

Projekt Dokumententransformation I  
WUJA: A JAVA extension for tree transformations

Martin Grabmüller <mgrabmue@cs.tu-berlin.de>  
Raimund Jacob <raimi@cs.tu-berlin.de>  
André Przywara <macleod@cs.tu-berlin.de>  
Thorsten Schütt <schuett@cs.tu-berlin.de>  
Andre Seidelt <iluvatar@cs.tu-berlin.de>

October 27 2000

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Usage details . . . . .	3
1.3	Requirements . . . . .	4
<b>2</b>	<b>Installation and Usage</b>	<b>5</b>
2.1	Audience . . . . .	5
2.2	Requirements for the tree . . . . .	5
2.3	Contents of a <code>.jslt</code> file . . . . .	5
2.3.1	The rule construct . . . . .	5
2.3.2	The pattern . . . . .	6
2.3.3	The conditions . . . . .	7
2.3.4	The body . . . . .	7
2.4	Application of rules . . . . .	7
<b>3</b>	<b>Compiler</b>	<b>9</b>
3.1	Structure . . . . .	9
3.2	Parser . . . . .	9
3.3	Code construction . . . . .	9
3.4	The translation of WUJA . . . . .	10
3.4.1	Conditions . . . . .	10
3.4.2	Variables . . . . .	11
<b>4</b>	<b>Matcher</b>	<b>14</b>
4.1	The Pattern Language . . . . .	14
4.1.1	Pattern Structure . . . . .	14
4.1.2	Overview . . . . .	15
4.1.3	Examples . . . . .	16
4.2	Implementation . . . . .	17
4.2.1	General – The Matcher . . . . .	17
4.2.2	Matching Tree Nodes – <code>NodeMatcher</code> . . . . .	17
4.2.3	Matching Class Names – <code>ClassNodeMatcher</code> . . . . .	17

4.2.4	Matching Arbitrary Strings – <code>StringNodeMatcher</code> . . . . .	18
4.2.5	Binding Variables – <code>VarNodeMatcher</code> . . . . .	18
4.2.6	Matching Paths – <code>PathMatcher</code> . . . . .	18
4.2.7	Descending The Tree – <code>DeepPathMatcher</code> . . . . .	18
4.2.8	Treewalker – <code>DeepMatcher</code> . . . . .	18
4.2.9	Childlist . . . . .	18
4.2.10	Storing Variable Bindings – <code>VarSet</code> . . . . .	19
4.3	Some thoughts about missing features . . . . .	20
4.3.1	Sequenced variables . . . . .	20
<b>5</b>	<b>Trafo</b>	<b>21</b>
<b>6</b>	<b>The Stream Feature</b>	<b>22</b>
6.1	Implementation . . . . .	22
6.2	Example . . . . .	23
<b>7</b>	<b>Summary</b>	<b>27</b>
7.1	Future work . . . . .	27
7.2	Conclusion . . . . .	28
<b>A</b>	<b>Grammars</b>	<b>29</b>
A.1	WUJA . . . . .	29
A.2	Pattern . . . . .	30
<b>B</b>	<b>Hints and common errors</b>	<b>31</b>
B.1	Common Errors . . . . .	31
B.2	Hints . . . . .	32
<b>C</b>	<b>Resources</b>	<b>33</b>

# Chapter 1

## Introduction

### 1.1 Overview

WUJA is an extension to the JAVA programming language. Like XSL you can use WUJA to transform XML documents. But with WUJA you can also transform any tree like data structure which implements the `wuja.tree.Tree` interface. This interface is kept simple so it can be easily adapted to other tree structures. Possible uses are:

- abstract syntax trees
- DOM trees
- TypedDOM trees

WUJA combines the idea of XSL stylesheet transformation with the power of a complete programming language. It is possible to specify patterns as it is with XSL but the transformation itself can be done using arbitrary complicated JAVA code. Since the host language is JAVA any JAVA programmer can easily learn to use WUJA. Because WUJA is designed to help with tree transformation one can easily use it for tree matching alone. It is easily possible to extract nodes from a tree that satisfy a certain path pattern and arbitrary conditions implemented in JAVA code.

### 1.2 Usage details

WUJA is implemented as a JAVA preprocessor. The user implements a JAVA class in a file having the `.jslt` extension. An invocation of `java wuja.driver.Main someclass.jslt` will transform `someclass.jslt` to `someclass.java` which can then be compiled with any JAVA 1.2 compiler. There is a wrapper shell script called `wujac` to simplify invocation.

All the compilation steps are supposed to be run from the `Makefile` that is used for

compiling the user's application.

All `rule` statements in the given `.jslt` file are converted into a constructor of `wuja.compiler.Trafo` which does the actual work. See the grammar (section A.1, page 29) and `Trafo` (chapter 5, page 21) sections to learn about details. Applications using WUJA need the `wuja.jar` file available in their classpath. It contains the matching and transformation runtime classes.

## 1.3 Requirements

WUJA needs several packages and external classes (have a look at the resource list on page 33).

- `gnu.getopt` for command line parsing
- ANTLR Version 2.7.0 is used to generate the parser for `.jslt` files
- If Jtidy/DOM is used the `org.w3c.dom` interface package is needed

WUJA is known to run in all JDK 1.2 environments.

