodenaufruf aus dem Objekt wieder zu extrahieren, dem sogenannten *unboxing*. Es kam zu Aufrufen von Methoden wie `intValue()` im Code.

Beispiel:
In diesem Beispiel sieht man das manuelle Verpacken und Auspacken primitiver Daten.

ManualBoxing.java
```
package name.panitz.boxing;
public class ManualBoxing{
  public static void main(String [] _){
    int i1 = 42;
    Object o = new Integer(i1);
    System.out.println(o);
    Integer i2 = new Integer(17);
    Integer i3 = new Integer(4);
    int i4 = 21;
    System.out.println((i2.intValue()+i3.intValue())*i4);
  }
}
```


Mit Java 1.5 können die primitiven Typen mit ihren entsprechenden Klassen synonym verwendet werden. Nach außen hin werden die primitiven Typen auch zu Objekttypen. Der Übersetzer nimmt fügt die notwendigen *boxing*- und *unboxing*-Operationen vor.

Beispiel:
Jetzt das vorherige kleine Programm ohne explizite *boxing*- und *unboxing*-Aufrufe.

AutomaticBoxing.java
```
package name.panitz.boxing;
public class AutomaticBoxing{
  public static void main(String [] _){
    int i1 = 42;
    Object o = i1;
    System.out.println(o);
    Integer i2 = 17;
    Integer i3 = 4;
    int i4 = 21;
    System.out.println((i2+i3)*i4);
  }
}
```

1.5 Aufzählungstypen

Häufig möchte man in einem Programm mit einer endlichen Menge von Werten rechnen. Java bot bis Version 1.4 kein ausgezeichnetes Konstrukt an, um dieses auszudrücken. Man war gezwungen in diesem Fall sich des Typs `int` zu bedienen und statische Konstanten dieses Typs zu deklarieren. Dieses sieht man auch häufig in Bibliotheken von Java umgesetzt. Etwas mächtiger und weniger primitiv ist, eine Klasse zu schreiben, in der es entsprechende Konstanten dieser Klasse gibt, die durchnummeriert sind. Dieses ist ein Programmiermuster, das Aufzählungsmuster.

Mit Java 1.5 ist ein expliziter Aufzählungstyp in Java integriert worden. Syntaktisch erscheint dieser wie eine Klasse, die statt des Schlüsselworts `class` das Schlüsselwort `enum` hat. Es folgt als erstes in diesen Aufzählungsklassen die Aufzählung der einzelnen Werte.

Beispiel:
Ein erster Aufzählungstyp für die Wochentage.

Wochentage.java
```
package name.panitz.enums;
public enum Wochentage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;
}
```

Auch dieses neue Konstrukt wird von Javaübersetzer in eine herkömmlige Javaklasse übersetzt. Wir können uns davon überzeugen, indem wir uns einmal den Inhalt der erzeugten Klassendatei mit `javap` wieder anzeigen lassen:

```
sep@linux:fh/> javap name.panitz.enums.Wochentage
Compiled from "Wochentage.java"
public class name.panitz.enums.Wochentage extends java.lang.Enum{
    public static final name.panitz.enums.Wochentage montag;
    public static final name.panitz.enums.Wochentage dienstag;
    public static final name.panitz.enums.Wochentage mittwoch;
    public static final name.panitz.enums.Wochentage donnerstag;
    public static final name.panitz.enums.Wochentage freitag;
    public static final name.panitz.enums.Wochentage sonnabend;
    public static final name.panitz.enums.Wochentage sonntag;
    public static final name.panitz.enums.Wochentage[] values();
    public static name.panitz.enums.Wochentage valueOf(java.lang.String);
    public name.panitz.enums.Wochentage(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
```

Eine der schönen Eigenschaften der Aufzählungstypen ist, daß sie in einer `switch`-Anweisung benutzt werden können.

Beispiel:
Wir fügen der Aufzählungsklasse eine Methode zu, um zu testen ob der Tag ein Werktag ist. Hierbei läßt sich eine `switch`-Anweisung benutzen.

Tage.java

```
package name.panitz.enums;
public enum Tage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;

  public boolean isWerktag(){
    switch (this){
      case sonntag    :
      case sonnabend  :return false;
      default         :return true;
    }
  }

  public static void main(String [] _){
    Tage tag = freitag;
    System.out.println(tag);
    System.out.println(tag.ordinal());
    System.out.println(tag.isWerktag());
    System.out.println(sonntag.isWerktag());
  }
}
```

Das Programm gibt die erwartete Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Tage
freitag
4
true
false
sep@linux:~/fh/java1.5/examples>
```


Eine angenehme Eigenschaft der Aufzählungsklassen ist, daß sie in einer Reihung alle Werte der Aufzählung enthalten, so daß mit der neuen `for`-Schleife bequem über diese iteriert werden kann.

Beispiel:
Wir iterieren in diesem Beispiel einmal über alle Wochentage.

IterTage.java

```
package name.panitz.enums;
public class IterTage {
  public static void main(String [] _){
    for (Tage tag:Tage.values())
      System.out.println(tag.ordinal()+": "+tag);
  }
}
```

Die erwarttete Ausgabe ist:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.IterTage
0: montag
1: dienstag
2: mittwoch
3: donnerstag
4: freitag
5: sonnabend
6: sonntag
sep@linux:~/fh/java1.5/examples>
```

Schließlich kann man den einzelnen Konstanten einer Aufzählung noch Werte übergeben.

Beispiel:
Wir schreiben eine Aufzählung für die Euroscheine. Jeder Scheinkonstante wird noch eine ganze Zahl mit übergeben. Es muß hierfür ein allgemeiner Konstruktor geschrieben werden, der diesen Parameter übergeben bekommt.

Euroschein.java

```
package name.panitz.enums;
public enum Euroschein {
   fünf(5),zehn(10),zwanzig(20),fünfzig(50),hundert(100)
  ,zweihundert(200),tausend(1000);
  private int value;
  public Euroschein(int v){value=v;}
  public int value(){return value();}

  public static void main(String [] _){
    for (Euroschein schein:Euroschein.values())
      System.out.println
        (schein.ordinal()+": "+schein+" -> "+schein.value);
  }
}
```

Das Programm hat die folgende Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Euroschein
0: fünf -> 5
1: zehn -> 10
2: zwanzig -> 20
3: fünfzig -> 50
4: hundert -> 100
5: zweihundert -> 200
6: tausend -> 1000
sep@linux:~/fh/java1.5/examples>
```

1.6 Statische Imports

Statische Eigenschaften einer Klasse werden in Java dadurh angesprochen, daß dem Namen der Klasse mit Punkt getrennt die gewünschte Eigenschaft folgt. Werden in einer Klasse sehr oft statische Eigenschaften einer anderen Klasse benutzt, so ist der Code mit deren Klassennamen durchsetzt. Die Javaentwickler haben mit Java 1.5 ein Einsehen. Man kann jetzt für eine Klasse alle ihre statischen Eigenschaften importieren, so daß diese unqualifiziert benutzt werden kann. Die `import`-Anweisung sieht aus wie ein gewohntes Paktimport, nur daß das Schlüsselwort `static` eingefügt ist und erst dem klassennamen der Stern folgt, der in diesen Fall für alle statischen Eigenschaften steht.

Beispiel:
Wir schreiben eine Hilfsklasse zum Arbeiten mit Strings, in der wir eine Methode zum umdrehen eines Strings vorsehen:

StringUtil.java

```java
package name.panitz.staticImport;
public class StringUtil {
  static public String reverse(String arg) {
    StringBuffer result = new StringBuffer();
    for (char c:arg.toCharArray()) result.insert(0,c);
    return result.toString();
  }
}
```

Die Methode `reverse` wollen wir in einer anderen Klasse benutzen. Importieren wir die statischen Eigenschaften von `StringUtil`, so können wir auf die Qualifizierung des Namens der Methode `reverse` verzichten:

UseStringUtil.java

```java
package name.panitz.staticImport;
import static name.panitz.staticImport.StringUtil.*;
public class UseStringUtil {
  static public void main(String [] args) {
    for (String arg:args)
     System.out.println(reverse(arg));
  }
}
```

Die Ausgabe dieses programms:


```
sep@linux:fh> java -classpath classes/ name.panitz.staticImport.UseStringUtil hallo welt
ollah
tlew
sep@linux:~/fh/java1.5/examples>
```

1.7 Variable Parameteranzahl

Als zusätzliches kleines Gimmik ist in Java 1.5 eingebaut worden, daß Methoden mit einer variablen Parameteranzahl definiert werden können. Dieses wird durch drei Punkte nach dem Parametertyp in der Signatur gekennzeichnet. Damit wird angegeben, daß eine bieliebige Anzahl dieser Parameter bei einem Methodenaufruf geben kann.

Beispiel:
Es läßt sich so eine Methode schreiben, die mit beliebig vielen Stringparametern aufgerufen werden kann.

VarParams.java

```java
package name.panitz.java15;

public class VarParams{
  static public String append(String... args){
    String result="";
    for (String a:args)
      result=result+a;
    return result;
  }

  public static void main(String [] _){
    System.out.println(append("hello"," ","world"));
  }
}
```

Die Methode `append` konkateniert endlich viele `String`-Objekte.

Wie schon für Aufzählungen können wir auch einmal schauen, was für Code der Javakompilierer für solche Methoden erzeugt.

```
sep@linux:~/fh/java1.5/examples> javap -classpath classes/ name.panitz.java15.VarParams
Compiled from "VarParams.java"
public class name.panitz.java15.VarParams extends java.lang.Object{
    public name.panitz.java15.VarParams();
    public static java.lang.String append(java.lang.String[]);
    public static void main(java.lang.String[]);
}

sep@linux:~/fh/java1.5/examples>
```

Wie man sieht wird für die variable Parameteranzahl eine Reihung erzeugt. Der Javakompilierer sorgt bei Aufrufen der Methode dafür, daß die entsprechenden Parameter in eine Reihung verpackt werden. Daher können wir mit dem Parameter wie mit einer Reihung arbeiten.

1.8 Ein paar Beispielklassen

In Java gibt es keinen Funktionstyp als Typ erster Klasse. Funktionen können nur als Methoden eines Objektes als Parameter weitergereicht werden. Mit generischen Typen können wir eine Schnittstelle schreiben, die ausdrücken soll, daß das implementieren Objekt eine einstellige Funktion darstellt.

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

public interface UnaryFunction<arg,result>{
  public result eval(arg a);
}
```

Ebenso können wir eine Schnittstelle für konstante Methoden vorsehen:

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

public interface Closure<result>{
  public result eval();
}
```

Wir haben die generischen Typen eingeführt anhand der einfachen Klasse `Box`. Häufig benötigt man für die Werterückgabe von Methoden kurzzeitig eine Tupelklasse. Im folgenden sind ein paar solche Tupelklasse generisch realisiert:

Tuple1.java
```
package name.panitz.crempel.util;

public class Tuple1<t1> {
  public t1 e1;
  public Tuple1(t1 a1){e1=a1;}
  String parenthes(Object o){return "("+o+")";}
  String simpleToString(){return e1.toString();}
  public String toString(){return parenthes(simpleToString());}
  public boolean equals(Object other){
    if (! (other instanceof Tuple1)) return false;
    return e1.equals(((Tuple1)other).e1);
  }
}
```

Tuple2.java
```
package name.panitz.crempel.util;

public class Tuple2<t1,t2> extends Tuple1<t1>{
  public t2 e2;
  public Tuple2(t1 a1,t2 a2){super(a1);e2=a2;}
  String simpleToString(){
    return super.simpleToString()+","+e2.toString();}
```



```
  public boolean equals(Object other){
    if (! (other instanceof Tuple2)) return false;
    return super.equals(other)&& e2.equals(((Tuple2)other).e2);
  }
}
```

Tuple3.java
```
package name.panitz.crempel.util;

public class Tuple3<t1,t2,t3> extends Tuple2<t1,t2>{
  public t3 e3;
  public Tuple3(t1 a1,t2 a2,t3 a3){super(a1,a2);e3=a3;}
  String simpleToString(){
    return super.simpleToString()+","+e3.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple3)) return false;
    return super.equals(other)&& e3.equals(((Tuple3)other).e3);
  }
}
```

Tuple4.java
```
package name.panitz.crempel.util;

public class Tuple4<t1,t2,t3,t4> extends Tuple3<t1,t2,t3>{
  public t4 e4;
  public Tuple4(t1 a1,t2 a2,t3 a3,t4 a4){super(a1,a2,a3);e4=a4;}
  String simpleToString(){
    return super.simpleToString()+","+e4.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple4)) return false;
    return super.equals(other)&& e4.equals(((Tuple4)other).e4);
  }
}
```

Zum Iterieren über einen Zahlenbereich können wir eine entsprechende Klasse vorsehen.

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

import java.util.Iterator;

public class FromTo implements Iterable<Integer>,Iterator<Integer>{
  private final int to;
  private int from;
  public FromTo(int f,int t){to=t;from=f;}
  public boolean hasNext(){return from<=to;}
  public Integer next(){int result = from;from=from+1;return result;}
  public Iterator<Integer> iterator(){return this;}
  public void remove(){new UnsupportedOperationException();}
}
```

Statt mit `null` zu Arbeiten, kann man einen Typen vorsehen, der entweder gerade einen Wert oder das Nichtvorhandensein eines Wertes darstellt. In funktionalen Sprachen gibt es hierzu den entsprechenden generischen algebraischen Datentypen:

HsMaybe.hs

```
data Maybe a = Nothing|Just a
```

Dieses läßt sich durch eine generische Schnittstelle und zwei generische implementierende Klassen in Java ausdrücken.

Maybe.java

```
package name.panitz.crempel.util;
public interface Maybe<a> {}
```

Nothing.java

```
package name.panitz.crempel.util;
public class Nothing<a> implements Maybe<a>{

  public String toString(){return "Nothing("+")";}
  public boolean equals(Object other){
    return (other instanceof Nothing);
  }
}
```

Just.java

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

public class Just<a> implements Maybe<a>{
  private a just;

  public Just(a just){this.just = just;}
  public a getJust(){return just;}

  public String toString(){return "Just("+just+")";}
  public boolean equals(Object other){
    if (!(other instanceof Just)) return false;
    final Just o= (Just) other;
    return just.equals(o.just);
  }
}
```

Die jetzt folgende Klasse möge der leser bitte ignorieren. Sie ist aus rein technischen internen Gründen an dieser Stelle.

CrempelTool.java

```
package name.panitz.crempel.tool;
public interface CrempelTool{String getDescription();void startUp();}
```


1.9 Aufgaben

Aufgabe 1

a) Schreiben Sie eine Schnittstelle `Smaller`, in der es eine Methode `le` mit zwei generischen Parametern gleichen Typs gibt.

Lösung

Smaller.java

```
package name.panitz.aufgaben;
public interface Smaller<at> {
  public boolean le(at x1,at x2);
}
```

b) Schreiben Sie eine Klasse, die `Smaller<Integer>`, so implementiert, daß gerade Zahlen kleiner als ungerade Zahlen sind.

Lösung

SmallEven.java

```
package name.panitz.aufgaben;
public class SmallEven implements Smaller<Integer> {
  public boolean le(Integer x1,Integer x2){
    if (x1%2 != x2%2){return x1%2==0;};
    return x1<=x2;
  }
}
```

c) Schreiben Sie eine Klasse, die `Smaller<String>`, so implementiert, daß kurze Zeichenketten kleiner als lange sind.

Lösung

SmallShort.java

```
package name.panitz.aufgaben;
public class SmallShort implements Smaller<String> {
  public boolean le(String x1,String x2){
    return x1.length()<=x2.length();
  }
}
```

d) Schreiben Sie den Methodenkopf für eine statische generische Methode `bubbleSort` zum Sortieren von Reihungen, die zwei Parameter hat: eine zu sortierende Reihung und ein passendes Objekt des Typs `Smaller`.

Lösung

Hier mit dem Methodenrumpf, der aber nicht verlangt war:

Bubble.java

```
package name.panitz.aufgaben;
public class Bubble {
  public static <at> void bubbleSort(at[] ar,Smaller<at> rel){
    boolean toBubble = true;
    int end=ar.length;
    while (toBubble){
      toBubble = bubble(ar,rel,end);
      end=end-1;
    }
  }

  static <at> boolean bubble(at[] ar,Smaller<at> rel,int end){
    boolean result = false;
    for (int i=0;i<end;i=i+1){
      try {
        if (!rel.le(ar[i],ar[i+1])){
          at o = ar[i];
          ar[i]=ar[i+1];
          ar[i+1]=o;
          result=true;
        }
      }catch (ArrayIndexOutOfBoundsException _){}
    }
    return result;
  }
}
```

e) Schreiben Sie zwei Beispielaufrufe der Methode `bubbleSort` mit `SmallerEven` bzw. `SmallerShort`.

Lösung

TestBubble.java

```
package name.panitz.aufgaben;
public class TestBubble{
  public static void main(String [] _){
    String [] xs = {"1","4444","22","55555","","333"};
    System.out.print("[");
    for (String x:xs)System.out.print(","+x);
    System.out.println("]");
    Bubble.bubbleSort(xs,new SmallShort());
    System.out.print("[");
    for (String x:xs)System.out.print(","+x);
    System.out.println("]");

    int [] ys_ = {2,3,4,5,6,7,78,8,8};
    Integer [] ys = new Integer[9];
    for (int i=0;i<9;i++) ys[i]=new Integer(ys_[i]);
    System.out.print("[");
```



```
    for (Integer x:ys)System.out.print(","+x);
    System.out.println("]");
    Bubble.bubbleSort(ys,new SmallEven());
    System.out.print("[");
    for (Integer x:ys)System.out.print(","+x);
    System.out.println("]");
  }
}
```

Kapitel 2

Algebraische Typen und Besucher

2.1 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.

2.1.1 Funktionale Programmierung

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

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]

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.

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

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.

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.

2.1.2 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.

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 Übersetzungszeit 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.

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.

2.2 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.

2.2.1 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.

2.2.2 Implementierungen in Java

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

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());
  }
}
```

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>())
       )));
  }
}
```

2.3 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.

2.3.1 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 [Pan04a] 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.

2.3.2 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
```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

```
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.

2.4 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.

2.4.1 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.

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);
  }
}
```

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.

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.

2.4.2 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 [Pan03b]. 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.

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.

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
```
  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
```
      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
```
  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
```
  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

```
  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()+" _);";
  }
}
```

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();
    }
}
```

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

```
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();}
  }
}
```


2.5 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.

2.5.1 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

```
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.

2.5.2 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
```
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();
  }
}
```

2.5.3 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 Ergeb-

nis 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.

2.5.4 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)
```

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"
}
```

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);}
}
```

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.

2.6 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;
```



```
}
}
```

2.7 Aufgaben

Aufgabe 2 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 3 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 4 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;
  }
}
```

Kapitel 3

XML

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Beispiel:
Der eigentliche Text des Dokuments sei:

```
The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

```
<cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
</cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem W3C (`http://www.w3c.org`) . Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)*, weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt. Die aktuelle Empfehlung für XML liegt seit anfang des Jahres als Empfehlung in der Version 1.1 vor [T. 04].

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran, daß es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so daß die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[1]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

3.1 XML-Format

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, daß es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

```
<?xml version="1.0"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.0. Ein Entwurf für die Version 1.1 liegt vor. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, daß es jemals eine Version 2.0 von XML geben wird. Zuviele weitere Techniken und Empfehlungen basieren auf XML, so daß die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

3.1.1 Elemente

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen*, der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit `<name>` und endet mit `</name>`. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, daß es genau ein einziges oberstes Element hat.

Beispiel:
Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

```
<?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText>
```

[1] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Beispiel:
Folgendes Dokument enthält ein Element ohne Inhalt:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page></page>
  <page>dritte Seite</page>
</skript>
```

Leere Elemente

Für ein Element mit Tagnamen *name*, das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Beispiel:
Das vorherige Dokument läßt sich somit auch wie folgt schreiben:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page/>
  <page>dritte Seite</page>
</skript>
```

Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)*.

Beispiel:
Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

```
<?xml version="1.0"?>
<myText>Der <landsmann>Italiener</landsmann>
<eigename>Ferdinand Carulli</eigename> war  als Gitarrist
ebenso wie der <landsmann>Spanier</landsmann>
<eigename>Fernando Sor</eigename> in <ort>Paris</ort>
ansäßig.</myText>
```

XML-Dokumente als Bäume

Die wohl wichtigste Beschränkung für XML-Dokumente ist, daß sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder geschlossen wurden.

> **Beispiel:**
> Das folgende ist kein gültiges XML-Dokument. Das Element <bf> wird geschlossen bevor das später geöffnete Element <em> geschlossen worde.

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em>kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

> Das Dokument wäre wie folgt als gültiges XML zu schreiben:

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift</bf><bf> <em>kursiv und fett</em></bf>
  <em>nur noch kursiv</em>.
</validDocument>
```

> Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur von XML-Dokumenten läßt sich sehr schön veranschaulichen, wenn man die Darstellung von XML-Dokumenten in Microsofts Internet Explorer betrachtet.

3.1.2 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet.

> **Beispiel:**
> Dokument mit einem Attribut für ein Element.

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```


3.1.3 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit <!-- begonnen und mit --> beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes.

> **Beispiel:**
> Im folgenden Dokument ist ein Kommentar eingefügt:

```
<?xml version="1.0"?>
<drehbuch filmtitel="Ben Hur">
<akt>
  <szene>Ben Hur am Vorabend des Wagenrennens.
    <!--Diese Szene muß noch ausgearbeitet werden.-->
  </szene>
</akt>
</drehbuch>
```

3.1.4 Character Entities

Sobald in einem XML-Dokument eine der spitze Klammern < oder > auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muß man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities*. Eine Character Entity beginnt in XML mit dem Zeichen & und endet mit einem Semikolon;. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

Entity	Zeichen	Beschreibung
<	<	(less than)
>	>	(greater than)
&	&	(ampersant)
"	"	(quotation mark)
'	'	(apostroph)

Somit lassen sich in XML auch Dokumente schreiben, die diese Zeichen als Text beinhalten.

> **Beispiel:**
> Folgendes Dokument benutzt Character Entities um mathematische Formeln zu schreiben:

```
<?xml version="1.0"?>
<gleichungen>
  <gleichung>x+1&gt;x</gleichung>
  <gleichung>x*x&lt;x*x*x für x&gt;1</gleichung>
</gleichungen>
```

3.1.5 CDATA-Sections

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeutung haben. XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge: `]]>`. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden.

Beispiel:
Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>
```

3.1.6 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction* mit `<?` und endet mit `?>`. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle.

Beispiel:
Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, daß das Skript in eine HTML-Darstellung umwandelt:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

3.1.7 Namensräume

Die *Tagnamen* sind zunächst einmal Schall und Rauch. Erst eine externes Programm wird diesen Namen eine gewisse Bedeutung zukommen lassen, indem es auf die *Tagnamen* in einer bestimmten Weise reagiert.

Da jeder Autor eines XML-Dokuments zunächst vollkommen frei in der Wahl seiner *Tagnamen ist*, wird es vorkommen, daß zwei Autoren denselben *Tagnamen* für die Markierung gewählt haben, aber semantisch mit diesem Element etwas anderes ausdrücken wollen. Spätestens dann, wenn verschiedene Dokumente verknüpft werden, wäre es wichtig, daß *Tagnamen* einmalig mit einer Eindeutigen Bedeutung benutzt wurden. Hierzu gibt es in XML das Konzept der Namensräume.

Tagnamen können aus zwei Teilen bestehen, die durch einen Doppelpunkt getrennt werden:

- dem Präfix, der vor dem Doppelpunkt steht.
- dem lokalen Namen, der nach dem Doppelpunkt folgt.

Hiermit allein ist das eigentliche Problem gleicher *Tagnamen* noch nicht gelöst, weil ja zwei Autoren den gleichen Präfix und gleichen lokalen Namen für ihre Elemente gewählt haben können. Der Präfix wird aber an einem weiteren Text gebunden, der eindeutig ist. Dieses ist der eigentliche Namensraum. Damit garantiert ist, daß dieser Namensraum tatsächlich eindeutig ist, wählt man als Autor seine Webadresse, denn diese ist weltweit eindeutig.

Um mit Namensräume zu arbeiten ist also zunächst ein Präfix an eine Webadresse zu binden; dies geschieht durch ein Attribut der Art:
`xmlns:myPrefix="http://www.myAdress.org/myNamespace"`.

Beispiel:
Ein Beispiel für ein XML-Dokument, daß den Präfix `sep` an einem bestimmten Namensraum gebunden hat:

```
<?xml version="1.0"?>
<sep:skript
   xmlns:sep="http://www.tfh-berlin.de/~panitz/dv2">
  <sep:titel>Grundlagen der DV 2</sep:titel>
  <sep:autor>Sven Eric Panitz</sep:autor>
</sep:skript>

```

Die Webadresse eines Namensraumes hat keine eigentliche Bedeutung im Sinne des Internets. Das Dokument geht nicht zu dieser Adresse und holt sich etwa Informationen von dort. Es ist lediglich dazu da, einen eindeutigen Namen zu haben. Streng genommen brauch es diese Adresse noch nicht einmal wirklich zu geben.

3.2 Codierungen

XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehrerern zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen die 8 Bit im Speicher belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt.

Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Beispiel:
Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes.

Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

3.3 Dokumente als Bäume in Java

Wie wir festgestellt haben, sind XML-Dokumente mit ihrer hierarchischen Struktur Bäume. In den Vorgängervorlesungen haben wir uns schon auf unterschiedliche Weise mit Bäumen beschäftigt. Von den dabei gemachten Erfahrungen können wir jetzt profitieren.

3.3.1 Ein algebraischer Typ für XML

Im letzten Kapitel haben wir ein sehr mächtiges Tool entwickelt, um Javacode für algebraische Datenstrukturen zu generieren. XML läßt sich sehr intuitiv als algebraische Datenstruktur formulieren. Bevor wir uns fertigen Java-Bibliotheken zum Bearbeiten von XML-Dokumenten zuwenden, entwickeln wir eine solche Bibliothek selbst. Wir schreiben eine Spezifikation des Typs XML. Die einzelnen Knotentypen bekommen dabei einen eigenen Konstrukor.

XML.adt
```
package name.panitz.xml;
import java.util.List;
data class XML{
  Element(Name name,List attributes,List children);
  Comment(String comment);
  Text(String text);
  CDataSection(String cdata);
  Entity(String name);
  ProcessingInstruction(String target,String instruction);
}
```

Damit haben wir knapp spezifizieren können, wie die Struktur eines XML-Dokuments aussieht[2].

Wir haben es vorgezogen Attribute nicht als eine Unterklasse der Klasse `XML` zu spezifizieren, sondern sehen hierfür eine eigene Klasse vor.

Attribute.java
```
package name.panitz.xml;
public class Attribute{
  Name name; String value;
  public Attribute(Name n,String v){name=n;value=v;}
  public String toString(){return name+" = \""+value+"\"";}
}
```

Tagnamen und Attributnamen sind Objekte einer eigenen Klasse, deren Objekte den Namen und den Prefix seperat speichern und die zusätzlich noch die Möglichkeit haben, für den Namensraum die URI abzuspeiechern.

Name.java
```
package name.panitz.xml;
import java.util.Map;
public class Name{
  String prefix; String name;String nas;
  public Name(String p,String n,String s){prefix=p;name=n;nas=s;}

  public String toString(){
    return prefix+(prefix.length()==0":":")+name;
  }
```

[2]Die Attribute und Kinder eines Elementknotens sind nicht als die generischen Instanzen `List<Attribute>` und `List<XML>` spezifiziert, da der benutze Parsergenerator `javacc` noch keine generischen Typen unterstützt.

Wir sehen für diese Klasse einige Gleicheitsmethoden vor. Die Standardgleichheit testet dabei auf gleichen Prefix und Namen und ignoriert die Bindung des Namensraumes an den Prefix.

Name.java
```java
  public boolean equals(Object other){
    if (other instanceof Name) return equals((Name)other);
    return false;
  }

  public boolean equals(Name other){
    return name.equals(other.name)&&prefix.equals(other.prefix) ;
  }
```

Einer weiteren Gleichheitsmethode wird eine Namensraumbindung der Prefixe in Form eines `Map` übergeben.

Name.java
```java
  public boolean equals
                (Name other,Map<String,String> namespaceBinding){
    return name.equals(other.name)
     && namespaceBinding.get(prefix)
        .equals(namespaceBinding.get(other.prefix));
  }
}
```

Mit wenigen Zeilen konnten wir einen Javatypen spezifizieren, um XML Dokumente in ihrer logischen Struktur zu speichern. Im Anhang ist eine rudimentäre Grammatik für den Parsergenerator `javacc` angegeben, der Instanzen des Typs `XML` erzeugt.

Besucher für den Typ XML

Unser Generatorprogramm für algebraische Typen erzeugt die Infrstruktur zum Schreiben von Besuchermethoden.

Stringdarstellung Ein erster Besucher zeige den XML-Baum wieder als gültigen XML-String an.

Show.java
```java
package name.panitz.xml;
import java.util.List;

public class Show extends XMLVisitor<StringBuffer> {
  StringBuffer result = new StringBuffer();

  public StringBuffer eval(Element element){
    result.append("<");
    result.append(element.getName().toString());
    for (Attribute attr:(List<Attribute>)element.getAttributes())
```



```java
      result.append("\u0020"+attr);
    if (element.getChildren().isEmpty())result.append("/>");
    else {
      result.append(">");
      for (XML xml:(List<XML>)element.getChildren())
        xml.visit(this);
      result.append("</");
      result.append(element.getName().toString());
      result.append(">");}
    return result; }
  public StringBuffer eval(Comment comment){
    result.append("<!--"+comment.getComment()+"-->");
    return result;}
  public StringBuffer eval(Text text){
    result.append(text.getText());
    return result;}
  public StringBuffer eval(CDataSection cdata){
    result.append("<![CDATA["+cdata.getCdata()+"]"+"]>");
    return result;}
  public StringBuffer eval(Entity entity){
    result.append("&"+entity.getName()+";");
    return result;}
  public StringBuffer eval(ProcessingInstruction pi){
    result.append("<?"+pi.getTarget()+"\u0020");
    result.append(pi.getInstruction()+"?>");
    return result;}
}
```

Elementselektion Ein erster interessanter Besucher selektiert aus einem XML-Dokument alle Elementknoten, deren Namen in einer Liste von Namen auftaucht.

SelectElement.java
```java
package name.panitz.xml;
import java.util.List;
import java.util.ArrayList;

public class SelectElement extends  XMLVisitor<List<XML>> {
  List<Name> selectThese;
  List<XML> result = new ArrayList<XML>();
  public SelectElement(List<Name> st){selectThese=st;}
  public SelectElement(Name n){
    selectThese=new ArrayList<Name>();
    selectThese.add(n);
  }

  public List<XML> eval(Element element){
    if (selectThese.contains(element.getName()))
      result.add(element);
    for (XML xml:(List<XML>)element.getChildren())
```

```
      xml.visit(this);
    return result;
  }
  public List<XML> eval(Comment comment){return result;}
  public List<XML> eval(Text text){return result;}
  public List<XML> eval(CDataSection cdata){return result;}
  public List<XML> eval(Entity entity){return result;}
  public List<XML> eval(ProcessingInstruction pi){return result;}
}
```

Knotenanzahl In diesem Besucher zählen wir auf altbekannte Weise die Knoten in einem Dokument.

