MASTER

ForestFIRE and FIREWood
a toolkit and GUI for tree algorithms

Strolenberg, Roger

Award date:
2007

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER’S THESIS

ForestFIRE and FIREWood
A Toolkit & GUI for Tree Algorithms

by
Roger Strolenberg

Date of defense 1st of June 2007
Tutor ir. L.G.W.A. Cleophas, TU/e
Supervisor  dr.ir. C. Hemerik, TU/e
Abstract

Many fields in computer science use trees to represent hierarchical data. Parsing these trees and performing searches for patterns in them are well known problems. Tree parsing, for example, is a known issue in compilers, because compilers can parse intermediate representation trees to help in translating such trees into sequences of machine dependent instructions.

Loek Cleophas’ PhD research focuses on tree domain problems. He gathered, structured and classified a collection of tree parsing, matching and acceptance algorithms. This thesis discusses a toolkit and graphical user interface that were developed to create an environment to experiment with these algorithms and to collect information on their properties. These properties can be used to select the most promising algorithms for the instruction selection process in compilers.

The process of building the toolkit started with a study of existing tree toolkits. The knowledge gained from this study was used to construct the toolkit and user interface. The resulting toolkit consists of a subset of the algorithms described by Cleophas. Finally the toolkit and GUI were used to experiment with two types of algorithms. The description and results of these experiments are included at the end of this report.
Acknowledgments

I devote this section to all people who provided me with the knowledge, inspiration, and support throughout my Master project. Starting with my tutor Loek Cleophas, PhD student at the Technische Universiteit Eindhoven, who guided me through the project, provided the necessary theory and helped improving the toolkit and thesis. I also want to thank Kees Hemerik, my supervisor at TU/e, who provided valuable input and guarded the progress.

Next, I want to thank my close friends, without whom things would have been very different: My old schoolmates, Nicky, Frank, Tim, and Manon, for their encouragements and kind words; Isabelle, for the challenging games and the pleasant conversations during the breaks.

There are also some colleagues that may not be forgotten. Rudy, a colleague and friend, which worked on a similar Master project. We had very useful reviews of each other’s work and discussed daily programming curiosities. I also want to thank my colleagues Jan and Harald for removing the small language imperfections from my thesis. The project would also not be the same without the hilarious Wednesday afternoon lunches at the TU/e with Harald, Erik and Rudy.

And of course I want to thank my parents and my sister who always support me. Without them, I would not be where I am today.
**Acronyms**

This section contains the acronyms mentioned in the document. This list can be used as a quick reference.

**DAC** Data Access Component  
**DFRTA** Deterministic Frontier-to-Root Tree Automaton  
**DRFTA** Deterministic Root-to-Frontier Tree Automaton  
**FR** Frontier-to-Root  
**JRE** Java Runtime Environment  
**LCL** Lazarus Component Library  
**LHS** Left Hand Side  
**MDV** Multiple Data View  
**NFRTA** Nondeterministic Frontier-to-Root Tree Automaton  
**NRFTA** Nondeterministic Root-to-Frontier Tree Automaton  
**RF** Root-to-Frontier  
**RHS** Right Hand Side  
**RTG** Regular Tree Grammar  
**SDV** Single Data View  
**STF** Smallest Tree First  
**SWT** Standard Widget Toolkit  
**TA** Tree Automaton  
**TTF** Tallest Tree First
Contents

Abstract iii
Acknowledgments v
Acronyms vii
0 Introduction 1

1 Domain 3
1.1 Basic concepts ........................................ 3
1.1.1 Trees ........................................ 3
1.1.2 Tree languages .................................. 5
1.1.3 Regular tree grammars ............................. 5
1.1.4 Tree patterns .................................... 9
1.1.5 Finite tree automata ............................... 9
1.2 Problems of interest .................................. 12
1.3 Application Areas .................................... 13

2 Research into existing toolkits 15
2.1 ATerms ............................................. 15
2.2 BEG .............................................. 16
2.3 BURG ............................................. 17
2.4 iBURG ............................................. 18
2.5 Timbuk ............................................. 19
2.6 Treebag ............................................. 20
2.7 Twig .............................................. 21
2.8 Summary ........................................... 21

3 The Toolkit and GUI 25
3.1 ForestFIRE ......................................... 25
3.1.1 Trees ........................................... 26
3.1.2 Regular tree grammars ........................... 28
3.1.3 Tree patterns .................................... 32
3.1.4 Tree automata ................................... 33
3.2 FIREWood ......................................... 36
3.2.1 Architecture .................................... 36
3.2.2 Resulting user interface ......................... 39
3.3 Implementation details ............................... 42

4 Experiments 45
4.1 Used tree grammars .............................................. 45
  4.1.1 Thesis grammar ........................................ 46
  4.1.2 iBurg standard grammars ............................. 46
  4.1.3 Report of ten Eikelder ................................. 47
  4.1.4 Mono project grammars ................................. 47
4.2 Grammar transformation experiments ....................... 48
  4.2.1 RED-Z ................................................ 49
  4.2.2 RED-U .............................................. 50
  4.2.3 Influence of RED-Z/RED-U order ..................... 50
  4.2.4 RED-Z*/U* order with reuse ......................... 55
  4.2.5 RED-Z node selection strategies .................... 59
  4.2.6 Conclusion .......................................... 66
4.3 Automaton construction experiments ....................... 66
  4.3.1 Measurement techniques .............................. 67
  4.3.2 Automaton construction: general issues ............... 68
  4.3.3 Constructions of nondeterministic automata .......... 73
  4.3.4 Constructions of deterministic automata ............ 76
  4.3.5 Conclusion .......................................... 88
4.4 DFRTA based tree parsing experiments ..................... 90
  4.4.1 The parsing algorithm ................................ 90
  4.4.2 Automaton comparison ................................ 98

5 Conclusions .................................................. 101
  5.1 Results ................................................ 101
  5.2 Recommendations for future work ....................... 103
  5.3 Evaluation ........................................... 104

A MSc Assignment description .................................. 105
  A.1 Original Assignment description ....................... 105
  A.2 Additional data structure requirements ................ 111

B Formal definitions ........................................... 115
  B.1 Tree related definitions ............................... 115
  B.2 Tree grammar related definitions ..................... 115
  B.3 Tree automata related definitions ..................... 116

C ForestFIRE library ........................................... 119
  C.1 Basic collections ...................................... 119
    C.1.1 List ............................................. 119
    C.1.2 Dictionary ....................................... 120
    C.1.3 Set ............................................. 121
  C.2 Trees ................................................ 122
    C.2.1 Data structures .................................. 122
    C.2.2 Invariants ....................................... 124
    C.2.3 Related algorithms ............................... 126
  C.3 Regular tree grammars ................................ 127
    C.3.1 Data structures .................................. 127
0 Introduction

The Software Engineering and Technology (SET) expertise group at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven (TU/e) has as its main objective to create methods and supporting tools for development and maintenance of reliable software. Compilers, to which this master’s project is related, play an important role in supporting the software development process.

A few years ago Loek G.W.A. Cleophas started his PhD research that focuses on algorithms for the tree parsing, matching and acceptance problems. These algorithms can be used in many fields. One of these fields is compilers. To be more precise: the final part of the software compilation process needs to translate a generated parse tree to a sequence of instructions that can be executed on a machine. This instruction selection process can be done efficiently by using tree parsing algorithms.

Cleophas collected a large collection of algorithms in this area and described them in a general shape in his PhD thesis [Cle07]. This collection contains parsing, matching and acceptance algorithms, but also supporting algorithms and structures that are used to solve these problems efficiently.

The goal of this master’s project was to build a toolkit which contained (a subset of) these algorithms, especially tree grammar transformations, tree automaton construction algorithms and tree parsing algorithms that use tree automata. This toolkit should, in combination with a graphical user interface, provide an environment to experiment with these algorithms. This practical experience was to be used to gain insight into these algorithms, their properties, and their applicability in practice.

The starting point for this Master’s project was the original assignment description created by Cleophas and an additional list of requirements (see Appendix A). The assignment description provided three lists of requirements for the toolkit with different priorities. These requirements describe the domain concepts, like trees and grammars, and algorithms that needed to be implemented. The additional list of requirements contained specific operations for the data structures that implement the domain structures. During this project as many of these requirements as possible were implemented.

The design and implementation of the toolkit was preceded by an investigation of existing tree (grammar) toolkits. This study was used to see how these toolkits implement basic domain structures, like trees and tree grammars. This knowledge and the formal descriptions of the domain structures and related algorithms in Cleophas’ draft PhD thesis were then used to implement the toolkit.

Finally the implemented toolkit and GUI were used to perform a collection of experiments. These experiments focus on tree grammar transformations, tree automaton constructions and tree parsing. In summary, this lead to the following project phases:

- Studying toolkits using tree structures and algorithms related to them
- Designing and implementing the toolkit and GUI
Performing experiments with the toolkit and GUI

These phases were executed in a time frame of about eight months. We will describe each of these phases in detail in the upcoming chapters. However, before discussing the details of the assignment itself there will be an introduction to the domain, including trees, regular tree grammars, tree automata and the problems of tree acceptance, matching and parsing.
1 Domain

This chapter introduces the problem domain in which the toolkit operates: the domain of regular tree languages on ranked trees. Concepts of this domain that are important to the toolkit are introduced in this chapter. These concepts can also be found in more detail in Loek Cleophas’ draft PhD thesis [Cle07, Chapter 3].

After introducing these concepts a description is given of the domain specific problems that are described in [Cle07]: tree acceptance, tree parsing and tree pattern matching.

Finally an overview is given of the application areas for these three domain problems.

- Basic concepts
  - Trees
  - Regular tree languages
  - Regular tree grammars
  - Tree patterns
  - Tree automata
- Problems of interest
  - Tree acceptance
  - Tree pattern matching
  - Tree parsing
- Application areas

1.1 Basic concepts

Each of the upcoming sections will discuss one of the basic concepts in the domain of regular tree languages. These concepts are introduced to simplify the explanation of the domain specific problems and to provide the basic knowledge needed to read the rest of the report.

All of the concepts discussed in this section are more or less generalizations of similar concepts in the string domain [Lin01]. Tree grammars are for instance comparable to string grammars. This tree-string relation will therefore sometimes be used to clarify the tree concepts.

1.1.1 Trees

As described in the introduction, this report focuses on the domain of regular tree languages on ranked trees, to be more precise ranked, ordered, node labeled trees. Node labeled means that each node in a tree contains a symbol as label (see Figure 1.1).
The term \textit{ranked} implies that the symbol labeling a node determines the number of child nodes of the node. It is not allowed for a symbol to appear in nodes with different numbers of child nodes. The tree in Figure 1.1 for example is not a ranked tree, because there are two \textit{a}-nodes with a different number of child nodes. Removing one of the \textit{c}-nodes for the root \textit{a}-node will turn this tree into a ranked tree. Figure 1.2 shows this modified tree. Symbol \textit{a} in this tree has rank 2, \textit{b} rank 1 and \textit{c} is of rank 0.

Ordered addresses the fact that the child nodes of a parent node are ordered. A node with a symbol of rank \( n \) has \( n \) child nodes, where the leftmost child is considered the first child and the rightmost child the \( n^{th} \) child. Exchanging child nodes/subtrees results in a different tree if the exchanged items are not equal (see Figure 1.3).

This description of trees is quite informal. Their formal definition is based on the concept of \textit{tree domains} (see Definition B.1.1). A tree domain describes the paths to all nodes and thereby describes the bare structure of a tree (see Example 1.1.1).

\textbf{Example 1.1.1} This example shows the structure of the tree of Figure 1.2 as tree domain. This is the tree domain in set notation:

\[
\{ \varepsilon, 1, 2, 1 \cdot 1, 1 \cdot 2, 1 \cdot 1 \cdot 1 \}
\]
Each of the elements is a path to a node. The $\varepsilon$ refers to the root, 1 to the first child of the root, 2 to the second child of the root, $1 \cdot 1$ to the first child of the node that is first child of the root and so on. This tree domain represents the complete tree structure.

The tree labeling function (Definition B.1.2) is used as an addition to the paths to define the labels for the nodes and thereby creates a labeled tree. This labeling function, based on an alphabet of symbols, defines a relation for each path (see Example 1.1.2).

**Example 1.1.2** This example shows the tree domain of Example 1.1.1 with the tree labeling function. This is the original tree domain:

\[ \{ \varepsilon, 1, 1 \cdot 1, 1 \cdot 2, 1 \cdot 1 \cdot 1 \} \]

This is the labeling function from the tree domain to alphabet \{ a, b, c \} that turns the tree domain into the tree of Figure 1.2

\[ \{ (\varepsilon, a), (1, a), (2, c), (1 \cdot 1, b), (1 \cdot 2, c), (1 \cdot 1 \cdot 1, c) \} \]

Each element from the tree domain is now linked to a symbol from the alphabet. This therefore corresponds to a tree where each node has a label and in this case results in the tree depicted in Figure 1.2.

The representation in Example 1.1.2 does not oblige symbols to have a fixed rank. A ranked labeled tree can be constructed by using a so called ranked alphabet (Definition B.1.2) in the labeling function instead of a normal alphabet. A ranked alphabet defines a rank for each symbol. Using the alphabet \{ (a, 2), (b, 1), (c, 0) \} in Example 1.1.2 would turn it into a ranked tree.

The domain representation, as shown, can be used to represent trees. These definitions are often used in descriptions of algorithms related to trees. However, this report will mostly use the graphical representation or prefix notation of the graphical representation, because it is more compact and easier to read.

### 1.1.2 Tree languages

The concept tree language is comparable to the concept string language. A tree language is defined as a set of trees, where as a string language is defined as a set of strings. The concept of tree language is important, because it helps describing subsets of all possible trees. These languages facilitate problem descriptions like: Given a tree $t$ and a tree language $l$ determine whether $t \in l$.

These tree languages are mostly not directly represented by a set of trees, but are defined indirectly by a tree grammar (covered in Section 1.1.3). This is again similar to string languages and string grammars.

### 1.1.3 Regular tree grammars

A regular tree grammar (rtg) is a grammar that can be used to generate trees. These trees are generated in the same way that strings are generated by string grammars: applying productions to nonterminal symbols, starting with the start symbol. A regular tree grammar
is defined by the five tuple consisting of: two symbol alphabets $N$ and $\Sigma$ (terminal and nonterminal symbols), a ranking function $r$, a start symbol $S$, and a set of production rules $Prod$ (sometimes abbreviated as $P$). Each production rule is of the form $A \rightarrow \alpha$, where $A$, the left hand side (LHS), is a nonterminal and $\alpha$, the right hand side (RHS), is a tree. This tree can contain both terminal and nonterminal symbols, with the restriction that nonterminals can only be present at leaves (nonterminals have therefore always rank 0). Such a regular tree grammar is comparable to a regular string grammar in right regular form.

Definition B.2.1 of Appendix B contains the formal definition. The nonterminal and terminal alphabet together with the ranking function define which symbols can be used in the production rules. Example 1.1.3 contains an example tree grammar. This grammar will be used as running example throughout this section. Furthermore in this report we will use capitals for nonterminals and standard characters for terminals.

**Example 1.1.3** Let $G = (N, \Sigma, r, Prods, S)$ where

- $N = \{ S, B \}$
- $\Sigma = \{ a, b, c, d \}$
- $r = \{ (S, 0), (B, 0), (a, 2), (b, 1), (c, 0), (d, 0) \}$
- $Prods = \{ (1) \ S \rightarrow a \ , \ (2) \ S \rightarrow b \ , \ (3) \ S \rightarrow c \ , \ (4) \ B \rightarrow b \ , \ (5) \ B \rightarrow S \ , \ (6) \ B \rightarrow d \}$

Creating a tree with such a grammar starts at the nonterminal start symbol. A rule (whose LHS is this start symbol) is applied to the start symbol and thereby replaces the symbol with the right hand side of the production rule. This process is repeated until no nonterminals remain. An example tree construction can be seen in Example 1.1.4.

**Example 1.1.4** Using Example 1.1.3, Figure 1.4 shows a possible sequence of applications of productions starting at the start symbol. The first application is the application of the first rule that replaces start symbol $S$ with the RHS tree of rule number one. This is followed by the application of rule four and six, and finally results in the tree on the right.

![Figure 1.4: Example construction of tree based on the 1.1.3 grammar](image)

The tree constructed in Example 1.1.4 is one of many possible trees that can be constructed using the example grammar. The complete set of trees that can be constructed is the tree language that is produced by that grammar, which is denoted as $L_{rtg}(G)$ for a regular tree grammar $G$. 
Before proceeding with tree automata some detail is given about possible special shapes of regular tree grammars. Two topics are discussed: grammar characteristics, and reachability and productivity.

Grammar characteristics

There are two special characteristics for regular tree grammars described in [Cle07]. These characteristics describe the presence of production rules with distinctively shaped right hand sides.

The first characteristic describes whether a grammar may contain \((U_+)\) or may not contain \((U_-)\) production rules in which the right hand side is a single non terminal. These rules are called chain rules or unit productions. The grammar in Example 1.1.3 is for instance a \((U_+)\) grammar, because rule number five is a chain rule. Removing this rule would convert this grammar into a \((U_-)\) grammar (and also change the language produced by this grammar).

The second characteristic concentrates on the opposite case where the right hand sides contain large trees with so called Z-nodes. These Z-nodes are non-root nodes that contain terminals (for example the \(d\)-node in the first rule of the example grammar). A grammar that may contain rules with Z-nodes is called a \(Z_+\) grammar, and the term Z\(^-\) grammar is used for the opposite case.

The example grammar is a \(Z_+, U_+\) grammar. Converting this grammar to \(Z_-\), \(U_-\) results in a grammar that only contains production rules where the RHS has one level containing a single terminal \((A \rightarrow B)\) or has at most two levels with no terminal leaves \((A \rightarrow a(B_0, .., B_n))\). The conversions are done by applying the transformations known as RED-Z and RED-U (which are described in Chapter 4) that respectively convert a \(Z_+\) grammar into a \(Z_-\) grammar and a \(U_+\) grammar into a \(U_-\) grammar. These conversions and the effects of these conversions on the shape of the grammars were subject of the experiments and are discussed in detail in Chapter 4 of this report.

Reachability and productivity

The subject of reachability and productivity discusses the usability of production rules and symbols of a grammar. Some production rules and symbols in a grammar may be useless to construct trees. These useless items can be divided into two types: start unreachable rules or symbols, and unproductive rules or symbols.

Start reachable symbols are symbols which can be reached from the start symbol and start reachable production rules are rules with an LHS that is a start reachable symbol. A symbol \(B\) is called reachable from another symbol \(A\) if \(B\) can be reached by applying any sequence of production rules from \(A\) (see Definition B.2.2 and B.2.3). Symbols and rules which are not reachable from the start symbol are useless in a grammar. Example 1.1.5 shows a grammar that contains such start unreachable symbols and rules.

Example 1.1.5 The running example grammar of this chapter (Example 1.1.3) does not contain any unreachable symbols or rules. The concept of reachability can be nicely illustrated by adding symbols and production rules to the example grammar. The first items to be added are the nonterminal \(C\) and the following production rule:

\[
(7) \quad C \rightarrow b \\
| \\
\quad d
\]
Nonterminal $C$ is not present in any of the other six rules and therefore unreachable from the start symbol $S$. A consequence of this is that rule seven also is unreachable. Nonterminals that are only reachable via $C$ are also unreachable. This can be shown by adding the following two production rules and nonterminal $D$:

\[(8) \quad C \rightarrow b \quad \text{and} \quad (9) \quad D \rightarrow c\]

Nonterminal $D$ is present in the right hand side of rule eight, but it is still qualified as unreachable, because rule eight is unreachable.

These unreachable productions can be removed from the grammar because they do not contribute to the tree derivation process.

Unproductive rules and symbols have similar problems. A nonterminal is called productive if there exists a productive production rule in the grammar with that symbol as LHS, and a terminal is always considered productive (see Definition B.2.4). A rule is marked as productive if there is no unproductive symbol in its right hand side. The uselessness of unproductive symbols and rules is illustrated by Example 1.1.6.

**Example 1.1.6** This example provides an example of unproductive symbols and production rules. Similar to Example 1.1.5 the standard example grammar does not contain unproductive items. Unproductiveness is illustrated by adding additional rules to the grammar of Example 1.1.3. Only nonterminal $C$ and the following rule are needed to illustrate both types of unproductive items:

\[(7) \quad C \rightarrow b \quad \text{and} \quad (9) \quad D \rightarrow c\]

There is no way to obtain a tree from this nonterminal $D$, hence this nonterminal is marked as unproductive. If one would apply rule seven then the derivation would get stuck, because one can not replace the nonterminal $D$. This also explains why a rule with an unproductive nonterminal in its right hand side is called unproductive.

Unproductive rules and symbols are just as useless as unreachable items. One may therefore remove them from a grammar.

However, we have to place the remark that not only unproductive and unreachable symbols/rules are useless. An unproductive symbol can also make other symbols and production rules useless even when these are not unproductive or unreachable. Let us assume having a RHS of a production rule contains a unproductive symbol $X$. If this RHS contains a (reachable and productive) symbol $Y$ that is only present in that RHS and no other RHS, then $Y$ is also useless. The same holds for the production rules with $Y$ as LHS. This illustrates that not only unproductive and unreachable symbols/rules are useless.
1.1.4 Tree patterns

Tree patterns are in essence trees in which variable symbols may occur, with the restriction that these variables may only be present in the leaf nodes. These trees are called patterns because these variables represent all possible subtrees. Figure 1.5 shows such a tree pattern with a variable \( \nu \). This pattern matches the tree on the right hand side of that figure.

\[
\begin{array}{c}
\text{matches:} \\
\nu \\
\end{array}
\]

![Figure 1.5: Tree pattern that matches a tree](image)

Computing whether a pattern matches a subtree is one of the interesting problems related to tree patterns. One of the three major domain problems is a variant of this matching problem. This domain problem will be discussed in detail in Section 1.2.

1.1.5 Finite tree automata

Finite tree automata (TAs) play the same role in the domain of regular tree languages as finite string automata in the area of regular string languages. Tree automata, like string automata, can be divided into categories based on two criteria: determinism and processing direction. There are deterministic and nondeterministic tree automata which in turn can be divided into root-to-frontier (top-down) and frontier-to-root (bottom-up) automata.

Let us first discuss the general shape of a tree automaton, before addressing the differences among the four variants. The formal definition of a tree automaton describes the automaton as a 6-tuple (see Definition B.3.1). Informally, such a tree automaton, like a string automaton, consists of a set of states, a set of transitions, root/leaf accepting states and an alphabet.

The root accepting states and leaf accepting states are comparable to start states and accepting states of string automata depending on the processing direction of the TA. Root-to-frontier (RF) processing starts at the top, so the root accepting states are used as start states and the leaf accepting states as accepting states, where each leaf with a symbol \( x \) must match a leaf accepting state with an incoming \( x \) transition. This is the other way around for frontier-to-root (FR) automata.

However, the shape of the transitions in such tree automata is different compared to string automata. This is caused by the ranked alphabet of symbols that is used in this domain.

![Figure 1.6: Transitions in tree automata](image)
an element from the set $Q \times Q$, where $Q$ is the set of states. In a tree automaton a transition is not a relation between two states, but between one state and a set of states; To be more precise, the transition relation for symbol $a$ is defined as $Q \times Q^n$ where $n$ is $r(a)$, in other words the rank of $a$. This is depicted in Figure 1.6 where each transition consists of a single edge, a small circle and $n$ branches. There are $n$ branches because a transition on a symbol $a$ of rank $n$ is a transition that corresponds to a node with symbol $a$ that has $n$ child nodes. Figure 1.7 shows how a tree automaton with multiple transitions can look like.

Tree automata are in fact a generalization of string automata. A tree automaton that only contains transitions for symbol with rank one can be treated as a string automata. Removing the small circles where the transitions normally split for symbols with a rank larger than two will more or less turn the tree automaton into a string automaton.

The transitions in such an automaton can be undirected, root-to-frontier directed or frontier-to-root directed. An undirected automaton allows processing in both root-to-frontier and frontier-to-root direction, and therefore uses undirected transitions. The other two variants are used respectively in RF automata and FR automata. Figure 1.7 for instance shows an RF automaton. The arrows in this automata point from the parent to the child states. The arrows in an FR automaton point in the other direction. This report will focus on the directed automata, because they are more useful in practice.

There is also a difference when looking at determinism, as discussed in the first paragraph. This difference between nondeterministic and deterministic tree automata will be discussed in the upcoming two sections. These sections will also provide further details on RF and FR variants for both these deterministic and nondeterministic automata.

**Nondeterministic tree automata**

The phenomenon nondeterminism can be described in a similar way as for string automata. Assume one has an RF tree automaton. If this tree automaton is nondeterministic then this means that there can be an arbitrary number of transitions for a symbol $a$ (or for the $\varepsilon$-symbol) from that state. This can also be seen in Definition B.3.2, which defines, for a nondeterministic automaton (NRFTA), the relation for a symbol $a$ ($R_a$) of such an automaton as $R_a \in Q \rightarrow \mathcal{P}(Q^n)$, where $n$ is again the rank of symbol $a$. 

![Figure 1.7: Example (RF) tree automaton](image-url)
Figure 1.7 shows such an NRFTA. The \( a \)-transitions starting in state \( q_s \) provide an example of this nondeterminism, because there are two \( a \)-transitions that result in different state vectors ((\( q_2, q_3 \)) and (\( q_3, q_4 \))).

The same can be observed for an NFRTA if we would convert the RF variant in Figure 1.7 into an FR one. Our starting point is now the state vector (\( q_3, q_4 \)). We again look at the possible \( a \)-transitions and encounter two possible transitions, leading to state \( q_s \) and \( q_3 \). This illustrates how both RF and FR tree automata can be nondeterministic.

Another shared property with string automata is the possible presence of so called \( \varepsilon \)-transitions in nondeterministic tree automata. These are transitions that can be taken without reading a symbol. Nondeterministic tree automata that contain such \( \varepsilon \)-transitions are denoted by the terms \( \varepsilon \)NRFTA and \( \varepsilon \)NFRTA depending on the processing direction. Figure 1.8 shows an example of an \( \varepsilon \)NRFTA.

![Figure 1.8: Example TA with \( \varepsilon \)-transitions](image)

State \( q_s \) in Figure 1.8 has three outgoing \( \varepsilon \)-transitions, so from \( q_s \) it is possible to proceed to \( q_0 \), \( q_1 \) and \( q_2 \) without reading a symbol. A consequence of this is that an automaton containing \( \varepsilon \)-transitions is a nondeterministic automaton, because being in state \( q_s \) means also being in the states reachable from \( q_s \) by an \( \varepsilon \)-transition.

This nondeterminism has an effect on the usage of these automata. If one for example has an NRFTA and starts processing the root symbol of the tree in the root accepting state then this can result in multiple state vectors for the child nodes due to the nondeterminism. All these vectors have to be considered when proceeding and reading the child symbols. This makes the use of nondeterministic tree automata less efficient, similar to the use of nondeterministic string automata.

**Deterministic tree automata**

For each state in a deterministic RF automaton (DRFTA) there is at most one transition for each possible symbol and their exist no \( \varepsilon \)-transitions, implying that there is at most one transition for each symbol/state vector-combination for the FR variant of a deterministic automata (DFRTA).

This determinism can also be seen in the definition of these tree automata (see Definitions B.3.4 and B.3.5). For instance, the transitions \( R_a \) for a symbol \( a \) of rank \( n \) in DRFTA are defined as: \( R_a \in Q \rightarrow Q^n \), where the transition set in NRFTA for the same symbol \( a \) is defined.
as: \( R_0 \in Q \rightarrow \mathcal{P}(Q^n) \). The absence of this powerset shows that each transition for a certain symbol and state can only result in one vector for a DRFTA.

It is important to mention that not all tree automata have the same acceptance power. It is well known that the accepting power of the DRFTA is less than the accepting power of the other three automata (see e.g. [Cle07, Lemma 3.6.30]), so in practice most of the attention in the area of deterministic tree automata will go the DFRTAs.

### 1.2 Problems of interest

As described earlier the PhD research of Loek Cleophas focuses on solutions for three algorithmic problems in the area of regular tree languages. These problems are defined in more detail here.

Let us start with the tree acceptance problem (Definition 1.2.1). This is the most simple of the tree problems. The research in [Cle07] focused on solving this problem by constructing tree automata based on the \( rtg \) and processing the input tree with these automata. If an automaton accepts the tree, then the tree can be generated by the \( rtg \).

**Definition 1.2.1 Tree acceptance:** Given a regular tree grammar and an input tree, determine whether the input tree can be generated by the regular tree grammar i.e. is part of the language denoted by the regular tree grammar.

The second problem focuses on tree patterns instead of tree grammars (Definition 1.2.2).

**Definition 1.2.2 Tree pattern matching:** Given a finite, non-empty set of trees (the pattern set) and an input tree, find all occurrences of the patterns in the input tree.

What needs to be determined for all nodes in the input tree is which patterns match for that node. This problem can again be solved by using tree automata, by constructing a tree automaton from the pattern set and processing the input tree with the tree automaton. The states of the automaton correspond to certain patterns of the pattern set, so what needs to be done is storing which state can be assigned to which node in the tree in an accepting computation of the tree automata.

The final problem, called tree parsing (Definition 1.2.3), is an extension of the tree acceptance problem. However, for tree parsing one does not only want to know whether a tree can be produced by a grammar, but also how it can be produced. In other words, which production rule needs to be applied for each node in the input tree to obtain the complete tree.

**Definition 1.2.3 Tree parsing:** Given a regular tree grammar and an input tree, determine all parses of the input tree that can be generated by the regular tree grammar. A variation of this problem that is often used is to determine a parse that is optimal (with respect to some cost function).

This can again be solved by creating an automaton based on the \( rtg \) and processing the tree with this automaton. However, one has to do some extra registration during processing (as for the tree pattern matching problem) with the automaton to obtain and store the parse information.
This report will focus on algorithms related to these problems, especially to the tree acceptance and tree parsing problem. The algorithms of interest manipulate the shape of the input grammars (to remove useless items as discussed in Section 1.1.3) and construct automata from these grammars. The algorithms and the experiments with these algorithms are discussed in detail in Chapter 4.

### 1.3 Application Areas

There is a significant collection of application areas where these three problems in the regular tree language domain play a role:

- Code generation in compilers, particularly for instruction selection or optimization
- Term rewriting and unification
- Genetics, in particular DNA/RNA pattern matching
- XML document processing
- Type inference
- Protocol verification, particularly for cryptography and network protocols

Concepts in these domains can be translated to trees. For instance terms, which can be represented as a tree. However, our focus is on the topic of code generation.

In a compiler a string (source code) is translated into a parse tree by using lexical and syntactical analysis [ALSU07]. The resulting parse tree has to be translated into a sequence of machine instructions. This translation into machine instructions is called code generation/instruction selection. Figure 1.9 gives a simplified overview of this compilation process for an example expression.

\[ R_1 := c_1 + M[c_2 + R_2] \Rightarrow \quad \begin{array}{c} c_1 \\ \downarrow \\ M \\ c_2 \\ \downarrow \\ R_2 \end{array} \Rightarrow \quad \begin{array}{c} \text{MOV} \ # \ c_2, R_3 \\
\text{ADD} \ R_2, R_3 \\
\text{MOV} \ M[R_2], R_2 \\
\text{MOV} \ # \ c_1, R_1 \\
\text{ADD} \ R_1, R_2 \end{array} \]

Figure 1.9: Compilation process: an expression converted into tree, followed by a translation of the tree to a sequence of instructions.

The phase where the intermediate tree is translated into instructions can be described as a tree parsing problem. To do this one has to translate the instruction set of the target architecture into a tree grammar. Each instruction has to be converted into a single production. The LHS of the production rule has to correspond to the register in which the result is stored of the instruction, where the RHS of the created production rule must represent the semantics of the instruction. An example of an instruction set with corresponding production rules can be seen in Table 1.1.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Production Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV # c, Ri</td>
<td>Ri → c</td>
</tr>
<tr>
<td>MOV M[Rj], Ri</td>
<td>Ri → M Rj</td>
</tr>
<tr>
<td>ADD Ri, Rj</td>
<td>Ri → + Rj</td>
</tr>
</tbody>
</table>

Table 1.1: Three instructions with their corresponding production rules

The derived tree grammar and intermediate tree can then be used as input to an algorithm that solves the tree parsing problem. This algorithm will then compute which rules from the grammar have to be applied to obtain the source tree and therefore describe which instructions have to be selected to compute the expressions that correspond to (parts of) the tree.

The experiments on rtg transformations and ta constructions at the end of this report will present algorithms that can be used in solutions to these problems. Especially the ta constructions are useful for solving the instruction selection problem. The ta construction algorithms can construct automata from tree grammars and these resulting automata can be used to parse the tree and thereby compute which production rules correspond to which nodes of the subject trees. Each rule corresponds to a instruction, so the parse result can then used to determine which instructions need to be executed. More details about how such an automaton is used to parse a tree can be found in the final section of the chapter about the experiments.
2 Research into existing toolkits

The starting point of the assignment was to investigate existing toolkits in the area of regular tree grammars and related algorithms. The primary goal was to analyze the design of the toolkits and to find out if some parts of these designs could be reused in the design of our toolkit. Our toolkit contains a large subset of the domain concepts that are discussed in Chapter 1. The research into existing toolkits therefore focused on the data structures that are used for representing the main domain concepts: trees, their nodes, tree grammars and tree automata.

Before discussing these toolkits in detail I would like to start with the remark that most of the structures are quite straightforward. Trees, for instance, are defined by a root node, this root node points to its child nodes and in this way recursively defines its structure. Nodes have a terminal symbol and zero or more children, sometimes with a maximum of two due to the area of instruction selection applications where most of the instructions have at most two arguments.

Many of the systems also use other structures, such as tree automata and tree grammars. Most of the systems talk about trees, others treat them as terms. In the next sections I will give an overview of each system and discuss the details of the data structures that are used for representing trees, tree grammars and tree automata.

2.1 ATerms

ATerms [vdBdJKO00] is a C-library for efficient representation and storage of terms (which can be used to build tree-like structures). The library offers a representation for storing the trees/terms in memory during program execution and a separate binary format for storing trees/terms after execution or exchanging them between programs that use the ATerms library. Some of this storage optimization is achieved by storing subtrees with multiple occurrences only once in a tree (called subtree sharing). Pointers are then created for these subtrees to that single instance.

As mentioned, ATerms treats trees as terms. A term in ATerms can be an integer, real, string, function application, list, placeholder, blob or annotation pair. Trees can be constructed using the function application. The function application consists of a function symbol and an arbitrary number of terms as arguments. The function symbol is used to represent a node symbol and the arguments represent the subtrees. For example: ‘PLUS’(term₁, term₂,...,termₙ). The function application ‘PLUS’ represents a node labeled by a terminal with the name ‘PLUS’ and children term₁ to termₙ. The internal representation in the software is quite straightforward. Each of the seven types of terms is mapped onto a piece of memory. The function application is stored as the name of the function application (PLUS in the example) and a list of pointers to the terms contained in the function application. The details for all of these terms can be found in [vdBdJKO00, Section 3.4]. This way a tree can be represented by the ATerm that represents the root node of the tree. This is very similar to the tree structure sketched in the introduction above.
Our toolkit does not only use trees, but also tree grammars etc. Unfortunately, there is no built-in support for data structures like tree grammars, or other data structures and algorithms related to our research field. However, one could construct tree grammars and tree automata using optimized trees as provided by ATerms.

