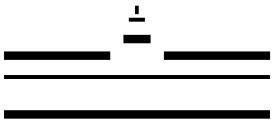



# Lethal

## Little Extensible Tree and Hedge Automaton Library

Manual

Westfälische Wilhelms-Universität Münster
Fachbereich Mathematik und Informatik
Institut für Informatik
Arbeitsgruppe Softwareentwicklung und -Verifikation
Prof. Dr. Markus Müller-Olm

by

P. Claves
D. Jansen
S. Jarrous Holtrup
M. Mohr
A. Reis
M. Schatz
I. Thesing

# Contents

# 1 Introduction

Lethal welcomes you to the world of finite tree automata.

## 1.1 About

Lethal is a Java library for working with finite tree and hedge automata. It supports a great variety of regular tree language operations and property evaluations. More advanced concepts like tree homomorphisms and transducers are also supported. Additionally the library comes with a scriptable graphical user interface to get started quickly.

## 1.2 Usage of this Manual

In order to start using the functionality of Lethal, it is not necessary to read the whole manual.

It is recommend that the reader is familiar with the basic theory of finite tree automata, but since there are different definitions in the literature, we explain the basic definitions used in Lethal in the second part of this introduction. They are based on the definitions used in [1].

Beside the programme Lethal itself, there is also a graphical user interface (GUI), which provides the possibility to deal with trees, tree and hegde automata and other objects in the theory of finite tree automata. If the reader is not familiar with the java programming language and want to get started with Lethal, read chapter 4, where it is explained how to use the GUI.

To use the libary in java applications we recommend chapter 3 where the functionalities of Lethal and ways to use them are explained. For a demonstration of the functionality also consider 5, where applying tree automata techniques to simple XML schemata is considered.

Lethal wishes you good luck.

## 1.3 Basic Definitions

To point out which definition of finite tree automata is used in the library, we list the basic definitions.

### Symbols and Trees

An ***alphabet*** is a non-empty, finite set whose elements are called ***symbols***. A ***ranked symbol*** is a symbol with an associated ***arity*** $\text{arity}(f) \in \mathbb{N}$. A ***constant*** is a ranked symbol with arity 0. A symbol without an arity is called an ***unranked symbol***. A ***ranked alphabet*** is an alphabet of ranked symbols. Similarily, an ***unranked alphabet*** is an alphabet of unranked symbols.

Given a ranked alphabet $\mathcal{F}$, the set $T(\mathcal{F})$ of ***ranked trees*** over $\mathcal{F}$ is defined inductively:

- If $f \in \mathcal{F}$ with $\text{arity}(f) = 0$, then $f \in T(\mathcal{F})$.
- If $f \in \mathcal{F}$ with $\text{arity}(f) = n \geq 1$ and if $t_0, \ldots, t_{n-1} \in T(\mathcal{F})$, then $f(t_0, \ldots, t_{n-1}) \in T(\mathcal{F})$.

Similarily, for an unranked alphabet $\mathcal{F}$, the set $U(\mathcal{F})$ of ***unranked trees*** over $\mathcal{F}$ is defined inductively:

- If $f \in \mathcal{F}$, then $f \in T(\mathcal{F})$.
- If $f \in \mathcal{F}$, $n \in \mathbb{N}$ and $t_0, \ldots, t_{n-1} \in U(\mathcal{F})$, then $f(t_0, \ldots, t_{n-1}) \in U(\mathcal{F})$.

Given a tree $t = f(t_0, \ldots, t_{n-1})$ over a ranked or an unranked alphabet, the symbol $f$ is also called ***root symbol***, and the $t_i$ are called ***subtrees***. If $n = 0$, we sometimes call $t$ a ***leaf***.

Note the difference between ranked and unranked trees: In a ranked tree, the arity of the root symbols is always equal to the number of subtrees, whereas in an unranked tree there can be arbitrarily many subtrees.

### Finite Tree Automata

A ***(bottom-up) finite tree automaton*** is a quadruple $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ where

- $\mathcal{F}$ is a ranked alphabet,
- $\mathcal{Q}$ is a finite set of ***states***,
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of ***final states***,

- $\mathcal{R}$ is a finite set of ***rules***, which have the form $r = f(q_0, \ldots, q_{n-1}) \to q$, where $f \in \mathcal{F}$, arity$(f) = n$ and $q_0, \ldots, q_{n-1}, q \in Q$. The states $q_0, \ldots, q_{n-1}$ are called ***source states*** and $q$ is called ***destination state*** of $r$.

Let $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ be a finite tree automaton and $t = f(t_0, \ldots, t_{n-1})$. We define that $t$ ***can be annotated with*** or ***transformed to a state*** $q$ by an inductive definition: $t$ can be annotated with $q$ if there is a rule $r = f(q_0, \ldots, q_{n-1}) \to q$ in $\mathcal{R}$ and $t_i$ can be annotated with $q_i$ for $i = 0, \ldots, n-1$. For $n = 0$ and a leaf $t$ this means: the tree $t$ can be annotated with a state $q$, if there is a rule $t \to q$ in $\mathcal{R}$.

Furthermore, there is a special type of rules, the ***epsilon rules***. They have the form $p \to q$ where $p, q$ are states. An epsilon rule $p \to q$ is applicable to $t$ if $t$ can be annotated with $p$. Applying the rule then then annotating $t$ with $q$.

We use the terminology that $t$ is ***accepted*** or ***recognized by*** $\mathcal{A}$, if $t$ can be annotated with a final state of $\mathcal{A}$. If $t$ cannot be annotated with a final state of $\mathcal{A}$, we say that $t$ is ***rejected*** by $\mathcal{A}$. The set of all trees recognized by $\mathcal{A}$ is also called the ***language of*** $\mathcal{A}$ and denoted by $\mathcal{L}(\mathcal{A})$.

This concept can be extended by means of the so-called ***configuration trees***. Regard states as special constants, then a ***configuration tree*** $t \in T(()\mathcal{F} \cup \mathcal{Q})$ is a tree consisting of ranked symbols, whose leave nodes can also be states. Thus a configuration tree $t = f(t_1, \ldots, t_n)$ can be annotated by a state $q$ if and only if either there is a rule $r = f(q_1, \ldots, q_n)$ of $\mathcal{R}$ and each $t_i$ can be annotated with $q_i$ or $t = q$. In this way a configuration tree can be identified with a set of trees (consisting only of ranked symbols), namely the trees constructed by replacing the state leaves by trees which the finite tree automaton can annotate with the states which are replaced.

Let $\mathcal{F}$ be a ranked alphabet and $\mathcal{L} \subseteq T(\mathcal{F})$. Then $\mathcal{L}$ is called ***regular tree language***, if and only if there is a finite tree automaton $\mathcal{A}$ with $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

### Hedge Automata

Let $\mathcal{S}$ be an alphabet, then by $\mathcal{R}$egExp$(S)$ we denote the set of regular expressions over $S$ with the usual meaning. If $r \in \mathcal{R}$egExp$(S)$ is such a regular expression, then $L(r)$ denotes the regular (word) language represented by r. A ***hedge automaton*** is a quadruple $\mathcal{H} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ where

- $\mathcal{F}$ is an unranked alphabet,
- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{Q}_f$ is a finite set of final states,
- $\mathcal{R}$ is a finite set of rules of the form $f(e) \to q$ with $f \in \mathcal{F}, e \in \mathcal{R}$egExp$(Q)$ and $q \in \mathcal{Q}$.

Let $\mathcal{H} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ be a hedge automaton and $t = f(t_0, \ldots, t_{n-1})$ a tree over $\mathcal{F}$. We say that $t$ can ***be annotated with*** or ***transformed to a state*** $q$, if there is a rule $r = f(e) \to q$ in $\mathcal{R}$ and if the $t_i$ can be annotated by states $q_i$ for $i = 0, \ldots, n-1$, such that the word $q_0 \ldots q_{n-1}$ matches $e$, i.e. $q_0 \ldots q_{n-1} \in L(e)$. We say that $t$ is ***accepted*** or ***recognized*** by $\mathcal{H}$, if $t$ can be annotated by a final state of $\mathcal{H}$. If $t$ cannot be annotated by a final state of $\mathcal{H}$, we say that $t$ is ***rejected*** by $\mathcal{H}$. The set of all trees recognized by $\mathcal{H}$ is also called the ***language of*** $\mathcal{H}$ and denoted by $\mathcal{L}(\mathcal{H})$.

### Tree Homomorphisms

Let $\mathcal{F}$ be a ranked alphabet and let $\mathcal{X}$ be a finite set of special constants, called ***variables*** with $\mathcal{F} \cap \mathcal{X} = \emptyset$. Then the set $T(\mathcal{F}, \mathcal{X})$ of trees over $\mathcal{F}$ and $\mathcal{X}$—or ***variable trees*** for short—is defined by $T(\mathcal{F}, \mathcal{X}) := T(\mathcal{F} \cup \mathcal{X})$.

Let $t \in T(\mathcal{F}, \mathcal{X})$ be a variable tree and let $\mathcal{X} = \{x_0, \ldots, x_{n-1}\}$ be variables (not neccessarily occurring in $t$). Then $t(x_0 \leftarrow t_0, \ldots, x_{n-1} \leftarrow t_{n-1})$ is the tree obtained by replacing every occurrance of $x_i$ by $t_i$ for each $i = 0, \ldots, n-1$.[1]

Let $\mathcal{F}$ and $\mathcal{G}$ be two ranked alphabets and let $hom : \mathcal{F} \to T(\mathcal{F}, \mathcal{X})$ be a map with the following property:

> If $f \in \mathcal{F}$ with $\text{arity}(f) = n$, then $hom(f)$ contains at most $n$ variables.

The map $hom$ defines a map $h = h(hom) : T(\mathcal{F}) \to T(\mathcal{G})$ inductively:

- If $t = c$ with $\text{arity}(c) = 0$, then $h(t) := hom(c)$. (By definition, $hom(c)$ does not contain any variables, so this is well-defined.)
- If $t = f(t_1, \ldots, t_n)$ for $arity(f) = n$ and if $hom(f)$ contains variables out of $x_1, \ldots, x_n$, then $h(t) := hom(f)(x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n))$. (By definition, $hom(f)$ contains at most $n$ variables, so every variable is replaced by a tree, leading to a tree of $T(\mathcal{G})$ which does not contains variables.)

If $h : T(\mathcal{F}) \to T(\mathcal{G})$ is a map with $h = h(hom)$ for a map $hom : \mathcal{F} \to T(\mathcal{F}, \mathcal{X})$ as above, $h$ is called a ***tree homomorphism*** with source alphabet $\mathcal{F}$ and destination alphabet $\mathcal{G}$.

### Tree Transducer

A ***(bottom-up) tree transducer*** is a quintuple $\mathcal{U} = (\mathcal{F}, \mathcal{G}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ where

- $\mathcal{F}$ and $\mathcal{G}$ are ranked alphabets,

[1] In particular, if a variable does not occur in a tree, replacing it by any tree has no effect.

- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states,
- $\mathcal{R}$ is a set of rules, which are of the form $f(q_0, \ldots, q_{n-1}) \to (q, u)$ with $f \in \mathcal{F}$, $\text{arity}(f) = n$, $q_0, \ldots, q_{n-1}, q \in \mathcal{Q}$ and $u \in T(\mathcal{G}, \mathcal{X})$ contains at most $n$ variables out of $x_0, \ldots, x_{n-1}$.

A tree transducer with epsilon rules can additionally have rules of the form $q \to (r, u)$ with $q, r \in \mathcal{Q}$, $u \in T(\mathcal{G}, \mathcal{X})$.

A tree transducer contains a finite tree automaton, which is obtained by omitting the variable trees occurring on the right sides of the rules. Thus the definitions introduced for finite tree automata can be transfered to tree transducers.

A tree $t = f(t_0, \ldots, t_{n-1}) \in T(\mathcal{F})$ can be transduced into a tree $t' \in T(\mathcal{G})$, if there is a rule $f(q_0, \ldots, q_{n-1}) \to (q, u)$, such that

- the underlying rule $r' = f(q_0, \ldots, q_{n-1}) \to q$ can be applied to $t$ (i.e. $t$ can be annotated by $q$ in the underlying finite tree automaton),
- $t' = u(x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n)$, i.e. each variable $x_i$ is replaced by a tree $s_i \in T(\mathcal{G})$ which $t_i$ can be transduced to by means of the tree transducer (for each $i \in \{0, \ldots, n-1\}$).

(D. Jansen, M. Mohr, I. Thesing)

# 2 Tutorial

For getting started we explain some features of Lethal in an example in this chapter. To see the results of the example and the solutions of the exercises, start the main-method of `testManual.Tutorial`. There the example is implemented.

Table 2.1: Overview over packages that are used in the tutorial

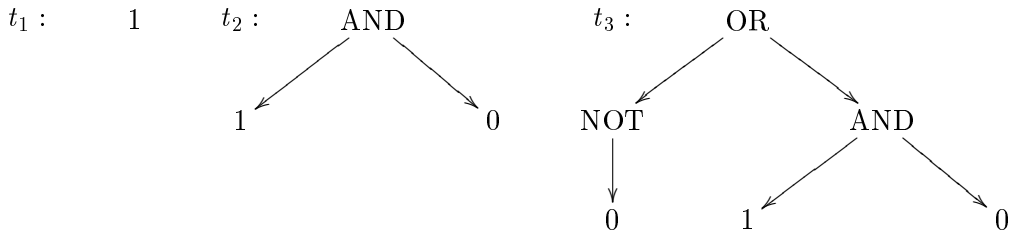
| Package | Subpackage | Description |
|---|---|---|
| symbols | | Different symbol types, ranked and unranked symbols |
| symbols | common | Interfaces for ranked, unranked and named symbols |
| trees | | Interface for trees and standard implementations. |
| trees | common | Interfaces for trees and operations on trees. |
| parser | | Different parsers to create trees, finite tree automata, homomorphism and other objects out of strings and XML-documents. |
| parser | tree | Parser to create a tree or a hedge out of a string. |
| parser | fta | Parser to create a finite tree automaton out of a string. |
| parser | homomorphism | Parser to create a homomorphism out of a string. |
| parser | hedgeautomaton | Parser to create a hedge automaton out of a string. |
| treeautomata | | Classes to represent finite tree automata and to operate on them. |
| treeautomata | easy | Used implementation of finite tree automata without generic type paramters. |
| hom | | Representing tree homomorphisms and operating with them. |
| hedgeautomata | | Classes to represent hedge automata and to operate on them. |

## 2.1 Beginning with an Example

Let us now begin to examine an example for the usage of Lethal. Consider the alphabet $\mathcal{B} = \{\text{AND}, \text{OR}, \text{NOT}, 0, 1\}$, which is used for boolean expressions. Note that every symbol in $\mathcal{B}$ has an arity, i.e. a number of arguments it expects. For example, 1 expects no argument and AND expects two arguments.

## 2.2 Trees

Some trees containing symbols of $\mathcal{B}$ are for example $t_1 = 1, t_2 = \text{AND}(1, 0)$ and $t_3 = \text{OR}(\text{NOT}(0), \text{AND}(1, 0))$ or in graphical notation:




The easiest way to create trees is using the `TreeParser`. The parser expects the string representation of the tree:

Example 2.1: Creating Trees with the TreeParser

```
1 Tree<RankedSymbol> tree_1
2   = TreeParser.parseString("1");
3 Tree<RankedSymbol> tree_2
4   = TreeParser.parseString("and(1,0)");
5 Tree<RankedSymbol> tree_3
6   = TreeParser.parseString("or(not(0),and(1,0))");
```

For further information on symbols and trees consider section 3.1.

## 2.3 Finite Tree Automata

To create finite tree automata it is possible to use the finite tree automaton parser (the same one that the GUI uses). The rules are given straight forward, close to the definitions in Chapter 1. The tailing „!“-mark denotes the final states of the automaton. It does not matter whether it is appended to all or only some occurrences of a state name. Section 4.2.2.2 lists the exact syntax.

Now we create a finite tree automaton accepting all true formulas:

Example 2.2: Creating a Finite Tree Automaton using the Parser

```
String ruleString = "";
ruleString += "1 -> t! \n";
ruleString += "0 -> f \n";
ruleString += "not(t) -> f \n";
ruleString += "not(f) -> t! \n";
ruleString += "and(t,t) -> t! \n";
ruleString += "and(t,f) -> f \n";
ruleString += "and(f,t) -> f \n";
ruleString += "and(f,f) -> f \n";
ruleString += "or(t,t) -> t! \n";
ruleString += "or(t,f) -> t! \n";
ruleString += "or(f,t) -> t! \n";
ruleString += "or(f,f) -> f \n";
EasyFTA fta_trueFormulas = FTAParser.parseString(ruleString);
```

Another example is the finite tree automaton accepting all positive formulas, i.e. all formulas containing only $0, 1$, AND and OR.

Example 2.3: Creating a Finite Tree Automaton using the Parser

```
ruleString = "";
ruleString += "1 -> q! \n";
ruleString += "0 -> q! \n";
ruleString += "not(q!) -> f \n";
ruleString += "and(q!,q!) -> q! \n";
ruleString += "or(q!,q!) -> q! \n";
EasyFTA fta_positiveFormulas = FTAParser.parseString(
    ruleString);
```

**Exercise 1.** Create a finite tree automaton `fta_disjuctiveNF` accepting all formulas in disjunctive normal form. These formulas have the form $\phi_1 \vee \cdots \vee \phi_n$, where $\phi_i = \psi_1 \wedge \cdots \wedge \psi_{r_i}$ and $\psi_j = 1$,$\psi_j = 0$, $\psi_j = \neg(1)$ or $psi_j = \neg(0)$.

Note: Since the arity of AND and OR is two, it is not possible to create bigger conjunctions or disjunctions directly. So do it inductively, for example $\phi_0 \wedge \phi_1 \wedge \phi_2 = \mathrm{AND}(\phi_0, \mathrm{AND}(\phi_1, \phi_2))$.

For further possibilities to create a finite tree automaton consider section 3.2.3.

### 2.3.1 Use Finite Tree Automata

After having defined some finite tree automata, it is possible to check whether a special tree is contained in the language of the finite tree automaton, i.e. whether the finite tree automaton accepts the tree.

Example 2.4: Decide

```
fta_trueFormulas.decide(tree_1); // yields true
fta_trueFormulas.decide(tree_2); // yields false
fta_trueFormulas.decide(tree_3); // yields true
fta_positiveFormulas.decide(tree_1);
fta_positiveFormulas.decide(tree_2);
fta_positiveFormulas.decide(tree_3);
```

**Exercise 2.** 1. Which results are returned by the last three lines?

2. Apply the finite tree automaton `fta_disjuctiveNF` on the example trees and check whether the results are correct.

### 2.3.2 Properties of a Finite Tree Automaton

Lethal provides functions to check several properties of a finite tree automaton. The class `FTAProperties` contains several methods, e.g. methods to check whether the finite tree automaton is deterministic or whether it is complete. If the property does not hold, it is possible to determinize or complete a finite tree automaton using the corresponding method of `EasyFTAOps`.

Regarding the further examples, `fta_trueFormulas` is deterministic and complete. The automaton `fta_positiveFormulas` is deterministic, but not complete, since it contains no formula with the state f on the left side of a rule.

Example 2.5: Checking some Properties

```
FTAProperties.checkDeterministic(fta_trueFormulas); // yields
    true
FTAProperties.checkComplete(fta_positiveFormulas); // yields
   false
EasyFTA fta_positiveFormulasComplete = EasyFTAOps.complete(
   fta_positiveFormulas);
EasyFTA fta_positiveFormulasReduced = EasyFTAOps.reduceFull(
   fta_positiveFormulas);
```

**Exercise 3.** Which properties has the finite tree automaton `fta_disjuctiveNF` of the previous excercise? Complete respectively determinize it, if it does not have the corresponding property.

Consider 3.3.2 for detailed information on properties of the finite automaton.

### 2.3.3 Operations on the Language of a Finite Tree Automaton

Further important features are operations on the language of the finite tree automata including complement, union and intersection. Moreover, properties like emptiness and finiteness can be checked.

For example, the regular tree language of all false boolean formulas can be obtained by complemenation of the language of all true boolean formulas. Furthermore, the intersection of `fta_trueFormulas` and `fta_disjuctiveNF` returns the regular tree language of all true formulas in disjunctive normal form.

Note that there are different intersection methods in Lethal. They differ in the way of applying automatic reduction to the result automaton. Details on that are found in Section 3.3.4.3. For now we will just use `intersectionTD`.