Size.java

```
package name.panitz.xml;
import java.util.List;

public class Size extends  XMLVisitor<Integer> {
  int result=0;

  public Integer eval(Element element){
    result=result+1;
    for (XML xml:(List<XML>)element.getChildren())
      xml.visit(this);
    return result;
  }
  public Integer eval(Comment comment){
    result=result+1;return result;}
  public Integer eval(Text text){result=result+1;return result;}
  public Integer eval(CDataSection cdata){
    result=result+1;return result;}
  public Integer eval(Entity entity){
    result=result+1;return result;}
  public Integer eval(ProcessingInstruction pi){
    result=result+1;return result;}
}
```

Baumtiefe Und als weiteres Beispiel folge der Besucher, der die maximale Pfadlänge zu einem Blatt im Dokument berechnet:

Depth.java

```
package name.panitz.xml;
import java.util.List;

public class Depth extends  XMLVisitor<Integer> {
  public Integer eval(Element element){
    int result=0;
    for (XML xml:(List<XML>)element.getChildren()){
      final int currentDepth = xml.visit(this);
```



```
      result=result<currentDepth?currentDepth:result;
    }
    return result+1;
  }
  public Integer eval(Comment comment){return 1;}
  public Integer eval(Text text){return 1;}
  public Integer eval(CDataSection cdata){return 1;}
  public Integer eval(Entity entity){return 1;}
  public Integer eval(ProcessingInstruction pi){return 1;}
}
```

Dokumententext Sind wir nur an den eigentlichen Text eines Dokuments, ohne die Markierungen interessiert, so können wir den folgenden Besucher benutzen.

Content.java

```
package name.panitz.xml;
import java.util.List;

public class Content extends XMLVisitor<StringBuffer> {
  StringBuffer result = new StringBuffer();

  public StringBuffer eval(Element element){
    for (XML xml:(List<XML>)element.getChildren())xml.visit(this);
    return result; }
  public StringBuffer eval(Comment comment){return result;}
  public StringBuffer eval(Text text){
    result.append(text.getText());
    return result;}
  public StringBuffer eval(CDataSection cdata){
    result.append(cdata.getCdata());
    return result;}
  public StringBuffer eval(Entity entity){return result;}
  public StringBuffer eval(ProcessingInstruction pi){
    return result;}
}
```

Dokument in einer HTML-Darstellung Schließlich können wir XML-Dokumente für die Darstellung als HTML konvertieren. Hierzu läßt sich das HTML-Tag `<df>` für eine *definition list* benutzen, mit den beiden Elementen `<dt>` für die Überschrift eines Listeintrages und `<dd>` für einen Listeneintrag. Wir konvertieren einen XML Baum so, daß jeder Elementknoten eine neue Liste erzeugt, deren Listenpunkte die Kinder des Knotens sind.

ToHTML.java

```
package name.panitz.xml;
import java.util.List;

public class ToHTML extends XMLVisitor<StringBuffer> {
  StringBuffer result = new StringBuffer();

```

```
  public StringBuffer eval(Element element){
    result.append("<dl>");
    result.append("<dt><tt><b><font color=\"red\">&lt;"
                  +element.getName()+"</font></b></tt>");
    if (!element.getAttributes().isEmpty()){
      result.append("<dl>");
      for (Attribute atr:(List<Attribute>)element.getAttributes()){
        result.append("<dd>");
        result.append("<b><font = color=\"blue\">"+atr.name);
        result.append("</font> = <font color=\"green\">");
        result.append("\""+atr.value+"\"</font></b>");
        result.append("</dd>");
      }
      result.append("</dl>");
    }
    result.append("<tt><b><font color=\"red\">");
    result.append("&gt;</font></b></tt></dt>");
    for (XML xml:(List<XML>)element.getChildren()){
      result.append("<dd>");
      xml.visit(this);
      result.append("</dd>");
    }
    result.append("<dt><tt><b><font color=\"red\">&lt;/"
                  +element.getName()+"&gt;</font></b></tt></dt>");
    result.append("</dl>");
    return result; }
  public StringBuffer eval(Comment comment){return result;}
  public StringBuffer eval(Text text){
    result.append(text.getText());
    return result;}
  public StringBuffer eval(CDataSection cdata){
    result.append(cdata.getCdata());
    return result;}
  public StringBuffer eval(Entity entity){return result;}
  public StringBuffer eval(ProcessingInstruction pi){
    return result;}
}
```

Abbildung 3.1 zeigt den Quelltext dieses Skriptes in seiner HTML-Konvertierung in einem Webbrowser angezeigt.

3.3.2 APIs für XML

Nachdem wir oben unsere eigene Datenstruktur für XML-Dokumente entwickelt haben, wollen wir jetzt einen Blick auf die gängigen Schnittstellen für die XML-Programmierung werfen.

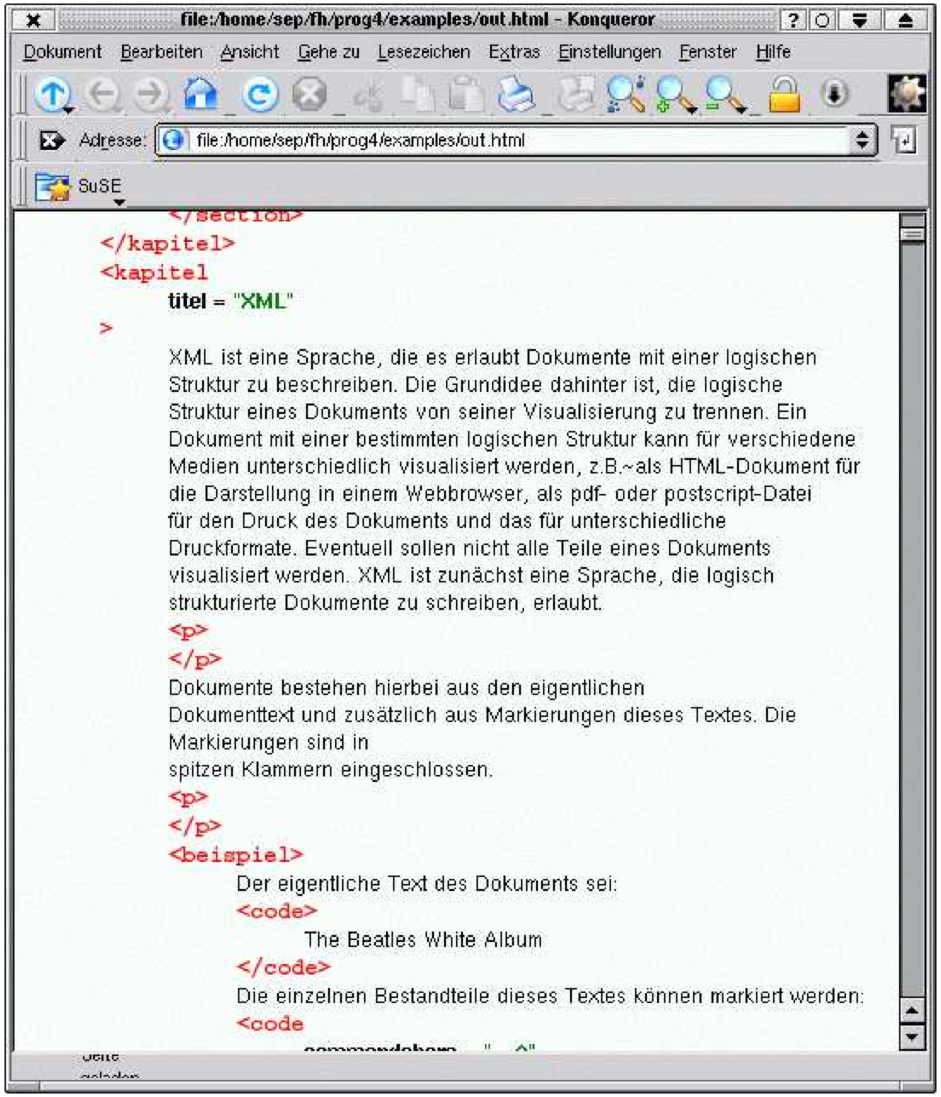

Abbildung 3.1: Anzeige des XML Dokumentes dieses Skriptes als HTML.

DOM

Die allgemeine Schnittstellenbeschreibung für XML als Baumstruktur ist das *distributed object modell* kurz *dom*[Arn04], für das das W3C eine Empfehlung herausgibt. Die Ursprünge von *dom* liegen nicht in der XML-Programmierung sondern in der Verarbeitung von HTML-Strukturen im Webbrowser über Javascript. Mit dem Auftreten von XML entwickelte sich der Wunsch, eine allgemeine, platform- und sprachunabhänge Schnittstellenbeschreibung für XML- und HTML-Baumstrukturen zu bekommen, die in verschiedenen Sprachen umgesetzt werden kann.

Da es sich um ein implementierungsunabhängiges API handelt, fiden wir *dom* in der Javabibliothek nur als Schnittstellen.

Wichtige Schnittstellen Die zentrale Schnittstelle in *dom* ist `Node`. Sie hat als Unterschnittstellen alle Knotentypen, die es in XML gibt. Folgende Graphik gibt über diese Knotentypen einen Überblick.

```
interface org.w3c.dom.Node
 |
 |--interface org.w3c.dom.Attr
 |--interface org.w3c.dom.CharacterData
 |    |
 |    |--interface org.w3c.dom.Comment
 |    |--interface org.w3c.dom.Text
 |        |
 |        |--interface org.w3c.dom.CDATASection
 |
 |--interface org.w3c.dom.Document
 |--interface org.w3c.dom.DocumentFragment
 |--interface org.w3c.dom.DocumentType
 |--interface org.w3c.dom.Element
 |--interface org.w3c.dom.Entity
 |--interface org.w3c.dom.EntityReference
 |--interface org.w3c.dom.Notation
 |--interface org.w3c.dom.ProcessingInstruction
```

Eine der entscheidenen Methoden der Schnittstelle `Node` selektiert die Liste der Kinder eines Knotens:
`public NodeList getChildNodes()`

Knoten, die keine Kinder haben können (Textknoten, Attribute etc.) geben bei dieser Methode die leere Liste zurück. Attribute zählen auch wie in unserer Modellierung nicht zu den Kindern eines Knotens. Um an die Attribute zu gelangen, gibt es eine eigene Methode:
`NamedNodeMap getAttributes()`

Wie man sieht, benutzt Javas *dom* Umsetzung keine von Javas Listenklassen zur Umsetzung einer Knotenliste, sondern nur genau die in *dom* spezifizierte Schnittstelle `NodeList`. Eine `NodeList` hat genau zwei Methoden:

```
int getLength()
Node item(int index)
```

Dieses ist insofern schade, da somit nicht die neue *for*-Schleife aus Java 1.5 für die Knotenliste des *dom* benutzt werden kann.

Einen Parser für XML Wir benötigen einen Parser, der uns die Baumstruktur eines XML-Dokuments erzeugt. In der Javabibliothek ist ein solcher Parser integriert, allerdings nur über seine Schnittstellenbeschreibung. Im Paket `javax.xml.parsers` gibt es nur Schnittstellen. Um einen konkreten Parser zu erlangen, bedient man sich einer Fabrikmethode: In der Schnittstelle `DocumentBuilderFactory` gibt es eine statische Methode `newInstance` und über das `DocumentBuilderFactory`-Objekt, läßt sich mit der Methode `newDocumentBuilder` ein Parser erzeugen.

Beispiel:
Wir können so eine statischen Methode zum Parsen eines XML-Dokuments schreiben:

ParseXML.java
```java
package name.panitz.domtest;

import org.w3c.dom.Document;
import javax.xml.parsers.*;
import java.io.File;

public class ParseXML {
  public static Document parseXml(String xmlFileName){
    try{
     return
      DocumentBuilderFactory
       .newInstance()
       .newDocumentBuilder()
       .parse(new File(xmlFileName));
    }catch(Exception _){return null;}
  }

  public static void main(String [] args){
   System.out.println(parseXml(args[0]));
  }
}
```

Wir können jetzt z.B. den Quelltext dieses Skripts parsen.

```
sep@linux:~/fh/prog4/examples> java -classpath classes/ name.panitz.domtest.ParseXML ../skript.xml
[#document: null]
```

Wie man sieht ist die Methode `toString` in der implementierenden Klasse der Schnittstelle `Document`, die unser Parser benutzt nicht sehr aufschlußreich.

Beispielalgorithmen auf DOM So wie wir im vorherigen Abschnitt auf unserem algebraischen Typ `XML` ein paar Methoden in Form von Besuchern geschrieben haben, können wir versuchen für das *dom*-Objekten ähnliche Methoden zu schreiben.

Zunächst zählen wir wieder alle Knoten im Dokument:

CountNodes.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;

public class CountNodes{
  static int count(Node node){
    int result = 1;
    for (Node n:new NoteListIterator(node.getChildNodes()))
      result=result+count(n);
    return result;
  }
  public static void main(String [] args){
    System.out.println(count(ParseXML.parseXml(args[0])));
  }
}
```

Hierbei haben wir in der **for**-Schleife für Objekte die die Schnittstelle `NodeList` implementieren einen Wrapper benutzt, der diese Objekte zu einem Iteratorobjekt verpackt.

NoteListIterator.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import java.util.Iterator;

public class NoteListIterator implements Iterator<Node>
                                       , Iterable<Node>{
  NodeList nodes;
  int current=-1;
  public NoteListIterator(NodeList n){nodes=n;}

  public Node next(){
    current=current+1; return nodes.item(current);}

  public boolean hasNext(){return current+1<nodes.getLength();}

  public void remove(){
    throw new UnsupportedOperationException();
  }

  public Iterator<Node> iterator(){return this;}
}
```

Wir können uns zum Beispiel die Anzahl der Knoten in diesem Skript ausgeben lassen:

```
sep@linux:~/fh/prog4> java -classpath classes/ name.panitz.domtest.CountNodes skript.xml
1316
```


Als einen weiteren Algorithmus können wir wieder die maximale Pfadlänge berechnen lassen:

DomDepth.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;

public class DomDepth{
  static int depth(Node node){
    int result = 0;
    for (Node n:new NoteListIterator(node.getChildNodes())){
      final int currentDepth = depth(n);
      if (result<currentDepth) result=currentDepth;
    }
    return result+1;
  }
  public static void main(String [] args){
    System.out.println(depth(ParseXML.parseXml(args[0])));
  }
}
```

Auch dieses läßt sich wunderbar mit dem Quelltext dieses Skriptes testen.

```
sep@linux:~/fh/prog4> java -classpath classes/ name.panitz.domtest.DomDepth skript.xml
13
```

Dom als Baummodell für JTree XML-Dokumente sind Bäume. In Swing gibt es eine Klasse, die es erlaubt Baumstrukturen darzustellen: `JTree`. Mit Hilfe der Klasse `JTree` läßt sich auf einfache Weise eine graphische Darstellung für ein XML-Dokument erzeugen. Hierzu sind `Node`-Objekte so zu kapsen, daß die Schnittstelle `javax.swing.tree.TreeNode` implementiert wird. Hierzu sind sechs Methoden zu implementieren, die sich aber alle als in wenigen Zeile umsetzbar darstellen:

DomTreeNode.java

```
package name.panitz.domtest;

import name.panitz.crempel.util.FromTo;
import java.util.Enumeration;
import java.util.Iterator;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Element;
import org.w3c.dom.CharacterData;
import javax.swing.tree.*;
import javax.swing.*;

public class DomTreeNode implements TreeNode{
  Node node;
  public DomTreeNode(Node n){node=n;}
```

```java

  public TreeNode getChildAt(int childIndex){
    return new DomTreeNode(node.getChildNodes().item(childIndex));
  }

  public int getChildCount(){
    return node.getChildNodes().getLength();}
  public boolean isLeaf(){
    return node.getChildNodes().getLength()==0;}
  public TreeNode getParent(){
    return new DomTreeNode(node.getParentNode());}
  public boolean getAllowsChildren(){
    return (node instanceof Element);}
  public int getIndex(TreeNode n){
    final NodeList children = node.getChildNodes();
    for (Integer index : new FromTo(0,children.getLength()-1)){
      if (children.item(index).equals(n)) return index;
    }
    return -1;
  }
  public Enumeration<Node>children(){
    return new Enumeration<Node>(){
      final Iterator<Node> it
         = new NoteListIterator(node.getChildNodes());
      public boolean hasMoreElements(){return it.hasNext();}
      public Node nextElement() {return it.next();}
    };
  }

  public String toString(){
    if (node instanceof CharacterData) return node.getNodeValue();
    return node.getNodeName();}

  public static void main(String [] args){
    JFrame f = new JFrame(args[0]);
    f.getContentPane()
     .add(
       new JTree(new DomTreeNode(ParseXML.parseXml(args[0]))));
    f.pack();
    f.setVisible(true);
  }
}
```

Wie man sieht, verlangt die Schnittstelle leider noch immer die veraltete Schnittstelle `Enumeration` als Rückgabetyp der Methode `children`.

Ein Beispielaufruf dieser Klasse mit diesem Skript als Eingabe ergibt die in Abbildung 3.2 zu sehende Anzeige. Wie man sehr schön an der Anzeige sieht, wird Weißraum erhalten und es finden sich eine ganze Reihe von Textknoten mit leerem Zwischenraum in der Anzeige.

Abbildung 3.2: Anzeige des XML Dokumentes dieses Skriptes als JTree.

Manuelles Manipulieren von Dom Das DOM Api ermöglicht nicht nur in einem XML-Baum beliebig zu navigieren, sondern auch diesen Baum zu manipulieren. Es lassen sich neue Knoten einhängen und bestehende knoten löschen. Hierzu stehen in der Schnittstelle `Node` entsprechende Methoden zur Verfügung:

```java
Node appendChild(Node newChild)      throws DOMException;
Node insertBefore(Node newChild,Node refChild)
                                      throws DOMException;
Node replaceChild(Node newChild,Node oldChild)
                                      throws DOMException;
Node removeChild (Node oldChild)      throws DOMException;
```

Speziellere Methoden zum Manipulieren der verschieden Baumknoten finden sich in den Unterschnittstellen von `Node`.

Zum Erzeugen eines neuen Knotens ist es notwendig den Dokumentknoten des zu manipulierenden Knotens zu kennen. Der Dokumentknoten eines Knotens läßt sich über die Methode `Document getOwnerDocument()` erfragen. Hier gibt es dann Methoden zur Erzeugung neuer Knoten:

```
Attr createAttribute(String name);
Attr createAttributeNS(String namespaceURI, String qualifiedName);
CDATASection createCDATASection(String data);
Comment createComment(String data);
DocumentFragment createDocumentFragment();
Element createElement(String tagName);
Element createElementNS
                   (String namespaceURI, String qualifiedName);
EntityReference createEntityReference(String name);
ProcessingInstruction createProcessingInstruction
                                  (String target, String data);
Text createTextNode(String data);
```

SAX

Oft brauchen wir nie das komplette XML-Dokument als Baum im Speicher. Eine Großzahl der Anwendungen auf XML-Dokumenten geht einmal das Dokument durch, um irgendwelche Informationen darin zu finden, oder ein Ergebnis zu erzeugen. Hierzu reicht es aus, immer nur einen kleinen Teil des Dokuments zu betrachten. Und tatsächlich hätte diese Vorgehensweise, bai allen bisher geschriebenen Programmen gereicht. Wir sind nie im Baum hin und her gegangen. Wir sind nie von einem Knoten zu seinem Elternknoten oder seinen vor ihm liegenden Geschwistern gegangen.

Ausgehend von dieser Beobachtuung hat eine Gruppe von Programmierern ein API zur Bearbeitungen von XML-Dokumenten vorgeschlagen, das nie das gesammte Dokument im Speicher zu halten braucht. Dieses API heißt *SAX*, für *simple api for xml processing*. *SAX* ist keine Empfehlung des W3C. Es ist außerhalb des W3C entstanden.

Die Idee von *SAX* ist ungefähr die, daß uns jemand das Dokument vorliest, einmal von Anfang bis Ende. Wir können dann auf das gehörte reagieren. Hierzu ist für einen Parse mit einem SAX-Parser stets mit anzugeben, wie auf das Vorgelesene reagiert werden soll. Dieses ist ein Objekt der Klasse `DefaultHandler`. In einem solchen *handler* sind Methoden auszuprogrammieren, in denen spezifiziert ist, was gemacht werden soll, wenn ein Elementstarttag, Elementendtag, Textknoten etc. vorgelsen wird. Man spricht bei einem SAX-Parser von einem ereignisbasierten Parser. Wir reagieren auf bestimmte Ereignisse des Parses, nämlich dem Starten/Enden von Elementen und so weiter.

Instanziieren eines SAX-Parsers Auch ein SAX-Parser liegt in Java nur als Schnittstelle vor und kann nur über eine statische Fabrikmethode instanziert werden.

SaxParse.java

```
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
import org.xml.sax.*;
```



```
import java.io.File;

public class SaxParse{
  public static void parse(File file,DefaultHandler handler)
                                 throws Exception{
    SAXParserFactory.newInstance()
                    .newSAXParser()
                    .parse(file,handler);
  }
}
```

Zählen von Knoten Als erstes Beispiel wollen wir unser altbekanntes Zählen der Knoten programmieren. Hierzu ist ein eigener `DefaultHandler` zu schreiben, der, sobald beim Vorlesen ihm der Beginn eines Elements gemeldet wird, darauf reagiert, indem er seinen Zähler um eins weiterzählt. Wir überschreiben demnach genau eine Methode aus dem `DefaultHandler`, nämlich die Methode `startElement`:

SaxCountNodes.java

```
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;
import org.xml.sax.*;

public class SaxCountNodes extends DefaultHandler{
  public int result = 0;
  public void startElement
   (String uri, String localName
    , String qName, Attributes attributes)
                           throws SAXException {
    result=result+1;
  }

  public static void main(String [] args) throws Exception{
    SaxCountNodes counter = new SaxCountNodes();
    SaxParse.parse(new java.io.File(args[0]),counter);
    System.out.println(counter.result);
  }
}
```

Selektion von Code-Knoten In einem nächsten Beispiel für einen Handler, schreiben wir einen Handler, der bestimmte Knoten selektiert und in einer Ergebnisliste sammelt.

Wir wollen die `code` Knoten aus diesem Skript selektieren.

Hierzu können wir als algebraischen Datentypen einfach eine Klasse vorsehen, die ein Codefragment aus dem Skript darstellt. Dieses hat einen Programmnamen, ein Paketnamen und schließlich den darin enthaltenen Code.

CodeFragment.adt

```
package name.panitz.saxtest;
data class CodeFragment {
```

```java
  CodeFragment(String progName,String packageName,String code);
}
```

Der entsprechende Handler sammelt die benötigte Information auf.

SelectNodes.java

```java
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;
import org.xml.sax.*;

public class SelectNodes extends DefaultHandler{
  final List<String> names;
  final List<CodeFragment> result = new ArrayList<CodeFragment>();

  public SelectNodes(List<String> ls){names=ls;}
  public SelectNodes(String ls){
    names=new ArrayList<String>(); names.add(ls);}

  private StringBuffer currentCode = new StringBuffer();
  private String currentProgName = "";
  private String currentPackageName = "";

  public void startElement
   (String uri, String localName, String qName, Attributes attributes)
               throws SAXException {
     if (names.contains(qName)){
       currentCode = new StringBuffer();
       currentProgName = attributes.getValue("class");
       currentPackageName = attributes.getValue("package");
     }
  }

  public void endElement
    (String uri, String localName, String qName)
               throws SAXException {
     if (names.contains(qName)){
       result.add(
         new CodeFragment
            (currentProgName
            ,currentPackageName
            ,currentCode.toString()));
     }
  }

  public void characters(char[] ch,int start,int length)
               throws SAXException {
    currentCode.append(ch,start,length);
  }
}
```


Den obigen Handler können wir jetzt z.B. benutzen, um aus dem Quelltext dieses Skriptes bestimmte Beispielklassen zu extrahieren und in eine Javadatei zu speichern.

GetCode.java

```java
package name.panitz.saxtest;

import javax.xml.parsers.*;
import java.io.*;

public class GetCode {
  public static void main(String [] args) throws Exception{
    final  SelectNodes selectCodeHandler = new SelectNodes("code");

    SaxParse.parse(new File(args[0]),selectCodeHandler);
    final Writer out = new FileWriter(args[1]+".java");

    for (CodeFragment cf:selectCodeHandler.result){
      if (args[1].equals(cf.getProgName()))
        out.write(cf.getCode());
    }
    out.flush();
    out.close();
  }
}
```

Als abschließendes Beispiel für SAX können wir den SAX-Parser benutzen um einen Baum unseren algebraischen Datentypens für XML zu erzeugen. Hierzu lassen wir uns von dem Parser das Dokumentvorlesen. Wir haben zwei Keller (*stacks*). Einen auf den das oberste Element die Kinder des zuletzt geöffneten XML-Elements sind, und einer auf dem das oberste die Attribute für dieses sind. Beim schließenden Tag eines Elements wird dann mit den oberen Kellerelementen das neue Element gebaut.

SaxToXML.java

```java
package name.panitz.xml;

import name.panitz.crempel.util.FromTo;
import name.panitz.saxtest.*;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;
import org.xml.sax.*;

public class SaxToXML extends DefaultHandler{
  Stack<List<XML>> children = new Stack<List<XML>>();
  Stack<List<Attribute>> attributes = new Stack<List<Attribute>>();
  public SaxToXML(){children.push(new ArrayList<XML>());}

  public void startElement
   (String uri, String localName, String qName, Attributes attrs)
               throws SAXException {
    children.push(new ArrayList<XML>());
```

```
    ArrayList<Attribute> myAttrs = new ArrayList<Attribute>();
    for (int i:new FromTo(0,attrs.getLength()-1))
      myAttrs.add(
         new Attribute
          (new Name(attrs.getLocalName(i)
                   ,attrs.getQName(i)
                   ,attrs.getURI(i))
          ,attrs.getValue(i)));
    attributes.push(myAttrs);
  }

  public void endElement
    (String uri, String localName, String qName)
                throws SAXException {
    List<XML> chds = children.pop();
    children.peek().add(new Element(new Name(localName,qName,uri)
                                   ,attributes.pop()
                                   ,chds));
  }

  public void processingInstruction(String target,String data)
                         throws SAXException {

  }

  public void characters(char[] ch,int start,int length)
                throws SAXException {
    String text = new String(ch,start,length);
    children.peek().add(
     new Text(text.replace("<","&lt;")
                  .replace(">","&gt;")
                  .replace("&","&amp;")
                  .replace("\"","&quot;")
                  .replace("'","&apos;")
    ));

  }

  public static void main(String [] args) throws Exception{
    SaxToXML toXML = new SaxToXML();
    SaxParse.parse(new java.io.File(args[0]),toXML);
    XML doc = toXML.children.pop().get(0);
    System.out.println(doc.visit(new ToHTML()));
  }
}
```

Wie man insgesamt sehen kann, ist das Vorgehen zum Schreiben eines *handlers* in SAX ähnlich zum Schreiben eines Besuchers unseres algebraischen Typens für XML-Dokumente.

3.4 Transformationen und Queries

Dieses Kapitel beschäftigt sich mit Sprachen, die darauf zugeschnitten sind, XML-Dokumente zu verarbeiten.

3.4.1 XPath: Pfade in Dokumenten

Wir haben XML-Dokumente als Bäume betrachtet. Eine der fundamentalen Konzepten in einem Baum, sind die Pfade innerhalb eines Baumes. So kann zum Beispiel jeder Knoten eines Baumes eindeutig mit einem Pfad beschrieben werden. Zum Beschreiben von Pfaden innerhalb eines XML existiert eine Sprache, die als Empfehlung des W3C vorliegt, XPath[CD99]. Eine sehr schöne formale Beschreibung einer denotationalen Semantik für XPath findet sich in [Wad00].

XPath lehnt sich syntaktisch an eine Beschreibungssprache für Pfade in Bäumen an, die sehr verbreitet ist, nämlich Pfadbeschreibungen im Dateibaum. Um eine Datei in einem Dateisystem zu adressieren geben wir einen Pfad von der Wurzel des Dateibaums bis zu dieser Datei an. Die Datei dieses Skriptes findet sich z.B. in folgenden Pfad auf meinem Rechner:
`/home/sep/fh/prog4/skript.xml`

Ein Schrägstrich bedeutet in dieser Syntax, es wird nach diesem Strich sich auf die Kinder des vor dem Strich spezifizierten Knotens bezogen. Ist vor dem Strich nichts spezifiziert, so beginnen wir mit der Wurzel des Baums. Nach einem Schrägstrich kann man die gewünschten Kinderknoten selektieren. Bei der Pfadangabe eines Dateibaums, spezifizieren wir die Knoten über die Datei- und Ordnernamen.

XPath greift diese Idee auf. Hier bezeichnen die Namen die Tagnamen der Elemente. Im vorliegenden Dokument läßt sich so der folgende Pfad beschreiben:
`/skript/kapitel/section/subsection/code`.

Damit sind alle Elementknoten dieses Dokuments gemeint, deren Tagname `code`, die ein Elternknoten `subsection` haben, deren Elternlnoten den Tagnamen `section` haben, deren Eltern `kapitel`, deren Eltern das Wurzelelement `skript` ist.

Wie man sieht, bezeichnen XPath-Ausdrücke in der Regel Mengen von Dokumentknoten.

XPath Pfade lassen sich in zwei Weisen benutzen:

- Zum Selektieren bestimmter Knoten in einem Dokument. Ausgehend von einem Wurzelknoten werden anhand des Ausdrucks aus dem Baum bestimmte Knoten selektiert.
- Zum Prüfen, ob ein Knoten die in dem Pfad beschriebene Eigenschaft hat. Ausgehend von einem Knoten wird anhand des Ausdrucks getestet, ob er diese beschriebene Pfadeigenschaft hat.

Beide Arten, XPath-Ausdrücke zu verwenden sind in der Praxis üblich. Beispiele hierfür werden wir in den nächsten Abschnitten sehen.

Um XPath möglichst gut kennenzulernen, wird im folgenden ein kleiner Prozessor implementiert, der für einen XPath-Ausdruck die Menge der von ihm beschriebenen Knoten ausgehend von einem Knoten selektiert.

Achsen

Eines der wichtigsten Konzepten von XPath sind die sogenannten Achsen. Sie beschreiben bestimmte Arten sich in einem Baum zu bewegen. XPath kennt 13 Achsen. Die am häufigste verwendete Achse ist, die Kinderachse. Sie beschreibt die Menge aller Kinder eines Knotens. Entsprechend gibt es die Elternachse, die den Elternknoten beschreibt. Eine sehr simple Achse beschreibt den Knoten selbst. Achsen für Vorfahren und Nachkommen sind der transitive Abschluß der Eltern- bzw. Kinderachse. Zusätzlich gibt es noch Geschwisterachsen, eine für die Geschwister, die vor dem aktuellen Knoten stehen und einmal für die Geschwister, die nach dem aktuellen Knoten stehen. Eigene Achsen stehen für Attribute und Namensraumdeklarationen zur Verfügung.

Mit den seit Java 1.5 zur Verfügung stehenden Aufzählungstypen, läßt sich in Java einfach ein Typ der alle 13 Achsen beschreibt implementieren.

AxisType.java