The ATerms tree structures also have some other interesting properties besides the efficient storage:

- ATerms can have terms with more than two children.
- Function applications can have a different number of arguments when they contain the same terminal, so the terminals have no fixed rank.
- Terms support annotation, which could be used in e.g. tree parsing.
- ATerms has automatic garbage collection for its terms, which simplifies the usage.

The annotation feature of the terms is very useful for our toolkit. It offers the possibility for algorithms and associated data structures to store node-specific information. And it is also possible to store multiple items in each node, because each block of information is stored with a specific key value as in a dictionary. These key values have to be used to set and get the data item for that key. This creates a nice way to store information for different purposes in a structured way. Such storage can be found in the other systems but not in such a structured way. The dictionary type annotation makes the ATerms nodes more flexible than nodes in other systems. The only disadvantage is that terms can only be annotated with other terms. This makes the ATerms annotation not very practical when it is used to store complex annotation data in parsing algorithms.

In summary, ATerms contains nice concepts. Some of these, like garbage collection and structured annotation, are interesting features. Other features like the optimized storage through subtree sharing are less interesting because they can result in difficulties when manipulating tree structures. When a subtree, for instance, has multiple occurrences in a tree (and is therefore shared) and is changed, then this subtree is changed everywhere. This is not always what a user wants. ATerms even calls such shared terms immutable. It is therefore not practical either to use ATerms in our toolkit or reuse this optimization approach.

2.2 BEG

BEG [ESL89] is a tool that generates a compiler back end (instruction selector/code generator) from a tree grammar that computes the minimal cost parse for an intermediate representation tree. The tree grammar specification is written in a language called BEGL. BEG uses such a specification to construct a Modula-2 back end that computes this minimal cost parse for an input tree.

The documentation of BEG gives a clear overview of the data structure it uses for representing nodes of the input tree. The structure is very similar to the structure described in the introduction: a tree node is represented by a structure that contains a string for the terminal symbol and a list of other nodes that represent the children.

BEG, like ATerms, has annotation possibilities in each node, but in BEG they are only targeted at the parsing algorithm. The parser stores in each node the matching production rules and their cost in two separate arrays. There is little or no attention paid to the description of the design, while there are other possibilities for storing this information. For instance,
BURG (discussed in Section 2.3) uses a slightly more complicated structure for storing parse data, which is more flexible and creates less pollution in the nodes. It could be that other designs are considered, but unfortunately no information about the design decisions can be found in the BEG description [ESL89].

Also, not much information can be found on how BEG represents tree grammars and their production rules. What can be retrieved from the code samples in [ESL89] is that the trees on the right hand side of the production rules are comparable to the normal BEG trees. Normal BEG tree nodes contain two arrays for storing match information. These arrays are in theory not necessary for the \texttt{rhs} of a production rule. However, one can not derive from the paper whether the same tree data structure is used or a stripped down variant.

In total, BEG uses straightforward data structures. The additional features, like the annotation option, are especially targeted at the parsing algorithm used by BEG. These features are therefore not applicable for our toolkit, because our goal is to be more flexible.

### 2.3 BURG

BURG [FHP92b] is, just like BEG, a code generator toolkit. BURG emits a tree parser, called BURM, as a C-program that computes the lowest cost parse of an input tree. The construction of this parser is done by using the BURS technique [Pro95]. BURG is used by providing a grammar and description of the tree structure to the toolkit. The BURS technique will then be used to create a tree automaton based parser that is used by BURM to parse each input tree.

Let us start by looking at the tree structures used by BURG. The first information about the data structures used by BURG was found in the BURG input file format. The input file, used for defining the grammar and shape of the input tree, showed that BURG uses conventional tree structures, where a tree is recursively defined starting at the root node. The user must define the basic shape of the input tree by defining (by using the C \#define directive) the four main fields of a node: the symbol field, the left and right child node, and integer annotation field.

The latter two items immediately introduce one of the shortcomings of structures used by the BURG toolkit. The restriction to symbols with a rank of at most 2 is not clarified, but it is probably related to the fact that BURG is used for instruction selection.

Another remarkable item is the integer field that is used for annotation. Other toolkits, like BEG, store quite a lot of data in the tree itself. BURG uses integer values that refer to states of the used BURS automata. Such an integer value can then be used to retrieve information related to this node. This information itself is stored in separate structures, like hash tables and arrays. This is a cleaner solution and more flexible solution than BEG’s solution.

Tree grammars are the next structures that need to be discussed. BURG reads the grammars that are specified in the input file and uses these to construct an automaton by applying the BURS technique. The used tree grammar data structures can in contrast to the trees not be found the output BURM c-file, but only in BURG itself. The paper describing BURG [FHP92b] did not contain any information about the data structures that were used for representing grammars, however the source code of BURG provided an impression of these structures. A grammar is represented by a set of lists: a list of nonterminals, a list of terminals, a list of production rules and some auxiliary lists such as a list of chain rules. The
2.4 iBURG

iBURG [FHP92a] is a variant of BURG, similar to BURG, constructs a tree parser from a tree grammar. iBURG uses the same input language as BURG, but uses a different parsing technique, because BURS was considered to be inflexible. This relation to BURG can also been seen in the data structures. iBURG uses roughly the same structures as BURG. The shape of the subject trees are completely the same compared to BURG, because iBURG uses the same type input file in which the user must define the basic tree structure.

The tree grammar data structure used by iBURG however, is more advanced than its BURG counterpart. The basis of the rules is as one would expect. It defines an LHS which is a nonterminal, an RHS which is a tree, and a cost value, but rules also have a pointer to another rule with a similar RHS (based on similar root or children) and a pointer to a chain rule with the same right hand side. Similar additions are made to the nonterminals, they also point to rules that use these nonterminals. These extras can simplify the parsing process, but it also creates a lot of extra initialization work because all these pointers have to be set.

Like BURG, iBURG generates an output C-program, for the specified grammar, that parses trees provided by the user. iBURG also uses a tree automaton to parse the input tree. However, in contrast to BURG, this is not a conventional tree automaton consisting of explicit states and transition. iBURG defines states in this C-program as integers and uses a function that uses a complicated generated switch statement instead of more conventional transition tables. This approach can however only be used when generating source code for a parser.
and not in a toolkit that constructs these automata in memory (which will be the case in the toolkit that is designed in this report).

Overall iBURG's basic structures for trees and grammars are as one would expect, but they also contain small extra ideas. Such an idea is the extra pointers to connect structures that for instance contain the same symbol on the same spot (e.g. two production rules with the same rhs root symbol). These pointers can become practical when computing matches based on these symbols.

### 2.5 Timbuk

Timbuk [GT01] is part of a collection of three tools. These three tools (Timbuk, Taml, Tabi) form a toolkit for term rewriting systems and tree automata. The Timbuk tool gives the user the ability to specify an automaton or a set of terms (for which an automaton that accepts these patterns can be constructed by the toolkit). The user can then test if his automaton/set of terms matches another automaton or set of terms if they are rewritten by some rewrite rules he has defined.

The Timbuk manual [GT] does not give much insight in the implementation of Timbuk. To find some more detailed information I took a look at the OCaml source code of Timbuk (version 2.0), where the definition of terms and automata can be found. However, there is not a lot of comment in the source code that describes the structures or why they are implemented the way they are.

Nevertheless, some things can be distilled from this source code. Timbuk defines terms/trees in another way than most of the other systems, due to the functional nature of the OCaml programming language. A term can have four shapes: variable (nonterminal), constant (leaf terminal), function (terminal with subterms) and 'special'. The function of this last one is not completely clear. The Timbuk manual does not discuss 'special' terms. The source code of the automata portion suggested that this symbol is used to represent automaton states as term, but the exact role is unclear.

Timbuk uses no tree grammars, because it focuses on automata constructed from tree patterns, instead of grammars. Examining the structure of these automata was not that easy, because only the source code was available, but the names used for variables give a good idea about the function of the different parts of the automaton. However, it was harder to get an idea how these different parts were used. The type is defined by:

```ocaml
type tree_automata = {
  alphabet: Alphabet_type.t;
  state_ops: Alphabet_type.t;
  states: State_set_type.t;
  final_states: State_set_type.t;
  prior: transition_table;
  transitions: transition_table
}
```

The role of parts like alphabet, states, final states and transitions are clear from their names. But for the other fields it is not so clear. The 'state_ops alphabet' is the collection of names used for the state. These are stored separately and have to be disjoint from the
2.6. Treebag

Treebag is a toolkit that enables a user to define tree grammars, algebras, transformations on trees and visualizations of trees. A user can define components of each of the four types. These components can then be connected, for instance, to visualize trees that are generated by a self-defined grammar and afterwards transformed by a self-defined transformation. These components can be defined in text files and loaded into the toolkit, where they can be connected with directed edges.

When examining the data structures I had to turn to the source code of the toolkit, because there was not much design/implementation information in the manual [Dre]. Tree nodes in Treebag are called terms and their structure is straightforward. There are no fields for algorithmic data, probably because the tree transformations should be loosely coupled to the trees.

Remarkable is that Treebag is the only system that uses terminals with ranks, and stores ranks in each node explicitly. This results in duplicate information. A dictionary-like data structure would be more efficient, if one would like to store it explicitly. A reason for this implementation could be that the retrieval of the rank is a bit faster, because it can be accessed directly in each node.

Trees in Treebag are created using tree grammars that are defined by the user. The system can do this randomly by applying random production rules or the user can do this by hand by applying the production rules he likes. The shape of these production rules is very similar to the production rules in the other systems. A production rule is defined by a structure containing a left hand side and a right hand side. The RHS is defined as a term, which can be any form of tree. The LHS is formed by one symbol only. Additionally there is a field called weight, which can be used to give a cost to each of the production rules (Treebag doesn’t oblige the user to specify this weight in his grammar definition).

It seems that the grammars in Treebag do not store a separate list of the terminals and nonterminals that are used in the production rules. This is probably done because these lists are not necessary to generate the trees.
Overall, Treebag’s structures for trees and grammars are quite similar to the structures in other systems and did not contain any striking differences. Yet like Timbuk, the absence of design documentation makes it hard to retrieve much information about the structures used and the design decisions made.

2.7 Twig

Twig is a tree manipulation language that enables a user to describe tree grammars and convert them, like BEG and BURG, into a code generator for a compiler. The paper [AGT89] contains the language to describe tree grammars (with costs for each production rule) and an algorithm that uses an special string automaton to compute the minimal cost parse for a given tree.

There is not much information on how the trees and grammars are represented in the Twig system. However, from the code samples in the paper one can derive that the tree nodes are defined as described in the introduction of this chapter, although the number of children is, like in other systems, limited to two. A Twig tree node also contains extra fields for storing algorithmic data for the parsing algorithm. Like in BEG, each node contains two arrays for storing subtree matches and their cost. There are still two extra fields that contain a state and a bit string. These are used to create a link between the nodes in the tree and the automaton that Twig uses for computing the match. So, in total there are four extra fields which are directly related to the matching algorithm used. Such a large collection of specific fields in each tree node would not be a good idea for a toolkit like ForestFIRE, which targets an expandable collection of algorithms. This would create a pollution of the tree nodes.

Unfortunately, information about the structures used for the tree grammars was not available in [AGT89]. It only contained a discussion about the high level conversion of the grammar into an automaton.

As already mentioned Twig constructs automata to check whether a grammar accepts a certain tree. These automata are not standard tree automata, but generalized variants of the Aho-Corasick string automaton [AC75]. However, these string automata are very similar to tree automata. Studying these automata in Twig was not easy, because only example automata are given without many details about internal representations of this structure. The only information about the structure was that it contains a function which computes the next state of an automaton for a given input state and the symbol that was being read.

Also, it was not possible to get a copy of the Twig system to examine it and to obtain more details about the data structures that Twig uses.

In summary, Twig tree and node structures are comparable to the structures of BEG and BURG. However, Twig shows that storing algorithmic data inside the tree can create a large number of extra fields in each node and is therefore not a very elegant solution. One can be brief about the structures used for the grammars and automata. The Twig description [AGT89] contains almost no information about these structures.

2.8 Summary

The systems studied provide some different ideas for implementing the domain concepts that were needed in our toolkit. As seen, many implementations of trees were available, but these
were not always well documented. Tree grammars and tree automata were not that often present in toolkits. Some of the ideas in these data structures, although not always directly inspired by these systems, can also be seen in our toolkit. Other features are deliberately not present in our data structures. Here I summarize the most remarkable features that were encountered in the three domain concepts that were studied in the systems.

**Trees**

As already mentioned in the introduction of this chapter, trees were mostly implemented in the same way. There were three aspects that caught our attention: storage of ranks, rank restriction, and annotation possibilities.

Remarkably, there was only one system (Treebag) that explicitly stores ranks in its trees. This system stores these ranks in each node of a tree. In our toolkit the choice was made to store the ranks explicitly. However, the storage is done in a different way by creating an explicitly ranked alphabet that is coupled to a tree instead of a rank that is stored in each node. The details of this implementation and the reasons for it can be found in Section 3.1.1.

The second aspect is the limitation of the rank. Many of the systems studied restrict their nodes to having only two child nodes. This means that all symbols in such a tree are at most of rank two. This is enough for most of the instruction selection applications, but also seems unnecessarily restrictive in many cases, because many algorithms used in these systems are also capable of handling symbols of a rank larger than two. Our implementation will not have this limitation and will therefore be more flexible than most of the systems studied here.

The first two observations described aspects from the existing systems that cannot be found in our toolkit. The annotation field is, unlike the other item, present in our toolkit (see Section 3.1.1), because it is practical in many algorithms. The implementation of this annotation structure is most comparable to the implementation of ATerms. The other systems provide annotation fields that are strongly related to the algorithms that use them. Our implementation, and also ATerms, focuses more on flexibility. This flexibility means that an arbitrary algorithm can use the annotation possibility provided by the nodes.

**Tree grammars**

Tree grammars, like trees, are mostly implemented in a straightforward manner by the studied systems. They consist of a collection of production rules that have a symbol as LHS and a tree as RHS. Strangely enough most systems do not wrap these grammars with classes. This is mostly due to the fact that they use a single grammar. This makes it less profitable to create such an additional structure. This situation is not applicable for our toolkit, because the goal is to be capable of handling more than one tree grammar at once.

The only non-standard feature encountered for tree grammars was the collection of references used by the iBURG toolkit. iBURG uses these collections to get fast access to symbols or rules with comparable symbols. This could also be used in our toolkit, but this was not done due to the additional implementation costs.

**Tree automata**

Finally there are the implementations of tree automata, but of the domain concepts I was interested in, this unfortunately was the one I encountered the least in the systems studied.
in this chapter. There was only one system (BURG) that uses tree automata and also docu-
mented it in a usable manner. This system described a way of indexing in transition tables
using states. A similar technique can also be found in the DFRTAs described in Section 4.3.
The primary goal of this research project was to design and build a new toolkit in the area of regular tree languages. This new toolkit, built with the knowledge gained by studying the existing toolkits, should facilitate experiments with algorithms related to the three domain problems mentioned in Section 1.2. The construction of this toolkit was shaped by two requirements:

- Contain an expandable collection of algorithms related to the domain problems (with focus on the tree grammar transformations, tree automaton construction algorithms and tree parsing algorithms, which were the subjects of the experiments)
- Provide a visual interface to use these algorithms in an easy way

What therefore was needed was a toolkit that implements all standard domain concepts (trees, tree grammars etc.), provides the possibility to implement algorithms that use these concepts and furthermore come with an user interface to experiment with these algorithms.

This resulted in a toolkit consisting of two parts. A library, called ForestFIRE, containing domain specific data structures and a collection of algorithms targeted at the domain problems (with special attention for the three specific groups of algorithms described above). The second part is FIREWood, a graphical user interface that provides the possibility to experiment with these algorithms and algorithms added in the future.

This chapter will provide the details of the design and construction of these two parts in two separate sections. The first section will discuss the heart of the system: the ForestFIRE library. The section describes which data structures and algorithms are implemented and how they relate to the concepts and algorithms that can be found in the domain. The second section provides details on the design of the graphical FIREWood application that should provide easy access to the algorithms inside the library. That section focuses on the major design decisions made during the construction of FIREWood. In a final section some details are given about the practical implementation process, including programming language chosen, platforms supported et cetera.

### 3.1 ForestFIRE

This section describes how the domain concepts of Chapter 1 are modeled and implemented in the ForestFIRE library. These concepts are divided into four sections: trees, regular tree grammars, tree patterns and tree automata.

Each of these sections discusses the concepts that are related to that topic. These concepts are translated to data structures/classes, where each concept can result in one or more classes. A tree for instance contains an alphabet, nodes etc. Each of the resulting classes is described in detail to clarify its role and design. Furthermore the relation between these different classes is visualized by a UML class diagram. Finally there are references to Appendix C,
which contains the precise interfaces of these classes. This appendix serves as a dictionary of all the data structures in *ForestFIRE*.

Additionally, each section discusses invariants for these classes. These invariants describe which properties in these classes have to be maintained such that the data structures do not violate properties that are important for the corresponding concept (a tree may for instance only use symbols that can be found in its alphabet).

The sections conclude with the algorithms that are related to the concept. For each of the concepts it is briefly described which algorithms are implemented. Additionally, Appendix C describes the classes and methods that implement these algorithms. The implementation of these algorithms is in many cases quite straightforward (e.g. counting chain rules), however others are more complicated. The details of these more complicated algorithms can be found in [Cle07]. The only complicated algorithms discussed in detail in this report are the algorithms that were subject of the experiments. The description of these algorithms can be found in Chapter 4.

Many of the data structures and algorithms discussed in this section can also be found in the original assignment description and additional list of requirements for the toolkit (see Appendix A).

### 3.1.1 Trees

This section describes all data structures related to the tree concept, like nodes, symbols, alphabets etc. These data structures are modeled in such a way that they fit the descriptions given in Chapter 1 and [Cle07].

A tree is represented by a collection of nodes with links that form the tree structure. The tree contains a reference to its root node and leaf nodes (to support easy RF and FR traversal). Each of the nodes in the tree structure contains a symbol and an unlimited sequence of child nodes. Using an unlimited sequence of nodes means that we do not restrict the number of child nodes to two as most of the existing toolkits do.

The tree also contains an alphabet. This alphabet contains the set of (ranked) symbols that can be used in the tree structure. Each node in this structure refers to its corresponding symbol in the alphabet. These references strongly resemble the tree function in the formal definition (see Definition B.1.2). None of the existing toolkits stored the rank with the symbol. However, we have chosen to do this, because it is efficient (rank is not stored in each node) and results in easy retrieval from each node.

These symbols can be used to represent terminals, nonterminals and variables. They contain a name (the label), type, and a rank in case it is a terminal. The type is used to denote whether it is a terminal, nonterminal or variable symbol. Nonterminals and variables are represented by using unranked symbols. The rank of these symbols are considered to be zero.

The choice for modeling a tree in this conventional way was made because this conventional shape is easier to implement and it provides an elegant way to point to subtrees by just using a reference to a node. Figure 3.1 shows an UML class diagram that shows how the tree, node and symbol classes are related.
Figure 3.1: UML Class diagram for tree and node related classes.

Figure 3.1 shows that the node structure has even more fields than described above. One field that is important to highlight is the annotation field. Many algorithms that use tree-like structures compute information for a certain node. These algorithms sometimes require space to store this information. The annotation dictionary is created to facilitate this. The dictionary allows an algorithm to store this information by using its own key(s).

Additionally a class was added to represent dotted trees. Dotted trees are used to point to a specific node within a tree. The representation of this dotted tree is similar to its formal 2-tuple definition, which defines a tree and points to a node. This node pointer is in the formal definition a path from the root to that node. However, we will define it as a direct pointer to that node, because this provides faster access to that node.

The details of all these data structures can be found in Section C.2. This section describes all classes separately and provides a description of the individual fields of these classes.

There are also some important invariants for these classes, because it is possible to construct illegal trees with these data structures if one does not follow certain rules. These rules are captured in class invariants, include for example:

- Each node that points to a tree as its parent tree, must also be part of this tree, and vice versa.
- A symbol used in a node of tree \( t \) must also be present in the alphabet of \( t \).
- If a node \( n_1 \) is in the list of child nodes of a node \( n_2 \), then \( n_1 \) must refer to \( n_2 \) as its parent.

All invariants and details about these invariants can be found in Section C.2.2. The data structures themselves offer facilities to avoid violation of these invariants. There exists for example a special method for assigning a node as root node. This method makes sure that all pointers to the parent tree in the tree structure are set to the corresponding tree.

However, these invariants can not all be handled by the library itself; the user of the library has to follow certain rules when using the data structures if he wants to make use of these properties. These rules can be found in the source code documentation. The invariants become important when a user starts manipulating the data structures in another way or manipulating the implementation of the data structures.

This combination of classes and invariants enables us to represent all concepts related to trees in a correct way.
3.1.2 Regular tree grammars

This section discusses the implementation of the data structures and algorithms related to regular tree grammars. We divide the discussion into a part for data structures and one for algorithms. Finally there is a third part that discusses the implementation of one algorithm in detail.

Data structures

The regular tree grammar definition for the toolkit is similar to its formal 5-tuple definition (see Definition B.2.1). This formal definition describes the following items:

- $N$, the nonterminal alphabet.
- $T$, the terminal alphabet.
- $r$, a ranking function for the terminals.
- $P$, the production rules or productions.
- $S$, the start symbol.

However, there are some differences. The regular tree grammar definition in the toolkit uses one combined alphabet for terminals and nonterminals, instead of two separate alphabets. This is done, because the symbols as defined contain a type field which indicates whether a symbol is a terminal or nonterminal, so there is no direct reason for separating them.

Another difference is the absence of an explicit ranking function. Ranks are stored within the symbol itself, as described in the previous section about trees or considered to be zero in case of an unranked symbol (nonterminal or variable).

The last two items of the formal definition are found in the toolkit implementation of a grammar: the `RegularTreeGrammar` class contains a start symbol and a set of production rules. These production rules are also shaped as in their formal definition. They contain a nonterminal $lhs$ and a tree $rhs$, but there is also a small addition, in the form of a cost field. This cost field is added because there exist parse algorithms that use a cost value to compute the minimum cost parse for trees. This field can then be used to store the cost for each production rule. A UML class diagram of these structures can be found in Figure 3.2.

![UML Class diagram for grammar related classes.](image)

Classes `Tree` and `Symbol` in Figure 3.2 were discussed in Section 3.1.1. As can be seen there is also an additional `DottedRule` class. The role of this class is similar to the role of the
DottedTree class. Instead of storing a reference of node inside a specific tree, it points to a node inside the RHS of a production rule. Dotted rules therefore contain a field that refers to a production rule and another that refers to a node. These dotted rules are not used in our experiments, however they can play a role in future parsing algorithms.

There are two important invariants for tree grammars. The first invariant describes that the left hand side symbol must be part of the alphabet of the grammar. The other describes that the alphabet of the right hand side tree must be a subset of the alphabet of the grammar itself. The formal definition of the new invariants can be found in Section C.3.2. Of course the general tree invariants discussed in Section 3.1.1 hold for the RHS as well.

Algorithms

The toolkit implements a large collection of algorithms related to tree grammars. The role of these algorithms varies. Some of them retrieve simple statistics like the number of nodes in the right hand sides of the production rules, but others perform more complicated operations like removing all chain rules or non-root terminal nodes in such a way that the language produced by that grammar is not changed.

All these algorithms are separated from the data structures themselves, to avoid garnishing these data structures with additional methods. This therefore results in a new collection of classes. The detailed interfaces of these classes can be found in Section C.3.3.

The algorithms are divided into different categories and each of these is discussed briefly in this section.

- Standard analysis
- Usability
- Grammar transformation
- Dotted rule and subtree retrieval

The first category provides algorithms that provide basic statistics of the grammar. This includes things like the number of nodes (mentioned earlier), but also the number of non-root terminal nodes. These algorithms only perform measurements and do not change the tree grammar.

The usability category also only measures statistics, but related to reachability and productivity of rules and symbols. The classes provide interfaces to retrieve both unproductive/unreachable/useless as well as productive/reachable/useful items.

The third category focusses on algorithms that manipulate the grammars. These so called transformations can be used to remove non-root terminal nodes, chain rules or useless production symbols and rules. This is realized by two new classes. The first class is targeted at removing chain rules or non-root terminal nodes and thereby converting the grammar into a grammar with the so called U− or Z− characteristic. The transformations related to usability are wrapped by the other class. These algorithms are easy to implement because removing useless items from a grammar will not influence the language produced by the grammar. This is however different for the transformations related to the U and Z characteristic. These transformations need to split RHS subtrees (to remove Z-nodes) or replace rules (to remove chain
rules) in a special way (described in Section 4.2.1) to obtain the desired characteristic without changing the language produced by the grammar. Experiments using these transformations are discussed in Chapter 4.

The final category contain algorithms that retrieve all subtrees that are present in the production rules of a tree grammar, where these substructures are called items. These items contain all unique subtrees in the RHS of the production rules and all unique LHS nonterminals. Figure 3.3 shows the item set of an example grammar. The items can retrieved as subtrees (in this case the subtrees are cloned into new trees) or as dotted rules where a dotted rule is created for each existing node in the production rules. This first technique is called subtree retrieval and the second one dotted tree retrieval. The difference between these two is that for subtree retrieval a tree is created for every unique item (so only one d instead of two for the example in Figure 3.3), while dotted tree retrieval results in dotted trees for all existing items even when their structure is the same. These collections of items will become useful when constructing automata from tree grammars. The tree automata that were used in the experiments make use of subtree retrieval to construct automata. Details about these automata can be found in Section 4.3.

\[ S \rightarrow a \]
\[ b \]
\[ d \]

Figure 3.3: Items of a tree grammar

Appendix C shows the related classes and interfaces of all these algorithms. Some of these algorithms are not very complex, like the retrieval of statistics and item sets. The other algorithms are more complex. Information about the transformation algorithms can be found in [Cle07, Section 3.3.3] for the usability transformations. The grammar transformation algorithms that remove chain rules and non-root terminal nodes are also discussed in detail in Section 4.2, because they were used in experiments. However, to get impression of the implementation of such an algorithm we will highlight the implementation of the transformation algorithm that removes chain rules in the next section.

**Example algorithm implementation**

The previous section gave a brief overview of the algorithms implemented by ForestFIRE in the area of tree grammars. This section will give an impression of how such an algorithm, the RED-U transformation algorithm, is implemented.

Many of the algorithms in ForestFIRE, like the RED-U transformation algorithm, originate from Cleophas’ PhD thesis. He described these algorithms using a more formal approach. These formal definitions were translated to the Java (programming language used for ForestFIRE) implementations that can be found in the toolkit. Many of these implementation have a large resemblance with their definition. This section will compare such a definition to its implementation to show this resemblance.

As described, the chosen algorithm to show this resemblance is the RED-U transformation step, that removes a single chain rule (also called unit production) from a grammar. Definition 3.1.1 shows the formal definition of this transformation step.
Definition 3.1.1 (Red-u, removing a single unit production). Let $G = (N, \Sigma, r, Prods, S)$ be an RTG with characteristic value $U_+$. Then there must be a production $A \rightarrow B \in Prods$ with $A, B \in N$. Then $G' = (N, \Sigma, r, Prods', S)$ where

$$Prods' = Prods \setminus \{A \rightarrow B\} \cup \left\{ \left. Set \right\{ C, \gamma : B \rightarrow^* C \land C \rightarrow \gamma \in Prod : A \rightarrow \gamma \right\} \wedge \gamma \notin N \right\}$$

is the resulting transformed grammar.

This definition describes that the chain rule $A \rightarrow B$ is removed and that new rules are added to replace the removed chain rule. These new rules $A \rightarrow \gamma$ are created for each RHS $\gamma$ of a rule $C \rightarrow \gamma$ that is reachable from $B$. The new rules make sure that $A$ can still produce the trees that were producible via the chain rule $A \rightarrow B$ (an example of this can be seen in Section 4.2.2).

Let us now look to Java implementation in Listing 3.1 which contains the method RemoveChainRule (also listed in Section C.3.3.3). The java code of this algorithm is, like the definition, divided into two parts. The first part (line 4) removes the chain rule that is provided to the method from the grammar. The remaining lines handle the creation of the rules that will replace this chain rule.

Line 9 – 13 compute, using Warshall’s algorithm, all nonterminals $C$ that can be reached from $B$ by using chain rules. Warshall’s algorithm computes the transitive closure of chain rules, this means that $B$ itself is not returned. Line 13 was therefore added to add $B$ itself to the set of nonterminals.

The remaining issue needed to be solved is finding all rules $C \rightarrow \gamma$ where $\gamma$ is no nonterminal. This can be seen in the final lines of code. These check for each rule if they have a closure nonterminal $C$ as LHS. If this is the case and the RHS is also no nonterminal, then a new rule is created using $A$ as LHS and $\gamma$ as RHS. This new rule is then added to a temporary collection of rules. These rules are at the end added to grammar. This has to be done afterwards, because else it will interfere with the ‘foreach’ loop.

```java
void RemoveChainRule(RegularTreeGrammar grammar, GrammarProductionRule rule) {
    //Remove chain rule from the set of production rules
    grammar.productionRules().remove(rule);

    //Compute nonterminal closure of chain rules
    //using Warshall and retrieve nonterminals
    //reachable from rule . rhs () . getRoot () . symbol ()
    Warshall closure = new Warshall(grammar);
    ArrayList<Symbol> closureSymbols =
        closure.getClosureSymbols(rule rhs().getRoot().symbol());

    //add RHS to obtain reflexive closure
    closureSymbols.add(rule rhs().getRoot().symbol());

    //Construct new rules that replace the chain rule
    ArrayList<GrammarProductionRule> newRules =
        new ArrayList<GrammarProductionRule>();
```
3.1. ForestFIRE

Chapter 3. The Toolkit and GUI

Listing 3.1: Java implementation of RED-U transformation step

This shows how closely related the formal definition and the implementation an algorithm are. This is similarly for the other algorithms, also for the algorithms related to the other domain concepts.

3.1.3 Tree patterns

Chapter 1 shows that a tree pattern is in essence a normal tree. A tree pattern is therefore implemented as a tree, with the exception that it not only contains terminals, but also variables. However, to represent patterns one can still use the normal tree classes, because the symbols used in nodes can be used to represent terminals, nonterminals and variables.

The only additional data structure needed in this area of tree patterns is a collection to wrap multiple patterns into a pattern set, because pattern sets are an important part of the tree pattern matching problem. Such a set contains a group of patterns and an alphabet that contains all symbols that may be used inside the patterns. These pattern sets can for instance be used to create automata that accept only patterns from this collection. These automata can be then be used to solve the tree pattern matching problem.

Figure 3.4 shows the existing tree class and the pattern collection class and how they relate.

Figure 3.4: UML Class diagram for pattern related classes.
All the invariants of the standard tree also apply to the tree pattern situation. The only extra invariant needed is that alphabets of the trees inside a pattern set are a subset of the alphabet of the pattern set. This is needed to ensure that the patterns do not contain symbols that are not present in the alphabet of the set (for details see Section C.4.2).

The only two algorithms for tree patterns that were implemented are for determining the collection of pattern subtrees respectively of dotted pattern trees (similar to the subtree and dotted rule retrieval for tree grammars). This first category retrieves subtrees as normal trees which results in a collection of cloned trees in which no duplicate tree structures can be found. The other category focusses on retrieval of subtree as dotted trees. In such a case dotted trees will be created for each node in the patterns. This means that there exist dotted rules that point to a similar tree structure if the pattern set contains multiple occurrences of a subtree.

### 3.1.4 Tree automata

This section discusses how concepts related to tree automata are implemented in the toolkit. At first the data structures are discussed that represent concepts like tree automata and tree automaton states. These data structures are accompanied by a collection of invariants just as in the previous sections. Finally all algorithms related to automata are discussed. These algorithms focus on the construction and usage of the automata.

**Data structures**

This section presents the classes needed for representing the different types of tree automata discussed in this report:

- Nondeterministic Root-to-Frontier Tree Automata (NRFTA)
- Deterministic Root-to-Frontier Tree Automata (DRFTA)
- Nondeterministic Frontier-to-Root Tree Automata (NFRTA)
- Deterministic Frontier-to-Root Tree Automata (DFRTA)

These automata were implemented with the help of two abstract classes (see Figure 3.5). One abstract class represents the general tree automaton and a specialized abstract DFRTA class supports different implementations of DFRTAs (the goal of this becomes clear in Chapter 4). The nondeterministic automata (with or without ε-transitions) inherit from the general abstract tree automaton class while DFRTAs inherit from the Abstract DFRTA class which in its turn inherits from the general abstract automaton class. DRFTAs are not implemented, because they have less acceptance power than the other three types, as mentioned in Chapter 1. However, they could be implemented in a similar way as the DFRTAs.

The general tree automaton class describes a standard tree automaton as the 5-tuple in the formal definition (see Definition B.3.1):

- $Q$, the state set.
- $(V, r)$, a ranked alphabet.
- $R$, the set of transition relations.
3.1. ForestFIRE Chapter 3. The Toolkit and GUI

- $Q_{ra}$, the root accepting states.
- $Q_{la}$, the leaf accepting states.

The state set, root accepting states and the ranked alphabet can be found in the Abstract-TreeAutomaton class as defined in Section C.5.1.1. However, this is different for the other items.

The set of transition relations is not defined by this class. The reason for this is, is that the optimal shape of these transition relations is different for RF and FR automata. These transitions are also not made accessible to the user in its raw form, because we want to hide the implementation from the user and give the user access to the transition relations via a NextState-method.

The same is done for adding transitions. A method is defined that allows the user to add a transition for a symbol relating a state and a vector of states. How these transitions are stored is hidden from the user, because there are many possible ways to do this: it depends on the direction of the automaton as mentioned before, but also on the type of construction used. The toolkit contains a collection of such implementations. The details of these implementations are discussed in Section 4.3.2, because they are closely related to different automaton constructions used in the experiments.

The other exception is the absence of leaf accepting states. These states can be distilled from the transition relations, because they have incoming transitions for symbols of rank 0 (as described in [Cle07, Remark 3.6.2]). The retrieval of the leaf accepting states is in practice implemented by defining a transition relation for each symbol of rank 0 from the empty tuple ‘()’ to the corresponding leaf accepting state. Calling the NextState-method with the empty tuple for each such symbol will return these states.

Figure 3.5 shows how all the tree automata classes relate to each other. Only a single implementation of the Abstractdftra class is shown. In practice there are five such implementations. These implementations all use different optimization techniques for the transition relations, but they do not change the external behavior of the automata. These are therefore not described here, but in Appendix C. Details about these optimizations can be found in Section 4.3.
Figure 3.5 also shows an *AbstractAutomatonState* class with three descendants. These classes are used to represent the states of an automaton. Each state refers to one (for non-deterministic automata) or a set (for deterministic automata) of items. Such an item or item set indicate which subtrees are matched when that state in the automaton is reached, and therefore indicates derivability (when the automaton is constructed from a grammar) or pattern matches.

