

ForestFIRE – Manual

R.G.W. Strolenberg L.G.W.A. Cleophas

February 15, 2009

Contents

1	Introduction	1
2	Installation	2
2.1	Compiling ForestFIRE	2
2.2	Using ForestFIRE	3
3	Trees and related structures	3
3.1	Symbols	3
3.2	Trees and nodes	4
4	Regular tree grammars & pattern sets	5
4.1	Regular tree grammars	5
4.2	Pattern sets	5
5	Tree automata	6
5.1	Nondeterministic tree automata	6
5.1.1	NTA constructions for RTGs	6
5.1.2	NTA constructions for pattern sets	8
5.2	Deterministic tree automata	8
5.2.1	DFRTA constructions for RTGs	9
5.2.2	DFRTA constructions for pattern sets	9
5.3	String path automata	10
5.3.1	AhoCorasick automata	10
5.3.2	String path DRFTAs	11
5.4	Bit vector based automata	11
6	Acceptors & matchers	12
6.1	Acceptors	12
6.2	Matchers	12

1 Introduction

FORESTFIRE is a JAVA toolkit for working with tree grammars, tree pattern sets and related algorithms like tree automata constructors, tree acceptors and tree pattern matchers.

A graphical user interface for use of the toolkit, called FIREWOOD, is available, but the toolkit may be used separately. This manual discusses how to use the different parts in the toolkit, from building simple trees to complex operations such as constructing automata from grammars or pattern sets (for use in acceptors and matchers). The installation of the FORESTFIRE toolkit is discussed as well. The user is expected to have some knowledge about regular tree grammars, tree automata, tree acceptance and tree pattern matching algorithms; see [6, 5] for details, including all of the tree acceptance and tree pattern matching algorithms and tree automata constructions implemented in the toolkit.

2 Installation

FORESTFIRE is implemented using JAVA, and distributed as part of the FIREWOOD source code package. The standard procedure is to compile the FORESTFIRE part and use the resulting FORESTFIRE JAR file that results from this compilation process. There are some prerequisites for this process:

1. Java JDK 5 or newer (<http://java.sun.com/>)
2. Ant 1.7 or newer (<http://ant.apache.org/>)

To compile the source code, first read Section 2.1. To use the compiled JAR file continue with Section 2.2.

2.1 Compiling ForestFIRE

Compiling the source code of FORESTFIRE can be done easily by using the ANT scripts contained in the source code package. Extracting this source code package results in the following directories:

- *ForestFIRE*, contains all FORESTFIRE source code.
- *FIREWood*, contains all FIREWOOD source code.
- *ForestFIREWood*, contains a set of subdirectories that contain compilation related items.
 - *ant* Contains all Ant scripts for building both FORESTFIRE and FIREWOOD.
 - *dist* Directory in which Ant stores the resulting JARS.

The single ANT script for compiling FORESTFIRE is called FF-BUILD.XML. This script can be started using the following command:

```
ant -f ff-build.xml
```

After the compilation process, the ForestFIREWood/dist directory will contain FORESTFIRE in the form of the FF.JAR file. This file can then be used for building an application that uses the toolkit. Details about using this JAR file can be found in the Section 2.2.

2.2 Using ForestFIRE

The compiled jar file (FF.JAR) can simply be included in a JAVA application using the toolkit's functionality. This is done by adding a class path reference to the JAR file when compiling a application that uses the toolkit. For example, for a JAVA program called MYPROGRAM:

```
javac -cp FF.jar MyProgram.java
java -cp FF.jar; MyProgram
```

This way the FORESTFIRE JAR file can be used in any JAVA program compatible with JAVA version 5 or newer.

3 Trees and related structures

The most basic structures of the FORESTFIRE toolkit are trees and related objects like nodes and symbols. This section will illustrate how a standard tree can be built using the toolkit. Since tree nodes are labeled by symbols, the use of symbol objects is discussed first. A subsequent section discusses how symbols are used in nodes and how nodes are used to build trees.

3.1 Symbols

FORESTFIRE offers two types of symbols, ranked symbols and unranked symbols. Ranked symbols are mostly used for terminals, while unranked symbols are used for nonterminals and variables, which are implicitly assumed to be of rank 0.