```
package name.panitz.xml.xpath;
public enum AxisType
  {self
  ,child
  ,descendant
  ,descendant_or_self
  ,parent
  ,ancestor
  ,ancestor_or_self
  ,following_sibling
  ,following
  ,preceding_sibling
  ,preceding
  ,namespace
  ,attribute
}
```

Für jede dieser Achsen geben wir eine Implementierung auf DOM. Eine Achsenimplementierung entspricht dabei einer Funktion, die für einen Knoten eine Liste von Knoten zurückgibt.

Achsenhilfsklasse Wir schreiben eine Klasse mit statischen Methoden zur Berechnung der Achsen. Da in DOM nicht die Listen aus `java.util` benutzt werden sondern die Schnittstelle `org.w3c.dom.Nodelist`, schreiben wir zunächst eine kleine Methode, die eine `NodeList` in eine `java.util.List<Node>` umwandelt.

Axes.java

```
package name.panitz.xml.xpath;
import name.panitz.crempel.util.FromTo;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import name.panitz.domtest.*;

public class Axes{
```



```
  public static List<Node> nodelistToList (NodeList nl){
    List<Node> result = new ArrayList<Node>();
    for (int i:new FromTo(0,nl.getLength()-1)){
      result.add(nl.item(i));
    }
    return result;
  }
```

self Die einfachste Achse liefert genau den Knoten selbst. Die entsprechende Methode liefert die einelementige Liste des Eingabeknotens:

Axes.java

```
  public static List<Node> self(Node n){
    List<Node> result = new ArrayList<Node>();
    result.add(n);
    return result;
  }
```

child Die gebräuchlichste Achse beschreibt die Menge aller Kinderknoten. Wir lassen uns die Kinder des DOM Knotens geben und konvertieren die `NodeList` zu einer Javaliste:

Axes.java

```
  public static List<Node> child(Node n){
    return nodelistToList(n.getChildNodes());
  }
```

descendant Die Nachkommen sind Kinder und deren Nachkommen. Für fügen jedes Kind zur Ergebnisliste hinzu, sowie alle deren Nachkommen:

Axes.java

```
  public static List<Node> descendant(Node n){
    List<Node> result = new ArrayList<Node>();
    for (Node child:child(n)){
      result.add(child);
      result.addAll(descendant(child));
    }
    return result;
  }
```

descendant-or-self In der Nachkommenachse taucht der Knoten selbst nicht auf. Diese Achse fügt den Knoten selbst zusätzlich ans Ergebnis an:

Axes.java

```
  public static List<Node> descendant_or_self(Node n){
    List<Node> result = self(n);
    result.addAll(descendant(n));
    return result;
  }
```

parent Die Kinderachse lief in den Baum Richtung Blätter eine Ebene nach unten, die Elternachse läuft diese eine Ebene Richtung Wurzel nach oben:

Axes.java
```
public static List<Node> parent(Node n){
   List<Node> result = new ArrayList<Node>();
   result.add(n.getParentNode());
   return result;
}
```

ancestor Die Vorfahrenachse ist der transitive Abschluß über die Elternachse, sie bezeichnet also die Eltern und deren Vorfahren:

Axes.java
```
public static List<Node> ancestor(Node n){
  List<Node> result = new ArrayList<Node>();
  for (Node parent:parent(n)){
    if (parent!=null){
      result.addAll(ancestor(parent));
      result.add(parent);
    }
  }
  return result;
}
```

ancestor-or-self Die Vorfahrenachse enthält nicht den Knoten selbst. Entsprechend wie auf der Nachkommenachse gibt es auch hier eine Version, die den Knoten selbst enthält:

Axes.java
```
public static List<Node> ancestor_or_self(Node n){
  List<Node> result = ancestor(n);
  result.addAll(self(n));
  return result;
}
```

following-sibling Diese Achse beschreibt die nächsten Geschwister. Wir erhalten diese, indem wir den Elternknoten holen, durch dessen Kinder iterieren und nachdem wir an unseren aktuellen Knoten gelangt sind, anfangen diese Kinderknoten ins Ergebnis aufzunehmen.

Axes.java
```
public static List<Node> following_sibling(Node n){
  List<Node> result = new ArrayList<Node>();
  int nodeType = n.getNodeType();
  boolean take = false;
  if (nodeType != Node.ATTRIBUTE_NODE){
    for (Node sibling:child(n.getParentNode())){
      if (take) result.add(sibling);
      if (sibling==n) take=true;
    }
```



```
  }
  return result;
}
```

preceding-sibling Entsprechend gibt es die Achse, die die vor dem aktuellen Knoten stehenden Geschwister extrahieren. Wir gehen ähnlich vor wie oben, sammeln jetzt jedoch die Kinder nur bis zum aktuellen Knoten auf.

Axes.java
```
public static List<Node> preceding_sibling(Node n){
  List<Node> result = new ArrayList<Node>();
  int nodeType = n.getNodeType();
  if (nodeType != Node.ATTRIBUTE_NODE){
    for (Node sibling:child(n.getParentNode())){
      if (sibling==n) break;
      result.add(sibling);
    }
  }
  return result;
}
```

following Die Nachfolgerachse bezieht sich auf die Dokumentordnung in serialisierter Form. Sie bezeichnet alle Knoten, deren Tag im gedruckten XML Dokument nach dem Ende des aktuellen Knotens beginnen. Dieses sind die nachfolgenden Geschwister und deren Nachkommen:

Axes.java
```
public static List<Node> following(Node n){
  List<Node> result = new ArrayList<Node>();
  for (Node follow_sib:following_sibling(n)){
    result.add(follow_sib);
    result.addAll(descendant(follow_sib));
  }
  return result;
}
```

preceding Analog hierzu funktioniert die Vorgängerachse. Auch sie bezieht sich auf die Dokumentordnung:

Axes.java
```
public static List<Node> preceding(Node n){
  List<Node> result = new ArrayList<Node>();
  for (Node preced_sib:preceding_sibling(n)){
    result.add(preced_sib);
    result.addAll(descendant(preced_sib));
  }
  return result;
}
```

attribute Attribute werden in einer gesonderten Achse beschrieben. Sie tauchen in keiner der vorherigen Achsen auf. Ausgenommen sind in dieser Achse die Attribute die eine Namensraumdefinition beschreiben:

Axes.java
```
  public static List<Node> attribute(Node n){
    List<Node> result = new ArrayList<Node>();
    if (n.getNodeType()==Node.ELEMENT_NODE){
      NamedNodeMap nnm = n.getAttributes();
      for (int i:new FromTo(0,nnm.getLength()-1)){
        Node current = nnm.item(i);
        if (!current.getNodeName().startsWith("xmlns"))
          result.add(current);
      }
    }
    return result;
  }
```

namespace Ebenso gesondert werden in der letzten Achse die Namensraumdefinitionen behandelt. XML-technisch sind diese Attribute, deren Attributname mit `xmlns` beginnt:

Axes.java
```
  public static List<Node> namespace(Node n){
    List<Node> result = new ArrayList<Node>();
    if (n.getNodeType()==Node.ELEMENT_NODE){
      NamedNodeMap nnm = n.getAttributes();
      for (int i:new FromTo(0,nnm.getLength()-1)){
        Node current = nnm.item(i);
        if (current.getNodeName().startsWith("xmlns"))
          result.add(current);
      }
    }
    return result;
  }
```

Achsenberechnung Für Aufzählungstypen ab Java 1.5 läßt sich eine schöne `switch`-Anweisung schreiben. So können wir eine allgemeine Methode zur Achsenberechnung schreiben, die für jede Achse die entsprechende Methode aufruft.[3]

Axes.java
```
  public static List<Node> getAxis(Node n,AxisType axis){
    return ancestor(n);
 /*   switch (axis){
      case ancestor           : return ancestor(n);
      case ancestor_or_self   : return ancestor_or_self(n);
      case attribute          : return attribute(n);
      case child              : return child(n);
```

[3]Die Version Beta1 von Java 1.5 hat einen Bug, so daß der javac Commpiler bei der folgenden Klasse während der Codegenerierung abstürzt. Daher ist der Code vorerst auskommentiert.


```
      case descendant         : return descendant(n);
      case descendant_or_self: return descendant_or_self(n);
      case following          : return following(n);
      case following_sibling  : return following_sibling(n);
      case namespace          : return namespace(n);
      case parent             : return parent(n);
      case preceding          : return preceding(n);
      case preceding_sibling  : return preceding_sibling(n);
      case self               : return self(n);
      default                 :throw new UnsupportedOperationException();
    }*/
  }
}
```

Knotentest

Die Kernausdrücke in XPath sind von der Form:

axisType`::`*nodeTest*

axisType beschreibt dabei eine der 13 Achsen. *nodeTest* ermöglicht es, aus den durch die Achse beschriebenen Knoten bestimmte Knoten zu selektieren. Syntaktisch gibt es 8 Arten des Knotentests:

- `*`: beschreibt Elementknoten mit beliebigen Namen.
- *pref*`:*`: beschreibt Elementknoten mit dem Prefix *pref* und beliebigen weiteren Namen.
- *pref*`:`*name*: beschreibt Elementknoten mit dem Prefix *pref* und den weiteren Namen *name*. Der Teil vor den Namen kann dabei auch fehlen.
- `comment()`: beschreibt Kommentarknoten.
- `text()`: beschreibt Textknoten.
- `processing-instruction()`: beschreibt Processing-Instruction-Knoten.
- `processing-instruction(`*target*`)`: beschreibt Processing-Instruction-Knoten mit einem bestimmten Ziel.
- `node()`: beschreibt beliebige Knoten.

Algebraischer Typ für Knotentests Wir können die acht verschiedene Knotentests durch einen algebraischen Typ ausdrücken:

NodeTest.adt
```
package name.panitz.xml.xpath;
import java.util.List;

data class NodeTest{
  StarTest();
```

```
  PrefixStar(String prefix);
  QName(String prefix,String name);
  IsComment();
  IsText();
  IsProcessingInstruction();
  IsNamedProcessingInstruction(String name);
  IsNode();
}
```

Textuelle Darstellung für Knotentests Für diesen algebraischen Typ läßt sich ein einfacher Besucher zur textuellen Darstellung des Typs schreiben. Er erzeugt für die einzelnen Knotentests die Syntax wie sie in XPath-Ausdrücken vorkommt.

ShowNodeTest.java

```
package name.panitz.xml.xpath;

public class ShowNodeTest extends NodeTestVisitor<String> {
  public String eval(StarTest _){return "*";}
  public String eval(PrefixStar e){return e.getPrefix()+":*";}
  public String eval(QName e){String result="";
    if (e.getPrefix().length()>0) result=e.getPrefix()+":";
    result=result+e.getName();
    return result;
  }
  public String eval(IsComment _){return "comment()";}
  public String eval(IsText _){return "text()";}
  public String eval(IsProcessingInstruction _){
    return "processing-instruction()";}
  public String eval(IsNamedProcessingInstruction e){
    return "processing-instruction("+e.getName()+")";
  }
  public String eval(IsNode _){return "node()";}
}
```

Auswerten von Knotentests Ein Knotentest ergibt für einen konkreten Knoten entweder `true` oder `false`. Hierfür können wir einen Besucher schreiben, der für einen Knotentest und einen konkreten Knoten diesen bool'schen Wert berechnet.

DoNodeTest.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import static org.w3c.dom.Node.*;
import org.w3c.dom.*;

public class DoNodeTest extends NodeTestVisitor<Boolean> {
  Node current;
  public DoNodeTest(Node n){current=n;}
```


Der Test auf einen Stern ist wahr für jedes Element oder Attribut:

DoNodeTest.java

```
  public Boolean eval(StarTest _){
    return    current.getNodeType()==ELEMENT_NODE
           || current.getNodeType()==ATTRIBUTE_NODE;
  }
```

Der Test auf ein bestimmtes Prefix ist wahr für jeden Knoten, der diesen Prefix hat:

DoNodeTest.java

```
  public Boolean eval(PrefixStar e){
    String currentPrefix = current.getPrefix();
    if (currentPrefix==null) currentPrefix="";
    return   new StarTest().visit(this)
         && currentPrefix.equals(e.getPrefix());
  }
```

Jeder Test nach einen qualifizierten Namen ist wahr, wenn der Knoten diesen Namen hat:

DoNodeTest.java

```
  public Boolean eval(QName e){
    return   new PrefixStar(e.getPrefix()).visit(this)
         &&  current.getNodeName().equals(e.getName());
  }
```

Der Test nach Kommentaren ist für Kommantarknoten wahr:

DoNodeTest.java

```
  public Boolean eval(IsComment _){
    return current.getNodeType()==COMMENT_NODE;}
```

Der Test nach Text ist für Textknoten wahr:

DoNodeTest.java

```
  public Boolean eval(IsText _){
    return current.getNodeType()==TEXT_NODE;}
```

Der Test nach Processing-Instruction ist für PI-Knoten wahr:

DoNodeTest.java

```
  public Boolean eval(IsProcessingInstruction _){
    return current.getNodeType()==PROCESSING_INSTRUCTION_NODE;}
```

Der Test nach Processing-Instruction mit bestimmten Namen ist für PI-Knoten mit diesen Namen wahr:

DoNodeTest.java

```
  public Boolean eval(IsNamedProcessingInstruction e){
    return current.getNodeType()==PROCESSING_INSTRUCTION_NODE
        && e.getName().equals(current.getNodeName());
  }
```

Der Test nach Knoten ist immer wahr:

DoNodeTest.java
```
  public Boolean eval(IsNode _){return true;}
```

Einen großen Teil eines XPath-Prozessors haben wir damit schon implmentiert. Wir können uns die Knoten einer Achse geben lassen und wir können diese Knoten auf einen Knotentest hin prüfen. Zusammen läßt sich damit bereits eine Methode zu Auswertung eines Kernausdrucks mit Achse und Knotentest schreiben. Hierzu iterieren wir über die Liste der durch die Achse spezifizierten Knoten und fügen diese bei positiven Knotentest dem Ergebnis zu:

DoNodeTest.java
```
  static public List<Node> evalAxisExpr
        (AxisType axis,NodeTest test,Node context){
    List<Node> result = new ArrayList<Node>();
    for (Node node:Axes.getAxis(context,axis)){
      if (test.visit(new DoNodeTest(node)))
        result.add(node);
    }
    return result;
  }
}
```

Pfadangaben

In obiger Einführung haben wir bereits gesehen, daß XPath den aus dem Dateissystem bekannten Schrägstrich für Pfadangaben benutzt. Betrachten wir XPath-Ausdrücke als Terme, so stellt der Schrägstrich einen Operator dar. Diesen Operator gibt es sowohl einstellig wie auch zweistellig.

Die Grundsyntax in XPath sind eine durch Schrägstrich getrennte Folge von Kernausdrücke mit Achsentyp und Knotentest.

> **Beispiel:**
> Der Ausdruck
>
> `child::skript/child::*/descendant::node()/self::code/attribute::class`
>
> beschreibt die Attribute mit Attributnamen `class`, die an einem Elementknoten mit Tagnamen `code` hängen, die Nachkommen eines beliebigen Elementknotens sind, die Kind eines Elementknotens mit Tagnamen `skript` sind, die Kind des aktuellen Knotens, auf dem der Ausdruck angewendet werden soll sind.
>
> Vorwärts gelesen ist dieser Ausdruck eine Selektionsanweisung:
> Nehme alle `skript`-Kinder des aktuellen Knotens. Nehme von diesen beliebige Kinder. Nehme von diesen alle `code`-Nachkommen. Und nehme von diesen jeweils alle `class`-Attribute.

Der einstellige Schrägstrichoperator bezieht sich auf die Dokumentwurzel des aktuellen Knotens.

Abkürzende Pfadangaben

XPath kennt einen zweiten Pfadoperator `//`. Auch er existiert jeweils einmal einstellig und einmal zweistellig. Der doppelte Schrägstrich ist eine abkürzende Schreibweise, die übersetzt werden kann in Pfade mit einfachen Schrägstrichoperator.

- `//`*expr* Betrachte beliebige Knoten unterhalt des Dokumentknotens, die durch *expr* charakterisiert werden.
- e_1`//`e_2 Betrachte beliebiege Knoten unterhalb der durch e_1 charkterisierten Knoten.und prüfe diese auf e_2.

Übersetzung in Kernsyntax Der Doppelschrägstrich ist eine abkürzende Schreibweise für: `/descendant-or-self::node()/`

Weitere abkürzende Schreibweisen

In XPath gibt es weitere abkürzende Schreibweisen, die auf die Kernsyntax abgebildet werden können.

der einfache Punkt Für die Selbstachse kann als abkürzende Schreibweise ein einfacher Punkt`.` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `.` ist die abkürzende Schreibweise für: `self::node()`.

der doppelte Punkt Für die Elternachse kann als abkürzende Schreibweise ein doppelter Punkt`..` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `..` ist die abkürzende Schreibweise für: `parent::node()`.

implizite Kinderachse Ein Kernausdruck, der Form `child::`*nodeTest* kann abgekürzt werden durch *nodeTest*. Die Kinderachse ist also der Standardfall.

Attributselektion Auch für die Attributachse gibt es eine abkürzende Schreibweise: ein Kernausdruck, der Form `attribute::`*pre*`:`*name* kann abgekürzt werden durch `@`*pre*`:`*name*.

> **Beispiel:**
> Insgesamt läßt sich der obige Ausdruck abkürzen zu: `skript/*//code/@class`

Vereinigung

In XPath gibt es ein Konstrukt, das die Vereinigung zweier durch einen XPath-Ausdruck beschriebener Listen beschreibt. Syntaktisch wird dieses durch einen senkrechten Strich `|` ausgedrückt.

Beispiel:
Der Audruck `/skript/kapitel | /skript/anhang/kapitel` beschreibt die `kapitel`-Elemente die unter top-level Knoten `skript` hängen oder die unter einen Knoten `anhang`, der unter dem `skript`-Element hängt.

Funktionen und Operatoren

XPath kommt mit einer großen Zahl eingebauter Funktionen und Operatoren. Diese sind in einer eigenen Empfehlung des W3C spezifiziert[?]. Sie können auf Mengen von Knoten aber auch auf Zahlen, Strings und bool'schen Werten definiert sein. So gibt es z.B. die Funktion `count`, die für eine Knotenliste die Anzahl der darin enthaltenen Knoten angibt.

Beispiel:
Der Ausdruck `count(//code)` gibt die Anzahl der in einem Dokument enthaltenen `code`-Elemente zurück.

Literale

Für die Rechnung mit Funktionen und Operatorn gibt es Literalen für Strings und Zahlen in XPath.

Beispiel:
Der Ausdruck `count(//kapitel)=5` wertet zu `true` aus, wenn des Dokument genau fünf `kapitel`-Elemente hat.

Qualifizierung

Bisher haben wir gesehen, wie entlang der Achsen Mengen von Teildokumenten von XML-Dokementen selektiert werden können. XPath sieht zusätzlich vor, durch ein Prädikat über diese Menge zu filtern. Ein solches Prädikat gibt dabei an, ob ein solcher Knoten in der Liste gewünscht ist oder nicht. Man nennt Ausdrücke mit einem Prädikat auch qulifizierte Ausdrücke.

Syntaktisch stehen die Prädikate eines XPath-Ausdrucks in eckigen Klammern hinter den Ausdruck. Das Prädikat ist selbst wieder ein XPath-Ausdruck. Die Auswertung eines qualifizierten XPath-Ausdrucks funktioniert nach folgender Regel:

> Berechne die Ergebnisliste des Ausdrucks. Für jedes Element dieser Liste als aktuellen Kontextknoten berechne das Ergebnis des Prädikats. Interpretiere dieses Ergebnis als bool'schen Wert und verwerfe entweder den Kontextknoten oder nimm ihn in das Ergebnis aus.

Je nachdem was das Prädikat für ein Ergebnis hat, wird es als wahr oder falsch interpretiert. Diese Interpretation ist relativ pragmatisch. XPath war ursprünglich so konzipiert, daß es vollkommen ungetypt ist und es während der Ausführung zu keinerlei Typcheck kommt, geschweige denn ein statisches Typsyystem existiert. Daher wird bei der Anwendung von Funktionen und Operatoren als auch bei der Auswertung eines Prädikats versucht, jeden Wert als jeden beliebige Typ zu interpretieren.

bool'sches Ergebnis Prädikate die direkt durch ein Funktionsergebnis oder eine Operatoranwendung einen bool'schen Wert als Ergebnis haben, können direkt als Prädikat interpretiert werden.

Beispiel:
Der XPath-Ausdruck `//code[@lang="hs"]` selektiert alle Code-Knoten des Dokuments, die ein Attribut `lang` mit dem Wert `hs` haben.

leere oder nichtleere Ergebnisliste Wenn der XPath-Ausdruck eines Prädikats zu einer Liste auswertet, dann wird eine leere Liste als der bool'sche Wert `false` ansonsten als `true` interpretiert.

Beispiel:
Der XPath-Ausdruck `//example[.//code]` selektiert alle `example`-Knoten des Dokuments, die mindestens einen `code`-Nachkommen haben. Also alle Beispiele, in denen ein Programm vorkommt.

eine Zahl als Ergebnis Über Funktionen kann ein XPath-Ausdruck auch eine Zahl als Ergebnis haben. In dem Fall, daß ein XPath-Ausdruck zu einer Zahl auswertet, wird ein Knoten selektiert, wenn er an der Stelle dieser Zahl in der Liste des qualifizierten Ausdrucks ist.

Beispiel:
Der Ausdruck `/skript/kapitel[3]` selektiert jeweils das dritte `kapitel`-Element innerhalb eines `skript`-Elements.

Klammerung

Schließlich kennt XPath noch die Klammerung von Teilausdrücken.

Beispiel:
Der Ausdruck `./skript/(./kapitel | ./anhang)` bezeichnet alle `kapitel`- oder `anhang`-Elemente ausgehend vom `skript`-Kind des Kontextknotens. Hingegen `./skript/kapitel | ./anhang` bezeichnet die `kapitel`-Elemente unter dem `skript`-Kinder des Kontextknotens oder `anhang`-Kinder des Kontextknotens.

Algebraischer Typ für XPath-Ausdrücke

In diesem Abschnitt wollen wir einmal versuchen einen eigenen zumindest rudimentären XPath Prozessor zu schreiben. Ein XPath Prozessor selektiert ausgehend von einem aktuellen Knoten anhand eines gegebenen XPath Ausdrucks eine Liste von Teildokumenten. Zunächst brauchen wir einen Typ, der XPath-Ausdrücke beschreiben kann. Wir können dieses in einem algebraischen Tyen einfach zusammenfassen. Für jedes XPath Konstrukt gibt es einen eigenen Konstruktor:

XPath.adt

```
package name.panitz.xml.xpath;
import java.util.List;

data class XPath {
  Axis(AxisType type,NodeTest nodeTest);
  RootSlash(XPath expr);
  RootSlashSlash(XPath expr);
  Slash(XPath e1,XPath e2);
  SlashSlash(XPath e1,XPath e2);
  TagName(String name);
  AttributeName(String name);
  Dot();
  DotDot();
  NodeSelect();
  TextSelect();
  PISelect();
  CommentSelect();
  Star();
  AtStar();
  Union(XPath e1,XPath e2);
  Function(String name,List<XPath> arguments);
  BinOperator(String name, XPath e1,XPath e2);
  UnaryOperator(String name, XPath expr);
  QualifiedExpr(XPath expr,XPath qualifier);
  NumLiteral(Double value);
  StringLiteral(String value);
}
```

Die Klammerung wird in Objekten diesen algebraischen Typs durch die Baumstruktur dargestellt.

Textuelle Darystellung von XPath Ausdrücken Zunächst folgt ein Besucher, der uns ein XPath-Objekt wieder in der XPath-Syntax darstellt:

ShowXPath.java

```
package name.panitz.xml.xpath;

public class ShowXPath extends XPathVisitor<StringBuffer> {
  StringBuffer result = new StringBuffer();
  public StringBuffer eval(Axis e){
    result.append(e.getType());
    result.append("::");
    result.append(e.getNodeTest().visit(new ShowNodeTest()));
    return result;
  }
  public StringBuffer eval(RootSlash e){
    result.append("/"); e.getExpr().visit(this);
    return result;}
```



```
  public StringBuffer eval(RootSlashSlash e){
    result.append("//"); e.getExpr().visit(this);
    return result;}
  public StringBuffer eval(Slash e){
    e.getE1().visit(this);result.append("/");e.getE2().visit(this);
    return result;}
  public StringBuffer eval(SlashSlash e){
    e.getE1().visit(this);result.append("//");e.getE2().visit(this);
    return result;}
  public StringBuffer eval(TagName e){result.append(e.getName());
    return result;}
  public StringBuffer eval(AttributeName e){
    result.append("@");result.append(e.getName());
    return result;}
  public StringBuffer eval(Dot e){result.append(".");
    return result;}
  public StringBuffer eval(DotDot e){result.append("..");
    return result;}
  public StringBuffer eval(NodeSelect e){result.append("node()");
    return result;}
  public StringBuffer eval(TextSelect e){result.append("text()");
    return result;}
  public StringBuffer eval(CommentSelect e){result.append("comment()");
    return result;}
  public StringBuffer eval(PISelect e){
    result.append("processing-instruction()");
    return result;}
  public StringBuffer eval(Star e){result.append("*");
    return result;}
  public StringBuffer eval(AtStar e){result.append("@*");
    return result;}
  public StringBuffer eval(Union e){
    e.getE1().visit(this);result.append("|");e.getE2().visit(this);
    return result;}
  public StringBuffer eval(Function e){
    result.append(e.getName());result.append("(");
    boolean first = true;
    for (XPath arg:e.getArguments()){
      if (!first) result.append(",");else first=false;
      arg.visit(this);
    }
    result.append(")");
    return result;}
  public StringBuffer eval(UnaryOperator e){
    result.append(e.getName()+"\u0020");e.getExpr().visit(this);
    return result;}
  public StringBuffer eval(BinOperator e){
    e.getE1().visit(this);result.append("\u0020"+e.getName()+"\u0020");
    e.getE2().visit(this);return result;}
  public StringBuffer eval(QualifiedExpr e){
```

```
    e.getExpr().visit(this);result.append("[");
    e.getQualifier().visit(this);result.append("]");
    return result;}
  public StringBuffer eval(NumLiteral e){result.append(""+e.getValue());
    return result;}
  public StringBuffer eval(StringLiteral e){
   result.append("\"");result.append(e.getValue()) ;result.append("\"");
   return result;}
}
```

Entfernen von abkürzenden Schreibweisen

Wir wollen XPath-Ausdrücke auf Dokumentknoten anwenden. Wenn wir zunächst alle abkürzende Schreibweisen in einem XPath-Ausdruck durch ihren Kernausdruck ersetzen, so brauchen wir bei der Anwendung eines XPath-Ausdrucks nicht mehr um abgekürzte Ausdrücke kümmern. Daher schreiben wir zunächst einen Besucher, der in einem XPath-Ausdruck alle Abkürzungen löscht:

RemoveAbbreviation.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import name.panitz.domtest.*;

public class RemoveAbbreviation extends XPathVisitor<XPath> {
```

Entfernung von: //e Die komplexeste abkürzende Schreibweise ist der Doppelschrägstrich. Er wird ersetzt durch den Ausdruck: `/descendant-or-self::node()/`. Für den einstelligen `//`-Operator ergibt das den Javaausdruck:

```
new RootSlash(new Slash(new Axis(AxisType.descendant_or_self,new IsNode()),expr))
```

Wir erhalten folgende Implementierung:

RemoveAbbreviation.java

```
public XPath eval(RootSlashSlash e){
  /* //expr -> /descendant-or-self::node()/expr  */
  return
    new RootSlash(
     new Slash(new Axis(AxisType.descendant_or_self,new IsNode())
              ,e.getExpr().visit(this)));}
```

Entfernung von: e1//e2 In gleicher Weise ist der doppelte Schrägstrich als zweistelliger Operator durch den Kernausdruck `/descendant-or-self::node()/` zu ersetzen.

RemoveAbbreviation.java

```
public XPath eval(SlashSlash e){
  final XPath e1 = e.getE1();
  final XPath e2 = e.getE2();
  /* e1//e2 -> e1/descendant-or-self::node()/e2 */
  return
   new Slash
     (e1.visit(this)
     ,new Slash(new Axis(AxisType.descendant_or_self,new IsNode())
               ,e2.visit(this)));}
```

Entfernung von: . Der einfache Punkt wird durch einen Kernausdruck auf der Selbstachse ersetzt.

RemoveAbbreviation.java

```
public XPath eval(Dot e){
  return new Axis(AxisType.self,new IsNode());}
```

Entfernung von: .. Der doppelte Punkt wird durch einen Kernausdruck auf der Elternachse ersetzt.

RemoveAbbreviation.java

```
public XPath eval(DotDot e){
  return new Axis(AxisType.parent,new IsNode());}
```

Entfernung von: qname Für einen einfachen Tagnamen wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath eval(TagName e){
  return new Axis(AxisType.child,new QName("",e.getName()));}
```

Entfernung von: @attr Für einen einfachen Attributnamen wird die implizit vorhandene Attributachse eingefügt.