The shape of this item sets depends on how these items are represented. As discussed in the section about tree grammars and tree patterns it is possible to store subtrees as subtrees, but also as dotted trees or dotted rules depending on the source (grammar or pattern). The *DottedTreeAutomatonState* is used by automata constructed from dotted trees (extracted from tree patterns), while the *DottedRuleAutomatonState* is used by automata constructed from dotted rules (obtained from a tree grammar). The *SubtreeAutomatonState* is used by automata constructed from subtrees extracted from either tree grammars or tree patterns.

Our tree automaton implementations focus on the automaton states based on subtrees and therefore use the *SubtreeAutomatonState* class.

There are also invariants for these data structures. The first one describes that the set of root accepting states is a subset of the state set of the automaton. The second invariant addresses that the states returned by the *NextState*-method are states that are present in the state set. These invariants are described in more detail in Section C.5.2.

**Algorithms**

The *ForestFIRE* library implements a large collection of algorithms related to tree automata. Most of these algorithms are targeted at the construction of these tree automata (from tree grammars) and the usage in tree acceptance and parsing.

The implemented construction algorithms can be used to construct (ε)NFRTAs, (ε)NRFTAs and DFRTAs from regular tree grammars. This is realized by only two algorithms: one for constructing nondeterministic automata and another for constructing the DFRTAs. The algorithm for nondeterministic automata can produce (ε)NFRTAs and (ε)NRFTAs, depending on the parameters used. The classes that implement these two algorithms and the methods that can be used to construct these automata can be found in Section C.5.3. These algorithms were also used in experiments and are therefore discussed in detail in Section 4.3.

The construction algorithms in turn make use of another group of algorithms called *item set providers*. These algorithms play an important role, in the construction of both nondeterministic and deterministic automata (see also Section 4.3), because they are involved in the state creation process. These item set providers retrieve a special set of subtrees from the grammar for which the automaton is constructed. The states of the automaton are then constructed based on these subtrees.

Another group of algorithms uses these automata to solve the tree acceptance problem. These are the so called acceptance algorithms. The toolkit implements three acceptance algorithms, one based on NFRTAs, another for NRFTAs and a final one for DFRTAs. These different algorithms are needed because the shape of the acceptance algorithms depends on the properties of the used automaton. An FR automaton processes trees in a bottom-up way, while an NRFTA works in the other direction. Acceptance algorithms for deterministic automata can for instance be optimized compared to the algorithms for nondeterministic ones, because they can make use of the determinism of the automaton.
Finally a parsing algorithm was implemented that solves the parsing problem using a DFRTA. This algorithm stores the parse information (which rule needs to be applied in which node) in the annotation fields of the nodes, such that one can find out which grammar rules need to be applied to derive the complete tree. There were also a set of experiments performed with this parsing algorithm in combination with different DFRTAs. These experiments are discussed in Section 4.4.

Details of the classes that implement all these algorithms can be found in Section C.5.3.

### 3.2 FIREWood

*FIREWood* is the application that provides a graphical user interface on top of the *ForestFIRE* library and supports easy access to tasks that make use of the different data structures and algorithms from the library. This includes e.g. visualizations of trees in prefix notation, but also providing access to tree acceptance and parsing algorithms.

In this project there was a special interest in tasks related to tree acceptance and parsing. However, there was a broad list of requirements for the *FIREWood* application:

- Support input/output of domain concepts (trees, tree grammars etc.) from/to files
- Provide an environment for experiments with RTG transformations, TA constructions and tree parsing algorithms
- Support the possibility to add new visualizations for new and existing concepts and related algorithm

This section discusses how these requirements were translated to an architecture and provides an overview of the main building blocks of the *FIREWood* application. The other part of this section contains a discussion of the resulting application and describes how the application can be used in practice to support the experiments in this domain. Details concerning the implementation (like programming language and graphical libraries used) can be found in Section 3.3.

#### 3.2.1 Architecture

The architecture of the *FIREWood* application describes how the requirements were translated into a user interface application and how the application functions on a high level.

The application should allow a user to load a collection of domain concepts like trees, tree grammars etc. Furthermore the user interface needed to provide a set of replaceable views that visualize the loaded concepts. These views provide information about the concept’s structure or enables the user to perform operations on it. A tree grammar is for instance one of the possible loaded concepts. If such a grammar is selected, the views provide the possibility to for instance inspect rules and remove chain rules.

This lead to three practical tasks that had to be implemented:

1. Reading and writing tree concepts from/to a file
2. Presenting loaded concepts to the user
3. Providing a (expandable) collection of views depending on the type of concept
Before discussing these tasks in detail we describe the basic concepts that the user can define in files and can be loaded in the *FIREWood* application: alphabet, tree, tree grammar, tree pattern and tree pattern collection. All these concepts play a role in the three domain problems (tree acceptance, tree parsing and tree pattern matching). The alphabet can mostly be found as a part of the other four concepts, however is was added because it is likely that the other concepts will use a similar alphabet if they are related to each other. This way a single alphabet definition can be used in definitions of multiple concepts.

The three tasks result in a *data access component*, *multiple data view*, and *single data view* components, which are detailed below.

**Data access component**

The data access component (DAC) is responsible for providing access to concepts stored in the *FIREWood* file format (for definition see Appendix D). The component can read files in this format and convert them into the corresponding *ForestFIRE* data structures and write them back to a file. This component therefore provides access to the user defined trees, tree grammars etc. All the other components access the data structures specified by the user through this component.

The DAC consists of three groups of data structures (see Figure 3.6). Firstly there are the file readers and writers. These readers convert a file into a so called *main data container*. This container contains all data structures read in smaller containers called *single data container*. These single data containers contain the *ForestFIRE* data structure(s) that correspond to a single concept. All these data structures are wrapped by a container to control the access to them. Adding for instance a new concept to a loaded collection of concepts can have consequences for the collection of alphabets used, because alphabets can be shared. The containers make sure that such internal dependencies are handled appropriately.

![Data Access Component Diagram](image)

Figure 3.6: Data access component that translated files into containers

The data access component is not restricted to the usage of the five concepts described above. The main data container is defined as a collection of abstract single data containers. An implementation of the abstract container is made for each concept that can be defined in the input files. New concepts can therefore be supported by creating a new implementation of the abstract single data container and extending the reader and writer classes with additional code to handle the new type of concept.

**Multiple data view**

The multiple data view component (MDV) makes the *ForestFIRE* data structures loaded by the DAC accessible to the user. How this access is realized is not explicitly defined by this
component. What the component does define is a way to communicate with the DAC. This is realized by an abstract class that defines a number of operations related to the complete collection of concepts. These operations have as role to react on changes in the main data container and to signal the application when a user selects a concept in the MDV. An implementation that inherits from this abstract class can then implement the visualization of these concept in the desired way (See Figure 3.7(a)). The idea behind this abstract class is that one can replace the visualization easily.

The FIREWood application also provides an implementation of a concrete MDV in the form of a tree view (see Figure 3.7(b)). This tree view contains all loaded concepts grouped by the different types of concepts. The user can inspect these concepts by clicking on them. The single data view component is then triggered to provide a visualization of the selected concept. This communication with the single data view component is provided by the abstract MDV class.

Single data view

The single data view component (SDV) has as goal to visualize a single concept and provide access to algorithms and statistics related to that concept. This single data view has also to support new visualizations when additional algorithms are added to the ForestFIRE library.

The design philosophy behind this component was similar to the philosophy behind the multiple data view component. The component consists of an abstract class and possible implementations of this abstract class. The abstract class provides an interface to respond to visualization requests generated by the MDV. The abstract class describes such a visualization only as an environment with a collection of subviews (mostly related to the different algorithms etc.). It depends on the implementation of the abstract SDV which subviews there are for a type of concept and how these are implemented. The concrete SDV presents these subviews to the user when the MDV signals that the user has selected a concept. Figure 3.8(a) shows how the SDV, MDV and DAC are related to each other.

As for the multiple data view component, an example implementation of the single data view was included, in the form of a tab control, where the subviews are the tab pages (see Figure 3.8(b)). When a concept is selected in the MDV (the tree view) a collection of tabs is opened in the tab control depending on the type of requested concept. Our standard implementation provides standard tabs for each type of concept. A creator of new algorithms
for a specific type of concept can then define new tab pages to access his algorithms. These pages are automatically shown to the user if he couples these tab pages to the corresponding concept type.

It is however also possible to use a completely different kind of control (or set of controls) for an implementation of the abstract single data view class.

![Diagram](image)

(a) SDV visualizing a single data container on request

(b) SDV implemented as tab control

Figure 3.8: Single Data View

The current toolkit contains a large collection of these tab pages, especially for the tree grammar concept. This concept was important because the toolkit was used as a platform for experiments using tree grammar transformations and tree automata constructions for tree grammars. The current version of *FIREWood* therefore contains tab pages that provide access to such algorithms. Examples of this can be seen in the next section, which contains a brief tour through the interface of *FIREWood* with the tree view MDV and tab control SDV.

### 3.2.2 Resulting user interface

The architecture of *FIREWood* described three main components, the DAC, MDV and SDV. All three components can be found physically in the user interface. The design already described that MDV and SDV respectively appear as a tree view and tab control. Let us start with an overview of the main window of the application to clarify how these three components are visually present:
When the application is started one can open a file through the menu bar on the top. The desired file can then be selected through a dialog. When the file is selected the DAC is triggered to parse it. After this the content of the file is visualized by the tree view on the left when the file contains no errors (if this is not the case then the loading is canceled and the errors are shown to the user). Clicking on one of the loaded concepts opens a collection of tab pages, where each of the tab pages has a different purpose. This can for instance be visualizing simple information, as the properties of the concept in Figure 3.9, but also providing access to the complicated parse algorithm.

Let us illustrate the possibilities of FIREWood by showing how the application was used in the experiments. Each of two experiments discussed a collection of algorithms. These algorithms can be found as tab pages in the tab control. Let us for instance look at the grammar transformations. These transformations remove the chain rules (\(u\)-rules) and non-root terminal nodes (\(z\)-nodes) that are discussed in Section 1.1.3. The tab page collection for tree grammars was therefore expanded with pages for analyzing and removing \(u\)-rules and \(z\)-nodes. Figure 3.10 shows the \(z/u\) analysis page for the grammar described in Example 1.1.3. This tab page contains a tree where \(u/z\)-items are marked with colored squares that indicate whether a rule/node is a chain rule or \(z\)-node. This tab page can therefore be used to study the grammar for the existence of special rules and nodes.
The row of tab pages in Figure 3.10 also contains three transformation tab pages (only two are visible in the figure). Two of these tab pages are devoted to the removal of $u$-rules and $z$-nodes where the other is targeted at the removal of unproductive and unreachable symbols and rules. As an example we look at the tab page that can be used to remove non-root terminal nodes by using the RED-Z reduction (this reduction is discussed in Section 4.2).
The red-z reduction page (see Figure 3.11) contains a tree view just like the analysis page does. However, this tree can now be used to mark non-root terminal nodes that should be removed. Additionally the user can select which red-z reduction style is used. After clicking the preview button, the two text boxes to the right will show which rules need to be removed and added to grammar to remove these z-nodes. These two boxes can be used to get some insight into the transformation effects (number of new rules, number of new nonterminals etc.). This kind of information was used in the chapter about the experiments to learn more about the characteristics of this red-z reduction.

Finally, one can use the apply button to remove/add the specified rules and store the new grammar as a new concept. This new grammar can then be inspected in more detail by selecting it in the MDV tree view at the left. The updated collection of concepts can afterwards also be stored in the original file or a new file.

This small example shows what these tab pages look like and how the user interface can contribute to the research process. Similar tab pages were created for the other transformations and the automaton construction algorithms. The experiments that were performed with the help of the toolkit will be discussed in Chapter 4.

### 3.3 Implementation details

The implementation of the toolkit had one important technical requirement: the toolkit & GUI should be usable on both the Microsoft Windows XP operation system and on the Apple Mac OS X operating system. This meant that a cross platform programming language was needed in combination with a cross platform graphics library. Research was therefore conducted to find combinations of programming languages and libraries that fulfil these requirements. This resulted in three candidates:

- Lazarus in combination with the Lazarus Component Library (LCL)
- Java in combination with the Standard Widget Toolkit (SWT)
- C++ in combination with a cross platform library (e.g. WxWidgets)

Lazarus [laz] was the language selected from these three candidates. The reason for this was the resemblance with Delphi that is used as the teaching language at the department of mathematics and computer science at the TU/e. Lazarus is an open source and cross platform implementation of Delphi and therefore a very promising candidate. However, a number of problems occurred with LCL library after the first weeks of implementation. The LCL tab control was burdened with a number of bugs. The LCL library was also not as cross platform as hoped, because controls behaved differently on the different platforms.

After a month of programming the choice was made to abandon Lazarus/LCL and switch to another candidate from the list. The choice was made to continue with Java and SWT. This choice was made because Java is considered to be a programming language more suitable for rapid development than C++ and it had the advantage of 'compile once, run anywhere’. After the choice was made the implementation continued and the already implemented part was translated to Java. Fortunately, this rewriting took only 2 weeks of time. The choice
of switching to another programming language and library turned out to be a good choice, because Java and SWT gave rise to significantly less problems than Lazarus with LCL.

The complete library and GUI application were implemented in Java version 1.5. In this implementation we mapped *ForestFIRE* and *FIREWood* to two Java packages, where each of the packages was divided into a collection of subpackages. This resulted in the following main hierarchy:

- **ForestFIRE**
  - Trees
  - Grammars
  - Patterns
  - Automata
- **FIREWood**
  - Data Access Component
  - Multiple Data View
  - Single Data View

The *ForestFIRE* packages follow the structure of the domain concepts that are described in Section 3.1. Each of the subpackages contains all the classes related to that concept (the *trees* package contains for instance the tree and node class). These subpackages can also contain other child packages that consist of classes that implement algorithms related to that concept. Table 3.1 shows the main packages listed above and the algorithmic subpackages with their corresponding Java name. This table shows, for instance, the *trees* subpackage and the *trees.algorithms* subpackage.

The *FIREWood* package is divided into set subpackages from which three directly correspond to the main components of the *FIREWood* application, where the subpackages of the MDV and SDV contain both the presented abstract classes and the example implementations.

Table 3.1 provides, next to the names of the (sub)packages, an impression of their dimension. The toolkit and GUI together contain around 10,000 lines of code and 100 classes. The table lists the number of classes and lines of code for each subpackage in *ForestFIRE* and *FIREWood*.

The table also contains some subpackages, like base, controls and extensions, that are not mentioned earlier. These packages contain classes that support the main *ForestFIRE* and *FIREWood* subpackages. The base subpackage, for example, contains the special collection types that are used in the toolkit (see Section C.1).
Table 3.1: Code statistics of toolkit and GUI

The *ForestFIRE* package (and its subpackages) can be used independently of the *FIRE-Wood* package, so it is for instance possible to use it in other Java applications. The *FIRE-Wood* application was also combined with the *ForestFIRE* library in a single JAR-archive. This executable archive was used throughout this project to perform the needed experiments.

All the packages and classes also contain JavaDoc documentation. This documentation simplifies the extension of both the toolkit and the GUI. We therefore realized the goal of building a toolkit and GUI that not only helped performing the experiments, but that can also be easily extended to support new tree algorithms.
4 Experiments

This chapter discusses the experiments that were carried out with the ForestFIRE/FIREWood toolkit. These experiments focussed on three subjects:

- Tree grammar transformations
- Tree automaton constructions
- Tree parsing

The tree grammar transformation experiments measure the influence of the grammar transformations that remove the chain rules and non-root terminal nodes. The grammar transformations were subject of the experiments because certain tree automaton constructions focus on grammars with the $Z_-$ or/and $U_-$ characteristic. The transformations realize these characteristics by removing existing rules from the grammar and adding new rules and non-terminals to it. It was therefore interesting to discover how the $Z_-$ and $U_-$ characteristic can be obtained while introducing as few new rules and nonterminals as possible.

The second set of experiments measures the characteristics of tree automaton constructions that are used to construct tree automata from tree grammars. The characteristics measured are running time and the size of the resulting automata. These experiments were carried out because tree automata constructed from tree grammars are very useful for solving the tree acceptance and tree parsing problem. The results of these experiments were therefore used to determine which construction algorithm is the most promising for building these automata.

The final collection of experiments focuses on an algorithm that uses deterministic tree automata to solve the tree parsing problem. The goal of these experiments was to find out which type of deterministic automaton discussed in the automaton construction experiments is most efficient for solving the parsing problem.

Before discussing these experiments it is important to provide some background on the tree grammars that are used in these experiments. Section 4.1 introduces these grammars. The remaining two sections, 4.2 and 4.3, will discuss the two categories of experiments.

4.1 Used tree grammars

The chosen grammars came from four sources: the draft PhD thesis of Loek Cleophas [Cle07], iBurg software [iBU], a report by Huub ten Eikelder [tE89] and the Burg-files of the Mono project. These grammars were either grammars related to instruction selection or small grammars that were easy to inspect during the experiments. In the next sections we will discuss each of these grammars in more detail. The details presented include basic statistics like number of nodes, rules, terminals and nonterminals, but also more advanced statistics like number of non-root terminal nodes and chain rules.
4.1. Thesis grammar

The first grammar was taken from the draft PhD thesis of Loek Cleophas. This grammar, 'Example 6.0.3', has the following characteristics:

<table>
<thead>
<tr>
<th>example603.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>6</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>12</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>2</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>4</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>3</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>1</td>
</tr>
</tbody>
</table>

4.1.2 iBurg standard grammars

Two grammars were taken from the collection of iBurg-files that is bundled with the iBurg software [iBU]. These grammars, 'Sample 4' and 'Sample 5', have the following characteristics:

<table>
<thead>
<tr>
<th>sample4.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>12</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>20</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>5</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>7</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>1</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sample5.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>10</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>23</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>4</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>5</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>3</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>2</td>
</tr>
</tbody>
</table>

These two grammars are related to instruction selection in compilers where intermediate representation trees are translated to instructions. The rules in for instance 'Sample 5' are rules that can be expected in a grammar that is used for instruction selection:

\[
\begin{align*}
\text{reg} & \rightarrow \text{con} \\
\text{loc} & \rightarrow \text{reg} \\
\text{reg} & \rightarrow \text{PLUS} \\
\text{reg} & \rightarrow \text{PLUS} \\
\text{MEM} & \rightarrow \text{reg} \\
\text{loc} & \rightarrow \text{reg}
\end{align*}
\]

Assume \text{reg} stands for register, \text{con} for constant and \text{loc} for memory location, then it can easily be seen that these rules are common instruction selection rules. These two grammars
are small, but representative of how the transformations perform on production rules that are likely to be found in the area of instruction selection.

### 4.1.3 Report of ten Eikelder

The report of ten Eikelder [tE89] describes the implementation of the bottom-up acceptor that is discussed by C. Hemerik and J. P. Katoen in [HK89]. This bottom-up acceptor is based on the use of DFRTAs. The report by ten Eikelder contained a grammar targeted at the instruction set of the 68000 architecture. This grammar was selected by us because one of the constructions considered in this report produces the same DFRTAs discussed by ten Eikelder.

The characteristics of this grammar are:

<table>
<thead>
<tr>
<th>68000.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>33</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>104</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>3</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>12</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>30</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>1</td>
</tr>
</tbody>
</table>

### 4.1.4 Mono project grammars

Finally three larger grammars were added, originating from the Mono Project [mon]. The Mono Project is a project that provides an open source implementation of the .NET Framework. These Mono grammars are used for translating parse trees that consist of machine independent commands into machine dependent code. Grammars for the following architectures were used: X86, IA64 and Sparc. The characteristics of these grammars are:

<table>
<thead>
<tr>
<th>mono-x86.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>505</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>1412</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>8</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>269</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>371</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mono-ia64.ini</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
<td>432</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>1064</td>
</tr>
<tr>
<td>Number of nonterminals</td>
<td>8</td>
</tr>
<tr>
<td>Number of terminals</td>
<td>262</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
<td>221</td>
</tr>
<tr>
<td>Number of chain rules</td>
<td>1</td>
</tr>
</tbody>
</table>
4.2 Grammar transformation experiments

Chapter 4. Experiments

<table>
<thead>
<tr>
<th>mono-sparc.ini</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of rules</td>
</tr>
<tr>
<td>Total number of nodes</td>
</tr>
<tr>
<td>Number of nonterminals</td>
</tr>
<tr>
<td>Number of terminals</td>
</tr>
<tr>
<td>Number of non-root terminal nodes</td>
</tr>
<tr>
<td>Number of chain rules</td>
</tr>
</tbody>
</table>

Files mono-x86.ini, mono-ia64.ini and mono-sparc.ini were built from the Burg-files that were contained in the Mono Source code package. They can be assembled by combining the standard Mono Burg-files: inssel.brg, inssel-float.brg. Combined with inssel-long.brg or inssel-long32.brg depending on the architecture (32-bit or 64-bit) and the machine dependent set of production rules that can be found in inssel-x86.brg, inssel-ia64.brg and inssel-sparc.brg. All these ini-files have the same single chain rule: base → reg. This chain rule originates from the inssel.brg file.

A pleasant property of these real world grammars is that all terminals and nonterminals have a meaning, just like in the preceding iBurg grammars. The nonterminals for instance represent the following concepts:

- stmt Statement
- reg Standard register
- lreg Long register
- freg Floating point register
- base Base address
- cflags Standard compare result flag
- fpcflags Floating point compare result flag
- i8con 8-bit integer constant

The advantage of these Mono grammars is that they are used in a real world application for instruction selection. They are therefore good candidates for the experiments.

4.2 Grammar transformation experiments

This section presents the results of the experiments that were carried out in the area of tree grammar transformations. Two transformations were the subject of the experiment:

- Removal of non-root terminal nodes in production rules (RED-Z*)
- Removal of chain rules (RED-U*)

RED-Z* refers to the complete transformation that removes all non-root terminals nodes, where RED-Z denotes a single transformation step that removes a single non-root terminal node. The same regular expression style abbreviation is used for RED-U* where RED-U* is the transformation that removes all chain rules and RED-U is one transformation step that removes one chain rule.

The goal of the experiments is to find out how these transformations have to be applied such that as few new rules and nonterminals as possible are created. This number can for instance
be reduced by changing the order of transformations and applying rule reuse techniques in the RED-Z transformation.

To clarify all this we start with a description of these two transformation algorithms, RED-Z(*) and RED-U(*). This discussion describes how these algorithms work and how they can be optimized by reusing symbols and rules. After these two sections the experiments with these transformations are discussed. There are three groups of experiments:

The first experiment measures the effect on the grammar size (number of rules, number of nonterminals) of the order of transformation steps: RED-Z*,RED-U*, RED-U*;RED-Z* and (RED-Z;RED-U)*. This experiment is carried out without reusing suitable nonterminals.

The effect of reuse of suitable nonterminals during the RED-Z transformation steps will be examined in the second experiment. The effects are investigated for all the orders of RED-Z and RED-U transformation steps used in the first experiment.

The last section will present the experiment that considers different strategies for selecting the next Z-node to be processed by a RED-Z transformation step: Shortest Tree First (STF), Tallest Tree First (TTF) or Random. This is done to measure the effect that these node selection strategies have on the level of reuse. This experiment will concentrate on the results of the RED-Z* transformations.

### 4.2.1 RED-Z

A RED-Z transformation step removes a single non-root terminal node from the right hand side of a production rule. This can be done by removing the subtree α rooted at this terminal node from that tree and replacing it with a new nonterminal X. A new production rules must then be created with as left hand side the nonterminal X and as right hand side the subtree α. Figure 4.1 shows such a RED-Z transformation step.

```
(1) S \rightarrow \begin{array}{c}
a \\
d \\
d \end{array} \implies (1a) S \rightarrow \begin{array}{c}
a \\
X \\
d \end{array} , (1b) X \rightarrow d
```

**Figure 4.1: Removal of a non-root terminal node**

This means that a new nonterminal and rule is created for each non-root terminal node that is removed from the grammar. We can now also remove the last non-root terminal node from the tree in Figure 4.1 by again applying RED-Z (see Figure 4.2) and thereby implicitly having applied RED-Z*.

```
(1a) S \rightarrow \begin{array}{c}
a \\
X \\
d \end{array} , (1b) X \rightarrow d \implies (1a) S \rightarrow \begin{array}{c}
a \\
X \\
Y \end{array} , (1b) X \rightarrow d, (1c) Y \rightarrow d
```

**Figure 4.2: Removal of all non-root terminal nodes**

However, one can also reuse rules and symbols if a grammar has two or more similar subtrees rooted at a non-root terminal node (e.g. the two d nodes in rule (1) of Figure 4.1). Such
subtrees can be replaced by the same nonterminal and therefore only result in one additional nonterminal and production rule (see Figure 4.3). The effects of this reuse are discussed in detail in Section 4.2.4.

\[
(1) \quad S \rightarrow a \atop \ \ \ \ \ \ \ \ \ \ d \atop \ \ d \quad \Rightarrow \quad (1a) \quad S \rightarrow a \atop X \ \ \ \ \ \ \ \ X ,
\]

\[
(1b) \quad X \rightarrow d
\]

Figure 4.3: Removal of non-root terminal nodes with reuse

The efficiency of this reuse is determined by the order in which the non-root terminal nodes are removed from a grammar. This is discussed in more detail in Section 4.2.5.

### 4.2.2 RED-U

The RED-U transformation step has, as already briefly discussed in Section 3.1.2, as goal to remove chain rules. Let us recapitulate how this transformation step works.

A chain rule \( A \rightarrow B \) can be removed if one creates a new rule \( A \rightarrow \alpha \) for each subtree \( \alpha \) (which may not consist of a single nonterminal) that is reachable from \( B \). An example of such a RED-U transformation step can be seen in Figure 4.4.

These reachable subtrees are gathered by computing the nonterminal closure for the nonterminal \( B \). The nonterminal closure of \( B \) is the set of nonterminals that is reachable from \( B \) by applying chain rules. Subtrees that are in the RHS of production rules that have as LHS a nonterminal that is contained in the closure are called the RHS subtrees reachable from \( B \).

\[
(1) \quad S \rightarrow T , \quad (2) \quad T \rightarrow a \atop c \ \ \ \ \ \ \ \ c \quad \Rightarrow \quad (1) \quad S \rightarrow a \atop c \ \ \ \ \ \ \ \ c ,
\]

\[
(2) \quad T \rightarrow a \atop c \ \ \ \ \ \ \ \ c
\]

Figure 4.4: Removal of a chain rule

### 4.2.3 Influence of RED-Z/RED-U order

The first experiment carried out compared the resulting Z\_\_ U\_\_ grammars after applying both transformations in different orders. Three possibilities are discussed: RED-Z\*;RED-U\*, RED-U\*;RED-Z\* and (RED-Z;RED-U)*. This last possibility transforms the grammar by randomly applying RED-Z and RED-U transformation steps until the Z\_\_ and U\_\_ characteristics are reached. Due to this nondeterminism this complete transformation is executed a thousand times to get a better view of the average results. The first two apply the reduction steps in a fixed order, so these only have to be executed once.

During this experiment the number of rules and number of nonterminals were measured after each transformation. This experiment was carried out for all grammars, but are not reported on for the IA64 and Sparc grammar of Mono, because their results are very similar to the results of the X86 Mono grammar. These are the results of this experiment:
Chapter 4. Experiments  

4.2. Grammar transformation experiments

Figure 4.5: 'Example 6.0.3'

Figure 4.6: 'Sample 4'
These charts tell that the resulting grammar is smaller when the original grammar has first undergone a RED-Z* transformation instead of RED-U* transformation. This can be explained by the fact that each RED-U step creates additional non-root terminal nodes for each rule with non-root terminal nodes that is reachable from that chain rule. These additional nodes will all result in a new nonterminal and a new production rule if one applies RED-Z transformation steps (without reuse of suitable nonterminals and rules) afterwards.

The Example 6.0.3 grammar for instance, contains three non-root terminal nodes. These
three nodes are in production rules that can be directly reached from the only chain rule in that grammar. So applying RED-U* generates three additional non-root terminal nodes which all result in an extra nonterminal and production rule when applying RED-Z*.

It is however not true that RED-Z*:RED-U* always performs better than RED-U*:RED-Z*. The (RED-Z | RED-U)* experiments showed results that confirm this statement. One would expect that the size of the (RED-Z | RED-U)* grammars is somewhere between the size of the RED-U*: RED-Z* and RED-Z*: RED-U* grammars. The average results confirm this statement, but some of the individual runs showed remarkable results. There were resulting grammars that were even larger than the one from the RED-U*: RED-Z* transformation. Figure 4.9 shows this unexpected result for the X86 Mono grammar.

![Figure 4.9: The minimum, median and maximum during the (RED-Z | RED-U)* transformations for the X86 Mono grammar.](image)

Even more important is the fact that the example grammar constructed to prove the bad (RED-Z | RED-U)* performance also showed that RED-U*:RED-Z* can produce a smaller grammar than RED-Z*:RED-U* when a grammar has a certain shape. This proves that the property presented in the first part (RED-Z*:RED-U* always performs better than RED-U*:RED-Z*) is certainly not true. Example 4.2.1 shows that RED-U*:RED-Z* can outperform RED-Z*:RED-U* and that the random variant can perform even worse than these two.

**Example 4.2.1** This Example shows that relation RED-Z*:RED-U* ≤ (RED-Z | RED-U)* ≤ RED-U*:RED-Z* does not always hold, because (RED-Z | RED-U)* can produce a larger grammar than RED-U*:RED-Z* and that RED-U*:RED-Z* can perform better than RED-Z*:RED-U*. The following grammar is used to illustrate this:

\[
\begin{align*}
T &= \{b, c\} \\
r &= \{(X,0), (Y,0), (b,1), (c,0)\}
\end{align*}
\]
First RED-U*;RED-Z* and RED-Z*;RED-U* will be applied to this grammar. This results in the following intermediate and final set of production rules:

\[
\text{Prod} = \{ (1) \ X \to Y , (2) \ X \to b , (3) \ Y \to b \}
\]

The result is that the grammar from the RED-U*;RED-Z* transformation is smaller than the one from the RED-Z*;RED-U* transformation. This is caused by the RED-U* transformation (2nd from above). The rule that is created by this transformation is already present in the grammar. The number of non-root terminals is therefore not increased by this transformation. This has a positive impact on the resulting grammar.

The (RED-Z | RED-U)* transformation can provide an even larger grammar by applying the transformation step in a particular order:

**Original set of rules:**

\[
P = \{ (1) \ X \to Y , (2) \ X \to b , (3) \ Y \to b \}
\]

Apply RED-Z to rule (2):

\[
P = \{ (1) \ X \to Y , (2) \ X \to b , (3) \ Y \to b , (2a) \ V \to c \}
\]

Apply RED-U to rule (1):

\[
P = \{ (2) \ X \to b , (3) \ Y \to b , (2a) \ V \to c , (4) \ X \to b \}
\]

Apply RED-Z to rule (3):

\[
P = \{ (2) \ X \to b , (3) \ Y \to b , (2a) \ V \to c , (4) \ X \to b , (3a) \ W \to c \}
\]
Apply RED-Z to rule (4):

\[
P = \{ (2) \ X \to b, \ (3) \ Y \to b, \ (2a) \ V \to c, \ (4) \ X \to b, \ (3a) \ W \to c, \ (4a) \ Z \to c \}
\]

The result of this \((RED-Z \mid RED-U)^*\) transformation is a grammar that is even larger than the one after \(RED-U^*;RED-Z^*\) and \(RED-Z^*;RED-U^*\).

This unexpected behavior is caused by the fact that RED-U steps do not always create new rules. If a new rule, say \(x\), is exactly the same as an existing rule \(y\) in the grammar, then the addition of \(x\) to the set of rules will not result in an additional rule. This situation changes when rule \(y\) contains a non-root terminal node and a RED-Z step is applied to \(y\) changing it into rule \(y'\). If this RED-Z step is followed by the same RED-U step, then this leads to an additional rule, because rule \(y'\) is not equal to rule \(x\) anymore.

Summarized, the best transformation order depends on the shape of the grammar. If a grammar contains no chain rules or only chain rules that are converted into unique rules by RED-U then one should choose the \(RED-Z^*;RED-U^*\) transformation. However, the larger the number of non-unique rules created by the \(RED-U^*\) transformation the wiser it is to opt for the \(RED-U^*;RED-Z^*\) transformation. A more advanced option would be to first remove the chain rules that do not produce new rules and then apply \(RED-Z^*\) followed by the remaining RED-U steps.

### 4.2.4 RED-Z*/U* order with reuse

This section discusses the effects of reusing suitable nonterminals and production rules. This experiment discusses the same transformation orders as in the previous section, except for the \((RED-Z \mid RED-U)^*\) transformation. Reusing new nonterminals and production rules could result in less rules and less nonterminals after the transformation when the original grammar contains production rules that have similar subtrees in their right hand sides.

As discussed earlier, this reuse takes place in the RED-Z transformation steps. The reuse efficiency may depend on the order in which the non-root terminal nodes are selected to undergo a RED-Z transformation step. This experiment uses the STF (Shortest Tree First) node selection strategy. In this selection strategy, the smallest subtrees rooted by non-root terminals are removed first, followed by the nodes with larger subtrees. The effect(s) of using other node selection strategies are investigated in Section 4.2.5.

Below are the results of this experiment for the \(RED-U^*;RED-Z^*\) and \(RED-Z^*;RED-U^*\) transformation sequences. The graphs show the results of the previous section (\(RED-Z^*\), no reuse) and the new results (\(RED-Z^*\), STF with reuse). Additionally we present the results for the Mono IA64 and Sparc grammar.
Figure 4.10: 'Example 6.0.3', RED-Z* with and without reuse.

Figure 4.11: 'Sample 4', RED-Z* with and without reuse.
Figure 4.12: 'Sample 5', RED-Z* with and without reuse.

Figure 4.13: 'Mono X86', RED-Z* with and without reuse.
The graphs show what one would expect. The RED-Z* transformations with reuse output grammars with fewer nonterminals and production rules than the corresponding ones without reuse. Below, a table is presented to provide a better view of these differences. This table compares the number of rules after such transformations with reuse to the number of rules after such transformations without reuse. This is performed by setting the number of rules in the resulting grammar with no reuse to 100%. The table then contains the relative sizes when reuse is enabled in the RED-Z transformation steps:
Chapter 4. Experiments

4.2. Grammar transformation experiments

The table shows that the reuse of nonterminals and production rules is very attractive. The number of rules in the large Mono grammars drops to 64–78% of its size without reuse. The effect on the number of nonterminals is even more impressive:

<table>
<thead>
<tr>
<th></th>
<th>Number of rules difference in percentages</th>
<th>Number of nonterminals difference in percentages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RED-Z* RED-U* ; RED-Z* RED-Z* ; RED-U*</td>
<td></td>
</tr>
<tr>
<td>Example 6.0.3</td>
<td>100% 79% 100%</td>
<td>100% 63% 100%</td>
</tr>
<tr>
<td>Sample 4</td>
<td>100% 92% 100%</td>
<td>100% 75% 100%</td>
</tr>
<tr>
<td>Sample 5</td>
<td>92% 89% 94%</td>
<td>86% 75% 86%</td>
</tr>
<tr>
<td>Mono X86</td>
<td>64% 67% 70%</td>
<td>16% 14% 16%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>71% 75% 78%</td>
<td>17% 15% 17%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>68% 70% 74%</td>
<td>17% 14% 17%</td>
</tr>
</tbody>
</table>

This last table shows that the number of nonterminals drops significantly for the large Mono grammars. The reuse of these nonterminals and rules is, as expected, very useful for large grammars.