These symbols are constructed in the toolkit using a name (the label) and type (terminal, nonterminal or variable). Ranked symbols also require an integer rank. Code sample 1 provides an example construction of a terminal symbol of rank 2 and an unranked/rank 0 nonterminal symbol.

```
1 import forestfire.trees.*;
2
3 public class Symbols {
4
5     public static void main(String[] args) {
6
7         Symbol S = new Symbol("S", SymbolType.NonTerminal);
8         RankedSymbol a = new RankedSymbol("a", SymbolType.Terminal, 2);
9     }
10 }
```

Code sample 1: Creating symbols

These FORESTFIRE symbols can be found in trees, but also in for instance an alphabet or as the start symbol of a tree grammar. In the upcoming subsection we see example applications for tree nodes and alphabets.

3.2 Trees and nodes

A FORESTFIRE tree node is an object that contains a symbol (ranked or unranked) as well as references to the node's parent node, child nodes and to the tree in which it occurs. Initially, a node object is created using just a symbol object (see line 5 & 8 in Code sample 2).

These fresh nodes have no parent or child nodes and are not part of a tree. Building a tree starts with building the tree structure itself: construct a node with a rank larger than zero and add a child node for each non negative index smaller than the rank (see line 9 in Code sample 2). Assigning such a child node does not only create a reference from the parent node to the child node, but also updates the parent reference of the child node. In this way a complete tree structure can be built from root to frontier.

```
1 Symbol S = new Symbol("S", SymbolType.NonTerminal);
2 RankedSymbol a = new RankedSymbol("a", SymbolType.Terminal, 2);
3
4 RankedSymbol b = new RankedSymbol("b", SymbolType.Terminal, 1);
5 Node bNode = new Node(b);
6
7 RankedSymbol c = new RankedSymbol("c", SymbolType.Terminal, 0);
8 Node cNode = new Node(c);
9 bNode.children().set(0, cNode);
```

Code sample 2: Creating nodes with structure

The next step is wrapping the tree structure with a tree object. Such a tree object provides easy access to additional information like the leaf nodes of the tree. Creating such a tree is done in two steps.

First one has to create an alphabet that is a superset of the set of symbols that is used in the tree structure. The symbols need to be the same objects as used in the tree's nodes, not just symbol objects with the same labels as the symbol objects used in the tree's nodes. After creating the alphabet it is assigned to an empty tree object.

The second step is simply assigning the root node of the tree structure to the root reference in the tree object. This assignment also updates the tree reference of each node in the tree structure.

Code sample 3 provides an example construction of such a tree using the nodes of Code sample 2. Finally it shows that the tree is correctly built by writing its term notation to the output. In this example, this results in output b(c).

```
1 Alphabet alphabet = new Alphabet();
2 alphabet.add(b);
3 alphabet.add(c);
4
5 Tree tree = new Tree();
6 tree.setAlphabet(alphabet);
7 tree.setRoot(bNode);
8
9 System.out.println(tree.toString());
```

Code sample 3: Creating a tree

4 Regular tree grammars & pattern sets

As regular tree grammars and tree pattern sets play an important role in tree acceptance and tree pattern matching respectively, the toolkit contains representations of both concepts.

4.1 Regular tree grammars

Regular tree grammars are modeled in the toolkit using a broad collection of structures. Like a tree, a regular tree grammar has an alphabet, and as discussed earlier it also contains a start nonterminal. Furthermore, it includes a set of production rules.

These production rules, also called productions, are structures that consist of a nonterminal left hand side and a right hand side tree. The construction of such a production object is based on these two items (see line 1 of Code sample 4).

A complete grammar can then be constructed using an empty tree grammar object, setting the alphabet (containing all symbols used in the productions and the start symbol) and start symbol, and adding all the productions to its set of production rules (see lines 3–11 of Code sample 4).

```
1 Production prod = new Production(S, tree);
2
3 Alphabet combinedAlphabet = new Alphabet();
4 combinedAlphabet.add(S);
5 combinedAlphabet.add(b);
6 combinedAlphabet.add(c);
7
8 RegularTreeGrammar grammar = new RegularTreeGrammar();
9 grammar.setAlphabet(combinedAlphabet);
10 grammar.setStartSymbol(S);
11 grammar.productionRules.add(prod);
```

Code sample 4: Creating a regular tree grammar

These grammars can be used to construct automata which in turn can be used in acceptors that solve the acceptance problem. Details about the automaton constructions and acceptors can be found in Sections 5 and 6.