RemoveAbbreviation.java

```
public XPath eval(AttributeName e){
  return new Axis(AxisType.attribute,new QName("",e.getName()));}
```

Entfernung von: node() Für einen einfache Knotentest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath eval(NodeSelect e){
  return new Axis(AxisType.child,new IsNode());}
```

Entfernung von: text() Für einen einfache Texttest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java
```
public XPath eval(TextSelect e){
  return new Axis(AxisType.child,new IsText());}
```

Entfernung von: processing-instruction() Für einen einfache Processing-Instruction-Test wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java
```
public XPath eval(PISelect e){
  return new Axis(AxisType.child,new IsProcessingInstruction());}
```

Entfernung von: comment() Für einen einfache Kommentartest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java
```
public XPath eval(CommentSelect e){
  return new Axis(AxisType.child,new IsComment());}
```

Entfernung von: * Für einen einfache Sterntest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java
```
public XPath eval(Star e){
  return new Axis(AxisType.child,new StarTest());}
```

Entfernung von: @* Für einen einfache Sterntest auf Attributen wird die implizit vorhandene Attributachse eingefügt.

RemoveAbbreviation.java
```
public XPath eval(AtStar e){
  return new Axis(AxisType.attribute,new StarTest());}
```

Nichtabgekürzte Ausdrücke Für die Ausdrücke, die keine abkürzende Schreibweise darstellen, wird der Besucher in die Unterausdrücke geschickt, um in diesen abkürzende Schreibweisen zu ersetzen.

RemoveAbbreviation.java
```
public XPath eval(Slash e){
  return new Slash(e.getE1().visit(this),e.getE2().visit(this));}
public XPath eval(Axis e){return e;}
public XPath eval(RootSlash e){
  return new RootSlash(e.getExpr().visit(this));}
public XPath eval(Union e){
  return new Union(e.getE1().visit(this),e.getE2().visit(this));}
public XPath eval(Function e){
```



```
  final List<XPath> args = e.getArguments();
  List<XPath> newArgs = new ArrayList<XPath>();
  for (XPath arg:args) newArgs.add(arg.visit(this));
  return new Function(e.getName(),newArgs);}
public XPath eval(BinOperator e){
  return new BinOperator(e.getName()
                        ,e.getE1().visit(this)
                        ,e.getE2().visit(this));}
public XPath eval(UnaryOperator e){
  return new UnaryOperator(e.getName(),e.getExpr().visit(this));}
public XPath eval(QualifiedExpr e){
  return new QualifiedExpr(e.getExpr().visit(this)
                          ,e.getQualifier().visit(this));}
public XPath eval(NumLiteral e){return e;}
public XPath eval(StringLiteral e){return e;}
}
```

Auswerten von XPath Ausdrücken

Wir haben nun alles beisammen, um einen XPath-Prozessor zu definieren.

Auswertungsergebnis Die Auswertung eines XPath-Ausdrucks kann eine Liste von Knoten, eine Zahl, ein String oder ein bool'sches Ergebnis haben. Hierfür sehen wir einen algebraischen Typ vor.

XPathResult.adt
```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import java.util.List;

data class XPathResult {
  BooleanResult(Boolean value);
  NumResult(Double value);
  StringResult(String value);
  Nodes(List<Node> value);
}
```

Auswertung Der eigentliche Prozessor wird als Besucher über einen XPath-Ausdruck geschrieben. Dieser Besucher braucht drei Informationen:

- den Kontextknoten, auf den der Ausdruck angewendet werden soll.
- die Größe der Liste, in der der Knotextknoten ist.
- die Position des Kontextknotens in der Liste, in der er sich befindet.

EvalXPath.java
```
package name.panitz.xml.xpath;

```

```
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import static org.w3c.dom.Node.*;
import name.panitz.domtest.*;

import static name.panitz.xml.xpath.Axes.*;

public class EvalXPath extends XPathVisitor<XPathResult> {
  Node current;
  int contextPosition=1;
  int contextSize=1;

  public EvalXPath(Node n){current=n;}
  public EvalXPath(Node n,int pos,int size){
    this(n);contextPosition=pos;contextSize=size;}
```

Einen Kernausdruck mit einer Achse können wir bereits auswerten. Die entsprechende Methode haben wir bereits in der Klasse `DoNodeTest` implementiert.

EvalXPath.java

```
  public XPathResult eval(Axis e){
    return new Nodes(
      DoNodeTest.evalAxisExpr(e.getType(),e.getNodeTest(),current));
  }
```

Für den einstelligen Schrägstrich ist der Dokumentknoten des Knotextsknotens zu beschaffen, um für diesen als neuen Kontextknoten den XPath-Ausdruck rechts von dem Schrägstrich anzuwenden.

EvalXPath.java

```
  public XPathResult eval(RootSlash e){
    Node doc ;
    if (current.getNodeType()==DOCUMENT_NODE) doc=current;
    else{
      doc = current.getOwnerDocument();
    }
    if (doc!=null) return e.getExpr().visit(new EvalXPath(doc));

    return new Nodes(new ArrayList<Node>());
  }
```

Für den zweistelligen Schrägstrich ist zunächst der linke Kinderausdruck zu besuchen. Falls dieses eine Knotenliste darstellt, ist für jedes Element dieser Liste als Kontextknoten der zweite Operand zu besuchen.

EvalXPath.java

```
  public XPathResult eval(Slash e){
    XPathResult res1 = e.getE1().visit(this);
```



```

    if (res1 instanceof Nodes){
      final List<Node> resultNodes = new ArrayList<Node>();
      final List<Node> e1s = ((Nodes)res1).getValue();
      final int size = e1s.size();
      int pos = 1;
      for (Node e1:e1s){
        XPathResult e2s = e.getE2().visit(new EvalXPath(e1,pos,size));
        if (e2s instanceof Nodes)
           resultNodes.addAll(((Nodes)e2s).getValue());
        else return e2s;
        pos=pos+1;
      }
      return new Nodes(resultNodes);
    }
    return res1;
  }
```

Für die Vereinigung zweier XPath-Ausdrücke, sind diese beide Auszuwerten und in Falle einer Knotenliste als Ergebnis beide Liste zu vereinigen.

EvalXPath.java

```
  public XPathResult eval(Union e){
    XPathResult r1 = e.getE1().visit(this);
    XPathResult r2 = e.getE1().visit(this);
    if (r1 instanceof Nodes && r2 instanceof Nodes){
      List<Node> resultNodes = ((Nodes)r1).getValue();
      resultNodes.addAll(((Nodes)r2).getValue());
      return new Nodes(resultNodes);
    }
    return r1;
  }
```

Ein numerisches Literal ist gerade nur der Wert dieses Literals das Ergebnis.

EvalXPath.java

```
  public XPathResult  eval(NumLiteral e){
    return new NumResult(e.getValue());}
```

Ein Stringliteral ist gerade nur der Wert dieses Literals das Ergebnis.

EvalXPath.java

```
  public XPathResult  eval(StringLiteral e){
    return new StringResult(e.getValue());}
```

Für einen qualifizierten Ausdruck ist erst der XPath-Ausdruck auszuwerten, und dann für jedes Element der Knotenliste das Prädikat. Es ist zu testen, ob dieses Prädikat als wahr oder falsch zu interpretieren ist. Hierzu wird ein Besucher auf `XPathResult` geschrieben.

EvalXPath.java

```
  public XPathResult  eval(QualifiedExpr e){
    XPathResult result1 = e.getExpr().visit(this);
    if (result1 instanceof Nodes){
      final List<Node> resultNodes = new ArrayList<Node>();
      final List<Node> rs = ((Nodes)result1).getValue();
      final int size = rs.size();
      int pos = 1;
      for (Node r : rs){
        XPathResult qs
           = e.getQualifier().visit(new EvalXPath(r,pos,size));
        if (qs.visit(new TestQualifier(r,pos,size)))
          resultNodes.add(r);
        pos=pos+1;
      }
      return new Nodes(resultNodes);
    }
    return result1;
  }
```

Die folgenden Fälle brauchen wir in der Auswertung nicht betrachten, weil wir sie der Elemination von abkürzenden Schreibweisen aus dem XPath-Baum entfernt haben

EvalXPath.java

```
  public XPathResult  eval(RootSlashSlash e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(SlashSlash e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(TagName e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(AttributeName e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(Dot e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(DotDot e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(NodeSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(TextSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(PISelect e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(CommentSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(Star e){
   throw new UnsupportedOperationException();}
  public XPathResult  eval(AtStar e){
   throw new UnsupportedOperationException();}
```

Operatoren und Funktionen sind in unserem Prozessor noch nicht implementiert.

EvalXPath.java

```
  public XPathResult  eval(Function e){
   throw new UnsupportedOperationException();
  }
  public XPathResult  eval(BinOperator e){
   throw new UnsupportedOperationException();
  }
  public XPathResult  eval(UnaryOperator e){
   throw new UnsupportedOperationException();
  }
```

Es folgen ein paar erste Beispielaufrufe:

EvalXPath.java

```
  public static XPathResult eval(XPath xpath,Node n){
    System.out.println(xpath.visit(new NormalizeXPath())
                            .visit(new ShowXPath()));

    System.out.println(xpath.visit(new NormalizeXPath())
                            .visit(new RemoveAbbreviation())
                            .visit(new ShowXPath()));

    return xpath.visit(new NormalizeXPath())
                .visit(new RemoveAbbreviation())
                .visit(new EvalXPath(n));
  }

  public static void main(String [] args){
    Node doc = ParseXML.parseXml(args[0]);

    XPathResult result
     =  eval(new QualifiedExpr
           (new RootSlashSlash
               (new Slash(new TagName("code"),new AttributeName("class")))
           ,new StringLiteral("XPath"))
            ,doc);
    System.out.println(result);

    result
     =  eval
         (new QualifiedExpr(new RootSlash
            (new Slash(new TagName("skript")
            ,new Slash(new TagName("kapitel")
            ,new Slash(new TagName("section"),new DotDot()))))
          ,new NumLiteral(1))
         ,doc);

    System.out.println(result);

    result
```

```
      =  eval
          (new RootSlash
             (new Slash(new TagName("skript")
             ,new Slash(new TagName("kapitel")
             ,new QualifiedExpr(new TagName("section"),new NumLiteral(2)))))
          ,doc);

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

Prädikatauswertung Es bleibt das Ergebnis eines Prädikats als bool'schen Wert zu interpretieren. Hierzu schreiben wir einen Besucher auf `XPathResult`.

TestQualifier.java

```
package name.panitz.xml.xpath;

import org.w3c.dom.Node;

public class TestQualifier extends XPathResultVisitor<Boolean> {
  final int pos;
  final int size;
  final Node context;

  public TestQualifier(Node c,int p,int s){context=c;pos=p;size=s;}

  public Boolean eval(BooleanResult e){return e.getValue();}
  public Boolean eval(NumResult e){return e.getValue()==pos;}
  public Boolean eval(StringResult e) {
   return e.getValue().equals(context.getNodeValue());}
  public Boolean eval(Nodes e) {return !e.getValue().isEmpty();}
}
```

NormalizeXPath.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import name.panitz.domtest.*;

public class NormalizeXPath extends XPathVisitor<XPath> {

public XPath eval(Axis e){return e;}

public XPath eval(RootSlash e){
  final XPath n1 = e.getExpr();
  return new RootSlash(e.getExpr().visit(this));}


```



```
public XPath eval(RootSlashSlash e){
  return new RootSlashSlash(e.getExpr().visit(this));}

/**  mache Slash(Slash(e1,e2),e3)
   zu Slash(e1,Slash(e2,e3))
und
  mache Slash(SlashSlash(e1,e2),e3)
   zu SlashSlash(e1,Slash(e2,e3))
*/

public XPath eval(Slash e){
  final XPath n1 = e.getE1();
  final XPath e3 = e.getE2();
  if (n1 instanceof Slash){
    final XPath e1 = ((Slash)n1).getE1();
    final XPath e2 = ((Slash)n1).getE2();
    return new Slash(e1,new Slash(e2,e3)).visit(this);
  }
  if (n1 instanceof SlashSlash){
    final XPath e1 = ((SlashSlash)n1).getE1();
    final XPath e2 = ((SlashSlash)n1).getE2();
    return new SlashSlash(e1,new Slash(e2,e3)).visit(this);
  }
  return new Slash(n1.visit(this),e3.visit(this));
}

public XPath eval(SlashSlash e){
  final XPath n1 = e.getE1();
  final XPath e3 = e.getE2();
  if (n1 instanceof Slash){
    final XPath e1 = ((Slash)n1).getE1();
    final XPath e2 = ((Slash)n1).getE2();
    return new Slash(e1,new SlashSlash(e2,e3)).visit(this);
  }
  if (n1 instanceof SlashSlash){
    final XPath e1 = ((SlashSlash)n1).getE1();
    final XPath e2 = ((SlashSlash)n1).getE2();
    return new SlashSlash(e1,new SlashSlash(e2,e3)).visit(this);
  }
  return new Slash(n1.visit(this),e3.visit(this));
}

public XPath eval(Union e){
  return new Union(e.getE1().visit(this),e.getE2().visit(this));}
public XPath eval(TagName e){return e;}
public XPath eval(AttributeName e){return e;}
public XPath eval(Dot e){return e;}
public XPath eval(DotDot e){return e;}
public XPath eval(NodeSelect e){return e;}
public XPath eval(TextSelect e){return e;}
```

```
public XPath eval(PISelect e){return e;}
public XPath eval(CommentSelect e){return e;}
public XPath eval(Star e){return e;}
public XPath eval(AtStar e){return e;}

public XPath eval(Function e){return e;}
public XPath eval(BinOperator e){return e;}
public XPath eval(UnaryOperator e){return e;}
public XPath eval(QualifiedExpr e){
  return new QualifiedExpr(e.getExpr().visit(this)
                           ,e.getQualifier().visit(this));}
public XPath eval(NumLiteral e){return e;}
public XPath eval(StringLiteral e){return e;}
}
```

3.4.2 XSLT: Transformationen in Dokumenten

XML-Dokumente enthalten keinerlei Information darüber, wie sie visualisiert werden sollen. Hierzu kann man getrennt von seinem XML-Dokument ein sogenanntes *Stylesheet* schreiben. XSL ist eine Sprache zum Schreiben von Stylesheets für XML-Dokumente. XSL ist in gewisser Weise eine Programmiersprache, deren Programme eine ganz bestimmte Aufgabe haben: XML Dokumente in andere XML-Dokumente zu transformieren. Die häufigste Anwendung von XSL dürfte sein, XML-Dokumente in HTML-Dokumente umzuwandeln.

Beispiel:
Wir werden die wichtigsten XSLT-Konstrukte mit folgendem kleinem XML Dokument ausprobieren:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
  <label>Apple</label>
 </cd>
 <cd>
  <artist>The Beatles</artist>
  <title>Rubber Soul</title>
  <label>Parlophone</label>
 </cd>
 <cd>
  <artist>Duran Duran</artist>
  <title>Rio</title>
  <label>Tritec</label>
 </cd>
 <cd>
  <artist>Depeche Mode</artist>
  <title>Construction Time Again</title>
```



```
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Yazoo</artist>
  <title>Upstairs at Eric's</title>
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Marc Almond</artist>
  <title>Absinthe</title>
  <label>Some Bizarre</label>
 </cd>
 <cd>
  <artist>ABC</artist>
  <title>Beauty Stab</title>
  <label>Mercury</label>
 </cd>
</cds>
```

Gesamtstruktur

XSLT-Skripte sind syntaktisch auch wieder XML-Dokumente. Ein XSLT-Skript hat feste Tagnamen, die eine Bedeutung für den XSLT-Prozessor haben. Diese Tagnamen haben einen festen definierten Namensraum. Das äußerste Element eines XSLT-Skripts hat den Tagnamen `stylesheet`. Damit hat ein XSLT-Skript einen Rahmen der folgenden Form:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

Einbinden in ein XML-Dokument Wir können mit einer *Processing-Instruction* am Anfang eines XML-Dokumentes definieren, mit welchem XSLT Stylesheet es zu bearbeiten ist. Hierzu wird als Referenz im Attribut `href` die XSLT-Datei angegeben.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>........
```

Templates (Formulare)

Das wichtigste Element in einem XSLT-Skript ist das Element `xsl:template`. In ihm wird definiert, wie ein bestimmtes Element transformiert werden soll. Es hat schematisch folgende

Form:
`<xsl:template match="`*Elementname*`">`
zu erzeugender Code
`</xsl:template >`

Beispiel:
Folgendes XSLT-Skript transformiert die XML-Datei mit den CDs in eine HTML-Tabelle, in der die CDs tabellarisch aufgelistet sind.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Startregel für das ganze Dokument. -->
<xsl:template match="/">
  <html><head><title>CD Tabelle</title></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>


<!-- Regel für die CD-Liste. -->
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates/>
  </table>
</xsl:template>

<xsl:template match="cd">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="artist">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="title">
  <td><xsl:apply-templates/></td>
</xsl:template>

<!-- Regel für alle übrigen Elemente.
     Mit diesen soll nichts gemacht werden -->
<xsl:template match="*">
</xsl:template>

</xsl:stylesheet>
```


Öffnen wir nun die XML Datei, die unsere CD-Liste enthält im Webbrowser, so wendet er die Regeln des referenzierten XSLT-Skriptes an und zeigt die so generierte Webseite wie in Abbildung 3.3 zu sehen an.

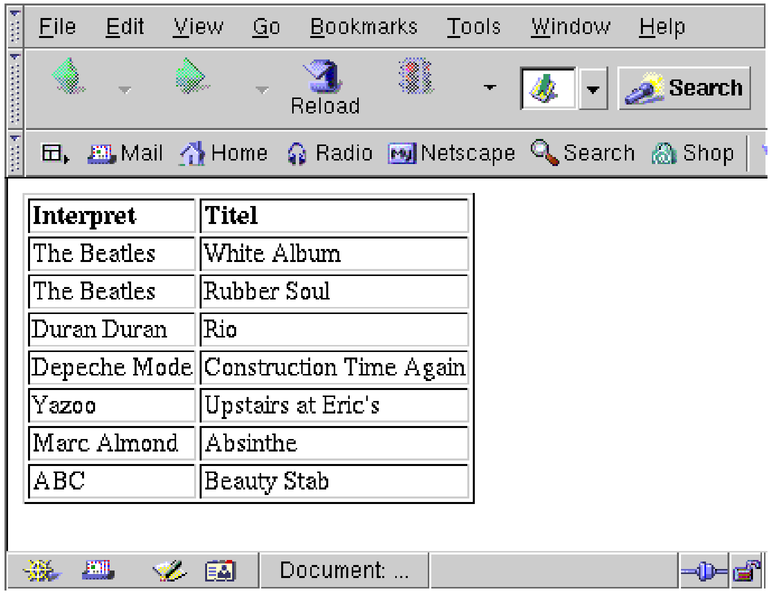

Abbildung 3.3: Anzeige der per XSLT-Skript generierten HTML-Seite.

Läßt man sich vom Browser hingegen den Quelltest der Seite anzeigen, so wird kein HTML-Code angezeigt sondern der XML-Code.

Auswahl von Teildokumenten

Das Element `xsl:apply-templates` veranlasst den XSLT-Prozessor die Elemente des XML Ausgangsdokuments weiter mit dem XSL Stylesheet zu bearbeiten. Wenn dieses Element kein Attribut hat, wie in unseren bisherigen Beispielen, dann werden alle Kinder berücksichtigt. Mit dem Attribut `select` lassen sich bestimmte Kinder selektieren, die in der Folge nur noch betrachtet werden sollen.

Beispiel:
Im folgenden XSL Element selektieren wir für `cd`-Elemente nur die `title` und `artist` Kinder und ignorieren die `label` Kinder.

```

<!-- für das Element "cd" -->
<xsl:template match="cd">
 <!--erzeuge ein Element "tr" -->
```

```
  <tr>
    <!-- wende das stylesheet weiter an auf die Kinderelemente
    "title" -->
    <xsl:apply-templates select="title"/>
    <!-- wende das stylesheet weiter an auf die Kinderelemente
    "artist" -->
    <xsl:apply-templates select="artist"/>
  </tr>
</xsl:template>
```

Sortieren von Dokumentteilen

XSLT kennt ein Konstrukt, um zu beschreiben, daß bestimmte Dokumente sortiert werden sollen nach bestimmten Kriterien. Hierzu gibt es das XSLT-Element `xsl:sort`. In einem Attribut `select` wird

angegeben, nach welchen Elementteil sortiert werden soll.

> **Beispiel:**
> Zum Sortieren der CD-Liste kann mit `xsl:sort` das Unterelement `artist` als Sortierschlüssel bestimmt werden.

```
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates select="cd">
      <xsl:sort select="artist"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
```

Weitere Konstrukte

XSLT kennt noch viele weitere Konstrukte, so z.B. für bedingte Ausdrücke wie in herkömmlichen Programmiersprachen durch einen **if**-Befehl und explizite Schleifenkonstrukte, Möglichkeiten das Ausgangsdokument, mehrfach verschiedentlich zu durchlaufen, und bei einem Element nicht nur auf Grund seines Elementnamens zu reagieren, sondern auch seine Kontext im Dokument zu berücksichtigen. Der interessierte Leser sei auf eines der vielen Tutorials im Netz oder auf die Empfehlung des W3C verwiesen.

XSLT in Java

XSLT.java

```
package name.panitz.xml.xslt;

import org.w3c.dom.Node;

import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
```



```
import javax.xml.transform.OutputKeys;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.TransformerException;

import java.io.StringWriter;
import java.io.File ;

public class XSLT {
  static public String transform(File xslt,File doc){
    return transform(new StreamSource(doc),new StreamSource(doc));
  }

  static public String transform(Source xslt,Source doc){
    try{
      StringWriter writer = new StringWriter();
      Transformer t =
         TransformerFactory
          .newInstance()
          .newTransformer(xslt);
      t.transform(doc,new StreamResult(writer));
      return writer.getBuffer().toString();
    }catch (TransformerException _){
      return "";
    }
  }

  public static void main(String [] args)throws Exception {
    System.out.println(transform(new File(args[0]),new File(args[1])));
  }

}
```

3.4.3 XQuery: Anfragen

fac.xquery

```
declare function fac($n) {if ($n=0) then 1 else $n*fac($n - 1)};
fac(5)
```

htmlfac.xquery

```
declare function fac($n) {if ($n=0) then 1 else $n*fac($n - 1)};

<html><body>fac(5)={fac(5)}</body></html>
```

countKapitel.xquery

```
count(doc("/home/sep/fh/prog4/skript.xml")//kapitel)
```

countCode.xquery

```
let $name := distinct-values
                (doc("/home/sep/fh/prog4/skript.xml")//code/@class)
return count($name)
```

exampleProgs1.xquery

```
<ul>{
for $name in distinct-values(doc("/home/sep/fh/prog4/skript.xml")//code/@class)
return <li>
         {string($name)}
       </li>
}</ul>
```

exampleProgs2.xquery

```
<ul>{
for $name in distinct-values(doc("/home/sep/fh/prog4/skript.xml")//code/@class)
order by $name
return <li>
         {string($name)}
       </li>
}</ul>
```

exampleProgs.xquery

```
<html>
<title>Klassen aus dem Skript</title>
<body>
<ul>{
let $codeFrags := doc("/home/sep/fh/prog4/skript.xml")//code[@class]
let
 $codes :=
   for $name in distinct-values($codeFrags/@class)
   let $codeFrag := $codeFrags[@class=$name][1]
   return
     <code>
        <className>{string($codeFrag/@class)}</classname>
        <lang>{if ($codeFrag/@lang)
              then string($codeFrag/@lang)
              else "java"}</lang>
        <package>{if ($codeFrag/@package)
                 then string($codeFrag/@package)
                 else "."}</package>
     </code>
for $code in $codes
order by $code/className
return
  let $name:= ($code/className/text(),".",$code/lang/text())
  let $fullname:= ($code/package/text(),"/",$name)
  return
    <li><a>{attribute href {$fullname}}{$name}</a></li>
}</ul></body></html>
```


3.5 Dokumenttypen

Wir haben bereits verschiedene Typen von XML-Dokumenten kennengelernt: XHTML-, XSLT- und SVG-Dokumente. Solche Typen von XML-Dokumenten sind dadurch gekennzeichnet, daß sie gültige XML-Dokumente sind, in denen nur bestimmte vordefinierte Tagnamen vorkommen und das auch nur in einer bestimmten Reihenfolge. Solche Typen von XML-Dokumenten können mit einer Typbeschreibungssprache definiert werden. Für XML gibt es zwei solcher Typbeschreibungssprachen: DTD und Schema.

3.5.1 DTD

DTD *(document type description)* ermöglicht es zu formulieren, welche Tags in einem Dokument vorkommen sollen. DTD ist keine eigens für XML erfundene Sprache, sondern aus SGML geerbt.

DTD-Dokumente sind keine XML-Dokumente, sondern haben eine eigene Syntax. Wir stellen im einzelnen diese Syntax vor:

- `<!DOCTYPE` *`root-element`* `[` *`doctype-declaration...`* `]>`
  Legt den Namen des top-level Elements fest und enthält die gesammte Definition des erlaubten Inhalts.
- `<!ELEMENT` *`element-name content-model`*`>`
  assoziiert einen *content model* mit allen Elementen, die diesen Namen haben.

  content models können wie folgt gebildet werden.:

  - `EMPTY`: Das Element hat keinen Inhalt.
  - `ANY`: Das Element hat einen beliebigen Inhalt
  - `#PCDATA`: Zeichenkette, also der eigentliche Text.
  - durch einen regulären Ausdruck, der aus den folgenden Komponenten gebildet werden kann.
    - Auswahl von Alternativen (oder): `(...|...|...)`
    - Sequenz von Ausdrücken: `(...,...,...)`
    - Option: `...?`
    - Ein bis mehrfach Wiederholung: `...*`
    - Null bis mehrfache Wiederholung: `...+`
- `<!ATTLIST`*`element-name attr-name attr-type attr-default`* `...>`
  Liste, die definiert, was für Attribute ein Element hat:

  Attribute werden definiert durch:

  - `CDATA`: beliebiger Text ist erlaubt
  - `(`*`value`*`|...)`: Aufzählung erlaubter Werte

  Attribut *default* sind:

  - `#REQUIRED`: Das Attribut muß immer vorhanden sein.

- `#IMPLIED`: Das Attribut ist optional
- `"value"`: Standardwert, wenn das Attribut fehlt.
- `#FIXED "value"`: Attribut kennt nur diesen Wert.

Beispiel:
Ein Beispiel für eine DTD, die ein Format für eine Rezeptsammlung definiert.

```
<!DOCTYPE collection SYSTEM "collection.dtd" [
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation
                 ,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step)*>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>]>
```

Aufgabe 5 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

3.5.2 Schema

Daß DTDs keine XML-Dokumente sind, hat die Erfinder von XML im Nachhinein recht geärgert. Außerdem war für die vorgesehen Zwecke im Bereich Datenverwaltung mit XML, die Ausdrucksstärke von DTD zum Beschreiben der Dokumenttypen nicht mehr ausdruckstark genug. Es läßt sich z.B. nicht formulieren, daß ein Attribut nur Zahlen als Wert haben darf. Aus diesen Gründen wurde eine Arbeitsgruppe ins Leben gerufen, die einen neuen Standard definieren sollte, der DTDs langfristig ersetzen kann. Die neue Sprache sollte in XML Syntax notiert werden und die bei DTDs vermissten Ausdrucksmittel beinhalten. Diese neue Typbneschreibungssprache heißt Schema und ist mitlerweile im W3C verabredet worden. Leider ist das endgültige Dokument über Schema recht komplex und oft schwer verständlich geworden. Wir wollen in dieser Vorlesung nicht näher auf Schema eingehen.

3.5.3 Geläufige Dokumenttypen

XHTML

SVG

Wir haben bisher XML-Dokumente geschrieben, um einen Dokumenttext zu strukturieren. Die Strukturierungsmöglichkeiten von XML machen es zu einem geeigneten Format, um beliebige Strukturen von Daten zu beschreiben. Eine gängige Form von Daten sind Graphiken. SVG *(scalable vector graphics)* sind XML-Dokumente, die graphische Elemente beschreiben. Zusätzlich kann in SVG ausgedrückt werden, ob und in welcher Weise Graphiken animiert sind.

SVG-Dokumente sind XML-Dokumente mit bestimmten festgelegten Tagnamen. Auch SVG ist dabei ein Standard, der vom W3C definiert wird.

Beispiel:
Folgendes kleine SVG-Dokument definiert eine Graphik, die einen Kreis, ein Viereck und einen Text enthält.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">
 <ellipse cx="100" cy="100" rx="48" ry="90" fill="limegreen" />
 <text x="20" y="115">SVG Textobjekt</text>
 <rect x="50" y="50" width="50" height="60" fill="red"/>
</svg>
```

Für ausführliche Informationen über die in SVG zur Verfügung stehenden Elemente sei der interessierte Student auf eines der vielen Tutorials im Netz oder direkt auf die Seiten des W3C verwiesen.

3.6 Aufgaben

Aufgabe 6 Die folgenden Dokumente sind kein wohlgeformetes XML. Begründen Sie, wo der Fehler liegt, und wie dieser Fehler behoben werden kann.

a)

```
<a>to be </a><b> or not to be</b>
```

Lösung
Kein top-level Element, das das ganze Dokument umschließt.

b)

```
<person geburtsjahr=1767>Ferdinand Carulli</person>
```

Lösung
Attributwerte müssen in Anführungszeichen stehen.

c)

```
<line>lebt wohl<br/>
<b>Gott weiß, wann wir uns wiedersehen</line>
```

Lösung
`<b>` wird nicht geschlossen.

d)

```
<kockrezept><!--habe ich aus dem Netz
  <name>Saltimbocca</name>
  <zubereitung>Zutaten aufeinanderlegen
               und braten.</zubereitung>
</kockrezept>
```

Lösung
Kommentar wird nicht geschlossen.

e)

```
<cd>
  <artist>dead&alive</artist>
  <title>you spin me round</title></cd>
```

Lösung
Das Zeichen & muß als character entity `&` geschrieben werden.

Aufgabe 7 Gegeben sind das folgende XML-Dokument:

TheaterAutoren.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<autoren>
<autor>
  <person>
    <nachname>Shakespeare</nachname>
    <vorname>William</vorname>
  </person>
  <werke>
    <opus>Hamlet</opus>
    <opus>Macbeth</opus>
    <opus>King Lear</opus>
  </werke>
</autor>
<autor>
  <person>
    <nachname>Kane</nachname>
    <vorname>Sarah</vorname>
  </person>
  <werke>
    <opus>Gesäubert</opus>
    <opus>Psychose 4.48</opus>
    <opus>Gier</opus>
  </werke>
</autor>
</autoren>
```

a) Schreiben Sie eine DTD, das die Struktur dieses Dokuments beschreibt.

Lösung

AutorenType.dtd

```
<!DOCTYPE autoren SYSTEM "AutorenType.dtd" [
<!ELEMENT autoren (autor+)>
<!ELEMENT autor (person,werke)>
<!ELEMENT werke (opus*)>
<!ELEMENT opus (#PCDATA)>
<!ELEMENT person (nachname,vorname)>
<!ELEMENT nachname (#PCDATA)>
<!ELEMENT vorname (#PCDATA)>]>
```

b) Entwerfen Sie Java Schnittstellen, die Objekte ihrer DTD beschreiben.

Lösung

Autoren.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Autoren {
  List<Autor> getAutorList();
}
```

Autor.java

```
package name.panitz.xml.exercise;
public interface Autor {
  Person getPerson();
  Werke  getWerke();
}
```

Person.java

```
package name.panitz.xml.exercise;
public interface Person {
  Nachname getNachname();
  Vorname  getVorname();
}
```

Werke.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Werke {
  List<Opus> getOpusList();
}
```

HasJustTextChild.java

```
package name.panitz.xml.exercise;
public interface HasJustTextChild {
  String getText();
}
```

Opus.java

```
package name.panitz.xml.exercise;
public interface Opus extends HasJustTextChild {}
```

Nachname.java
```
package name.panitz.xml.exercise;
public interface Nachname extends HasJustTextChild {}
```

Vorname.java
```
package name.panitz.xml.exercise;
public interface Vorname extends HasJustTextChild {}
```

c) Schreiben Sie ein XSLT-Skript, das obige Dokument in eine Html Liste der Werke ohne Autorangabe transformiert.