The results in Figure 4.10 – 4.15 also show that there is no difference in grammar size between RED-Z* ; RED-U* and RED-U* ; RED-Z*, when reusing nonterminals and production rules. This is also proven [Cle07, Section 3.3.3.1 and Remark 3.3.37]. First applying the RED-U* transformation can still result in copying non-root terminal nodes. However, these are now, due to STF reuse, replaced by the same nonterminal when applying RED-Z*. So, RED-U* ; RED-Z* will not, like before, perform worse than RED-Z* ; RED-U*.

Experiments were also carried out with the (RED-Z | RED-U)* transformation. With reuse enabled they also produced grammars that had exactly the same size as the grammars produced by RED-Z* ; RED-U* and RED-U* ; RED-Z*.

In summary, this means that the RED-Z transformation steps with reuse have a positive effect on the application order of RED-U and RED-Z, and that they are more efficient in terms of the size of the resulting grammar.

4.2.5 RED-Z node selection strategies

The final experiment in the area of grammar transformations concentrates on different RED-Z node selection strategies. The three node selection strategies differ in the order in which they remove all the non-root terminal nodes and therefore could have an impact on the reuse of existing nonterminals and rules. The three node selection strategies that were investigated are:


- STF (Shortest Tree First)
- TTF (Tallest Tree First)
- Random

The first node selection strategy, STF (used in the previous experiment), removes the non-root terminal nodes based on the size of the subtree rooted at that node and starting with a node that has the smallest subtree. The TTF selection strategy does the opposite by starting with a node that has the tallest subtree. The last variant removes the nodes in a random order.

This experiment will compare these three node selection strategies with the original grammar and the results of the RED-Z\(^*\) transformation without reuse. This is done for all the grammars described in Section 4.1.

Two remarks on the results displayed below:
- The random selection strategy is not deterministic; the results for this selection strategy are determined by taking the average number of rules and nonterminals over a thousand runs. The minimum, maximum and median of these runs are also visualized for the X86, IA64 and Sparc Mono grammars.
- The results for the 'Sample 4'-grammar are not listed due to the fact that it has only one non-root terminal node. This will not create different situations for the node selection strategies.

![Figure 4.16: 'Example 6.0.3', results of the different RED-Z node selection strategies.](image-url)
Figure 4.17: 'Sample 5', results of the different RED-Z node selection strategies.
4.2. Grammar transformation experiments

Chapter 4. Experiments

Figure 4.18: ’Mono X86’, results of the different RED-Z node selection strategies together with minimum, maximum and median of the random runs.
4.2. Grammar transformation experiments

Figure 4.19: 'Mono IA64', results of the different RED-Z node selection strategies together with minimum, maximum and median of the random runs.
Figure 4.20: 'Mono Sparc', results of the different RED-Z node selection strategies together with minimum, maximum and median of the random runs.

The graphs show that STF and TTF node selection strategy result in smaller grammars than the average random selection strategy. However, the difference between STF/TTF and random selection is very small (around 1.5%). This difference can be explained by the random processing of similar subtrees that have a non-root terminal as root. Processing these subtrees by chance in different directions (STF or TTF) can have a negative influence on the reuse. The chance that this happens is quite small because it requires parallel processing of
similar subtrees in opposite order. Example 4.2.2 shows a situation where the random selection strategy performs worse compared to the STF and TTF node selection strategies.

**Example 4.2.2** This example shows that the random node selection strategy can perform worse than the STF and TTF strategies. This is illustrated with the following grammar:

\[
T = \{ a, b, c \} \\
r = \{ (X,0), (a,2), (b,1), (c,0) \} \\
P = \{ (1) X \rightarrow a \\
    b \\
    b \\
    c \\
    c \}
\]

Applying RED-Z* with TTF based reuse to this grammar results in:

\[
P = \{ (1) X \rightarrow a \\
    S \\
    S \\
    Z \}
\]

More rules and nonterminals can be produced however, if the RED-Z steps are applied in a particular order:

First one of the bottom nodes is removed. This results in the following set of rules:

\[
P = \{ (1) X \rightarrow a \\
    S \\
    T \\
    Z \}
\]

The is followed by the removal of the two b-nodes, which results in the following rules:

\[
P = \{ (1) X \rightarrow a \\
    S \\
    T \\
    \}
\]

After this the last non-root terminal node in rule 3 can be removed:

\[
P = \{ (1) X \rightarrow a \\
    S \\
    T \\
    \}
\]

The resulting grammar has one additional rule compared to the resulting grammar of the RED-Z* transformation with TTF node selection. This is caused by the fact that the random selection can change subtrees that were exactly the same, to subtrees that are visually different due to replacement of nodes. These changed subtrees are then replaced by different nonterminals instead of the same one (S and T instead of one S in the grammar above).

Figures 4.16–4.20 also suggest that the STF and TTF node selection perform similarly. Theoretically TTF can be less effective than STF. Unfortunately this was not visible in the results of the experiment, due to the shape of the grammars. Example 4.2.3 was constructed to show that this can happen.

**Example 4.2.3** This example illustrates that there exist grammars for which the TTF node selection strategy in the RED-Z* transformation performs worse than the STF selection strategy. The following grammar is used to illustrate this:
4.3 Automaton construction experiments

In this section, the experiments with tree automaton construction algorithms are discussed. These automaton constructions build automata that can be used for solving the tree accept-
There are different algorithms for different types of automata. Let us therefore first give a quick overview of the types of tree automata. As discussed in Chapter 1 there are two main types of automata: nondeterministic and deterministic automata, where nondeterministic automata can also contain \( \varepsilon\)-transitions. The two types can themselves be divided into two categories depending on the acceptance direction of the automaton: root-to-frontier (RF) or frontier-to-root (FR). This results in six types of automata.

- **Nondeterministic**
  - Root-to-frontier \((\varepsilon)\text{NRFTA}\)
  - Frontier-to-root \((\varepsilon)\text{NFRTA}\)

- **Deterministic**
  - Root-to-frontier \((\text{DRFTA})\)
  - Frontier-to-root \((\text{DFRTA})\)

We already mentioned that the accepting power of the \(\text{DRFTA}\) is less than the accepting power of the other three automata. The experiments in this section focus on the constructions of the remaining five types of automata. The conducted experiments were used to measure the efficiency of the different construction algorithms and the characteristics of the resulting automata:

- Number of states
- Number of transitions
- Memory use

The measurements were used to compare the different construction algorithms and types of automata. These results may be used to select the most efficient construction technique for automata used in tree acceptance and tree parsing applications. Here we mean most efficient in terms of construction speed and/or automaton memory use.

The construction methods will be discussed separately for deterministic and nondeterministic automata, because they differ in construction algorithms and optimization possibilities. However, some general topics are treated before discussing these two groups of automata.

First the measurement techniques used in the experiments are described. This is followed by a discussion about general automaton construction issues. Finally the specific construction algorithms for nondeterministic and deterministic automata are discussed.

### 4.3.1 Measurement techniques

The automaton construction experiments measure two characteristics. The first characteristic is the construction process itself (running time). The other is the constructed automaton (number of states, number of transitions, memory size). Characteristics like number of states and number of transitions are measured easily by counting them. Measuring running time
and memory usage is a bit more complicated due to restrictions of the Java programming language and runtime environment.

**Runtime measurements**

The time consumption measurements were performed by using built-in methods of Java. The Java class library provides the `System.nanoTime()`-method which returns the difference, measured in nanoseconds, between the current time and specific point in time (chosen by the JRE). This method is called before and after a construction. The difference between these two measurements is considered to be the running time of the construction algorithm.

**Memory usage measurements**

The memory usage measurements were carried out in a similar way to the time measurements. Java provides the possibility to request the total size (in bytes) of the reserved memory pool. It also provides a method that returns how many bytes are still available in this pool. Subtracting the free space from the total space will result in the space that is currently used. This value was computed before and after the construction (before and after the time measurements to be more precise).

However, there were some difficulties with these measurements caused by the existence of the garbage collector. Measuring the free memory inside the memory pool is only reliable if the garbage collector has removed all unreferenced objects. This can be achieved by manually triggering the garbage collector until the amount of free memory stabilizes, because a single garbage collection run does not remove all unreferenced objects.

**Hardware & Java environment**

The last measurement aspect to discuss is the environment of the experiments: the hardware and Java Runtime Environment. These facts can be found in Table 4.1.

<table>
<thead>
<tr>
<th>CPU</th>
<th>AMD Athlon64 3500+ (Winchester)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1024MB Dual Channel DDR400</td>
</tr>
<tr>
<td>Operating system</td>
<td>Microsoft Windows XP SP2 (32bit)</td>
</tr>
<tr>
<td>Java version</td>
<td>Java SE 6 Update 1</td>
</tr>
</tbody>
</table>

Table 4.1: Used hardware & software

**4.3.2 Automaton construction: general issues**

Some general tree automaton subjects are discussed before discussing the specific deterministic and nondeterministic variants, namely state construction and transition storage.

**State construction**

The tree automata in the upcoming experiments are constructed from regular tree grammars. Each of the states in such an automaton corresponds to a subtree of a production rule in the grammar (for nondeterministic automata) or a set of such subtrees (for deterministic automata). Each construction algorithm discussed in this report receives a set of subtrees,
called *Item Set*, from the outside world and creates states corresponding to (sets of) these subtrees.

The standard set of subtrees used for these algorithms is the *All-Sub* item set. This set is defined in the following way:

\[
\text{Items} = \text{Subtrees}(\text{RHS}(\text{Prods}')) \text{ where } \text{Prods}' = \text{Prods} \cup S' \rightarrow S
\]

This definition describes that the *All-Sub* item set contains all subtrees in the right hand side of the production rules and the start symbol. Example 4.3.1 shows such an *All-Sub* item set for an example grammar.

As mentioned, states are constructed from the subtrees in these item sets. A consequence of this is that providing a smaller item set to a construction algorithm can result in an automaton with less states (this can be seen in the upcoming sections about the experiments with the construction algorithms). However, one must ensure that the items in this item set can still be used to represent all possible subtrees of the production rules if one shrinks such a set.

There are two smaller item sets used in the experiments:

- Proper-N
- Proper-S

The *Proper-N* set is constructed by using the following definition.

\[
\text{Items} = \text{N} \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))
\]

The proper subtrees in this definition are subtrees that do not only occur as a complete right hand side. Trees that only occur as complete right hand sides can be removed, because they can also be represented by a nonterminal that directly produces this tree.

Example 4.3.1 shows a *Proper-N* item set as well. The tree with root *a* is removed compared to the *All-Sub* set, because it is no proper subtree in any right hand side. This tree is therefore obsolete because it can be represented by nonterminal *A*.

The *Proper-S* item set is similar to the *Proper-N* item set and can be constructed using the following definition.

\[
\text{Items} = \text{S} \cup \text{ProperSubtrees}(\text{RHS}(\text{Prods}'))
\]

This item set contains the same number of items or less when compared to *Proper-N*, depending on the shape of the grammar. Unlike *Proper-N*, the *Proper-S* does not contain nonterminals that are not the start symbol and are not present as a proper subtree in right hand side of a production rule. These removed nonterminals can be divided into two types:

- Nonterminals that are only present as complete right hand side of a rule (and as left hand side).
- Nonterminals that are only present as left hand side of a rule.

In the first case the corresponding rule is a chain rule and this nonterminal can be represented by the left hand side nonterminal of that rule. The nonterminals of the second case are unreachable as discussed in Section 1.1.3. Unreachable (and unproductive) nonterminals
and rules are useless and therefore removed from the grammars before constructing the item set.

Example 4.3.1 also provides the Proper-S item set for an example grammar. It shows that nonterminal \( A \) is removed compared to Proper-N. This nonterminal can be removed because it is superfluous due to the chain rule and its absence as a proper subtree in the rules.

**Example 4.3.1** This example illustrates for the grammar below which subtrees can be found in the All-Sub, Proper-N and Proper-S item set.

\[
\begin{align*}
N &= \{S, A\} \\
\Sigma &= \{a, b, c\} \\
r &= \{(S,0), (A,0), (a,2), (b,1), (c,0)\} \\
Prods &= \{ (1) S \rightarrow A, (2) A \rightarrow \begin{array}{c} a \\ b \\ c \end{array} \} \\
\end{align*}
\]

The three item sets that can be constructed for this grammar are:

\[
\begin{align*}
\text{All-Sub:} & \quad S, A, \begin{array}{c} a \\ b \\ c \end{array} \\
\text{Proper-N:} & \quad S, A, \begin{array}{c} b \\ c \end{array} \\
\text{Proper-S:} & \quad S, \begin{array}{c} b \\ c \end{array} \\
\end{align*}
\]

Experiments will be performed with all three sets in the construction algorithms, because they could have a large impact on the size of the resulting automata. The results can be found in separate sections about the nondeterministic and deterministic construction algorithms.

**Transition storage**

For each terminal, a transition relation exists that relates a single state to a vector of states of length \( n \), where \( n \) is the rank of the terminal. Such a relation can be undirected, FR-directed or RF-directed. These two types can be found in respectively Frontier-to-Root TAS and Root-to-Frontier TAS. Only these two types automata are discussed in this report, because they provide the possibility to optimize the transition storage for this specific type of transition and thereby optimize the usage of the automata.

FR-directed transitions go from a vector of states to a single state, RF ones go the other way around. This means that the transition for a symbol \( a \) of rank \( n > 0 \) can be represented as two types of functions.
Chapter 4. Experiments

4.3. Automaton construction experiments

For nondeterministic TAS
FR:  \( R_a \in Q^n \rightarrow P(Q) \)
RF:  \( R_a \in Q \rightarrow P(Q^n) \)

For deterministic TAS
FR:  \( R_a \in Q^n \rightarrow Q \)
RF:  \( R_a \in Q \rightarrow Q^n \)

These functions can also be used for symbol of rank 0, where \( Q^0 \) is then the empty tuple. However there was chosen to remove this empty tuple for the case where \( n=0 \), because it contains no information. This then leads to the following functions:

For nondeterministic TAS
FR:  \( R_a \in P(Q) \)
RF:  \( R_a \in Q \)

For deterministic TAS
FR:  \( R_a \in Q \)
RF:  \( R_a \in Q \)

The transition storage implementations for deterministic automata used in the experiments use conventional techniques. As discussed earlier only DFRTAs are implemented, because their RF counterparts have less acceptance power. The storage of the FR-directed transitions for these automata is achieved by creating an \( n \)-dimensional transition table for each symbol of rank \( n > 0 \) (transition tables for symbols of rank 0 simply consist of a single record). In practice these are integer indexed tables. Lookup is therefore done by translating input states to unique integer values (like in BURG, see Section 2.3).

These transition tables are constructed by using nested dynamic arrays. Dynamic arrays were chosen due to the characteristics of the construction algorithms for these DFRTAs. The algorithms compute the state set in an iterative way, so one does not know the final amount of states before the end of the construction and one therefore does not know the length of these arrays. Dynamic arrays provide the possibility to enlarge the transition table as the number of states grows and thereby solve this problem in an easy way.

There is also a penalty paid. Dynamic arrays have more overhead than static arrays. The practice showed that this overhead is quite large. This is caused by the large number of arrays that has to be used. The formula below shows the number of dynamic arrays used in a DFRTA with \( |Q| \) states.

\[
\sum_{t \in \text{terminals}} \left( \sum_{i=0}^{t.\text{rank}-1} |Q|^i \right) \quad \text{where } t.\text{rank} > 0
\]

There were also transition tables implemented based on a combination of static and dynamic arrays. This implementation is not discussed in detail, because it was only used to compare the efficiency of this storage. This second implementation uses a low number of dynamic arrays:

71
When these two implementations were compared they confirmed the statement that the dynamic arrays use a large amount of memory. An automaton for a large grammar uses almost 150 megabytes when using the transition tables that completely are constructed from dynamic arrays. Switching to the second implementation resulted in a drop of 40 megabytes while nothing else changed in the automaton and construction algorithm. However, the first implementation was chosen for the experiments because the second version could only be used for tables where all dimensions have the same size. This was not true for the filtered DFRTAs that are discussed at the end of Section 4.3.4. The first implementation was therefore chosen to be able to compare the different DFRTA construction algorithms in a fair way.

We now describe the implementation of the transition tables for nondeterministic TAs. As described in the beginning of this section, nondeterministic automata can contain multiple transitions on a symbol for one state or state vector.

Figure 4.21: An example NFRTA with multiple transitions for the same input vector/symbol combination.

Figure 4.21 shows for example how state vector \((q_1, q_2)\) has two \(a\)-transitions originating from it. This means that the transition storage must not store a single result state or result vector, but a set of states/vectors. There were two other aspects that played a role when designing the transition tables for the nondeterministic TAs, namely the direction of the transition relations (FR or RF) and the fact that the tables should be optimized for the relatively low number of transitions (compared to deterministic TAs).

The transition storage is based on hash tables instead of normal tables, since nondeterministic automata mostly have a low number of transitions for each state. The exact shape of the transition storage depends on whether it is targeted at an FR or RF automaton.

The transition tables for NRFTAs were easy to construct: a single hash table was created for each symbol with a rank larger than zero. This table stores for each state a set of vectors that can be reached with a transition for that symbol from this source state.

The transition tables for NFRTAs are comparable to those for DFRTAs, but NFRTAs use nested hash tables instead of nested dynamic arrays, because they are more efficient. This is caused by the fact that NFRTAs, unlike DFRTAs, do not have a transition for each possible symbol/state vector combination. Another difference is that the DFRTA transition storage points to a single state for each vector stored in the multi-dimensional storage while the NFRTA storage points to a set of result states.

The different implementations to store transitions have different properties. These properties will emerge and be discussed in the upcoming experiments.
4.3.3 Constructions of nondeterministic automata

The constructions for nondeterministic tree automata are based on the constructions described in [Cle07, Section 6.6, Construction of tree acceptors]. This section describes FR and RF versions of four constructions for nondeterministic tree automata.

- TGA-TA:ALL-SUB
- TGA-TA:ALL-SUB:REM-ε
- TGA-TA:PROPER-N:REM-ε
- TGA-TA:PROPER-S:REM-ε

These four constructions are based on the different item sets that can be used for constructing automaton states. The first two are based on the item set All-Sub and the last two on Proper-N and Proper-S. Remarkable is the REM-ε addition to the last three constructions. The first construction is the only construction that generates an automaton with these ε-transitions. The other three are targeted at ε-free tree automata (and therefore contain this REM-ε addition), because these automata lead to more efficient acceptance and parsing algorithms.

Eight different types of automata can be constructed based on the two directions and four constructions. The upcoming four sections discuss the four construction algorithms and the characteristics of the constructed NFRTA’s and NRFTA’s.

**TGA-TA:ALL-SUB**

The TGA-TA:ALL-SUB automaton construction creates a nondeterministic automaton based on the All-Sub item set and is described in [Cle07, Construction 6.6.2, TGA-TA:ALL-SUB]. This construction algorithm creates a state for each unique subtree that can be found in the item set. Symbol transitions between these states are generated from the parent-child relations of these subtrees. This collection of transitions is then extended by an ε-transition for each rule between the state for the left hand side nonterminal and the state for the right hand side tree.

The first and fifth line of Tables E.1 – E.14 contain the measurements for these constructions (both FR and RF), especially the Basic Statistics tables. A pleasant property of this first construction algorithm is the short construction time and small amount of used memory. This is due to the fact that the number of states is proportional to the number of different subtrees and the number of transitions is proportional to the number of unique parent-child node relationships and the number of production rules. This advantage will be seen in all nondeterministic automaton constructions.

There is one difference between the constructed εNFRTAs and εNRFTAs. This difference is found in the way the transitions are stored. The tables in Appendix E show that the FR variant is slightly larger in memory usage than the RF variant. The difference can be explained by the data structures used for transition storage.

The FR variant uses a multidimensional hash table for each terminal in the grammar, as described in Section 4.3.2. The dimension of the table depends on the rank of the corresponding terminal. The RF variant uses only a single dimension hash table for each terminal. An
RF automaton therefore contains fewer hash tables to store transitions. All these hash tables contain some overhead. This overhead is the major contributor to the small difference in size between NFRTAs and NRFTAs.

**TGA-TA:ALL-SUB:REM-ε**

The disadvantage of the previous construction is that it produces an automaton with ε-transitions. This results in computing the ε-closure each time a state is visited in an acceptance algorithm that uses these automata.

The construction in this section constructs an ε-free automaton based on the *All-Sub* item set. A description of this construction can be found in [Cle07, Construction 6.6.10, TGA-TA:ALL-SUB:REM-ε]. This construction computes the ε-closure during the construction and adds additional non ε-transitions to replace the ε-transitions. For example: state α has an a-transition to state β and state γ can be reached from β by an ε-transition, then an extra a-transition is added from α to γ to replace this ε-transition. The ε-transition can be removed if this is done for all transitions to β.

Lines two and six in Tables E.1 – E.14 show the measurements when these constructions are applied. Obviously, only the number of transitions has changed if one compares these automata to the automata with ε-transitions.

There are also some unexpected results, for instance the occupied memory of the FR automaton shrinks and that of the RF automaton grows. The single dimension hash table that contains the ε-transitions disappears in both variants. However, the effect of the replacement of these transitions is different. This too is caused by the different implementations of the FR and RF transition storage.

The RF automaton has to store these new transitions as new vectors in the single dimension transition tables. These new vectors contain many existing states. This illustrated with Figure 4.22. The RF variant of this automaton has the a-transition $q_s \rightarrow (q_0, q_1, q_2)$. Removing the epsilon transition results in the addition transition: $q_x \rightarrow (q_0, q_1, q_2)$. This results in storing an additional vector of size three and thereby duplicating pointers to all three states.

![Figure 4.22: Undirected TA with an ε-transition.](image)

The FR automaton does not have the disadvantage of the RF variant due to the multidimensional hash tables. This can result in creating additional hash tables, but in many cases it only results in adding the result state to the set of result states for an existing vector. This last case is also true for the example in Figure 4.22. The vector $(q_0, q_1, q_2)$ already exists and only the result set \{q_s\} is expanded with state $q_x$. This addition is in practice smaller than the memory that is freed by the removal of the hash map that stored the ε-transitions.
The \textit{TGA-TA:ALL-SUB:REM-}\textsubscript{\varepsilon} construction produces \(\varepsilon\)-free automata, but these automata contain states which are not reachable from the root accepting state. These states are the states that have a non proper subtree or an unreachable nonterminal (when not removed from the grammar) as corresponding match. Constructing the automaton based on the \textit{Proper-N} item set will not create states for a non-proper subtree. This section discusses the construction that can be found in \cite[Cle07, Construction 6.6.19, TGA-TA:PROPER-N:REM-\varepsilon]{TGA-TA}. The construction produces an \(\varepsilon\)-free automaton based on this \textit{Proper-N} item set.

Tables E.1 – E.14 confirm that the construction produces automata with less states and transitions. The measurements show that the resulting automaton is constructed in less time and uses only 20\% of the memory of the standard automaton based on the \textit{All-Sub} item set. Table 4.2 compares the \textit{FR} automata created with the \textit{TGA-TA:PROPER-N:REM-}\varepsilon construction to the \textit{FR} automata created with the \textit{TGA-TA:ALL-SUB:REM-}\varepsilon construction.

<table>
<thead>
<tr>
<th>Example 6.0.3</th>
<th># States</th>
<th># Transitions</th>
<th>Memory Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample 4</td>
<td>46.2%</td>
<td>75.9%</td>
<td>67.3%</td>
</tr>
<tr>
<td>Sample 5</td>
<td>50.0%</td>
<td>72.7%</td>
<td>71.9%</td>
</tr>
<tr>
<td>68000</td>
<td>25.0%</td>
<td>58.5%</td>
<td>62.0%</td>
</tr>
<tr>
<td>Mono X68</td>
<td>11.8%</td>
<td>61.1%</td>
<td>59.0%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>9.1%</td>
<td>62.8%</td>
<td>58.4%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>10.4%</td>
<td>61.6%</td>
<td>58.4%</td>
</tr>
</tbody>
</table>

Table 4.2: \textit{nfrta} characteristics for the \textit{TGA-TA:PROPER-N:REM-}\varepsilon construction compared to the \textit{TGA-TA:ALL-SUB:REM-}\varepsilon construction

The table shows that there is a significant drop in memory usage for all grammars. Especially the large grammars seem to benefit from this different item set.

One can construct the same table for the \textit{RF} automata. The columns with the numbers of states and number of transitions will contain the same values. However, there are again some small differences in memory usage. These differences are the same as described in the previous section about the standard \textit{All-Sub} construction and are caused by the usage of different data structures.

**TGA-TA:PROPER-S:REM-\varepsilon**

This construction is closely related to the previous construction. The construction algorithm \cite[Cle07, Construction 6.6.15, TGA-TA:PROPER-S:REM-\varepsilon]{TGA-TA} is the same with the exception that it uses the \textit{Proper-S} item set instead of the \textit{Proper-N} item set. This construction can result in automata that are smaller than the ones constructed by the \textit{Proper-N} variant. This is the case if the grammar contains nonterminals that are not present as a proper subtree or as start symbol.

The tables in Appendix E show that there is no difference between the constructions with the \textit{Proper-N} and \textit{Proper-S} item set for our grammars. This is due to the fact that all used grammars do not contain nonterminals that are not present as proper subtree or start symbol. The behavior and the result of this construction is therefore the same as for the previous one.
4.3.4 Constructions of deterministic automata

This section focuses on the construction of DFRTAs. DRFTAs are not covered because their acceptance/parse power is less than their DFRTA counterparts. This section discusses five construction algorithms for DFRTAs.

- DFRTA Standard Construction
- DFRTA Construction with Subtree Filtering
- DFRTA Construction with Index Filtering
- DFRTA Construction with Symbol Filtering
- DFRTA Construction with Index & Symbol Filtering

Deterministic automata in general have more states and transitions than their corresponding nondeterministic variants. The difference can be very large when no optimization techniques are used during the construction of deterministic automata. The first construction technique is a technique without any special optimization techniques. The bottom four algorithms focus on decreasing the size of the automaton.

All these constructions can be based on all three item sets (All-Sub, Proper-N and Proper-S). The upcoming subsections discuss the characteristics of these five constructions and the effect of the different item sets.

Standard DFRTA construction

The standard construction of a DFRTA is different compared to the construction of the nondeterministic automata. The construction of a DFRTA has two phases.

The first phase creates states for all terminals of rank zero. A state $q$ is created and this terminal is added to the match set of this state. Next, all production rules with this terminal as RHS are gathered. The transitive chain rule closure (computed by using Warshall’s Algorithm [War62]) on nonterminals is computed. All nonterminals that are reachable from these RHSs are then added to the match set of the state. Finally a start transition, of the form $(\epsilon) \rightarrow q$, is added for each of these terminals. These are simply stored as $q$ since $(\epsilon) \rightarrow q \cong q$.

Example 4.3.2 provides an example of this first phase.

Example 4.3.2 This example shows how the initial states are created from the input grammar in the first phase of the standard DFRTA construction algorithm. This example is based on the following grammar:

$$
N = \{S, X, Y\} \\
\Sigma = \{a, b, c, d\} \\
r = \{(S,0), (X,0), (Y,0), (a,2), (b,1), (c,0), (d,0)\} \\
Prods = \{(1) \, S \rightarrow a, (2) \, S \rightarrow b, (3) \, X \rightarrow c, (4) \, Y \rightarrow d\}
$$

Terminals $c$ and $d$ are the terminals of rank zero, this results in two initial states.

- $q_0$: \{c\}
Chapter 4. Experiments 4.3. Automaton construction experiments

- $q_1: \{d\}$

The match sets need to be expanded with the nonterminals that produce these terminals. This results in these updated match sets.

- $q_0: \{c, X\}$
- $q_1: \{d, Y\}$

Finally the match set has to be updated with the closure of $X$ respectively $Y$. However, there are no chain rules that produce $X$ or $Y$, so the two sets/states form the state set after phase 1. Transitions are created for these states and their corresponding terminal as described in the introduction. This results in the following two ‘transitions’.

$T_c \ q_0$
$T_d \ q_1$

These states and transitions can be considered as the start state and transitions of this automaton.

The second phase creates new states based on the current state set and the item set (All-Sub, Proper-N or Proper-S). This is done in an iteration over all existing states and the states that are created during this phase. When processing state $q_i$ in this iteration the algorithm looks at all terminals with a rank larger than zero. It creates the following state vectors of length $n$, where $n$ is the rank of the terminal:

$\{q_0, \ldots, q_i\}^n / \{q_0, \ldots, q_{i-1}\}^n$

This means that all vectors of length $n$ are created that contain only states $q_j$ where $j \leq i$ and the vector has at least one occurrence of $q_i$.

The algorithm determines for each of these vectors which subtrees from the item set matches the terminal with this child state vector. These matches are stored in a match set for that terminal in combination with that vector.

If the subtrees in this match set are also present as the complete right hand side of a production rule, then the match set is expanded by the nonterminal left hand side of that rule and the closure of this nonterminal. A new state is created if there is no state with this match set. Transitions between the states from the vector and the (newly) constructed state are added to the automaton. This second phase is illustrated in Example 4.3.3.

**Example 4.3.3** This example shows how new states are created from the existing states in the second phase of the standard DFRTA construction algorithm that uses the All-Sub item set. States $q_0$ and $q_1$ are the states that came out of the first phase.

- $q_0: \{c, X\}$
- $q_1: \{d, Y\}$

The All-Sub item set that contains all possible matches for new states is: $\{a(X, Y), b(Y), X, Y, c, d\}$. Now we can start executing phase 2. This means computing new states for possible state vectors (consisting of existing states). This starts with case $i = 0$ that checks
all vectors that only contain states $q_j$ where $j \leq i$ and that have at least one occurrence of $q_i$ (this automatically holds for $i = 0$).

**Case $i=0$:**
Terminal $a$ will be the first symbol for which the vectors are constructed. Creating these vectors for state $q_0$ and terminal $a$ of rank 2 results in this single vector: $(q_0, q_0)$. The following tree is obtained if this vector is visualized as a tree with root $a$.

```
  a
 /\  /
{c, X} {c, X}
```

There is no subtree in the All-Sub item set that matches with this subtree (all possibilities for the child match sets are considered). This means that a new state $q_2$ will be created with an empty match set. This results in this small table.

<table>
<thead>
<tr>
<th>$T_a$</th>
<th>$q_0$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The same procedure is followed for terminal $b$, resulting in vector $(q_0)$ which also matches with no subtree and therefore results in the same state $q_2$.

<table>
<thead>
<tr>
<th>$T_b$</th>
<th>$q_0$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both symbols are processed for state $q_0$, so one can proceed to the second iteration.

**Case $i=1$:**
Proceeding with $i = 1$ means that we continue by handling state vectors containing $q_0$ and $q_1$. For symbol $a$ this results in three vectors that need to be processed: $(q_0, q_1)$, $(q_1, q_0)$ and $(q_1, q_1)$. The first vector can be visualized with the tree below.

```
  a
 /\  /
{c, X} {d, Y}
```

Subtree $a(X,Y)$ from the item set matches with this tree. This results in a new state $q_3$ with match set $\{a(X,Y), S\}$. The second and third vector result in an empty match, so they result in a transition between these vectors and the existing state $q_2$. The complete iteration for terminal $a$ therefore results in the following transition table.

<table>
<thead>
<tr>
<th>$T_a$</th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The same is done for terminal $b$, where vector $(q_1)$ results in a new state $q_4$ with match set $\{b(Y), S\}$ and thus gives the following one-dimensional transition table.

<table>
<thead>
<tr>
<th>$T_b$</th>
<th>$q_0$</th>
<th>$q_2$</th>
<th>$q_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The second phase will be repeated until all states (cases $i = 2$ to $i = 4$) are processed. This does not result in new states for this example grammar. The state set below is therefore the complete set of states for the resulting automaton.

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>match set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>{c, X}</td>
</tr>
<tr>
<td>$q_1$</td>
<td>{d, Y}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_2$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>{a(X, Y), S}</td>
</tr>
<tr>
<td>$q_4$</td>
<td>{b(Y), S}</td>
</tr>
</tbody>
</table>

The only changes in the iterations for $q_2$, $q_3$ and $q_4$ are in the transition table of terminal $a$ and $b$. New vectors can be constructed for both terminals based on these states. However, transitions from these vectors do not result in new states. This results in these final transition tables:

<table>
<thead>
<tr>
<th>$T_a$</th>
<th>$q_0$</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$T_b$</th>
<th>$q_0$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_0$</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_1$</td>
<td>$q_4$</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td>$q_4$</td>
<td>$q_2$</td>
<td></td>
</tr>
</tbody>
</table>

The final result is an automaton with 5 states and 32 transitions. These additional two transitions are the two transitions for terminals of rank 0 of phase 1.

This second phase is very time consuming, because what it does in practice is creating all possible state vectors of length $n$ for each terminal of rank $n$. These vectors are compared to all the subtrees in the item set.

This comparison is not trivial because each state vector for a symbol has to be compared to all subtrees $\alpha$ in the grammar with that symbol as root. The complexity is increased by the fact that each state on position $i$ of the vector can contain multiple matching subtrees and that each of these subtrees has to be compared with the subtree on position $i$ of subtree $\alpha$. However an optimization is applied here: the construction stores all subtrees in the production rules of the grammar based on their root symbol, which avoids searching for subtrees $\alpha$ that need to be checked against the vectors. This optimizes the computation of the matches, but still requires checking many possible matches for a large number of vectors for each symbol.

The first three dfrta lines in the tables in Appendix E show the results of the experiments with this standard dfrta construction, for all three item sets introduced before.

The experiments with the All-Sub item set show the most remarkable results. The number of states is almost equal to the number of states of the nondeterministic variants, but the construction time needed for the large Mono grammars is very high. The algorithm needs about five minutes to construct an automaton and the resulting automaton also needs more than one hundred megabytes to store its states and transitions. The large number of transitions (e.g. 25 million for the Mono X86 grammar) explains this memory consumption.
4.3. Automaton construction experiments

Switching to the Proper-N or Proper-S item set makes a big difference. The automaton constructed with such an item set uses 98% less memory. Table 4.3 shows the relative memory size and construction time for the Mono automata constructed with the Proper-N set, where 100% is the value for the construction with the All-Sub item set.

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86</td>
<td>1.66%</td>
<td>0.63%</td>
</tr>
<tr>
<td>IA64</td>
<td>1.22%</td>
<td>0.31%</td>
</tr>
<tr>
<td>Sparc</td>
<td>1.47%</td>
<td>0.51%</td>
</tr>
</tbody>
</table>

Table 4.3: Standard Proper-N construction relative to All-Sub

The difference in size and construction time can be easily explained. The standard DFRTA construction inspects all combinations of states for each terminal in the grammar to construct all transitions. This results in the following number of checks/transitions for an automaton with |Q| states.

$$\sum_{t \in \text{terminals}} |Q|^{t\text{.rank}}$$

The X86 Mono grammar for example contains around 80 terminals of rank 2. With 557 states constructed from the All-Sub set, these 80 terminals lead to approximately 25 million transitions. These transitions are stored in n-dimensional tables which are modeled by nested dynamic arrays. There is a cell in these tables for each transition and each of these cells contains a pointer to the resulting state. This results in 95 Megabytes of pointers for the X86 mono grammar. The remaining 50 megabytes are occupied by the overhead of the dynamic arrays that represent the tables. These 50 megabytes can be reduced by using static arrays instead of dynamic arrays, as discussed in Section 4.3.2. Dynamic arrays were however used to create a fair comparison between standard DFRTAs and filtered DFRTAs (which can not be constructed efficiently using static arrays).