4.2 Pattern sets

Pattern sets are sets of tree patterns, i.e. of trees containing zero or more variables. The construction of these patterns is similar to the construction of normal trees. The pattern set itself is a new structure. This structure consists of a set of patterns and an alphabet. This alphabet must be a superset of the union of the alphabets of all patterns in the pattern set. Code sample 5 shows the straightforward construction of such a pattern set.

```
1 Tree pattern1 = new Tree();
2 //Construct pattern structure & set its alphabet ...
3
4 Tree pattern2 = new Tree();
5 //Construct pattern structure & set its alphabet ...
6
```

```

7 PatternSet ps = new PatternSet();
8
9 Alphabet union = new Alphabet();
10 union.addAll(pattern1.getAlphabet());
11 union.addAll(pattern2.getAlphabet());
12 ps.setAlphabet(union);
13
14 ps.add(pattern1);
15 ps.add(pattern2);

```

Code sample 5: Creating a pattern set

The constructed pattern set can be used to construct automata that can be used by tree pattern matchers, as detailed in Sections 5 and 6 below.

5 Tree automata

(Tree) automata play an important role in algorithms solving the tree acceptance and tree pattern matching problems. FORESTFIRE therefore implements a large number of (tree) automata types. The toolkit also contains a set of so called automaton generators which can be used to construct automata from tree grammars or tree pattern sets. The following types of automata are supported by the toolkit:

- Nondeterministic tree automata
- Deterministic tree automata
- String path automata
- Bit vector based (string path) automata

Each of these types can be constructed for a pattern set. For tree grammars, only nondeterministic tree automata and deterministic tree automata can currently be constructed. This section will discuss how these automata can be constructed for pattern sets and/or tree grammars using the generators provided by the toolkit.

More details on the automata constructions, the automata, and their role in various tree acceptance and tree pattern matching algorithms can be found in [5].

5.1 Nondeterministic tree automata

Nondeterministic tree automata can differ in direction (RF or FR), presence of ε -transitions, and item sets. The construction of these automata for tree grammars and for pattern sets are treated separately.

5.1.1 NTA constructions for RTGs

The construction of nondeterministic tree automata from tree grammars is provided by the NTAGENERATOR class. This class can be used to construct 8 different types of nondeterministic tree automata:

- NFRTA

- based on ALL-SUB item set, with ε -transitions
 - based on ALL-SUB item set, without ε -transitions
 - based on PROPER-N item set, without ε -transitions
 - based on PROPER-S item set, without ε -transitions
- NRFTA
 - based on ALL-SUB item set, with ε -transitions
 - based on ALL-SUB item set, without ε -transitions
 - based on PROPER-N item set, without ε -transitions
 - based on PROPER-S item set, without ε -transitions

As described earlier the construction depends on three parameters. The first one, direction, is controlled by passing either an empty NFRFTA or NRFTA object to the GENERATEAUTOMATON method of the NTAGENERATOR class. The other two parameters are also represented by a parameter in that method.

The second parameter determines the item set to be used. Items form the basis for nondeterministic tree automaton states, and the available item sets differ with respect to the *type of set* (ALL-SUB, PROPER-N or PROPER-S, based respectively on all subtrees of production right hand sides, just the proper subtrees among them and all nonterminals, or just the proper subtrees and the grammar's start symbol). For the purpose of nondeterministic automata constructions for tree grammars, the *type of item* is restricted to the SUBTREE type (for currently supported and future constructions for string path automata, different item types exist). There are so called item set providers for many combinations of set types and item types. These providers create an appropriate item set based on a tree grammar or pattern set. As the nondeterministic automata constructions for tree grammars all use SUBTREE type items, the number of different item set types is just three.

Code sample 6 contains some example constructions.

Note that it is possible to trigger the generator with a PROPER-N or PROPER-S item set while also enabling epsilon transitions. Such a call will not result in a useful automaton and may even cause an exception. The user should therefore take care to pass valid argument combinations only.

```

1 //Create empty automaton
2 NFRFTA<Subtree, AutomatonState<Subtree>> automaton =
3     new NFRFTA<Subtree, AutomatonState<Subtree>>();
4 //or
5 NRFTA<Subtree, AutomatonState<Subtree>> automaton =
6     new NRFTA<Subtree, AutomatonState<Subtree>>();
7
8 //Create item set
9 ISPAAllSubSubtree itemSetProvider = new ISPAAllSubSubtree(grammar);
10 //or
11 ISPProperNSubtree itemSetProvider = new ISPProperNSubtree(grammar);
12 //or
13 ISPProperSSubtree itemSetProvider = new ISPProperSSubtree(grammar);
14
15 //With epsilon transitions
16 boolean withEpsilonTransitions = true;
17 //or without