# Chapter 2

## Installation and Usage

### 2.1 Audience

In this chapter it is assumed that you are familiar with JAVA concepts like anonymous inner classes. Refer to the JAVA language specification for further information.

### 2.2 Requirements for the tree

To work on a given tree WUJA requires it to implement the `Tree` interface as defined in the package `wuja.tree`. For detailed description of the methods have a look at the API documentation of `Tree` and `ChildnodeList` in that package.

### 2.3 Contents of a `.jslt` file

A `.jslt` is basically the same as an ordinary JAVA file. In addition to JAVA it may contain one or more *rules*.

#### 2.3.1 The rule construct

A rule declaration is allowed wherever a `new` expression may be used. The WUJA compiler takes those rules and compiles them to a `Trafo` object with an anonymous subclass of `VarSet` as the argument. A rule declaration looks like this (a detailed description of every part is given below):

```

class foo {
    // some class stuff
    int example_int = 23;

    public String getAnswer() {
        return("42");
    }

    public Trafo example_rule = rule {
        // declare pattern
        pattern {
            @"foo" \ x = @"bar"
        }

        // declare zero or more conditions
        condition(x) {
            return(((bar) x).canBar());
        }

        // declare a body that does something
        body {
            System.out.println("Found " + x.toString());

            return(null);
        }
    };
}

```

In the above example `example_rule` is the name of this rule and can be used to access the created rule. A rule consists of three parts:

- a pattern declaration with zero or more variable bindings (and optional casts). For the complete description of the pattern language see 4.1 (page 14).
- zero or more conditions.
- a rule body that is executed if the pattern matches and all conditions return `true`.

### 2.3.2 The pattern

In this part the pattern for this rule is declared as described in 4.1 (page 14). Every variable declared in the pattern is accessible as a `Tree` object in the conditions and the body of this rule. It is possible to insert casts here. E.g.:

```
@"foo" \ x = (MyTree) @"bar"
```

will insert a cast that makes it possible to access `x` as an instance of `MyTree`. Casting is done the same way as it is done in `JAVA`.

### 2.3.3 The conditions

You may define additional conditions to control the executions of a rule's body. In the above example the body is only executed if the method `canBar()` of `x` returns `true`. If more than one condition is given all of them are executed until one of them returns `false` or no more conditions exist (the results of the conditions are AND'ed). Conditions are executed in the order they are declared. A condition is equivalent to a `JAVA` method with the following signature:

```
boolean condition_X(<list of declared variables with their types>);
```

If no condition is given the body is always executed.

Unlike the body a condition has to name all variables it uses in a comma separated list between the parenthesis following the keyword `condition`. Keep in mind that conditions with a single variable are executed every time an assignment is done. Conditions with more than one variable are evaluated just before the body would be executed.

### 2.3.4 The body

The body is called every time a pattern matched the tree and all conditions returned `true`. A body may do everything like a normal `JAVA` method and has to return an object implementing `Tree` (or `null` if you do not need a return value). Please remember that `apply()` also returns `null` if no match was found. Therefore it should return something different if you want to use `apply()` instead of `applyAll()` (e.g. `NullTree` from `wuja.tree`). All variables declared in the pattern are accessible here (with `Tree` as the default type if no cast is given).

## 2.4 Application of rules

To apply rules the `apply()` or `applyAll()` method of the wanted rule must be called. For a detailed description of these methods have a look at the API documentation of the `Trafo` object in `wuja.compiler`.

Example:

```
class foo {
  // rule of previous example
  [...]

  public final static void main(String argv) {
    // do something magical to get a Tree
    Tree tree = (Tree) myGetTree();

    /* create instance of object containing the rule
    ** and apply it to the tree */
    foo f = new foo();
    Trafo trafo = f.example_rule;
    Tree result[] = trafo.applyAll(t);
  }
}
```

For complex trees (and complex tasks) it is common behaviour to have several simple rules that call other rules with a subtree of their own input (see `examples/xml2tex.jslt` for an example of complex patterns).

# Chapter 3

## Compiler

### 3.1 Structure

The compiler has a parsing and a code constructing phase. It does no type-checking so it is important for the user to locate errors easily. One approach would be to use a compiler which does not change the line numbers between input and output. In the timeframe of this project it was not possible to build such a compiler. So we tried another approach. The compiler pretty prints its output and all classes which are used will be fully qualified to avoid collisions of namespaces.

### 3.2 Parser

The parser is based on a JAVA grammar which ships with ANTLR. For the grammar see section A.1, page 29.

### 3.3 Code construction

JAVA is the target language for the compiler. This implies that WUJA has no real benefits. A language extension which can be expressed by the target language can only simplify the construction of certain statements. The following pattern is translated into 550 characters.

```
y = foo \\ x = bla[?*, bla, bla]
```

For a user it is nearly impossible to use complex patterns by hand or debug programs which contain complex patterns.

## 3.4 The translation of WUJA

In the following example we will explain the translation step by step.

```
...
public Trafo showJPGs = rule {
    pattern {
        x = (Node)@"html" \ y = (Element)@"img"
    }

    condition(x){
        return x == x;
    }

    condition(y){
        return y == y;
    }

    condition(x, y){
        return y.getAttribute("src").toLowerCase().endsWith(".jpg");
    }

    body {
        System.out.println(y.getAttribute("src"));
        return null;
    }
};
...
```

The rule expression is translated into a new instance of `wuja.compiler.Trafo`. `Trafo` will get two parameters.