Summarized, the automata created with this standard DFRTA construction algorithm have a large collection of transitions even when using the Proper-N or Proper-S item sets. This leads to impractical automata. However, there are filtering techniques to reduce the number of transitions. The following four sections discuss constructions that use these filter techniques.

DFRTA with subtree filtering

The DFRTA construction with the subtree filtering technique tries to reduce the transition table size. This is done by reducing the possible indices for these tables (all states in the standard DFRTA construction).

The DFRTA construction with subtree filtering adds an additional filtered match set table (R-table) and a translation table ($\phi$-table) to the automaton. This filtered match set table contains all unique match sets from the original state set after removing subtrees that are not proper subtrees. This results in a filtered match set table which contains only match sets with proper subtrees. This can result in fewer match sets than states, because two match sets from two original states can become equal after removing non-proper subtree matches. The $\phi$-table describes which filtered match set corresponds to which original state.
These filtered match sets are used as indices for the transition tables instead of the original states. However, this replacement does not affect the acceptance/parse power of the automaton, because a transition based on a non-proper subtree will result in a state with the empty match set and is therefore useless. This is easy to explain, because the match set of the resulting state will only contain subtrees in which this non-proper subtree is a child tree and that is a contradiction with the fact that it is a non-proper subtree. This transitions therefore result in the state with the empty match set.

The result of using these filtered match sets instead of the (match sets of the) original states is that the transition tables for each of the terminals can be based on these smaller filtered match sets instead of the original states. This results in less transitions, less memory use and faster constructions. An example of this can be seen in Example 4.3.4.

Example 4.3.4 Example 4.3.2 and 4.3.3 show the standard DFRTA construction. This example shows what happens when applying subtree filtering during the construction. The set \{X, Y\} is the set of proper subtrees of the grammar and therefore used for filtering. The table below shows the original match sets and the match sets after filtering with these proper subtrees.

<table>
<thead>
<tr>
<th>Phase</th>
<th>match set</th>
<th>filtered match set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q0</td>
<td>{c, X}</td>
<td>{X}</td>
</tr>
<tr>
<td>q1</td>
<td>{d, Y}</td>
<td>{Y}</td>
</tr>
<tr>
<td>Phase 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>\ø</td>
<td>\ø</td>
</tr>
<tr>
<td>q3</td>
<td>{a(X, Y), S}</td>
<td>\ø</td>
</tr>
<tr>
<td>q4</td>
<td>{b(Y), S}</td>
<td>\ø</td>
</tr>
</tbody>
</table>

This results in the following filter table \(R\) and translation table \(\phi\):

<table>
<thead>
<tr>
<th>R-Table</th>
<th>(\phi)-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_0)</td>
<td>\ø</td>
</tr>
<tr>
<td>(R_1)</td>
<td>{X}</td>
</tr>
<tr>
<td>(R_2)</td>
<td>{Y}</td>
</tr>
<tr>
<td></td>
<td>(q_4)</td>
</tr>
</tbody>
</table>

The filter table contains only three different sets while there are five states (and corresponding match sets), so using these filtered sets as base for the transition tables reduces the size of each transition table from \(5^n\) entries to \(3^n\) entries, where \(n\) is the rank of the terminal.

The transition tables for terminal \(a\) are printed in Table 4.4 to illustrate this. The left part of the table contains the standard table based on the original states and the right side the transition table based on the subtree filtered match sets.
Appendix E shows the results of the experiments with this filtering technique for all grammars and these results confirmed the expectations. If we compare the standard construction and the construction with subtree filtering for the All-Sub item set then we see a large difference. The normal set of states is not influenced, but the number of transitions drops significantly.

The R-Entries column in the ‘Table Statistics’ tables in Appendix E show how large the filtered match set tables are. Take for example the Mono IA64 grammar. The standard dfrta for this grammar contains 438 states. A 2-dimensional transition table of a terminal of rank 2 will contain $438^2$ entries. The filtered match set table in the subtree filtered automaton contains only 41 states. The transition table for the same terminal of rank 2 will now contain $41^2$ entries, this is 0.9% of the entries of the standard automaton. This difference shrinks when using the Proper-N or Proper-S item set. This is caused by the fact that the states of these automata do not contain non-proper subtrees (except the nonterminals N or the start symbol S) due to the restricted item set. Table 4.5 compares the standard construction and the construction with subtree filtering for the Mono grammars.

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>X86 – All-Sub</td>
<td>1.6%</td>
<td>2.5%</td>
<td>2.8%</td>
</tr>
<tr>
<td>X86 – Proper-N</td>
<td>97.0%</td>
<td>99.0%</td>
<td>243.0%</td>
</tr>
<tr>
<td>IA64 – All-Sub</td>
<td>0.9%</td>
<td>2.4%</td>
<td>2.0%</td>
</tr>
<tr>
<td>IA64 – Proper-N</td>
<td>95.4%</td>
<td>99.1%</td>
<td>217.7%</td>
</tr>
<tr>
<td>Sparc – All-Sub</td>
<td>1.1%</td>
<td>2.4%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Sparc – Proper-N</td>
<td>92.7%</td>
<td>95.5%</td>
<td>291.6%</td>
</tr>
</tbody>
</table>

Table 4.5: Construction characteristics for the Mono grammars for the subtree filter construction compared to the standard construction

The table confirms that there is a large gain when applying subtree filtering in combination with the All-Sub item set and that there is almost no gain when Proper-N is used as the item set. The construction time actually increases for this last item set due to the extra work that is needed for building the filter tables. The results for the Proper-S set are omitted in the table because this item set contains the same subtrees as Proper-N for these Mono grammars and therefore results in the same automata.

Overall, the subtree filtering has a positive influence on the automaton size, but there are more efficient filter techniques, even when using the Proper-N or Proper-S item set. These constructions are presented in the upcoming sections.
DFRTA with index filtering

The construction with the index filtering technique is similar to the one with subtree filtering, but instead of creating a single table of filtered match sets that are used for indexing the transition tables, multiple filtered tables (and $\phi$-tables, but these are not discussed in detail) are constructed. The index filtering technique constructs $n$ tables, where $n$ is the maximal rank in the grammar. The $i^{th}$ table from these $n$ tables contains the match sets filtered on the subtrees that occur as $i^{th}$ child of any terminal (See Example 4.3.5).

Example 4.3.5 This example shows the effect of index filtering based on Example 4.3.2 and 4.3.3. The used grammar uses terminals with rank at most two. As described the $i^{th}$ filter table is based on the subtrees that are present as $i^{th}$ child of a terminal (the fact that they are children of a terminal implies that they are also proper subtrees). The table below shows the possible subtrees for all indices.

<table>
<thead>
<tr>
<th>Index 1</th>
<th>Index 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>${X, Y}$</td>
<td>${Y}$</td>
</tr>
</tbody>
</table>

These two sets can now be used to filter the match sets of the states for both indices:

<table>
<thead>
<tr>
<th>Phase</th>
<th>match set</th>
<th>filtered for Index 1</th>
<th>filtered for Index 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${c, X}$</td>
<td>${X}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${d, Y}$</td>
<td>${Y}$</td>
<td>${Y}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${a(X, Y), S}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>${b(Y), S}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

The table for index 1 contains 3 match sets and for index 2 only 2 sets. The index filtering technique results in a 3x2 table. The subtree filtering technique would result in a table of nine entries for a symbol of rank two. This shows the usefulness of the index filtering technique.

The construction with index filtering was performed for all grammars. Appendix E shows the results of these experiments. The focus is mainly on the Mono grammars, due to their size. These grammars do not contain terminals with a rank larger than 2, so the corresponding automata use two filtered match set tables (this can also be seen in the R-Tables column). The total number of entries in these two tables is slightly more than the number of entries in the single subtree filter table, yet each index table is smaller than the subtree filtering table. The advantage of this can be seen in the number of transitions and memory use. The number of transitions is halved with respect to subtree filtering. The results for this construction based on the Proper-N item set is summarized in Table 4.6 and 4.7.

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>47.6%</td>
<td>53.4%</td>
<td>32.2%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>42.3%</td>
<td>54.3%</td>
<td>29.0%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>46.7%</td>
<td>53.9%</td>
<td>29.6%</td>
</tr>
</tbody>
</table>

Table 4.6: Construction characteristics for the index filter construction when compared to the subtree filter construction (Constructions with Proper-N item set)
4.3. Automaton construction experiments

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>46.1%</td>
<td>52.9%</td>
<td>78.4%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>40.4%</td>
<td>53.8%</td>
<td>63.2%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>43.3%</td>
<td>51.5%</td>
<td>86.2%</td>
</tr>
</tbody>
</table>

Table 4.7: Construction characteristics for the index filter construction when compared to the standard dfrta construction (Constructions with Proper-N item set)

This table shows that the index filtering method is a good technique to reduce the size of the automaton and the construction time itself. The construction time that was added with the subtree filtering has now almost disappeared for the Mono grammars (see last column of Table 4.7 compared to same column in Table 4.5). There are two similar filtering techniques that perform even better. These filters will be discussed in the next two sections.

### DFRTA with symbol filtering

The dfRTA construction with symbol filtering is based on filtering match sets with subtrees that are present as child trees of a certain symbol. This is realized by a filter table and $\phi$-table for each symbol. Such a filter table contains all unique match sets from the states after filtering them with the criterium. This method can be very efficient when subtrees are only present as subtrees of certain terminals, because these subtrees will then be only used as indices in the tables for these symbols. Example 4.3.6 shows the effect of this filtering technique for the example grammar.

**Example 4.3.6** This example shows the effect of symbol filtering based on Example 4.3.2 and 4.3.3. The symbol filtering will create two tables. The first table contains the unique match sets after filtering them with the subtrees that occur as a child of terminal a. The second table does the same for terminal b. The table below shows which subtrees can be found in the filtered match sets.

<table>
<thead>
<tr>
<th>Symbol a</th>
<th>Symbol b</th>
</tr>
</thead>
<tbody>
<tr>
<td>${X, Y}$</td>
<td>${Y}$</td>
</tr>
</tbody>
</table>

These two sets can then be used to create the filtered tables for both symbols:

<table>
<thead>
<tr>
<th>Phase</th>
<th>match set</th>
<th>filtered for Symbol a</th>
<th>filtered for Symbol b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_0$</td>
<td>${c, X}$</td>
<td>${X}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${d, Y}$</td>
<td>${Y}$</td>
<td>${Y}$</td>
</tr>
<tr>
<td>Phase 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${a(X, Y), S}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>${b(Y), S}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

This results in a filter table with 3 match sets for terminal a and a table with 2 match sets for terminal b. This means that the transition table of symbol a contains $3 \times 3$ records and the transition table of b only contains 2 records.

The example shows that the symbol filtering technique performs slightly worse when compared to the index filtering technique (index filtering: 9 transitions (see previous section),
symbol filtering: 11 transitions), but this changes when using grammars that are targeted at instruction selection (like the Mono grammars). Such grammars contain large numbers of terminals and many of the subtrees in the rules are only found as child trees of certain terminals. This results in a large collection of very small filter tables. If one applies index filtering to these grammars then this will result in larger transition tables, because subtrees are mostly not applicable to a certain index only. They therefore result in two (the used instruction selection grammars all use terminals with rank at most two) very large filtered tables. This is illustrated by Example 4.3.7.

Example 4.3.7 This example shows the benefit of symbol filtering compared to index filtering when a grammar is used where subtrees are only present as children for certain terminals. This is the used grammar:

\[
\begin{align*}
N & = \{S, X, Y\} \\
\Sigma & = \{a, b, c, d\} \\
r & = \{(S,0), (X,0), (Y,0), (a,2), (b,1), (c,0), (d,0)\} \\
Prods & = \{(1) S \rightarrow aX, (2) S \rightarrow aY, (3) X \rightarrow b, (4) Y \rightarrow d\}
\end{align*}
\]

The set of states (with corresponding match set) for both the index-filtered and symbol-filtered DFRTA is as follows if one uses the Proper-N item set in the construction:

- \(q_0\): \{c\}
- \(q_1\): \{Y\}
- \(q_2\): \{\}
- \(q_3\): \{S\}
- \(q_4\): \{X\}

Both filter techniques can be applied to this state set to construct the transition tables. This is first done for the index filtering technique. Applying this filtering will result in two filter tables, where row \(R_{x,y}\) contains the \(y^{th}\) filtered set for index \(x\):

\[
\begin{align*}
\text{Index 1} & \quad \text{Index 2} \\
R_{1,0} & : \{c\} & R_{2,0} & : \{\} \\
R_{1,1} & : \{Y\} & R_{2,1} & : \{Y\} \\
R_{1,2} & : \{\} & R_{2,2} & : \{X\} \\
R_{1,3} & : \{X\} & & \\
\end{align*}
\]

Using these filtered items as indices for the transition tables results in \(4 \times 3\) cells for symbols of rank 2 and a table with 4 cells for a symbol of rank 1. This results for this grammar in a total number of \(16 + 2\) transitions (two additional transitions for the two symbols of rank 0).

Applying the symbol filtering technique also results in two tables, but now for each terminal with a rank larger than 1:

\[
\begin{align*}
\text{Symbol } a & \quad \text{Symbol } b \\
R_{a,0} & : \{\} & R_{b,0} & : \{c\} \\
R_{a,1} & : \{Y\} & R_{b,1} & : \{\} \\
R_{a,2} & : \{X\} & & \\
\end{align*}
\]

These filter tables are clearly smaller than the filter tables of the index filtering technique.
This is caused by the fact that the symbols X and Y are only present as children of symbol a, while symbol c is only present as a child of b. Index filtering include both matches in the filter tables for index 1 and 2 because X and Y are present at both indices.

These filtered items result in a 3×3 table for symbols of rank 2 and a table with only 2 records for a symbol of rank 1. The total number of transitions for this automaton is therefore 11 + 2 transitions.

This example shows that the symbol filtering can perform better than index filtering for a grammar were subtrees are related to specific symbols instead of specific indices. This is often the case for instruction selection grammars.

The Mono tables in Appendix E confirm this statement. The filter tables (R-Tables) contain many entries, but they are divided over a large number of tables, resulting in smaller tables and therefore smaller transition tables. Table 4.8 compares the average number of R-table entries for index filtering and symbol filtering.

<table>
<thead>
<tr>
<th></th>
<th>Average Index R-Table</th>
<th>Average Symbol R-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>44 entries</td>
<td>3.0 entries</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>28 entries</td>
<td>2.7 entries</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>35.5 entries</td>
<td>2.9 entries</td>
</tr>
</tbody>
</table>

Table 4.8: Average filter table comparison between index filtering and symbol filtering.

The average size of the tables for these grammars drops with more than a factor 10. This has an enormous impact on the number of transitions, because the average transition table of a terminal of rank 2 does have not more than 3×3 entries. This also has a large impact on the construction time. Table 4.9 compares the number of transitions, memory usage and construction time between the symbol filtering and index filtering. The same comparison is made between the symbol filtering and standard construction in Table 4.10.

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>1.3%</td>
<td>69.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>2.5%</td>
<td>112.5%</td>
<td>6.8%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>1.5%</td>
<td>92.1%</td>
<td>4.1%</td>
</tr>
</tbody>
</table>

Table 4.9: Construction characteristics for the symbol filter construction when compared to the index filter construction (Constructions with Proper-N item set)

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>0.6%</td>
<td>36.7%</td>
<td>3.9%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>1.0%</td>
<td>60.5%</td>
<td>4.3%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>0.7%</td>
<td>47.4%</td>
<td>3.6%</td>
</tr>
</tbody>
</table>

Table 4.10: Construction characteristics for the symbol filter construction when compared to the standard dfrta construction (Constructions with Proper-N item set)

As expected, the table shows a large reduction in transition table entries and construction time, caused by the smaller amount of transitions that need to be constructed. However,
the memory usage is surprising; it stays around the level of the index filtering, while the number of the number of transitions drops. This is partly caused by the extra filter tables and so-called $\phi$-tables (these are necessary for translating states to filtered match sets, when using the automaton in acceptance/parse applications). These additional data structures generate extra data. The number of $\phi$-entries is for instance 10 times larger than the number of transitions (there is an entry for each state/filter table combination). Another cause of the extra data is the overhead of the large number of dynamic Java data structures that are used for the representation of the filter tables and the $\phi$-tables.

However, this filtering technique is very attractive due to the reduced construction time. Table 4.9 shows that less than 7% of the construction time of the index filtering remains for the Mono grammars while the memory usage stays on a similar level.

**DFRTA with index & symbol filtering**

The final construction algorithm discusses a construction that uses a combination of the index and symbol filtering technique. With this filtering technique, filter tables and $\phi$ translation tables are created for each index $i$ of each terminal $a$. The match sets in such a filter table are then filtered with subtrees that are present as a subtree for that index $i$ at terminal $a$. Example 4.3.8 illustrates this filtering technique.

**Example 4.3.8** This example shows the effect of combined symbol and index filtering for the automaton of Example 4.3.2 and 4.3.3. This filtering technique will create three tables. The first two tables refer to symbol $a$, where the first table contains all filtered match sets for symbol $a$ at index 1 and the second table does the same for index 2. The third table contains the filtered match sets for index 1 of symbol $b$. The table below shows which subtrees can be found in the filtered match sets.

<table>
<thead>
<tr>
<th>Symbol a, Index 1</th>
<th>Symbol a, Index 2</th>
<th>Symbol b, Index 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>${X}$</td>
<td>${Y}$</td>
<td>${Y}$</td>
</tr>
</tbody>
</table>

These three sets can then be used to create the filtered tables:

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>match set</th>
<th>Symbol a, Index 1</th>
<th>Symbol a, Index 2</th>
<th>Symbol b, Index 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${c, X}$</td>
<td>${X}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${d, Y}$</td>
<td>$\emptyset$</td>
<td>${Y}$</td>
<td>${Y}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_3$</td>
</tr>
<tr>
<td>$q_4$</td>
</tr>
</tbody>
</table>

This results in very small filter tables, where each table contains only two match sets. The resulting transition table for symbol $a$ will only contain 4 entries and the table for symbol $b$ only 2.

The example shows that the filter tables shrink even more when applying the combined symbol and index filtering. This can also be seen in the experimental results in Appendix E. The gain between Symbol filtering and combined Symbol/Index filtering is not as large as the gain between Index filtering and Symbol filtering, but the combination still seems promising.
Table 4.11 shows the average filter table size for index, symbol and combined symbol/index filtering.

<table>
<thead>
<tr>
<th></th>
<th>Average Index R-Table</th>
<th>Average Symbol R-Table</th>
<th>Average Index &amp; Symbol R-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>44 entries</td>
<td>3.0 entries</td>
<td>2.7 entries</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>28 entries</td>
<td>2.7 entries</td>
<td>2.5 entries</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>35.5 entries</td>
<td>2.9 entries</td>
<td>2.6 entries</td>
</tr>
</tbody>
</table>

Table 4.11: Average filter table comparison between index filtering, symbol filtering and combined filtering.

Not only should these average values be considered, also the size of the transition tables is important. The transition tables for the Mono grammars are halved, but there is also a penalty: the additional filter tables and $\phi$-tables double the memory usage. Table 4.12 below shows the relative results between the symbol filtering (100\%) and combined symbol/index filtering.

<table>
<thead>
<tr>
<th></th>
<th># Transitions</th>
<th>Memory Use</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono X86</td>
<td>57.6%</td>
<td>175.9%</td>
<td>101.4%</td>
</tr>
<tr>
<td>Mono IA64</td>
<td>67.7%</td>
<td>172.0%</td>
<td>110.0%</td>
</tr>
<tr>
<td>Mono Sparc</td>
<td>66.6%</td>
<td>175.6%</td>
<td>108.6%</td>
</tr>
</tbody>
</table>

Table 4.12: Construction characteristics for the combined filter construction when compared to the symbol filter construction (Constructions with Proper-N item set)

One may conclude that the gain of the grammars that combine the two filters is negated largely by the overhead incurred. The combined filter reduces the transitions tables even more when comparing it to the symbol filtering technique, but for the Mono grammars it is more profitable to opt for the symbol filtering alone instead of the combined filtering. This combined filtering technique can however become interesting when grammars contain many subtrees that are only present in certain indices for certain terminals.

### 4.3.5 Conclusion

The goal of these experiments was to examine which of the discussed construction algorithms creates an automaton which performs well in tree acceptance and parsing, but is also constructed fast by the construction algorithm.

Constructions for nondeterministic automata offer small automata combined with very short construction times (even less then 50 milliseconds for the Mono grammars). However, nondeterministic automata are not practical in acceptance and parsing algorithms, due to the fact the nondeterminism leads to growing number of states that needs to be processed for each transition taken in the automaton. This means that accepting and parsing trees using nondeterministic automata can take a lot of time.

The deterministic automata eliminate this problem. However, as shown, the standard DFRTA construction is disappointing. The standard construction constructs automata that are too large for real life applications, even when one uses a Proper-$N$ or Proper-$S$ item set. Fortunately there are promising results for the constructions that use filtering techniques to
reduce the size of the transition tables in DFRTAs. Especially the construction with symbol filtering, which takes advantage of subtrees that can only be found as child trees of certain terminals (which often the case for instruction selection grammars). This symbol filtering reduces the number of transitions to 0.6% of these in the standard construction for the Mono X86 grammar (using the Proper-N item set in both cases).

Figures 4.23 and 4.24 give an overview of the size of the DFRTAs and NFRTAs constructed from the Mono grammars using the Proper-N item set. They show how the number of transitions and memory usage changes when the different filter techniques are applied for the constructions of the Mono grammars.

![Figure 4.23: Relative number of transitions of the Mono DFRTAs and NFRTA constructed with the Proper-N item set](image)

![Figure 4.24: Relative memory usage of the Mono DFRTAs and NFRTA constructed with the Proper-N item set](image)

It depends on the shape of the grammar which filtering technique is the most efficient. Automata for grammars that contain subtrees that are only present for certain indices should
be constructed using index filtering. If the subtrees are present for certain terminals, then
one should opt for symbol filtering. Combined filtering becomes profitable for grammars that
have both properties.

Summarized, the most important fact is that the DFAAs constructed with a well-chosen
filtering technique not much larger (around a factor 3 for the Mono grammars) than the
nondeterministic variants while still offering the property of being deterministic. Automata
constructed with these filter based construction algorithms are therefore very promising for
real world acceptance, matching and parsing applications.

4.4 DFRTA based tree parsing experiments

The automata discussed in this thesis are very useful for solving the tree acceptance and
tree parsing problem. This section discusses the experiments that were carried out with an
algorithm, implemented in the toolkit, that determines the lowest cost parse of an input tree
with the help of a DFRTA. This parsing algorithm [FSW94, Section 7] can solve the parsing
problem with each type of DFRTA discussed in Section 4.3.4.

In this section we discuss the parsing algorithm itself and provide example parses for a
DFRTA constructed with the All-Sub item set and the Proper-S item set. The differences in
filtering techniques can not be seen in the parsing algorithms because they are encapsulated
by the automata. These differences become only visible when measuring the execution speeds
with these different filtering techniques. These measurements will be presented in Section
4.4.2.

4.4.1 The parsing algorithm

The parsing algorithm consists of two phases. The first phase labels the subject tree from
bottom to top using the DFRTA. This labeling stores for each node which rules can be applied
to produce the subtree rooted at that node. Using these labels, the second phase determines
from top to bottom which sequence of rules results in the lowest cost parse of the tree.

This algorithm is not exactly the same for each type of DFRTA. There are some small
adaptations needed when using an automaton created from the Proper-N or Proper-S item
set, instead of the All-Sub item set. We start with discussing the two phases for the All-Sub
item set in the two upcoming sections. In the final section we discuss the changes necessary
for working with Proper-N and Proper-S.

First phase

The first phase processes the subject tree with the DFRTA in a frontier-to-root direction.
During the traversal each node is linked to a state (as by a normal acceptor). Each leaf node
with a symbol \( x \) is linked to the state with the incoming \( x \)-transition. The rest of the tree is
linked as one would expect by taking transitions that match the symbols that are encountered
in the tree. Let us illustrate this by constructing a DFRTA for the grammar in Example 1.1.3.
Figure 4.26 repeats the production rules and shows the match sets of the states, while Figure
4.25 contains the complete automaton (without trap state \( q_2 \)).
Chapter 4. Experiments

4.4. DFRTA based tree parsing experiments

Figure 4.25: DFRTA constructed with All-Sub

\[
\begin{align*}
(1) & \quad S \rightarrow a, \\
(2) & \quad S \rightarrow a, \\
(3) & \quad S \rightarrow c, \\
(4) & \quad B \rightarrow b, \\
(5) & \quad B \rightarrow S, \\
(6) & \quad B \rightarrow d
\end{align*}
\]

(a) Production rules of the grammar

\[
\begin{align*}
q_0 \{ c, B, S \}, & \quad q_1 \{ d, B \}, & \quad q_3 \{ b, b, B \}, & \quad q_4 \{ a, B, S \}, \\
q_5 \{ b, B \}, & \quad q_6 \{ a, B, S \}, & \quad q_7 \{ a, a, B, S \}
\end{align*}
\]

(b) Match sets of automaton in Figure 4.25

Figure 4.26: Production rules and match set of the constructed automaton.

The subject tree that will be parsed with the help of the automaton can be seen in Figure 4.27(a). For this example the algorithm starts assigning states \( q_0 \) and \( q_1 \) to the leaf nodes of the tree. When proceeding at the left, by reading the \( b \) node, one arrives in state \( q_3 \). This process is repeated until the complete tree is visited (see Figure 4.27(b)).
However, the goal of this phase is not just to label all nodes with states, but to retrieve the production rules that should be applied in each node \( n \) to produce the subtree \( \alpha \) rooted at \( n \) as cheap as possible. The algorithm therefore not only links a state to node when it visits a node, but also records, by using the match set of the state, which rules could be applied.

Let us for now simplify the problem by ignoring chain rules and costs, and by only looking at standard rules. A subtree \( \alpha \) rooted at a node \( n \) can be produced with an arbitrary nonterminal as starting point. So, what the algorithm does is store in each node for every nonterminal \( m \) in the match set of the linked state to each production rule \( r \). If the RHS of \( r \) is equal to \( m \), then \( r \) can be used to produce \( \alpha \) from the LHS nonterminal of \( r \).

Let us illustrate this by using the example subject tree. Node \( b \) is matched by \( q_3 \), where \( q_3 \) contains three matches. The match \( b(B) \) can be found as RHS of rule 4 (see Figure 4.28). This means that node \( b \) can be produced from nonterminal \( B \) (the LHS of rule 4) by applying rule 4. Figure 4.29 shows the result when these production rules are gathered for all the nodes.
The whole subtree rooted at node \( b \) can now be obtained from \( B \) by applying rule 4 and recursively applying rules for the nonterminals in the \( \text{rhs} \) of rule 4. Rule 4 for instance contains the nonterminal \( B \). This nonterminal overlaps with node \( c \) in the subject tree. This means that the \( B \) production rule for this node must be applied. However, there is no such rule due to the fact that we ignore chain rules (\( B \rightarrow S; S \rightarrow c \) would be a possibility). After handling the chain rules node \( c \) will contain \( B \rightarrow S; S \rightarrow c \) for nonterminal \( B \) (see Figure 4.30) and the complete subtree \( b(c) \) can be obtained from nonterminal \( B \) by applying: 

\[
B \rightarrow b(B); B \rightarrow S; S \rightarrow c
\]

A remaining question is: how does the algorithm handle chain rules? Like for the automaton constructions, this is done by computing the nonterminal closure. If a production rule \( r \) is added for a node, as described above, we compute the closure based on the \( \text{lhs} \) nonterminal of that rule. For the nonterminals in the closure we store the sequence of chain rules that have to be applied plus the production rule \( r \).

The nonterminal closure for nonterminal \( S \) is for instance computed when rule \( S \rightarrow c \) is added. This closure computation will tell us that chain rule \( B \rightarrow S \) enables the possibility to produce subtree \( c \) from nonterminal \( B \). Figure 4.30 shows the additional matches that are created by chain rules for the example grammar.

![Figure 4.30: Subject tree with matching rules and chain rules](image)

The last aspect taken care of by the algorithm is to provide the lowest cost parse. It could for instance be possible to produce a subtree rooted at a node \( n \) by applying different rules from the same nonterminal (see for instance the two possibilities for nonterminal \( S \) in the root node in Figure 4.30). What we want is only a lowest cost sequence of rules.

Let us illustrate the effects of these costs by assigning a cost of 1 to each rule. While determining the matching rules for each node we then can also compute the cost of using that rule. This cost is determined by the cost of the rule itself and the costs that are registered in the nodes that correspond to nonterminal leaves of that production rule.

The \( c \)-leaf for example, can be produced from \( S \) by a single rule and therefore has cost one. Producing \( c \) from \( B \) is more expensive due to the additional rule and therefore has cost 2. The \( b \)-node above \( c \) can be produced from nonterminal \( B \) by applying rule \( B \rightarrow b(B) \). The rule itself is of cost 1, but we also have to look at the cost for nonterminal \( B \) in the \( \text{rhs} \) of the rule. This nonterminal corresponds to the \( c \)-node below, so we have to add the cost of \( B \) in this \( c \)-node. This results in a final cost of 3.

This way the cost is computed for each sequence of rules added to a node. Figure 4.31 shows the costs for all rules in the tree.
4.4. DFRTA based tree parsing experiments

The root node contains two sequences for both nonterminal $B$ and $S$. However, we are only interested in the lowest cost solution, so we do not have to store the more expensive solutions. The algorithm therefore stores only one record for each nonterminal in a node. This record is updated if a production rule (or a sequence in case chain rules are involved) is encountered that has a lower cost. This first stage of the algorithm therefore results in the following labeled subject tree:

Figure 4.32: Subject tree after phase 1

Summarized, this leads to the following approach. For each node $n$ we determine the corresponding state $q$. For every match $m$ in the match set of $q$ we gather all production rules $r$ that have a RHS that is equal to $m$. If applying $r$ is cheaper than the current solution for the LHS nonterminal, then we replace it by this new rule $r$. Finally we compute the nonterminal closure with costs, by using the Floyd-Warshall algorithm [Flo62], and update the nonterminals in $n$ if the chain rule sequence plus $r$ is cheaper than current value for that nonterminal. This results in the following recursive method that visits the nodes of the subject tree:
Chapter 4. Experiments 4.4. DFRTA based tree parsing experiments

LABELTREE(n, a, rtg)

**Input:** A node n from the subject tree, a DFRTA a and the regular tree grammar rtg from which the automaton is constructed and the parse should be determined.

**Output:** The automaton state of n (and the subtree rooted at n is annotated).

1. \( \text{vector} \leftarrow \emptyset; \)
2. \( \text{for } i \leftarrow 0 \text{ to } n.\text{children}.\text{count} \)
3. \( \text{vector}[i] \leftarrow \text{LABELTREE}(n.\text{children}[i], a, rtg) \)
4. \( q \leftarrow a.\text{nextState}(n.\text{symbol}, \text{vector}); \)
5. \( \text{foreach } \text{match} \in q.\text{matches} \)
6. \( \text{foreach } \text{rule} \in rtg.\text{productionRules} \text{ where } \text{rule.rhs} = \text{match} \)
7. \( \text{cost} \leftarrow \text{rule.cost} + \sum_{l \in \text{match.leaves}} \text{leafCost}(n, l); \)
8. \( \text{if } \text{cost} < n.\text{annotation}[\text{rule.lhs}].\text{cost} \)
9. \( n.\text{annotation}[\text{rule.lhs}].\text{cost} \leftarrow \text{cost}; \)
10. \( n.\text{annotation}[\text{rule.lhs}].\text{rules} \leftarrow \text{rule}; \)
11. \( \text{foreach } s \in rtg.\text{alphabet} \text{ where } s.\text{type} = \text{nonterminal} \)
12. \( \text{costAndRules} \leftarrow \text{getClosureCostAndRules}(\text{rule.lhs}, s); \)
13. \( \text{if } (\text{costAndRules.cost + cost}) < n.\text{annotation}[s].\text{cost} \)
14. \( n.\text{annotation}[s].\text{cost} \leftarrow \text{cost + costAndRules.cost}; \)
15. \( n.\text{annotation}[s].\text{rules} \leftarrow \text{ruleconcat}\text{costAndRules.rules}; \)
16. \( \text{return } q; \)

**Second phase**

The second phase is less complex compared to phase one. This phase traverses the subject tree starting at the root with the tree grammar start nonterminal and applies the rule listed for that nonterminal in the root node. The algorithm executes the same process for the nonterminals that can be found in the production rule that is applied. These nonterminals correspond to certain nodes in the subject tree and the rule for that nonterminal in this node can then again be applied.

Let us clarify this with the example subject tree of phase one. The start nonterminal of the used grammar (see Example 1.1.3) is the nonterminal \( S \). The algorithm therefore starts by applying rule \( S \rightarrow a(b(c), B) \) to the start nonterminal (see Figure 4.33 and 4.34). This results in a tree with one nonterminal \( B \). This nonterminal is on the location of the \( d \) node in the subject tree. We therefore proceed by applying the production rule listed for \( B \) in this node: \( B \rightarrow d \). This finally results in the complete subject tree.
Adaptations for Proper-N and Proper-S

DFRTAs created with the Proper-N and Proper-S item set differ from All-Sub, because the match sets of the states do not contain all RHS trees of production rules as subtrees. This creates a problem because the algorithm compares in the first phase all subtrees from the match sets with the RHS of the production rules. The absence of some of these trees means that some rules are not coupled to nodes.

To illustrate this problem we again parse the subject tree of Figure 4.27(a), but now by using an automaton constructed from the Proper-N item set. Figure 4.35 shows this automaton (again without trap state $q_2$) and Figure 4.36 shows the corresponding match sets.
Chapter 4. Experiments 4.4. DFRTA based tree parsing experiments

\[
q_0 \{ c, B, S \} \\
q_1 \{ d, B \} \\
q_3 \{ b, B \} \\
q_4 \{ c, B, S \} \\
q_5 \{ B \}
\]

Figure 4.36: Match sets of automaton in Figure 4.35

Linking the nodes to states is done in the same way as when using the All-Sub item set. Figure 4.37 shows the resulting subtree. The problem becomes visible if we start looking at the match set for state \(q_3\) for the \(b\)-node of the subject tree. The match set of this state does not contain the \(b(B)\) as in the All-Sub case, because it is no proper subtree. The result of this is that the rule \(B \rightarrow b(B)\) will not be registered in this node if we would proceed with the standard approach.

\[
q_4 - a \\
q_3 - b \\
q_0 - c \\
d - q_1
\]

Figure 4.37: Subject tree with matching states

Luckily there is a solution to this problem. Instead of comparing the subtrees in the match set of a node \(n\) to the RHS of the production rules, we compare the RHS of the production rules to the match sets of the child nodes of \(n\) in combination with the symbol at \(n\). In other words the algorithm has to check two things in node \(n\) for each production rule \(r\):

- Is the symbol of the root node of the RHS of \(r\) equal to the symbol of node \(n\).
- Is there for each subtree at index \(i\) of the root node of \(r\) an equal subtree in the match set of the \(i^{th}\) child node of \(n\).