Lösung

WerkeListe.xsl
```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
     <html><head><title>Werke</title></head>
       <body><xsl:apply-templates /></body>
     </html>
  </xsl:template>

  <xsl:template match="autoren">
     <ul><xsl:apply-templates select="autor/werke/opus"/></ul>
  </xsl:template>

  <xsl:template match="opus">
     <li><xsl:apply-templates/></li>
  </xsl:template>
</xsl:stylesheet>
```

d) Schreiben Sie eine Javamethode
`List<String> getWerkListe(Autoren autoren);`,
die auf den Objekten Ihrer Schnittstelle für `Autoren` eine Liste von Werknamen erzeugt.

Lösung

GetWerke.java
```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;

public class GetWerke{
  public List<String> getWerkListe(Autoren autoren){
    List<String> result = new ArrayList<String>();
    for (Autor a:autoren.getAutorList()){
      for (Opus opus: a.getWerke().getOpusList())
        result.add(opus.getText());
    }
```



```
    return result;
  }
}
```

Aufgabe 8 Gegeben sei folgendes XML-Dokument:

Foo.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<x1><x2><x5>5</x5><x6>6</x6></x2><x3><x7>7</x7></x3>
<x4><x8/><x9><x10><x11></x11></x10></x9></x4></x1>
```

a) Zeichnen Sie dieses Dokument als Baum.

b) Welche Dokumente selektieren, die folgenden XPath-Ausdrücke ausgehend von der Wurzel dieses Dokuments

- `//x5/x6`
- `/descendant-or-self::x5/..`
- //x5/ancestor::*
- /x1/x2/following-sibling::*
- /descendant-or-self::text()/parent::node()

Aufgabe 9 Schreiben Sie eine Methode
`List<Node> getLeaves(Node n);`,
die für einen DOM Knoten, die Liste aller seiner Blätter zurückgibt.

Lösung

GetLeaves.java
```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;

public class GetLeaves{
  public List<Node> getLeaves(Node n){
    List<Node> result = new ArrayList<Node>();
    NodeList ns = n.getChildNodes();
    if (ns.getLength()==0){
      result.add(n); return result;
    }
    for (int i=0;i<ns.getLength();i++){
      result.addAll(getLeaves(ns.item(i)));
    }
    return result;
  }
}
```

Kapitel 4

Parsing XML Document Type Structures

4.1 Introduction

Parsers are a well understood concept. Usually a parser is used to read some text according to a context free grammar. It is tested if the text can be produced with the rules given in the grammar. In case of success a result tree is constructed. Different strategies for parsing can be applied. Usually a parser generator program is used. A textual representation of the grammar controls the generation of the parser. In functional programming languages parsers can easily be hand coded by way of parser cominators[HM96]. We will show, how to apply the concept of parser combinators to XML document type decriptions in Java.

4.1.1 Functional Programming

The main building block in functional programming is a function definition. Functions are first class citizens. They can be passed as argument to other functions, whithout being wrapped within some data object. Functions which take other function as argument are said to be of *higher order*. A function that creates a new result function from different argument functions is called a *combintor*.

Parser Combinators

A parser is basically a function. It consumes a token stream and results a list of several parse results. An empty list result can be interpreted as, no successfull parse[Wad85].

The following type definition characterizes a parser in Haskell.

HsParse.hs
```
type Parser result token = [token] -> [(result,[token])]
```

The type `Parser` is generic over the type of the token and the type of a result. A parser is a function. The result is a pair: the first element of the pair is the result of the parse,

the second element is the list of the remaining token (the parser will have consumed some tokens from the token list).

In functional programming languages a parser can be written by way of combinators. A parser consists of several basic parsers which are combined to a more complex parser. Simple parsers test if a certain token is the next token in the token stream.

The basic parsers, which tests exactely for one token, can in Haskell be written as:

HsParse.hs
```
getToken tok [] = []
getToken tok (x:xs)
  |tok==x = [(x,xs)]
  |otherwise = []
```

There are two fundamental ways to combine parsers to more complex parser:

- the sequence of two parsers.
- the alternativ of two parsers.

In functional programming languages combinator functions can be written, to implement these two ways to construct more complex parsers.

In the alternativ combination of two parser, first the first parser is applied to the token stream. Only if this does not succeed, the second parser is applied to the token stream.

In Haskell this parser can be implemented as:

HsParse.hs
```
alt p1 p2 toks
  |null res1 = p2 toks
  |otherwise = res1
 where
  res1 = p1 toks
```

In the sequence combination of two parsers, first the first parser is applied to the token stream and then the second parser is applied to the next token stream of the results of the first parse.

In Haskell this parser can be implemented as:

HsParse.hs
```
seqP p1 p2 toks = concat
  [[((rs1,rs2),tks2) |(rs2,tks2)<-(p2 tks1) ]
  |(rs1,tks1)<-p1 toks]
```

The result of a sequence of two parser is the pair of the two partial parses. Quite often a joint result is needed. A further function can be provided to apply some function to a parse result in order to get a new result:

HsParse.hs
```
mapP f p toks = [(f rs,tks)|(rs,tks)<-p toks]
```

With these three combinators basically all kinds of parsers can be defined.

Beispiel:
A very simple parser implementation in Haskell:

HsParse.hs
```
parantheses = mapP (\((x1,x2),x3) -> x2)
      (getToken '(' `seqP` a `seqP` getToken ')')

a = getToken 'a' `alt` parantheses

main = do
  print (parantheses "((((a))))")
  print (parantheses "((((a)))")
  print (parantheses "((((a))))bc")
```

The program has the following output:

```
sep@linux:~/fh/xmlparslib/examples> src/HsParse
[('a',"")]
[]
[('a',"bc")]
sep@linux:~/fh/xmlparslib/examples>
```

In the first call the string could complete be parsed, in the second call the parser failed and in the third call the parser succeeded but two further tokens where not consumed by the parser.

Further parser combinators can be expressed with the two combinators above. Typical combinators define the repetitive application of a parser, which is expressed by the symbols + and * in the extended Backus-Naur-form[NB60].

4.1.2 XML

XML documents are tree like structured documents. Common parser libraries are available to parse the textual representation of an XML document and to build the tree structure. In object orientated languages a common modell for the resulting tree structure is used, the *distributed object modell (DOM)*[**?**].

Document Type Definition

The document type of an XML document describes, which tags can be used within the document and which children these elements are allowed to have. The document type defiition (DTD) in XML is very similar to a context free grammar. [1] A DTD describes what kind of children elements a certain element can have in a document. In a DTD we find the typical elements of grammars:

- sequences of elements, denoted by a comma `,`.

[1] Throughout this paper we will ignore XML attributes.

- alternatives of elements donoted by the vertical bar `|`.
- several kinds of repetition denoted by `+`, `*` and `?`.

Consider the following DTD, which we will use as example throughout this paper:

album.dtd
```
<!DOCTYPE musiccollection SYSTEM "musiccollection.dtd" [
  <!ELEMENT musiccollection (lp|cd|mc)*>
  <!ELEMENT lp (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT cd (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT mc (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT track (title,timing?)>
  <!ELEMENT note (#PCDATA|author)*>

  <!ELEMENT timing (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT artist (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT recordingyear (#PCDATA)>
]>
```

The following XML document is build according to the structure defined in this DTD.

mymusic.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<musiccollection>
  <lp>
    <title>White Album</title>
    <artist>The Beatles</artist>
    <recordingyear>1968</recordingyear>
    <track><title>Revolution 9</title></track>
    <note><author>sep</author>my first lp</note>
  </lp>
  <cd>
    <title>Open All Night</title>
    <artist>Marc Almond</artist>
    <track><title>Tragedy</title></track>
    <note>
     <author>sep</author>
     Marc sung tainted love in the bandSoft Cell</note>
  </cd>
</musiccollection>
```

Another much simpler example is the following document:

simple.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<musiccollection/>
```

As can be seen, DTDs are very similar to grammars. And in fact, the same techniques can be applied for DTDs as for grammars. In [**?**] a Haskell combinator library for DTDs is presented. In the next sections we will try to apply these techniques in Java.

4.1.3 Java

In Java the main building block is a class, which defines a set of objects. A class can contain methods. These methods can represent functions. It is therefore natural to represent a Parser type by way of some class and the different parser combinators by way of subclasses in Java.

Generic Types

As we have seen in functional programming, parser combinators make heavily use of generic types. Fortunatly with release 1.5 of Java generic types are part of Java's type system. This makes the application of parser combinator techniques in Java tracktable.

4.2 Tree Parser

As we have seen a DTD is very similar to a grammar. The main difference is, that a grammar describes the way tokens are allowed in a token stream, whereas a DTD describes which elements can occur in a tree structure. Therefore a parser which we write for a DTD will not try to test, if a token stream can be build with the rules, but if a certain tree structure can be build with the rules in the DTD. What we get is a parser, which takes a tree and not a stream as input.

4.2.1 Parser Type

We will now define classes to represent parsers over XML trees in Java.

First of all, we need some class to represent the result of a parse. In the Haskell application above we used a pair for the result of a parse. The first element of the pair represented the actual result data constructed, the second element the remaining token stream. In our case, we want to parse over XML trees. Therefore the token List will be a list of XML nodes as defined in DOM. We use the utility class `Tuple2` as defined in [Pan04a] to represent pairs.

The following class represents the result of a parse.

ParseResult.java
```
package name.panitz.crempel.util.xml.parslib;

import name.panitz.crempel.util.Tuple2;
import java.util.List;
import org.w3c.dom.Node;

public class ParseResult<a> extends Tuple2<a,List<Node>>{
  public ParseResult(a n,List<Node> xs){super(n,xs);}

  public boolean failed(){return false;}
}
```

This class is generic over the actual result type. We added a method for testing, whether the parse was successful.

We define a special subclass for parse results, which denotes unsuccessful parses.

Fail.java
```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

public class Fail<a> extends ParseResult<a>{
  public Fail(List<Node> xs){super(null,xs);}
  public boolean failed(){return true;}
}
```

Now where we have defined what a parse result looks like, we can define a class to describe a parser. A parser is some object, which has a method `parse`. This maethod takes a list of DOM nodes as argument and returns a parse result:

Parser.java
```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.UnaryFunction;

public interface  Parser<a>{
  public ParseResult<a> parse(List<Node> xs);
```

4.2.2 Combinators

Now, where we have a parser class, we would like to add combinator function to this class. We add the following combinators:

- `seq` to create a sequence of this parser with some other parser.
- `choice` to create an alternative of this or some other parser.
- `star` to create a zero or more repetition parser of this parser.
- `plus` to create a one or more repetition parser of this parser.
- `query` to create a zero or one repetition parser of this parser.
- `map` to create a parser from this parser, which modifies the parse result with some further method.

This leads to the following method signatures in the interface `Parser`:

Parser.java

```
  public <b> Parser<Tuple2<a,b>> seq(Parser<b> p2);
  public <b extends a> Parser<a> choice(Parser<b> p2);
  public Parser<List<a>> star();
  public Parser<List<a>> plus();
  public Parser<Maybe<a>> query();
  public <b> Parser<b> map(UnaryFunction<a,b> f);
}
```

The sequence combinator creates a parser which has a pair of the two partial results as common result. The alternative parser will only allow a second parser, which has the same result type as this parser. This will lead to some complication later. Results of repetitions of more than one time are represented as lists of the partial results. For the optional repretition of zero or one time we use the class `Maybe` as of [Pan04a] result type. It has two subclasses, `Just` to denote there is such a result and `Nothing` to denote the contrary.

In the following we assume an abstract class `AbstractParser`, which implements these functions. We will give the definition of `AbstractParser` later in this paper.

Optional Parser

Let us start with the parser which is optional as denoted by the question mark `?` in a DTD. We write a class to represent this parser. This class contains a parser. In the method `parse`, this given parser is applied to the list of nodes. If this fails a `Nothing` object is returned, otherwise a `Just` object, which wrappes the result of the successful parse, is returned.

We get the following simple Java class:

Optional.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.Nothing;
import name.panitz.crempel.util.Just;
import org.w3c.dom.Node;

public  class Optional<a> extends  AbstractParser<Maybe<a>> {
  final Parser<a> p;
  public Optional(Parser<a> _p){p=_p;}
  public ParseResult<Maybe<a>> parse(List<Node> xs){
    final ParseResult<a> res = p.parse(xs);
    if (res.failed())
      return new ParseResult<Maybe<a>>(new Nothing<a>(),xs);
    return new ParseResult<Maybe<a>>(new Just<a>(res.e1),res.e2);
  }
}
```


Sequence

Next we implement a Java class which represents the sequence of two parsers. This class contains two inner parsers. The first one gets applied to the list of nodes, in case of success, the second is applied to the remaining list of the successful first parse.

We get the following simple Java class:

Seq.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Tuple2;
import org.w3c.dom.Node;

public  class Seq<a,b> extends  AbstractParser<Tuple2<a,b>> {
  final Parser<a> p1;
  final Parser<b> p2;

  public Seq(Parser<a> _p1,Parser<b> _p2){p1=_p1;p2=_p2;}

  public ParseResult<Tuple2<a,b>> parse(List<Node> xs){
    ParseResult<a> res1 = p1.parse(xs);
    if (res1.failed())
      return new Fail<Tuple2<a,b>>(xs);
    ParseResult<b> res2 = p2.parse(res1.e2);
    if (res2.failed())
      return new Fail<Tuple2<a,b>>(xs);
    return new ParseResult<Tuple2<a,b>>
                 (new Tuple2<a,b>(res1.e1,res2.e1),res2.e2);
  }
}
```

Choice

The alternative parser of parser can be defined in almost exactly the way as has been done in the intrductory Haskell example. First the first parser ist tried. If this succeeds its result is used, otherwise the second parser is applied. However, we need to create new objects of classes `ParseResult` and `Fail`. This is due to the fact that even if `b extends c`, `ParseResult<b>` does not extend `ParseResult<c>`.

Choice.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import name.panitz.crempel.util.Tuple2;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class  Choice<c,a extends c,b extends c>
```

```
  extends  AbstractParser<c> {
  final Parser<a> p1;
  final Parser<b> p2;

  public Choice(Parser<a> _p1,Parser<b> _p2){p1=_p1;p2=_p2;}

  public ParseResult<c> parse(List<Node> xs){
    final ParseResult<a> res1 =  p1.parse(xs);
    if (res1.failed()){
      final ParseResult<b> res2 = p2.parse(xs);
      if (res2.failed()) return new Fail<c>(xs);
      return new  ParseResult<c>(res2.e1,res2.e2);
    }
    return new  ParseResult<c>(res1.e1,res1.e2);
  }
}
```

Repetitions

Reptetitive parsers try to apply a parser several times after another. We define one common class for repetitive parses. Subclasses will differentiate if at least one time the parser needs to be applied successfully.

Repetition.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
```

We need an parser to be applied in a repetition and a flag to signify, if the parser needs to be applied sucessfulle at least one time:

Repetition.java

```
public  class Repetition<a> extends  AbstractParser<List<a>> {
  final Parser<a> p;
  final boolean atLeastOne;
  public Repetition(Parser<a> _p,boolean one){
    p=_p;atLeastOne=one;
  }
```

Within the method `parse` the parser is applied is long as it does not fail. The results of the parses are added to a common result list.

Repetition.java

```
  public ParseResult<List<a>> parse(List<Node> xs){
    final List<a> resultList = new ArrayList<a>();
    int i = 0;

```



```
    while (true){
      final ParseResult<a> res = p.parse(xs);
      xs=res.e2;

      if (res.failed()) break;
      resultList.add(res.e1);
    }

    if (resultList.isEmpty()&& atLeastOne)
      return new Fail<List<a>>(xs);
    return new ParseResult<List<a>>(resultList,xs);
  }
}
```

Simple subclasses can be defined for the two kinds of repetition in a DTD.

Star.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;

public class Star<a> extends Repetition<a> {
  public Star(Parser<a> p){super(p,false);}
}
```

Plus.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;

public class Plus<a> extends Repetition<a> {
  public Plus(Parser<a> p){super(p,true);}
}
```

Map

Eventually we define the parser class for modifying the result of a parser. To pass the function to be applied to the parser we use an object, which implements the interface `UnaryFunction` from [Pan04a].

The implementation is straight forward. Apply the parser and in case of success create a new parse result object from the original parse result.

Map.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;
import name.panitz.crempel.util.UnaryFunction;

```

```
public class Map<a,b> extends AbstractParser<b> {
  final Parser<a> p;
  final UnaryFunction<a,b> f;

  public Map(UnaryFunction<a,b> _f,Parser<a> _p){f=_f;p=_p;}

  public ParseResult<b> parse(List<Node> xs){
    final ParseResult<a> res = p.parse(xs);
    if (res.failed()) return new Fail<b>(xs);
    return new ParseResult<b>(f.eval(res.e1),res.e2);
  }
}
```

Abstract Parser Class

Now where we have classes for all kinds of parser combinators, we can use these in an abstract parser class, which implements the combinator methods of the parser interface. We simply instantiate the corresponding parser classes.

AbstractParser.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.UnaryFunction;

public abstract class AbstractParser<a> implements Parser<a>{
  public <b> Parser<Tuple2<a,b>> seq(Parser<b> p2){
    return new Seq<a,b>(this,p2);
  }

  public <b extends a> Parser<a> choice(Parser<b> p2){
    return new Choice<a,a,b>(this,p2);
  }

  public Parser<List<a>> star(){return new Star<a>(this);}
  public Parser<List<a>> plus(){return new Plus<a>(this);}
  public Parser<Maybe<a>> query(){return new Optional<a>(this);}
  public <b> Parser<b> map(UnaryFunction<a,b> f){
    return new Map<a,b>(f,this);
  }
}
```

Note that the method `choice` cannot be written as general as we would like to. We cannot specify that we need one smallest common superclass of the two result types.

4.2.3 Atomic Parsers

Now we are able to combine parsers to more complex parsers. We are still missing the basic parser like the function `getToken` in our Haskell parser above. In XML documents there are several different basic units: PCDATA, Elements and Empty content.

PCDATA

The basic parser, which checks for a PCDATA node simply checks the node type of the next node. If there is a next node of type text node, then the parser succeeds and consumes the token. Otherwise it fails. We define this parser with result type `String`. The resulting `String` will contain the contents of the text node.

PCData.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

public class PCData extends AbstractParser<String> {

  public ParseResult<String> parse(List<Node> xs){
    if (xs.isEmpty()) return new Fail<String>(xs);
    final Node x = xs.get(0);
    if (x.getNodeType()==Node.TEXT_NODE)
      return new ParseResult<String>
                  (x.getNodeValue(),xs.subList(1,xs.size()));
System.out.println("expected pcdata but got: "+x);
    return new Fail<String>(xs);
  }
}
```

Empty

A very simple parser, does always succeed with some result and does not consume any node from the input.

Return.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

public class Return<a> extends AbstractParser<a> {
  final a returnValue;
  public Return(a r){returnValue=r;}
  public ParseResult<a> parse(List<Node> xs){
    return new ParseResult<a>(returnValue,xs);
  }
}
```

Element

The probably most interesting parser reads an element node and proceeds with the children of the element node. The children of the element node are processed with a given parser.

Therefore the element parser needs two arguments:

- the name of the element to be read.
- the parser to be applied to the children of the element.

Element.java
```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import name.panitz.crempel.util.Closure;


public class Element<a> extends AbstractParser<a> {
  Parser<a> p=null;
  private final Closure<Parser<a>> pc;
  final String elementName;

  public Element(String n,Parser<a> _p){elementName=n;p=_p;pc=null;}
  public Element(String n,Closure<Parser<a>> _p){elementName=n;pc=_p;}

  public Parser<a> getP(){if (p==null) p=pc.eval(); return p;}
```

For simplicity reasons, we will neglect text nodes from list of nodes, which consist solely of whitespace:

Element.java
```
  public ParseResult<a> parse(List<Node> xs){
    if (xs.isEmpty()) {return new Fail<a>(xs);}
    Node x = xs.get(0);
    while (   x.getNodeType()==Node.TEXT_NODE
           && x.getNodeValue().trim().length()==0
          ){
      xs=xs.subList(1,xs.size());
      if (xs.isEmpty()) return new Fail<a>(xs);

      x = xs.get(0);
    }
```

Then we compare the node name with the expected node name:

Element.java
```
    final String name = x.getNodeName();
System.out.println(name+" <-> "+elementName);
    if (!name.equals(elementName))
      return new Fail<a>(xs);
```

Eventually we get the children of the node and apply the given parser to this list.

Element.java
```
    final List<Node> ys = nodelistToList(x.getChildNodes());
    ParseResult<a> res = getP().parse(ys);

    if( res.failed()) return new Fail<a>(xs);
```

If this succeeds we ensure that only neglectable whitespace nodes follow in the list of nodes

Element.java
```
    for (Node y :res.e2){
      if (y.getNodeType()!=Node.TEXT_NODE
            || y.getNodeValue().trim().length()!=0)
         return new Fail<a>(xs);
    }
```

Eventually the result can be constructed.

Element.java
```
    return new ParseResult<a>(res.e1,xs.subList(1,xs.size()));
  }
```

The following method has been used to convert DOM node list into a java list of nodes.

Element.java
```
  static public List<Node> nodelistToList(NodeList xs){
    List<Node> result = new ArrayList<Node>();
    for (int i=0;i<xs.getLength();i=i+1){
      result.add(xs.item(i));
    }
    return result;
  }
}
```

4.2.4 Building parsers

We have got everything now to implement parsers for a DTD in Java.

Defining a Parser

We can write a parser, which checks if a certain XML document is build according to the type definition as defined for our music collection example. As result for the parse we simply return some `String` object.

MusiccollectionParser.java
```
package name.panitz.crempel.test;

import java.util.List;
import org.w3c.dom.Node;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.parslib.*;

public class MusiccollectionParser extends AbstractParser<String>{
```

First of all we define some objects of different `UnaryFunction` types, which are needed to construct the final result of the parses. First an object for concatenating two `String` objects:

MusiccollectionParser.java
```
  final private UnaryFunction<Tuple2<String,String>,String> concat
    = new UnaryFunction<Tuple2<String,String>,String>(){
         public String eval(Tuple2<String,String> p){
           return p.e1+p.e2;
         }
      };
```

The second function retries a `String` object from a `Maybe<String>` object.

MusiccollectionParser.java
```
  final private UnaryFunction<Maybe<String>,String> getString
    = new UnaryFunction<Maybe<String>,String>(){
         public String eval(Maybe<String> p){
           if (p instanceof Nothing) return "";
           return ((Just<String>)p).getJust();
         }
      };
```

Eventually we provide a function object, which concatenates a list of `String` object into a single `String` object.

MusiccollectionParser.java
```
  final private UnaryFunction<List<String>,String> concatList
    = new UnaryFunction<List<String>,String>(){
         public String eval(List<String> xs){
           final StringBuffer result = new StringBuffer();
           for (String x:xs) result.append(x);
           return result.toString();
         }
      };
```


Now we can define parsers for the elements as defined in the dtd. The most simple parsers are for those elements, which have only text nodes as children:

MusiccollectionParser.java
```
  final private Parser<String> recordingyear
    = new Element<String>("recordingyear",new PCData());
  final private Parser<String> artist
    = new Element<String>("artist",new PCData());
  final private Parser<String> title
    = new Element<String>("title",new PCData());
  final private Parser<String> timing
    = new Element<String>("timing",new PCData());
  final private Parser<String> author
    = new Element<String>("author",new PCData());
```

Parsers for the remaining elements are combined from these basic parsers.

MusiccollectionParser.java
```
  final private Parser<String> note
    = new Element<String>
        ("note",author.seq(new PCData()).map(concat));

  final private Parser<String> track
    = new Element<String>("track"
         ,title.seq(timing.query().map(getString)).map(concat));

  final private Parser<String> content
    =    title
         .seq(artist).map(concat)
         .seq(recordingyear.query().map(getString)).map(concat)
         .seq(track.plus().map(concatList)).map(concat)
         .seq(note.star().map(concatList)).map(concat);

  final private Parser<String> lp
    = new Element<String>("lp",content);
  final private Parser<String> cd
    = new Element<String>("cd",content);
  final private Parser<String> mc
    = new Element<String>("mc",content);

  final private Parser<String> musiccollection
   = new Element<String>
      ("musiccollection",lp.choice(cd).choice(mc)
                             .star().map(concatList));

  public ParseResult<String> parse(List<Node> xs){
    return musiccollection.parse(xs);
  }
}
```

Starting the parser

We have a tree parser over XML documents of type `musiccollection`. This parser can be applied to a list of dom objects:

MainParser.java
```java
package name.panitz.crempel.test;

import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.parslib.*;

import org.w3c.dom.*;
import java.util.*;
import java.io.*;
import javax.xml.parsers.*;

public class MainParser{
  public static void main(String [] args){
    System.out.println(pFile(args[0],args[1]));
  }

  public static Object pFile(String parserClass,String fileName){
    try{
      DocumentBuilderFactory factory
        = DocumentBuilderFactory.newInstance();

      factory.setIgnoringElementContentWhitespace(true);
      factory.setCoalescing(true);
      factory.setIgnoringComments(true);

      Document doc
        =factory.newDocumentBuilder()
        .parse(new File(fileName)) ;
      doc.normalize();
      List<Node> xs = new ArrayList<Node>();
      xs.add(doc.getChildNodes().item(0));

      Parser<Object> p
       = (Parser<Object>)Class.forName(parserClass).newInstance();

      Tuple2 res = p.parse(xs);

      return res.e1;
    }catch (Exception e){e.printStackTrace();return null;}
  }
}
```

The parser checks the document for its structure and returns the text of the document:

```
sep@linux:~/fh/xmlparslib/examples> java -classpath classes/  name.panitz.crempel.test.MainParser
```



```
 name.panitz.crempel.test.MusiccollectionParser src/mymusic.xml
White AlbumThe Beatles1968Revolution 9sepmy first lpOpen All NightMarc AlmondTragedysepMarc sung tainted love
sep@linux:~/fh/xmlparslib/examples>
```

4.3 Building a Tree Structure

In the last section we presented a library for writing tree parsers for a certain DTD. The actual parser needed to be hand coded for the given DTD. As has been seen this was very tedious and boring mechanical hand coding, which can be made automatically. As a matter of fact, a parser can easily be generated, but for the result type of the parser. In this sections we will develop a program that will generate tree classes for a DTD and a parser, which has an object of these tree classes as result.

4.3.1 Java types for DTDs

4.3.2 An algebraic type for DTDs

In this section we will make heavy use of the algebraic type framework as presented in [**?**]. First of all we define an algebraic type to represent a definition clause in a DTD (i.e. the right hand side of an element definition).

In a DTD there exist the following building blocks:

- PCData
- some element name
- Any
- Empty
- one or more repetition
- zero or more repetition
- zero or one repetition
- a list of choices
- a list of sequence items

We can express this in the following algebraic type definition.

DTDDef.adt
```
package name.panitz.crempel.util.xml.dtd.tree;

data class DTDDef {
  DTDPCData();
  DTDTagName(String tagName);
  DTDAny();
  DTDEmpty();
  DTDPlus(DTDDef dtd);
```

```
  DTDStar(DTDDef dtd);
  DTDQuery(DTDDef dtd);
  DTDSeq(java.util.List<DTDDef> seqParts);
  DTDChoice(java.util.List<DTDDef> choiceParts);
}
```

For this algenbraic type definition we get generated a set of tree classes and a visitor framework.

Simple visitors vor DTDDef

As a first exercise we write two simple visitor classes for the type `DTDDef`.

Show The first visitor returns a textual representation of the `DTD` object.

DTDShow.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;

public class DTDShow extends DTDDefVisitor<String>{
  public String eval(DTDPCData _){return "#PCDATA";};
  public String eval(DTDTagName x){return x.getTagName();}
  public String eval(DTDEmpty x){return "Empty";}
  public String eval(DTDAny x){return "Any";}
  public String eval(DTDPlus x){return show(x.getDtd())+"+";};
  public String eval(DTDStar x){return show(x.getDtd())+"*";};
  public String eval(DTDQuery x){return show(x.getDtd())+"?";};
  public String eval(DTDSeq s){
    return aux(this,",",s.getSeqParts());}
  public String eval(DTDChoice c){
  return aux(this,"|",c.getChoiceParts());}

  private String aux
            (DTDShow visitor,String sep,List<DTDDef> parts){
    StringBuffer result=new StringBuffer("(");
    boolean first = true; for (DTDDef x:parts){
      if (first) first=false; else result.append(sep);
      result.append(show(x));
    }
    result.append(")");
    return result.toString();
  }

  public String show(DTDDef def){return def.visit(this);}
}
```


ShowType Now we write a visitor, which returns in Java syntax the Java type, which we associate to a certain DTD construct:

ShowType.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
import java.util.List;
import java.util.ArrayList;

public class ShowType extends DTDDefVisitor<String>{
  public String eval(DTDPCData x){return "String";}
  public String eval(DTDTagName x){return x.getTagName();}
  public String eval(DTDEmpty x){return "Object";}
  public String eval(DTDAny x){return "Object";}
  public String eval(DTDPlus x){return "List<"+x.getDtd().visit(this)+">";}
  public String eval(DTDStar x){return "List<"+x.getDtd().visit(this)+">";}
  public String eval(DTDQuery x){return "Maybe<"+x.getDtd().visit(this)+">";
  public String eval(DTDSeq x){
    return listAsType((List<DTDDef>) x.getSeqParts());}
  public String eval(DTDChoice x){
    List<DTDDef> parts = x.getChoiceParts();
    if (parts.size()==1) return parts.get(0).visit(this);
    StringBuffer result=new StringBuffer("Choice");
    for (DTDDef y:((List<DTDDef>) parts))
      result.append("_"+typeToIdent(y.visit(this)));
    return result.toString();
  }

  private  String listAsType(List<DTDDef> xs){
    int size=xs.size();

    if (size==1) return xs.get(0).visit(this);
    StringBuffer result = new StringBuffer();
    for (Integer i:new FromTo(2,size)){
      result.append("Tuple2<");
    }
    boolean first=true;
    for (DTDDef dtd:xs){
      if (!first) result.append(",");
      result.append(dtd.visit(this));
      if (!first) result.append(">");
      first=false;
    }
    return result.toString();
  }

  public static String showType(DTDDef def){
   return def.visit(new ShowType());}

```