```

```

18 boolean withEpsilonTransitions = false;
19
20 //Construct generator
21 NTAGenerator<Subtree> generator = new NTAGenerator<Subtree>();
22 //Fill empty automaton by invoking generate method
23 generator.generateAutomaton(grammar, automaton,
24     itemSetProvider.getItemSet(), withEpsilonTransitions);
25 //Automaton ready to use

```

Code sample 6: Constructing a NTA from a RTG

Note the presence of some generic parameters. Automata are mostly parameterized with two parameters, one for the item type they use (in this case subtrees) and one for the type of automaton state they use (instances of a generic type AUTOMATONSTATE). The last parameter again is parameterized by the type of item stored in the match set of the state. This parameter should have the same value as the item type parameter of the automaton.

Such an item type parameter can also be found in the generator. The generator should use the same type as the automaton object provided to it, otherwise it will not function properly.

5.1.2 NTA constructions for pattern sets

Nondeterministic tree automata can be constructed from pattern sets in a similar way as from tree grammars. The item set type is restricted to ALL-SUB however and constructions that create ε -transitions are not supported. The only variability therefore is in direction of the constructed automata. This simplifies the choices for the construction, but the skeleton of the construction is very similar. The same NTAGENERATOR is used, but its GENERATEAUTOMATON method is called with a pattern set instead of a grammar and the list of parameters is shorter (see Code sample 7).

```

1 //Create empty automaton
2 NFRFTA<Subtree, AutomatonState<Subtree>> automaton =
3     new NFRFTA<Subtree, AutomatonState<Subtree>>();
4 //or
5 NRFTA<Subtree, AutomatonState<Subtree>> automaton =
6     new NRFTA<Subtree, AutomatonState<Subtree>>();
7
8 //Create item set
9 ISPAllSubSubtree itemSetProvider = new ISPAllSubSubtree(patternSet);
10
11 //Construct generator
12 NTAGenerator<Subtree> generator = new NTAGenerator<Subtree>();
13 //Fill empty automaton by invoking generate method
14 generator.generateAutomaton(patternSet, automaton, itemSetProvider.getItemSet());
15 //Automaton ready to use

```

Code sample 7: Constructing a NTA from a pattern set

5.2 Deterministic tree automata

In FORESTFIRE DFRTAs can be constructed for both regular tree grammars and pattern sets. FORESTFIRE also implements four special filtered DFRTAs that have reduced construction

time or memory usage (see [6] and [5, Chapters 5–6 and 8] for details):

- DFRTAFILTERSUBTREE for automata using subtree filtering,
- DFRTAFILTERSYMBOL for those using symbol filtering,
- DFRTAFILTERINDEX for those using index filtering, and
- DFRTAFILTERSYMBOLINDEX for those using both symbol and index filtering.

This results in five different types of automata (all descendants of ABSTRACTDFRTA) that can be constructed for both tree grammars and tree pattern sets.

5.2.1 DFRTA constructions for RTGs

The DFRTA construction for regular tree grammars has two degrees of variability. The first one is the earlier discussed filtering optimization. The second degree is the type of item set on which the construction is based. Like the item set used for nondeterministic tree automata (see Section 5.1.1) this set can differ in item type (SUBTREE or DOTTED RULE) and set characteristics (ALL-SUB, PROPER-N or PROPER-S).

The construction of such a DFRTA using FORESTFIRE is very similar to the construction of a NFRTA. The passed automaton determines the type of filtering and another parameter is used to pass the item set that should be used. Code sample 8 contains an example construction of a symbol filtered DFRTA using the PROPER-S item set of subtrees.

```
1 //Create empty automaton
2 DFRTAFilterSymbol<Subtree, AutomatonState<Subtree>> automaton =
3     new DFRTAFilterSymbol<Subtree, AutomatonState<Subtree>>();
4
5 //Create item set
6 ISPProperSSubtree itemSetProvider = new ISPProperSSubtree(grammar);
7
8 //Construct generator
9 DFRTAGenerator<Subtree> generator = new DFRTAGenerator<Subtree>();
10 //Fill empty automaton by invoking generate method
11 generator.generateAutomaton(grammar, automaton, itemSetProvider.getItemSet());
12 //Automaton ready to use
```