If all these checks are positive then the production rule \(r\) can be used to produce the subtree rooted at \(n\). Let us look at the \(b\)-node in the subject tree and rule \(B \rightarrow b(B)\) to clarify this approach (see Figure 4.38). At first we have to compare the symbol \(b\) of the node to the root node of the production rule. These are both \(b\), so now we have to compare the subtrees in the match set of the \(c\)-node (the first and only child node of \(b\)) to the first subtree of the root node of the production rule \((B)\). The \(c\)-node has as match set \(\{ c, B, S \}\) in which \(B\) is contained. Thus the first child node of \(n\) matches the first child node of the RHS of the production rule. This production rule therefore can be used to produce the \(b\)-node.
So instead of comparing two subtrees the algorithm has to compare two symbols and a collection of subtrees. The rest of the algorithm is equal to the All-Sub variant. This small difference makes the Proper-N/Proper-S variant more complex than the All-Sub variant. This could result in longer running times for the Proper-N/Proper-S variant. However, Proper-N and Proper-S automata can be constructed faster than All-Sub automata. Section 4.4.2 compares the two algorithms and measures if the gain caused by the construction algorithms for the Proper-N and Proper-S automata is not removed by the more complex parsing algorithm.

### 4.4.2 Automaton comparison

The parsing algorithm described can parse trees with the help of all types of DFRTAs that are described in Section 4.3.4, but the algorithm can perform differently depending on the type of DFRTA used. As discussed earlier there are two variants of the algorithms. One variant that only can be used for automata constructed with the All-Sub item set and another variant that can also be used for automata constructed with the Proper-N and Proper-S item set. Furthermore there are different filter techniques that are used inside the automata. These filter techniques are not noticeable for the algorithm. However, these techniques can influence the performance of the algorithm, because they influence the state retrieval complexity.

This section therefore measures the running time of the parse algorithm when using different types of DFRTAs. We have chosen to execute this parsing algorithm for the Mono X86 and Mono IA64 grammar in combination with a collection of generated trees for each of the grammars. These collections contain one hundred trees with respectively a total of 71504 and 73248 nodes.

Figure 4.39 shows the running time of these parses for both grammars.
Chapter 4. Experiments

4.4. DFRTA based tree parsing experiments

The graph shows that all filter techniques perform similarly and that differences are mostly found between the automata constructed from the All-Sub item set and the automata constructed from the Proper-N or Proper-S item set. This performance difference for the item sets is caused by the two variants of the parsing algorithm for the item sets. Table 4.13 shows the relative parse time differences for the Proper-S item set compared to the All-Sub item set.
4.4. DFRTA based tree parsing experiments

<table>
<thead>
<tr>
<th>Filtering technique</th>
<th>X86</th>
<th>IA64</th>
</tr>
</thead>
<tbody>
<tr>
<td>No filtering</td>
<td>137.0%</td>
<td>124.1%</td>
</tr>
<tr>
<td>Subtree filtering</td>
<td>137.7%</td>
<td>125.8%</td>
</tr>
<tr>
<td>Index filtering</td>
<td>137.4%</td>
<td>125.6%</td>
</tr>
<tr>
<td>Symbol filtering</td>
<td>136.1%</td>
<td>124.2%</td>
</tr>
<tr>
<td>Symbol &amp; index filtering</td>
<td>135.0%</td>
<td>122.7%</td>
</tr>
</tbody>
</table>

Table 4.13: Parse time with DFRTAs constructed with Proper-S item set compared to the All-Sub item set, for the different filter techniques

These results show that parsing with DFRTAs constructed from the Proper-S (and also Proper-N) set is between around 22% and 38% slower compared to their All-Sub variant for these two example collections of trees. Section 4.3.4 however shows that DFRTAs can be constructed faster when using the Proper-N or Proper-S item set instead of All-Sub. It depends on the usage whether Proper-N/Proper-S is most efficient or All-Sub. If the DFRTA is constructed for parsing a single small tree, then the construction time of the DFRTA will have the largest impact on the running time. When using a single automaton many times for parsing large trees then it could be wiser to opt for the All-Sub item set because most of the processor time is consumed by the parsing algorithm.

Summarized, the filtering technique used in the DFRTAs does not have a noticeable impact on the running time of the parsing algorithm. The used item set however has an impact, but it depends on the size of the subject trees and the number of parses executed for a single DFRTA which item set is the most profitable.
5 Conclusions

This chapter will give an overview of the achievements and results, and provide suggestions for future research. The chapter will be finalized by an evaluation of the project.

5.1 Results

The master's project resulted in a toolkit, GUI and a collection of results from the conducted experiments. Before providing an overview of the conclusions of the experiments we highlight the key features of the constructed toolkit and GUI.

Toolkit and GUI

The ForestFIRE toolkit implements a large collection of data structures and algorithms related to trees. This collection contains the data structures necessary for representing the domain concepts (e.g. trees, tree grammars, tree patterns, and tree automata) and implementing the tree algorithms. The tree algorithms themselves mainly focus on three areas:

- Tree grammar transformations
- Tree automaton constructions
- Tree parsing and acceptance

The group of transformation algorithms consists of algorithms that remove chain rules (RED-U), non-root terminal nodes (RED-Z), and useless symbols and/or production rules. The second group contains algorithms that construct $(\varepsilon)NRFTAs$, $(\varepsilon)NRFTAs$ and (optimized) $DFRTAs$ from tree grammars. The final group consists of a set of algorithms that solve the tree acceptance and tree parsing problem by using tree automata. Next to these algorithms a collection of algorithms was implemented that support these algorithms.

Let us look back at the original requirements for the toolkit. As described in the introduction of this thesis the goal was to implement as many features as possible, from the features listed in Appendix A.

The features listed in the original assignment description (Section A.1) are divided into three groups with each a different priority. Almost all data structures and algorithms listed in the most important group are implemented. Tree automaton constructions that construct automata from patterns set, and constructions based on dotted rules/trees are not implemented due to the restricted time frame. Unfortunately there was also not much time left for the other two groups. This resulted in only one implemented feature from these groups. This feature was the operation for generating random trees from tree grammars.
However, the remaining features can be implemented very easily, due to the large collection of basic data structures and algorithms the toolkit provides. This advantage was also encountered during the implementation of the parsing algorithm. The availability of tree automata and operations to compare trees made it possible to implement the parser within a few hours.

The additional list of requirements in Section A.2 describes requirements for the data structures that implement the basic domain concepts. These requirements consisted of special operations and wellformedness rules. All these can be in ForestFIRE as respectively class methods and invariants.

The FIREWood GUI created to the toolkit was a very useful addition to the toolkit, because it provides easy access to the algorithms inside the toolkit. Furthermore the GUI provides the possibility to load user defined trees, tree grammars in special text files. FIREWood converts these structures into corresponding ForestFIRE data structures. These structures can then be used as input for one the algorithms.

FIREWood was a very useful addition to the toolkit during the experiments, but it was not just targeted at the currently implemented algorithms. FIREWood was, like ForestFIRE, built with extendibility in mind. This means that new algorithms implemented for ForestFIRE can be made accessible very easily.

I therefore want to conclude by saying that ForestFIRE and FIREWood are practical tools for experimenting with the currently implemented algorithms, and that they have a large potential for supporting an even larger collection of tree algorithms.

**Tree grammar transformation experiments**

The tree grammar transformation experiments focused on different optimization techniques and orders for removing chain rules (using RED-U) and non-root terminal nodes (using RED-Z) from a grammar. This section contains a compact overview of the results of these experiments:

1. The optimal transformation order (RED-Z/RED-U) depends on the grammar shape, when no reuse is applied in the RED-Z steps.
2. Reusing nonterminals and production rules in RED-Z(*) reduces the number of rules to 64–78% of the no-reuse case for the used Mono grammars.
3. The *shortest tree first* node selection strategy for the RED-Z transformation steps results in the most efficient reuse.
4. The *shortest tree first* reuse removes the differences between the different transformation orders.

**Tree automaton construction experiments**

The tree automaton experiments measured the efficiency and size of resulting tree automata for different automaton construction techniques. This is an overview of the most important conclusions of these experiments:
1. Constructions of nondeterministic automata are considerably faster than the constructions of deterministic ones.

2. Constructions of NFRTAs/NRFTAs for the Mono grammars result in smaller automata (e.g. 38–39% less transitions) when using the Proper-N or Proper-S item set instead of the All-Sub item set.

3. The Standard DFRTA construction is unusable in practice due to long running times and extreme memory use.

4. Filtering techniques for the DFRTAs reduce memory usage and construction time such that constructions perform almost as good as the constructions for the nondeterministic automata.

5. For the Mono grammars the symbol filtering technique performed the best. This filter technique will probably also perform well for other grammars related to instruction selection.

**Tree parsing experiments**

The final group of experiments measured the effect of using different types of DFRTAs in a parsing algorithm that solves the parsing problem using these automata. These are the main results of these experiments.

1. DFRTAs constructed with the All-Sub item set instead of Proper-N or Proper-S perform better, due to needed adaptations to the parsing algorithm for the automata created with Proper-N or Proper-S. The adaptations resulted in a 24–38% longer running time.

2. There is no remarkable difference when using different filter techniques in the DFRTAs.

### 5.2 Recommendations for future work

This master project raised some new questions related to the three groups of experiments. This section provides a list of interesting research subjects divided over these three experiment subjects.

**Tree grammar transformations:**

1. Measure the effect of reusing original nonterminals with more than one occurrence as LHS in the RED-Z(*) transformation.

**Tree automaton constructions:**

1. Measure the influence of tree automaton constructions with U−/Z− grammars instead of U+/Z+.
2. Compare constructions based on subtrees to constructions based on dotted rules.
3. Study additional transition table reduction techniques.

**Tree parsing:**

1. Investigate optimization techniques for parsing with automata constructed from the Proper-N and Proper-S item set.
2. Compare the implemented parsing algorithm to other existing tree parsing algorithms.

### 5.3 Evaluation

This master’s project was completed in eight months. As each project, this project had some pleasant parts and less pleasant parts. Before discussing these there needs to be mentioned that the topic of tree algorithms is a very interesting topic due to its broad application area (instruction selection, term rewriting, genetics etc.).

The toolkit building process and the experiments were the nicest parts of the project. Especially when the construction of the toolkit was near to its completion. The final algorithms could then be implemented elegantly by using already implemented data structures and algorithms. Warshall’s algorithm is, for instance, implemented in a single class. This class could be reused in many other algorithms.

The experiments had the pleasant property that they helped to tune the implementation of the more complex algorithms, like the automaton construction algorithms. Another advantage of these experiments was that they revealed unexpected behavior. An example of this can be seen with the grammar transformations. The expectance was that a certain RED-Z/RED-U order would always perform equal or better than another order. The experiments with these transformations suddenly provided contradiction results. This was probably not discovered without the experiments.

Unfortunately, there were also some drawbacks. The initially chosen programming language Lazarus resulted in many problems, because its LCL library was not as platform independent as claimed by its developers. This resulted in some delay in the development process, because all Lazarus code was converted to Java code. Fortunately, this went faster as expected due my C# knowledge.

Overall, it was a pleasant and instructive project, which introduced me to the topic of tree algorithms and which with the cooperation of Loek Cleophas and Kees Hemerik resulted in practical toolkit and some useful experimental results.
A MSc Assignment description

This appendix contains the description of the original master’s assignment and an additional list of requirements that described the main goals of this project. The assignment description introduces the domain and provides three lists of features and experiments, sorted on importance, that needed be implemented/performed. The list with additional requirements focuses on specific operations that should be implemented for the data structures that represent the main domain concepts. This list contains operations like: ‘retrieve the number of chain rules in a grammar’.

A.1 Original Assignment description

Context

The MSc assignment will take place in the context of the PhD research of Loek Cleophas, which is related to regular tree languages. The area of regular tree languages has a rich theory, with many results that are generalizations of regular string languages, and many relations between the two areas. Parts of this theory have broad (potential) applicability in a number of areas, among which are code generation in compilers—particularly for instruction selection or optimization—and term rewriting.

Underlying these and other practical applications are the following three important algorithmic problems within the field of regular tree languages:

1. Tree acceptance. Given a regular tree grammar and an input tree, determine whether the input tree can be generated by the regular tree grammar i.e. is part of the language denoted by the regular tree grammar.

2. Tree pattern matching. Given a finite, non-empty set of trees (the pattern set) and an input tree, find the set of all occurrences of the patterns in the input tree.

3. Tree parsing. Given a regular tree grammar and an input tree, determine all parses of the input tree that can be generated by the regular tree grammar. A variation of this problem that is often used is to determine a parse that is optimal (with respect to some cost function).

These problems are related—e.g. tree parsing generalizes/extends tree acceptance and tree pattern matching is used in tree acceptance algorithms—and involve many of the same algorithmic ingredients. Many algorithms solving these problems have been described in the literature. Unfortunately, a number of deficiencies exists:

1. The field is rather unaccessible. Much of the theory is scattered over the literature, with only a few overview publications, and none of them being algorithm oriented.

2. The algorithms that have been published are hard to compare due to differences in presentation style and level of formality.
3. Many practical algorithms have been published with little or no reference to the theory or correctness arguments.

4. No large collection of implementations of the algorithms solving these algorithmic problems exist.

5. It is hard to choose between different algorithms for practical applications.

To solve these deficiencies, an overview of relevant parts of the theory has been constructed and literature research to find algorithms solving the above problems has been performed. Based on these, the algorithms have been rephrased in a common presentation style and a preliminary classification of algorithms for tree acceptance (currently bottom-up/frontier-to-root ones only) and tree pattern matching has been created. Taken together, this work helps to solve the first three deficiencies mentioned.

Assignment Goal & Expected Results

The proposed MSc assignment has the broad goal of providing a starting point in solving the fourth and fifth deficiencies, based on the results from Cleophas’s PhD research. Ideally it should result in:

- an (extendable) collection of algorithms & data structures related to trees, in the form of a toolkit (‘Forest FIRE’); this collection should implement foundational data structures & algorithms related to tree acceptance, tree pattern matching and tree parsing algorithms; time permitting, some of these algorithms themselves should be implemented;
- a graphical environment (‘FIRE Wood’) to experiment with and compare the implemented algorithms & data structures;
- a comparison of the efficiency and trade-offs involved in such algorithms & data structures;
- reporting on the assignment, in the form of regular meetings with the tutor and/or supervisor, an intermediate presentation, and an MSc thesis together with its oral presentation and defense.

The extent to which these results can be obtained will likely be limited by the fixed amount of time available for an MSc project (1120 hours, i.e. 40 ECTS * 28 hours).

Realistically, the extent to which the first three results are completed is likely to be limited. Further on, features will be classified as either required, desired or nice to have. The required features together with the mandatory reporting form the minimal requirements for completion of the assignment. (Even if desired or nice to have features are excluded due to time constraints, the design & implementation of included features should be extendable so excluded features can be included at a later time.)

Planning

A detailed planning should be created by the student and discussed with his tutors/supervisors during the first phase of the assignment. Roughly, the assignment will be divided into the following phases:

1. read & investigate: draft PhD thesis, existing implementations (A Terms, Timbuk, Treebag, TWIG, BEG, iBURG, . . .); create detailed planning
2. define general toolkit & GUI structure, interfaces, representations (for required features, taking desired and nice to have ones into account), decide on implementation language (Delphi? Java? ...?); report on this

3. implement required features; report on this

4. experiment with resulting toolkit & GUI; report on this

5. time permitting, repeat last two steps for desired features and for nice to have features

6. create final report, give presentation, thesis defense

As mentioned earlier, the total amount of time available for the assignment is 1120 hours; how this time is (planned to be) divided over the various phases will be part of the detailed planning created in the first phase.
Required Features

Data structures

- trees over ranked alphabets; without variables, with a single variable, with multiple variables (?)
- sets of trees (pattern sets)
- regular tree grammars (RTGs), consisting of a nonterminal set, a ranked terminal alphabet, production rules (form \( A \rightarrow \alpha \), or more general \( \beta \rightarrow \alpha \)) and start nonterminal
- tree automata (TAs); directed frontier-to-root (FR), root-to-frontier (RF); nondeterministic with and without epsilon-transitions (epsilon-less NFRTA / NRFTA), deterministic (DFRTA / DRFTA)

Issues:
- what alternative representations are there? look at literature to see what is used and why; other representations? (dis)advantages of them
- for TAs in particular, how to represent transitions tables; different per TA kind? how to compress transition tables (e.g. use general table compression techniques, use filtering techniques based on possibility of subpatterns occurring as \( i^{th} \) son of symbol \( a \), use maximal subterm sharing of ATerms?)

Input/Output

- ability to define ranked alphabets, (ordered ranked) trees, pattern sets, nonterminal sets, production rules, RTGs
- ability to import and export trees, pattern sets, RTGs, TAs

Algorithms

- RTG transformations
  - removing unit productions (e.g. \( A \rightarrow B \))
  - removing productions whose rhs has any non-root node labeled by a terminal symbol
- TA constructions & transformations
  - construction of TA based on RTG
  - construction of TA based on tree pattern set
  - construction based on subtrees vs. on dotted trees/production rules
  - directed FR, RF; nondeterministic(?), deterministic

Experiments

- investigate influence of RTG transformations on RTG size, influence of order of the transformations; effect of transformations on resulting TAs
- compare time and space efficiency of TA constructions
Desired Features

Data structures

- Aho-Corasick automata (ACAs); basic goto-version, version specific to use for tree stringpath matching (see Prague Stringology Conference 2005 paper)

Issues:
- what alternative representations are there? look at literature to see what is used and why; other representations? (dis)advantages of them

Input/Output

- ability to generate (pseudo-random) trees, pattern sets, RTGs

Algorithms

- TA constructions & transformations
  - allow separate transformations to remove epsilon-transitions, perform (reachability based) subset construction, in addition to direct construction of epsilon-less and deterministic TAs?
- ACA constructions (see PSC 2005 paper)
  - basic goto-version
  - stringpath-specific version

Experiments

- compare time and space efficiency of TA and ACA constructions
- compare RTG transformations followed by TA construction to direct TA construction followed by TA transformations
'Nice to Have’ Features

Data structures
- trees over sorts
- rewrite rules
- extension to hedge trees/ordered unranked trees (trees whose nodes have a list of children; these can be used to model XML trees)

Input/Output
- ability to define, import, export rewrite rules
- ability to generate C code for rewriting

Algorithms
- TPMn/TGAn algorithms (tree pattern matching resp. tree (grammar) acceptance, for every subject node)
  - on-the-fly FR/BU
  - FR/BU using DFRTA; with/without filtering, other compression techniques (?)
  - RF/TD (see PSC 2005 paper)
    * using DRFTA
    * using ACA
    * using stringpath-specific ACA

Experiments
- compare speed of TPMn/TGAn algorithms using different TA/ACA kinds
- effect of TA transformations on TPMn/TGAn algorithm speed
A.2 Additional data structure requirements

This section contains a list of requirements for the domain concepts that needed to be implemented in the toolkit. This list was defined in addition to the original assignment description that can be found in Section A.1

Original content:

This document gives a brief overview of the types of analyses, transformations and other operations needed for trees, pattern sets and Regular Tree Grammars, together with references giving more details on their definition or use. It does not include very basic analyses and operations e.g. creating these objects or counting the number of alphabet symbols.

Trees

- analysis
  - wellformedness with respect to a ranked alphabet: does a tree use symbols from the alphabet only, and respect the ranking function? (likely not an analysis to be invoked explicitly; instead, to be enforced at tree creation time)

Pattern sets

- analysis
  - number of patterns
  - wellformedness, see above
- operations
  - obtain dotted tree set for it (see "Tree notions" in handwritten notes, IJFCS paper Definition 11)
    * analysis:
      - number of elements
      - number of elements when flattened (see handwritten notes for definition of flattening function \( fl \))
    * operation: flattening (to obtain subtree set for pattern set, as below)
  - obtain subtree set for it (see IJFCS paper page 4 underneath Def. 4; expressible using \( fl \) and previously mentioned operation to obtain dotted tree set)
    * analysis:
      - number of elements

Regular Tree Grammars

- analysis
  - number of
    * rules
A.2. Additional data structure requirements

Chapter A. MSc Assignment description

* nodes, summed over all rules
* chain rules
* non-root nodes labeled by terminal symbol, summed over all rules
* rules with non-root nodes labeled by terminal symbol

- wellformedness, see above
- existence of (can be expressed in terms of "number of" analyses)
  * chain rules
  * non-root nodes labeled by terminal symbol, over all rules

- "usability" (see thesis draft, Section 4.3.1.2 Removing Useless Symbols and Productions)
  * reachable/unreachable
    - terminals
    - nonterminals
    - production rules
  * productive/unproductive
    - terminals
    - nonterminals
    - production rules
  * useful/useless: useful if and only if reachable and productive, useless if and only if not useful

- transformations; for each one, allow it to be performed one rule at a time (either with manual or with automatic pseudo-random selection), or for all rules at once; default behavior is to make copy of RTG and perform transformations on this copy, perhaps provide setting to toggle between this and not making a copy & modifying the original RTG

  - remove useless (see Transformation 4.3.21 in Section 4.3.1.2)
    * productions
    * nonterminals
    * terminals
  
  - perhaps variant of remove useless that only removes useless productions & nonterminals, leaving associated terminal alphabet unchanged
  
  - remove chain rules (see Sections 6.1.1 and 6.1.2)
  
  - remove rules with non-root nodes labeled by terminal symbol (see same)

- (other) operations

  - obtain dotted rule set for it (see "Tree notions" in handwritten notes)
    * analysis:
      - number of elements
number of elements when right-flattened (see handwritten notes for definition of right-flattening function rfl)

* operation: right-flattening (to obtain subtree set for rule’s right hand sides, as below)

- obtain right hand sides’ subtree set for it (see handwritten notes; expressible using rfl and previously mentioned operation to obtain dotted rule set)

* analysis:
  - number of elements
B Formal definitions

This appendix contains all formal definitions of the concepts described in the Domain Chapter. All these definitions are taken from [Cle07].

B.1 Tree related definitions

**Definition B.1.1 (Tree domain)** Given a set of edge labels $E$, a tree domain is a finite non-empty subset $D$ of $E^*$ such that $\text{pref}(D) \subseteq D$, i.e. $D$ is prefix-closed. In particular, $\varepsilon \in D$ for any tree domain $D$.

We use $\diamond$ to indicate concatenation of elements of $E$. Unless explicitly noted otherwise, we assume the edge label set $E$ to be $\mathbb{N}_+$, the positive natural numbers.

**Definition B.1.2 (Tree)** Given a tree domain $D$ and an alphabet $\Sigma$, a (node labeled) tree $t$ is a function $t \in D \rightarrow \Sigma$. We use $t(n)$ for the label of a node $n \in D$.

**Definition B.1.3 (Ranked alphabet)** A ranked alphabet is a pair $(\Sigma, r)$ such that $\Sigma$ is an alphabet (a finite, non-empty set of symbols) and $r : \Sigma \rightarrow \mathbb{N}$ is a ranking function. For $a \in \Sigma$, we call $r(a)$ the rank or arity of $a$.

**Definition B.1.4 (Ranked Tree)** A ranked tree is a node labeled tree $t$ whose alphabet is a ranked alphabet $(\Sigma, r)$ and for which, for all $n \in D$,

$$r(t(n)) = \langle \# i : i \in E \land n \diamond i \in D_t : i \rangle.$$ 

**Definition B.1.5 (Ordered tree domain, ordered tree)** A tree domain $D$ is ordered if and only if the underlying edge label set $E$ is well ordered (i.e. has a minimal element and is totally ordered) and, for all $n \in D$ and $i \in E$, $n \diamond i \in D$ ⇒ $\langle \forall j : j \in E \land j < i : n \diamond j \in D \rangle$ holds. An ordered tree is a tree $t$ whose tree domain $D_t$ is ordered.

B.2 Tree grammar related definitions

**Definition B.2.1 (rtg, Regular Tree Grammar)** A regular tree grammar (rtg) $G$ is a 5-tuple $(N, \Sigma, r, \text{Prod}, S)$ such that

- $N$ is an alphabet, the nonterminals
• \( \Sigma \) is an alphabet, the terminals
• \( N \cap \Sigma = \emptyset \)
• \((N \cup \Sigma, r)\) is a ranked alphabet with \( N = N_0 \) (i.e. nonterminals have rank 0)
• \( \text{Prod} \subseteq N \times \text{Tree}(N \cup \Sigma, r) \), the finite set of (production) rules or productions
• \( S \in N_0 \), the start symbol

Definition B.2.2 (Reachable symbol) Let \( G \) be an rtg. A symbol \( X \in N \cup \Sigma \) is reachable from a nonterminal \( A \) if and only if
\[
\exists \alpha : \alpha \in \text{Tree}(N \cup \Sigma, r) \land A \Rightarrow \alpha : X \in \alpha(D\alpha)
\]
(Note that \( \alpha(D\alpha) \) denotes the set of all node labels of tree \( \alpha \).)

Definition B.2.3 (Start-reachable symbol) Let \( G \) be an rtg. A symbol \( X \in N \cup \Sigma \) is start-reachable if and only if it is reachable from the start symbol \( S \).

Definition B.2.4 (Productive nonterminal, productive terminal) Let \( G \) be an rtg. A nonterminal \( B \in N \) is productive if and only if
\[
\exists t : t \in \text{Tree}(\Sigma, r) : B \Rightarrow t
\]
All terminals are productive (assuming \( \Sigma_0 \neq \emptyset \)).

B.3 Tree automata related definitions

Definition B.3.1 (Tree Automaton) A tree automaton \((\text{TA}) M\) is a 6-tuple \((Q, \Sigma, r, R, Q_{ra}, Q_{la})\) such that
• \( Q \) is a finite set, the state set
• \((\Sigma, r)\) is a ranked alphabet
• \( R = \langle \text{Set} a : a \in \Sigma : R_a \rangle \cup \{ R_\varepsilon \} \) is the set of transition relations, where \( R_a \subseteq Q \times Q^n \) for all \( a \in \Sigma \) and \( R_\varepsilon \subseteq Q \times Q \) (the epsilon transition relation)
• \( Q_{ra} \subseteq Q \), the root accepting states
• \( Q_{la} \subseteq Q \), the leaf accepting states,
  defined by \( Q_{la} = \langle \text{Set} a, q : a \in \Sigma_0 \land (q, ()) \in R_a : q \rangle \)

Definition B.3.2 (\( \varepsilon \)-Nondeterministic Root-to-Frontier Tree Automaton) An \( \varepsilon \)-nondeterministic root-to-frontier tree automaton \((\varepsilon\text{NRFTA}) M = (Q, \Sigma, r, R, Q_{ra}, Q_{la})\) is a TA where \( R_a \in Q \rightarrow \mathcal{P}(Q^n) \) for all \( a \in \Sigma \) such that \( \overrightarrow{q} \in R_a(p) \equiv pR_a\overrightarrow{q} \), and \( R_\varepsilon \in Q \rightarrow \mathcal{P}(Q) \) such that \( q \in R_\varepsilon(p) \equiv pR_\varepsilon q \).
Definition B.3.3 (ε-Nondeterministic Frontier-to-Root Tree Automaton) An ε-nondeterministic frontier-to-root tree automaton \( (\varepsilon \text{NFTA}) \) \( M = (Q, \Sigma, r, R, Q_{ra}, Q_{la}) \) is a TA where \( R_a \in Q^n \rightarrow \mathcal{P}(Q) \) for all \( a \in \Sigma \) such that \( p \in R_a(q) \equiv pR_aq \), and \( R_{\varepsilon} \in Q \rightarrow \mathcal{P}(Q) \) such that \( p \in R_{\varepsilon}(q) \equiv pR_{\varepsilon}q \).

Definition B.3.4 (Deterministic Root-to-Frontier Tree Automaton) A deterministic root-to-frontier tree automaton \( (\text{DRFTA}) \) \( M = (Q, \Sigma, r, R, Q_{ra}, Q_{la}) \) is an NRFTA where \( R_a \in Q \rightarrow Q^n \) for all \( a \in \Sigma \) —i.e. the \( R_a \) are functions yielding a single state tuple for every state—and \( Q_{ra} = \{ q_{ra} \} \) —i.e. there is a unique root accepting state.

Definition B.3.5 (Deterministic Frontier-to-Root Tree Automaton) A deterministic frontier-to-root tree automaton \( (\text{DFRTA}) \) \( M = (Q, \Sigma, r, R, Q_{ra}, Q_{la}) \) is an NFRTA where \( R_a \in Q^n \rightarrow Q \) for all \( a \in \Sigma \) —i.e. the \( R_a \) are functions yielding a single state for every state tuple.
C  ForestFIRE library

This appendix contains descriptions of all important classes (related to domain concepts and algorithms) and invariants in the ForestFIRE library. These descriptions are divided over the different concepts. A section is created for each concept (tree, tree grammar, tree pattern and tree automata). However, this chapter starts with the introduction to some basic collection types that were part of the data structures described in this appendix.

The remark has to be placed that the classes and interfaces presented here are not implemented exactly as described due to the restrictions of Java, the target programming language. Java for instance does not support properties, so the properties in this appendix are implemented as getter and setter methods. However, all the functionality described here is available in the library.

C.1 Basic collections

ForestFIRE uses a selection of collection types to represent (parts of) domain concepts (e.g. alphabets, lists of nodes). This section will discuss the most important types of collections that are used in ForestFIRE:

- List
- Dictionary
- Set

The next three sections will discuss the characteristics, interfaces and implementation issues for each of these collection types.

C.1.1 List

A list is one of the most well known collection types. A list is a sequence of items, where each item in the list is accessible with a integer index. Many operations are based on these integer indexes. One can add items to the back of such a list or insert an item on a certain index. This was the designed interface (in Object Pascal syntax) of a list for items of type $T$:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count: Integer</td>
<td>Returns the number of items that are contained in the list.</td>
</tr>
<tr>
<td>Capacity: Integer</td>
<td>Returns or sets the maximum number of items that can be stored in the list.</td>
</tr>
<tr>
<td>Item[index: Integer]: $T$</td>
<td>Returns or stores (only possible when index already exists) the item on the defined index.</td>
</tr>
</tbody>
</table>
C.1. Basic collections

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add(item: T): Integer</td>
<td>Adds the item to the end of the list and returns the index of that position.</td>
</tr>
<tr>
<td>Insert(item: T, index: Integer)</td>
<td>Insert an item to the list on the position defined by the index.</td>
</tr>
<tr>
<td>Remove(item: T): Integer</td>
<td>Removes the defined item from the list if it is present and returns the position at which the item was stored.</td>
</tr>
<tr>
<td>IndexOf(item: T): Integer</td>
<td>Returns the index of the defined item if it is present in the list.</td>
</tr>
</tbody>
</table>

Most of the properties and methods that are introduced above are straightforward. However, we also introduce a capacity property. This property provides the possibility to set the maximum size of the list. This can be very useful for instance when defining the list of child nodes. The number of child nodes is fixed by the rank of the symbol. The capacity of the list can then be used to ensure that no more children are added to node.

The remaining question was: how to implement such a list? There are many ways to do this: linked list, dynamic array. However, most of the modern programming languages already provide this type of collection. The list was therefore implemented by reusing the standard list of the programming language library (ArrayList in Java). A wrapper was created around that list to add the functionality that was not directly supported by the standard. This resulted in a new list that provides the interface described above.

C.1.2 Dictionary

A Dictionary is a collection type that stores key/value pairs, where each key can be used only once. These keys are used as a kind of indexes to store or retrieve the value items. The types used for these keys and values can be depend on the type of application. Sometimes one wants to use strings as keys (just like a normal word-dictionary), but it can also be any other type of data. The following interface (in Object Pascal syntax) was designed for a dictionary that stores item of type $V$ by using keys of type $K$:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count: Integer</td>
<td>Returns the number of key/value pairs that are contained in the list.</td>
</tr>
<tr>
<td>Item[key: $K$]: $V$</td>
<td>Returns or stores (only possible when the key already exists) the item for the defined key.</td>
</tr>
</tbody>
</table>
Add(key: K, value: V)  Adds the value with the specified key to the dictionary.
Remove(key: K)  Removes the item from the dictionary that is stored for the defined key.
ContainsKey(key: K)  Returns a truth value that indicates whether the defined key is used as key within the dictionary.
ContainsValue(value: V)  Returns a truth value that indicates whether the defined value can be found within the dictionary.

We see that the Insert and IndexOf methods are not present, due to usage of different indexing strategy. There is also no capacity property, because this functionality is not strictly necessary for our goals. The main role of this dictionary type as annotation container will be presented in Section C.2. This role does not need the possibility to define a maximum size, because it depends on the user of the tree how many items are stored in this annotation dictionary.

What remains is to discuss how this collection type can be implemented. The most common way to implement such a structure is by using a hash table. One should then define a hash function that computes a hash value for the types of objects that are used for indexing. The target programming language Java already provided an implementation of a hash table. This class reused and wrapped by a new class, just as with the list class, to provide the missing functionality.

C.1.3 Set

The last major collection type used in the ForestFIRE library is the set. A set is a pool of items, where items are stored without any index and where each item is contained at most once. Items can be added or removed by only passing the item itself to the method. The add operation itself must take care that no duplicate items are added to the collection. This is the designed Object Pascal interface for a set containing items of the type T:

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count: Integer</td>
<td>Returns the number of items that are contained in the list. (is already in use).</td>
</tr>
<tr>
<td>Content: Array of T</td>
<td>Returns an array that contains all the items in the set.</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add(item: T)</td>
<td>Adds the defined item to the set if it is not present.</td>
</tr>
<tr>
<td>Remove(item: T)</td>
<td>Removes the defined item from the set if it is present.</td>
</tr>
</tbody>
</table>

The remaining question is, how should one implement such a set? There are two well known
ways to implement a set [Ski98, Section 8.1.5]. One is to model the set is like a bit vector. To do this one has to know the universe of the items that can be contained in the set. A bit vector is an array of bits for each item of the universe. A one in the bit vector means that that item is contained in the set and zero means that it is not present.

Another method is to use the standard list as defined above. One stores the items of the set in this list. When an item is added, one checks if it is already present in the array. If this is not the case the item is added to the list. This can be done quite quickly if one keeps the list sorted. However, not all items can be sorted. It depends on the type of item that is stored in the list whether this optimization can be used. This internal list can then be hidden to the user of the set by creating a class that wraps it and that provides the standard interface that is described above.

This last option was chosen, because for many sets in the library it is not possible to determine what the precise universe is. This is for example the case in an terminal alphabet of a tree. Items can be added to this set of terminals all the time. This would mean that the bit vector should be extended every time this changed. The second method can handle such dynamical sets easier, because it does not use the universe to set up its internal data structure.

### C.2 Trees

This section contains brief descriptions of the data structures related to the concept of ranked, ordered, labeled trees. This are all the classes that can be found in Figure 3.1. Additionally a set of invariants are described that have to be maintained to ensure that no illegal structures are created.

#### C.2.1 Data structures

This section contains descriptions of all classes of the library that are used for representing trees and related concepts like nodes, symbols and alphabets.