- a pattern object.
- an anonymous subclass of `VarSet`.

The translation of the pattern is straight forward. For most of the rules in the pattern grammar exists an equivalent constructor in `wuja.matcher`. The main task of the compiler in this part is to reduce the work of writing patterns and prevent the user from writing illegal patterns.

### 3.4.1 Conditions

Each condition is translated into a predicate. It is important to understand that the conditions have to return a `boolean` value. If the pattern contains type casts the pa-

parameters are provided with the given types. If the cast is illegal a `ClassCastException` will occur at runtime.

Casts are inserted in the pattern like this:

```
x = (Library) Library \\ y = (Book) Book
```

In this example one can be sure to get no `ClassCastException`, because in this case it will be matched against classnames.

### 3.4.2 Variables

`apply()` fulfills three tasks:

1. to declare, cast and assign variables
2. to check conditions with more than one parameter
3. to execute the body of the rule

The variables are saved in the provided `VarSet`. When the body is executed `apply()` will copy the variables and type cast them. Now conditions with more than one parameter are checked. If there were no conditions or all of them returned `true` the user-defined body is executed. The code of the body must return an object that implements `Tree`.

The method `testVariable()` will be executed by the matcher each time it wants to store a variable in the `VarSet`. Depending on the index of the variable the corresponding conditions will be checked.

To write efficient rules it is important to understand at which point each condition is executed. Conditions with one parameter should be used to limit the search-tree. They are executed while traversing the tree. Time-consuming conditions which will not prune the tree significantly should be modified to have at least two parameters.

```
pattern{
  x = (Library) Library \\ y = (Book) Book
}
condition(y, x){
  return
  predicateWhichWillProbablyFailNeverButMustCalculateAllDigitsOfPi(y);
}
```

In this case one should add the unused variable `x` to the condition.

In following you will see the result of `wujac` for the code above on page 10.

```

...
public Trafo test2variables = new wuja.compiler.Trafo(
// pattern
    new wuja.matcher.DeepPathMatcher(
        new wuja.matcher.VarNodeMatcher(
            0,
            new wuja.matcher.StringNodeMatcher("html")
        ),
        new wuja.matcher.VarNodeMatcher(
            1,
            new wuja.matcher.StringNodeMatcher("img")
        )
    ),
    new wuja.matcher.VarSet(2){

// condition(x)
    private boolean condition_0(Node x){
        return x==x;
    }

// condition(y)
    private boolean condition_1(Element y){
        return y==y;
    }

// condition(x, y)
    private boolean condition_2(Element y, Node x){
        return y.getAttribute("src").toLowerCase().endsWith(".jpg");
    }

// body
    public wuja.tree.Tree apply()
        throws wuja.compiler.ConditionFailedException{
        Element y = (Element)vars[1];
        Node x = (Node)vars[0];
        if(!condition_2(y, x))
            throw new wuja.compiler.ConditionFailedException(
                "condition 2 failed");
        System.out.println(y.getAttribute("src"));
        return null;
    }
}

```

```
public boolean testVariable(int index, wuja.tree.Tree value){
    if(index == 1){// y
        return true && condition_1((Element)value);
    }
    if(index == 0){// x
        return true && condition_0((Node)value);
    }
    return true;
}
});
...
```

# Chapter 4

## Matcher

WUJA performs tree transformations. In order to select which subtrees to operate on, the user can specify a pattern to describe the tree. This pattern will be matched against the source tree in order to find where to apply the transformations. The module which implements the matching algorithm is the *Matcher*.

In this chapter, we will describe the Matcher from two different points of view:

**User:** How the patterns must be written and what functionality is provided.

**Internal:** How the matching process works.

### 4.1 The Pattern Language

In this section, we will give a brief, informal overview of the features of the pattern language.

#### 4.1.1 Pattern Structure

It is important to understand how patterns must be read. There is

1. a vertical direction, and
2. a horizontal direction.

Using the operators for the vertical direction, it is possible to specify how the tree elements (or *nodes*) have to relate to each other. The operators for this direction are the *path separator* ( $\backslash$ ) and the *deep path separator* ( $\backslash\backslash$ ). The *path separator* demands from the left node to be a direct parent of the right node (the left node has to be *above* the right node). The *deep path separator* however allows any number of levels to appear between the given nodes.

The horizontal direction specifies what the child list of a given tree node has to look like. The operators for this direction are the *pattern list delimiters* ([ and ]), the *sequence operator* (,) and the *disjunction operator* (|).

## 4.1.2 Overview

Basically, a pattern describes what a certain path in a tree has to look like to produce a match. For the time being, it is not possible to match complete subtrees because of the limited pattern grammar.

The basic pattern is called *node name*, which can either be the name of a JAVA class (optionally including a package name), or an arbitrary string, which is written with an at-sign (@), followed by a JAVA string constant. With the first method, you can specify the exact JAVA type of the node you want to match (as given by `object.getClass().getName()`). With the other method you can match trees of weakly typed structure. The `ClassNameMatcher` matches against the class names of the nodes whereas the `StringNameMatcher` matches against the result of the `getNodeName()` method of the node. The third method to match elements is the so-called *Joker* (?), which can be used when only the existence of any node is required. This pattern ignores the name/type of the node.