Code sample 8: Constructing a DFRTA from a RTG

5.2.2 DFRTA constructions for pattern sets

The construction of DFRTAs for pattern sets is similar to the construction for regular tree grammars. The same degrees of freedom exist: filter type and item set. The choices for item set are however different. The item type can be either SUBTREE or DOTTED-TREE, and the type of set is restricted to ALL-SUB, as PROPER-N and PROPER-S make no sense for pattern sets. Code sample 9 contains an example construction based on an unfiltered DFRTA and the ALL-SUB item set consisting of dotted trees.

```

1 //Create empty automaton
2 DFRTAStandard<DottedTree, AutomatonState<DottedTree>> automaton =
3     new DFRTAStandard<DottedTree, AutomatonState<DottedTree>>();
4
5 //Create item set
6 ISPAAllSubDottedTree itemSetProvider = new ISPAAllSubDottedTree(grammar);
7
8 //Construct generator
9 DFRTAGenerator<DottedTree> generator = new DFRTAGenerator<DottedTree>();
10 //Fill empty automaton by invoking generate method
11 generator.generateAutomaton(patternSet, automaton, itemSetProvider.getItemSet());
12 //Automaton ready to user

```

Code sample 9: Constructing a DFRTA from a pattern set

5.3 String path automata

String path based automata form another group of automata in FORESTFIRE. The current version of the toolkit only implements these automata for tree pattern matching, not for tree acceptance.

There are two basic different types of string path automata: Aho-Corasick string path automata and string path DRFTAs. Both are descendants of an ABSTRACTSPA class based on their commonalities: both use string paths, and use states of type STRINGPATHAUTOMATONSTATE to store such string paths.

The construction of the two types contains some differences and is therefore discussed separately in the upcoming two subsections.

5.3.1 AhoCorasick automata

Aho-Corasick string path automata in FORESTFIRE are essentially specific versions [2, 4] of the string pattern matching automata first described by Alfred V. Aho and Margaret J. Corasick [1]. These automata can be constructed in the toolkit using the ORIGINALACGENERATOR.

These Aho-Corasick automata can be further optimized because they only focus on trees. Original Aho-Corasick automata allow sequences of transitions on any possible symbol. In the context of tree string paths, one is only interested in sequences of alternating symbol names and branch numbers. This means that some transitions of the automata can be stripped [3, 4]. Such an optimized construction is performed by the STRINGPATHACGENERATOR.

Both these generators descend from the ABSTRACTACGENERATOR class and therefore have a similar interface. The construction of automata of both types is identical, but quite different from the construction of normal tree automata: the generator only takes a pattern set as input, without any additional degrees of freedom. It returns a complete automaton, unlike the constructions for normal TAs where the user should create an empty automaton for the generator. Code sample 10 provides an example construction of an Aho-Corasick automaton.

```

1 //Construct generator
2 AbstractACGenerator generator = new OriginalACGenerator();
3 //or

```

```

4 AbstractACGenerator generator = new StringPathACGenerator();
5
6 //Create automaton by invoking generate method
7 AhoCorasickAutomaton automaton = generator.generateAutomaton(patternSet);
8 //Automaton ready to use

```

Code sample 10: Constructing a AhoCorasick automaton from a pattern set

5.3.2 String path DRFTAs

String path DRFTAs differ from Aho-Corasick automata in structure and construction. Their structure is similar to normal tree automata with the exception that they use states corresponding to string paths instead of subtrees. The construction of such a SPDRFTA starts with the construction of an NRFTA based on string path states. This NRFTA is then converted to a deterministic DRFTA using a reachability based subset construction.

This construction process is hidden from the user. Like the construction for AhoCorasick automata there are no configuration options for the construction, apart from the pattern set used (see Code sample 11).

```

1 //Construct generator
2 SPDRFTAGenerator generator = new SPDRFTAGenerator();
3
4 //Create automaton by invoking generate method
5 SPDRFTA automaton = generator.generateAutomaton(patternSet);
6 //Automaton ready to use

```

Code sample 11: Constructing a SPDRFTA from a pattern set

5.4 Bit vector based automata

Besides the standard tree automata and string path based automata FORESTFIRE also implements a low level automaton type: the bit vector based automaton. This automaton type works with automaton states that contain bit vectors that represent match sets (BITVECTORAUTOMATONSTATE class). Operations in these automata mostly consist of bit vector operations like AND and OR.

