Parsing XML Document Type Structures

An Exercise in Java 1.5
(Work in Progress)

Sven Eric Panitz
TFH Berlin

Version 5th March 2004

The Java 1.5 release will introduce generic types to Java which have been known in functional programming for quite a while.

This paper presents a Java 1.5 library for parsing xml document structures. The library is written in the functional style of parser combinators. It makes extensive use of generic types and function objects.

The source of this paper is an XML-file. The sources are processed by the XQuery processor *quip*, XSLT scripts and LaTeX in order to produce the different formats of the paper.

Contents

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 be easily hand coded by way of parser cominators[HM96]. We will show, how to apply the concept of parser generators to XML document type decriptions in Java.

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*.

1.1.1 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 parser to more complex parser:

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

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

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 parser, 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.

Example:
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 coud be complete 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].

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)*[**?**].

1.2.1 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 `,`.
- alternatives of elements donoted by the vertical bar `|`.
- several kinds of repetition denoted by `+`, `*` and `?`.

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

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.

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.

1.3.1 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.

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.

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 abovem 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 [Pan04] 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);
```

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 [Pan04] 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.

2.2.1 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);
  }
}
```


2.2.2 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);
  }
}
```

2.2.3 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 List<Node> ys = copy(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);
  }

/*  static <a> List<a> copy(List<a> xs){
    List<a> result=new ArrayList<a>();
    for(a x:xs) result.add(x);
    return result;
  }*/
}
```

2.2.4 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);}
}
```

2.2.5 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 [Pan04].

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);
  }
}
```

2.2.6 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.

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.

2.3.1 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()));
    return new Fail<String>(xs);
  }
}
```

2.3.2 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);
  }
}
```

2.3.3 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;

public class Element<a> extends AbstractParser<a> {
  final Parser<a> p;
  final String elementName;

  public Element(String n,Parser<a> _p){elementName=n;p=_p;}
```

For simplicity reasons, we will neglect text nodes from list of nodes, which contain 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();
    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 = p.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;
  }
}
```

2.4 Building parsers

We have got everything now to implement parsers for a DTD in Java.

2.4.1 Defining a Parser

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>{
  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;
         }
      };

  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();
         }
      };

  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();
         }
      };

  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());

  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);
  }
}
```

2.4.2 Starting the parser

MainParser.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;}
 }
}
```

```
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>
```

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.

3.1 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.

3.1.1 Simple visitors vor DTDDef

As a first exercise we write two simple visitor classes for the type `DTDDef`.

Show

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

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('.','_');
  }
}
```

3.2 Generation of tree classes

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();}
  }

}
```

3.3 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+"\",");
  }

  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(")");
    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(")");
  }

  public static String writeParser
                    (DTDDef def,String n,boolean isGenerated){
   return def.visit(new WriteParser(n,isGenerated));}
}
```

3.4 Flattening of a DTD definition

FlattenResult.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

```
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;
  }

  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){
               return new DTDChoice(dtds);}
           });
  }

  private FlattenResult  single(DTDDef x){return new FlattenResult(x);}

  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)){
      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);
    }
    FlattenResult innerRes  = content.visit(this);
    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)) newParts.add(part);
      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)
           ||
     (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());}
}
```

3.5 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);

    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 Beispiel: Javaklassengenerierung für eine DTD

5 Example

5.1 Ausgangs-DTD

play.dtd

```
<!DOCTYPE PLAY SYSTEM "play.dtd" [
<!ELEMENT PLAY
   (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT, INDUCT?
   ,PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
```



```
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT INDUCT
   (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT
   (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
<!ELEMENT SCENE
   (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD  (#PCDATA)>]>
```

5.2 Konvertierung nach LaTeX

PLAYToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public  class PLAYToLaTeXVisitor
                 extends PLAYVisitor<StringWriter>{
  StringWriter out=new StringWriter();

  public StringWriter eval(PLAY pl){
    final TITLE title=pl.getTheTITLE();
    final FM fm=pl.getTheFM();
    final PERSONAE personae=pl.getThePERSONAE();
    final SCNDESCR scndesc =pl.getTheSCNDESCR();
    final PLAYSUBT playsubt=pl.getThePLAYSUBT();
    final Maybe<INDUCT> induct = pl.getTheMaybe_INDUCT_();
    final Maybe<PROLOGUE> prologue= pl.getTheMaybe_PROLOGUE_();
    final List<ACT> acts=pl.getTheList_ACT_();
    final Maybe<EPILOGUE> epilogue = pl.getTheMaybe_EPILOGUE_();

    out.write("\\documentclass[10pt,a4paper]{article}\n");
    out.write("\\usepackage[english]{babel}\n");
    out.write("\\begin{document}\n");
    out.write("\\title{"+title.getPcdata()+"}\n");
    for (ACT act:acts){
      act.visit(new ACTToLaTeXVisitor(out));
    }
```

```
    out.write("\\end{document}\n");

    return out;
  }

  public static void main(String [] args) throws Exception{
    final String arg= args[0];
    PLAY play
     = (PLAY)de.tfhberlin.crempel.test.MainParser.pFile
          ("examples.theatre.PLAYParser",arg);
    StringWriter tex = play.visit(new PLAYToLaTeXVisitor());
    Writer out
      = new FileWriter(
                 arg.substring(0,arg.lastIndexOf('.'))+".tex");
    out.write(tex.toString());
    out.close();
  }
}
```

ACTToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public class ACTToLaTeXVisitor extends ACTVisitor<StringWriter>{
  StringWriter out;
  public ACTToLaTeXVisitor(StringWriter o){out=o;}
  public StringWriter eval(ACT act){
    final TITLE title = act.getTheTITLE();;
    final List<SUBTITLE> subtitles = act.getTheList_SUBTITLE_();
    final Maybe<PROLOGUE> prologue=act.getTheMaybe_PROLOGUE_();
    final List<SCENE> scenes=act.getTheList_SCENE_();
    final Maybe<EPILOGUE> epilogues=act.getTheMaybe_EPILOGUE_();

    out.write("\\section*{"+title.getPcdata()+"}\n");
    for (SCENE scene:scenes){
      scene.visit(new SceneToLaTeXVisitor(out));
    }
    return out;
  }}
```

SceneToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

```



```
import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public  class SceneToLaTeXVisitor extends SCENEVisitor<StringWriter>{
  StringWriter out;
  public SceneToLaTeXVisitor(StringWriter o){out=o;}

  public StringWriter eval(SCENE sc){
    final TITLE title=sc.getTheTITLE();
    final List<SUBTITLE> subtitles=sc.getTheList_SUBTITLE_();
    final List<Choice_SPEECH_STAGEDIR_SUBHEAD> spDirSubheads
     =sc.getTheList_Choice_SPEECH_STAGEDIR_SUBHEAD_();
    out.write("\\subsection*{"+title.getPcdata()+"}\n");
    for (Choice_SPEECH_STAGEDIR_SUBHEAD spDirSubhead:spDirSubheads)
      spDirSubhead.visit(new SpeechStageDirSubheadToLaTeX(out));
    return out;
  }}
```

SpeechStageDirSubheadToLaTeX.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;
import java.io.*;
import java.util.List;
import de.tfhberlin.crempel.util.Maybe;

public  class SpeechStageDirSubheadToLaTeX
 extends Choice_SPEECH_STAGEDIR_SUBHEADVisitor<StringWriter>{

  StringWriter out;
  public SpeechStageDirSubheadToLaTeX(StringWriter o){out=o;}

  public StringWriter eval
                  (CChoice_SPEECH_STAGEDIR_SUBHEADSPEECH sp){
    SPEECH speech = sp.getTheSPEECH();
    out.write("\\begin{description}\n");
    out.write("\\item[");
    for (SPEAKER speaker:speech.getTheList_SPEAKER_()){
      out.write(" "+speaker.getPcdata());
    }
    out.write("]");
    List<Choice_LINE_STAGEDIR_SUBHEAD> lineStagedirSubheads
      = speech.getTheList_Choice_LINE_STAGEDIR_SUBHEAD_();

    for (Choice_LINE_STAGEDIR_SUBHEAD lineStagedirSubhead
        :lineStagedirSubheads){
      lineStagedirSubhead.visit(new LineStageDirSubheadToLaTeX(out));
    }

```

```

    out.write("\\end{description}\n");
    return out;
  }
  public StringWriter eval
               (CChoice_SPEECH_STAGEDIR_SUBHEADSTAGEDIR st){
    STAGEDIR stageDir = st.getTheSTAGEDIR(); ;
    return out;
  }
  public StringWriter eval
               (CChoice_SPEECH_STAGEDIR_SUBHEADSUBHEAD subH){
    SUBHEAD subhead = subH.getTheSUBHEAD(); ;
    return out;
  }
  public StringWriter eval(Choice_SPEECH_STAGEDIR_SUBHEAD xs){
    throw new RuntimeException(xs.getClass().toString());
  }}

```

LineStageDirSubheadToLaTeX.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;
import java.io.*;
import java.util.List;
import de.tfhberlin.crempel.util.Maybe;

public  class LineStageDirSubheadToLaTeX
 extends Choice_LINE_STAGEDIR_SUBHEADVisitor<StringWriter>{

  StringWriter out;
  public LineStageDirSubheadToLaTeX(StringWriter o){out=o;}

  public StringWriter eval(CChoice_LINE_STAGEDIR_SUBHEADLINE sp){
    LINE line = sp.getTheLINE();
    for (Choice_String_STAGEDIR stringStageDir:line.getXs()){
      if (stringStageDir instanceof CChoice_String_STAGEDIRString){
        out.write
         (((CChoice_String_STAGEDIRString) stringStageDir)
                                  .getTheString());
        out.write("\\\\");
      }
    }
    return out;
  }
  public StringWriter eval
           (CChoice_LINE_STAGEDIR_SUBHEADSTAGEDIR st){
    STAGEDIR stageDir = st.getTheSTAGEDIR(); ;
    return out;
  }
```



```
  public StringWriter eval
             (CChoice_LINE_STAGEDIR_SUBHEADSUBHEAD subH){
    SUBHEAD subhead = subH.getTheSUBHEAD(); ;
    return out;
  }
}
```

6 Conclusion

A Javacc input file for DTD parser

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;
  }
}
```

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{
      List<String> fileNames = new ArrayList<String>();

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

      for (String arg:fileNames){
        File f = new File(arg);

        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();

        GenerateClassesForDTD
```



```
          .generateAll("examples.theatre",path,dtd.e3,dtd.e1);
      }
    }catch (Exception _){_.printStackTrace();System.out.println(_);}
  }
}
```

References

[HM96] Graham Hutton and Eric Meijer. Monadic parser combinators. submitted for publication,
`www.cs.nott.ac.uk/Department/Staff/gmh/bib.html`, 1996.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

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

[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.