The *path separator* element denotes the parent-child relationship between two elements. A pattern like `foo\bar` describes a subtree where a node of type `foo` has (at least) one child `bar`. A match is generated for every single child `bar`.

Similar to the *path separator* operator, the *deep path* operator specifies an ancestor-successor relation. Its behaviour differs in the way that it allows any number of elements to appear between the left and the right pattern of the operator. Using this pattern element it is possible to test for the existence of a certain pattern anywhere in a (sub)tree.

In patterns, subpatterns can be bound to variables. This is written as a valid JAVA identifier followed by an equal sign (=) prepended to a node specification, e.g. one of the following:

```
x = foo
foo \\ x = bar
```

The first one will bind the variable `x` to the node matched by `foo`. The second will bind the variable `x` to all nodes matched by `bar` somewhere below `foo`. The bindings are performed whenever a condition or the body block of a rule is invoked.

The type of a given variable in conditions and in the body block is normally `wuja.tree.Tree`. To use a variable it is nearly always necessary to cast it to the type you are expecting. This is a tedious task when a great number of rules is maintained or when the class of the tree nodes changes, and there are a lot of places to make the change. Therefore, the pattern language permits the use of a cast operator after the equal sign in a variable assignment. The class name between the parentheses is taken as the expected type, and the variable will be declared with that class name in conditions and the body of the rule.

The following table summarizes all available language elements.

Operator	Description
<i>element</i>	Specifies a tree element as a class name.
@" <i>element</i> "	Specifies a tree element as a string.
\	Separates path elements.
\\	Separates path elements with an undefined number of elements between them.
<i>X=pattern</i>	Binds a pattern to a variable
<i>X=(type)pattern</i>	Binds a pattern to a variable automatically casting it to <i>type</i>
?	Matches any class- or node name
[ <i>pattern-list</i> ]	List of patterns which can be connected by the following patterns:
<i>element*</i>	Matches <i>pattern</i> zero or many times.
<i>element+</i>	Matches <i>pattern</i> many times, at least once.
<i>pattern</i> , <i>pattern-list</i>	<i>pattern</i> must be followed by <i>pattern-list</i>
( <i>element</i>   <i>element</i>   ... )	one of the elements must exist, first fitting generates match

### 4.1.3 Examples

In order to show how the pattern language is to be used, we have included some examples. The language can be used to define patterns of much more complexity, but these may show the general idea.

Pattern	Description
<code>foo</code>	Matches the tree, which consists of the element <code>foo</code> , which may have children of any kind.
<code>x=foo</code>	Matches the same tree, and binds it to the variable <code>x</code> .
<code>foo\bar</code>	Matches the tree with the root <code>foo</code> , if a child element exists which matches <code>bar</code> .
<code>foo[bar]</code>	Matches <code>foo</code> , if the only child is a <code>bar</code> element.
<code>foo[bar,braz]</code>	Matches <code>foo</code> with the only children <code>bar</code> and <code>braz</code> (in that order).
<code>foo[(bar braz)]</code>	Matches <code>foo</code> if one child <code>bar</code> or one child <code>braz</code> exists.
<code>foo\\bar</code>	Matches <code>foo</code> , if anywhere below a child <code>bar</code> can be found.
<code>foo[?* ,bar ,?*</code>	Matches <code>foo</code> , if a child <code>bar</code> does exist (synonym to <code>foo\bar</code> ).
<code>foo[bar*]</code>	Matches <code>foo</code> , if all children match <code>bar</code> .
<code>foo[bar+]</code>	Matches <code>foo</code> , if at least one child exists and all children match <code>bar</code> .

## 4.2 Implementation

This section contains some introductory notes about the implementation of the matcher module of WUJA. As a programming reference, please refer to the JAVADOC documentation, which can be created from the distribution package.

### 4.2.1 General – The Matcher

All `Matcher` classes inherit from the `Matcher` base class. This class defines the method `match()`, which is the basic interface to the matcher functionality. `match()` is applied to an instance of class `VarSet` and a tree. It returns a stream (see chapter 6, page 22 for details) of variable bindings on success, or an empty stream if no match was found. More on variable bindings in section 4.2.10, page 19.

### 4.2.2 Matching Tree Nodes – NodeMatcher

This is the base class for both `ClassNodeMatcher` and `StringNodeMatcher`. Both need to match against a tree and optionally have to match the childlist pattern against the children of the tree. This common functionality is provided by `NodeMatcher`.

### 4.2.3 Matching Class Names – ClassNodeMatcher

Nodes in WUJA are either nodes identified by their class name, or nodes identified by the result of their `getNodeName()` method. The class `ClassNodeMatcher` is instantiated

with a class name, and its `match()` method tests whether the given tree has that class name or not.

#### 4.2.4 Matching Arbitrary Strings – `StringNodeMatcher`

`StringNodeMatcher` is the corresponding matcher class to `ClassNodeMatcher`, which tests whether the result of the `match()` method matches a given string. If it does, the match was successful, otherwise it was not and the empty stream is returned.

#### 4.2.5 Binding Variables – `VarNodeMatcher`

The class `VarNodeMatcher` performs the binding of matched subtrees to variables. The association between variables and their values is maintained by objects of class `VarSet` (section 4.2.10, page 19). The `match()` method simply tests whether the associated matcher object (given to the constructor) matches, and performs the binding if it does.