```
  static public String typeToIdent(String s ){
    return s.replace('<','_').replace('>','_').replace('.','_');
  }
}
```

4.3.3 Generation of tree classes

In this section we write a

GenerateADT.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.adt.parser.ADT;
import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.Writer;
import java.io.StringReader;
import java.io.FileWriter;
import java.io.StringWriter;
import java.io.IOException;

public class GenerateADT extends DTDDefVisitor<String>{

  final String elementName;
  public GenerateADT(String e){elementName=e;}

  public String eval(DTDPCData x){return "Con(String pcdata);";}
  public String eval(DTDTagName x){
    final String typeName = ShowType.showType(x);
    return "Con("+typeName+" the"+typeName+");";}
  public String eval(DTDEmpty x){return "";}
  public String eval(DTDAny x){return "";}
  public String eval(DTDPlus x){
    return "Con(List<"+ShowType.showType(x.getDtd())+"> xs);";}
  public String eval(DTDStar x){
    return "Con(List<"+ShowType.showType(x.getDtd())+"> xs);";}
  public String eval(DTDQuery x){
    return "Con(Maybe<"+ShowType.showType(x.getDtd())+"> xs);";}
  public String eval(DTDSeq x){
    StringBuffer result = new StringBuffer("Con(");
    boolean first = true;
    for (DTDDef dtd :x.getSeqParts()){
      if (!first) result.append(",");
      final String typeName = ShowType.showType(dtd);
      result.append(typeName+" the"+ShowType.typeToIdent(typeName));

```



```
      first=false;
    }
    result.append(");");
    return result.toString();
  }
  public String eval(DTDChoice x){
    StringBuffer result = new StringBuffer();
    for (DTDDef dtd :x.getChoiceParts()){
      String typeName = ShowType.showType(dtd);
      final String varName = ShowType.typeToIdent(typeName);

      result.append("\n  C"+elementName+varName
                              +"("+typeName+" the"+varName+");");
    }
    return result.toString();
  }

  public static String generateADT(String element,DTDDef def){
   return def.visit(new GenerateADT(element));}

  public static void generateTreeClasses
                         (List<Tuple3<Boolean,String,DTDDef>> xs){
    try {
      for (Tuple3<Boolean,String,DTDDef>x:xs){
        Writer out = new FileWriter(x.e2+".adt");
        out.write("import java.util.List;\n");
        out.write("import name.panitz.crempel.util.Maybe;\n");
        out.write("data class "+x.e2+"{\n");
        out.write(generateADT(x.e2,x.e3));
        out.write("}");
        out.close();
      }
    }catch (IOException e){e.printStackTrace();}
  }

  public static void  generateADT
     (String paket,String path,List<Tuple3<Boolean,String,DTDDef>> xs){
    try {
      List<AbstractDataType> adts = new ArrayList<AbstractDataType>();
      for (Tuple3<Boolean,String,DTDDef>x:xs){
        StringWriter out = new StringWriter();//FileWriter(x.e2+".adt");
        out.write("package "+paket+";\n");
        out.write("import java.util.List;\n");
        out.write("import name.panitz.crempel.util.Maybe;\n");
        out.write("data class "+x.e2+"{\n");
        out.write(generateADT(x.e2,x.e3));
        out.write("}");
        out.close();

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

```
        ADT parser = new ADT(new StringReader(out.toString()));
        AbstractDataType adt = parser.adt();
        adts.add(adt);
        adt.generateClasses(path);
       }
     }catch (Exception e){e.printStackTrace();}
  }

}
```

4.3.4 Generation of parser code

ParserCode.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.Writer;
import java.io.StringWriter;
import java.io.IOException;

public class ParserCode extends DTDDefVisitor<String>{
  final String elementName;
  public ParserCode(String e){elementName=e;}

  public String eval(DTDPCData x){return "new PCData()";}
  public String eval(DTDTagName x){return "getV"+x.getTagName()+"()";}
  public String eval(DTDEmpty x){return "new Return(null)";}
  public String eval(DTDAny x){return null;}
  public String eval(DTDPlus x){return x.getDtd().visit(this)+".plus()";}
  public String eval(DTDStar x){return x.getDtd().visit(this)+".star()";}
  public String eval(DTDQuery x){return x.getDtd().visit(this)+".query()";}
  public String eval(DTDSeq x){
    StringBuffer result = new StringBuffer();
    boolean first = true;
    for (DTDDef dtd:(List<DTDDef>) x.getSeqParts()){
      if (!first){result.append(".seq(");}
      result.append(dtd.visit(this));
      if (!first){result.append(")");}
      first=false;
    }
    return result.toString();
  }

  public String eval(DTDChoice x){
    final List<DTDDef> xs =  x.getChoiceParts();
```



```
    if (xs.size()==1) {
      final DTDDef ch = xs.get(0);
      final String s  = ch.visit(this);
      return s;
    }
    StringBuffer result = new StringBuffer();
    boolean first = true;
    for (DTDDef dtd:(List<DTDDef>) xs){
      final String argType = ShowType.showType(dtd);
      final String resType = elementName;
      if (!first){result.append(".choice(");}
      result.append(dtd.visit(this));
      result.append(".<"+resType+">map(new UnaryFunction<");
      result.append(argType);
      result.append(",");
      result.append(resType);
      result.append(">(){");
      result.append("\n    public "+resType+" eval("+argType+" x){");

      String typeName = ShowType.showType(dtd);
      final String varName = ShowType.typeToIdent(typeName);
      result.append("\n      return ("+resType+")new C"
             +elementName+varName
             +"(x);");
      result.append("\n    }");
      result.append("\n  })");
      if (!first){result.append(")");}
      first=false;
    }
    return result.toString();
  }

  public static String parserCode(DTDDef def,String n){
   return def.visit(new ParserCode(n));}
}
```

WriteParser.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.Writer;
import java.io.StringWriter;
import java.io.IOException;

public class WriteParser extends DTDDefVisitor<String>{
  final String elementName;
```

```
  final boolean isGenerated  ;
  String contentType = null;
  String typeDef = null;
  String fieldName = null;
  String getterName = null;

  public WriteParser(String e,boolean g){elementName=e;isGenerated=g;}

  private void start(DTDDef def,StringBuffer result){
    contentType = ShowType.showType(def);
    typeDef = !isGenerated?elementName:contentType;
    fieldName = "v"+elementName;
    getterName = "getV"+elementName+"()";

    result.append("\n\n  private Parser<"+typeDef+"> "+fieldName+" = null;");
    result.append("\n  public Parser<"+typeDef+"> "+getterName+"{");
    result.append("\n    if ("+fieldName+"==null){");
    result.append("\n      "+fieldName+" = ");
    if (!isGenerated) {
      result.append("new Element<"+typeDef+">(\""+typeDef+"\"");
      result.append("\n     ,");
      result.append("new Closure<Parser<"+typeDef+">>(){public Parser<"+typeDef+"
    }

  }

  private String f(DTDDef def){
    StringBuffer result=new StringBuffer();
    start(def,result);
    result.append(ParserCode.parserCode(def,elementName));
    if (!isGenerated){
      result.append("\n     .<"+typeDef+">map(new UnaryFunction");
      result.append("<"+contentType+","+typeDef+">(){");
      result.append("\n       public "+typeDef+" eval("+contentType+" x){");
      result.append("\n         return new "+typeDef+"(x);");
      result.append("\n       }");
      result.append("\n     })");
    }
    end(def,result);
    return result.toString();
  }

  private void end(DTDDef def,StringBuffer result){
    if (!isGenerated){
      result.append("\n;}}//end of closure\n");
      result.append(")");

    }
    result.append(";");
    result.append("\n    }");
```



```
    result.append("\n    return "+fieldName+";");
    result.append("\n  }");
  }

  private String startend(DTDDef def){
    StringBuffer result=new StringBuffer();
    start(def,result);
    result.append(ParserCode.parserCode(def,elementName));
    end(def,result);
    return result.toString();
  }

  public String eval(DTDPCData x){return f(x);}
  public String eval(DTDTagName x){return f(x);}
  public String eval(DTDEmpty x){return f(x);}
  public String eval(DTDAny x){return null;}
  public String eval(DTDPlus x){return f(x);}
  public String eval(DTDStar x){return f(x);}
  public String eval(DTDQuery x){return f(x);}
  public String eval(DTDChoice x){return startend(x);}
  public String eval(DTDSeq x){
    StringBuffer result = new StringBuffer();
    start(x,result);
    result.append(ParserCode.parserCode(x,elementName));
    result.append(".map(new UnaryFunction<"
                 +ShowType.showType(x)+","+elementName+">(){");
    result.append("\n    public "+elementName);
    result.append(" eval("+ShowType.showType(x) +" p){");
    result.append("\n      return new "+elementName);
    unrollPairs(x.getSeqParts().size(),"p",result);
    result.append(";");
    result.append("\n    }");
    result.append("\n  }");
    result.append(")");
    end(x,result);
    return result.toString();
  }

  private void unrollPairs(int i,String var,StringBuffer r){
    String c = var;
    String result="";
    for (Integer j :new FromTo(2,i)){
       result=","+c+".e2"+result;
       c= c+".e1";
    }
    result=c+result;
    r.append("(");
    r.append(result);
    r.append(")");
  }
```

```java

  public static String writeParser
                      (DTDDef def,String n,boolean isGenerated){
   return def.visit(new WriteParser(n,isGenerated));}
}
```

4.3.5 Flattening of a DTD definition

FlattenResult.java

```java
package name.panitz.crempel.util.xml.dtd;
import name.panitz.crempel.util.xml.dtd.tree.DTDDef;
import name.panitz.crempel.util.Tuple3;
import name.panitz.crempel.util.Tuple2;
import java.util.List;
import java.util.ArrayList;


public class FlattenResult
   extends Tuple2<DTDDef,List<Tuple3<Boolean,String,DTDDef>>>{

  public FlattenResult(DTDDef dtd,List<Tuple3<Boolean,String,DTDDef>> es){
    super(dtd,es);
  }

  public FlattenResult(DTDDef dtd){
    super(dtd,new ArrayList<Tuple3<Boolean,String,DTDDef>>() );
  }
}
```

DTDDefFlatten.java

```java
package name.panitz.crempel.util.xml.dtd;
import name.panitz.crempel.util.xml.dtd.IsAtomic.*;
import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.Tuple3;
import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.UnaryFunction;
import java.util.List;
import java.util.ArrayList;
import java.util.TreeSet;
import java.util.Comparator;

public class DTDDefFlatten extends DTDDefVisitor<FlattenResult>{
  final String elementName;
  final boolean isGenerated;
  private int counter = 0;
  private String getNextName(){
    counter=counter+1;
    return elementName+"_"+counter;
  }

```



```java
  public DTDDefFlatten(boolean g,String n){elementName=n;isGenerated=g;}

  public FlattenResult eval(DTDPCData x){
    return single((DTDDef)x);}
  public FlattenResult eval(DTDTagName x){
    return single((DTDDef)x);}
  public FlattenResult eval(DTDEmpty x){
    return single((DTDDef)x);}
  public FlattenResult eval(DTDAny x){
    return single((DTDDef)x);}
  public FlattenResult eval(DTDPlus x){
    if (IsAtomic.isAtomic(x.getDtd())) return single((DTDDef)x);
    return flattenModified(elementName,x.getDtd()
          ,new UnaryFunction<DTDDef,DTDDef>(){
             public DTDDef eval(DTDDef dtd){return new DTDPlus(dtd);}
           });
  }
  public FlattenResult eval(DTDStar x){
    if (IsAtomic.isAtomic(x.getDtd())) return single((DTDDef)x);
    return
     flattenModified(elementName,x.getDtd()
          ,new UnaryFunction<DTDDef,DTDDef>(){
             public DTDDef eval(DTDDef dtd){return new DTDStar(dtd);}
           });
  }
  public FlattenResult eval(DTDQuery x){
    if (IsAtomic.isAtomic(x.getDtd())) return single((DTDDef)x);
    return flattenModified(elementName,x.getDtd()
          ,new UnaryFunction<DTDDef,DTDDef>(){
             public DTDDef eval(DTDDef dtd){return new DTDQuery(dtd);}
           });
  }
  public FlattenResult eval(DTDSeq x){
    return flattenPartList(x.getSeqParts()
          ,new UnaryFunction<List<DTDDef>,DTDDef>(){
             public DTDDef eval(List<DTDDef> dtds){
               return new DTDSeq(dtds);}
           });
  }
  public FlattenResult eval(DTDChoice x){
    return flattenPartList(x.getChoiceParts()
          ,new UnaryFunction<List<DTDDef>,DTDDef>(){
             public DTDDef eval(List<DTDDef> dtds){
System.out.println("the new choice"+dtds);
               return new DTDChoice(dtds);}
           });
   }

  private FlattenResult  single(DTDDef x){return new FlattenResult(x);}

```

```java
  private FlattenResult flattenModified
          (final String orgElem,DTDDef content
          ,UnaryFunction<DTDDef,DTDDef> constr){
    List<Tuple3<Boolean,String,DTDDef>> result
     = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
    if (needsNewElement(content)){
    System.out.println("owo needs new element: "+content );
      final String newElemName
        = ShowType.typeToIdent(ShowType.showType(content));

      result.add(new Tuple3<Boolean,String,DTDDef>
         (true,newElemName,content));
      return new FlattenResult
              (constr.eval(new DTDTagName(newElemName)),result);
    }
    System.out.println("does not need new element");
    FlattenResult innerRes  = content.visit(this);
    System.out.println(innerRes);
    return new FlattenResult(constr.eval(innerRes.e1),innerRes.e2);
  }

  private FlattenResult flattenPartList
      (List<DTDDef> parts,UnaryFunction<List<DTDDef>,DTDDef> constr){
    final List<Tuple3<Boolean,String,DTDDef>> result
      = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
    if (parts.size()==1) {return single(parts.get(0));}

    List<DTDDef> newParts = new ArrayList<DTDDef>();
    for (DTDDef part:parts){
      if (IsAtomic.isAtomic(part)) {System.out.println("atomic part:"+part);newPa
      else if (needsNewElement(part)){
        final String newElemName
          = ShowType.typeToIdent(ShowType.showType(part));
        result.add(new Tuple3<Boolean,String,DTDDef>
             (true,newElemName,part));
        newParts.add(new DTDTagName(newElemName));
      }else{
        FlattenResult innerRes  = part.visit(this);
        newParts.add(innerRes.e1);
        result.addAll(innerRes.e2);
      }
    }
    return new FlattenResult(constr.eval(newParts),result);

  }

  static private boolean needsNewElement(DTDDef d){
    return
     (d instanceof DTDSeq &&  ((DTDSeq)d).getSeqParts().size()>1)
            ||
```



```java
     (d instanceof DTDChoice &&((DTDChoice)d).getChoiceParts().size()>1);
  }

  static public List<Tuple3<Boolean,String,DTDDef>>
       flattenDefList(List<Tuple3<Boolean,String,DTDDef>> defs){
    boolean changed = true;
    List<Tuple3<Boolean,String,DTDDef>> result = defs;
    while (changed){
      Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>> once
        = flattenDefListOnce(result);
      changed=once.e1;
      result=once.e2;
    }

    TreeSet<Tuple3<Boolean,String,DTDDef>> withoutDups
      = new TreeSet<Tuple3<Boolean,String,DTDDef>>(
         new Comparator<Tuple3<Boolean,String,DTDDef>>(){
           public int compare(Tuple3<Boolean,String,DTDDef> o1
                             ,Tuple3<Boolean,String,DTDDef> o2){
              return o1.e2.compareTo(o2.e2);
           }
         });
    withoutDups.addAll(result);

    result = new ArrayList<Tuple3<Boolean,String,DTDDef>> ();
    result.addAll(withoutDups);
    return result;
  }

  private static Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>>
       flattenDefListOnce(List<Tuple3<Boolean,String,DTDDef>> defs){

    final List<Tuple3<Boolean,String,DTDDef>> result
     = new ArrayList<Tuple3<Boolean,String,DTDDef>>();

    boolean changed = false;

    for (Tuple3<Boolean,String,DTDDef> def:defs){
      final FlattenResult singleResult
        = def.e3.visit(new DTDDefFlatten(def.e1,def.e2));
      changed=changed||singleResult.e2.size()>0;
      result.addAll(singleResult.e2);
      result.add(
       new Tuple3<Boolean,String,DTDDef>(
         singleResult.e2.isEmpty()&&def.e1,def.e2,singleResult.e1));
    }
    return new Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>>
                    (changed,result);
  }
```

```
  public static FlattenResult flatten(DTDDef def,String n){
   return def.visit(new DTDDefFlatten(false,n));}
}


```

IsAtomic.java
```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;

public class IsAtomic extends DTDDefVisitor<Boolean>{
  public Boolean eval(DTDPCData x){return true;}
  public Boolean eval(DTDTagName x){return true;}
  public Boolean eval(DTDEmpty x){return true;}
  public Boolean eval(DTDAny x){return true;}
  public Boolean eval(DTDPlus x){return x.getDtd().visit(this);}
  public Boolean eval(DTDStar x){return x.getDtd().visit(this);}
  public Boolean eval(DTDQuery x){return x.getDtd().visit(this);}
  public Boolean eval(DTDSeq x){return false;}
  public Boolean eval(DTDChoice x){return false;}

  public static Boolean isAtomic(DTDDef def ){
   return def.visit(new IsAtomic());}
}
```

4.3.6 Main generation class

GenerateClassesForDTD.java
```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;
import java.util.ArrayList;
import java.io.*;
import name.panitz.crempel.util.*;

public class GenerateClassesForDTD{
  public static void generateAll
     (String paket,String path,String n,List<Tuple2<String,DTDDef>>dtds){
    List<Tuple3<Boolean,String,DTDDef>> xs
      = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
    for (Tuple2<String,DTDDef> t:dtds)
      xs.add(new Tuple3<Boolean,String,DTDDef>(false,t.e1,t.e2));


    xs = DTDDefFlatten.flattenDefList(xs);

    System.out.println("vereinfacht und flachgemacht");
    for (Tuple3<Boolean,String,DTDDef> t:xs){
```



```
      System.out.println(t.e2);
      System.out.println(t.e3);
      System.out.println("");
    }


    final String parserType = dtds.get(0).e1;
    try{
      Writer out = new FileWriter(path+"/"+n+"Parser"+".java");
      out.write("package "+paket+";\n");

      out.write("import name.panitz.crempel.util.xml.parslib.*;\n");
      out.write("import name.panitz.crempel.util.*;\n");
      out.write("import java.util.*;\n");
      out.write("import org.w3c.dom.Node;\n\n");

      out.write("public class "+n+"Parser ");
      out.write("extends AbstractParser<"+parserType+">{\n");

      out.write("public ParseResult<"+parserType+"> ");
      out.write("parse(List<Node> xs){");
      out.write("  return getV"+parserType+"().parse(xs);}\n\n");

      for (Tuple3<Boolean,String,DTDDef> x :xs)
        out.write(WriteParser.writeParser(x.e3,x.e2,x.e1));

      out.write("}");
      out.close();
    }catch (IOException e){e.printStackTrace();}
    GenerateADT.generateADT(paket,path,xs);
  }
}
```

4.3.7 Main generator class

We provide the main class for generating the parser and algebraic type for a given dtd. Two arguments are passed on the command line. The file name of the DTD file and a package for the generated classes.

MainDTDParse.java
```
package name.panitz.crempel.util.xml.dtd.parser;

import java.io.*;
import java.util.*;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.xml.dtd.*;

public class MainDTDParse {
```

```
  public static void main(String [] args){
    try{
      final String dtdFileName = args[0];
      final String packageName = args[1].replace('/','.');

      File f = new File(dtdFileName);
      final String path
          = f.getParentFile()==null?"":f.getParentFile().getPath();

      final DTD parser = new DTD(new FileReader(f));


      final Tuple3<List<Tuple2<String,DTDDef>>,Object,String> dtd
          = parser.dtd();

      for (Tuple2<String,DTDDef> t:dtd.e1){
        System.out.println(t.e1);
        System.out.println(t.e2);
        System.out.println("");
      }

      GenerateClassesForDTD
          .generateAll(packageName,path,dtd.e3,dtd.e1);
    }catch (Exception _){_.printStackTrace();System.out.println(_);}
  }
}
```

```
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album> java -c
lasspath /home/sep/fh/xmlparslib/examples/classes/:/home/sep/fh/java1.5/exa
mples/classes/:/home/sep/fh/adt/examples/classes/ name.panitz.crempel.util.
xml.dtd.parser.MainDTDParse album.dtd name.panitz.album
package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class Choice_String_author{

  CChoice_String_authorString(String theString);
  CChoice_String_authorauthor(author theauthor);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class Choice_lp_cd_mc{

  CChoice_lp_cd_mclp(lp thelp);
  CChoice_lp_cd_mccd(cd thecd);
  CChoice_lp_cd_mcmc(mc themc);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class artist{
Con(String pcdata);}

package name.panitz.album;
```



```
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class author{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class cd{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class lp{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class mc{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class musiccollection{
Con(List<Choice_lp_cd_mc> xs);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class note{
Con(List<Choice_String_author> xs);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class recordingyear{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class timing{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class title{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class track{
Con(title thetitle,Maybe<timing> theMaybe_timing_);}
```

```
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album> ls
CChoice_String_authorString.java    lpVisitor.java
CChoice_String_authorauthor.java    mc.java
CChoice_lp_cd_mccd.java             mcVisitable.java
CChoice_lp_cd_mclp.java             mcVisitor.java
CChoice_lp_cd_mcmc.java             musiccollection.java
Choice_String_author.java           musiccollectionParser.java
Choice_String_authorVisitable.java  musiccollectionVisitable.java
Choice_String_authorVisitor.java    musiccollectionVisitor.java
Choice_lp_cd_mc.java                note.java
Choice_lp_cd_mcVisitable.java       noteVisitable.java
Choice_lp_cd_mcVisitor.java         noteVisitor.java
album.dtd                           recordingyear.java
artist.java                         recordingyearVisitable.java
artistVisitable.java                recordingyearVisitor.java
artistVisitor.java                  timing.java
author.java                         timingVisitable.java
authorVisitable.java                timingVisitor.java
authorVisitor.java                  title.java
cd.java                             titleVisitable.java
cdVisitable.java                    titleVisitor.java
cdVisitor.java                      track.java
lp.java                             trackVisitable.java
lpVisitable.java                    trackVisitor.java
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album>
```

4.3.8 JaxB

```
sep@linux:~/fh/xmlparslib/> ~/jwsdp-1.3/jaxb/bin/xjc.sh -dtd
  album.dtd -p name.panitz.jaxb.album
parsing a schema...
compiling a schema...
name/panitz/jaxb/album/impl/runtime/GrammarInfo.java
name/panitz/jaxb/album/impl/runtime/AbstractUnmarshallingEventHandlerImpl.java
name/panitz/jaxb/album/impl/runtime/PrefixCallback.java
name/panitz/jaxb/album/impl/runtime/Discarder.java
name/panitz/jaxb/album/impl/runtime/ValidatableObject.java
name/panitz/jaxb/album/impl/runtime/SAXUnmarshallerHandlerImpl.java
name/panitz/jaxb/album/impl/runtime/ContentHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/ValidatorImpl.java
name/panitz/jaxb/album/impl/runtime/UnmarshallerImpl.java
name/panitz/jaxb/album/impl/runtime/GrammarInfoFacade.java
name/panitz/jaxb/album/impl/runtime/XMLSerializable.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingEventHandler.java
name/panitz/jaxb/album/impl/runtime/DefaultJAXBContextImpl.java
name/panitz/jaxb/album/impl/runtime/SAXMarshaller.java
name/panitz/jaxb/album/impl/runtime/GrammarInfoImpl.java
name/panitz/jaxb/album/impl/runtime/MSVValidator.java
name/panitz/jaxb/album/impl/runtime/UnmarshallableObject.java
name/panitz/jaxb/album/impl/runtime/SAXUnmarshallerHandler.java
name/panitz/jaxb/album/impl/runtime/ErrorHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/NamespaceContext2.java
name/panitz/jaxb/album/impl/runtime/Util.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingEventHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/ValidationContext.java
name/panitz/jaxb/album/impl/runtime/ValidatingUnmarshaller.java
name/panitz/jaxb/album/impl/runtime/MarshallerImpl.java
name/panitz/jaxb/album/impl/runtime/XMLSerializer.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingContext.java
name/panitz/jaxb/album/impl/runtime/NamespaceContextImpl.java
name/panitz/jaxb/album/impl/ArtistImpl.java
```



```
name/panitz/jaxb/album/impl/AuthorImpl.java
name/panitz/jaxb/album/impl/CdImpl.java
name/panitz/jaxb/album/impl/JAXBVersion.java
name/panitz/jaxb/album/impl/LpImpl.java
name/panitz/jaxb/album/impl/McImpl.java
name/panitz/jaxb/album/impl/MusiccollectionImpl.java
name/panitz/jaxb/album/impl/NoteImpl.java
name/panitz/jaxb/album/impl/RecordingyearImpl.java
name/panitz/jaxb/album/impl/TimingImpl.java
name/panitz/jaxb/album/impl/TitleImpl.java
name/panitz/jaxb/album/impl/TrackImpl.java
name/panitz/jaxb/album/Artist.java
name/panitz/jaxb/album/Author.java
name/panitz/jaxb/album/Cd.java
name/panitz/jaxb/album/Lp.java
name/panitz/jaxb/album/Mc.java
name/panitz/jaxb/album/Musiccollection.java
name/panitz/jaxb/album/Note.java
name/panitz/jaxb/album/ObjectFactory.java
name/panitz/jaxb/album/Recordingyear.java
name/panitz/jaxb/album/Timing.java
name/panitz/jaxb/album/Title.java
name/panitz/jaxb/album/Track.java
name/panitz/jaxb/album/jaxb.properties
name/panitz/jaxb/album/bgm.ser
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album>
```

4.4 Conclusion

We applied techniques as known from functional programming to Java. Algebraic types and parser combinators enables us to write a complicated library tool, which generates classes and parsers for a given DTD. Java's generic types provide some good means to express these concepts in a static type safe manner.

With *JaxB (JavaTM Architecture for XML Binding)* Sun provides a library and tool which addresses the same problem: for a given schema generate classes and a parser. However, the resulting classes and parsers in *JaxB* do not provide the visitor pattern for the resulting class and do not use the concepts as known from functional programming.

4.5 Javacc input file for DTD parser

In this section you can find a simple not complete parser for DTDs as input grammar for the parser generator *javacc*.

dtd.jj

```
options {STATIC=false;}

PARSER_BEGIN(DTD)
package name.panitz.crempel.util.xml.dtd.parser;

import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;
```

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

public class DTD {}
PARSER_END(DTD)

TOKEN :
{<ELEMENTDEC: "<!ELEMENT">
|<DOCTYPE:    "<!DOCTYPE">
|<ATTLIST:    "<!ATTLIST">
|<REQUIRED:   "#REQUIRED">
|<IMPLIED:    "#IMPLIED">
|<EMPTY: "EMPTY">
|<PCDATA: "#PCDATA">
|<CDATA: "CDATA">
|<ANY: "ANY">
|<SYSTEM: "SYSTEM">
|<PUBLIC: "PUBLIC">
|<GR: ">">
|<QMARK: "?">
|<PLUS: "+">
|<STAR: "*">
|<#NameStartCar: [":","A"-"Z","_","a"-"z"
                 ,"\u00C0"-"\u00D6"
                 ,"\u00D8"-"\u00F6"
                 ,"\u00F8"-"\u02FF"
                 ,"\u0370"-"\u037D"
                 ,"\u037F"-"\u1FFF"
                 ,"\u200C"-"\u200D"
                 ,"\u2070"-"\u218F"
                 ,"\u2C00"-"\u2FEF"
                 ,"\u3001"-"\uD7FF"
                 ,"\uF900"-"\uFDCF"
                 ,"\uFDF0"-"\uFFFD"]>
//                 ,"\u10000"-"\uEFFFF"]>
|<#InnerNameChar: ["-", ".","0"-"9", "\u00B7"
                  ,"\u0300"-"\u036F"
                  ,"\u203F"-"\u2040"]>
|<#NameChar: <InnerNameChar>|<NameStartCar>>
|<Name: <NameStartCar> (<NameChar>)* >

|<#ALPHA:       ["a"-"z","A"-"Z","_","."]>
|<#NUM:                ["0"-"9"]>
|<#ALPHANUM:    <ALPHA> | <NUM>>
|<EQ: "=">
|<BAR: "|">
|<LPAR: "(">
|<RPAR: ")">
|<LBRACKET: "{">
|<RBRACKET: "}">
```



```
|<LSQBRACKET: "[">
|<RSQBRACKET: "]">
|<LE: "<">
|<SEMICOLON: ";">
|<COMMA: ",">
|<QUOTE: "\"">
|<SINGLEQUOTE: "'">
}

SKIP :
{< ["\u0020","\t","\r","\n"] >}



void externalID():
{}
{<SYSTEM> systemLiteral()
 |<PUBLIC> systemLiteral() systemLiteral()
}

void systemLiteral():{}
{<QUOTE><Name><QUOTE>|<SINGLEQUOTE><Name><SINGLEQUOTE>
}

Tuple3 dtd():
{
  Token nameToken;
  List els = new ArrayList();
  List atts = new ArrayList();
  Tuple2 el;
}
{<DOCTYPE> nameToken=<Name> externalID() <LSQBRACKET>

   (el=elementdecl(){ els.add(el);}
   |AttlistDecl())*
 <RSQBRACKET><GR>
  {return new Tuple3(els,atts,nameToken.toString());}
}

Tuple2 elementdecl():
{Token nameToken;
 DTDDef content;}
{<ELEMENTDEC> nameToken=<Name>  content=contentspec()  <GR>
  {return new Tuple2(nameToken.toString(),content);}
}


DTDDef contentspec():
{DTDDef result;}
{  <EMPTY>{result=new DTDEmpty();}
```

```
 | <ANY>{result=new DTDAny();}
 | (<LPAR>(result=Mixed()
          |result=children()))
 { return result;}
}

DTDDef children():
{ List cps;
  DTDDef cp;
  Modifier mod = Modifier.none;
  boolean wasChoice = true;
}
{cp=cp()
  (cps=choice()| cps=seq(){wasChoice=false;})
   {cps.add(0,cp);}
   <RPAR>(<QMARK>{mod=Modifier.query;}
     |<STAR>{mod=Modifier.star;}
     |<PLUS>{mod=Modifier.plus;})?
  {DTDDef result=wasChoice?ParserAux.createDTDChoice(cps)
                          :ParserAux.createDTDSeq(cps);
   return mod.mkDTDDef(result);
  }
}

List choice():
{ DTDDef cp;
  List result = new ArrayList();}
{(<BAR> cp=cp() {result.add(cp);} )+
  {return result;}
}

DTDDef cp():
{ Token nameToken=null;
  List cps=new ArrayList();
  DTDDef cp;
  Modifier mod=Modifier.none;
  boolean wasChoice = true;
  boolean wasTagName = false;
}
{((nameToken = <Name>{wasTagName=true;})
 |(<LPAR>cp=cp()
         (cps=choice()|cps=seq(){wasChoice=false;})
    {cps.add(0,cp);}
   <RPAR>
  )
 )
     (<QMARK>{mod=Modifier.query;}
     |<STAR>{mod=Modifier.star;}
     |<PLUS>{mod=Modifier.plus;})?
{
```



```
 DTDDef result;
 if (wasTagName) result=new DTDTagName(nameToken.toString());
 else result=wasChoice?ParserAux.createDTDChoice(cps)
                      :ParserAux.createDTDSeq(cps);
 return mod.mkDTDDef(result);
}
}

List seq():
{  List result = new ArrayList();
   DTDDef cp;
}
{(<COMMA> cp=cp(){result.add(cp);} )*
  {return result;}
}

DTDDef Mixed():
{Token nameToken;
 List xs = new ArrayList();
 Modifier mod=Modifier.none;
}
{  <PCDATA> {xs.add(new DTDPCData());}
  ((<BAR> nameToken=<Name>
    {xs.add(new DTDTagName(nameToken.toString()));}
   )* <RPAR>(<STAR>{mod=Modifier.star;})?)
{return mod.mkDTDDef(ParserAux.createDTDChoice(xs));}

}

void AttlistDecl():
{Token nameToken;}
{<ATTLIST> nameToken=<Name> (AttDef() )*
{}}

void AttDef():
{Token nameToken;
 boolean isRequired;}
{nameToken=<Name> AttType() isRequired=DefaultDecl()
  {}
}

void AttType():
{}
{ StringType()}//|TokenizedType()|EnumeratedType()}

void StringType():
{}
{<CDATA>}
```

```
boolean DefaultDecl():
{ boolean isRequired=false;}

{(<REQUIRED>{isRequired=true;} | <IMPLIED>)
//| (('#FIXED' S)? AttValue)
{return isRequired;}
}
```

ParserAux.java

```
package name.panitz.crempel.util.xml.dtd.parser;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;

public class ParserAux{
  static public DTDDef createDTDSeq(List xs){
    return new DTDSeq((List<DTDDef>) xs);
  }
  static public DTDDef createDTDChoice(List xs){
    return new DTDChoice((List<DTDDef>) xs);
  }
}
```

Modifier.java

```
package name.panitz.crempel.util.xml.dtd.tree;

public enum Modifier { none, star, plus, query;

  public String toString(){
    switch(this) {
          case star:   return "*";
          case plus:   return "+";
          case query:  return "?";
    }
    return "";
  }

  public DTDDef mkDTDDef(DTDDef dtd){
    switch(this) {
      case star:   return new DTDStar(dtd);
      case plus:   return new DTDPlus(dtd);
      case query:  return new DTDQuery(dtd);
    }
    return dtd;
  }
}
```

Kapitel 5

Javas Trickkisten: Bibliotheken und Mechanismen

5.1 Webapplikationen mit Servlets

In diesen Kapitel betrachten wir die Möglichkeit mit Javaprogrammen die Funktionalität eines Webservers zu erweitern. Damit lassen sich dynamische Webseiten erzeugen. Eine Webadresse wird vom Server umgeleitet auf ein Javaprogramm, das die entsprechende Antwortseite erzeugt.