Just like string path automata these automata can currently be constructed from pattern sets only. Code sample 12 contains an example of the construction.

```

1 //Construct generator
2 BVAGenerator generator = new BVAGenerator();
3
4 //Create automaton by invoking generate method
5 BitVectorAutomaton automaton = generator.generateAutomaton(patternSet);
6 //Automaton ready to use

```

Code sample 12: Constructing a bit vector based automaton from a pattern set

6 Acceptors & matchers

One of the main goals of FORESTFIRE's development was to get a collection of tree acceptance and tree pattern matching algorithms. The acceptors and matchers implemented use the automata of Section 5 to solve the tree acceptance or tree pattern matching problem. These acceptors and matchers are created based on such an automaton (created for a particular regular tree grammar or tree pattern set) and can then be used to solve the particular problem for a sequence of input trees. The upcoming two subsections discuss the acceptors and matchers and give details on their use.

6.1 Acceptors

The FORESTFIRE tree acceptors solve the tree acceptance problem using a tree automaton constructed from a regular tree grammar. This tree automaton is passed to an acceptor when invoking its constructor. There is a set of acceptors, each of which is intended for specific automata types:

- NRFACCEPTOR: NRFTAs
- NFRACCEPTOR: NFRTAs
- DFRACCEPTOR: all descendants of ABSTRACTDFRTA

All these acceptors implement the IACCEPTOR interface that defines an ACCEPT method. This method takes a tree and returns a boolean that indicates whether the tree is accepted or not. Code sample 13 provides an example of the construction and application of such an acceptor.

```
1 AbstractDFRTA automaton = /** any type of DFRTA constructed from a RTG */
2
3 //Construct acceptor
4 DFRAcceptor acceptor = new DFRAcceptor(automaton);
5
6 //Apply the acceptor
7 boolean result = acceptor.accept(tree);
```

Code sample 13: Constructing and using a tree acceptor

6.2 Matchers

Tree matchers, like tree acceptors, are constructed based on an automaton. These automata however should be automata constructed from a tree pattern set. As for acceptors there are different matchers for different types of automata:

- NRFMATCHER: NRFTAs
- NFRMATCHER: NFRTAs
- DFRMATCHER: all descendants of ABSTRACTDFRTA

- STANDARDSTRINGPATHMATCHER: all descendants of ABSTRACTSPA
- BITVECTORMATCHER: Bit vector automata

The constructor of such a matcher requires an appropriate automaton as well as the pattern set from which the automaton is constructed. The latter is required by the implementation, to store matching patterns in the nodes of the subject tree.

Like the acceptors all matchers implement a single interface: IMATCHER. This interface defines the MATCH method that can be invoked on a subject tree. This method will then apply the matching algorithm and store in each node the set of matching patterns (if there are any) using the matcher's annotation key (uniquely defined for each matcher, and retrievable using the interface's ANNOTATIONKEY method). Code sample 14 provides an example of the construction and application of the matcher followed by the retrieval of the matching patterns of the root node of the subject tree.

```

1 BitVectorMatcher automaton = /** a bit vector automaton constructed from a pattern set */
2
3 //Construct pattern matcher
4 BitVectorMatcher matcher = new BitVectorMatcher(automaton, patternSet);
5
6 //Apply the matcher
7 matcher.match(tree);
8
9 //Print matching patterns of root node
10 if (tree.getRoot().annotation().containsKey(matcher.annotationKey()))
11 {
12     Set<Tree> matches = tree.getRoot().annotation().get(matcher.annotationKey());
13     for (Tree matchingPattern: matches)
14         System.out.println(matchingPattern.toString());
15 }

```

Code sample 14: Constructing and using a tree pattern matcher

References

- [1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
- [2] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.
- [3] Loek Cleophas, Kees Hemerik, and Gerard Zwaan. A missing link in root-to-frontier tree pattern matching. In *Proceedings of the Prague Stringology Conference (PSC) 2005*, August 2005.
- [4] Loek Cleophas, Kees Hemerik, and Gerard Zwaan. Two related algorithms for root-to-frontier tree pattern matching. *International Journal of Foundations of Computer Science*, 17(6):1253–1272, December 2006.

- [5] Loek G. W. A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, April 2008.
- [6] Roger Strolenberg. *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2007.