#### 4.2.6 Matching Paths – `PathMatcher`

This class is the implementation of the path separator of the pattern language. The constructor is called with two matcher objects, the left and the right hand side of the operator. The matching process then first checks whether the left operand matches or not. If it does not, the whole path match failed, otherwise the right operand is matched against all children of the current tree object. For each successful match, the result stream contains the variable bindings performed by the match.

#### 4.2.7 Descending The Tree – `DeepPathMatcher`

`DeepPathMatcher` generalizes the functionality of `PathMatcher` because it allows any number of elements between the left and the right operand. For the left operand, it works exactly like `PathMatcher`, but for the right operand an instance of `DeepMatcher` is created, which performs the actual descend in the tree.

#### 4.2.8 Treewalker – `DeepMatcher`

The `DeepMatcher` is a helper class used only by `DeepPathmatcher`. It works by applying the pattern it represents against a tree, and then recursively against all children, computing all matches lazily and delivering them in a stream.

#### 4.2.9 Childlist

All `NodeMatchers` can be constructed with an additional `ChildlistMatcher` argument. This class represents a matching instance that recognizes lists of tree nodes. The pat-

tern for the list to be recognized is constructed from arguments of the constructor of `ChildlistMatcher`. A *list pattern* consists of a number of *list item matchers* that implement the `ChildlistMatcher.ListItemMatcher` interface. The construction method of each list item matcher has a `next` field which points to another list item matcher (or is `null` at the end). The following *list item matchers* are available:

- `eatItem`
- `eatWhile`
- `eatUntil`
- `alternation`

As the names suggest the matching process itself is a greedy list consuming globber. The *list item matchers* consume parts of the given node list. A node list matches a `ChildlistMatcher` if the whole node list can be consumed and all *list item matchers* of the `ChildlistMatcher` processed. Variable bindings can occur with an `eatItem` and with an `alternation` item.

The various list item matchers are implemented as inner classes of `ChildlistMatcher` which make the construction quite difficult.

For example: to create an instance that matches a node list consisting of exactly one `Foo` node the following constructor can be used:

```
new ChildlistMatcher(  
    new ChildlistMatcher().getEatItem(  
        new ClassNodeMatcher("Foo"),  
        null)  
    );
```

The `null` parameter in this example could be another `new ChildlistMatcher().getXXX()` construct.

#### 4.2.10 Storing Variable Bindings – `VarSet`

The `VarSet` class encapsulates the mapping between variable indices and values. The index assignment is done by the WUJA preprocessor. In normal operation, the user will not have to deal with objects of this class. The generated programs use anonymous subclasses of `VarSet` when instantiating `Trafo` objects (see chapter 5, page 21).

## 4.3 Some thoughts about missing features

### 4.3.1 Sequenced variables

Early versions of WUJA included a feature called 'sequenced variables' (SV). SVs were thought as an easy way to collect elements of the same type, e.g. imagine patterns like this:

`foo \ x[] = bar`: In this example the SV `x` would contain all `bar` nodes below a `foo` collected in an array.

`foo [ x[] = bar* , z[] = frotz* ]` Here the SVs would be a nice way to collect all equal elements (that are already implicitly grouped by the star)

Nonetheless this concept was removed on a second thought:

- Even simple patterns like `foo \ x[] = bar` cause semantic trouble. The first interpretation would be to collect all `bar` nodes that are below a single `foo` and return them. Another would be to collect all `bar` no matter below which `foo` they are. This gets even worse with a pattern like `foo \ \ x[] = bar`.
- A pattern like `x[] = foo \ y = bar` makes no real sense. One way to deal with this would be to collect all `foo` and then call the body for all `bar` or to forbid all *normal* variables below a SV. Both alternatives did not seem reasonable to us.
- SVs are a great way to destroy the lazy evaluation, because they require a full search in the whole subtree to fill the array.
- If the user wants to collect items he knows best where and what to collect and can implement this functionality easily with JAVA classes like `Vector` or `ArrayList`.

# Chapter 5

## Trafo

Each rule-expression is translated into a new instance of `wuja.compiler.Trafo`. For user-interaction with the rules there are 3 functions.

`Tree apply(Tree t)` tries to transform the tree `t`. If the tree can not be transformed `null` will be returned.

`Tree[] applyAll(Tree t)` transforms all sub-trees which match the defined pattern. The laziness of the matcher will be lost because all transformed trees will be returned in one array.

`Stream applyLazyAll(Tree t)` transforms also all sub-trees but returns a `Stream` instead of the array. In this way the laziness is preserved. The returned `Stream` consists of transformed `Trees`.

`Trafo concatConditionally(Trafo trafo)` is a combinator. It takes a second `Trafo` and creates a new `Trafo` which depends on both. Under normal circumstances the second won't be used but if the first/`this Trafo` returns no transformed `Tree` the second will be used.

# Chapter 6

## The Stream Feature

One feature of the matcher is that the generated matches and variable bindings are computed lazily, which means that every match which has been found will be delivered immediately to the caller, without any further computation except for some internal bookkeeping for managing the laziness. This is accomplished by utilizing the concept of *streams*. Since this programming paradigm is familiar to most readers, we will only spend a few sentences about it.