Example 2.6: Operations on Finite Tree Automata

```
1  EasyFTA fta_falseFormulas = EasyFTAOps.complement(
       fta_trueFormulas);
2  EasyFTA fta_trueConjuctiveNF = EasyFTAOps.intersectionTD(
       fta_trueFormulas,fta_disjunctiveNF);
3  EasyFTA fta_truePositiveFormulas = EasyFTAOps.intersectionTD(
       fta_positiveFormulas,fta_trueFormulas);
4  EasyFTA fta_allFormulas = EasyFTAOps.union(fta_trueFormulas,
       fta_falseFormulas);
5
6  FTAProperties.subsetLanguage(fta_trueConjuctiveNF,
       fta_trueFormulas); // yields true
7  FTAProperties.subsetLanguage(fta_positiveFormulas,
       fta_trueFormulas); // yields false
8  FTAProperties.subsetLanguage(fta_falseFormulas,
       fta_allFormulas); // yields true
9  FTAProperties.sameLanguage(fta_falseFormulas,
       fta_positiveFormulas); // yields false
10 FTAProperties.finiteLanguage(fta_positiveFormulas); // yields
       false
11 FTAProperties.emptyLanguage(GenFTAOps.complement(
       fta_allFormulas)); // yields true
```

There are many operations on finite tree automata which are not yet explained. The reader is asked to have a closer look at chapter 3 for detailed explanation.

## 2.4 Homomorphism

Trees can be transformed by means of a homomorphism. A homomorphism is completely determined by its behavior on the symbols of the alphabet, which in the standard example is $\mathcal{B}$.

In the following example we shall define a homomorphism which transforms formulas containing symbols of $\mathcal{B}$ into equivalent formulas not containing AND.[1]

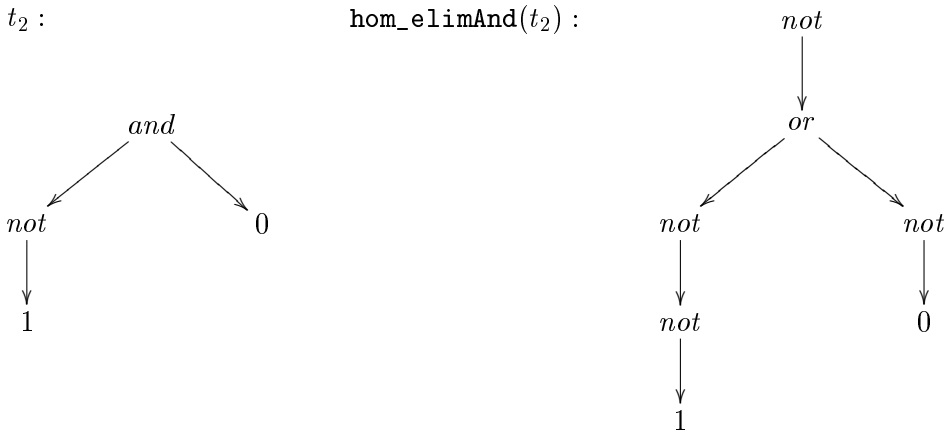
Again a parser which transforms the basic map of a homomorphism is used. To declare it, the variables are specified on the left hand side, on the right hand side the image tree with variables is given.

For more information on the used syntax see 4.2.2.3.

Example 2.7: Creating a Homomorphism

```
1 String homMap = "";
2 homMap += "0 -> 0 \n";
3 homMap += "1 -> 1 \n";
4 homMap += "not(x) -> not(x) \n";
5 homMap += "or(x,y) -> or(x,y) \n";
6 homMap += "and(x,y) -> not(or(not(x),not(y)))";
7 EasyHom hom_elimAnd = HomomorphismParser.parseString(homMap);
```

Applying this homomorphism on `tree2` supplies:




[1] This is possible, since the junctor system {OR, NOT} is complete.

### 2.4.1 Applying the homomorphism

Furthermore it is possible to apply a linear homomorphism (or the inverse one) to a regular tree language.

Example 2.8: Applying a homomorphism

```
// tree_1_elimAnd contains no AND an therefore is not changed
// tree_2_elimAnd and tree_3_elimAnd both contain AND
// and therefore are changed
Tree<RankedSymbol> tree_1_elimAnd
  = hom_elimAnd.apply(tree_1);
Tree<RankedSymbol> tree_2_elimAnd
  = hom_elimAnd.apply(tree_2);
Tree<RankedSymbol> tree_3_elimAnd
  = hom_elimAnd.apply(tree_3);
if (hom_elimAnd.isLinear()){ // supplies true
  EasyFTA fta_true_elimAnd
    = hom_elimAnd.applyOnAutomaton(fta_trueFormulas);
  EasyFTA fta_foo
    = hom_elimAnd.applyOnAutomaton(fta_positiveFormulas);
}
```

The `fta_true_elimAnd` accepts all true formulas over $\mathcal{B} - \{\text{AND}\}$. Note that the formulas accepted by `fta_foo` are in general not positive.

For more information on homomorphisms the reader is asked to have a closer look on 3.

## 2.5 Hedge Automata

As finite tree automata merely work with ranked symbols, they are quite restrictive. A good example is presented in the first exercise, where we can not use the functions AND or OR with an arbitrary number of arguments.

Fortunately, Lethal supports hedge automata, which use trees with unranked symbols (called hedges).

In the following examples, the alphabet $\mathcal{C} = \{\text{AND\_ALL}, \text{OR\_ALL}, \text{NOT}, 0, 1\}$ will be used. All those symbols can have an arbitrary number of subtrees, i.e. the number of them is only determined by rules of the hedge automaton they are used in. Thereby AND_ALL is to represent the conjunction of arbitrary many formulas, whereas OR_ALL is the disjunction. The other symbols are used as above.

### 2.5.1 Hedges

In order to create unranked trees for the hedge automata to work with, the method `parseStringAsHedge()` of the tree parser can be used.

Example 2.9: Defining Trees with Unranked Symbols

```
1 Tree<UnrankedSymbol> tree_4 = TreeParser.parseStringAsHedge("
    ANDALL(not(0),0,ORALL(1))");
2 Tree<UnrankedSymbol> tree_5 = TreeParser.parseStringAsHedge("
    ORALL(ANDALL(1,not(0)),ANDALL(0),ANDALL(not(1),not(0),1))"
    );
```

Notice that `tree5` is in disjunctive normal form, whereas `tree4` is not.

### 2.5.2 Defining Hedge Automata

The following hedge automaton is produced by the `HedgeAutomatonParser`. It recognizes all formulas in disjunctive normal form.

Example 2.10: Defining a Hedge Automaton

```
1  String haString = "";
2  haString += "1 -> p \n";
3  haString += "0 -> p \n";
4  haString += "1 -> q \n";
5  haString += "0 -> q \n";
6  haString += "not(0) -> q \n";
7  haString += "not(1) -> q \n";
8  haString += "ANDALL(q*) -> r \n";
9  haString += "ORALL(r*) -> s!";
10 EasyHedgeAutomaton ha = HedgeAutomatonParser.parseString(
     haString);
```

On the left side of the rule a hedge automaton accepts a regular expression of states, not only a list like finite tree automata.

### 2.5.3 Operations of Hedge Automata

Lethal provides several operations on hedge automata in the class `EasyHAOps`.

It is possible to decide whether a tree is contained in the language of a hedge automaton. Furthermore, the operations on languages (like `union` or `sameLanguage`) of finite tree automata are applicable on hedge automata via `EasyHAOps` as well. Note that not all finite tree automaton operations are supported for hedge automata—e.g. determinize.

Example 2.11: Using a Hedge Automaton

```
EasyHAOps.decide(ha,tree_4); // yields false
EasyHAOps.decide(ha,tree_5); // yields true
EasyHAOps.finiteLanguage(ha); // yields false
EasyHedgeAutomaton ha_complement = EasyHAOps.complement(ha);
```

More on hedge automata and their use is explained in chapter 3.8.

(D. Jansen, I. Thesing)

# 3 The Library

## 3.1 Symbols and Trees

As it does not make any sense to talk about tree automata without talking about trees, let us consider trees first: `Tree` is an interface that represents a tree over symbols. To specify different trees over different symbol types, there exists an interface `Symbol`. There are different types of symbols: Symbols with an arity are represented by the interface `RankedSymbol`, symbols without arity by `UnrankedSymbol`.

### 3.1.1 The Several Symbol Types

For several symbol types there are interfaces, which can be implemented in order to use the library for user-defined symbol types. It is necessary to implement the methods `equals()` and `hashCode()` in a consistent way, i.e. that the hash code is the same if the objects are equal.

#### 3.1.1.1 NamedSymbol

`NamedSymbol` is a subtype of `Symbol`, which attributes a name to a symbol.

The standard implementation is AbstractNamedSymbol. For using this implementation, it is necessary to make sure that the name is not changed after creating the symbol. Use the subclasses `StdNamedRankedSymbol` and `StdNamedUnrankedSymbol`.

#### 3.1.1.2 RankedSymbol

A ranked symbol is a symbol which has an arity in $\mathbb{N}$. In the library it is used for building trees and finite tree automata which can be applied to these trees. To implement `RankedSymbol`, it is necessary that the method `getArity()` returns a non-negative value.

The standard implementation `StdNamedRankedSymbol` implements the interfaces `NamedSymbol` and `RankedSymbol` in a canonical way. It is parametrized by the type of the name. To create a ranked symbol of this type a name and an arity has to be provided.

Example 3.1: Creating a new RankedSymbol

```
1 RankedSymbol s = new StdNamedRankedSymbol<String>("s",1);
```

#### 3.1.1.3 UnrankedSymbol

An unranked symbol is a symbol which explicitly has no arity, i.e. there can be arbitrarily many subtrees if the symbol is used in a tree. Trees over `UnrankedSymbol` are also called hedges. Thus unranked symbols are used to build hedges and hedge automata which can be applied to them.

The standard implementation `StdNamedUnrankedSymbol` implements the interfaces `NamedSymbol` and `UnrankedSymbol` in a canonical way. To create a ranked symbol of this type a name is required.

Example 3.2: Creating a new UnrankedSymbol

```
1 UnrankedSymbol s
2  = new StdNamedUnrankedSymbol<String>("s");
```

#### 3.1.1.4 BiSymbol

There are several applications with trees consisting of two different symbol types, where the second symbol type only occurs in the leaves of a tree. The inner type must be a symbol. An example are trees containing variables or states at the leaves. For examples consider also the section about homomorphisms (3.7).

In order to implement this interface, make sure that the following conditions hold:

- `isInnerType()==true` implies `asInnerSymbol()!= null`
- `isLeafType()==true` implies `asLeafSymbol()!= null`

The standard implementation consists of `InnerSymbol<I>` and `LeafSymbol<L>`. The class `InnerSymbol` wraps a symbol of type `I`, so that it is used as an inner node, i.e. it can have subtrees. `LeafSymbol` wraps an arbitrary object and is only to be used as a leaf node in trees.

To create a tree representing an arithmetic expression, it is possible to create the corresponding symbols like this:

Example 3.3: Using BiSymbols

```
1 BiSymbol<StdNamedRankedSymbol<String>,Integer> plus
2   = new InnerSymbol<StdNamedRankedSymbol<String>,Integer>(
3       new StdNamedRankedSymbol<String>("+",2));
4 BiSymbol<StdNamedRankedSymbol<String>,Integer> mult
```

```
5    = new InnerSymbol<StdNamedRankedSymbol<String>,Integer>(
6        new StdNamedRankedSymbol<String>("*",2));
7  BiSymbol<StdNamedRankedSymbol<String>,Integer> x
8    = new InnerSymbol<StdNamedRankedSymbol<String>,Integer>(
9        new StdNamedRankedSymbol<String>("x",0));
10 BiSymbol<StdNamedRankedSymbol<String>,Integer> y
11   = new InnerSymbol<StdNamedRankedSymbol<String>,Integer>(
12       new StdNamedRankedSymbol<String>("y",0));
13 BiSymbol<StdNamedRankedSymbol<String>,Integer> i1
14   = new LeafSymbol<StdNamedRankedSymbol<String>,Integer>(
15       new Integer(1));
16 BiSymbol<StdNamedRankedSymbol<String>,Integer> i42
17   = new LeafSymbol<StdNamedRankedSymbol<String>,Integer>(
18       new Integer(42));
```

### 3.1.2 Creating Trees

Every tree consists of symbols of a certain type. For normal usage of finite tree automata a subtype of `RankedSymbol` is required, whereas hedge automata need subtypes of `UnrankedSymbol`.

A tree consists of a root (which is a symbol) and of subtrees. It is given by the interface `Tree`. It is possible to implement this interface directly or to use the standard implementation `StdTree`. In order to use two different symbol types, it is necessary to create a tree over `BiSymbol`. To make it easier, Lethal offers the standard implementation `StdBiTree<I,L>`.

#### 3.1.2.1 Using the Tree Factory

The recommended way to create trees is to use the tree factory. A tree factory is returned by the method `getTreeFactory()`. With this factory trees can be made by supplying the root symbol and—if necessary—the subtrees as a list.

Example 3.4: Creating Trees using a TreeFactory

```
1 RankedSymbol x = new StdNamedRankedSymbol<String>("x",0);
2 RankedSymbol plus = new StdNamedRankedSymbol<String>("+",2);
3
4 TreeFactory fact = TreeFactory.getTreeFactory();
5 Tree<RankedSymbol> tree_x = fact.makeTreeFromSymbol(x);
6 List<Tree<RankedSymbol>> subtrees = new LinkedList<Tree<
    RankedSymbol>>();
7 subtrees.add(tree_x);
8 subtrees.add(tree_x);
```

```
9 Tree<RankedSymbol> tree_1 = fact.makeTreeFromSymbol(plus,
    subtrees);
```

#### 3.1.2.2 Using the Tree Parser

In order to use the factories with named symbols of type `String`, it is also possible to use the parser. The parser creates trees over symbols of type `StdNamedRankedSymbol` out of a given String term representation.

Example 3.5: Creating Trees using the Parser

```
1 Tree<RankedSymbol> tree_2 = TreeParser.parseString("x");
2 Tree<RankedSymbol>tree_3 = TreeParser.parseString("+(*(x,y),y
    )");
```

### 3.1.3 Operations on Trees

For working with trees, the library provides some operations on trees, which are located in the class `tree.common.TreeOps`.

Example 3.6: Some examples in TreeOps

```
1  Tree<RankedSymbol> tree_2
2    = TreeParser.parseString("x");
3  Tree<RankedSymbol> tree_3
4    = TreeParser.parseString("+(*(x,y),y)");
5  // calculates height
6  int height = TreeOps.getHeight(tree_3);
7  // collects all symbols occurring in the tree
8  Set<RankedSymbol> symbols = TreeOps.getAllContainedSymbols(
     tree_3);
9  // checks whether a symbol is contained in the tree
10 boolean f = TreeOps.containsSymbol(tree_2, new
     StdNamedRankedSymbol<String>("x",0));
```

It is possible to convert a tree over one symbol type into a tree over another symbol type by implementing the `Converter` class, which converts one symbol into the other one:

Example 3.7: Using a Converter

```
1 Converter<RankedSymbol,BiSymbol<RankedSymbol,Integer>> conv
2   = new Converter<RankedSymbol,
3       BiSymbol<RankedSymbol,Integer>>(){
4   @Override
```

```
5   public BiSymbol<RankedSymbol,Integer> convert(RankedSymbol
        a) {
6     return new InnerSymbol<RankedSymbol,Integer>(a);
7   }
8 };
9
10 RankedSymbol x = new StdNamedRankedSymbol<String>("x",0);
11 Tree<RankedSymbol> tree_x
12   = TreeFactory.getTreeFactory().makeTreeFromSymbol(x);
13
14 // converts a tree over the first symbol type into a tree
       over the second symbol type
15 Tree<BiSymbol<RankedSymbol,Integer>> bitree_x = TreeOps.
       convert(tree_x, conv,
16   new StdTreeCreator<BiSymbol<RankedSymbol,Integer>>());
```

For information about `TreeCreator` and `Converter` consider chapter 3.4.

(D. Jansen, M. Mohr, I. Thesing)

## 3.2 Finite Tree Automata

After having explained trees and symbols, we concentrate now on finite tree automata. `FTA` is an interface that represents a finite tree automaton having an alphabet, states, final states and rules.

According to the definition, a finite tree automaton $\mathcal{A}$ consists of an alphabet, a set of states, a set of final states and a set of rules, where the final states are a subset of the set of states and each symbol of the alphabet has a non-negative arity. Thus the symbols are `RankedSymbol`s. The rules have the form $f(q_1, \ldots, q_n) \to q$, where $q_1, \ldots, q_n, q$ are states and $f$ is a symbol of the alphabet with arity $n$.

A tree $t = f(t_1, \ldots, t_n)$ is annotated with a state $q$ by the finite tree automaton $\mathcal{A}$, if $\mathcal{A}$ has a rule $f(q_1, \ldots, q_n) \to q$ and $t_1, \ldots, t_n$ can be annotated with $q_1, \ldots, q_n$ by $\mathcal{A}$. $\mathcal{A}$ ***accepts*** or ***recognizes*** $t$, if $\mathcal{A}$ can annotate $t$ with a final state.

In order to create finite tree automata, states and rules are required. So we shall consider these first.

### 3.2.1 States

For states there is the marker interface `State`. In order to use a user-defined state type, this type must implement the interface `State`.

There is a special state type called `NamedState<T>`. A state of this type is identified with an arbitrary object of type T, which is called the ***name*** of this state. Additionally, there is a state type called `NumberedState`. Its behaviour is very similar to

`NamedState<Integer>`, but for efficiency reasons, the name is just an *int*. This state type is used for `RegularTreeLanguage<F>`.

#### 3.2.1.1 Creating States

States can be created by directly instantiating or by using the `StateFactory`. A standard state factory which returns states of type `NamedState` can be obtained by calling `StateFactory.getStateFactory()`.

The state factory has two methods: `makeState()`, which returns a new state with a generated name and `makeState(T name)`, which returns a state identified by the given name. The type of the name is arbitrary, for example a String or a pair of states.

Example 3.8: Creating States using a StateFactory

```
1 StateFactory stateFact = StateFactory.getStateFactory();
2 State s1 = stateFact.makeState();
3 State s2 = stateFact.makeState("s2");
4 State s3 = stateFact.makeState(new Pair<State,State>(s1,s2));
```

**Warning.** *Due to performance issues, the hash code of `NamedState` is precomputed using the supplied name. Thus, the name must not be changed after creation of a `NamedState`, since this might cause problems like in the following example.*

Example 3.9: Fun with `NamedState`

```
1 LinkedList<Integer> name = new LinkedList<Integer>();
2 name.add(42);
3 name.add(17);
4 State q = new NamedState<List<Integer>>(name);
5 name.add(23);
```

*Here the name of `q` corresponds to the list `[42,17,23]`, but the hash code still corresponds to `[42,17]`, so the name and hash code of `q` do not fit any more. Thus the state is not identified by its name anymore: If you create another state out of the list `[42,17,23]`, it will not be equal to `q`.*[1]

#### 3.2.1.2 User-defined State Types

To create special state types, implement the interface `State`. Creating new States of this type can be accomplished by means of the constructor or by implementing a custom state factory.

[1] Since the precomputed hash code is included into the `equals`-method to comply with the java contract of `hashCode` and `equals`.

### 3.2.2 Rules

Recall that a rule of a finite tree automaton has the form $f(q_1, \ldots, q_n) \to q$, where $q_1, \ldots, q_n, q$ are states and $f$ is a symbol with arity $n$.

#### 3.2.2.1 EasyFTARule

Rules can be created by using `EasyFTARule` where types of states and symbols are fixed and need not to be supplied. There are two constructors:

```
EasyFTARule(RankedSymbol symbol,
  List<State> srcStates,
  State destState)
```

and

```
EasyFTARule(RankedSymbol symbol,
  State destState,
  State... states)
```

where the source states are provided in a vararg notation.

Note the different orders in the two constructors: In the first one the source states are named first and the destination state at the end, whereas in the second constructor the destination state is requested first.

#### 3.2.2.2 GenFTARule

The second possibility is to use the generic rule type `GenFTARule<F,Q>` where precise ranked symbol type, `F`, and state type, `Q` are explcitly supplied. This gives a more precise type information than `EasyFTARule` does.

Example 3.10: Creating finite tree automaton rules

```
1  RankedSymbol plus = new NamedRankedSymbol<String>("+",2);
2  RankedSymbol x = new NamedRankedSymbol<String>("x",0);
3
4  // s1 and s2 are some earlier defined states
5  RankedSymbol plus_1
6    = new StdNamedRankedSymbol<String>("+",2);
7  RankedSymbol x_1
8    = new StdNamedRankedSymbol<String>("x",0);
9  EasyFTARule r1
10   = new EasyFTARule(x_1,s1);
11 EasyFTARule r2
12   = new EasyFTARule(plus_1,s2,s1,s1);
13 List<State> src
```

```
14   = new LinkedList<State>();
15 src.add(s1);
16 src.add(s1);
17 GenFTARule<RankedSymbol,State> r3
18   = new GenFTARule<RankedSymbol,State>(plus_1,src,s2);
```

### 3.2.3 Creating Finite Tree Automata

Basically, there are two possibilities for using finite tree automata: The classes `GenFTA<F,Q>` and `EasyFTA`. `GenFTA<F,Q>` has two type parameters: The type `F` is the type of the symbols which are to be processed, and the type `Q` is the state type of the finite tree automaton represented. `EasyFTA` simply uses `RankedSymbol` and `State`. Apart from the type parameters, there are virtually no differences between `GenFTA<F,Q>` and `EasyFTA`.