##### C.2.1.1 Tree class

The *Tree* class represents a tree that consists from a set of nodes, where each of the nodes contains a symbol that is present in the alphabet of the tree. The alphabet can therefore be used to check if nodes that are added to the tree contain valid symbols that are contained in the alphabet (see the invariant section).

The root node and leaf node references can be used to visit the nodes of the tree in a top-down or bottom-up way.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabet: List of Symbol</td>
<td>Set of symbols that can be used inside the nodes of the tree.</td>
</tr>
<tr>
<td>Root: Node</td>
<td>The root node of the tree.</td>
</tr>
<tr>
<td>Leaves: List of Node</td>
<td>The leaf nodes of the tree.</td>
</tr>
</tbody>
</table>
C.2.1.2 Node class

The Node class represents a node in the tree structure. A node contains a symbol that tells whether it is a terminal, nonterminal or variable. A node has also references to its children and parent node to provide easy top-down and bottom-up traversal of the tree. Further there is a reference to the tree in which this node is contained, this can for instance be used to get access to the alphabet of a tree for checking or modification issues (see invariant section).

Additionally a node contains an annotation field that can be used by algorithms to store information in a node.

**Properties**

- **Parent**: Node
  - Parent node of this node. 'Null' when node is the root node.
- **ParentTree**: Tree
  - The tree that contains this node.
- **Annotation**: Dictionary
  - A dictionary like data structure to store annotation data in a structured way.
- **Symbol**: Symbol
  - Reference to the symbol that contains the name and type of the symbol in this node.
- **Children**: List of Node
  - Child nodes of this node.

**Methods**

None

C.2.1.3 Symbol class

The Symbol class is a class that provides a structure for representing symbols like:

- Terminal
- Nonterminal
- Variable

Each symbol contains a type that tell whether the symbol is a terminal, nonterminal or variable and a name field which contains the name of the symbol.
C.2. Trees

| Properties | | |
| --- | --- | |
| Name: String | The name of a symbol. | |
| Type: Enumeration | Contains the type of symbol: Terminal, Nonterminal or Variable. | |

| Methods | | |
| --- | --- | |
| None | | |

C.2.1.4 RankedSymbol class

The `RankedSymbol` class extends the `Symbol` class to represent a ranked symbol like a ranked terminal. This class is not used for nonterminals and variables if a tree uses ranked terminals. These nonterminals and variables are considered to have rank zero.

| Properties | | |
| --- | --- | |
| Name: String | The name of a symbol. | |
| Type: Enumeration | Contains the type of symbol: Terminal, Nonterminal or Variable. | |
| Rank: Integer | The rank of the symbol. | |

| Methods | | |
| --- | --- | |
| None | | |

C.2.1.5 DottedTree class

This class represents a dotted tree as its 2-tuple definition. It is used for referring to a node within a tree. It therefore contains a reference to the tree and a node inside this tree.

| Properties | | |
| --- | --- | |
| Tree: `Tree` | Reference to the original tree. | |
| Node: `Node` | Reference to a node inside the tree. | |

| Methods | | |
| --- | --- | |
| None | | |

C.2.2 Invariants

This section discusses several invariants for the data structures that are presented in Section C.2.1. These invariants must be kept valid to ensure that only wellformed trees are represented by these classes. We start by giving a couple of short definitions to shorten the notations of
the invariants. First we have to define what it means that a node is part of a tree, this done by Definition C.2.1 and C.2.2.

**Definition C.2.1** Whether a node is part of a tree (Node \( n \in \text{Tree} \ t \) ) is defined by the following definition:

\[
\text{Node} \ n \in \text{Tree} \ t = \text{NodeInTree}(t.\text{root}, n)
\]

The remaining thing to do is formally define the \( \text{NodeInTree}(m, n) \) function. This is done by Definition C.2.2.

**Definition C.2.2** \( \text{NodeInTree}(m, n) \) expresses recursively whether node \( n \) is part of the sub-tree that starts in node \( m \):

\[
\text{NodeInTree}(m, n) = (m = n) \lor \left( \bigvee \ r : r \leq m.\text{symbol.rank} : \text{NodeInTree}(m.\text{children}[r], n) \right)
\]

The upcoming sections will discuss a variety of invariants. Where each of the sections discusses invariants that cover a certain aspect of trees.

**C.2.2.1 Trees and symbols**

The first and most important invariant is that each node of a tree must use a symbol that is present in the alphabet of that tree. This is formalized by Invariant C.2.3.

**Invariant C.2.3** Each node \( n \) of a tree \( t \) must use a symbol that is present in the alphabet of that tree:

\[
\forall n, t : \text{Tree} \ t \land \text{Node} \ n \in t : n.\text{Symbol} \in t.\text{Alphabet}
\]

Another invariant in the area of symbols is the invariant that discusses the role of the symbol in individual nodes. Each node \( n \) may only have as many children as defined by the rank of its symbol. This is described in formal way by Invariant C.2.4.

**Invariant C.2.4** The number of children in a node \( n \) must be equal to the rank of the symbol that is stored in the node. The \#-symbol in the predicate below denotes the number of items in the list of children.

\[
\forall n : \text{Node} \ n : n.\text{Symbol.Rank} = \#(n.\text{Children})
\]

The last invariant of this section discusses the symbols that are used for representing nonterminals and variables. Nodes that use these symbols can only be used as leaf nodes. To achieve this we define an invariant that describes that nodes with a nonterminal or variable symbols have no child nodes:

**Invariant C.2.5** All symbols that represent a non terminal or variable are of rank zero.

\[
\forall n : \text{Node} \ n \land (n.\text{Symbol.Type} = \text{NonTerminal} \lor n.\text{Symbol.Type} = \text{Variable}) : \#(n.\text{Children}) = 0
\]
C.2.2.2 Child-parent relation

This section contains invariants that describe the parent and child relation between nodes of a tree. The first invariant expresses that the tree has a tree like shape, because children of a node n, must point to n as its parent node. This property ensures that the data structure has not got a different structure. For the formal definition see Invariant C.2.6.

Invariant C.2.6  
For each node n of a tree t the parent of each of the children of n is equal to n:

\[
\forall n, t : Tree \land Node \ n \in t : (\forall m : m \in n.children : m.parent = n)
\]

The next invariant ensures that each node n of tree t has a path back to the root node of the tree. This invariant is comparable to Definition C.2.1, with the difference that it ensures the existence of a path from the node to the top, instead of a path from the root to the node n:

Invariant C.2.7  
Each node n of a tree t must have a path to the root node of tree t:

\[
\forall n, t : Tree \land Node \ n \in t : (\exists i : i \in \mathbb{N} : n(.parent)^i = t.root)
\]

Finally we introduce an small invariant that makes sure that the ParentTree property of a node is set to the tree in which it is contained:

Invariant C.2.8  
The ParentTree property for every node n in tree t must contain t.

\[
\forall n, t : Tree \land Node \ n \in t : n.ParentTree = t
\]

C.2.2.3 Dotted tree invariant

There is also an important invariant in the area of dotted trees. A dotted tree contains a node and a tree. This node must be part of the specified tree:

Invariant C.2.9  
The specified node of the dotted tree must be part of the specified tree.

\[
\forall dt : DottedTree \ dt : dt.Node \in dt.Tree
\]

C.2.3 Related algorithms

This section describes the classes and methods that implement tree related algorithms. These algorithms are created to support some special tree grammar algorithms and mostly focus on providing tree statistics like the number of nodes. All these algorithms are implemented by the newly introduced TreeAnalyzer class.
C.2.3.1 Number of nodes

The number of nodes in a tree can be retrieved by the `getNumberOfNodes`-method of the `TreeAnalyzer` class. The method traverses the tree from the root to the leaves and counts the encountered nodes.

**TreeAnalyzer class**

```
getNumberOfNodes(t : Tree) : Integer
```

Returns the number of nodes inside the tree.

C.2.3.2 Number of non-root terminal nodes

The number of non-root terminal nodes can be retrieved by the `getNumberOfZPlusNodes`-method. This method is implemented in a similar way as the `getNumberOfNodes`-method. The only difference is that this method counts the nodes with a terminal symbol with the exception of the root node.

**TreeAnalyzer class**

```
getNumberOfZPlusNodes(t : Tree) : Integer
```

Returns the number of non-root nodes that contain a terminal.

C.3 Regular tree grammars

This section about regular tree grammars described all classes that implement data structures that are related to the concept regular tree grammar of ranked, ordered, labeled trees. Additionally there is a section that discusses invariants on these data structures and a third section that discusses classes that implement algorithms related to regular tree grammars.

C.3.1 Data structures

This section contains descriptions of all classes of the library that are used for representing regular tree grammars.

C.3.1.1 RegularTreeGrammar class

The standard `RegularTreeGrammar` class represents a tree grammar. A tree grammar is constructed from a set of production rules, an alphabet that contains terminals and nonterminals that can be used as start symbol or inside the production rules and a start nonterminal.
C.3. Regular tree grammars

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabet: Set of Symbol</td>
<td>The alphabet of terminals and nonterminals that are used in the grammar and its production rules.</td>
</tr>
<tr>
<td>ProductionRules: Set of ProductionRule</td>
<td>Production rules of the grammar.</td>
</tr>
<tr>
<td>StartSymbol: Symbol</td>
<td>The nonterminal symbol that forms the start symbol for the grammar.</td>
</tr>
</tbody>
</table>

Methods

None

C.3.1.2 GrammarProductionRule class

The GrammarProductionRule class represents a single grammar production rule. A production rule has just like its formal definition a left hand side symbol (of the type nonterminal) and a right hand side tree. Finally there is a cost field that can be used to store the cost applying that production rule (mostly used in the area of instruction selection).

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LHS: Symbol</td>
<td>The left hand side nonterminal of a production rule.</td>
</tr>
<tr>
<td>RHS: Tree</td>
<td>The right hand side tree of the production rule.</td>
</tr>
<tr>
<td>Cost: Integer</td>
<td>The cost when applying this rule.</td>
</tr>
</tbody>
</table>

Methods

None

C.3.1.3 DottedRule class

This class represents a dotted tree, that can be used to point to a specific node inside the RHS of a rule, as its 2-tuple definition. It contains a reference to the production rule and a node inside this RHS tree of the production rule.

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductionRule: GrammarProductionRule</td>
<td>Reference to the source production rule.</td>
</tr>
<tr>
<td>Node: Node</td>
<td>Reference to a node inside the RHS tree.</td>
</tr>
</tbody>
</table>

Methods

None
C.3.2 Invariants

As can be seen in Figure 3.2, the grammar classes are strongly related. We introduce a set of invariants to only that allow wellformed grammars are created by these classes.

The first invariant expresses the problem of the different alphabets in the right hand sides of the production rules. These alphabet may not contain symbols that can not be found in the general alphabet of the grammar. This invariant is defined precisely by Invariant C.3.1.

**Invariant C.3.1** The alphabet of a production rule r of a grammar g, must be a subset of the general alphabet of grammar g.

\[ \forall g, r : Grammar \ g \land Rule \ r \in g : r.RHS.Alphabet \subseteq g.Alphabet \]

Another concern is the left hand side of the production rules. One must ensure that these LHS symbols are nonterminals and that they present in the alphabet of the grammar. This is defined by Invariant C.3.2.

**Invariant C.3.2** The left hand side symbol of a rule r of grammar g, must be contained in the alphabet of grammar g and the symbol must be a nonterminal.

\[ \forall g, r : Grammar \ g \land Rule \ r \in g : r.LHS.Type = NonTerminal \land r.LHS \in g.Alphabet \]

The symbols in the right hand side tree also need to be part of the alphabet, but this invariant already holds due to the combination of Invariant C.3.1 and Invariant C.2.3 of the previous chapter.

Finally, there is an invariant in the area of dotted rules. A dotted rule contains a production rule and a node. This node must be part of the right hand side tree of that production rule:

**Invariant C.3.3** The specified node of the dotted rule must be part of the tree in the RHS of the specified production rule.

\[ \forall dr : DottedRule \ dr : dr.Node \in dr.ProductionRule.RHS \]

C.3.3 Related algorithms

This section contains all implemented algorithms that are related to regular tree grammars. A description is given of the classes that implement these algorithms. Detailed descriptions of these algorithms are not given in this section. These can be found in Chapter 4 for the grammar transformation algorithms and in [Cle07] for the algorithms that were not part of the experiments.

This section will divided the algorithms in several categories: standard analysis, usability, grammar transformation, dotted rule retrieval and subtree retrieval. Each of the upcoming subsections will provide details on the algorithms in that particular area.

C.3.3.1 Standard analysis

The standard analysis algorithms measure basic statistics for a tree grammar. This can be simple statistics like number of nodes and rules, but also more complicated statistics like presence or number of chain rules or non-root terminal nodes. All these algorithms can found in a newly created `RTGStandardAnalyzer` class.
Number of nodes The number of nodes in the production rules of a grammar can be retrieved by inspecting the `getNumberOfNodes` property of the `RTGStandardAnalyzer` class. This property is implemented by using `getNumberOfNodes` property for the `TreeAnalyzer` class. The values of all trees in the production rules can then be added to obtain the complete number of nodes.

```
RTGStandardAnalyzer class addition

GetNumberOfNodes(g : RTG) : Integer
Returns the number of nodes inside the tree in the right hand side of the production rules.
```

Number of rules The number of production rules in a grammar can be retrieved by inspecting a standard `count` property on the list that stores the production rules inside the grammar.

```
RTGStandardAnalyzer class addition

Count(g RTG) : Integer
Returns the number of production rules inside the grammar.
```

Number of chain rules The number of chain rules in a grammar can be retrieved by inspecting the `getNumberOfChainRules` property of the `RTGStandardAnalyzer` class. This property is implemented by defining a `IsChainRule` property for each production rule. Counting the rules that return a positive result will then result in the number of chain rules.

We also provide an additional `getChainRules` method that does not provide the number of chain rules but the rules themselves. This method is used together with the transformation operation that removes chain rules. The method can then be used to show which chain rules will/can be removed.

The existence of chain rules can be retrieved by inspecting the the `HasChainRules` property in the `RTGStandardAnalyzer` class. This property is implemented by testing if the `getNumberOfChainRules` property returns zero.

```
RTGStandardAnalyzer class addition

IsChainRule(r : ProductionRule) : Bool
Returns a boolean value that describes if the rule is chain rule or not.

GetNumberOfChainRules(g : RTG) : Integer
Returns the number of chain rules inside the grammar.

GetChainRules(g : RTG) : List of ProductionRule
Returns the list of chain rules within the grammar.

HasChainRules(g : RTG) : Bool
Checks whether the grammar contains chain rules or not.
```

Non-root nodes labeled by terminals The `getNumberOfZPlusNodes` method in the `RTGStandardAnalyzer` class provides the possibility to retrieve the number of non-root terminal
nodes. This is realized by using the `TreeAnalyzer` class that contains a method that returns the number of non-root terminal nodes for a tree. Calling this method for all RHS trees of the production rules and adding the return values will result in the total number of non-root terminal nodes of the grammar.

Just like in the previous section about chain rules we will also provide an additional method that does not only count the non-root terminal nodes, but also lists them. This is done by the `getZPlusNodes` method that provides a collection of dotted rules. These dotted rules point to the non-root terminals within the rules.

The existence of nodes in the RHS of the production rules that contain non-root terminals can be retrieved by inspecting the `HasZPlusNodes` property. This property can easily be implemented by testing if the `getNumberOfZPlusNodes` property returns zero.

### Rules that contain non-root nodes labeled by terminals

This operation that inspects the number of rules that contain non-root nodes labeled by terminals is comparable to the previous operation. The terminal counting in the tree can be implemented by the tree itself as described in the previous section. The `getNumberOfZPlusRules` method in the `RTGStandardAnalyzer` class can then count the rules with trees that contain one or more non root terminals.

### (Un)reachable symbols/production rules

One usability aspect is the reachability of symbols and rules. There are three different grammar operations defined to get the set of reachable symbols and production rules. The same operations are defined for retrieving unreachable symbols and rules. These three additional operations were easily be implemented by computing the difference between the complete set of terminals/nonterminals/rules and the reachable
items.

\textit{RTGUsabilityAnalyzer} class

\begin{tabular}{|l|}
\hline
GetReachableTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of reachable terminals. \\
GetReachableNonTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of reachable nonterminals. \\
GetReachableRules\((g : RTG)\) : Set of \textit{ProductionRule} & Returns Set of reachable production rules. \\
\hline
\end{tabular}

\textbf{(Un)productive terminals/nonterminals/production rules} Productiveness is another usability characteristic. The \textit{RTGUsabilityAnalyzer} class defines operations for this characteristic in a similar way then the reachability characteristic. The non-productive items can again be computed by taking the difference between the complete set of terminals/nonterminals/rules and their productive set.

\textit{RTGUsabilityAnalyzer} class addition

\begin{tabular}{|l|}
\hline
GetProductiveTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of productive terminals. \\
GetProductiveNonTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of productive nonterminals. \\
GetProductiveRules\((g : RTG)\) : Set of \textit{ProductionRule} & Returns Set of productive production rules. \\
\hline
\end{tabular}

\textbf{Useful/useless symbols/production rules} The useful/useless property is a combination of reachability and productiveness property. A terminal, nonterminal or production rule is useful if and only if it is reachable and productive. There are three \textit{RTGUsabilityAnalyzer} class operations defined for retrieving these useful items. The useless items can again be retrieved by computing the difference between all items and the useful items.

\textit{RTGUsabilityAnalyzer} class addition

\begin{tabular}{|l|}
\hline
getUsefulTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of useful terminals. \\
getUsefulNonTerminals\((g : RTG)\) : Set of \textit{Symbol} & Returns Set of useful nonterminals. \\
getUsefulRules\((g : RTG)\) : Set of \textit{ProductionRule} & Returns Set of useful production rules. \\
\hline
\end{tabular}
C.3.3.3 Transformations

This section describes the classes that implement algorithms related to tree grammar transformations. This section describes two additional classes. The `RTGUselessItemRemover` class that implements the algorithms for removing unreachable and unproductive items, and the `RTGStandardRemover` class that implements the algorithms for removing chain rules and non-root terminal nodes. These last two algorithms can be used to convert grammars such that they have the special characteristics that are described in Chapter 1. These last algorithms were also subject of experiments. More details about the implementation can therefore be found in Chapter 4.

Remove useless symbols/production rules  The three operations, for removing useless terminals, nonterminals and rules, will be handled by one class that contains a method for each of the removal operations. Each of these methods takes a regular tree grammar as input and creates a new grammar that does not contain the useless terminals, nonterminals or production rules. The three different methods can be used in sequence to remove all useless items.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoveUselessTerminals(g: RTG)</td>
<td>Outputs a copy of grammar $g$ that does not contain any useless terminals.</td>
</tr>
<tr>
<td>RemoveUselessNonTerminals (g: RTG) : RTG</td>
<td>Outputs a copy of grammar $g$ that does not contain any useless nonterminals.</td>
</tr>
<tr>
<td>RemoveUselessRules(g: RTG) : RTG</td>
<td>Outputs a copy of grammar $g$ that does not contain any useless production rules.</td>
</tr>
</tbody>
</table>

Remove chain rules  Chain rules are removed in a similar way as the useless terminals and other useless items. An additional class is introduced that contains an operation that takes a grammar and outputs a copy of the grammar that does not contain chain rules without changing the language produced by that grammar. This class is called `RTGStandardRemover` class, because it contains transformations that have corresponding analysis operations in the `RTGStandardAnalyzer` class.

We also introduced an additional method that provides the possibility to remove single chain rules from a grammar. This is done to investigate the effect of the removal of certain chain rules. This method is called `RemoveChainRule`.

More information about the transformation algorithm implemented by this class can be found in Section 4.2.2 and [Cle07, Transformation 3.4.34]

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoveChainRules(g: RTG) : RTG</td>
<td>Outputs a copy of grammar $g$ that doesn’t contain any chain rules.</td>
</tr>
<tr>
<td>RemoveChainRule(g: RTG, r: GrammarProductionRule)</td>
<td>Transforms the grammar $g$ in such a way that the chain rule $r$ is not present anymore.</td>
</tr>
</tbody>
</table>
C.3. Regular tree grammars

Remove rules containing non-root terminals  To remove rules that contain non-root terminals in their right hand side we add an additional method to the `RTGStandardRemover` class. This `RemoveZPlusRules`-method takes a regular tree grammar and creates a new grammar that does not contain trees with non-root terminals without changing the language produced by that grammar.

We also introduce an additional method that allows us to only remove one non-root terminal node in a production rule. This way we can analyze the effect of the removal of different nodes. This method is called `RemoveZPlusNode`. It takes a grammar and a dotted rule as input parameter and transform the grammar in such a way that doesn’t contain that node.

More information about the transformation algorithm implemented by this class can be found in Section 4.2.1 and [Cle07, Section 3.4.3]

<table>
<thead>
<tr>
<th><code>RTGStandardRemover</code> class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RemoveZPlusRules(g: RTG) : RTG</code></td>
</tr>
<tr>
<td><code>RemoveZPlusNode(g: RTG, n: DottedRule)</code></td>
</tr>
</tbody>
</table>

C.3.3.4 Dotted rules

This sections discusses operations to obtain and analyze dotted rules from the production rules of a grammar. These operations are implemented as separate classes to avoid garnishing the regular tree grammar class with a large collection of methods which are not vital for the grammar itself.

Get dotted rule set  This operation produces all possible dotted rules from the set of the production rules of a grammar. To realize this the `DottedRuleSetProvider` class is constructed that contains the `GetDottedRules`-method that consumes a grammar and produces these dotted rules. The produced dotted trees are wrapped in a new `DottedRuleCollection` class, because there additional operations defined on such a collection of dotted rules.

<table>
<thead>
<tr>
<th><code>DottedRuleSetProvider</code> class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetDottedRules(g: RTG) : DottedRuleCollection</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>DottedRuleCollection</code> class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules: Set of <code>DottedRule</code></td>
</tr>
</tbody>
</table>

**Flatten**  The flatten function converts a collection of dotted rules into a set of subtrees. This is realized by creating a new tree for each dotted rule that contains a cloned structure of the subtree referred by the dotted rule. Flattening a set of dotted rules, can result in duplicate tree structures. These duplicate trees are removed such that only unique subtrees are outputted.
**DottedRuleCollection** class addition

| Flatten() : Set of Tree | Returns the list of unique subtrees that are defined by the collection of dotted rules. |

**Number of dotted rules** The *DottedRuleCollection* class is inherits from the standard list that is defined in Section C.1.1. The number of dotted rules in the collection can therefore be retrieved by inspecting the inherited *count* property.

**DottedRuleCollection** class addition

| Count : Integer | Returns the number of dotted rules in the collection. |

**Number of flattened dotted rules** This operation returns the number subtrees that can be obtained from a set of dotted rules when they are flattened using the flatten function.

**DottedRuleCollection** class addition

| getNumberOfElements-WhenFlattened() : Integer | Returns the number subtrees if the dotted rules in the collection were flattened. |

### C.3.3.5 Subtrees

This sections discusses operations to directly obtain all subtrees as standard trees from a grammar. These operations are implemented in the same way as for the dotted rules by constructing a new class. This is again done to avoid garnishing the grammar tree with a large collection of methods.

**Get subtree set** This operation produces all unique items as subtrees from that can be extracted from the right hand side trees of the production rules. This operation is realized by the new *SubtreeSetProvider* class that contains a method that consumes a grammar and outputs a set of trees.

There is no special subtree collection class constructed, because there is no special functionality needed from the set of subtrees. The only needed functionality is a function for retrieval of the number of subtrees, but this is provided by the standard collection types.

**SubtreeSetProvider** class

| GetSubTrees(g: RegularTreeGrammar) : Set of Tree | Produces all unique subtrees from the right hand side trees of the production rules inside grammar *g*. |

**Number of Elements** The number of elements in a set of subtrees is retrieved by inspecting the standard count property of the used set type (see Section C.1), because there is no separate class defined for storing subtrees.
C.4 Tree patterns

This section contains a description of all data structures and invariants related to tree patterns.

C.4.1 Data structures

Most of the data structures for tree patterns are derived directly from trees, because patterns are represented as trees. The only additional data structure needed is a class that wraps a collection of tree patterns such that a group of tree patterns can easily be manipulated.

C.4.1.1 PatternSet class

The PatternSet class represents a collection of tree patterns. This collection also contains an alphabet that specifies which terminals, nonterminals and variables can be used in the containing patterns.

<table>
<thead>
<tr>
<th>Properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alphabet: Set of Symbol</td>
<td>Set of symbols that can be used in patterns.</td>
</tr>
<tr>
<td>Patterns: Set of Tree</td>
<td>The set of tree patterns.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

C.4.2 Invariants

There are not many invariants for this chapter, because patterns also reuse the tree invariants. However, it is important to create an invariant that describes that the symbols in the alphabet of a pattern also have to be present in the alphabet of the pattern collections where this pattern is part of. This is formalized by Invariant C.4.1.

Invariant C.4.1 The alphabet of a pattern p in a pattern set s must be a subset of the alphabet of the collection s.

\( \forall s, p : \text{PatternSet}\ s \land \text{Tree}\ p \in s : p.\text{Alphabet} \subseteq s.\text{Alphabet} \)

C.4.3 Related algorithms

The algorithms related to tree patterns have except one all as goal to retrieve (properties of) the subtrees that can be retrieved from a pattern collection. This section describe the classes that implement these algorithms.

C.4.3.1 Number of patterns

The number of patterns can be retrieved by inspecting the count property of the PatternSet class, because the class is implemented as a descendant of the list class that provides standard count property.
PatternSet class addition

Count : Integer  Returns the number of patterns that are stored in the set.

C.4.3.2 Dotted trees

This sections discusses operations to obtain dotted trees from a pattern collection. These operations are implemented as separate classes to avoid garnishing the PatternCollection class with a large collection of complicated methods.

Get dotted tree set  This operation produces all possible dotted trees from a collection of patterns. The previous chapter already introduced a class that distills all possible dotted rules from the production rules of grammar (See Section C.3.3.4). A similar class is introduced to retrieve all dotted trees from a collection of tree patterns.

DottedTreeSetProvider class

GetDottedTrees(p: PatternSet) : DottedTreeCollection  Produces all possible dotted trees from a set of tree patterns.

DottedTreeCollection class

Trees: Set of DottedTree  The set of dotted trees that are stored in the dotted tree collection.

Flatten  The flatten function converts a collection of dotted trees into a set of standard trees, by cloning the referenced tree structure. This function is similar to the Flatten function for dotted rules (See Section C.3.3.4). Therefore a similar solution is provided by extending the DottedTreeCollection class with a Flatten function.

DottedTreeCollection class addition

Flatten() : Set of Tree  Returns the list of unique subtrees that are defined by the collection of dotted trees.

Number of dotted trees  The DottedTreeCollection class is inherits from the standard list that is defined in Section C.1.1. The number of dotted rules in the collection can therefore be retrieved by inspecting the inherited count property.

DottedTreeCollection class addition

Count : Integer  Returns the number of dotted trees in the collection.

Number of flattened dotted trees  This operation returns the number items that can be obtained from a set of dotted trees when they are flattened using the flatten function. Such
operation is already described for the *DottedRuleCollection* class (See Section C.3.3.4). The same solution is chosen for the *DottedTreeCollection* class.

### DottedTreeCollection class addition

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getNumberOfElements-WhenFlattened()</code></td>
<td>Returns the number of subtrees if the dotted trees in the collection were flattened.</td>
</tr>
</tbody>
</table>

#### C.4.3.3 Subtrees

This section discusses operations to obtain all unique items as standard trees from a set of tree patterns. These operations are implemented in the same class used for subtree extraction from tree grammars.

**Get subtree set**

This operation retrieves all unique subtrees that can be extracted from a set of tree patterns. This is realized by cloning all unique subtrees that can be found in the patterns.

### SubtreeSetProvider class addition

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetSubTrees(p: PatternSet): Set of Tree</code></td>
<td>Produces all unique subtrees from the collection of patterns <em>p</em>.</td>
</tr>
</tbody>
</table>

**Number of elements**

The number of elements in a set/list of subtrees is retrieved by inspecting the standard count property of the used set/list (see Section C.1), because there is no separate class defined for storing a collection of subtrees.

### C.5 Tree automata

This section describes the classes that implement tree automata and classes that contain the algorithms that are related to tree automata. Furthermore this section describes a collection of invariants for the tree automata classes.

#### C.5.1 Data structures

The data structures section describes all classes that are implemented to represent $(\varepsilon)$NFRTAs, $(\varepsilon)$NRFTA and DFRTAs. This are the classes that can be found in Figure 3.5 of Chapter 3.

1. **AbstractTreeAutomaton class**

The *AbstractTreeAutomaton* class is an abstract class that presents the standard interface for a tree automaton based on the formal 5-tuple definition. Three items of the 5-tuple can be found as property: state set, alphabet and root accepting states. The reason why the other two items are not present in the base class can be found in Section 3.1.4.

This class also defines an abstract method for adding transitions. This possible because the general shape of a single transition is same for all automata if one abstracts from the
direction. However, it depends on the type of automaton what the most efficient way is to store them. This interface can therefore be implemented differently by all automata.

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateSet: Set of AutomatonState</td>
<td>The states that can be found inside the automaton.</td>
</tr>
<tr>
<td>Alphabet: Set of Symbol</td>
<td>The ranked alphabet used inside the automaton.</td>
</tr>
<tr>
<td>RootAccepting: Set of AutomatonState</td>
<td>The root accepting states of the automaton.</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddTransition (ss: AutomatonState, rs: AutomatonState^n, s: Symbol)</td>
<td>Adds an transition based on symbol s with rank n between state ss and the state vector rs of length n.</td>
</tr>
</tbody>
</table>

#### C.5.1.2 NRFTA class

When using an NRFTA (or DRFTA) one walks from the top to the bottom of the tree to find which states match which nodes. If one is performing such a traversal and one wants to use an NRFTA to find the child states for a node for which the state is known, then one has to feed this current state and the symbol of the current node to the automaton. An NRFTA will then return a set of vectors of n states (this set also contains all vectors that can be reached after executing all possible ε-transitions) that represent the possible states for these child nodes. This traversal is realized by the NextState-method.

Additionally there is a property that expresses whether the automata contains epsilon transitions.

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsEpsilonTransitions:</td>
<td>Indicated whether the automata contains epsilon transitions.</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextState (cs: AutomatonState, s: Symbol) : Set of AutomatonState^n</td>
<td>See NRFTA class description.</td>
</tr>
</tbody>
</table>

#### C.5.1.3 NFRTA class

When using an NFRTA (or DFRTA) one walks from the bottom to the top of the tree to find which states match which nodes. If one knows all the states for child nodes of a certain node α and wants to use an NFRTA to find the state of this node α, then one has to feed all these child node states and the symbol of α to the automaton. An NFRTA will then return
a set of states for the parent node (this set also contains all states that can be reached after executing all \( \varepsilon \)-transitions). This traversal is realized by the `NextState`-method.

When one uses an NFRTA one starts with the leaf accepting states. As described earlier there is no explicit field that contains these leaf accepting states. These states for a specific symbol of rank 0 can be retrieved by calling the `NextState`-method with an empty set of states and the symbol. The method will then return the leaf accepting states that accept this symbol.

### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsEpsilonTransitions</td>
<td>Indicated whether the automata contains epsilon transitions.</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NextState(cs: List of AutomatonState, s: Symbol) : Set of AutomatonState</code></td>
<td>See NFRTA introduction.</td>
</tr>
</tbody>
</table>

### C.5.1.4 AbstractDFRTA class

The goal of the `AbstractDFRTA` class is to support different DFRTA implementations. The interface of this abstract class is the same when compared to the `NFRTA` class, even the `NextState` Method is the same. However, its output is restricted to a singleton set containing only one state.

There are five types of DFRTA classes that inherit from this abstract class. These five classes all implement different optimization techniques that reduce the size of the automaton and the time needed to construct the automaton. These five types of DFRTAs are directly related to the five tree automaton construction algorithms that are discussed in Section 4.3 of Chapter 4:

- `DFRTAStandard` class
- `DFRTAFilterSubtree` class
- `DFRTAFilterIndex` class
- `DFRTAFilterSymbol` class
- `DFRTAFilterIndexSymbol` class

All these automata can be constructed using the `DTAGenerator` class. Calling the `generate` method of the generator class with a fresh automaton, tree grammar and corresponding item set will fill the automaton with the states and transitions such that it accepts the input grammar. The applied filtering technique depends on the type of automaton that is provided to the `generate` method.

The `AbstractDFRTA` class also contains methods that are not listed below. These methods are used to facilitate easy construction by the `DTAGenerator` class and optimize the mem-
ory usage of the automata. A description of these methods can be found in the JavaDoc
documentation of the library.

<table>
<thead>
<tr>
<th>Properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextState (cs: List of AutomatonState, s: Symbol) : (Singleton) Set of AutomatonState</td>
</tr>
</tbody>
</table>

**C.5.1.5 AbstractAutomatonState class**

This class represents an abstract state of an automaton. Each state has a name and the
descending class will define each in their own way to represent the subtrees that indicate
which subtrees are matched in that state. Each automaton can then use the type of state it
wants/needs.

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

**C.5.1.6 DottedTreeAutomatonState class**

This class represents a state based on dotted tree matches. This kind of states can be used
in automata constructed from pattern sets. The dotted trees in the match set of a state tell
which node(s) of which pattern match with that state. This match information can then be
used when for instance solving the tree pattern matching problem.

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name: String</td>
</tr>
<tr>
<td>Matches: Set of DottedTree</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
</tr>
</tbody>
</table>
C.5.1.7 DottedRuleAutomatonState class

This class represents a state based on dotted rule matches. This kind of states can be used in automata constructed from tree grammars. The dotted rules in the match set of a state tell which node(s) of which production rule match with that state. This match information can then be used when for instance solving the tree parsing problem.

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>String</th>
<th>The name of the state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matches</td>
<td>Set of DottedRule</td>
<td>The set of dotted rules tells which node of a certain production rule matches with this state.</td>
</tr>
</tbody>
</table>

Methods

None

C.5.1.8 SubtreeAutomatonState class

This class represents a state based on standard tree matches. This kind of states can be used in automata constructed from either pattern sets or tree grammars. These type of states can therefore be found many types of automata. The only disadvantage of these standard tree matches is that it is impossible to quickly retrieve their origin (which pattern or production rule), because they are cloned from their source.

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>String</th>
<th>The name of the state.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matches</td>
<td>Set of Tree</td>
<td>The set of subtrees tells which node of in a production rule or pattern matches with this state.</td>
</tr>
</tbody>
</table>

Methods

None

C.5.2 Invariants

This section presents all invariants that ensure that the classes above can only present well-formed automata. The first invariants that we have to define are the invariants that limit usage of states to the states defined in the state set of an automaton.