A *stream* is a data structure which contains an object and the functionality for creating a new stream. By taking the object from the stream and fetching the next stream, we can use a stream like a sequence of objects. Since every sequence element is only computed if it is actually fetched from the stream, streams avoid unneeded computation. Furthermore, streams need not necessarily be finite, like other sequence-like data structure, which can only be as large as the available computer storage.

In addition to the delayed computation of stream elements, the code for calculating a specific stream element is never executed more than once. Whenever the stream element is requested for the first time, it is calculated and the result of the calculation is remembered in the stream. This improves the performance of stream processing even more.

The drawback in using streams is that more management overhead is required than for simple non-lazy sequences. The reason is that JAVA requires the creation of helper objects in order to simulate closures.

The stream functionality is realized in the `Stream` class and the `ISuspension` interface.

### 6.1 Implementation

A stream consists of a head (which is usually a simple object) and a tail. This tail is a rule which describes how to create a successor. When the successor element is fetched, the rule is used for creating the actual successor and a new rule. This rule is also known

as **Suspension** in literature (think of a list, where the creation of elements is **suspended** after creating one object).

Since the *stream* paradigm originates in a functional world, the suspension is usually a simple, but generic function. In the JAVA implementation there are classes which provide a more or less functional environment used by the implementation. The main role in this structure plays the **ISuspension** interface, which defines a frame for a generic *function*. This function takes no arguments and returns an object, which should be a new stream (but is declared as a general object). Information needed for the generation of the new stream should be handed to the **ISuspension** implementing class with the constructors and stored in local fields, which are used when the actual **eval** method is called. This **eval** method should now generate a successor element and provide a new stream for further successors. The head element of this new stream should be set to the computed successor.

To ease the use of streams there are some high-order functions, which operate on streams and keep the laziness, if possible. These high-order functions need general function envelopes for e.g. combining two streams or doing a filter operation on stream elements. These constructs are defined in the **IFunction** interface, which declares methods for evaluating functions with single or double arguments and a predicate version. The use of this interface has been kept general since the data type for these functions is **Object**. To simplify the implementation of a single function there is semi-abstract implementation for this interface in the class **Function**, which can be easily overridden using anonymous subclassing. The current implementation of the **Stream** class contains the well-known high-order functions **map**, **filter**, **zip** and **reduce** as well as a method for concatenating two streams (**conc**), methods for string representation and some practical helper methods.

Please note that some of these high-order functions rely on the existence of the constructing stream when the new stream is used, this is due to the concept of laziness. This is also a problem, when you change one source stream (for instance cutting it after **n** elements), since this affects the new stream, too. This is a limitation of the JAVA language, which is not free of side effects. Another hazard in using the stream methods lies in the potentially infinite runtime of some algorithm, since the streams can be infinite, too. This is a feature and not a bug, so you have to care yourself in the case you want for instance print out the members of an infinite stream.

## 6.2 Example

A few examples should make the use of the **Stream** class easier. I use a stream consisting of prime numbers to show the main concepts, this stream is infinite and demonstrates the power of this paradigm.

First of all we need a class which implements the **ISuspension** Interface and is able to construct new members (read: prime numbers) if desired. To clarify the concept of

a stream I use a rather dumb method of searching for prime numbers, but since we need only one (the next) prime number, this approach should be sufficient.

```
import wuja.util.*;
public class PrimeNumbers implements ISuspension {
```

Further on we need a variable, which holds the last known prime number. This value has to be stored in the instance, because there is a gap between creating the stream (using the constructor) and doing the actual computation (calling `eval()`).

```
int lastknown;
```

In the constructors we should pass the information from previous members of the stream to the next. In this special case this is the last known prime number. If the zero constructor is used, we should assume ONE as the last known number, since the algorithm would find TWO as the next.

```
    public PrimeNumbers () {lastknown=1;}
    public PrimeNumbers (int lk) {lastknown=lk;}
```

The actual computation is done in the implemented `eval` method, which takes the last known prime number and finds the next.

```
public Object eval () {
    int i,trynext=lastknown;
    if (trynext==1) trynext=2; else {
        do {
            trynext++;
            for (i=2;(i<trynext/2) && ((trynext % i)!=0);i++);
        } while ((trynext%i)==0);
    }
}
```

The next prime number is now stored in `trynext` and should be returned to the user. Since this is a stream, we have to take care of our successors and should provide a mean to compute further prime numbers. So we create a new stream, where the head is set to the newly computed value and the tail is set to a new instance of our `PrimeNumbers` class, where the next invocation of `eval` should produce another prime number.

```
    return new Stream (new Integer(trynext),new PrimeNumbers(trynext));
}
}
(END OF FILE)
```

Now we need a program which creates some streams using the above class and making some modifications with them.