We now introduce the several constructors.

- ```
  EasyFTA(Collection<RankedSymbol> newAlphabet,
    Collection<State> newStates,
    Collection<State> newFinalStates,
    Collection<? extends FTARule<RankedSymbol,State>>
      newRules)
  ```

  This constructor creates a new finite tree automaton with the given alphabet—i.e. the created finite tree automaton works on trees over this alphabet—, the given states, the final states [2] and the rules.

- ```
  EasyFTA(Collection<? extends FTARule<RankedSymbol,
    State>> rules2,
    Collection<State> finalStates2)
  ```

  For defining a finite tree automaton the alphabet is not needed, since it can be calculated from the provided rules. Nevertheless, it can be reasonable to supply the alphabet explicitly, e.g. in order to compute the complement of the automaton's language with respect to a certain alphabet[3]. The states of the finite tree automaton can also be calculated from its rules, so it is sufficient to provide the final states and the rules.

[2] the states by which the automaton annotates an accepted tree
[3] more precisely: with respect to all trees over a certain alphabet

- `EasyFTA(Collection<? extends FTARule<RankedSymbol, State>> newRules, Collection<? extends FTAEpsRule<State>> newEpsRules, Collection<State> newFinals)`

  Extending the previous constructor, also epsilon rules can be used to define a finite tree automaton. An epsilon rule has the form $q \to p$, where $q$ and $p$ are states.[4] If this constructor is used, the epsilon rules are eliminated immediately.

- `EasyFTA(FTA<RankedSymbol,State,? extends FTARule<RankedSymbol,State>> fta)`

  Copies a finite tree automaton with the right state and symbol types into a new `EasyFTA`.

- `EasyFTA(Collection<EasyFTARule> rules2, State... finalStates2)`

  To avoid supplying the final states as a list, it is possible to use the vararg constructor—this is a specialty of EasyFTA. All the other constructors can also be used in the generic form.

Example 3.11: Creating an `EasyFTA`

```
1  // let s1 and s2 be earlier defined states,
2  // r1 and r2 defined rules and
3  // plus and x some ranked symbols as above.
4  Set<EasyFTARule> rules = new HashSet<EasyFTARule>();
5  rules.add(r1); rules.add(r2);
6  Set<State> states = new HashSet<State>();
7  states.add(s1); states.add(s2);
8  Set<State> finalStates = new HashSet<State>();
9  finalStates.add(s2);
10 Set<RankedSymbol> alphabet = new HashSet<RankedSymbol>();
11 alphabet.add(plus_1); alphabet.add(x_1);
12
13 // possiblities to create EasyFTA
14 EasyFTA fta1
15   = new EasyFTA(alphabet,states,finalStates,rules);
16 EasyFTA fta3
17   = new EasyFTA(rules,finalStates);
```

[4] The intended meaning of such a rule is that a tree which is annotated with $q$ can directly be annotated with $p$, without consuming any trees. Just recall finite automata ... ☺

```
EasyFTA fta5
  = new EasyFTA(rules, s2);
EasyFTA fta4
  = new EasyFTA(fta1);
System.out.println(fta1);
```

Example 3.12: Creating a **GenFTA**

```
// let s1 and s2 be earlier defined states of the
// type NamedState<String>,
StdNamedRankedSymbol<String> plus_2
  = new StdNamedRankedSymbol<String>("+",2);
StdNamedRankedSymbol<String> x_2
  = new StdNamedRankedSymbol<String>("x",0);
GenFTARule<StdNamedRankedSymbol<String>,State> r4
  = new GenFTARule<StdNamedRankedSymbol<String>,State>(
     x_2,new LinkedList<State>(),s1);
List<State> src2 = new LinkedList<State>();
src2.add(s1); src2.add(s1);
GenFTARule<StdNamedRankedSymbol<String>,State> r5
  = new GenFTARule<StdNamedRankedSymbol<String>,State>(
     plus_2, src2, s2);

Set<GenFTARule<StdNamedRankedSymbol<String>,State>> rules2
  = new HashSet<GenFTARule<StdNamedRankedSymbol<String>,
     State>>();
rules2.add(r4); rules2.add(r5);
Set<State> states2 = new HashSet<State>();
states2.add(s1); states2.add(s2);
Set<State> finalStates2 = new HashSet<State>();
finalStates.add(s2);
Set<StdNamedRankedSymbol<String>> alphabet2
  = new HashSet<StdNamedRankedSymbol<String>>();
alphabet.add(plus_2); alphabet.add(x_2);

// possiblities to create GenFTA
GenFTA<StdNamedRankedSymbol<String>,State> ftag1
  = new GenFTA<StdNamedRankedSymbol<String>,State>(
     alphabet2,states2,finalStates2,rules2);
GenFTA<StdNamedRankedSymbol<String>,State> ftag2
  = new GenFTA<StdNamedRankedSymbol<String>,State>(
     rules2,finalStates2);
GenFTA<StdNamedRankedSymbol<String>,State> ftag3
  = new GenFTA<StdNamedRankedSymbol<String>,State>(
     ftag1);
```

### 3.2.4 Adding Epsilon Rules

Additionally to „normal“rules, epsilon rules can be added to a finite tree automaton by `addEpsilonRule(Q qsrc,Q qdest)`, where `qsrc` is the source state of the epsilon rule and `qdest` the destination state. Those rules are converted into normal rules at the next call of `getRules()`.

Example 3.13: Adding an `EasyEpsRule`

```
1  RankedSymbol h = new NamedRankedSymbol<String>("h",1);
2  RankedSymbol x = new NamedRankedSymbol<String>("x",0);
3  StateFactory stateFact = StateFactory.getStateFactory();
4  State p = stateFact.makeState("p");
5  State q = stateFact.makeState("q");
6  Collection<EasyFTARule> rules = new HashSet<FTARule>();
7  rules.add(new EasyFTARule(x, p)); // rule x->p
8  rules.add(new EasyFTARule(f, q, p)); // rule f(p)->q
9  EasyFTA fta = new EasyFTA(rules, q);
10
11 // add the epsilon rule q -> p
12 fta.addEpsilonRule(q,p);
13 // this will return the rules x->p, f(p)->q and f(p)->p
14 fta.getRules();
```

(D. Jansen, M. Mohr, I. Thesing)

## 3.3 Operations on Finite Tree Automata

Finite tree automata alone are useless—interesting are their languages and the algorithms applicable to them. Thus in the following section the possiblities are shown.

There are four classes dealing with this problems:

- `FTAProperties` where some static methods can check properties of arbitrary finite tree automata. All methods return a boolean value.
- `FTAOps` provides a very general and configurable implementation of the operations. [5]
- `GenFTAOps` is a more specialized version, where you get `GenFTA<F,Q>`s as results.
- `EasyFTAOps` is the version which with minimal configuration effort, where you get `EasyFTA<F,Q>`s as results.

[5] This is where the action is - but it may be a bit too much action for most use cases...

### 3.3.1 Decide

The most important operation of a tree automaton is to decide whether a given tree is accepted by a a given finite tree automaton. You can find this functionality in `FTAProperties` or directly in your finite tree automaton implementation.

Example 3.14: decide

```
// let fta1 be an automaton over trees of arithmetic
   expressions, which recognizes all trees containing no y.
Tree<RankedSymbol> tree_1 = null;
Tree<RankedSymbol> tree_2 = null;
try {
  tree_1 = TreeParser.parseString("+(*(x,y),y)");
  tree_2 = TreeParser.parseString("+(x,x)");
} catch (ParseException e) {
  e.printStackTrace();
}

FTAProperties.decide(fta1,tree_1); // should yield false
fta1.decide(tree_2);               // should yield true
```

### 3.3.2 Properties of a Finite Tree Automaton

#### 3.3.2.1 Determinstic

A finite tree automaton is called ***deterministic***, if every left hand side of a rule occurs at most once in the rules of the finite tree automaton.

Lethal can check whether a given tree automaton is deterministic using the method `checkDeterministic` in `FTAProperties` or it can construct an equivalent deterministic finite tree automaton with `determinize`. Since a subset construction is used for determination, this could lead to an exponential blow-up.[6]

Example 3.15: determinize

```
EasyFTA fta1 = ...;
if (!FTAProperties.checkDeterministic(fta1)){
  EasyFTA detfta1 = EasyFTAOps.determinize(fta1);
}

GenFTA<NamedRankedSymbol<String>,State> ftag2 = ...;
if (!FTAProperties.checkDeterministic(ftag2)){
  GenFTA<StdNamedRankedSymbol<String>,
         NamedState<Set<State>>> detfta2
```

[6] for short: It could take some time.

```
10    = GenFTAOps.determinize(ftag2);
11 }
```

#### 3.3.2.2 Complete

A finite tree automaton is called ***complete*** if every combination of symbols in the alphabet and source states occur on a left side of a rule. (In mathematical notation: $\forall f \in \text{alphabet}\, \forall q_1, \ldots, q_{\text{arity}(f)} \in \text{states}\, \exists q \in \text{states}\, \exists r \in \text{rules}\, .r = f(q_1, \ldots, q_n) \to q.$)

In order to check whether a given finite tree automaton is complete, it is possible to use the method `FTAProperties.checkComplete()` for an arbitrary finite tree automaton. To complete a given `EasyFTA` use `EasyFTAOps.complete`, to complete a `GenFTA<F,Q>` use `GenFTAOps.complete`.

Example 3.16: complete

```
1 if (!FTAProperties.checkComplete(fta1)){
2   fta1 = EasyFTAOps.complete(fta1);
3 }
4 GenFTA<StdNamedRankedSymbol<String>,State> detftag2 =
     GenFTAOps.complete(ftag1, new NamedState<String>("qbot"));
5 FTAProperties.checkComplete(detftag2);
```

#### 3.3.2.3 Reduced

A finite tree automaton is called ***(bottom-up) reduced***, if for each state $q$, there is at least one tree $t$, whose root symbol can be annotated with $q$. This means that all states are reachable. The method `reduceBottomUp()` reduces a finite tree automaton.

A finite tree automaton is called ***top-down reduced*** if it only contains states from which a final state is reachable. To reduce a finite tree automaton there is the method `reduceTopDown()` in the corresponding classes.

A finite tree automaton is called ***fully reduced*** if it is both top-down and bottom-up reduced. So for each rule of the fully reduced automaton there is a tree $t$ and a final state $q_f$, such that $t$ can be annotated with $q_f$ by this rule.

This is done by the method `reduceFull` in the corresponding classes.

Example 3.17: reduce

```
1 EasyFTA ftar = EasyFTAOps.reduceBottomUp(fta1);
2 EasyFTA ftar2 = EasyFTAOps.reduceTopDown(ftar);
3 EasyFTA ftar3 = EasyFTAOps.reduceFull(fta1);
4
5 GenFTA<StdNamedRankedSymbol<String>,State> ftar4
6   = GenFTAOps.reduceBottomUp(ftag2);
```

```
7  GenFTA<StdNamedRankedSymbol<String>,State> ftar5
8    = GenFTAOps.reduceTopDown(ftar4);
9  GenFTA<StdNamedRankedSymbol<String>,State> ftar6
10   = GenFTAOps.reduceFull(ftag2);
```

#### 3.3.2.4 Minimized

For each deterministic and complete finite tree automaton, there exists an equivalent finite tree automaton which has a minimal number of states. In order to minimize a finite tree automaton, it is possible to use the method `minimize()`. Note that the number of states of the resulting automaton does not have to be minimal. If the given automaton is not complete, it is completed first und thus enriched by one state—so, the resulting automaton could have one state more than absolutely necessary. Furthermore, for a finite tree automaton which is not deterministic, one should not attempt to minimize—the algorithm Lethal uses does not work for non-deterministic automata. Thus, an exception is thrown.

Example 3.18: minimize

```
1  EasyFTA fta1 = ...;
2  EasyFTA fta1m = EasyFTAOps.minimize(fta1);
3
4  GenFTA<NamedRankedSymbol<String>,State>  ftag2 = ...;
5  GenFTA<StdNamedRankedSymbol<String>,
6         NamedState<Set<State>>> fta2m =
7    GenFTAOps.minimize(ftag2, new NamedState<String>("qnew"));
```

### 3.3.3 Properties of the Language

In the following we shall describe the methods checking properites of the regular tree languages represented by the finite tree automata. All the methods are found in `FTAProperties`.

| method | description |
|---|---|
| `emptyLanguage()` | Checks whether a given finite tree automaton has an empty language, i.e. whether it does not accept any tree. Note that a finite tree automaton with empty language does not have to be empty in the sense of having neither states nor symbols nor rules. |

| | |
|---|---|
| `subsetLanguage()` | Takes two finite tree automata and checks whether the language of the first automaton is a subset of the second one. If it yields true, every tree recognized by the first automaton is also recognized by the second one. |
| `sameLanguage()` | Checks whether two finite tree automata recognize the same regular tree language. They are called ***equivalent*** then. |
| `finiteLanguage()` | Checks whether the language of a given finite tree automaton recognizes only finitely many trees. |

Example 3.19: Properties of finite tree automaton languages

```
1  EasyFTA fta1 = ...;
2  EasyFTA fta2 = ...;
3  FTAProperties.subsetLanguage(fta1,fta2);
4  FTAProperties.sameLanguage(fta1,fta2);
5
6  GenFTA<StdNamedRankedSymbol<String>,State> ftag3 = ...;
7  if (!FTAProperties.emptyLanguage(fta3)){
8    if (FTAProperties.finiteLanguage(fta3)){
9      System.out.println("The regular tree language is finite,
           but not emtpy.");
10   }
11 }
```

### 3.3.4 Operations on the Languages

Some more algorithms provide set operations on regular tree languages represented by finite tree automata. As for `EasyFTA`, all these operations can be found in `EasyFTAOps`. As far as `GenFTA<F,Q>` is concerned they are contained in `GenFTAOps`.

#### 3.3.4.1 Complement

The ***complement*** of a regular tree language $L$ is the set of all trees over the same alphabet which are not contained in $L$. If $L$ is described by a finite tree automaton $\mathcal{A}$, the complement of $L$ is described by a finite tree automaton which accepts a tree if and only if $\mathcal{A}$ rejects it. The method `complement()` constructs a finite tree automaton which recognizes the complement of the language of the given automaton $\mathcal{A}$. Thus, the complement of the language of $\mathcal{A}$ with respect to the alphabet of $\mathcal{A}$ is given as

output. The method `complementAlphabet()` also constructs a complementary finite tree automaton, yet with respect to an alphabet which contains the given symbols and the symbols of $\mathcal{A}$.

Example 3.20: complement

```
1 EasyFTA fta1 = ...;
2 EasyFTA fta1c = EasyFTAOps.complement(fta1);
3 Set<RankedSymbol> alphabet6 = new HashSet<RankedSymbol>();
4 alphabet6.add(plus_1);alphabet6.add(x_1);alphabet6.add(y_1);
5 EasyFTA fta1ca
6   = EasyFTAOps.complementAlphabet(fta1,alphabet6);
```

#### 3.3.4.2 Union

The method `union()` constructs a finite tree automaton which accepts the union of the languages of both given finite tree automata. More precisely, the resulting automaton accepts a tree if and only if one or both of the given automata accepts it.

#### 3.3.4.3 Intersection

The methods `intersectionWR()`, `intersectionTD()` and `intersectionBU()` construct a finite tree automaton which accepts the intersection of the languages of both given finite tree automata. More precisely, the resulting automaton accepts a tree if and only if both the given automata accept it. The algorithm uses a product construction. Since such a product construction can be very expensive[7], there are three different versions—one without any reduction (`intersectionWR()`), and the other two with additional bottom-up or top-down reductions.

#### 3.3.4.4 Difference

The method `difference(FTA fta1, FTA fta2)` constructs a finite tree automaton which accepts the difference of the languages of both given finite tree automata. This means that the resulting automaton accepts every tree which is accepted by the first finite tree automaton, but not by the second one.

Example 3.21: Several operations on languages

```
1 EasyFTA fta1 = ...;
2 EasyFTA fta2 = ...;
3 EasyFTA ftaUnion = EasyFTAOps.union(fta1,fta2);
4 EasyFTA ftaIntersection
```

[7] with respect to both time complexity and the size of the resulting automaton

```
5   = EasyFTAOps.intersectionTD(fta1,fta2);
6 EasyFTA ftaDifference = EasyFTAOps.difference(fta1,fta2);
```

### 3.3.5 Further Operations

Apart from the operations presented so far, there are some more operations concerning regular tree languages and finite tree automata.

#### 3.3.5.1 Construct Tree Witness

Given a finite tree automaton with a non-empty language, a witness tree which is contained in the language can be created by the method `constructTreeFrom(FTA fta)`. If the language of the given automaton is empty, `null` is returned. Furthermore, you can specifiy a minimal height and use `constructTreeWithMinHeightFrom()`, which constructs a tree contained in the given language being at least as high as specified.

#### 3.3.5.2 Construct Special Finite Tree Automata

There are two special finite tree automata Lethal can construct:

The method `computeSingletonFTA()` computes a finite tree automaton which accepts exactly the given tree and no other one. On the other hand, a finite tree automaton which recognizes all the trees over a given alphabet can be obtained by using the method `computeAlphabetFTA()`.

Example 3.22: Constructing A Witness and A Very Tolerant Finite Tree Automaton

```
1 EasyFTA fta1 = ...;
2 Tree<RankedSymbol> witness = EasyFTAOps.constructTreeFrom(
    fta1);
3 EasyFTA ftaSinglet = EasyFTAOps.computeSingletonFTA(witness);
4
5 Set<StdNamedRankedSymbol<String>> alphabet7 = new HashSet<
    StdNamedRankedSymbol<String>>();
6 alphabet7.add(plus_2);alphabet7.add(x_2);alphabet7.add(y_2);
7 GenFTA<StdNamedRankedSymbol<String>,State> ftaAlphg =
    GenFTAOps.computeAlphabetFTA(alphabet7);
```

#### 3.3.5.3 Restrict Trees To A Given Height

The method `restrictToMaxHeight()` constructs the regular tree language containing all trees of this regular tree language which do not exceed the specified height.

#### 3.3.5.4 Substitute Languages Into A Tree

The class of regular tree languages is not only closed under the ordinary set operations, but also under an operation called ***substitution***. Given a tree with $n$ distinguished leaves $l_1, \ldots, l_n$ and $n$ regular tree languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$, the language of all trees which are obtained by substituting every $l_i$ by a tree from $\mathcal{L}_i$ is also a regular tree language. Lethal provides such a substitution by supplying the tree and a mapping. This mapping relates the distinguished leaves to the regular tree languages—i.e. the automata describing them—which the leaves shall be substituted by.

Example 3.23: Substitution

```
// Let F extend StdNamedRankedSymbol<String> and Q extend
// NamedState<String>. Both shall be equipped with
// appropriate constructors. Let list() be a method which
// takes arbitrarily many objects of a certain type
// and returns a list of these objects.
F x_3 = new F("x",0);
F y_3 = new F("y",0);
F plus_3 = new F("+",2);

// fta1 accepts everything containing no x or y, i.e. nothing
GenFTA<F,Q> fta1s = new GenFTA<F,Q>();
// fta2 accepts every tree containg no y
Q stat = new Q("string");
Set<Q> finalStates4 = new HashSet<Q>();
finalStates4.add(stat);
Set<GenFTARule<F,Q>> rules4 = new HashSet<GenFTARule<F,Q>>();
rules4.add(new GenFTARule<F,Q>(
  y_3,new LinkedList<Q>(), stat));
rules4.add(new GenFTARule<F,Q>(
  plus_3, Util.makeList(stat,stat), stat));
GenFTA<F,Q> fta2s = new GenFTA<F,Q>(rules4,finalStates4);

Map<F,GenFTA<F,Q>> languages = new HashMap<F,GenFTA<F,Q>>();
languages.put(x_3,fta1s);
languages.put(y_3,fta2s);

Tree<F> t = new StdTree<F>(plus_3,
  Util.makeList(new StdTree<F>(x_3),new StdTree<F>(y_3)));

GenFTA<F, NamedState<?>> ftasubs
  = GenFTAOps.substitute(t,languages);
```

(D. Jansen, M. Mohr, I. Thesing)

## 3.4 Extended Operations on Finite Tree Automata

We already dealt with implementing user-defined symbols at the beginning of the last chapter. Here we explain how to extend states and finite tree automata. Furthermore it is explained how to control the resulting finite tree automata of operations. In particaluar we regard the state types of the resulting finite tree automata, because they are not flat if the methods of `GenFTAOps` are used. That means, that the types of the states in the resulting finite tree automata are not the same as in the input finite tree automata.