Dieses Kapitel versucht ein einfaches Kochbuch zum Schreiben von Webanwendungen zu sein.

5.1.1 Servlet Container

Standardmäßig unterstützt ein Webserver nicht die Fäigkeit Javaprogramme für bestimmte URLs aufzurufen. Hierzu ist der Webserver zunächst um eine Komponente zu erweitern, die ihn dazu befähigt. Eine solche Komponente wird *servlet container* bezeichnet. Einer der gebräuchlichsten *servlet container* ist der *tom cat* (`http://jakarta.apache.org/tomcat/`) . Dieser kann genutzt werden um z.B. den *apache* Webserver zu erweitern, so daß auf ihm servlets laufen können. Der *tom cat* selbst ist jedoch auch bereits ein eigener Webserver und kann, so wie wir es in diesem Kapitel tun, auch als eigenständiger Webserver betrieben werden.

5.1.2 Struktur einer Webapplikation

Der *tom cat* kommt mit einem *ant* Skript zum Bauen einer Webapplikation. Neben den üblichen Zielen wie `compile` enthält dieses Skript die Ziele `install`, `remove` und `reload` um eine Webapplikation auf einen laufenden *tom cat* zu instalieren.

Dieses *ant* Skript erwartet eine bestimmte Ordnerstruktur, der folgenden Form:

```
applikationsname
  |--- src
```

```
  |--- docs
  |--- web
        |--- WEB-INF
        |--- images
  |--- build
  |--- dist
```

Im Order `src` sind alle Java-sourcen zu plazieren. Im Order `web` stehen die einzelnen Webseiten für die Webapplikation. Im Unterordner `WEB-INF` ist die Hauptkonfigurationsdatei `web.xml`. In dieser sind die einzelnen Servlets mit ihren zugehörigen Javaklassen und URLs verzeichnet.

5.1.3 Anwendungsdaten

Wir wollen eine sehr kleine Webanwendung in diesem Kapitel entwickeln, in der Emailadressen zu Namen verwelatet werden. Wir definieren für unser kleines Beispiel die Struktur der Anwendungsdaten in einer DTD.

AdressenType.dtd

```
<!DOCTYPE Addresses SYSTEM "AddressenType.dtd" [
  <!ELEMENT Addresses  (Address)+ >
  <!ELEMENT Address  (Fullname,Email) >
  <!ELEMENT Fullname  (#PCDATA) >
  <!ELEMENT Email  (#PCDATA) >
]>
```

Für diese DTD können wir uns mit dem im letzten Kapitel entwickelten Tool Klassen und einen Baumparser generieren lassen.

5.1.4 Anwendungslogik

In einem nächsten Schritt entwickeln wir die Anwendungslogik unser Webapplikation. Wir wollen dabei ein Adressenobjekt bearbeiten können. Dabei wollen wir Adressen nachschlagen, hinzufügen und entfernen können. Wir sehe eine entsprechende Schnittstelle vor:

Adressverwaltung.java

```java
package name.panitz.webexample.address;
import java.io.IOException;
import java.io.Writer;

public interface Adressverwaltung {
  Addresses getAddresses( );
  void save()throws Exception;
  Email lookup(Fullname n);
  void add(Address a) throws Exception;
  void remove(Fullname n) throws Exception;
  void writeAddresses(Writer w) throws IOException;
}
```


Die meisten Methoden lassen sich unabhängig von der Persistenzschicht der Adressverwaltung implementieren. Dieses können wir in einer abstrakten Klasse erledigen:

AbstractAdressverwaltung.java

```java
package name.panitz.webexample.address;
import java.io.IOException;
import java.io.Writer;

abstract public class AbstractAdressverwaltung
                          implements Adressverwaltung {
  synchronized public Email lookup(Fullname n){
    for (Address address:getAddresses().getXs()){
      if (address.getTheFullname().equals(n))
        return address.getTheEmail();
    }
    return null;
  }

  synchronized public void add(Address a) throws Exception{
    getAddresses().getXs().add(a);
    save();
  }

  synchronized public void remove(Fullname n) throws Exception{
    for (Address address:getAddresses().getXs()){
      if (address.getTheFullname().equals(n)){
        getAddresses().getXs().remove(address);
        save();
        return;
      }
    }
  }

  public void writeAddresses(Writer w) throws IOException{
    final Addresses adrs = getAddresses();
    w.write("<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>");
    w.write("\n<Addresses>");

    for (Address address:adrs.getXs()){
      w.write("\n <Address><Fullname>");
      w.write(address.getTheFullname().getPcdata());
      w.write("</Fullname><Email>");
      w.write(address.getTheEmail().getPcdata());
      w.write("</Email></Address>");
    }
    w.write("\n</Addresses>");
  }
}
```

In unserem Beispiel benutzen wir die einfachste Art der Datenhaltung in der Persistenzschicht, die einer Datei. Wir implementieren eine auf einer Datei basierende Adressverwaltung. Dabei werden wir über eine Fabrikmethode dafür sorgen, daß für eine bestimmte Adressdatei stets nur ein Objekt zur Adressverwaltung intsnziiert wird. Damit wird verhindert, daß mehrere Adressverwaltungsobjekte dieselbe Datei manipulieren und sich dabei ins Gehege kommen.

AdressFile.java

```
package name.panitz.webexample.address;
import java.io.*;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import java.util.*;
import javax.xml.parsers.*;

import name.panitz.crempel.util.xml.*;

public class AdressFile extends AbstractAdressverwaltung{
  static Map<File,AdressFile> instances
   = new HashMap<File,AdressFile>();

  File source;
  Addresses addresses = null;

  public static AdressFile getInstance(String source)
                                           throws Exception{
    File dataFile = new File(source);
    AdressFile instance = instances.get(dataFile);

    if (instance == null){
      instance = new AdressFile(dataFile);
      instances.put(dataFile,instance);
    }
    return instance;
  }

  private AdressFile(File source) throws Exception{
    this.source=source;

    Document doc =
      DocumentBuilderFactory
       .newInstance()
       .newDocumentBuilder()
       .parse(source);
    final List<Node> ns = new ArrayList<Node>();
    ns.add(doc.getDocumentElement());
    addresses = new AddressesParser().parse(ns).e1;
  }

  synchronized public Addresses getAddresses( ){return addresses;}
```



```
  synchronized public void save()throws Exception{
    FileWriter write = new FileWriter(source);
    writeAddresses(write);
    write.close();
  }
}
```

Wir ermöglichen die Konfiguration anderer Implementierungen der Adressverwaltung durch eine Favrikmethode, die den Namen der Klasse aus einer Systemeigenschaft liest.

AdressverwaltungFactory.java

```
package name.panitz.webexample.address;
import java.lang.reflect.Method;

public class AdressverwaltungFactory{
  static Method method=null;

  public static Adressverwaltung
                getAdressverwaltung(String source)throws Exception{
    if (method==null){
      String className
        = System.getProperty("adressverwaltung.klasse");
      if (className==null){
        className="name.panitz.webexample.address.AdressFile";
      }
      final Class verwalter=Class.forName(className);
      final Class[]  paramtypes = {String.class};
      method=verwalter.getMethod("getInstance",paramtypes);
    }
    final String[] params = {source};
    return (Adressverwaltung)method.invoke(null,params);
  }
}
```

5.1.5 Servletkonfiguration

Die wichtigste Konfigurationsdatei für eine Webapplikation ist die Datei `web.xml`. In ihr wird die Webapplikation mit ihren Servlets beschrieben. Für jedes Servlet wird ein Name vergeben. Dieser Name wird gebunden an eine Servletklasse, die die entsprechende Funktionalität zur Verfügung stellt und an die URL über den auf das Servlet zugegriffen wird. Das Format der Datei `web.xml` ist in einer DTD definiert. In unseren Beispiel sehen wir vier Servlets vor:

web.xml

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
```

```
  <display-name>Adressverwaltung</display-name>
  <description>
    Kleines Adressverwaltungstool.
  </description>

  <servlet>
    <servlet-name>ListServlet</servlet-name>
    <servlet-class>
       name.panitz.webexample.address.ListAddresses
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ListServlet</servlet-name>
    <url-pattern>/list</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>LookupServlet</servlet-name>
    <servlet-class>
       name.panitz.webexample.address.LookupAddress
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>LookupServlet</servlet-name>
    <url-pattern>/lookup</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>AddServlet</servlet-name>
    <servlet-class>
       name.panitz.webexample.address.AddAddress
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AddServlet</servlet-name>
    <url-pattern>/add</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>DeleteServlet</servlet-name>
    <servlet-class>
       name.panitz.webexample.address.DeleteAddress
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>DeleteServlet</servlet-name>
    <url-pattern>/delete</url-pattern>
  </servlet-mapping>

</web-app>
```


Für unsere Webapplikation sehen wir noch eine minimale Einstiegswebseite vor:

index.html

```
<html>
<head>
<title>Adressenverwaltung</title>
</head>
<body bgcolor="white">

<h1>einfache Adressverwaltung</h1>

<ul>
<li><a href="list">Auflisten der gespeicherten Adressen</a>.</li>
<li><a href="lookup.html">Adresse suchen</a>.</li>
<li><a href="add.html">Adresse hinzufügen</a>.</li>
<li><a href="delete.html">Adresse löschen</a>.</li>
</ul>
</body>
</html>
```

5.1.6 Servletklassen

Bisher haben wir die Anwendungslogik und die Persistenzschicht unserer Anwendung implementiert. Wir haben beschrieben, welche Dienste unsere Webanwendung über Servlets anbieten soll. Es sind nun die eigentlichen Servletklassen zu schreiben. Leider ist das Servlet API nicht Bestandteil der Java Bibliotheken aus dem Standard SDK. Die Jar-Datei und Dokumentation für das Servlet-API ist separat herunterzuladen. In der Distribution von *tomcat* ist das Servlet-API enthalten.

Ein Servlet ist eine Klasse, die die Schnittstelle `javax.servlet.Servlet` implementiert. Hierin finden sich fünf Methoden, zwei die Informationen über dass Servlet angeben, eine die bei der Initialisierung des Servlets vom Servletcontainer ausgeführt wird, eine die ausgeführt wird, wenn der Servletcontainer den Dienst abstellt und schließlich eine, die den eigentlichen Dienst beschreibt:

```
void destroy();
ServletConfig getServletConfig();
java.lang.String getServletInfo();
void init(ServletConfig config);
void service(ServletRequest req, ServletResponse res);
```

Zum schreiben von Servlets für HTTP wird nicht direkt diese Schnittstelle implementiert sondern eine abstrakte Klasse erweitert, die Klasse `javax.servlet.http.HttpServlet`, die ihrerseits die Schnittstelle Servlet implementiert. `HttpServlet` ist zwar eine abstrakte Klasse, es können also nicht direkt Objekte dieser Klasse instanziiert werden, aber die Klasse hat keine abstrakten Methoden.

Generierung von Seiten

Die einfachste Form eines Servlets schickt eine dynamisch erzeugte HTML-Seite an den Client. Hierzu ist die Methode `doGet` der Klasse `HttpServlet` zu überschreiben. Die Methode hat zwei Argumente:

- ein `HttpServletRequest`, in dem Informationen über den Aufruf des Servlets zu finden sind.
- eine `HttpServletResponse`, die zur Erzeugung der Antwortseite benutzt wird.

Auf der `HttpServletResponse` läßt sich der Typ des Inhalts setzen, in den meisten Fällen wahrscheinlich `text/html`. Weiterhin enthält es einen `PrintWriter`, auf den der Inhalt der zu generierenden Seite auszugeben ist.

Unter Berücksichtigung der Ausnahme, die auftreten können, läßt sich nun leicht ein Servlet implementieren. Die folgende Klasse realisiert das Servlet zum Auflisten der gespeicherten Adressen. Dabei wird als Adressdatei die Datei `adressen.xml` abgenommen. Diese soll im Hauptordner der Webapplikation liegen.

ListAddresses.java

```java
package name.panitz.webexample.address;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import java.io.File;

public final class ListAddresses extends HttpServlet {
  public void doGet(HttpServletRequest request
                    ,HttpServletResponse response)
        throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();

    writer.println("<html>");
    writer.println("<head>");
    writer.println("<title>Gespeicherte Adressen</title>");
    writer.println("</head>");
    writer.println("<body bgcolor=\"white\">");

    try{
      Adressverwaltung adv
         = AdressverwaltungFactory
              .getAdressverwaltung(getServletConfig()
                                    .getServletContext()
                                    .getRealPath(".")
```



```java
                                    +"/addresses.xml");

      writeAddressesAsHTML(adv.getAddresses(),writer);
    }catch(Exception e){writer.println(e);}
    writer.println("</body>");
    writer.println("</html>");
  }

  public static void writeAddressesAsHTML
                 (Addresses adrs,Writer w) throws IOException{
    w.write("\n<table>");
    w.write("\n<tr><th>Name</th><th>email</th></tr>");
    for (Address address:adrs.getXs()){
      w.write("\n <tr><td>");
      w.write(address.getTheFullname().getPcdata());
      w.write("</td><td>");
      w.write(address.getTheEmail().getPcdata());
      w.write("</td></tr>");
    }
    w.write("\n</table>");
  }
}
```

Lesen von Parametern

Im letzten Abschnitt haben wir ein Servlet geschrieben, das eine Html-Seite generiert. Dabei werdeb keine vom Client übermittelten Informationen genutzt. Um eine bestimmte Emailadresse aus unserem System abzufragen, müssen wir diese vom Client übermittelt bekomme. Hierfür gibt es in in Html das `form` Tag.

lookup Als nächstes Servlet implementieren wir die Nachfrage nach einer bestimmten Emailadresse. Hierzu benötigen wir die Html-Seite zur Eingabe des Namens nach dem gesucht wird.

lookup.html

```html
<html><body bgcolor="white">
<h1>Adresssuche</h1>
<form action="lookup" method="POST">
Name:
<input type="text" size="20" name="fullname">
<br>
<input type="submit">
</form>
</body>
</html>
```

Um jetzt auf die durch ein `POST` vom Client an den Webserver übermittelte Daten zugreifen zu können, überschreiben wir die Methode `doPost` der Klasse `HttpServlet`. Hier können wir auf dem `HttpServletRequest`-Objekt mit der Methode `getParameter` auf den Wert eines mitgeschickten Eingabefeldes zugreifen, und dieses für die Antwort benutzen.

LookupAddress.java

```java
package name.panitz.webexample.address;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;
import java.io.File;

import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.util.*;

public final class LookupAddress extends HttpServlet {
  public void doPost(HttpServletRequest request,
                    HttpServletResponse response)
        throws IOException, ServletException {
    response.setContentType("text/html");
    final PrintWriter writer = response.getWriter();

    writer.println("<html>");
    writer.println("<head>");
    writer.println("<title>Gespeicherte Adressen</title>");
    writer.println("</head>");
    writer.println("<body bgcolor=white>");

    try{
      final String fullname = request.getParameter("fullname");
      final Adressverwaltung adv
         = AdressverwaltungFactory
              .getAdressverwaltung(getServletConfig()
                                    .getServletContext()
                                    .getRealPath(".")
                                    +"/addresses.xml");
      final Email email = adv.lookup(new Fullname(fullname));
      if (email!=null){
        writer.println("Die gesuchte Adresse ist:<br />" );
        writer.println(email.getPcdata());
      }else
       writer.print("zum gesuchten Namen ist");
       writer.println(" keine Adresse gespeichert.");
    }catch(Exception e){writer.println(e);}
    writer.println("</body>");
    writer.println("</html>");
  }

  public void
   doGet(HttpServletRequest request,HttpServletResponse response)
                            throws IOException, ServletException{
```



```java
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("GET Request. No Form Data Posted");
  }
}
```

add Das Hinzufügen neuer Adresseinträge läuft analog. Zunächst die entsprechende Html-Seite.

add.html

```html
<html><body bgcolor="white">
<h1>Adresssuche</h1>
<form action="add" method=POST>
Name:
<input type="text" size="20" name="fullname">
<br>
Email:
<input type="text" size="20" name="email">
<br>
<input type="submit">
</form>
</body>
</html>
```

Folgend die entsprechende Servletimplementierung:

AddAddress.java

```java
package name.panitz.webexample.address;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import java.io.File;
import java.util.*;

public final class AddAddress extends HttpServlet {
  public void
   doGet(HttpServletRequest request,HttpServletResponse response)
                            throws IOException, ServletException{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("GET Request. No Form Data Posted");
  }

```

```java
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
        throws IOException, ServletException {
    response.setContentType("text/html");
    final PrintWriter writer = response.getWriter();

    writer.println("<html>");
    writer.println("<head>");
    writer.println("<title>Gespeicherte Adressen</title>");
    writer.println("</head>");
    writer.println("<body bgcolor=white>");

    try{
      final String fullname = request.getParameter("fullname");
      final String email = request.getParameter("email");
      final Adressverwaltung adv
         = AdressverwaltungFactory
              .getAdressverwaltung(getServletConfig()
                                   .getServletContext()
                                   .getRealPath(".")
                                   +"/addresses.xml");
      adv.add(new Address
                (new Fullname(fullname),new Email(email)));
      writer.println("added: "+fullname+": "+email);
    }catch(Exception e){writer.println(e);}
    writer.println("</body>");
    writer.println("</html>");
  }
}
```

delete Zu guter letzt noch das analoge Löschen von Adresseinträgen. zunächst die Html-Seite:

delete.html

```html
<html><body bgcolor="white">
<h1>Adresssuche</h1>
<form action="delete" method=POST>
Name:
<input type="text" size="20" name="fullname">
<br>
<input type="submit">
</form>
</body>
</html>

```

Die entsprechende Servletimplementierung:

DeleteAddress.java

```java
package name.panitz.webexample.address;

```



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

import javax.servlet.ServletException;
import javax.servlet.http.*;

import java.io.File;
import java.util.*;

public final class DeleteAddress extends HttpServlet {
  public void
   doGet(HttpServletRequest request, HttpServletResponse response)
                    throws IOException, ServletException{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("GET Request. No Form Data Posted");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
        throws IOException, ServletException {
    response.setContentType("text/html");
    final PrintWriter writer = response.getWriter();


    writer.println("<html>");
    writer.println("<head>");
    writer.println("<title>Gespeicherte Adressen</title>");
    writer.println("</head>");
    writer.println("<body bgcolor=white>");

    try{
      final String fullname = request.getParameter("fullname");
      final Adressverwaltung adv
         = AdressverwaltungFactory
              .getAdressverwaltung(getServletConfig()
                                   .getServletContext()
                                   .getRealPath(".")
                                   +"/addresses.xml");
      final Fullname name = new Fullname(fullname);
      adv.remove(name);
      writer.println("removed: "+fullname);
    }catch(Exception e){writer.println(e);}
    writer.println("</body>");
    writer.println("</html>");
  }
}
```

Die gesammte Webapplikation läßt sich mit dem beim *tomcat* mitgelieferten *ant*-Skript installieren.

5.1.7 Java Server Pages

In den obigen Servletbeispielen konnte man schon sehen, daß ein Servlet zwei sehr unterschiedliche Aufgabe wahrnimmt:

- die angefragten Daten berechnen.
- die angefragte Html-Seite mit den berechneten Daten generieren.

Dieses sind zwei sehr unterschiedliche Aufgaben, die wahrscheinlich in der Regel auch personell getrennt entwickelt werden. Das Schreiben der Html-Seite für die Ausgabe kommt bei Servlets etwas zu kurz. Es wird nur Java-Code geschrieben. Die Ausgabe muß mühsam mit `println`-Aufrufen generiert werden.

Diesem Problem begegnen *Java Server Pages(JSP)*. Diese sehen im Prinzip wie Html-Seiten aus, in denen an ausgesuchten Stellen kleine Java Code Schnipsel stehen. Der Java Code berechnet an diesen Stellen, wie der Wert für die Webseite dort berechnet wird.

Die Syntax für JSP ist recht umfangreich und wir werden hier nur ein kleines Beispiel betrachten:

Beispiel:
Die Seite zum Auflisten der Adressen in unserer Webapplikation läßt sich wesentlich kürzer als JSP schreiben.

Adressliste.jsp

```
<%@ page contentType="text/html"%>
<html><head>
<title>Adressliste</title></head>
<body bgcolor="white">
<%
     java.io.PrintWriter writer = response.getWriter();
     name.panitz.webexample.address
     .Adressverwaltung adv
         =  name.panitz.webexample.address
              .AdressverwaltungFactory
              .getAdressverwaltung(getServletConfig()
                                   .getServletContext()
                                   .getRealPath(".")
                                   +"/addresses.xml");
     name.panitz.webexample.address.ListAddresses
     .writeAddressesAsHTML(adv.getAddresses(),writer);
 %>
</body>
</html>

```


Interessant ist das Ausführungsmodell von JSP. Aus der JSP-Seite wird eine Javaklasse generiert, die mit einem Javakompilierer übersetzt wird. Die generierte Klasse wird dann als Servlet geladen. All dieses wird vom Servlet Conainer übernommen. Eine JSP braucht nicht von Hand übersetzt zu werden. Allerdings bekommt man auch erst vom Servlet Container einer Fehlermeldung, wenn das generierte Servlet nicht übersetzbar ist.

5.2 Reflektion

Java hat die interessante Eigenschaft, daß es in Java möglich ist, über das vorliegende API Anfragen zu stellen. Hierfür gibt es die Reflektion. Mit Reflektion lassen sich folgende Aufgaben lösen:

- es läßt sich die Klasse erfragen, von der ein Objekt erzeugt wurde.
- es lassen sich Informationen über bestimmte Klassen erfragen
- neue Instanzen lassen sich von Klassen erzeugen, von denen nur der Name als String vorliegt.
- Methoden und Felder können benutzt werden, obwohl deren Name erst während der Laufzeit berechnet wird.

Typische Anwendungnen für Reflektion sind insbesondere die Programmierung von Entwicklungsumgebungen, z.B. bei der Programmierung von:

- Klassenbrowsern
- Debuggern
- Gui-Buildern

Reflektion sollte nur in Ausnahmefällen benutzt werden. Zum einen ist Reflektion langsam, zum anderen verliert man den statischen Typcheck.

5.2.1 Eigenschaften von Klassen erfragen

Eine der ersten Fähigkeiten von Reflektion ist, für Klassen nach Ihren Eigenschaften zu fragen. Die Klasse `java.lang.Class` beschreibt Klassen mit ihren Eigenschaften.

Es gibt drei Arten, wie man an ein Objekt der Klasse `Class` gelangen kann:

- In der Klasse Objekt existiert die Methode `Class getClass()`. Sie gibt für Objekte ein Klassenobjekt der Klasse, von der sie erzeugt wurden zurück.
- Für jede Klasse kann über den Ausdruck `KlssenName.class` das `Class`-Objekt erhalten werden, das diese Klasse beschreibt.
- In der Klasse `Class` ist die statische Methode `forName(String klassenName)`, die für einen als String vorliegenden Klassennamen das entsprechende `Class`-Objekt berechnet. Falls keine Klasse mit entsprechenden Namen vorhanden ist wird eine Ausnahme des Typs `ClassNotFoundException` geworfen.

`Class`-Objekte stellen nicht nur Klassen dar, sondern auch Schnittstellen und auch primitive Typen.

RefectClass.java

```java
package name.panitz.reflect;
public class RefectClass{
  public static void main(String [] args)throws Exception{
    Object o = "hallo";
    System.out.println(o.getClass());
    System.out.println(int.class);
    System.out.println(Class.forName("java.lang.Comparable"));
  }
}
```

Das Programm führt zu folgender Ausgabe:

```
sep@linux:~/fh/prog4/examples> java -classpath classes/ name.panitz.reflect.RefectClass
class java.lang.String
int
interface java.lang.Comparable
sep@linux:~/fh/prog4/examples>
```

Die drei Arten um an ein `Class`-Objekt zu kommen haben drei unterschiedliche Typen als Ergebnis. Seit Java 1.5 ist `Class` eine generische Klasse. Ihre Typvariabel wird instanziiert mit dem Typ, für den das `Class`-Objekt eine Beschreibung liefert. Der Vorteil ist, daß sich Methoden in der Klasse `Class` auf diesen Typ beziehen können. So gibt es z.B. die Methode `cast`, die ein beliebiges Objekt auf den Typ der durch das `Class`-Objekt repräsentierten Klasse zusichert.

RefectTypeVar.java

```java
package name.panitz.reflect;
public class RefectTypeVar{
  public static void main(String [] args)throws Exception{
    Object o = "hallo";
    Class<String> klasse = String.class;
    String s = klasse.newInstance();
  }
}
```

Der Typchecker kann bis zu einen gewissen gerade selbst überprüfen, ob der Klassentyp für ein Objekt korrekt gewählt wurde. Folgende Klasse führt zu einen Übersetzungsfehler:

```java
package name.panitz.reflect;
public class TypeVarError {
  public static void main(String [] args) throws Exception{
    Class<String> klasse = Comparable.class;
  }
}
```

Anders sieht es mit der Methode `getClass` aus. Hier läßt sich nicht der Laufzeittyp eines Objekts statisch feststellen. Die folgende Klasse führt zu einen Übersetzungsfehler.


```java
package name.panitz.reflect;
public class TypeVarError2 {
  public static void main(String [] args) throws Exception{
    Object o = "";
    Class<String> klasse = o.getClass();
  }
}
```

Die Methode `getClass` hat einen relativ skurilen Typ:

```java
public final Class<? extends Object> getClass()
```

Das Fragezeichen steht für einen unbekannten Typ.

UnknownType.java

```java
package name.panitz.reflect;
public class UnknownType {
  public static void main(String [] args) throws Exception{
    Object o = "";
    Class< extends Object> klasse = o.getClass();
  }
}
```

Interessanter Weise dürfen wir den nicht mit `Object` identifizieren:

```java
package name.panitz.reflect;
public class UnknownError {
  public static void main(String [] args) throws Exception{
    Object o = new Object();
    Class<Object> klasse = o.getClass();
  }
}
```

Die statische Methode `forName(String clasName)` verzichtet derzeit vollkommen darauf, eine konkrete Instanz für die Typvariable anzugeben.

Ein noch spannenderes Konstrukt finden wir für die Signatur des Rückgabetyps der Methode, die die Superklasse für eine Klasse zurückgibt:

```java
 Class<? super T>getSuperclass();
```

Wir erhalten ein `Class`-Objekt für einen unbekannten Typ, von dem wir nur wissen, daß er ein Supertyp eines bestimmten Typs ist.

Typen und Klassen

Das Reflektions-API hilft noch einmal sehr gut, den Unterschied zwischen Typen und Klassen zu verstehen. Eine Klasse ist das, was in Javaquelltext definiert wird und für das eine Klassendatei erzeugt wird. Ein Typ kann eine bestimmte generische Instanz dieser Klasse sein, oder auch eine Typvariabel, die für einen beliebigen aber festen Typ steht.

Dieses spiegelt sich im Reflektions-API wieder. Neben der Klasse `Class` existiert seit Java 1.5 auch eine Schnittstelle `Type`. Es gibt nun z.B. zwei Methoden um etwas über die Oberklasse einer Klasse zu erfragen:

- `Class<? super T> getSuperclass()`: gibt die Klasse Oberklasse zurück, wie sie in ihrer Klassendatei definiert ist.
- `Type getGenericSuperclass()` gibt über den konkreten Typ der Superklasse zurück.

TupleSuper.java
```java
package name.panitz.reflect;
import name.panitz.crempel.util.*;

public class TupleSuper {
  public static void main(String [] args){
    Tuple2<String,Integer> t
      = new Tuple2<String,Integer>("hallo",42);
    Class<? extends Tuple2<String,Integer>> klasse = t.getClass();
    System.out.println(klasse.getSuperclass());
    System.out.println(klasse.getGenericSuperclass());
  }
}
```

Das Programm hat folgende Ausgabe:

```
sep@linux:~/fh/prog4/examples> java  name.panitz.reflect.TupleSuper
class name.panitz.crempel.util.Tuple1
sun.reflect.generics.reflectiveObjects.ParameterizedTypeImpl@a62fc3
sep@linux:~/fh/prog4/examples>
```

Bisher (Java 1.5beta1) ist die Schnittstelle `Type` noch leer und man erhält noch keine weiteren Informationen über Objekte diesen Typs.