Since streams can be both finite and infinite, there are streams which have an 'end'. Such an end is created with a special constructor. If one tries to evaluate such an end-of-stream, a `LazyStreamException` will be thrown, that is why we have to specify the exception here.

```
import wuja.util.*;
public class PrimeNum {
    public static void main(String[] args) throws LazyStreamException {
```

The creation of streams is simple, since we provided a class and a zero constructor above.

```
        Stream prime=new Stream(new PrimeNumbers());
        Stream prime2=new Stream(new PrimeNumbers());
```

The first action we do is to filter the second stream and remove the prime numbers smaller than 100. If you use such operations more often it is recommended to create a helper class, where you define some basic functions to avoid the ugly-reading anonymous subclassing and typecasting.

```
Stream primefrom=prime2.filter (new Function() {
    public boolean predicate (Object x) {
        Integer cont= (Integer) x;
        return (cont.intValue())>=100);
    }
} );
```

Now we can do a fully lazy operation on both streams. There we multiply the elements of both stream with each other. Note that the actual computation is only done when the content of the stream is read.

```
Stream primemul=prime.zip (new Function(){
    public Object evalbinary (Object x, Object y) {
        Integer xi= (Integer) x;
        Integer yi= (Integer) y;
        return new Integer(xi.intValue()*yi.intValue());
    }
},primefrom);
```

To print the contents of the stream with the provided methods we must truncate the streams to a reasonable size and force a termination.

```
    primemul.truncate (10);
    System.out.println (primemul.toString(" "));
    return;
}
}
(END OF FILE)
```

Please note that the above example is only one way to use a stream. Another view on this issue could be a **pipe**, where the input can be a complex algorithm, which is suspended after the generation of the next element. This can be handy when you have to use backtracking to accomplish your aims.

# Chapter 7

## Summary

### 7.1 Future work

- First of all the childlist patterns have to be extended. It should be possible to specify a complete pattern inside a childlist pattern where only a simple node pattern can be used right now. This change will make the semantics of the pattern a bit more complicated and will need proper documentation. Only with this change it will be possible to match real subtrees from trees and not just paths.
- Furthermore real regular expressions might be allowed as childlist patterns. The implementation can (hopefully) be done by extending an existing regex library like `gnu.regex`.
- The compiler can be enhanced, too. Better error messages should be implemented.
- The compiler should check for common errors the JAVA compiler may find later.
- The compiler should not remove the comments from the file. The `.java` file that is created does not contain any JAVADOC anymore. The compiler should also add comments like the pattern a `Trafo` object was created from.
- To help the user in tracking errors it would be nice if the compiler places the transformed code in the same lines as they were in the source file.
- For easier adaption of WUJA to other tree structures (especially those the user does not have the sources of) an automatic adaption creator should be implemented. The `java.lang.reflect` methods can help here.
- The `apply()` methods of `wuja.compiler.Trafo` should have at least one additional argument. Since the application of rules is controlled by the application it should be possible to pass arguments to the rules. Global variables are not the way to do it.

- The code generator for rules should be revised, some constructs could be translated to better code.
- Variables in patterns which use the class name matching feature could be automatically cast to the class names from the pattern. Then only `@"..."` elements would need casts.
- Some help for debugging patterns should be implemented. It is hard to write working patterns for trees one is not familiar with.

## 7.2 Conclusion

We have implemented a preprocessor for the JAVA language, which adds a new language construct, and a library for tree transformation. The library provides the functionality needed by the generated JAVA code. The new expression type, a *rule*, specifies a rule to perform a transformation on trees. These trees need to implement a very simple interface which should be easy to integrate into other tree structures.

To demonstrate the possibilities of the transformation functionality, we implemented demo programs. `xml2tex.jslt` transforms XML of a specific DTD to L<sup>A</sup>T<sub>E</sub>X source, and `webtest.jslt` extracts information from HTML pages.

There is still some work to be done. The pattern language can be improved. It is also important to help the user with good documentation and hints how to solve standard problems. More test programs need to be implemented to recognize these standard problems. However, WUJA has proven to be functional and has found practical application.

# Appendix A

## Grammars

### A.1 WUJA

<i>RuleExpression</i>	::=	<u>rule</u> { <i>RuleBody</i> }
<i>Rule</i>	::=	<i>PatternExpression</i> ( <i>ConditionExpression</i> )* <i>BodyExpression</i>
<i>PatternExpression</i>	::=	<u>pattern</u> { <i>Pattern</i> }
<i>ConditionExpression</i>	::=	<u>condition</u> ( <i>ConditionArgList</i> ) <i>compoundStatement</i>
<i>BodyExpression</i>	::=	<u>body</u> <i>compoundStatement</i>
<i>ConditionArgList</i>	::=	( <i>Identifier</i> ,)* <i>Identifier</i>
<i>Identifier</i>	::=	('a'..'z' 'A'..'Z' '_' '\$') ( 'a'..'z' 'A'..'Z' '_' '0'..'9' '\$' )*