### 3.4.1 Basic Interfaces

According to the definition, a finite tree automaton consists of an alphabet, a set of states, a set of final states and a set of rules. The final states are a subset of the set of states and each symbol of the alphabet has a non-negative arity. So it is a `RankedSymbol`. The rules have the form $f(q_0, ..., q_{n-1}) \to q$, where $f$ is a symbol and the $q_0, \ldots, q_{n-1}, q$ are states. This is implemented by the interface `FTA<F,Q,R>`. The user now can extend the functionalities of a finite tree automaton by implementing this interface. `F` is used for a type of ranked symbol, `Q` is the type of states used in the finite tree automaton and `R` is the type of rules extending `FTARule<F,Q>`, which can be implemented with additional functionalities, too.

An interface which also allows to change the finite tree automaton on the run is `ModifiableFTA` with methods that can add rules, make states to final states or remove rules or states.

#### 3.4.1.1 Abstract and Standard Implementations

If one uses modified finite tree automata, yet does not want to re-implement all the basic functionalities of a finite tree automaton, Lethal provides an abstract implementation with basic functionalities, `AbstractFTA<F,Q,R>`, and a modifable version of that one, `AbstractModFTA<F,Q,R>`. For the modifiable versions it is necessary to implement the abstract method for creating a new rule.

For using already implemented rules, the easiest way is to use the standard generic implementations, `GenFTARule<F,Q>` and `GenFTAEpsRule<Q>`.

#### 3.4.1.2 FTACreator

A `FTACreator<F,Q,R,T>` encapsulates some functionalities needed for creating a finite tree automaton of type `T`, where `T` extends `FTA<F,Q,R>` with states of type `Q`, ranked symbols of type `F` and rules of type `R` extending `FTARule<F,Q>`:

- `createRule(F f,List<Q> src,Q dest)`

  which creates a rule of type `R`.

- `createFTA(Collection<? extends FTARule<F,Q>> newRules, Collection<Q> newFinals)`

  which creates a new finite tree automaton out of a collection of rules and a collection of final states.

- `createFTA(Collection<F> alphabet,Collection<Q> states, Collection<Q> finalStates,Collection<? extends FTARule<F,Q>> rules)`

  which creates a finite tree automaton out of the given characteristic data.

- `createFTA(Collection<F> alphabet,Collection<Q> states, Collection<Q> finalStates,Collection<? extends FTARule<F,Q>> rules,Collection<? extends FTAEpsRule <Q>> epsRules)`

  which creates a new finite tree automaton without epsilon rules out of the characteristic data of a finite tree automaton with epsilon rules. That means, it constructs a finite tree automaton by eliminating the epsilon rules.

- `eliminateEpsilonRules(Collection<? extends FTARule<F,Q >> rules, Collection<? extends FTAEpsRule<Q>> epsRules)`

  which eleminates the given epsilon rules by creating corresponding rules.

- `makeFTAFromGrammar(java.util.Collection<Q> grammarStart, java.util.Collection<? extends RTGRule<F,Q>> grammarRules, Converter<java.lang. Object,Q0> stateBuilder)`

  which, given a regular tree grammar, constructs an equivalent finite tree automaton (i.e. the resulting finite tree automaton recognizes a tree if and only if it is generated by the given tree grammar).

This class and the abstract methods must be implemented for applying operations on the user-defined finite tree automata. An example for this is the implementation `GenFTACreator<F,Q>`, which creates rules of the type `GenFTARule` and finite tree automata of the type `GenFTA`.

### 3.4.2 Controlling Results of Finite Tree Automata Operations

By means of the class `FTAOps`, operations can be applied on arbitrary finite tree automata implementing `FTA`. However, it is necessary to specify, how the new states and the resulting finite tree automaton are to be produced. To control the type of the finite tree automaton, it is necessary to supply a `FTACreator` (see above) (s. 3.4.1.2). For creating new states, it is necessary to suppy a single state or a converter, which converts some data into a new state of the wished type.

Each method in `FTAOps` expects some of these parameters. We want to explain them in an example, where we implement a new state type which saves the architecture of states in a string and enumerates them. Then we use the generic finite tree automata version, `GenFTAOps`, and create a mask for operations on finite tree automata of the type `GenFTA<F,SpecialState>`.

Example 3.24: Extension – Special States

```
1  public class StateCounter{
2    protected static int count = 0;
3
4    public static increment(){
5     count++;
6    }
7
8    public int getNr(){
9      increment();
10     return count-1;
11   }
12 }
13
14 public class SpecialState implements State{
15   protected int nr;
16   protected String arch;
17
18   public SpecialState(){
19     nr = StateCounter.getNr();
20     arch = Integer.toString(nr);
21   }
22
23   public SpecialState(String str){
24     nr = StateCounter.getNr();
25     arch = str;
26   }
27
28   public int hashCode() {
29    final int prime = 31;
```

```
30     int result = 1;
31     result = prime * result + nr;
32     return result;
33   }
34
35   public boolean equals(Object obj) {
36     if (this == obj)
37       return true;
38     if (obj == null)
39       return false;
40     if (getClass() != obj.getClass())
41       return false;
42     final SpecialState other = (SpecialState) obj;
43     if (nr != other.nr)
44       return false;
45     return true;
46   }
47 }
```

### 3.4.3 Properties of a Finite Tree Automaton

For an explanation of the functionalities of the methods regarding properties of a finite tree automaton consider the corresponding methods in 3.3.2. Here we merely explain what additionally must be supplied to apply the methods.

For example, the method

```
determinize(FTA<F,Q,? extends FTARule<F,Q>> A, FTACreator<
    F,Q0,R0,U> fc, Converter<Set<Q>,Q0> sc)
```

has the resulting type `U extends FTA<F,Q0,R0>>`.

It works with a subset construction, thus beside the finite tree automaton it expects a `Converter`, which converts a set of states into a new state and an `FTACreator` for creating the result.

Example 3.25: Extension – Determinize

```
1 public class ExampleOps
2
3   private class SetStateConverter
4       implements Converter<Set<SpecialState>,SpecialState>{
5     public SpecialState convert(Set<SpecialState> set){
6       String arch = set.toString();
7       return new SpecialState(arch);
8     }
9   }
```

```
10
11   public <F extends RankedSymbol> GenFTA<F,SpecialState>
        determinize(GenFTA<F,SpecialState> fta){
12     return FTAOps.determinize(fta,
13       new GenFTACreator<F,SpecialState>(),
14       new SetStateConverter());
15   }
16 }
```

The application of the following methods is similar:

- `complete(FTA<F,Q,? extends FTARule<F,Q>> A,Q qbot,`
  `FTACreator<F,Q,R,T> fc)`

  `reduceFull(FTA<F,Q,? extends FTARule<F,Q>> fta,`
  `FTACreator<F,Q,R,T> fc)`

- `minimize(FTA<F,Q,? extends FTARule<F,Q>> fta,Q qbot,`
  `FTACreator<F,Q,R,T> fc,Converter<Set<Q>,Q0> sc0,`
  `FTACreator<F,Q0,R0,U> fc0)`

Example 3.26: Extension – Other FTA Properties

```
1  public class ExampleOps
2    ...
3
4    public <F extends RankedSymbol> GenFTA<F,SpecialState>
        complete(GenFTA<F,SpecialState> fta){
5      return FTAOps.complete(fta,
6        new SpecialState(),
7        new GenFTACreator<F,SpecialState>());
8    }
9
10   public <F extends RankedSymbol> GenFTA<F,SpecialState>
        reduceFull(GenFTA<F,SpecialState> fta){
11     return FTAOps.reduceFull(fta,
12       new GenFTACreator<F,SpecialState>());
13   }
14
15   public <F extends RankedSymbol> GenFTA<F,SpecialState>
        minimize(GenFTA<F,SpecialState> fta){
16     GenFTACreator<F,SpecialState> fc = new GenFTACreator<F,
          SpecialState>();
```

```
17      return FTAOps.minimize(fta,
18        new SpecialState(),
19        fc,
20        new SetStateConverter(),
21        fc);
22    }
23
24  }
```

#### 3.4.3.1 Operations on the Languages

For an explanation of the functionalities of the methods regarding operations on the languages of finite tree automata consider the corresponding methods in 3.3.4. Here we only explain the ingredients which are neccessary to apply the methods.

For example, in order to intersect two finite tree automata, a product automaton is constructed. So Lethal needs a `Converter`, which converts a pair of states into a new one. The signature of the method is

```
intersectionTD(
  FTA<F,Q1,? extends FTARule<F,Q1>> automaton1,
  FTA<F,Q2, ? extends FTARule<F,Q2>> automaton2,
  Converter<Pair<Q1,Q2>,Q3> pc,
  FTACreator<F,Q3,R,T> fc)
```

To construct the union of two finite tree automata, the states must be embedded injectively into a third state type. This is done by means of two state converters. Since the result automaton has a definite alphabet, the symbols must be converted, too. The signature of this method is

```
union(FTA<F1,Q1,? extends FTARule<F1,Q1>> automaton1,
  FTA<F2,Q2,? extends FTARule<F2,Q2>> automaton2,
  Converter<Q1,Q3> c13,
  Converter<Q2,Q3> c23,
  Converter<F1,F3> smb13,
  Converter<F2,F3> smb23,
  FTACreator<F3,Q3,R,T> fc)
```

The operations

```
complement(FTA<F,Q,? extends FTARule<F,Q>> automaton,
  Converter<Set<Q>,Q0> sc,
  FTACreator<F,Q0,R0,T0> fc0)
```

and

```
difference(FTA<F,Q1,? extends FTARule<F,Q1>> fta1,
  FTA<F,Q2,? extends FTARule<F,Q2>> fta2,
  Converter<Set<Q2>,Q20> sc2,
  FTACreator<F,Q20,R20,T20> fc20,
  Converter<Pair<Q1,Q20>,Q3> pc,
  FTACreator<F,Q3,R3,T3> fc3)
```

work similar.

In the example we expect to have only one generic type, the symbol type `F`.

Example 3.27: Extension – Language Operations

```
1  public class ExampleOps{
2
3    private class PairStateConverter implements
4       Converter<Pair<SpecialState,SpecialState>,SpecialState>{
5      public SpecialState convert(Pair<SpecialState,
           SpecialState> pair){
6        String arch = pair.toString();
7        return new SpecialState(arch);
8      }
9    }
10
11   private class FirstStateConverter implements
12      Converter<SpecialState,SpecialState>{
13     public SpecialState convert(SpecialState state){
14       String arch = "(a," + state.toString() + ")";
15       return new SpecialState(arch);
16     }
17   }
18
19  private class SecondStateConverter implements
20      Converter<SpecialState,SpecialState>{
21     public SpecialState convert(SpecialState state){
22       String arch = "(b," + state.toString() + ")";
23       return new SpecialState(arch);
24     }
25   }
26
27   private class SymbolConverter<F> implements Converter<F,F>{
28     public F convert(F f){
29       return f;
30     }
31   }
32
```

```
33    public <F extends RankedSymbol> GenFTA<F,SpecialState>
34       intersection(GenFTA<F,SpecialState> fta1,
35         GenFTA<F,SpecialState> fta2){
36      return FTAOps.intersectionTD(fta1,fta2,new
           PairStateConverter(), new GenFTACreator<F,SpecialState
           >());
37    }
38
39    public <F extends RankedSymbol> GenFTA<F,SpecialState>
40       union(GenFTA<F,SpecialState> fta1,
41         GenFTA<F,SpecialState> fta2){
42      SymbolConverter<F> sc = new SymbolConverter<F>();
43      return FTAOps.union(fta1,
44        fta2,
45        new FirstStateConverter(),
46        new SecondStateConverter(),
47        sc,
48        sc,
49        new GenFTACreator<F,SpecialState>());
50    }
51
52    public <F extends RankedSymbol> GenFTA<F,SpecialState>
53       complement(GenFTA<F,SpecialState> fta){
54      return FTAOps.complement(fta,
55        new SetStateConverter(),
56        new GenFTACreator<F,SpecialState>());
57    }
58
59    public <F extends RankedSymbol> GenFTA<F,SpecialState>
60       difference(GenFTA<F,SpecialState> fta1,
61         GenFTA<F,SpecialState> fta2){
62      GenFTACreator<F,SpecialState> fc =
63        new GenFTACreator<F,SpecialState>();
64      return FTAOps.difference(fta1,
65        fta2,
66        new SetStateConverter(),
67        fc,
68        new PairStateConverter(),
69        fc);
70    }
71
72  }
```

#### 3.4.3.2 Further Operations

Since the usage of `FTAOps` was explained in the last two subsections and the example above, we only list the signature of some more operations described in 3.3.5.

- ```
  computeSingletonFTA(Tree<F0> t,
    FTACreator<F,Q,R,U> fc,
    Converter<Object,Q> stateBuilder)
  ```

  ```
  computeAlphabetFTA(Collection<F> alphabet,
    Q qdest,
    FTACreator<F,Q,? extends FTARule<F,Q>,? extends T>
       fc)
  ```

- ```
  substitute(Tree<F> tree,
    Map<? extends F,? extends FTA<F,Q,? extends FTARule<
       F,Q>>> languages,
    Converter<? super Pair<Q,Integer>,Q0> intStateConv,
    Converter<? super Integer,Q0> intConv,
    Converter<? super Tree<F>,Q0> treeStateConv,
    FTACreator<F,Q0,R0,T> fc)
  ```

Only in the method

```
constructTreeFrom(FTA<F,Q,? extends FTARule<F,Q>> fta,
  TreeCreator<F,T> tc)
```

Lethal needs another special type, the `TreeCreator<S,T>`, to create the witness tree.

**TreeCreator** The `TreeCreator<S,T>` is an interface with two methods for creating trees of type `T` using symbols of type `S`. The method `T makeTree(S symbol)` creates a tree out of a single symbol, whereas `T makeTree(S symbol, List<T> subTrees)` creates a tree having subtrees.

A standard implementation is the class `StdTreeCreator`, which works with a `TreeFactory`. It can be used in our example.

(D. Jansen, M. Mohr, I. Thesing)

## 3.5 Tree Grammars

A regular tree grammar consists of a set of start symbols and a set of regular tree grammar rules. In particular, the rules have the form $A \to t$, where $A$ is a special kind of state, called ***non-terminal***, and $t$ is a configuration tree. A regular tree grammar generates a tree by—starting with a start symbol—replacing a non-terminal by a configuration tree until no non-terminal symbols are left. A non-terminal can be replaced by a configuration tree if there is a grammar rule having this non-terminal as left side and the configuration tree as right side. The thereby resulting set of trees forms a regular tree language.

### 3.5.1 Grammar Rules

A regular tree grammar rule consists of a non-terminal and a configuration tree. This is given by the interface `RTGRule<F,Q>`, where $F$ is of type `RankedSymbol` and `Q` of type `State`. The interface provides the methods `getLeftSide()`, which returns the non-terminal of type `Q` and `getRightSide()`, which returns the configuration tree of the rule having type `Tree<BiSymbol<F,Q>>`.

To instantiate a grammar rule, the class `GenRTGRule` can be used. It stores the part of a regular tree grammar rule within the attributes `left` and `right` and can be created by the constructor

```
GenRTGRule(Q left, Tree<BiSymbol<F,Q>> right)
```

which works in the obvious way.

### 3.5.2 Regular Tree Grammars

For working with a regular tree language there is the interface `RTG<F,Q>` where `F` is a type of `RankedSymbol` and `Q` of type `State`. A regular tree grammar can be instantiated by using a constructor of the class `GenRTG<F,Q>`. There are two possibilities:

```
public GenRTG(Collection<Q> start, Collection<? extends
   RTGRule<F,Q>> rules)
```

which needs a collection of the start symbols and the rules of a grammar.

```
public GenRTG(Collection<Q> start)
```

which creates a grammar without any rules with the given start symbols.

Besides the usual get-methods, this class provides the method `addRule(Q left, Tree<BiSymbol<F,Q>> right)`, which creates a `GenRTGRule` out of the given data and adds it to the rules, and `addStartSymbol(Q symbol)`, which adds the given symbol to the start symbols. Additionally, the class `GenRTG` implements the `FTA` interface. Thus there exist also get-methods for the characteristic data of a finite tree automaton.

### 3.5.3 Operations

As the class `GenRTG` implements the `FTA` interface, all operations for finite tree automata of regular tree languages can also be done with a regular tree grammar implementing `GenRTG`. Furthermore the class `RTGOps` provides the method

```
makeGrammarFromFTA(FTA<F,Q,? extends FTARule<F,Q>> fta
```

which creates a regular tree grammar of type `RTG<F,Q>` that accepts the same language as the given finite tree automaton.

(D. Jansen, M. Mohr, I. Thesing)

## 3.6 Regular Tree Languages

Finite tree automata represent regular tree languages. The class `RegularTreeLanguage` also provides a view on a regular tree language, but hides the finite tree automaton representing it. The focus lies on the language itself.

The class provides the language-specific algorithms on finite tree automata. It is possible to construct a `RegularTreeLanguage` out of a finite tree automaton, which is then converted into an internal, more efficient representation, where the states are essentially numbers. So the fundamental properties of the finite tree automaton are conserved, but the finite tree automaton itself is not.

There are several constructors:

- `RegularTreeLanguage(Collection<F> alphabet)`

  creates a new regular tree language, which consists of all trees over a given alphabet.

- `RegularTreeLanguage(FTA<F,Q,? extends FTARule<F,Q>> fta)`

  creates a new regular tree language out of an arbitrary finite tree automaton.

- `RegularTreeLanguage()`

  creates a new empty tree language .

- `RegularTreeLanguage(Collection<Q> grammarStart, Collection<? extends RTGRule<F,Q>> grammarRules)`

  creates a new regular tree language out of the components of a regular tree grammar.

Since all algorithms have already been explained, we simply list the methods provided by `RegularTreeLanguage`:

**addTree** `addTree(Tree<F> t)`

adds a tree to this language (this essentially unites the language with the singleton language accepting only this tree).

**addTrees** `addTrees(RegularTreeLanguage<F> lang)`

adds a whole regular tree language to this language.

**applyHom** `<G extends RankedSymbol> RegularTreeLanguage<G> applyHom(Hom<F,G,? extends Variable> h)`

applies a linear tree homomorphism to this regular tree language.

**applyHomInv** `<E extends RankedSymbol> RegularTreeLanguage<E> applyHomInv(Hom<E,F,? extends Variable> h)`

given a tree homomorphism h, computes the preimage of this regular tree language under h, i.e. all trees t with h(t) in L.

**complement** `RegularTreeLanguage<F> complement()`

returns the complement of this regular tree language.

**constructWitness** `Tree<F> constructWitness()`

returns a tree contained in this language, it is not empty.

**contains** `boolean contains(Tree<F> t)`

decides whether a given tree is contained in this regular tree language.

**getFTA** `GenFTA<F,? extends State> getFTA()`

returns the underlying finite tree automaton.

**isEmpty** `boolean isEmpty()`

returns whether this language is empty.

**isFinite** `boolean isFinite()`

returns whether this language is finite.

**removeTree** `void removeTree(Tree<F> t)`

removes a tree from this language, if it is contained.

**removeTrees**
`void removeTrees(RegularTreeLanguage<F> lang)`

removes a whole regular tree language from this language.

**restrictToMaxHeight** `RegularTreeLanguage<F>`
`restrictToMaxHeight(int maxHeight)`

constructs the regular tree language containing all trees which are contained in this regular tree language and which are not higher than specified.

**retainAllTrees**
`void retainAllTrees(RegularTreeLanguage<F> lang)`

intersects this language with a given regular tree language.

**sameAs** `boolean sameAs(RegularTreeLanguage<F> lang)`

checks whether this regular tree language is the same as the given one.

**subsetOf** `boolean subsetOf(RegularTreeLanguage<F> lang)`

checks whether this regular tree language is a subset of the given one.

**substitute** `static <F extends RankedSymbol,G extends F>`
`RegularTreeLanguage<F> substitute(Tree<F> tree, Map`
`<G,RegularTreeLanguage<F>> languages)`

returns the regular tree language obtained by substituting several regular tree languages into a given tree.

Further it is possible to iterate through a language, if this language is finite, as `RegularTreeLanguage` implements `Iterable`. This may take some time.