Methoden, Konstruktoren und Felder

Ebenso wie für Klassen gibt es in Java auch für Methoden, Felder und Konstruktoren Klassen, die diese Beschreiben. Dieses sind die Klassen:

- `java.lang.reflect.Field`
- `java.lang.reflect.Method`
- `java.lang.reflect.Constructor`

Für ein `Class`-Objekt lassen sich die entsprechenden Objekte über Methoden erfragen:

- `Field[] getFields()`
- `Method[] getMethods()`
- `Constructor[] getConstructors()`

Die Klasse `Constructor` ist generisch über den Typen, den der Konstruktor erzeugt. Da in Reihungen keine generischen Typen aufgenommen werden können, spiegelt sich das in der Methode `getConstructors` nicht wieder. Es gibt aber eine Methode, mit der ein bestimmten Konstruktor einer Klasse erfragt werden kann:

```java
Constructor<T> getConstructor(Class... parameterTypes)
```

Hier erhält man auch den entsprechenden Ergebnistyp des Konstruktors.

Beispiel: Klasseninformation als XML-Format

Als kleines Beispiel der Rumpf eines Programms, das für beliebige Klassennamen ein XML-Dokument erzeugt und dieses graphisch anzeigt:

ClassAsXML.java
```java
package name.panitz.crempel.tool.classInfo;

import java.lang.reflect.Method;
import name.panitz.domtest.DomTreeNode;
import javax.swing.*;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

public class ClassAsXML {
  static Document convert(Class klasse)
                  throws ParserConfigurationException{
    Document doc
      = DocumentBuilderFactory
        .newInstance()
        .newDocumentBuilder()
        .newDocument() ;

    Element classEl
     =doc.createElement(klasse.isInterface()?"interface":"class");
    doc.appendChild(classEl);

    Element nameEl =doc.createElement("name");
```

```
    nameEl.appendChild(doc.createTextNode(klasse.getName()));
    classEl.appendChild(nameEl);

    for (Method m:klasse.getMethods()){
      final Element methodEl=getMethodInformation(doc,m);
      classEl.appendChild(methodEl);
    }
    return doc;
  }

  static Element getMethodInformation(Document doc,Method m){
    final Element methodEl=doc.createElement("method");
    final Element nameEl =doc.createElement("name");
    nameEl.appendChild(doc.createTextNode(m.getName()));
    methodEl.appendChild(nameEl);
    for (Class param:m.getParameterTypes()){
     final Element paramEl=doc.createElement("parameter");
     paramEl.appendChild(doc.createTextNode(param.getName()));
     methodEl.appendChild(paramEl);
    }
    final Element resultEl=doc.createElement("result");
    resultEl.appendChild
     (doc.createTextNode(m.getReturnType().getName()));
    methodEl.appendChild(resultEl);
    return methodEl;
  }

  public static void main(String [] args) throws Exception{
    JFrame f = new JFrame(args[0]);
    f.getContentPane()
     .add(new JTree(new DomTreeNode
                      (convert(Class.forName(args[0])))));
    f.pack();
    f.setVisible(true);
  }
}
```

Das durch dieses Programm geöffnete Fenster ist in Abbildung 5.1 für die Klasse `ParameterizedTypeImpl` zu bewundern.

5.2.2 Instanziieren dynamisch berechneter Klassen

InitAnyClass.java

```
package name.panitz.reflect;
public class InitAnyClass{
  private static String defaultClassName="java.lang.String";
  private static Object o=null;

  static Object getObject()throws Exception{
```



```
    if (o==null){
      final String userClass = System.getProperty("user.class");
      final String name = userClass==null?defaultClassName:userClass;
      o=Class.forName(name).newInstance();
    }
    return o;
  }

  public static void main(String[]args)throws Exception{
    System.out.println(getObject());
    System.out.println(getObject().getClass());
  }
}
```

5.3 Klassen Laden

5.4 RMI

5.5 Persistenz

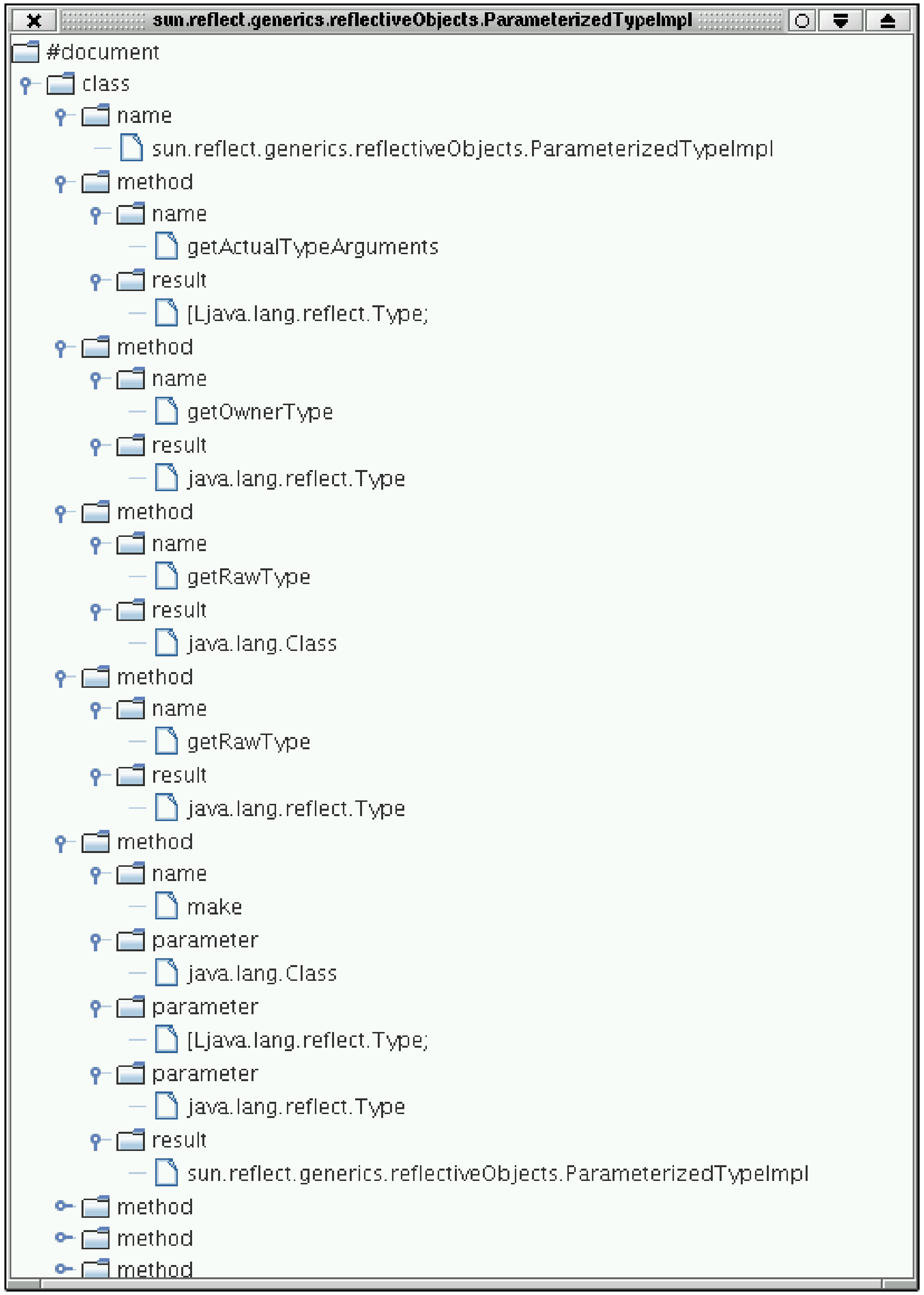

Abbildung 5.1: Informationen über die Schnittstelle ParameterizedTypeImpl als XML Dokument.

Anhang A

Crempel

Als Übungsplatform zur Vorlesung dient ein gemeinsames Projekt für alle Teilnehmer. In diesem Projekt gibt es mehrere einzelne Komponenten. Das Gesamtprojekt heit *Crempel*, wobei dieses als Abkürzung für **c***reative* **r***esearch* **e***nvironment for* **m***aking* **p***oints during the* **e***ntire* **l***ecture* stehen könnte.

A.1 Eingesetzte Werkzeuge

A.1.1 Java 1.5

Die primäre Programmiersprache für Crempel ist Java in der Version 1.5. Derzeit liegt der Javaübersetzer für Java Version 1.5 nur als beta Version vor. Dieser Übersetzer ist im SWE-Labor als Standardübersetzer installiert. In meiner Vorbereitung auf die Vorlesung sind mir nur zwei Bugs in dieser Version aufgfallen:

- bei der Ausgabe von Fehlermeldungen kommt es ab und an zu internen Fehlern des Übersetzers.
- Eine for Schleife, über eine generische Sammlung von `Integer`, deren Elemente als `int` iteriert, aber im Rum pf nicht benutzt werden, führt zu fehlerhaften byte code: `for (int i:someIntegerCollection){}`.

Um Java 1.5 Quelltext mit dem Javaübersetzer zu übersetzen ist es notwedig eine bestimmte Option zu setzen. Wird dieses nicht getan, so kann nur Quelltext nach der Java 1.3 Spezifikation übersetzt werden. Die entsprechende Option ist: `-source 1.5`.

Leider ist bisher es noch nicht möglich in *eclipse* diese Option zu setzen.

A.1.2 CVS

Das gemeinsame Repository für Crempel wird über CVS geführt. CVS (`http://www.cvshome.org/`) ist ein Kommandozeilen basiertes Programm zur Revisionskontrolle. Auf dem Rechner `pcx22` an der TFH Berlin ist hierzu das Crempel Repository aufgesetzt worden. Mit Einkleben in die Vorlesung wird jedem Teilnehmer ein Account mit Schreibrechten auf dieses Repository eingerichtet. Der Benutzername wird dabei sein: `cvs_nachname`.

Um mit CVS arbeiten können muß CVS auf dem Client-Rechner installiert sein. Ist dieses der Fall, so ist zunächst die Umgebungsvariabel `CVSROOT` auf das entsprechende Repository zu setzen. In Unix/Linux in der *bash* z.B. mit:

```
export CVSROOT=:pserver:benutzername@pcx22.tfh-berlin.de:/home/cvsrepository
```

Nun ist als nächstes sich mit dem vergebenen Passwort auf dem Repository anzumelden:

```
cvs login
```

Jetzt können die aktuellen Quelltexte geholt werden:

```
cvs checkout panitz
```

Nun befindet sich auf der Festplatte ein Ordner `panitz` mit einem Unterordner `crempel`. Hier befinden sich die Quelltexte und eine kleine `README`-Datei.

```
cd panitz/crempel
less README.txt
```

Jetzt kann mit dem Projekt gearbeitet werden. Dateien können geändert werden und neue Dateien angelegt werden. Drei weitere wichtige CVS Kommandos werden für das tägliche Arbeiten benötigt (oben haben wir bereits die Kommandos `login` und `checkout` kennengelernt):

- `update`: zum Holen der aktuellen neuen Quellen. Wenn also jemand anderes in der Zwischenzeit etwas neues eingecheckt hat, so bekommt man diese Änderungen auf seine lokale Kopie des Repositories.
- `add`: zum Hinzufügen weiterer Dateien in Repository, wobei der Befehl `add` noch nicht dafür sorgt, daß die neue Datei auf den Server geschrieben wird, sondern nur vermerkt wird, daß diese zum Repository beim nächsten `commit` hinzugefügt werden soll.
- `commit`: Änderungen, die in der lokalen Kopie gemacht wurden, werden auf den Server geschrieben. Nun werden auch die mit `add` hinzugefügten Dateien auf den Server geschrieben. Es ist zu empfehlen vor einem `commit` immer erst ein `update` durchzuführen, um sicher zu stellen, daß es zu keinen Konflikten mit zwischenzeitlich von anderen Programmierern gemachten Änderungen kommt. Beim einchecken von Änderungen verlangt CVS einen Kommentar. Standardmäßig wird der Editor `vi` geöffnet, um diesen Kommentar einzugeben. Diese Einstellung kann auf einen anderen Editor umgesetzt werden.

CVS ist ein Kommandozeilen Programm. Es gibt eine Reihe von Programmen, die als graphisches Frontend für CVS dienen wie *LinCVS* und *Cervisia* für Linux und *WinCVS* für Windows. Diese Programme machen die Arbeit mit CVS eventuell etwas übersichtlicher, allerdings sollte man die CVS Befehle kennen.

A.1.3 ANT

Crempel bedient sich der Umgebung *Ant* zum Übersetzen und Bauen. Ant (`http://ant.apache.org/`) ist ein speziell für Java in Java geschriebenes Build-Tool. Ant wird über XML-Dateien gesteuert. In einer Datei `build.xml` werden Ziele definiert, die untereinander abhängig sein können. In Crempel gibt es eine Datei `build-lib.xml` in der Ziele definiert sind, die in den einzelnen `build.xml` Dateien der Unterkomponenten von Crempel eingebunden werden.

Bevor mit *ant* Crempel gebaut und auch gestartet werden kann, ist zunächst eine Zeile in der Datei `build-lib.xml` zu ändern, so daß dort auf die lokale Installation des von `javacc` verwiesen wird. Der entsprechende Ordner läßt sich einfach über `which javacc` lokalisieren.

```
vi build-lib.xml
```

Für die Subkomponente `jugs` ist notwendig, daß die Datei `tools.jar` der Javainstallation im Klassenpfad steht. Diese ist also dem Klassenpfad hinzuzufügen. Auch diese läßt sich am einfachsten über den Befehl `which javac` lokalisieren. Der Klassenpfad kann erweitert werden mit einem entsprechenden Shell-Befehl:

```
export CLASSPATH=/usr/java/j2sdk1.5/lib/tools.jar:$CLASSPATH
```

Nach diesen Vorbereitungen kann Crempel gebaut und auch gestartet werden mit:

```
ant run
```

Damit wird *Ant* aufgefordert, das Ziel `run` nach der Steuerdatei `build.xml` zu realisieren.

A.1.4 JavaCC

Einige Komponenten von Crempel benutzen den Parsergenerator `javacc`.JavaCC (`https://javacc.dev.java.net/`) ist ein Parsergenerator in der Tradition von `yacc`, der speziell für Java konzipiert ist. Leider gibt es noch keine Version von `javacc`, die Java 1.5 versteht.

A.2 Crempel Architektur

Crempel besteht aus einem Hauptfrontend, das eine Startleiste für die Subkomponenten zur Verfügung stellt. Für jede Crempelkomponente existiert ein eigener Unterordner im Ordner Crempel. Das Hauptfrontend befindet sich im Unterordner `main`. Das Hauptfrontend besteht derzeit aus einer einzigen Klasse: `name.panitz.crempel.Crempel`. Über die XML-Steuerdatei `Properties.xml` wird spezifiziert, welche einzelnen Komponenten für Crempel geladen werden sollen.

A.2.1 Crempel Komponenten

Jede einzelne Unterkomponente von Crempel muß die Schnittstelle:
`name.panitz.crempel.tool.CrempelTool`
implementieren. Unterkomponenten können und sollten eigene Startklassen mit einer Hauptmethoden enthalten, so daß sie auch als Einzelprogramm gestartet werden können.

Jede Unterkomponente hat eine eigene `build.xml` Datei. In dieser ist vermerkt, von welchen anderen Komponenten die Komponenten abhängt.

brauser

Das Untermodul `brauser` realisiert einen minimalsten Webbrowser.

classinfo

Das Untermodul `classinfo` realisiert ein kleines Programm, um Informationen über Klassen und Schnittstellen über Reflektion zu erfragen.

filebrowser

Das Untermodul `filebrowser` realisiert die Anfänge eines Filebrowsers.

jugs

Das Untermodul `jugs` realisiert einen interaktiven Javainterpreter. Er ist in dem Bericht [Pan04b] beschrieben. Als Quelltext dieses Untermoduls dient die XML-Datei des Berichts über *Jugs*. Die eigentlichen Javaquelltexte werden aus dieser XML Quelle im *Ant*-Prozess extrahiert.

midi

Das Untermodul `midi` realisiert die Anfänge eines Programmes zur Wiedergabe von Mididateien.

ping

Das Untermodul `ping` realisiert eine minimale Version des alten Telespiels *Ping* (oder hieß es *Pong*?).

pittoresque

Das Untermodul `pittoresque` kommt derzeit als Programm für eine SlideShow der Bilddateien im aktuellen Verzeichnis daher. Durch Drücken der Leertaste wird das nächste Bild dargestellt. Die Steuertaste `F12` dreht das Bild im Uhrzeigersinn, die Tasten `F1` bis `F3` manipulieren die Farben des angezeigten Bildes.

xmlparslib

Das Untermodul `xmlparslib` enthält die Bibliothek zum Parsen nach XML DTDs, wie sie in diesem Skript vorgestellt wurden. Die Java Quelltexte werden im *Ant*-Prozess aus derm XML-Quelle extrahiert.

xpath

Das Untermodul `xpath` enthält ein kleines Werkzeug zur XPath Projektion auf XML-Dokumenten.

adt

Das Untermodul `adt` enthält das in diesem Skript vorgestellte Werkzeug zur Generierung von Klassen für algebraische Typen. Andere Module benutzen dieses Werkzeug. Der als Beispiel entwickelte kleine Interpreter für die Sprache *klip* ist als *CrempelTool* in diesem Modul mit enthalten.

corelib

Im Untermodul `corelib` befinden sich eine Reihe von Klassen, die von vielen Komponenten benötigt werden.

Derzeit wird noch keine Javadoc Dokumentation für Crempel generiert. Es wäre eine wichtige und hilfreiche Tat, wenn jemand dies in den *Ant*-Prozess integrieren würde.

Anhang B

Grammatiken

B.1 JavaCC Grammatik für einen rudimentären XML Parser

Im folgenden kommentarlos eine javacc-Grammatik, die einen Parser für XML-Dokumente definiert.

XMLParser.jj

```
options {
   STATIC=false;
}

PARSER_BEGIN(XMLParser)
package name.panitz.xml;

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

public class XMLParser  {
  public static void main(String [] args)throws Exception{
    XML doc = new XMLParser(new FileReader(args[0])).xml();

    System.out.println(doc.visit(new Show()));
    System.out.println(
     doc.visit(new SelectElement(new Name("","code",""))));
    System.out.println(doc.visit(new Content()));
    System.out.println(doc.visit(new Size()));
    System.out.println(doc.visit(new Depth()));

  }
}
PARSER_END(XMLParser)

```



```
TOKEN :
{< NCNAME : ( <Letter> | "_" ) ( <NCNAMECHAR> )* >}

TOKEN :
{ < NCNAMECHAR : ( <Letter> | <Digit> | "." | "-" | "_" | <Extender> )
| < #Letter    : ( <BaseChar> )
| < #BaseChar  : ["\u0041"-"\u005A","\u0061"-"\u007A","\u00C0"-"\u00D6",
                  "\u00D8"-"\u00F6","\u00F8"-"\u00FF"]
| < #Digit     : ["\u0030"-"\u0039"]                              >
| < #Extender  : ["\u00B7"]                                       >
}

<CharDataSect> TOKEN :
{ < CHARDATA       : ( <CharDataStart> )+         >
| < #CharDataStart : ( ~["<","&"] )                 >
}

<DEFAULT, CharDataSect> TOKEN:
{ < STAGO:   "<"                                        > : DEFAULT
| < ETAGO:   "</"                                       > : DEFAULT
| < PIO          :   "<?"                               > : DEFAULT
| < COMMENT: "<!--" (~["-"])* ("-" (~["-"])+)* "-->"        > : DEFAULT
}

TOKEN :
{<ETAGC: "/>">
|<GT: ">">
|<EQ: "=">
}


<DEFAULT,AttValueSect> TOKEN :
{  < DQUOTED: "\""        > : DEFAULT
|  < SQUOTED: "'"        > : DEFAULT
}

<AttValueSectD> TOKEN :
{ < AttValueDRest : ( ~["<","&","\""] )+ >
| < AttValueDEnd  : "\""                > : DEFAULT
}

<AttValueSectS> TOKEN :
{ < AttValueSRest : ( ~["<","&","'"] )+ >
| < AttValueSEnd  : "'"                 > : DEFAULT
}

<DEFAULT, CharDataSect> TOKEN :
{ < CDataStart : "<![CDATA[" > : CDataSect}

<CDataSect> TOKEN :
```

```
{   < CDataContent :
     ( ~["]"] )+ ( (( "]" ( ~["]"] ) ) | ( "]]" ~[">"] )) ( ~["]"] )* )*
}

<DEFAULT, CDataSect> TOKEN :
{ < CDataEnd     : "]]"">"> : DEFAULT}

<DEFAULT,CharDataSect, AttValueSectD, AttValueSectS> TOKEN :
{   < AMP: "&"         > : DEFAULT}

<RefSect> TOKEN :
{ < EOENT : ";" > : DEFAULT }


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

XML xml():
{XML xml;}
{(xml=element()
 |xml=text()
 |xml=cdata()   { token_source.SwitchTo(CharDataSect); }
 |xml=comment() { token_source.SwitchTo(CharDataSect); }
 |xml = entity(){ token_source.SwitchTo(CharDataSect); }
// |xml=pi()      { token_source.SwitchTo(CharDataSect); }
 )
 {return xml;}}

Element element():
{List attributes=new ArrayList();
 List children=new ArrayList();
 Attribute attribute;
 XML xml;
 Name name;
 Name endName;
}
{<STAGO> name=name()
 (attribute=attribute() {attributes.add(attribute);} )*
 ((<ETAGC>  {token_source.SwitchTo(CharDataSect); })
 |(<GT> {token_source.SwitchTo(CharDataSect);}
    (xml=xml() {children.add(xml);})* <ETAGO>endName=name()<GT>
   {token_source.SwitchTo(CharDataSect); })
 )
 {return new Element(name,attributes,children);}
}

```



```
Text text():
{Token tok;}
{tok=<CHARDATA>{return new Text(tok.image);}}

CDataSection cdata() : { String data = null; }
{ <CDataStart>
  [ data = CData() ]
  <CDataEnd>
  {return new CDataSection(data);}
}

String CData() : { Token tok; }
{ tok = <CDataContent> { return tok.image; } }

Comment comment():{Token comment;}
{comment=<COMMENT>
  {String com = comment.image;
   com = com.substring(4,com.length()-3);
   return new Comment(com);}
}

Entity entity():
{ String name; }
{ <AMP>
  name = ncName() { token_source.SwitchTo(RefSect); }
  <EOENT>
    { return new Entity(name); }
}

Attribute attribute():
{ Name n;
  String value;}
{ n=name() <EQ>
   { token_source.SwitchTo(AttValueSect); }
  value = AttValue()
{return new Attribute(n,value);}}

String AttValue() :
{ StringBuffer sb;
  char chr;
  Token tok;
  String str = null;
  char[] ac = null;
}
{ (
    <DQUOTED> {
      token_source.SwitchTo(AttValueSectD);
       sb = new StringBuffer();
    }
    ( tok = <AttValueDRest> { sb.append(tok.image); })*
```

```
    <AttValueDEnd> { return sb.toString(); }
  )
  |
  (
    <SQUOTED> {
      token_source.SwitchTo(AttValueSectS);
       sb = new StringBuffer();
    }
    ( tok = <AttValueSRest> { sb.append(tok.image); })*
    <AttValueSEnd> { return sb.toString(); }
  )
}

String ncName() : { Token tok; }
{tok = <NCNAME> { return tok.image; }}

Name name():
{String pre="";
 String n="";}
{ pre=ncName()
  [ ":" n=ncName(){System.out.println(n);}]
  {if (n.length()==0) return new Name("",pre,"");
   return new Name(pre,n,"");}
}
```

B.2 Javacc Grammatik für abgekürzte XPath Ausdrücke

Anhang C

Gesammelte Aufgaben

Aufgabe 1 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

Aufgabe 2 Die folgenden Dokumente sind kein wohlgeformetes XML. Begründen Sie, wo der Fehler liegt, und wie dieser Fehler behoben werden kann.

a)

```
<a>to be </a><b> or not to be</b>
```

Lösung
Kein top-level Element, das das ganze Dokument umschließt.

b)

```
<person geburtsjahr=1767>Ferdinand Carulli</person>
```

Lösung
Attributwerte müssen in Anführungszeichen stehen.

c)

```
<line>lebt wohl<br/>
<b>Gott weiß, wann wir uns wiedersehen</line>
```

Lösung
`<b>` wird nicht geschlossen.

d)

```
<kockrezept><!--habe ich aus dem Netz
  <name>Saltimbocca</name>
  <zubereitung>Zutaten aufeinanderlegen
               und braten.</zubereitung>
</kockrezept>
```

Lösung
Kommentar wird nicht geschlossen.

e)
```
<cd>
  <artist>dead&alive</artist>
  <title>you spin me round</title></cd>
```

Lösung
Das Zeichen & muß als character entity `&` geschrieben werden.

Aufgabe 3 Gegeben sind das folgende XML-Dokument:

TheaterAutoren.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<autoren>
<autor>
  <person>
    <nachname>Shakespeare</nachname>
    <vorname>William</vorname>
  </person>
  <werke>
    <opus>Hamlet</opus>
    <opus>Macbeth</opus>
    <opus>King Lear</opus>
  </werke>
</autor>
<autor>
  <person>
    <nachname>Kane</nachname>
    <vorname>Sarah</vorname>
  </person>
  <werke>
    <opus>Gesäubert</opus>
    <opus>Psychose 4.48</opus>
    <opus>Gier</opus>
  </werke>
</autor>
</autoren>
```

a) Schreiben Sie eine DTD, das die Struktur dieses Dokuments beschreibt.
Lösung

AutorenType.dtd
```
<!DOCTYPE autoren SYSTEM "AutorenType.dtd" [
<!ELEMENT autoren (autor+)>
<!ELEMENT autor (person,werke)>
<!ELEMENT werke (opus*)>
<!ELEMENT opus (#PCDATA)>
<!ELEMENT person (nachname,vorname)>
<!ELEMENT nachname (#PCDATA)>
<!ELEMENT vorname (#PCDATA)>]>
```


b) Entwerfen Sie Java Schnittstellen, die Objekte ihrer DTD beschreiben.
Lösung

Autoren.java
```
package name.panitz.xml.exercise;
import java.util.List;
public interface Autoren {
  List<Autor> getAutorList();
}
```

Autor.java
```
package name.panitz.xml.exercise;
public interface Autor {
  Person getPerson();
  Werke  getWerke();
}
```

Person.java
```
package name.panitz.xml.exercise;
public interface Person {
  Nachname getNachname();
  Vorname  getVorname();
}
```

Werke.java
```
package name.panitz.xml.exercise;
import java.util.List;
public interface Werke {
  List<Opus> getOpusList();
}
```

HasJustTextChild.java
```
package name.panitz.xml.exercise;
public interface HasJustTextChild {
  String getText();
}
```

Opus.java
```
package name.panitz.xml.exercise;
public interface Opus extends HasJustTextChild {}
```

Nachname.java
```
package name.panitz.xml.exercise;
public interface Nachname extends HasJustTextChild {}
```

Vorname.java
```
package name.panitz.xml.exercise;
public interface Vorname extends HasJustTextChild {}
```

c) Schreiben Sie ein XSLT-Skript, das obige Dokument in eine Html Liste der Werke ohne Autorangabe transformiert.
Lösung

```
WerkeListe.xsl
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
     <html><head><title>Werke</title></head>
       <body><xsl:apply-templates /></body>
     </html>
  </xsl:template>

  <xsl:template match="autoren">
     <ul><xsl:apply-templates select="autor/werke/opus"/></ul>
  </xsl:template>

  <xsl:template match="opus">
     <li><xsl:apply-templates/></li>
  </xsl:template>
</xsl:stylesheet>
```

d) Schreiben Sie eine Javamethode
`List<String> getWerkListe(Autoren autoren);`,
die auf den Objekten Ihrer Schnittstelle für `Autoren` eine Liste von Werknamen erzeugt.

Lösung

```
GetWerke.java
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;

public class GetWerke{
  public List<String> getWerkListe(Autoren autoren){
    List<String> result = new ArrayList<String>();
    for (Autor a:autoren.getAutorList()){
      for (Opus opus: a.getWerke().getOpusList())
        result.add(opus.getText());
    }
    return result;
  }
}
```

Aufgabe 4 Gegeben sei folgendes XML-Dokument:

```
Foo.xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<x1><x2><x5>5</x5><x6>6</x6></x2><x3><x7>7</x7></x3>
<x4><x8/><x9><x10><x11></x11></x10></x9></x4></x1>
```

a) Zeichnen Sie dieses Dokument als Baum.

b) Welche Dokumente selektieren, die folgenden XPath-Ausdrücke ausgehend von der Wurzel dieses Dokuments

- `//x5/x6`
- `/descendant-or-self::x5/..`
- //x5/ancestor::*
- /x1/x2/following-sibling::*
- /descendant-or-self::text()/parent::node()

Aufgabe 5 Schreiben Sie eine Methode
`List<Node> getLeaves(Node n);`,
die für einen DOM Knoten, die Liste aller seiner Blätter zurückgibt.

Lösung

```
GetLeaves.java
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;

public class GetLeaves{
  public List<Node> getLeaves(Node n){
    List<Node> result = new ArrayList<Node>();
    NodeList ns = n.getChildNodes();
    if (ns.getLength()==0){
      result.add(n); return result;
    }
    for (int i=0;i<ns.getLength();i++){
      result.addAll(getLeaves(ns.item(i)));
    }
    return result;
  }
}
```

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Arn04] Arnaud Le Hors and Philippe Le Hégaret and Lauren Wood and Gavin Nicol and Jonathan Robie and Mike Champion and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification Version 1.0. W3C Recommendation, April 2004. `http://www.w3.org/TR/P3P/`.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) 1.0. W3C Recommendation, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116.xml`.

[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.

[HM96] Graham Hutton and Eric Meijer. Monadic parser combinators. submitted for publication, `www.cs.nott.ac.uk/Department/Staff/gmh/bib.html`, 1996.

[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.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

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

[Pan00] Sven Eric Panitz. Generische Typen in Bolero. *Javamagazin*, 4 2000.

[Pan03a] Sven Eric Panitz. Programmieren I. Skript zur Vorlesung, 2. revidierte Auflage, 2003. `www.panitz.name/prog1/index.html`.

[Pan03b] Sven Eric Panitz. Programmieren II. Skript zur Vorlesung, TFH Berlin, 2003. `www.panitz.name/prog2/index.html`.

[Pan03c] Sven Eric Panitz. Programmieren III. Skript zur Vorlesung, TFH Berlin, 2003. `www.panitz.name/prog3/index.html`.

[Pan04a] Sven Eric Panitz. Erweiterungen in Java 1.5. Skript zur Vorlesung, TFH Berlin, 2004. `www.panitz.name/java1.5/index.html`.

[Pan04b] Sven Eric Panitz. The Making of Jugs. Skript zur Vorlesung, TFH Berlin, 2004. `www.panitz.name/jugs/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.

[T. 04] T. Bray, and al. XML 1.1. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml11`.

[Wad85] Phil Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

[Wad00] Philip Wadler. A formal semantics of patterns in XSLT, March 2000.