## A.2 Pattern

<i>Pattern</i>	::=	<i>VarNode</i> \ <i>Pattern</i>
<i>Pattern</i>	::=	<i>VarNode</i> $\overline{\overline{\quad}}$ <i>Pattern</i>
<i>Pattern</i>	::=	<i>FinalVarNode</i>
<i>FinalVarNode</i>	::=	<i>Assignment</i> <i>FinalNode</i>
<i>FinalVarNode</i>	::=	<i>FinalNode</i>
<i>VarNode</i>	::=	<i>Assignment</i> <i>Node</i>
<i>VarNode</i>	::=	<i>Node</i>
<i>FinalNode</i>	::=	<i>Nodename</i> ( <i>List</i> )?
<i>Node</i>	::=	<i>Nodename</i> ( <i>List</i> )?
<i>Nodename</i>	::=	<i>Identifier</i>
<i>Nodename</i>	::=	@ <i>StringLiteral</i>
<i>Nodename</i>	::=	<u>?</u>
<i>List</i>	::=	[ <i>List1</i> ]
<i>List1</i>	::=	<u>?*</u> <sub>2</sub> <i>Item1</i> <sub>2</sub> <i>List1</i>
<i>List1</i>	::=	<u>?+</u> <sub>2</sub> <i>Item1</i> <sub>2</sub> <i>List1</i>
<i>List1</i>	::=	<u>?*</u> <sub>2</sub> <i>Item1</i>
<i>List1</i>	::=	<u>?+</u> <sub>2</sub> <i>Item1</i>
<i>List1</i>	::=	<i>Item</i> <sub>2</sub> <i>List1</i>
<i>List1</i>	::=	<i>Item</i>
<i>List1</i>	::=	<u>?*</u>
<i>List1</i>	::=	<u>?+</u>
<i>Item</i>	::=	<i>Nodename</i> <u>*</u>
<i>Item</i>	::=	<i>Nodename</i> <u>±</u>
<i>Item</i>	::=	<i>Assignment</i> <i>Nodename</i>
<i>Item</i>	::=	<i>Assignment</i> <i>Alternation</i>
<i>Item</i>	::=	<i>Nodename</i>
<i>Item</i>	::=	<i>Alternation</i>
<i>Item1</i>	::=	( <i>Assignment</i> )? <i>Nodename</i>
<i>Alternation</i>	::=	( <i>Node</i> )
<i>Alternation</i>	::=	( <i>Node</i>   <i>AltList</i> )
<i>AltList</i>	::=	<i>Node</i>   <i>AltList</i>
<i>AltList</i>	::=	<i>Node</i>
<i>Assignment</i>	::=	<i>Identifier</i> $\equiv$ ( <i>Identifier</i> )
<i>Assignment</i>	::=	<i>Identifier</i> $\equiv$

# Appendix B

## Hints and common errors

### B.1 Common Errors

- Be sure to write a semicolon after the `rule`-expression if you use it in a variable declaration (as you normally do). If you leave this semicolon out you usually get a syntax error at the very next token (privacy modifier of the next variable declration, e.g. `'public'`).
- All transformations can only be done as a recursive descent, no look-ahead, parser. On the other hand your patterns should be good enough to make everything you want possible.
- The `wujac` preprocessor complains about syntax errors in your `.jslt` file (even in the plain `JAVA` parts). Those errormessage are not the same as your `JAVA` compiler would generate.
- If the `WUJA` compiler does not complain the generated `.java` file may still fail to compile to a `.class` file. This is because semantic errors (in the plain `JAVA` parts, the conditions and in the body parts of your `WUJA` rules) are only recognized by the `JAVA` compiler. This typically appears in one of the following situations.
  - The `JAVA` compiler complains about a missing or invalid return statement in a method called `condition_X` where `X` is a number. This means one of your condition parts in a rule is missing a valid `boolean` return statement.
  - The `JAVA` compiler complains about a missing or invalid return statement in a method called `apply()` in `wuja.compiler.Trafo`. Then your body part in a rule does not return an object that is an instance of `wuja.tree.Tree`.

## B.2 Hints

- One sometimes does not want to perform a tree transformation but just wants to search-and-extract certain information from a tree. Just make your body parts `return null`; and ignore the return values of the `applyXXX` methods of `wuja.compiler.Trafo`.
- Returning a `null` value when no transformation is done results in another problem: The `apply()` method of `Trafo` returns `null` when no match was found and no body part was executed. So when the you invoke `apply()` on one of the rules and get `null` as return value it is impossible to tell wether no match was found or the first match returned a `null` tree. `applyAll()` is no problem since it returns `null` when no match was found and an array of `nulls` when all bodies returned `null`.
- To avoid the above problems `wuja` provides `wuja.tree.NullTree`. This dummy class that implements the `wuja.tree.Tree` interface but does not provide any functionality. The body part of a rule can `return new wuja.tree.NullTree` to indicate that it matched but does not want to return anything useable. The user can check for `instanceof wuja.tree.NullTree` to distinguish between an empty search and a `null` return.
- There are standard implementations of `wuja.tree.Tree` and `wuja.tree.ChildnodeList`.
  - In `wuja.tree.StandardTree` you can store a `ChildnodeList`, the node name and an arbitrary payload data object.
  - `wuja.tree.Nodelist` is an `ArrayList` with the methods added to satisfy the `ChildnodeList` interface.

# Appendix C

## Resources

- The wuja API documentation can be generated from the source package. Change to the `src/` subdirectory and `make doc`. The javadoc output can then be found in the `doc/api/` directory. A prepared version can always be found at <http://user.cs.tu-berlin.de/~raimi/studium/wuja-apidoc/>.
- The ANTLR parser generator page is at <http://www.antlr.org/>.
- The Document Object Model (DOM) page is at <http://www.w3.org/DOM/>.
- Sun's java tutorial is at <http://java.sun.com/docs/books/tutorial/>.
- The JTidy page is at <http://lempinen.net/sami/jtidy>. This is a work in progress and new versions of JTidy may (and probably will) be incompatible to what we use now.
- The `gnu.getopt` package can be found at <http://www.urbanophile.com/arenn/hacking/download.html>.
- Sun's XML related products are at <http://java.sun.com/products/xml/>.
- JAVA development kits for various platforms and JAVA documentation can be found at <http://java.sun.com/>.