(D. Jansen, M. Mohr, I. Thesing)

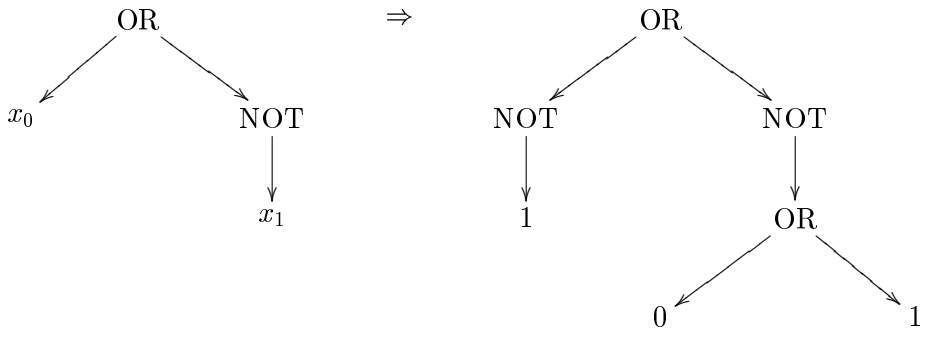
## 3.7 Tree Homomorphisms

Homomorphisms on tree languages transform the trees into new trees over the same or another alphabet.

A homomorphism on trees $h : T(\mathcal{F}) \to T(\mathcal{G})$ is given by a mapping $\text{hom} : F \to T(\mathcal{G}, \mathcal{X})$, where the image of a symbol $f$ with arity $n$ contains at most $n$ different variables out of $x_0, \ldots, x_{n-1}$. The homomorphism $h$ is then defined inductively:

- $h(a) := \text{hom}(a)$ for all constants $a$ in $\mathcal{F}$,
- $h(f(t_0, \to, t_{n-1})) := \text{hom}(f)\{x_0 \leftarrow h(t_0), \ldots, x_{n-1} \leftarrow h(t_{n-1})\}$, i.e. every variable is replaced by the image of the corresponding subtree under the homomorphism.

Example of replacing variables $x_0$ and $x_1$ by trees $not(1)$ and $or(0, 1)$:




### 3.7.1 Defining a Tree Homomorphism

Again there are two different ways of defining a homomorphism. `GenHom<F,G>` works with generic type parameters for the symbols of both alphabets. They must be subtypes of `RankedSymbol`. For an easier usage, `EasyHom` works with `RankedSymbol`s and has no type parameters. In the following sections, we concentrate on the functionality of `GenHom<F,G>`—the usage of `EasyHom` is mostly the same.

#### 3.7.1.1 Creating a Homomorphism Using Variables

The first possibility to create a homomorphism is the constructor `GenHom(Map<F, Tree<? extends BiSymbol<G,Variable>>> h)`.

As seen in the definition, each symbol $f \in F$ is mapped to a tree containing variables with numbers between 0 and $\text{arity}(f) - 1$. Thus there is a special symbol type, `Variable`, which stores this number. To produce a new variable, use `StdVariable( int nr)`.

In order to produce the variable trees, the methods for creating trees with `BiSymbol` s can be used, where the leaf type is `Variable` and the inner type is `G`, i.e. the symbol type of the destination alphabet. See at chapter 3.1.1.4 for more information on the different symbols.

In `EasyHom`, an additional alphabet can be provided. The homomorphism is then extended by the identity function on all the symbols for which no image is defined.

Consider the following example, where boolean formulas containing the junctors AND and NOT are transformed into equivalent formulas containing OR and NOT.

Example 3.28: Defining a homomorphism

```
// creating the used symbols
RankedSymbol t1 = new StdNamedRankedSymbol<String>("1",0);
RankedSymbol f = new StdNamedRankedSymbol<String>("0",0);
RankedSymbol not = new StdNamedRankedSymbol<String>("not",1);
RankedSymbol and = new StdNamedRankedSymbol<String>("and",2);
RankedSymbol or = new StdNamedRankedSymbol<String>("or",2);

// creating used variables
Variable v0 = new StdVariable(0);
Variable v1 = new StdVariable(0);

// as BiSymbols
BiSymbol<RankedSymbol,Variable> bt1
  = new InnerSymbol<RankedSymbol,Variable>(t1);
BiSymbol<RankedSymbol,Variable> bf
  = new InnerSymbol<RankedSymbol,Variable>(f);
BiSymbol<RankedSymbol,Variable> bnot
  = new InnerSymbol<RankedSymbol,Variable>(not);
BiSymbol<RankedSymbol,Variable> band
  = new InnerSymbol<RankedSymbol,Variable>(and);
BiSymbol<RankedSymbol,Variable> bor
  = new InnerSymbol<RankedSymbol,Variable>(or);
BiSymbol<RankedSymbol,Variable> bv0
  = new LeafSymbol<RankedSymbol,Variable>(v0);
BiSymbol<RankedSymbol,Variable> bv1
  = new LeafSymbol<RankedSymbol,Variable>(v1);

// for creating trees
TreeFactory tf = TreeFactory.getTreeFactory();
Tree<BiSymbol<RankedSymbol,Variable>> vtree =
    tf.makeTreeFromSymbol(bnot,
  Util.makeList(tf.makeTreeFromSymbol(bor,
    Util.makeList(tf.makeTreeFromSymbol(bnot,
      Util.makeList(tf.makeTreeFromSymbol(bv0))),
```

```
35        tf.makeTreeFromSymbol(bnot,
36          Util.makeList(tf.makeTreeFromSymbol(bv1)))))));
37
38 // build up, possibility 1
39 Map<RankedSymbol,
40   Tree<? extends BiSymbol<RankedSymbol, Variable>>> map1
41   = new HashMap<RankedSymbol,
42       Tree<? extends BiSymbol<RankedSymbol, Variable>>>();
43 map1.put(t1,tf.makeTreeFromSymbol(bt1));
44 map1.put(f,tf.makeTreeFromSymbol(bf));
45 map1.put(not,tf.makeTreeFromSymbol(bnot,
46   Util.makeList(tf.makeTreeFromSymbol(bv0))));
47 map1.put(and,vtree);
48 EasyHom hom1 = new EasyHom(map1);
49
50 // build up, possibility 2 (only in EasyHom)
51 Set<RankedSymbol> alphabet9 = new HashSet<RankedSymbol>();
52 alphabet9.add(t1);
53 alphabet9.add(f);
54 alphabet9.add(not);
55 Map<RankedSymbol,
56   Tree<? extends BiSymbol<RankedSymbol, Variable>>> map2
57   = new HashMap<RankedSymbol,
58       Tree<? extends BiSymbol<RankedSymbol, Variable>>>();
59 map2.put(and,vtree);
60 EasyHom hom = new EasyHom(map2, alphabet);
```

#### 3.7.1.2 Creating a Homomorphism Without Using Variables

In order not to deal with variables, Lethal provides a second method to construct tree homomorphisms, the constructor

`GenHom(Map<G,Integer> toInts, Map<F,Tree<G>> hmap)`

Instead of variables, constants of the destination alphabet are used. In order to treat those constants like variables, the map `toInts` assigns each of these constants to a number. This number corresponds to the number of the `Variables` used in the homomorphism creation as mentioned above.

Furthermore, the constructor uses a map `h`. This map assigns each symbol of the source alphabet to a tree of the destination alphabet.

Make sure that the key set of this map contains constants only, i.e. symbols with arity 0.

The following example establishes a result equivalent to the last one.

Example 3.29: Creating a homomorphism

```
// creating the used symbols
...
RankedSymbol wv0 = new StdNamedRankedSymbol<String>("v0",0);
RankedSymbol wv1 = new StdNamedRankedSymbol<String>("v1",0);

// for creating trees
Tree<RankedSymbol> vtree3
  = tf.makeTreeFromSymbol(not,
    Util.makeList(tf.makeTreeFromSymbol(or,
      Util.makeList(tf.makeTreeFromSymbol(not,
          Util.makeList(tf.makeTreeFromSymbol(wv0))),
        tf.makeTreeFromSymbol(not,
          Util.makeList(tf.makeTreeFromSymbol(wv1)))))));

// build up
Map<RankedSymbol,Integer> toInts
  = new HashMap<RankedSymbol,Integer>();
toInts.put(wv0,new Integer(0));
toInts.put(wv1,new Integer(1));
Map<RankedSymbol,Tree<RankedSymbol>> map3
  = new HashMap<RankedSymbol,Tree<RankedSymbol>>();
map3.put(t1,tf.makeTreeFromSymbol(t1));
map3.put(f,tf.makeTreeFromSymbol(f));
map3.put(not,tf.makeTreeFromSymbol(not,
  Util.makeList(tf.makeTreeFromSymbol(wv0))));
map3.put(and,vtree3);
EasyHom hom2 = new EasyHom(toInts,map3);
```

### 3.7.2 Properties of a Homomorphism

There are several properties of homomorphisms, which can be checked.

| method | description |
|---|---|
| `isLinear()` | Checks whether the homomorphism is ***linear***, i.e. whether each variable appears at most once in each variable tree by which the homomorphism is defined. |
| `isEpsilonFree()` | checks whether a homomorphism is ***epsilon-free***, i.e. whether no symbol is mapped to a variable. |

| | |
|---|---|
| `isSymbolToSymbol()` | checks whether the homomorphism is ***symbol to symbol***, i.e. whether every symbol is mapped to another symbol and not to a complex tree. |
| `isComplete()` | checks whether the homomorphism is ***complete***, i.e. whether all variables with numbers between 0 and $n-1$ occur in the image of a symbol $f$ of arity $n$. |
| `isDelabeling()` | checks whether the homomorphism is ***delabeling***, i.e. whether it is linear, complete and symbol to symbol. A delabeling homomorphism only changes the label of the input symbol and possibly the order of subtrees. |
| `isAlphabetic()` | checks whether the homomorphism is ***alphabetic***, i.e whether it is delabeling and preserves the order and number of the subtrees. |

### 3.7.3 Applying a tree Homomorphism

Having defined the homomorphism succesfully, there are several possibilities to apply the homomorphism.

#### 3.7.3.1 On a Tree

The method `apply(Tree t)` applies the homomorphism to a tree as in the definition above.

#### 3.7.3.2 On a Finite Tree Automaton

apply!homomorphism to automaton Another possibility is to apply a linear homomorphism to a finite tree automaton, i.e. to apply the homomorphism to each tree of the language that is given by the finite tree automaton. The finite tree automaton accepting exactly those trees is obtained by the method `applyOnAutomaton(FTA fta )`.

*Remark*: If the homomorphism is not linear, the result is not a regular tree language and cannot be represented by a finite tree automaton. Thus an exception is thrown.

#### 3.7.3.3 Apply the Inverse Homomorphism On a Finite Tree Automaton

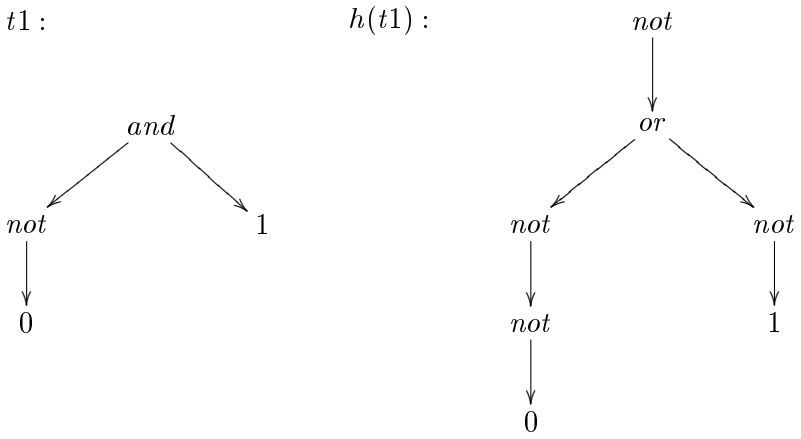
The method `applyOnInverseAutomaton(FTA fta)` computes the preimage of a regular tree language—given by a finite tree automaton—under a homomorphism.

Example 3.30: Applying a homomorphism

```
// build up the homomorphism hom1 as above
...
// apply to trees
Tree<RankedSymbol> t3 = tf.makeTreeFromSymbol(t1);
Tree<RankedSymbol> t2 = tf.makeTreeFromSymbol(and,
  Util.makeList(tf.makeTreeFromSymbol(not,
    Util.makeList(tf.makeTreeFromSymbol(f))),tf.
       makeTreeFromSymbol(t1)));
Tree<RankedSymbol> th1 = hom1.apply(t3);
Tree<RankedSymbol> th2 = hom1.apply(t2);

// apply to tree automaton
EasyFTA ftah1 = ...;
// build up a tree automaton accepting true formulas over
// the alphabet {and,not,0,1}
if (hom1.isLinear()){
  EasyFTA ftah3 = hom1.applyOnAutomaton(ftah1);
}
// apply to inverse automaton
EasyFTA ftah2 = ...;
// build up a tree automaton working on
// the alphabet {or,not,0,1}
EasyFTA ftah4 = hom1.applyInverseOnAutomaton(ftah2);
```

An example for applying such a homorphism on a tree is



### 3.7.4 Extension of Homomorphisms

The homorphism classes are built up similarly to the classes of finite tree automata. There is the interface `Hom<F,G,V>` which contains only a few getters for the basic parameters, for example the image variable tree of a given symbol. `F` and `G` are the symbol types of start and destination alphabet, `V` is the type of variables occuring in the variable trees needed for the definition of a homomorphism.

Furthermore, Lethal has the abstract class `AbstractHom<F,G,V>`, where the constructors and basic methods are implemented. It can be used as a base class.

The corresponding algorithms for `AbstractHom<F,G,V>` or a user-defined implementation of `Hom<F,G,V>` are implemented in the class `HomOps`. They are very flexible with their type parameters. In the following we explain how to use the methods.

#### 3.7.4.1 Compute Destination Alphabet

The method `computeDestAlphabet(Hom<F,G,V> hom)` returns a set which contains all symbols in the destination states, i.e. the destination alphabet.

#### 3.7.4.2 Apply to Tree

The method

```
apply(Hom<F,G,V> hom,
  X startTree,
  TreeCreator<G,Y> tc)
```

applies the homomorphism to a tree using a `TreeCreator`, which creates trees of the deserved type. Consider 3.4.3.2 to see how to implement a `TreeCreator`.

#### 3.7.4.3 Apply to a Finite Tree Automaton

Applying a homomorphism can be done with the method

```
X applyOnAutomaton(Hom<F,G,V> hom,
  FTA<F,Q1,R0> ta,
  Converter<Q1,Q2> c1,
  Converter<Pair<R0,List<Integer>>,Q2> c2,
  FTACreator<G,Q2,R,X> fc)
```

Here `hom` is the homomorphism, which is applied on the finite tree automaton over the same alphabet (with symboltype `F`), `ta`. The `FTACreator fc` is used to create the resulting finite tree automaton, in a similar fashion as the `Converters c1,c2` are used to create the states in the new automaton. The two `Converters` must form an injective mapping into `Q2`, which represents the state type of the resulting finite tree

automaton. Consider chapter 3.4.2 for further information on `Converter`, `FTACreator` and `TreeCreator`.

**Warning.** *The method only is applicable, if the homomorphism is linear.*

#### 3.7.4.4 Apply the Inverse Homomorphism to a Finite Tree Automaton

The method

```
applyInverseOnAutomaton(Hom<F,G,V> hom,
  FTA<G,Q2,? extends FTARule<G,Q2>> ta,
  Q1 s,
  Converter<Q2,Q1> c,
  TreeCreator<G,U> tc,
  FTACreator<F,Q1,R,Y> fc,
  TreeCreator<BiSymbol<G,Q2>,V0> btc)
```

applies the inverse homomorphism of `hom` to a finite tree automaton `ta`. This is possible even if the homomorphism is not linear. The `Converter c` converts states of the given finite tree automata into states of the resulting one, the `TreeCreator`s `tc` and `btc` are used for building trees inside the algorithm and the `FTACreator fc` again is needed for creating the resulting finite tree automaton.

(D. Jansen, M. Mohr, I. Thesing)

## 3.8 Hedge Automata

### 3.8.1 Hedge

Hedges are trees containing unranked symbols. When the finite tree automata work with trees containing ranked symbols, the hedge automata work with the hedges accordingly.

Definition of a hedge:

- The empty term sequence is a hedge.
- If $n \in \mathbb{N}$ and $h_0 \dots h_{n-1}$ are hedges and $a$ ia a symbol, then $a(h_0, \dots, h_{n-1})$ is a hedge.

A hedge consists of a root and a subhedges. As a root can be used any symbol which is an implementation of the interface `UnrankedSymbol`. For creation of hedge you need a symbol and a list of subhedges (which can be empty).

Example 3.31: Creation of hedge

```
// a and b are symbols
List<Tree<UnrankedSymbol>> subHedges1
  = new LinkedList<Tree<UnrankedSymbol>>();
// hedge a()
Tree<UnrankedSymbol> hedge1
  = new StdTree<UnrankedSymbol>(a, subHedges);
List<Tree<UnrankedSymbol>> subHedges2
  = new LinkedList<Tree<UnrankedSymbol>>();
subHedges2.add(hedge1);
subHedges2.add(hedge1);
subHedges2.add(hedge1);
Tree<UnrankedSymbol> hedge2
  = new StdTree<UnrankedSymbol>(b, subHedges2);
```

### 3.8.2 Hedge Automaton

Hedge automata are applicable on hedges, as finite tree automata are applicable on trees. In this section, we explain how to define and use hedge automata.

#### 3.8.2.1 States

Unlike the symbols, for the creation of the hedge automaton no special states are needed. The hedge automaton uses same states as the finite tree automaton. There is just one necessary condition for the used states: the implementation of the interface `State`.

#### 3.8.2.2 Rules

The rules of the hedge automaton have the form $f(expr) \to q$, where

- $f$ is a symbol implementing the interface `UnrankedSymbol`.
- $expr$ is an implementation of the interface `RegularLanguage`. It can be an expression, a word automaton or any user-defined expression.
- $q$ is a state.

Example 3.32: Creation of a rule

```
// a is a symbol, q is a state
// creationof the rule a() -> q
HedgeRule rule = new HedgeRule(a, new EmptyExpression(), q);
```

#### 3.8.2.3 RegularExpression

The interface `RegularExpression` extends the interface `RegularLanguage` to be used in the expressions. The `Expression`is one possible implementation of this interface. A regular expression is defined as $expression^{n...m}$, where $n \geq 0$ and $m \geq 0$ or $m = -1$ for infinity.

The expression is divided into two parts:

- Part 1 containing the $()^{n...m}$ represented by `Expression`
- Part 2 containing the *expression* represented by `SingeExpression`

#### 3.8.2.4 Expression

The `Expression` represents the part 1 of the `RegularExpression`. It saves how many times the `SingleExpression` shall repeat itself. This way it is possible to use same `SingleExpression` with different repetitions (which improves the efficiency of the resulting automaton).

#### 3.8.2.5 SingleExpression

A `SingleExpression` represents the part 2 of the `RegularExpression`.

There are several `SingleExpressions` to use:

- `BasicExpression`: contains a list of states of the hedge automaton (like the rule of a finite tree automaton),
- `OrExpression`: contains two expressions, which are combined with an 'OR',
- `ConcatExpression`: contains two expressions, which are combined with an 'AND',
- `JoeExpression`: contains one expression. Used for constructs such as $((exp)^k)^l$.

Example 3.33: Creation of the expressions

```
1 // q1 and q2 are states
2 List<State> states = new LinkedList<State>();
3 states.add(q1);
4 // creation of the basic expression (q1)
5 BasicExpression bExp1 = new BasicExpression(states);
6 states = new LinkedList<State>();
7 states.add(q2);
8 // creation of the basic expression (q2)
9 BasicExpression bExp2 = new BasicExpression(states);
```

```
// creation of the expression (q1)*
Expression exp1 = new Expression(0, -1, bExp1);
// creation of the expression (q2)*
Expression exp2 = new Expression(0, -1, bExp2);
// creation of the expression ((q1)* | (q2)*)
OrExpression orExp = new OrExpression(exp1, exp2);
// creation of the expression ((q1)* | (q2)*)^1
Expression exp3 = new Expression(1, 1, orExp);
```

#### 3.8.2.6 Word Automaton

A word automaton is second possible implementation of the `RegularLanguage`.

A (finite) `Word Automaton` over $\mathcal{F}$ is a tuple $\mathcal{A} = (\mathcal{Q}, \mathcal{F}, \mathcal{R}, q_0, \mathcal{Q}_f)$ where

- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{F}$ is a finite set of symbols,
- $\mathcal{R}$ is a finite set of rules, which have the form $r = f(q_1) \to q_2$, where $f \in \mathcal{F}$ and $q_1, q_2 \in Q$.
- $q_0 \in \mathcal{Q}$ is an initial state
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states.

Upon the creation of the `Word Automaton` the sets $\mathcal{Q}$, $\mathcal{R}$, $\mathcal{Q}_f$ and $q_0$ are used, while $\mathcal{F}$ is constructed consisting of the symbols used in the rules from $\mathcal{R}$.

Since this implementation of a word automaton can only be used for the creation of the hedge automaton rules, it has no operations, like the tree and hedge automata do. Since the hedge automaton works with expressions over states, the word automaton describes the sequence of the states. Therefore the symbols of the word automaton are states.