We start with defining an invariant that describes the fact that all root accepting states should be contained in the state set:

Invariant C.5.1 The set of root accepting states of an automaton must be subset of the set of all possible states.

\[ \forall a : AbstractTreeAutomaton \ a : a.RootAccepting \subseteq a.StateSet \]
We could define a similar invariant for the leaf accepting states, but the problem is that these states are not defined explicitly, because they can be obtained by using the \texttt{NextState}-method. It is however important to mention that all the states that are returned by this method must be part of the state set of the corresponding automaton:

\textbf{Invariant C.5.2} \textit{The states of an automaton that are produced by the \texttt{NextState}-method must be subset of the set of all possible states (State } q \in a.\texttt{NextState}, \textit{means that } q \textit{can be produced by the } \texttt{NextState}-\textit{method of automaton } a).

\[ (\forall a, q : \text{AbstractTreeAutomaton } a \land \text{State } q \in a.\text{NextState} : q \in a.\text{StateSet}) \]

The last invariant we introduce expresses the fact that each symbol of the alphabet of a automaton must have a role in the automaton. This means that every symbol must be part of the transition relations of the automaton:

\textbf{Invariant C.5.3} \textit{Each symbol of the alphabet of a automaton must be part of at least one transition (There is a separate definition for the RFTA's and FRTA's).}

\[ (\forall a, s : \text{NRFTA } a \land \text{Symbol } s \in a.\text{Alphabet} : \exists q : q \text{State } \in a.\text{StateSet} : a.\text{NextState}(q, s) \neq \emptyset) \]

\[ (\forall a, s : \text{NFRTA } a \land \text{Symbol } s \in a.\text{Alphabet} : \exists qs : \text{List of State } qs \subseteq a.\text{StateSet} : a.\text{NextState}(qs, s) \neq \emptyset) \]

\section*{C.5.3 Related algorithms}

This section contains all the interfaces and classes related to tree automaton algorithms. These algorithms are divided into three categories: state construction algorithms, automaton construction algorithms and tree acceptance/parse algorithms.

The first group discusses the interfaces of algorithms that provide item sets that are used for state construction in automaton construction algorithms. The second part discusses the interfaces of the construction algorithms themselves and finally the interfaces of the match and parse algorithms that use tree automata are discussed.

\subsection*{C.5.3.1 Automaton state construction}

Tree automaton states are, as described in Section, constructed based on a set of subtrees called \textit{item sets}. These item sets consists of a set of unique subtrees which can be represented by dotted rules, dotted trees or standard (cloned) trees. This section discusses the classes that can be used to construct such item sets that contain standard trees.

\textbf{AbstractItemSetProvider class} \textit{The AbstractItemSetProvider class is an abstract class from which all classes are derived that implement algorithms to obtain item sets, as cloned standard trees, from tree grammars. This class therefore specifies a constructor function for which an instance of the item set provider can be create based on a grammar. The item set can be retrieved by calling the GetSubTrees method. This method returns a tuple that contains}
the item set (a set of unique subtrees of the grammar) and the subtree from the item set that refers to the start symbol. This second item is interesting when automata are constructed, because they now which subtree must be present in the match set of the root accepting state.

There are three classes that inherit from this abstract class: `ProviderAllSub`, `ProviderProperN` and `ProviderProperS`. These classes implement the retrieval of the three different item sets that are discussed in tree automaton constructions in Chapter 4.

<table>
<thead>
<tr>
<th>AbstractItemSetProvider class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create(RegularTreeGrammar g)</td>
</tr>
<tr>
<td>GetSubTrees():(Tree, Set of Tree)</td>
</tr>
</tbody>
</table>

### C.5.3.2 Automaton construction

There are only two automaton construction algorithms implemented in ForestFIRE, one for nondeterministic automata and one for deterministic automata. However, these algorithms can be used to construct many different types of automata. This is caused by the separation of responsibilities. A part of the logic is transferred from the construction algorithm to the automata. The construction algorithms for nondeterministic automata can for instance create FR and RF automata based on the type of automaton. The automaton itself defines how the transitions are stored. This resulted in a low amount of duplicated code.

**NTAGenerator class** The `NTAGenerator` class is the class that implements the construction algorithm for non-deterministic automata. This algorithm (discussed in detail in Chapter 4) can be used to construct RF and FR automata with and without ε-transitions, based on one of the tree item sets. These different types of automata can be created by using the correct parameters in the `GenerateAutomaton`-method.

<table>
<thead>
<tr>
<th>NTAGenerator class</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateAutomaton(RTG g, AbstractTA ta, AbstractISP isp, boolean epsilon)</td>
</tr>
</tbody>
</table>

**DTAGenerator class** The `DTAGenerator` class is the class that implements the construction algorithm for DFRTAs. The algorithm can be used to construct a DFRTA from a tree grammar based on one of the three item sets. The experiments in Chapter 4 discuss filter techniques that can be used in the construction. These optimizations can be applied by using one of the the four special DFRTAs that inherit from the `AbstractDFRTA` class. These automata hide the filtering techniques from the construction process such that only this single construction algorithm is needed.
**DTAGenerator class**

GenerateAutomaton(RTG g, AbstractDFRTA ta, AbstractISP isp)  
Generates an DFRTA based on grammar g and the item set provided by isp and stores this automaton in ta.

### C.5.3.3 Tree acceptance & parsing

Two of the application areas of tree automata are the tree acceptance and tree parsing problem. This section defines classes that based on a tree automaton solve these two problems. There are three acceptor classes defined that solve the acceptance problem with a specific type of automata (NFRTA, NRFTA and DFRTA) and one parser class that solves the parsing with a provided DFRTA.

**NFRAcceptor class**  
The **NFRAcceptor** class implements an algorithm that determines whether an NFRTA accepts a certain input tree. The class realizes this by implementing the algorithm described in [Cle07, Algorithm 6.5.2].

**NFRAcceptor class**

Accept(NFRTA ta, Tree t) : Boolean  
Returns whether tree t is accepted by NFRTA ta.

**NRFAcceptor class**  
The **NRFAcceptor** class implements an algorithm that determines whether an NRFTA accepts a certain input tree. The class realizes this by implementing the algorithm described in [Cle07, Algorithm 6.4.2].

**NRFAcceptor class**

Accept(NRFTA ta, Tree t) : Boolean  
Returns whether tree t is accepted by NRFTA ta.

**DFRAcceptor class**  
The **DFRAcceptor** class implements an algorithm that determines whether a DFRTA accepts a certain input tree. The class realizes this by implementing the algorithm described in [Cle07, Algorithm 6.5.4].

**DFRAcceptor class**

Accept(DFRTA ta, Tree t) : Boolean  
Returns whether tree t is accepted by DFRTA ta.

**OptimalCostParser class**  
The **OptimalCostParser** class implements a parser that parses a tree for an input a grammar by using a DFRTA. This parsing algorithm delivers the minimal cost parse (based on the costs specified for each production rule) for that tree, if a parse exists.

This parsing algorithm constructs internally a DFRTA from the grammar and uses this grammar to compute which subtrees from the production rules are matched in which node.
Comparing these subtrees with the complete right hand side of the rules indicates which rules can applied. These matching production rules are then stored (with the LHS nonterminal as key) in the annotation dictionary of the node. Each node will after the parsing action contain a sequence (caused by chain rules) of production rules for each nonterminal, that describes which rules have to be applied such that node matches this nonterminal.

This is the interface of the *OptimalCostParser* class.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create(<em>Grammar</em> <em>g</em>)</td>
<td>Constructs a parser that can parse trees that are part of the language generated by <em>g</em>.</td>
</tr>
<tr>
<td>Parse(<em>Tree</em> <em>t</em>)</td>
<td>Parses the tree based on a automaton constructed from the tree grammar and stores in each node which rules have to applied to get the subtree rooted at this node.</td>
</tr>
</tbody>
</table>
D  FIREWood file format

This appendix discusses the FIREWood file format for defining trees, tree grammars etc. The goal of this file format is to provide an easy way to define input structures for the FIREWood application. The file format is based on the INI-format. We start by defining a def-list. This is a list of user defined structures. Each such a definition will be called a def-item.

These are five possible structures that can be defined:

• alphabet
• trees
• tree grammars
• tree patterns
• tree pattern collections

This resulted in the following EBNF-grammar for defining different structures:

\[
\text{def-list} = \text{def-item}\{\text{def-item}\}
\]

\[
\text{def-item} = \text{def-alphabet} | \text{def-tree} | \text{def-grammar} | \text{def-pattern} | \text{def-patterncollection}
\]

\[
\text{string} = \text{character}\{\text{character}\}
\]

\[
\text{character} = \text{a..z,A..Z}
\]

The definitions of the five different concepts have general shape, based on the general shape of an INI definition. Each definition of a concept consists of a name between brackets (the reference name of the concept) and a list of variable definitions, where one of the variable definitions specifies the type of the concept. This is for instance the general shape of a tree definition:

\[
[\text{mytree}]
\text{type=} \text{Tree}
\text{....=....}
\text{etc.}
\]
Each type of concept will contain a special list of variable definitions, next to the type variable definition, that need to be present. A tree for instance needs to define its corresponding alphabet and its structure. See for instance the example specification in Section D.5. The next sections will discuss a format for each of the five different types of structures.

D.1 Alphabets

This section describes an alphabet definition. There were two possibilities for this definition. One could define one default alphabet for the complete file that every tree, tree grammar etc. uses or one could offer the possibility to define multiple alphabets. The last option was chosen, to provide the possibility of defining structures with different alphabets in a single file.

The list of variable definitions for the alphabet contains two elements: the type definition and the alphabet itself. The type must be set to Alphabet and the alphabet itself is defined by a comma separated sequence of symbols and an optional rank (separated by the ':'-symbol). Where the symbols of the alphabet are strings of the characters a to z. Symbols with no rank are stored by ForestFIRE as unranked symbol. This is for instance used for nonterminals and variables.

This is the EBNF-grammar for this definition:

```plaintext
def-alphabet =
    "[/" string"]"
    type=Alphabet
    alphabet=def-alphabet-list

def-alphabet-list =
    "{ def-symbol {,def-symbol} }"

def-symbol =
    string [def-rank]

def-rank =
    ":" number

number =
    digit {digit}

digit =
    0..9
```

This is an example definition of an alphabet with name alphabetx:

```
[alphabetx]
type = Alphabet
symbols = {w:3, r:2, s:1, t:0, u:0}
```
D.2 Trees

Trees are defined by an alphabet and a definition of the tree structure itself. The alphabet is defined by a string that points to the defined alphabet with the same name. The tree structure is described in a prefix notation, where each node symbol is placed before a sequence of child nodes. This is the grammar of the tree definition, together with an example:

```plaintext
def-tree =
    "[" string"]"
    type=Tree
    alphabet=string
    structure=def-tree-structure

def-tree-structure =
    string | string "(" def-tree-children")"

def-tree-children =
    def-tree-structure 
    ,def-tree-structure
```

This is an example definition of a tree with name `treex`:

```
[ treex]
type = Tree
alphabet = alphabetx
structure = w(r(t,u), s(u), s(s(t)))
```

D.3 Tree grammars

Grammars are defined by a terminal alphabet, nonterminal alphabet, start nonterminal and a set of production rules. The terminal alphabet and nonterminal alphabet are defined in the same way as the alphabet of a tree by a string that points to a predefined alphabet. There is no separate part of the grammar definition that defines the start symbol. Instead the start symbol is defined as the first symbol in the nonterminal alphabet. This approach shortens the definition of a grammar.

The production rule list description is a bit more complicated. The rules are divided by semicolons and each rule is described by a string that represents the LHS followed by an ':' and a RHS tree structure as used in the tree definition. It is also possible to define a cost for each rule. This is done by putting an integer value after a "#" behind the production rule.

This is the grammar of the tree grammar definition, together with an example tree grammar definition:
def-grammar =
    "["string"]"
    type=Grammar
    terminal-alphabet=string
    nonterminal-alphabet=string
    rules=def-productions

def-productions =
    "{"def-prule ; {def-prule}""

def-prule =
    string ":" def-tree-structure [def-cost]

def-cost =
    "#" number

This is an example definition of a tree grammar with name grammarx, together with an nonterminal alphabet called alphabety:

[alphabety]
type = Alphabet
symbols = {$S$, $X$, $Y$}

[grammarx]
type = Grammar
terminal-alphabet = alphabetx
nonterminal-alphabet = alphabety
rules = {S: w(X,Y,Y) \# 1; X: r(t,X) \# 3; X: u \# 0; Y: s(Y) \# 1; Y: t \# 2; Y: X \# 1}

D.4 Tree patterns and pattern collections

Let us start with the definition of tree patterns. The tree pattern definition is almost the same as the normal tree definition. There is only one difference. The pattern definition also contains an alphabet of variables. This alphabet is described just as a standard alphabet of tree, by pointing to the desired alphabet definition.

This is the grammar of the tree pattern definition together with an example of such a definition:

def-pattern =
    "["string"]"
    type=Pattern
    terminal-alphabet=string
    variable-alphabet=string
    structure=def-tree-structure

This is an example definition of a tree grammar with name patternx, together with an variable alphabet called alphabetz::
Finally the pattern collection definition can be described. Such a collection is defined by the type 'PatternCollection' and a comma separated list of pattern variable names. These patterns should be defined in the same file before the definition of the pattern collection.

This is the grammar of this definition together with an example:

```
def-patterncollection =
    "[" string"]"
type=PatternCollection
patterns=def-pattern-list

def-pattern-list =
    "{" string {, string}"}"
```

This is an example definition of a tree grammar with name patterncolx with two imaginary names for the patterns:

```
[patterncolx]
type = PatternCollection
patterns = {patternx, patterny}
```

### D.5 Example

This section shows the definition file for the grammar described in Example 1.1.3:

```
[alphabetT]
type=Alphabet
symbols={a:2, b:1, c:0, d:0}

[alphabetNT]
type=Alphabet
symbols={$S$, $B$}

[mytree]
type=Tree
alphabet=alphabetT
structure = a(b(c), b(b(d)))
```

```
[mygrammar]
```
type=Grammar
terminal−alphabet=alphabetT
nonterminal−alphabet=alphabetNT
rules={S: a (B, d); S: a (b(c), B); S: c; B: b(B); B: S; B: d}
E Tree automaton construction – results

The appendix contains all measurement data collected during the automaton construction experiments. These results are divided over the different tree grammars. A table of measurements is presented for each of these grammars.
Chapter E. Tree automaton construction – results

### Table E.1: Example 6.0.3 – Basic Statistics

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>8</td>
<td>12</td>
<td>0.07</td>
<td>6344</td>
<td>6.2</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>8</td>
<td>14</td>
<td>0.07</td>
<td>4472</td>
<td>4.4</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>5</td>
<td>11</td>
<td>0.07</td>
<td>3512</td>
<td>3.4</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>5</td>
<td>11</td>
<td>0.07</td>
<td>3512</td>
<td>3.4</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>8</td>
<td>12</td>
<td>0.08</td>
<td>5400</td>
<td>5.3</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>8</td>
<td>14</td>
<td>0.09</td>
<td>6472</td>
<td>6.3</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>5</td>
<td>11</td>
<td>0.06</td>
<td>4584</td>
<td>4.5</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>5</td>
<td>11</td>
<td>0.06</td>
<td>4584</td>
<td>4.5</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>8</td>
<td>74</td>
<td>0.73</td>
<td>17160</td>
<td>16.8</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>6</td>
<td>44</td>
<td>0.35</td>
<td>8776</td>
<td>8.6</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>6</td>
<td>44</td>
<td>0.3</td>
<td>8776</td>
<td>8.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>8</td>
<td>32</td>
<td>0.4</td>
<td>18048</td>
<td>17.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>6</td>
<td>32</td>
<td>0.25</td>
<td>9824</td>
<td>9.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>6</td>
<td>32</td>
<td>0.24</td>
<td>9824</td>
<td>9.6</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>8</td>
<td>18</td>
<td>0.73</td>
<td>18816</td>
<td>18.4</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>6</td>
<td>18</td>
<td>0.22</td>
<td>10496</td>
<td>10.3</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>6</td>
<td>18</td>
<td>0.21</td>
<td>10496</td>
<td>10.3</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>8</td>
<td>21</td>
<td>0.39</td>
<td>19708</td>
<td>19.3</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>6</td>
<td>21</td>
<td>0.22</td>
<td>11208</td>
<td>10.9</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>6</td>
<td>21</td>
<td>0.22</td>
<td>11208</td>
<td>10.9</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>8</td>
<td>14</td>
<td>0.43</td>
<td>21048</td>
<td>20.6</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>6</td>
<td>14</td>
<td>0.2</td>
<td>12312</td>
<td>12.0</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>6</td>
<td>14</td>
<td>0.2</td>
<td>12312</td>
<td>12.0</td>
</tr>
</tbody>
</table>

### Table E.2: Example 6.0.3 – Table Statistics

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th># R Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>18</td>
</tr>
</tbody>
</table>
### Table E.3: Sample 4 – Basic Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFTA - All-Sub</td>
<td>13</td>
<td>20</td>
<td>0.29</td>
<td>10900</td>
<td>9.8</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>13</td>
<td>29</td>
<td>0.14</td>
<td>6976</td>
<td>6.8</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>6</td>
<td>22</td>
<td>0.08</td>
<td>4696</td>
<td>4.6</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>6</td>
<td>22</td>
<td>0.07</td>
<td>4696</td>
<td>4.6</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>13</td>
<td>20</td>
<td>0.11</td>
<td>9048</td>
<td>8.8</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>13</td>
<td>29</td>
<td>0.1</td>
<td>12856</td>
<td>12.6</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>6</td>
<td>22</td>
<td>0.07</td>
<td>8424</td>
<td>8.2</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>6</td>
<td>22</td>
<td>0.07</td>
<td>8424</td>
<td>8.2</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>9</td>
<td>183</td>
<td>0.84</td>
<td>24728</td>
<td>24.1</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>7</td>
<td>115</td>
<td>0.56</td>
<td>10144</td>
<td>9.9</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>7</td>
<td>115</td>
<td>0.49</td>
<td>10144</td>
<td>9.9</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>9</td>
<td>87</td>
<td>0.57</td>
<td>25384</td>
<td>24.8</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>7</td>
<td>87</td>
<td>0.44</td>
<td>11408</td>
<td>11.1</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>7</td>
<td>87</td>
<td>0.43</td>
<td>11408</td>
<td>11.1</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>9</td>
<td>43</td>
<td>0.5</td>
<td>25848</td>
<td>25.2</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>7</td>
<td>43</td>
<td>0.33</td>
<td>11776</td>
<td>11.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>7</td>
<td>43</td>
<td>0.33</td>
<td>11776</td>
<td>11.5</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>9</td>
<td>32</td>
<td>0.5</td>
<td>28296</td>
<td>27.6</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>7</td>
<td>32</td>
<td>0.32</td>
<td>13888</td>
<td>13.6</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>7</td>
<td>32</td>
<td>0.32</td>
<td>13888</td>
<td>13.6</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>9</td>
<td>17</td>
<td>0.48</td>
<td>30496</td>
<td>29.8</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>7</td>
<td>17</td>
<td>0.28</td>
<td>15736</td>
<td>15.4</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>7</td>
<td>17</td>
<td>0.27</td>
<td>15736</td>
<td>15.4</td>
</tr>
</tbody>
</table>

### Table E.4: Sample 4 – Table Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>6</td>
<td>13</td>
<td>6</td>
<td>54</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>6</td>
<td>13</td>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>6</td>
<td>13</td>
<td>6</td>
<td>42</td>
</tr>
</tbody>
</table>
### Table E.5: Sample 5 – Basic Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)-NFTA - All-Sub</td>
<td>12</td>
<td>18</td>
<td>0.27</td>
<td>9960</td>
<td>9.7</td>
</tr>
<tr>
<td>NFTA - All-Sub</td>
<td>12</td>
<td>22</td>
<td>0.18</td>
<td>6536</td>
<td>6.8</td>
</tr>
<tr>
<td>NFTA - ProperN</td>
<td>6</td>
<td>16</td>
<td>0.07</td>
<td>4984</td>
<td>4.9</td>
</tr>
<tr>
<td>NFTA - ProperS</td>
<td>6</td>
<td>16</td>
<td>0.06</td>
<td>4984</td>
<td>4.9</td>
</tr>
<tr>
<td>(\varepsilon)-NRFTA - All-Sub</td>
<td>12</td>
<td>18</td>
<td>0.12</td>
<td>8128</td>
<td>7.9</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>12</td>
<td>22</td>
<td>0.09</td>
<td>9552</td>
<td>9.3</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>6</td>
<td>16</td>
<td>0.06</td>
<td>5736</td>
<td>5.6</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>6</td>
<td>16</td>
<td>0.06</td>
<td>5736</td>
<td>5.6</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>9</td>
<td>173</td>
<td>0.84</td>
<td>26392</td>
<td>25.8</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>7</td>
<td>107</td>
<td>0.53</td>
<td>9976</td>
<td>9.7</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>7</td>
<td>107</td>
<td>0.48</td>
<td>9976</td>
<td>9.7</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>9</td>
<td>80</td>
<td>0.56</td>
<td>27048</td>
<td>26.4</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>7</td>
<td>80</td>
<td>0.44</td>
<td>11240</td>
<td>11.0</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>7</td>
<td>80</td>
<td>0.43</td>
<td>11240</td>
<td>11.0</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>9</td>
<td>32</td>
<td>0.5</td>
<td>27640</td>
<td>27.0</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>7</td>
<td>32</td>
<td>0.3</td>
<td>11736</td>
<td>11.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>7</td>
<td>32</td>
<td>0.3</td>
<td>11736</td>
<td>11.5</td>
</tr>
</tbody>
</table>

### Table E.6: Sample 5 – Table Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># (\phi)-Tables</th>
<th># (\phi)-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\varepsilon)-NFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(\varepsilon)-NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>5</td>
<td>13</td>
<td>5</td>
<td>35</td>
</tr>
</tbody>
</table>
Table E.7: ten Eikelder 68000 – Basic Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>36</td>
<td>66</td>
<td>0.37</td>
<td>32864</td>
<td>32.1</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>36</td>
<td>65</td>
<td>0.25</td>
<td>23536</td>
<td>23.0</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>9</td>
<td>38</td>
<td>0.12</td>
<td>14584</td>
<td>14.2</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>9</td>
<td>38</td>
<td>0.12</td>
<td>14584</td>
<td>14.2</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>36</td>
<td>66</td>
<td>0.3</td>
<td>25064</td>
<td>24.5</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>36</td>
<td>65</td>
<td>0.24</td>
<td>27856</td>
<td>27.2</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>9</td>
<td>38</td>
<td>0.12</td>
<td>10504</td>
<td>10.3</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>9</td>
<td>38</td>
<td>0.12</td>
<td>10504</td>
<td>10.3</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>39</td>
<td>9208</td>
<td>62.11</td>
<td>171848</td>
<td>167.8</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>10</td>
<td>624</td>
<td>3.15</td>
<td>24536</td>
<td>24.0</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>10</td>
<td>624</td>
<td>3.12</td>
<td>24536</td>
<td>24.0</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>39</td>
<td>508</td>
<td>3.49</td>
<td>115776</td>
<td>113.1</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>10</td>
<td>508</td>
<td>2.79</td>
<td>26184</td>
<td>25.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>10</td>
<td>508</td>
<td>2.81</td>
<td>26184</td>
<td>25.6</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>39</td>
<td>354</td>
<td>2.74</td>
<td>117120</td>
<td>114.4</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>10</td>
<td>354</td>
<td>2.12</td>
<td>26136</td>
<td>25.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>10</td>
<td>354</td>
<td>2.11</td>
<td>26136</td>
<td>25.5</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>39</td>
<td>153</td>
<td>2.02</td>
<td>131718</td>
<td>128.6</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>10</td>
<td>153</td>
<td>1.47</td>
<td>33080</td>
<td>32.3</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>10</td>
<td>153</td>
<td>1.5</td>
<td>33080</td>
<td>32.3</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>39</td>
<td>89</td>
<td>1.86</td>
<td>149960</td>
<td>146.4</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>10</td>
<td>89</td>
<td>1.18</td>
<td>40648</td>
<td>39.7</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>10</td>
<td>89</td>
<td>1.16</td>
<td>40648</td>
<td>39.7</td>
</tr>
</tbody>
</table>

Table E.8: ten Eikelder 68000 – Table Statistics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>15</td>
<td>2</td>
<td>78</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>15</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>15</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>8</td>
<td>33</td>
<td>8</td>
<td>312</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>8</td>
<td>33</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>8</td>
<td>33</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>14</td>
<td>49</td>
<td>14</td>
<td>546</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>14</td>
<td>49</td>
<td>14</td>
<td>140</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>14</td>
<td>49</td>
<td>14</td>
<td>140</td>
</tr>
</tbody>
</table>
Table E.9: Mono X86 – Basic Statistics

<table>
<thead>
<tr>
<th>Method</th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>532</td>
<td>1029</td>
<td>5.97</td>
<td>328848</td>
<td>0.5</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>532</td>
<td>1205</td>
<td>4.98</td>
<td>381248</td>
<td>0.4</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>63</td>
<td>736</td>
<td>1.32</td>
<td>225056</td>
<td>0.2</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>63</td>
<td>736</td>
<td>1.32</td>
<td>225056</td>
<td>0.2</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>532</td>
<td>1029</td>
<td>7.54</td>
<td>391040</td>
<td>0.4</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>532</td>
<td>1205</td>
<td>4.97</td>
<td>508560</td>
<td>0.5</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>63</td>
<td>736</td>
<td>1.33</td>
<td>207880</td>
<td>0.2</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>63</td>
<td>736</td>
<td>1.34</td>
<td>207880</td>
<td>0.2</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>557</td>
<td>2348299</td>
<td>7661.07</td>
<td>3833328</td>
<td>3.7</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>65</td>
<td>348299</td>
<td>1714.59</td>
<td>2535240</td>
<td>2.4</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>65</td>
<td>348299</td>
<td>1701.26</td>
<td>2535240</td>
<td>2.4</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>557</td>
<td>337821</td>
<td>4166.51</td>
<td>2510232</td>
<td>2.4</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>65</td>
<td>337821</td>
<td>4159.49</td>
<td>2510232</td>
<td>2.4</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>65</td>
<td>337821</td>
<td>4159.49</td>
<td>2510232</td>
<td>2.4</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>557</td>
<td>160651</td>
<td>3157.93</td>
<td>2688280</td>
<td>2.6</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>65</td>
<td>160651</td>
<td>1343.52</td>
<td>1341568</td>
<td>1.3</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>65</td>
<td>160651</td>
<td>1343.87</td>
<td>1341568</td>
<td>1.3</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>557</td>
<td>2097</td>
<td>67.58</td>
<td>931144</td>
<td>0.9</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>65</td>
<td>2097</td>
<td>67.58</td>
<td>931144</td>
<td>0.9</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>65</td>
<td>2097</td>
<td>67.58</td>
<td>931144</td>
<td>0.9</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>557</td>
<td>1207</td>
<td>381.24</td>
<td>11805000</td>
<td>11.3</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>65</td>
<td>1207</td>
<td>68.53</td>
<td>1637712</td>
<td>1.6</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>65</td>
<td>1207</td>
<td>68.53</td>
<td>1637712</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Table E.10: Mono X86 – Table Statistics

<table>
<thead>
<tr>
<th>Method</th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>64</td>
<td>1</td>
<td>557</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>64</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>64</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>88</td>
<td>2</td>
<td>1114</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>88</td>
<td>2</td>
<td>130</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>88</td>
<td>2</td>
<td>130</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>238</td>
<td>722</td>
<td>238</td>
<td>132566</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>238</td>
<td>722</td>
<td>238</td>
<td>15470</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>238</td>
<td>722</td>
<td>238</td>
<td>15470</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>318</td>
<td>872</td>
<td>318</td>
<td>177126</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>318</td>
<td>872</td>
<td>318</td>
<td>20670</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>318</td>
<td>872</td>
<td>318</td>
<td>20670</td>
</tr>
</tbody>
</table>
Table E.11: Mono IA64 – Basic Statistics

<table>
<thead>
<tr>
<th></th>
<th># States</th>
<th># Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>441</td>
<td>865</td>
<td>4.13</td>
<td>447136</td>
<td>0.4</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>441</td>
<td>1078</td>
<td>3.45</td>
<td>319984</td>
<td>0.3</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>40</td>
<td>677</td>
<td>0.91</td>
<td>186880</td>
<td>0.2</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>40</td>
<td>677</td>
<td>0.91</td>
<td>186880</td>
<td>0.2</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>441</td>
<td>865</td>
<td>5.69</td>
<td>330025</td>
<td>0.3</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>441</td>
<td>1078</td>
<td>3.37</td>
<td>456080</td>
<td>0.4</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>44</td>
<td>677</td>
<td>0.91</td>
<td>199768</td>
<td>0.2</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>44</td>
<td>677</td>
<td>0.91</td>
<td>199768</td>
<td>0.2</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>438</td>
<td>14075158</td>
<td>203738</td>
<td>86757368</td>
<td>82.7</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>42</td>
<td>135562</td>
<td>622.21</td>
<td>1060696</td>
<td>1.0</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>42</td>
<td>135562</td>
<td>621.49</td>
<td>1060696</td>
<td>1.0</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>438</td>
<td>129342</td>
<td>4127.71</td>
<td>2073816</td>
<td>2.0</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>42</td>
<td>129342</td>
<td>1351.51</td>
<td>1051240</td>
<td>1.0</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>42</td>
<td>129342</td>
<td>1351.51</td>
<td>1051240</td>
<td>1.0</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>438</td>
<td>54706</td>
<td>1659.59</td>
<td>1612144</td>
<td>1.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>42</td>
<td>54706</td>
<td>393.34</td>
<td>570560</td>
<td>0.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>42</td>
<td>54706</td>
<td>393.02</td>
<td>570560</td>
<td>0.5</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>438</td>
<td>1351</td>
<td>193.08</td>
<td>4721856</td>
<td>4.5</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>42</td>
<td>1351</td>
<td>26.92</td>
<td>642160</td>
<td>0.6</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>42</td>
<td>1351</td>
<td>26.73</td>
<td>642160</td>
<td>0.6</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>438</td>
<td>915</td>
<td>266.39</td>
<td>9121960</td>
<td>8.7</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>42</td>
<td>915</td>
<td>29.59</td>
<td>1104440</td>
<td>1.1</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>42</td>
<td>915</td>
<td>29.41</td>
<td>1104440</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table E.12: Mono IA64 – Table Statistics

<table>
<thead>
<tr>
<th></th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>41</td>
<td>1</td>
<td>438</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>41</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>41</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>56</td>
<td>2</td>
<td>876</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>56</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>56</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>234</td>
<td>643</td>
<td>234</td>
<td>102492</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>234</td>
<td>643</td>
<td>234</td>
<td>9828</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>234</td>
<td>643</td>
<td>234</td>
<td>9828</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>307</td>
<td>776</td>
<td>307</td>
<td>134466</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>307</td>
<td>776</td>
<td>307</td>
<td>12894</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>307</td>
<td>776</td>
<td>307</td>
<td>12894</td>
</tr>
</tbody>
</table>
# Chapter E. Tree automaton construction – results

## Table E.13: Mono Sparc – Basic Statistics

<table>
<thead>
<tr>
<th>Method</th>
<th>States</th>
<th>Transitions</th>
<th>Time (ms)</th>
<th>Memory (bytes)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFTA - All-Sub</td>
<td>491</td>
<td>967</td>
<td>5.05</td>
<td>494104</td>
<td>0.5</td>
</tr>
<tr>
<td>NFTA - All-Sub</td>
<td>491</td>
<td>1145</td>
<td>4.26</td>
<td>352840</td>
<td>0.3</td>
</tr>
<tr>
<td>NFTA - ProperN</td>
<td>51</td>
<td>705</td>
<td>1.08</td>
<td>206040</td>
<td>0.2</td>
</tr>
<tr>
<td>NFTA - ProperS</td>
<td>51</td>
<td>705</td>
<td>1.09</td>
<td>206040</td>
<td>0.2</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>491</td>
<td>967</td>
<td>6.77</td>
<td>364456</td>
<td>0.3</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>491</td>
<td>1145</td>
<td>4.25</td>
<td>482160</td>
<td>0.5</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>51</td>
<td>705</td>
<td>1.07</td>
<td>200064</td>
<td>0.2</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>51</td>
<td>705</td>
<td>1.07</td>
<td>200064</td>
<td>0.2</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>487</td>
<td>1834296</td>
<td>188725</td>
<td>112685992</td>
<td>107.5</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>53</td>
<td>225066</td>
<td>965.44</td>
<td>1653248</td>
<td>1.6</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>53</td>
<td>225066</td>
<td>957.41</td>
<td>1653248</td>
<td>1.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>487</td>
<td>208720</td>
<td>4371.86</td>
<td>2756664</td>
<td>2.6</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>53</td>
<td>208720</td>
<td>2815.37</td>
<td>1578624</td>
<td>1.5</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>53</td>
<td>208720</td>
<td>2810.04</td>
<td>1578624</td>
<td>1.5</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>487</td>
<td>97543</td>
<td>1726.73</td>
<td>2049936</td>
<td>2.0</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>53</td>
<td>97543</td>
<td>832.37</td>
<td>851064</td>
<td>0.8</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>53</td>
<td>97543</td>
<td>829.99</td>
<td>851064</td>
<td>0.8</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>487</td>
<td>1502</td>
<td>230.77</td>
<td>5305448</td>
<td>5.1</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>53</td>
<td>1502</td>
<td>34.39</td>
<td>783600</td>
<td>0.7</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>53</td>
<td>1502</td>
<td>34.06</td>
<td>783600</td>
<td>0.7</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>487</td>
<td>1001</td>
<td>314.37</td>
<td>10456104</td>
<td>10.0</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>53</td>
<td>1001</td>
<td>37.35</td>
<td>1375792</td>
<td>1.3</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>53</td>
<td>1001</td>
<td>37.15</td>
<td>1375792</td>
<td>1.3</td>
</tr>
</tbody>
</table>

## Table E.14: Mono Sparc – Table Statistics

<table>
<thead>
<tr>
<th>Method</th>
<th># R-Tables</th>
<th># R-Entries</th>
<th># φ-Tables</th>
<th># φ-Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>εNFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>εNRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NRFTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - All-Sub</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - ProperS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - All-Sub</td>
<td>1</td>
<td>51</td>
<td>1</td>
<td>487</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperN</td>
<td>1</td>
<td>51</td>
<td>1</td>
<td>53</td>
</tr>
<tr>
<td>DFRTA - SubTree Filtering - ProperS</td>
<td>1</td>
<td>51</td>
<td>1</td>
<td>53</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - All-Sub</td>
<td>2</td>
<td>71</td>
<td>2</td>
<td>974</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperN</td>
<td>2</td>
<td>71</td>
<td>2</td>
<td>106</td>
</tr>
<tr>
<td>DFRTA - Index Filtering - ProperS</td>
<td>2</td>
<td>71</td>
<td>2</td>
<td>106</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - All-Sub</td>
<td>242</td>
<td>700</td>
<td>242</td>
<td>117854</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperN</td>
<td>242</td>
<td>700</td>
<td>242</td>
<td>12826</td>
</tr>
<tr>
<td>DFRTA - Symbol Filtering - ProperS</td>
<td>242</td>
<td>700</td>
<td>242</td>
<td>12826</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - All-Sub</td>
<td>319</td>
<td>841</td>
<td>319</td>
<td>155353</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperN</td>
<td>319</td>
<td>841</td>
<td>319</td>
<td>16907</td>
</tr>
<tr>
<td>DFRTA - Symbol &amp; Index Filtering - ProperS</td>
<td>319</td>
<td>841</td>
<td>319</td>
<td>16907</td>
</tr>
</tbody>
</table>
Bibliography