Example 3.34: Creation of the finite word automaton

```
// state for the word automaton can be created by the using a
    StateFactory
StateFactory stFact = StateFactory.getStateFactory();
State q1 = stFact.makeState("q1");
// set of states for the word automaton
Set<State> wStates = new HashSet<State>();
wStates.add(q1);
// rule for the word automaton: s(q1) -> q1
State s = stFact.makeState("s");
WordRule wRule = new WordRule(s, q1, q1);
```

```
// set of rules for the word automaton
Set<WordRule> wRules = new HashSet<WordRule>();
wRules.add(wRule);
// set of final states for the word automaton
Set<State> wFinStates = new HashSet<State>();
wFinStates.add(q1);
// creation of the word automaton
WordAutomaton wa
  = new WordAutomaton(wStates, wRules, q1, wFinStates);
```

#### 3.8.2.7 Hedge Automaton

A hedge automaton over $\mathcal{F}$ is a tuple $\mathcal{A} = (\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \mathcal{R})$ where

- $\mathcal{F}$ is a finite set of symbols,
- $\mathcal{Q}$ is a finite set of states,
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ is a set of final states.
- $\mathcal{R}$ is a finite set of rules, which have the form $r = f(exp) \to q$, where $f \in \mathcal{F}$ $exp$ a regular language over $\mathcal{Q}$ and $q \in Q$.

Upon the creation of the word automaton the sets $\mathcal{Q}$, $\mathcal{R}$ and $\mathcal{Q}_f$ are used, while $\mathcal{F}$ is constructed consisting of the symbols used in the rules from $\mathcal{R}$.

Example 3.35: Creation of the hedge automaton

```
// a and b are symbols
// state for the hedge automaton can be created by the using
   a StateFactory
StateFactory stFact = StateFactory.getStateFactory();
State q1 = stFact.makeState("q1");
State q2 = stFact.makeState("q2");
// set of states for the hedge automaton
Set<State> states = new HashSet<State>();
states.add(q1);
states.add(q2);
// 1. rule of the hedge automaton: a() -> q1
HedgeRule rule1
  = new HedgeRule(a, new EmptyExpression(), q1);
// 2. rule of the hedge automaton: b(q1*) -> q2
List<State> base = new LinkedList<State>();
base.add(q1);
HedgeRule rule2
```

```
17   = new HedgeRule(b,
18       new Expression(0, -1, new BasicExpression(base)), q2);
19 // set of rules of the hedge automaton
20 Set<HedgeRule> rules = new HashSet<HedgeRule>();
21 rules.add(rule1);
22 rules.add(rule2);
23 // set of final states for the hedge automaton
24 Set<State> finStates = new HashSet<State>();
25 finStates.add(q2);
26 // creation of the hedge automaton
27 HedgeAutomaton ha
28   = new HedgeAutomaton(states, finStates, rules);
```

### 3.8.3 Operations

#### 3.8.3.1 decide

The method `decide()` checks whether the given hedge is in the language of the given hedge automaton.

Example 3.36: decide

```
1 HedgeAutomaton ha = ...;
2 Tree<UnrankedSymbol> hedge = ...;
3 if (HAOps.decide(ha, hedge)){
4         System.out.println("This hedge is in the language of
              this hedge automaton!")
5 }
```

#### 3.8.3.2 Empty Language

The method `emptyLanguage()` checks whether a given hedge automaton has an empty language, i.e. whether it does not accept any hedge.

Example 3.37: Empty Language

```
1 HedgeAutomaton ha = ...;
2 if (HAOps.emptyLanguage(ha)){
3         System.out.println("The language of this hedge
              automaton is empty!")
4 }
```

#### 3.8.3.3 Finite Language

The method `finiteLanguage()` checks whether the language of a given hedge automaton recognizes only finite number of hedges.

Example 3.38: Finite Language

```
HedgeAutomaton ha = ...;
if (HAOps.finiteLanguage(ha)){
        System.out.println("The language of this hedge
            automaton is finite!")
}
```

#### 3.8.3.4 One language is a subset of the other one

The method `subsetLanguage()` takes two hedge automata and checks whether the language of the first automaton is a subset of the second one. If it yields true, every hedge recognized by the first automaton is also recognized by the second one.

Example 3.39: Subset Language

```
HedgeAutomaton ha1 = ...;
HedgeAutomaton ha2 = ...;
if (HAOps.subsetLanguage(ha1, ha2)){
        System.out.println("The language of the first hedge
            automaton is the subset of the language of the
            second hedge automaton!")
}
```

#### 3.8.3.5 Two automata describe the same language

The method `sameLanguage()` checks whether two hedge automata recognize the same language. This method returns `true`, if the language of the first hedge automaton is a subset of the language of the second hedge automaton, and the language of the second hedge automaton is a subset of the language of the first hedge automaton.

Example 3.40: The same Language

```
HedgeAutomaton ha1 = ...;
HedgeAutomaton ha2 = ...;
if (HAOps.sameLanguage(ha1, ha2)){
        System.out.println("This hedge automata recognize the
             same language!")
}
```

#### 3.8.3.6 Complement

The method `complement` constructs a hedge automaton which recognize hedges with the same alphabet as the given hedge automaton, but with complementary language.

Example 3.41: Complement

```
1 HedgeAutomaton ha = ...;
2 HedgeAutomaton complementHA = HAOps.complement(ha1, ha2);
```

#### 3.8.3.7 Union

The method `union()` constructs a hedge automaton, which accepts the union of the languages of the two given hedge automata. More precisely, the resulting automaton accepts a hedge if and only if one of the two given automata accepts it.

Example 3.42: Union

```
1 HedgeAutomaton ha1 = ...;
2 HedgeAutomaton ha2 = ...;
3 HedgeAutomaton unionHA = HAOps.union(ha1, ha2);
```

#### 3.8.3.8 Intersection

The method `intersectionBU()` constructs a hedge automaton which accepts the intersection of the languages of the two given hedge automata. More precisely, the resulting automaton accepts a hedge if and only if both of the two given automata accept it.

Example 3.43: Intersection

```
1 HedgeAutomaton ha1 = ...;
2 HedgeAutomaton ha2 = ...;
3 HedgeAutomaton intersectionHA
4   = HAOps.intersectionBU(ha1, ha2);
```

#### 3.8.3.9 Difference

The method `difference()` constructs a hedge automaton which accepts the difference of the languages of the two given hedge automata. This means that the resulting automaton accepts every hedge which is accepted by the first hedge automaton, but not by the second one.

Example 3.44: Difference

```
1 HedgeAutomaton ha1 = ...;
2 HedgeAutomaton ha2 = ...;
3 HedgeAutomaton differenceHA = HAOps.difference(ha1, ha2);
```

#### 3.8.3.10 Construct Tree Witness

Given a hedge automaton with a non-empty language, `constructTreeFrom()` constructs a witness hedge which is contained in the language. Note that `null` is returned, if the given automaton is empty, or if the given automaton has an empty language.

Example 3.45: Construct Tree Witness

```
1 HedgeAutomaton ha = ...;
2 Tree hedge = HAOps.constructTreeFrom(ha1);
```

(A. Reis, M. Schatz)

## 3.9 Tree Transducer

As an additonal feature, Lethal provides the basic functionality of tree transducers. This feature is found in the package `treetransducer`.

Recall that a ***tree transducer*** is very similar to a finite tree automaton, but has an additional output. A tree transducer rule hast the form $f(q_0, \ldots, q_{n-1}) \to (q, u)$. In these rules, $f$ is a source symbol of arity $n$, $q, q_0, \ldots, q_{n-1}$ are states and $u$ is a tree containing destination symbols and at most $n$ variables. It is possible to put epsilon rules in, those are rules of the form $q \to (p, u)$, where $p$ and $q$ are states and $u$ is a tree containing at most one variable. By omitting the output from the rules, a finite tree automaton is obtained. It can be retrieved by the method `getFTAPart()`.

### 3.9.1 Creating Tree Transducer

Again Lethal has two classes representing tree transducers, one for easy usage and one for usage with type parameters. The parametrized implementation is `GenTT<F,G,Q>`, where `F` is the type of the start alphabet, `G` is the type of the destination alphabet and `Q` is the type of states.

Creating rules of the type `TTRule`, `TTEpsRule`, `EasyTTRule` or `EasyTTEpsRule` is very similar to creating rules for finite tree automata. Lethal demands that if there are two rules $f(q_0, \ldots, q_{n-1}) \to (q, u)$ and $f(q_0, \ldots, q_{n-1}) \to (q, v)$ the variable trees $u$ and $v$ are equivalent. This can always be achieved by adding an additional state for one of the two right sides of the rules.

For constructing a tree transducer one of the following constructors can be used:

- ```
  GenTT(Collection<Q> finalStat,
    Collection<F> startAlph,
    Collection<G> destAlph,
    Collection<TTRule<F,G,Q>> rul,
    Collection<TTEpsRule<G,Q>> epsRul)
  ```

  where all parameters for the tree transducer have to be supplied.

- ```
  GenTT(Collection<Q> finalStat,
    Collection<TTRule<F,G,Q>> rul,
    Collection<TTEpsRule<G,Q>> epsRul)
  ```

  where the states and alphabets are calculated out of the final states and the given rules.

Note that all epsilon rules are elimated while creating the tree transducer. The same constructors exist without demanding for epsilon rules.

### 3.9.2 Run a Tree Transducer

The method `doARun(Tree)` applies a tree transducer to a tree. It returns the set of all trees which are constructed within an accepting run of the tree transducer. This set is empty if and only if the tree is not accepted. It contains at most one tree, if the tree transducer is deterministic.

### 3.9.3 Decide

As a finite tree automaton, a tree transducer can decide whether a given tree is contained in the tree language it represents. This is done by the method `decide(Tree)`.

### 3.9.4 Properties of a Tree Transducer

A tree transducer is called ***linear***, if all variable trees occurring on the right sides of the rules are linear, i.e. if they contain no variable twice[8]. The linearity of a tree transducer can be checked by the method `isLinear()`.

A tree transducer is called ***deterministic*** or ***complete*** if the finite tree automaton encapsulated in the tree transducer is deterministic or complete, respectively. These

[8] twice means „twice or more often“

properties can be checked with the corresponding methods in `treeautomata.common.FTAProperties`. Consider 3.3.2 for further information.

The methods checking properites of the recognized language in `FTAProperties` give no information about the destination languages of tree transducers. Two tree transducers recognizing the same language need not have the same destination trees.

### 3.9.5 More Operations with Tree Transducers

Some more operations which can be applied to a tree transducer are found in the class `TTOps`. Normally the methods are applicable to tree transducers of type `GenTT<F,G,Q>`, but also to the subclass `EasyTT`. If the resulting type is a `GenTT<RankedSymbol,RankedSymbol,State>` (which is the use if you apply it on an `EasyTT`), it is possible to transform it into an `EasyTT` by means of the method `convertToEasyTT()`.

#### 3.9.5.1 Union of Two Tree Transducers

With the method `union(GenTT<F,G,Q1> tt1,GenTT<F,G,Q2> tt2)` it is possible to form the union of the languages recognized by two different tree transducers. The method is based on the corresponding method in `FTAOps`. The two tree transducers must have the same symbol types. The return type is an `GenTT<F,G,State>`, where the states are produced by different `NamedStates`.

By contrast, there are no methods for complement or intersection.

#### 3.9.5.2 Apply a Tree Transducer to a Finite Tree Automaton

With the method `runOnAutomaton()` it is possible to apply a tree transducer to a finite tree automaton, i.e. the tree transducer is applied to all trees contained in the language of the given finite tree automaton. The result is again a finite tree automaton, if the tree transducer is linear. Therefore the tree transducer has to be linear, otherwise an exception is thrown. Note that only those symbols are considered that are contained in both alphabets: the one of the tree transducer and the one of the finite tree automaton.

There are two different variants:

- ```
  lst{T runOnAutomaton(GenTT<F,G,Q> tt,
      FTA<F,Q1,R1> fta,
      Converter<Pair<Q,Q1>,Q2> sc,
      Converter<Triple<TTRule<F,G,Q>,R1,Tree<BiSymbol<G,
         Variable>>>,Q2> tsc,
      FTACreator<G,Q2,R2,T> creator)
  ```
  ```

where the result type of the finite tree automaton can be controlled with different converters and an `FTACreator`. For more information on how to use these, consider chapter 3.4.

- `GenFTA<G,State> runOnAutomaton(GenTT<F,G,Q> tt, GenFTA<F,Q1> fta)`

  where the state type is generated automatically. In fact it is a complicated construction of named states, saved only as `State`. Here only the two main parameters must be supplied.

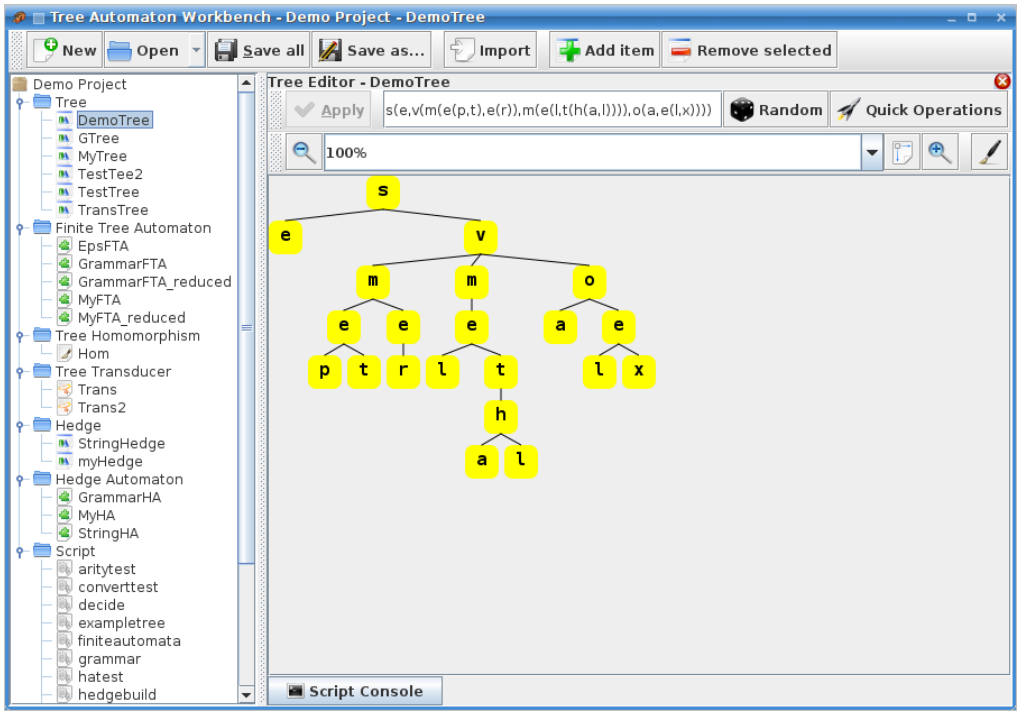
(D. Jansen, M. Mohr, I. Thesing)

# 4 Workbench GUI

## 4.1 Introduction

The Tree Automaton Workbench is a simple GUI that allows to quickly define and apply tree and hedge automata. It is organized as a project workbench with a project item list and item editors.

Figure 4.1: Workbench




Project items are grouped into the fixed categories of supported classes. These are:

1. Tree — Standard (fixed arity) trees with ranked symbols.
2. Finite Tree Automaton — Standard (non-determininistic) Tree Automata.

3. Tree Homomorphism — Linear and non-linear Tree homomorphisms.
4. Hedge — Unranked trees.
5. Hedge Automata — Automata on hedges with Regular expression matching on states.
6. Script — Simple scripting language to automate multiple operations on items.

## 4.2 Usage

### 4.2.1 Managing Projects

The main window toolbar contains the essential functionality for managing projects:

1. „New“ creates a new project. The current project is closed without beeing saved.
2. „Open“ opens a previously saved project. The current project is closed without being saved.
3. „Save all“ saves the current project, if the project has not been saved before the filename is prompted. Note that „Save all“will try to apply all changes made to items. If an item contains invalid changes save will be aborted.
4. „Save As...“ saves the current project, the filename is prompted.
5. „Import“ imports external files as project items. Currently only tree and hedge import from XML is supported.
6. „Add item“ creates a new item in the project. The name must be unique and it must only contain alphanumerical characters starting with a letter.
7. „Remove item“ deletes the currently selected item from the project.

The „Add and remove functionality„is also available in the context menu of the project item list.

### 4.2.2 Defining Items

#### 4.2.2.1 Trees

„Add item“ → „Tree“ adds a new Tree item to the Project.
The tree is entered into the topmost text field. The syntax is:

Syntax Tree

```
1 TREE ::= SYMBOL_NAME["("TREE(","TREE)*")"]
2 SYMBOL_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
```

Note that because trees are ranked, symbols with identical names but different arities (number of subtrees) are considered as different symbols.

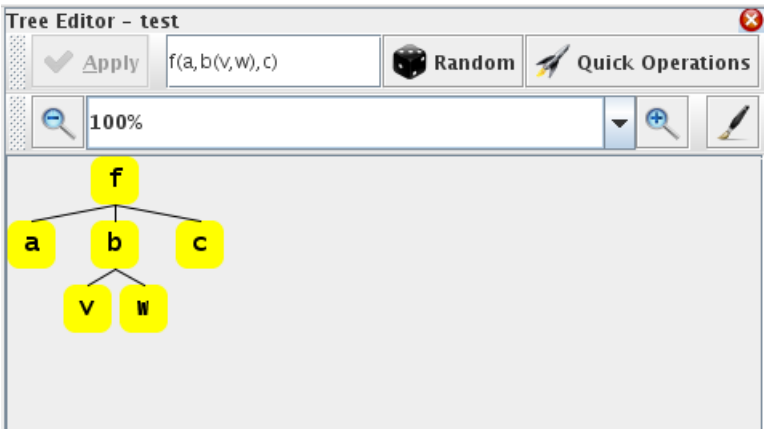
As a shortcut quotations marks can be used to generate a string tree with individual character symbols, e.g. „mytext“ can be used as a shortcut for „_string_(m,y,t,e,x,t)“.

When entered the tree is graphically displayed below the input field.

The tree nodes can be shown as circles, ovals, rectangles or round rectangles. Zooming in and out is supported.

The „Random“ button can be used to quickly generate a random tree for demo purposes.

Figure 4.2: Example Tree




#### 4.2.2.2 Finite Tree Automatons

„Add item“ → „Finite Tree Automaton“ adds a new FTA item to the Project.

Finite tree automata are entered as a set of rules or as a tree grammar into the editor window. The syntax in rule mode is:

Syntax Finite Tree Automata

```
1 RULE = NORMAL_RULE | ESPILON_RULE
2 NORMAL_RULE = SYMBOL_NAME["("STATE_NAME(","STATE_NAME)*")"]"->"
      STATE_NAME
3 EPSILON_RULE = STATE_NAME"=>"STATE_NAME
```

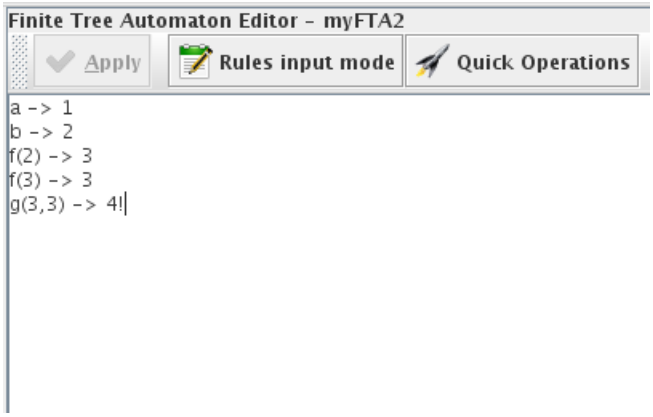
```
4 SYMBOL_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
5 STATE_NAME  ::= ("0"..."9"|"a"-"z"|"A"-"Z")+["!"]
```

The „!“ is used to mark the final states of the automaton. Multiple occurances of a name will be treated as the same final state if at least one is followed by a !-mark.

Note that because trees are ranked, symbols with identical names but different arities (number of subtrees) are considered as different symbols.

The symbol names in the rules correspond to the symbol names in the trees the automaton is applied on.

Figure 4.3: Finite Tree Automaton Rules Example




The syntax in grammar mode is:

Syntax Finite Tree Automata

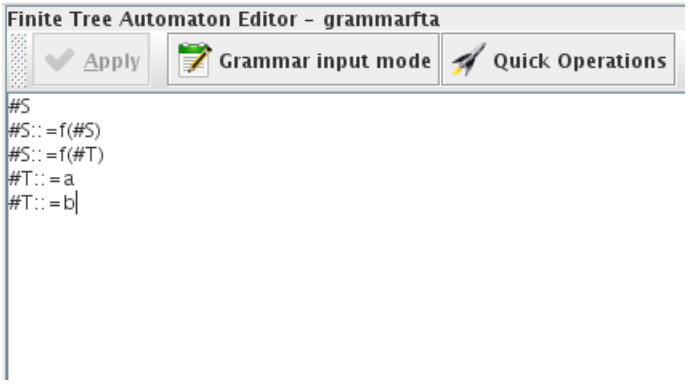
```
1 GRAMMAR_LINE ::= GRAMMAR_RULE | NON_TERMINAL
2 GRAMMAR_LINE ::= NON_TERMINAL "::=" GRAMMAR_TREE
3 GRAMMAR_TREE ::= TERMINAL | NON_TERMINAL ["("GRAMMAR_TREE(","
      GRAMMAR_TREE)*")"]
4 TERMINAL ::= NAME
5 NON_TERMINAL ::= ("#"|"$") NAME
6 NAME  ::= ("0"..."9"|"a"-"z"|"A"-"Z")+["!"]
```

The Terminal names in the grammar correspond to the symbol names in the trees the automaton is applied on.

#### 4.2.2.3 Tree Homomorphisms

„Add item“ → „Tree Homomorphism“ adds a new Homomorphism item to the Project.

Figure 4.4: Finite Tree Automaton Grammar Example




A tree homomorphism is a tree transforming entitiy with rules describing the transformation

Homomorphisms are entered as a set of rules into the editor window. The rule syntax is:

Syntax Homomorphisms

```
1 RULE = SYMBOL_NAME["("VARIABLE_NAME(","VARIABLE_NAME)*")"]"->"
      TREE
2 TREE ::= (SYMBOL_NAME["("TREE(","TREE)*")"]) |VARIABLE_NAME
3 SYMBOL_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
4 VARIABLE_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
```

The left side of a rule defines the symbol the homomorpism is applied on and declares the variable names which refer to the subtrees of the symbol when the rule is applied. On the right side of the rule there is the variable tree. A variable tree is basically a normal tree. However the variable names introduced on the left side can be used to refer to the subtrees the rule matches. Note that the only difference between a symbol and a variable name in the tree is that variables are „declared“ on the left side of the rule, whereas other names are treated a symbols. Variables must be leaves of the variable tree because they are placeholders for a subtree.

#### 4.2.2.4 Tree Transducer

„Add item“ → „Tree Transducer“ adds a new tree transducer item to the Project.

A tree transducer is a combination of a tree homomorphism and a finite tree automaton. Rules therefore contain information which serves to decide wether a tree is accepted as well as information to modify its structure.

It is entered as a set of rules into the editor window. The rule syntax is:

Syntax Tree Transducer

```
1 RULE::= NORMAL_RULE|EPSILON_RULE
2 NORMAL_RULE::=SYMBOL_NAME [ "(" STATE_NAME [ ":" VAR_NAME ] ( ",
      " STATE_NAME [ ":" VAR_NAME ] )* ")" ] "->" STATE_NAME [ ","
       VAR_TREE ]
3 EPSILON_RULE::= STATE_NAME [ ":" VAR_NAME ] "=>" STATE_NAME [ ",
      " VAR_TREE ]
4 VAR_TREE ::= ( SYMBOL_NAME [ "(" VAR_TREE ( "," TREE )* ] ) |
      VAR_NAME
5 SYMBOL_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+ ["!"]
6 VAR_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
```

There are two types of rules:

Normal rules are similar to finite tree automata rules but can also contain tree structure modification information. For this purpose variable names can be added to left side of the rule (right behind the state names). These names identify the subtrees the left side of the rule matches. On the right side the destination state can be followed by a comma and a variable tree. The variable tree describes the new structure of the tree produced when the rule is applied. A variable tree is basically a normal tree but the variable names introduced on the left side can be used to refer to the subtrees the rule matches. Note that the only difference between a symbol and a variable name in the tree is that variables are „declared“ on the left side of the rule, whereas the other names are treated a symbols. Variables must be leaves of the variable tree, as they are placeholders for a subtree.

Epsilon rules can be used to introduce tree nodes to the output or to change automaton states without consuming a tree node on the input (the latter is similar to epsilon rules in finite tree automata). The left side defines the state in which the epsilon rule can be applied and the optional variable name which identifies the tree the rule is applied on. The right side consists of the destination state that is entered when the rule is applied and an optional variable tree which may refer to the left side variable defining the new structure inserted when the rule is applied.

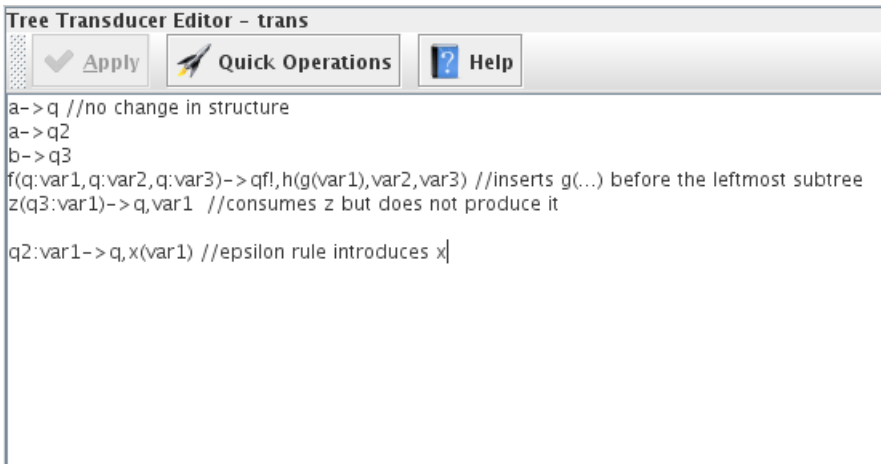
As for finite tree automata the „!“ is used to mark the final states of the tree transducer.

#### 4.2.2.5 Hedges

„Add item“ → „Hedge“ adds a new Hedge item to the project.

Hedges are entered similar to trees. However symbols in hedges are unranked and thus symbols with the same name are treated as the same symbol for any number of subtrees.

Figure 4.5: Tree Transducer Example




#### 4.2.2.6 Hedge Automatons

„Add item“ → „Hedge Automaton“ adds a new hedge automaton item to the project.

Hedge automata can be entered as a set of rules or as a grammar into the editor window.

The syntax in rule mode is:

Syntax Hedge Automata

```
1 RULE ::= SYMBOL_NAME["("REGEXP")"]"->"STATE_NAME
2 REGEXP ::= (STATE_NAME|"(" REGEXP")")["*"|"+"|"?"|"{"[NUM]","[
      NUM]"}"][("|"REGEXP)|(","? REGEXP)]
3 SYMBOL_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
4 STATE_NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+["!"]
```

The „!“ is used to mark the final states of the automaton. Multiple occurances of a name will be treated as the same final state if at least one is followed by a !-Mark.

The symbol names in the rules correspond to the symbol names in the hedges the automaton is applied on.

The syntax in grammar mode is:

Syntax Hedge Grammars

```
1 GRAMMAR_LINE ::= GRAMMAR_RULE|NON_TERMINAL
2 GRAMMAR_RULE ::= NON_TERMINAL"::="TERMINAL"("REGEXP")"
3 REGEXP ::= (NON_TERMINAL|TERMINAL"("REGEXP")"|"("REGEXP")")["*"|
      "+"|"?"|"{"[NUM]","[NUM]"}"][("|"REGEXP)|(","REGEXP)]
```

```
4 TERMINAL ::= NAME
5 NON_TERMINAL ::= ("#"|"$")NAME
6 NAME ::= ("0"..."9"|"a"-"z"|"A"-"Z")+
```

Lines which only contain a NON_TERMINAL symbol mark this symbol a start symbol for the grammar.

The Terminal names in the grammar will be matched against the symbol names in the hedges the automaton is applied on.

#### 4.2.2.7 Script

„Add item“ → „Script“ adds a new script item to the Project.

Scripts can be used to automate a sequence of operations on trees and hedges. The documentation can be found in the next section.

## 4.3 Script Language

### 4.3.1 Introduction

The built-in script interpreter allows for easy automation of various hedge- and tree(automata) operations. It features a ruby-inspired syntax and support for classes and objects. It allows a procedural as well as a functional programming style.

Primitive values such as integers are treated as first order objects as well as functions. As all values are objects they can be passed around as arguments and have attributes and functions (methods).

### 4.3.2 Syntax Basics

#### 4.3.2.1 Using Variables

Variables are defined implicitly on first assignment and can be assigned without prior introduction. However, before reading a variable it must have been assigned at least once or an „Unknown name“ error will be raised.

Example 4.1: Variable Usage

```
1 myVariable = "Hello World"
2 otherVar = myVariable
```

Variable names are case sensitive.

#### 4.3.2.2 Control Structures

There are straight forward „if-else“ condition and „while“ statements.

Example 4.2: If condition

```
a = ...
if a then
        print("a is true")
else
        print("a is false")
end
```

The „else“ clause may be omitted.

Example 4.3: While loop

```
a = 6
n = 1
while a > 1
        n = n * a
        a = a - 1
end
print("6! = "  + n)
```

#### 4.3.2.3 Truth

Truth values are represented by the „Boolean“ class and the possible values „true“ and „false“. However, control statements like „while“ and „if“ and boolean operators may be used with expressions of any type. In this case every value that is not „false“ or „null“ is considered as „true“İn contrast to languages like C the Integer value 0 is considered as „true“.

#### 4.3.2.4 Using Functions

There are two ways of defining functions. The first is the procedural language like way of using the „def“ keyword. It defines a function and assigns it to a name.

Example 4.4: Function Definiton 1

```
def sayHello(name)
        print("Hello " + name)
end
```

Such a function can be executed by writing its name. Brackets are required only if parameters are passed to it. When executed the function will run in the context of

the object or class it is defined in. Functions defined outside of any object are runned in the implicitly present main environment.

The second way of defining functions is a functional language like construct that defines a function and returns a Method object as a result.

Example 4.5: Function Definiton 2

```
sayHello = proc{|name|
        print("Hello " + name)
}
```

The returned Method object contains the function defined and a reference to its original definition environment. It provides the „call“ function to execute the contained function.

Example 4.6: Calling Functions

```
def sayHelloWorld()
        print("Hello World")
end

sayHelloWorld  #prints "Hello World".

helloMethod = proc{
        print("Hello World")
}
helloMethod.call #prints "Hello World".
```

This example shows the two different ways of defining and calling functions to achieve the same result. The ability to use a function as an object without calling it allows for a functional programming style:

Example 4.7: Passing Functions Around

```
sayHelloWorld = proc{
        print("Hello World")
}

def doItTwice(what)
        what.call()
        what.call()
end

doItTwice(sayHelloWorld)   #prints "Hello World" twice
```

Function parameter names are written as a comma separated list on the function header enclosed in round brackets for the „def“-style definitons and in vertical bar (pipe) symbols for the functional „proc“-style definitions:

Example 4.8: Function Parameters

```
def add(a,b)
        a+b
end
print(add(1,2))

addition = proc{|a,b|
        a+b
}
print(addition.call(1,2))
```

Function return values are defined as the return value of the last statement of the function.

Additionally to explicitly passing of functions to methods as arguments a shorthand notation named „block passing“ is available. An anonymous function (a block) is defined directly in the function call and passed to the called function.

Example 4.9: Implicitly Passes Functions (block passing) 1

```
5.times{|i|
        print("Hello World " + i)
}
#prints "Hello World 1" \n "Hello World 2" \n ... "Hello
   World 5"
```

This will pass the anonymous block to the function „times“ of the object „5“ which will call the block five times.

Using block passing the above „doItTwice“ example can be simplified to:

Example 4.10: Implicitly passes functions (block passing) 2

```
def doItTwice()
        yield()
        yield()
end

doItTwice(){
        print("Hello World")
} #prints "Hello World" twice
```

The „yield“ will execute the block passed to the function. To determine whether a block has been passed to a function call, the boolean „block_given?“ value can be read inside every function.

### 4.3.3 Using Classes and Objects

A set of built-in classes as well as the ability for user defined classes are supported. However, there is currently no support for inheritance or interfaces.

Values (including functions) that are defined in class or object context are accessed with . expressions.

Example 4.11: Accessing objects

```
result = myObject.objectMember
```

Objects are created from classes by using the „new“ function on classes.

Example 4.12: Instancing classes

```
arr = Array.new(1,2,3,4,5)
arr.remove(3)
arr.add(0)
print(arr)  #prints "[1,2,4,5,0]"
```

### 4.3.4 Defining Classes

There is limited support for defining custom classes in the script interpreter. To define a class the “class” statement is used:

Example 4.13: Defining classes 1

```
class Greeter
        def initialize(persontogreet)
                this.name = persontogreet
        end

        def greet
                print("Hello " + name)
        end
end

g = Greeter.new("Joe")
g.greet  #prints "Hello Joe"
```

The „class“ statemtent is followed by the name of the class, the body and ends with the usual „end “ keyword. Inside the class body any script code can be written and is executed in the context of the class when the class definition is processed. In the example above the body only contains two function definitons. The function name „initialize “ is reserved for the constructor of the class. When a new instance of the class is created, „initialize “ will be called with the arguments passed to the „new “

call on the class.
The prefix „this “ is used to refer to the current object context. It is used in the „initialize “ method to store the variable „name “ and its value explicitly in the object. Without the „this“ prefix a local variable „name“ would be created and an „Undefined name“ error gets raised when trying to call „greet“ on the object, since the variable „name“ gets lost when „initialize“ ends.

Remember that the class body is processed in class context, so using „this“ outside of a method will make it refer to the current class.

To explicitly refer to the class context no matter whether the code is running in object or class context the „class“ prefix can be used.

Example 4.14: Defining classes 2

```
class Greeter
        name = "World"

        def initialize(name)
                this.name = name
        end

        def greet
                print("Hello " + name)
        end

        def greetWorld
                print("Hello " + class.name)
        end
end

g1 = Greeter.new("Don")
g2 = Greeter.new("Joe")
g1.greet          #prints "Hello Don"
g2.greet          #prints "Hello Joe"
g1.greetWorld     #prints "Hello World"
g2.greetWorld     #prints "Hello World"
Greeter.greet     #prints "Hello World"
```

The first statement inside the class binds the „name“ in the Greeter class context to the string „World“. When creating an instance „name“ is additionally bound in the object context and is assigned the value of the local variable „name“.
The method „greet“ reads the binding from the innermost context. In the first two test calls, the innermost context is the current object and „greet“ reads and prints the value passed to the constructor.
The method „greetWorld“reads the binding from the class context and thus prints

the String „World“ bound in the class context.
The last call of „greet“ is done in class context rather than object context as it was done in the first two calls, thus it also reads the class context binding of „name“.

### 4.3.5 Build-in Classes and Objects

The script interpreter comes with a number of build-in classes that allow working with Lethal objects as well as a limited set of general purpose data types and structures.

#### 4.3.5.1 Class: Project

Project is the namespace for access to all items in the project currently loaded in the GUI.
Class Members:

- Method: ***each***(class1,class2,...){|item ... |}
  iterates over all items in the project that are of any of the given classes. Valid classes are „Tree“,„FTA“,„Homomorphism“,„Hedge“,„HA“. The given block is called once for each item. If no arguments are given it will iterate over all items in the project.

  Example 4.15: Show all hedges in the current project

```
1 Project.each(Hedge){|item| show(item)}
```

- Method: ***all***(class1,class2,...)
  Returns an array of all project items that are of any of the given classes. Valid classes are „Tree“,„FTA“,„Homomorphism“,„Hedge“,„HA“. If no arguments are given all items in the current project will be returned.

  Example 4.16: Select all finite language FTAs from the project

```
1 Project.all(FTA).select(){|fta| fta.finite_language()}
```

- Property: <item_name>
  Each item in the project is bound to it's name in the project class. For example, if there is an item „myFTA“ in the current project, it can be accessed as „Project.myFTA“ by scripts.

  Example 4.17: Apply project items

```
1 print(Project.myFTA.decide(Project.myTree))
```

#### 4.3.5.2 Class: FTA

FTA represents the class of finite tree automata.
Class Members:

- Method: ***new***(rule1, ..., ruleN)
  Creates a new FTA object with the given rules. The rule definition is similar to one given in the GUI Finite Tree Automaton section. (See → 4.2.2.2 Finite Tree Automaton) Note: Multiple rules in a single argument are allowed if separated by linebreaks

Object Members:

- Method: ***decide***(tree)
  Takes a Tree object and applies this automaton on it. It returns true if the FTA accepts the given tree and false if not.

- Method: ***trace***(tree)
  Takes a Tree object and applies this automaton on it. It returns a FTATrace object that contains information to visualize the FTA run on the given tree. It can be displayed using the global show(obj) method.

Example 4.18: FTA trace example

```
1 show(myFTA.trace(myTree))
```

- Method: ***reduce***()
  Computes and returns a (bottom-up) reduced version of this finite tree automaton. (See → 3.3.2.3 FTAOps reduce)

- Method: ***reduce_top_down***()
  Computes and returns a (top-down) reduced version of this finite tree automaton. (See → 3.3.2.3 FTAOps reduce)

- Method: ***reduce_full***()
  Computes and returns a bottom-up and top-down reduced version of this finite tree automaton. (See → 3.3.2.3 FTAOps reduce)

- Method: ***minimize***()
  Computes and returns a mimized version of this finite tree automaton. (See → 3.3.2.4 FTAOps minimize)

- Method: ***complete***()
  Computes and returns a completed version of this finite tree automaton. (See → 3.3.2.2 FTAOps complete)

- Method: ***complement***()
  Computes and returns the complement of this finite tree automaton. (See → 3.3.4.1 FTAOps complement)

- Method: ***determinize***()
  Computes and returns a deterministic version of this finite tree automaton. (See → 3.3.2.1 FTAOps determinize)

- Method: ***union***(fta)
  Computes the union of this finite tree automaton and the given finite tree automaton as argument. The resulting finite tree automaton is returned. (See → 3.3.4.2 FTAOps union). This method is equivalent to using the operator * on finite tree automata objects. (See → 4.3.5.2 FTA Operator *)

- Method: ***intersect***(fta)
  Computes the intersection of this finite tree automaton and the given finite tree automaton as argument. The resulting finite tree automaton is returned. (See → 3.3.4.3 FTAOps intersectionTD). This method is equivalent to using the operator + on finite tree automata objects. (See → 4.3.5.2 FTA Operator +)

- Method: ***subtract***(fta)
  Computes an finite tree automaton that recognizes the set difference of the language of this and the given finite tree automaton. (See → 3.3.4.4 FTAOps difference). This method is equivalent to using the operator - on finite tree automata objects. (See → 4.3.5.2 FTA Operator -)

- Method: ***empty_language***()
  Returns true if the language of this finite tree automaton is empty, false if not. (See → 3.1 FTAOps emptyLanguage)

- Method: ***finite_language***()
  Returns true if the language of this finite tree automaton is finite, false if not. (See → 3.1 FTAOps finiteLanguage)

- Method: ***is_complete***()
  Returns true if this FTA is complete, false if not. (See → 3.3.2.2 FTAOps checkComplete)

- Method: ***is_deterministic***()
  Returns true if this FTA is deterministic, false if not. (See → 3.3.2.1 FTAOps checkComplete)

- Method: ***same_language***(fta)
  Returns true if the language of this and the given finite tree automaton is the same. (See → 3.1 FTAOps sameLanguage). This method is equivalent to using the operator == on finite tree automata objects. (See → 4.3.5.2 FTA Operator ==)

- Method: ***subset_language***(fta)
  Returns true if the language of this finite tree automaton is a subset (or equal) of the language of the given finite tree automaton. (See → 3.1 FTAOps subsetLanguage). This method is equivalent to using the operator <= on finite tree automata objects. (See → 4.3.5.2 FTA Operator <=)

- Method: ***example_tree***([min_height])
  Returns a tree object that is recognized by this finite tree automaton. The minium height of the tree can be given as an argument. If min_height is not given the tree will be as small as possible. null is returned if the language of this finite tree automaton is empty or does only contains trees smaller than min_height. (See → 3.3.5.1 FTAOps constructTreeFrom)

- Method: ***rename_symbols***(){|name,arity| ...}
  Renames all symbols in this finite tree automaton. The method takes a block which is called for each symbol. The block result is converted to a string and used as the new symbol name. The resulting finite tree automaton is returned.

- Method: ***rename_states***(){|name| ...}
  Renames all states in this finite tree automaton. The method takes a block which is called for each state. The block result is converted to a string and used as the new state name. The resulting finite tree automaton is returned.

- Property: states
  An array of state names in this finite tree automaton.

- Property: final_states
  An array of final state names in this finite tree automaton.

- Operator: fta1 * fta2
  Equivalent to fta1.intersect(fta2)

- Operator: fta1 + fta2
  Equivalent to fta1.union(fta2)

- Operator: fta1 - fta2
  Equivalent to fta1.subtract(fta2)

- Operator: fta1 == fta2
  Equivalent to fta1.same_language(fta2)

- Operator: fta1 <= fta2
  Equivalent to fta1.subset_language(fta2)

- Operator: fta1 < fta2
  Returns true if the language of this FTA is a real subset (not equal) of the language of the given finite tree automaton. fta1 < fta2 is equivalent to ((fta1 <= fta2) && !(fta2 <= fta1)).

#### 4.3.5.3 Class: Tree

Represents a ranked tree for processing by finite tree automata Class Members:

- Method: ***new***(string)
  Creates a tree object from a string representation similar to the one used in the GUI tree editor. (See → 4.2.2.1 Tree)

- Method: ***new***(symbol_name, subtree1, subtree2,....)
  Creates a tree object from a name for the root symbol and an arbitary argument list of tree objects as subtrees. (Arrays of tree objects may also be included).

- Method: ***random***()
  Creates a random tree.

Object Members:

- Method: ***subtrees***()
  Returns an array of the subtrees of this tree.

- Method: ***singleton_fta***()
  Returns a finite tree automaton that only accepts this tree.

- Property: symbol
  Returns the name of the root symbol of this tree.

- Property: height
  Returns the height of this tree. The height is the number of levels in the tree. A tree without subtrees has height 1.

#### 4.3.5.4 Class: Homomorphism

Homomorphism represents the class of tree homomorphisms ((See → 3.7 Section Homomorphism)) Class Members:

- Method: ***new***(rule1, ..., ruleN)
  Creates a new Homomorphism object with the given rules. The rule definition is similar to one given in the GUI Homomorphism section. (See → 4.2.2.3 Homomorphism) Note: Multiple rules in a single argument are allowed if separated by linebreaks.

Object Members:

- Method: ***apply***(obj)
  Applies the Homomorphism on a given tree (See → 3.7.3.1 Homomorphism.apply) or Finite Tree Automaton (See → 3.7.3.2 Homomorphism.applyOnAutomaton), mapping the tree or finite tree automaton according to the homomorphism rules.
- Method: ***apply_inverse***(fta)
  Inversely applies the Homomorphism on a given finite tree automaton. (See → 3.7.3.3 Homomorphism.applyInverseOnAutomaton)
- Method: ***linear***()
  Returns true if the homomorphism is linear, false if not. (See → 3.2 Homomorphism.isLinear)

#### 4.3.5.5 Class: TreeTransducer

TreeTransducer represents the class of tree transducers (See → 3.9 Section Tree Tranducer) Class Members:

- Method: ***new***(rule1, ..., ruleN)
  Creates a new tree transducer object with the given rules. The rule definition is similar to one given in the GUI Tree Transducer section. (See → 4.2.2.4 Homomorphism) Note: Multiple rules in a single argument are allowed if separated by linebreaks.

Object Members:

- Method: ***apply***(tree)
  Applies the tree transducer to a given tree mapping the tree according to the tree transducer rules. (See → 3.9.2 TreeTransducer.doARun) The result is an array of the resuling trees. It is empty if the tree is not accepted by the tree transducer.

- Method: ***decide***(tree)
  Applies the TreeTransducer to a given tree. Returning true if the tree is accepted, false if not. (See → 3.9.3 TreeTransducer.decide)

- Method: ***to_fta***()
  Extracts the FTA that is part of the tree transducer. (See → 3.9 TreeTransducer.getFTAPart)

- Method: ***linear***()
  Returns true if the Homomorphism is linear, false if not. (See → 3.2 Homomorphism.isLinear)

#### 4.3.5.6 Class: HA

HA represents the class of hedge automata. Note: The names „HA“ and „HedgeAutomaton“ are equivalent.

Class Members:

- Method: ***new***(rule1, ..., ruleN)
  Creates a new HA object with the given rules. The rule definition is similar to one given in the GUI hedge automaton section. (See → 4.2.2.6 Hedge Automaton)

Object Members:

- Method: ***decide***(hedge)
  Takes a Hedge object and applies this automaton to it. It returns true if the hedge automaton accepts the given tree and false if not.

- Method: ***complement***(alphabet)
  Computes and returns the complement of this hedge automaton (inside the given alphabet) (See → 3.8.3.6 HAOps complementAlphabet). The Alphabet is given as an Array of symbol names.

- Method: ***union***(ha)
  Computes the union of this hedge automaton and the hedge automaton given as argument. The resulting hedge automaton is returned. (See → 3.8.3.7 HAOps union) This method is equivalent to using the operator * on hedge automata objects. (See → 4.3.5.6 HA Operator *)

- Method: ***intersect***(ha)
  Computes the intersection of this hedge automaton and the given hedge automaton as argument. The resulting hedge automaton is returned. (See → 3.8.3.8

HAOps intersectionBU) This method is equivalent to using the operator + on hedge automata objects. (See → 4.3.5.6 HA Operator +)

- Method: ***subtract***(ha)
  Computes a hedge automaton that recognizes the set difference of the language of this and the given hedge automaton. (See → 3.8.3.9 HAOps difference). This method is equivalent to using the operator - on hedge automata objects. (See → 4.3.5.6 HA Operator -)

- Method: ***empty_language***()
  Returns true if the language of this hedge automaton is empty, false if not. (See → 3.8.3.2 HAOps emptyLanguage)

- Method: ***finite_language***()
  Returns true if the language of this hedge automaton is finite, false if not. (See → 3.8.3.3 HAOps finiteLanguage)

- Method: ***same_language***(ha)
  Returns true if the language of this and the given hedge automaton is the same. (See → 3.8.3.5 HAOps sameLanguage). This method is equivalent to using the operator == on hedge automata objects. (See → 4.3.5.6 HA Operator ==)

- Method: ***subset_language***(ha)
  Returns true if the language of this hedge automaton is a subset (or equal) of the language of the given hedge automaton. (See → 3.8.3.4 HAOps subsetLanguage). This method is equivalent to using the operator <= on hedge automata objects. (See → 4.3.5.6 FTA Operator <=)

- Method: ***states***()
  An Array of state names in this hedge automaton.

- Method: ***final_states***()
  An Array of final state names in this hedge automaton.

- Operator: ha1 * ha2
  Equivalent to ha1.intersect(ha2).

- Operator: ha1 + ha2
  Equivalent to ha1.union(ha2).

- Operator: ha1 - ha2
  Equivalent to ha1.subtract(ha2).

- Operator: ha1 == ha2
  Equivalent to ha1.same_language(ha2).

- Operator: ha1 <= ha2
  Equivalent to ha1.subset_language(ha2).

- Operator: ha1 < ha2
  Returns true if the language of this hedge automaton is a real subset (not equal) of the language of the given hedge automaton. ha1 < ha2 is equivalent to ((ha1 <= ha2) && !(ha2 <= ha1)).

#### 4.3.5.7 Class: Hedge

Represents a hedge (unranked tree) for processing by HedgeAutomata Class Members:

- Method: ***new***(string)
  Creates a Hedge object from a string representation similar to the one used in the GUI hedge editor. (See → 4.2.2.5 Hedge)

- Method: ***new***(symbol_name, subtree1, subtree2,....)
  Creates a hedge object from a name for the root symbol and an arbitary argument list of hedge objects as subtrees (Arrays of Hedge objects may also be included).

- Method: ***random***()
  Creates a random Hedge

Object Members:

- Method: ***subtrees***()
  Returns an array of the subtrees of this Hedge.

- Method: ***to_tree***()
  Returns a ranked tree representation of this Hedge.

- Property: symbol
  Returns the name of the root symbol of this hedge.

- Property: height
  Returns the heigh of this hedge. The hight is the number of level in the hedge. A hedge without subtrees has height 1.

#### 4.3.5.8 Class: Null

Class for the „null“ singleton value Object Members:

- Method: ***to_i***()
  Returns 0
- Method: ***to_s***()
  Returns an empty string

#### 4.3.5.9 Class: Boolean

truth value class. Object Members:

- Method: ***to_i***()
  Returns 1 if this is true or 0 otherwise.
- Method: ***to_s***()
  Returns „true“ if this is true or „false“ otherwise.

#### 4.3.5.10 Class: Integer

32 Bit signed integer class. Object Members:

- Method: ***even***()
  Returns true if the Integer is even, false if not.
- Method: ***odd***()
  Returns true if the Integer is odd, false if not.
- Method: ***times***(){| i | ... }
  Calls the given block for each value from 0 (inclusive) to the value of this Integer object (exclusive). This method can be used for counting loop implementation similar to „`... for (int i = 0; i < n; i++)`“ in C or Java.

Example 4.19: Integer.times

```
1 3.times{|i| print(i)} #prints lines with 0 1 2
```

- Method: ***to_f***()
  Converts this Integer to a Float object with the same numeric value.
- Method: ***to_i***()
  Returns this.

- Operator: a < b
  True if a is less than b. Applicable for Integer and Float.
- Operator: a <= b
  True if a is less than or equal to b. Applicable for Integer and Float.
- Operator: a + b
  Arithmetical sum of a and b. Applicable for Integer and Float.
- Operator: a - b
  Arithmetical difference of a and b. Applicable for Integer and Float.
- Operator: a * b
  Arithmetical product of a and b. Applicable for Integer and Float.
- Operator: a / b
  Arithmetical division of a and b. Applicable for Integer and Float.
- Operator: a % b
  Modulo of a and b.
- Operator: a ** b
  Exponentiation of a and b: $a^b$

#### 4.3.5.11 Class: Float

Double precision floating point number class. Object Members:

- Method: ***to_f***()
  returns this
- Method: ***to_i***()
  Rounds this Float down to the next Integer value and returns it.
- Operator: a < b
  True if a is less than b. Applicable for Integer and Float.
- Operator: a <= b
  True if a is less than or equal to b. Applicable for Integer and Float.
- Operator: a + b
  Arithmetical sum of a and b. Applicable for Integer and Float.
- Operator: a - b
  Arithmetical difference of a and b. Applicable for Integer and Float.

- Operator: a * b
  Arithmetical product of a and b. Applicable for Integer and Float.
- Operator: a / b
  Arithmetical division of a and b. Applicable for Integer and Float.
- Operator: a ** b
  Exponentiation of a and b: $a^b$

#### 4.3.5.12 Class: Array

A simple 1-Dimentional list of arbitrary elements.
Note: Array can also contain other Arrays if multi-dimensonal fields are needed.
Class Members:

- Method: ***new***(elem1, elem2, ...)
  Creates an array object with the given objects as content. Elements are indexed starting at 0.

Object Members:

- Method: ***add***(elem1, elem2 ,...)
  Adds the given objects to the end of the array.
- Method: ***remove***(elem1,elem2,...)
  Removes the given objects from the array (if present).
- Method: ***remove_at***(idx1,idx2,...)
  Removes the elments at the given indices.
- Method: ***get***(idx)
  Returns the value at the given index.
- Method: ***set***(idx,elem)
  Overwrites the element at the given index with the given value.
- Method: ***insert***(idx,elem)
  Adds an element at the given index, old elements are moved aside.
- Method: ***size***()
  Returns the number of elements in this array.
- Method: ***contains***(elem)
  Returns true if the given object is somewhere in the array, false if not

- Method: ***empty***()
  Returns true if there are no elements in this array.

- Method: ***each***(){|item| ...}
  Iterates over all elements in the array, starting at index 0. It calls the given block once for each element in the Array.

- Method: ***each_rev***(){|item| ...}
  Iterates over all elements in the array in reverse order, starting at the index size()-1. It calls the given block once for each element in the array.

- Method: ***collect***(){|item| ...}
  Iterates over all elements in the array, calling the given block once for each element it constructs a new array consisting of the block return values for each element.

Example 4.20: Array.collect

```
1 arr = Array.new(1,3,8,10)
2 double = arr.collect{|item| item*2}
3 print(double) #double now contains an Array of
    [2,6,16,20]
```

- Method: ***select***(){|item| ...}
  Iterates over all elements in the Array, calling the given block once for each element. It constructs a new array consisting of the elements for which the block returnd true.

Example 4.21: Array.select: filtering even entries

```
1 arr = Array.new(1,3,8,10)
2 even = arr.select{|item| item.even}
3 print(double) #double now contains an Array of [8,10]
```

- Method: ***reject***(){|item| ...}
  Iterates over all elements in the Array, calling the given block once for each element. It constructs a new array consisting of the elements for which the block returns false.

Example 4.22: Array.reject: filtering odd entries

```
1 arr = Array.new(1,3,8,10)
2 even = arr.reject{|item| item.even}
3 print(double) #double now contains an Array of [1,3]
```

- Method: ***intersect***(otherArr)
  Returns an array consisting of the elements which are in both, the current and the given array. This method is equivalent to using the * operator on arrays. (See → 4.3.5.12 Array Operator *)

- Method: ***union***(otherArray)
  Returns an array consisting of the elements which are in any, the current or the given array. This method is equivalent to using the + operator on arrays. (See → 4.3.5.12 Array Operator +)

- Method: ***sort***()
  Returns a sorted version of this Array, the elements are sorted by the „<=“ operator definition of the elements. If not all elements in the array support this operator, an exception will occur and the execution is aborted. If the „<=“ definiton does not define a total order on the elements, the order of the elements in the result is undefined.

- Operator: arr1 + arr2
  Equivalent to calling arr1.union(arr2). (See → 4.3.5.12 Array.union)

- Operator: arr1 * arr2
  Equivalent to calling arr1.intersect(arr2). (See → 4.3.5.12 Array.intersect)

- Operator: arr1 - arr2
  Returns an array of all elements in arr1 that are not in arr2.

#### 4.3.5.13 Class: Hash

A map of key-value pairs. Values can be stored and retrieved using a key value as identifier. Keys and values can be of arbitary types. Null keys and null values are permitted. Class Members:

- Method: ***new***(key1,value1,key2,value2,...)
  Creates a new hash instance. Initial content can be given as an even number of arguments with alternating keys and values.

Object Members:

- Method: ***set***(key,value)
  Adds a new key value pair to the Hash.

- Method: ***remove***(key1,key2,...)
  Removes the given keys and their corresponding values from the hash map. A key that is not actually present in the Hash is ignored.

- Method: ***each***(){|key,value| ... }
  Iterates over all key value pairs in the hash calling the given block once for each pair. Key and value of the pair are passed to the block as arguments. Note: There is no fixed order of elements in the hash. The iteration order does not correspond to the insertion order and may even change between different calls.

- Method: ***each_key***(){|key| ... }
  Iterates over all keys in the hash calling the given block once for each pair. The key of the pair is passed to the block as an argument. Note: There is no fixed ordering of elements in the hash. The iteration order does not correspond to the insertion order and may even change between different calls

- Method: ***each_value***(){|value| ... }
  Iterates over all values in the hash calling the given block once for each pair. The value of the pair is passed to the block as an argument. Note: There is no fixed ordering of elements in the Hash. The iteration order does not correspond to the insertion order and may even change between different calls.

- Method: ***contains_key***(key)
  Returns true if the given key is present in the hash.

- Method: ***contains_value***(value)
  Returns true if the given value is present in the hash.

- Method: ***size***()
  Returns the number of key value pairs in the hash.

- Method: ***empty***()
  Returns true if there are no key value pairs in the hash.

#### 4.3.5.14 Class: Range

A range of integer numbers Class Members:

- Method: ***new***(min,max)
  Creates a new range with the given bounds (both inclusive). Alternatively, a range can be created with range operator: „min..max“.

Object Members:

- Method: ***each***(){|i| ... }
  Iterates over all Integer numbers in the range, calling the block once for each number, starting at min, counting upwards.

- Method: ***each_rev***(){ |i| ... }
  Iterates over all Integer numbers in the range, calling the block once for each number, starting at max, counting downwards.

- Method: ***to_a***()
  Creates an array of all numbers from min to max (inclusive) and returns it.

- Property: min
  The lower bound of the Range.

- Property: max
  The upper bound of the Range.

(P. Claves, S. Jarrous Holtrup)

# 5 Example of Use: XML Schemata

A typical example of using the tree and hedge automata library are XML-schemas. They are used to check whether an XML-document fits to a given schema, for example whether it is an html-schema. XML-documents are hedges, the used symbols have in general no arity and can enclose arbitrary many other symbols.

```
<doc>
  <title> list </title>
  <ul>
    <li> first </li>
    <li> second </li>
    <li> third </li>
  </ul>
</doc>
```

The hedge representation is `doc(title,ul(li,li,li))`.

## 5.1 Schema and Hedge Grammar

For the possibility to input schemes, the library supports hedge grammars, which can represent XML-schemes.

A schema is first of all a regular expression over an `UnrankedSymbol` subtype (lets call it `F`). Look at the following definition of regular Expressions:

$$R ::= \ eps \mid (R|R) \mid (R,R) \mid f(R) \mid M \mid R\{n,m\} \mid R^* \mid R^+ \mid R? \mid g$$

A grammar rule has the form $N \to f(R)$, where $R$ is a regular expression, $f$ a symbol of F and N a nonterminal smybol. $R\{n,m\}$ means a $n$ to $m$ repetition of $R$.

To put in a schema as grammar rules, use the class `HedgeGrammar<F>`. For the constructor

```
HedgeGrammar(Set<GrammarRule<F>> rule, Set<Nonterminal<F>>
    start)
```

some rules and nonterminals (as start symbols) must be produced. The nonterminals are grammar expressions, represented by `Nonterminal<F>`, which accepts a string as input parameter. One rule, a `GrammarRule<F>` contains a nonterminal, which forms the left side of the rule, a ranked symbol of type `F` which begins the right side and a regular expression, a socalled `GrammarExpression<F>`.

For each possible form, there is one class to instantiate a corresponding grammar rule:

- `Nonterminal<F>` for a nonterminal which is replaced with the help of other rules.
- `Terminal<F>` which wrapps a terminal symbol f into an expression.
- `Epsilon<F>` for an empty expression.
- `Concatenation<F>` for the concatenation of two regular expressions.
- `Alternation<F>` for the alternation of two regular expressions.
- `Range<F>` for the repetition of one regular expression. Two parameters limit the count of repetitions. A negative limit number means arbitrary many repititions.

Example 5.1: Defining a Hedge Grammar

```
1  // Let doc, title, ul, li be unranked symbols of type F
2  Nonterminal<F> n = new Nonterminal<F>("n");
3  Nonterminal<F> s = new Nonterminal<F>("s");
4
5  GrammarExpression<F> exp1 = new Concatenation<F>(title,s);
6  GrammarRule<F> rule1 = new GrammarRule<F>(n,doc,exp1);
7  GrammarExpression<F> exp2 = new Range<F>(li,0,-1);
8  GrammarRule<F> rule1 = new GrammarRule<F>(s,ul,exp2);
9
10 Set<GrammarRule<F>> rules = new HashSet <GrammarRule<F>>();
11 rules.add(rule1);
12 rules.add(rule2);
13
14 Set<Nonterminal<F>> start = new HashSet<Nonterminal<F>>();
15 start.add(n);
16
17 HedgeGrammar<F> grammar = new HedgeGrammar<F>(rules,start);
```

## 5.2 Convert Hedge Grammar to Hedge Automaton & Use It

After having defined a `HedgeGrammar` as above, it is possible to convert it into a hedge automaton with the method `getHA()`. Then with the class `HAOps`, which is described in chapter 3.8.3 all operations on hedges and hedge automata are possible.

For example, read in a document from xml as hedge (use the corresponding parser) and look if it satisfies the defined schema.

Example 5.2:

```
// Let xmlfile be a fitting file
Tree<UnrankedSymbol> hedge = parser.xml.XMLTreeParser.
    parseHedgeFromXML(xmlfile);
HedgeAutomaton<F,State> ha = grammar.getHA();
HAOps.decide(ha,hedge);
boolean finiteLanguage = HAOps.finiteLanguage(ha);
```

## 5.3 Automatical schema checker

Here we present an automatical schema checker:

Example 5.3: XML Schema checker example

```
public static void main(String[] args) throws
    ParserConfigurationException, SAXException, IOException,
    ParseException {
        if (args.length != 2) {System.err.println("Usage:
            xmlcheck schama_file xml_file"); System.exit(1);}
        File schemaFile = new File(args[0]);
        File xmlfile = new File(args[1]);
        Tree<UnrankedSymbol> hedge = parser.xml.XMLTreeParser
            .parseHedgeFromXML(xmlfile);
        FileReader fis = new FileReader(schemaFile);
        char[] cbuf = new char[10240];
        int len = fis.read(cbuf);
        EasyHedgeAutomaton ha = HedgeGrammarParser.
            parseString(new String(cbuf,0, len)).getHA();
        System.out.println("Accept: " + EasyHAOps.decide(ha,
            hedge));
}
```

(P. Claves, I. Thesing)

And now: Have fun with Lethal!

# Index

# Bibliography

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.