Compilers for Tree Grammars and Tree Transducers

Lars Vikberg

February 11, 2004

Abstract

This thesis describes a run-time compiler addition for the TREEBAG system. TREEBAG is used to generate and display objects using various tree generators. The addition to the system compiles two types of tree generators, regular tree grammars and top-down tree transducers, with the purpose of increasing performance. Since TREEBAG is written in Java, the target language for the run-time compilation is also Java.

Contents

Chapter 1

Introduction

The objective and organization of this thesis is presented in this chapter, as well as a short introduction to the role of trees in formal language theory. There is also a short presentation of the ideas behind Treebag.

1.1 Objective

Treebag is a program that uses tree grammars and tree transformations to generate trees. The most basic tree generators available in Treebag are regular tree grammars and top-down tree transducers. A tree generator is defined in an ordinary text file with a syntax depending on the type of tree generator. To derive trees, Treebag creates an internal representation of the tree generator from such a text file. Efficiency can be an issue when large trees are being generated. The objective of the work presented in this thesis has been to create extensions to Treebag that compile specific regular tree grammars and top-down tree transducers into code that implements that specific tree generator. Suitable data structures must be developed and some algorithms used in the generators must be improved to achieve better performance. Also, the functionality and behavior of a compiled generator

must be the same as that of an uncompiled one.

1.2 Trees in Formal Language Theory

A language can be seen as something that has form and meaning. The form describes what objects are part of the language (in natural languages the objects are sentences) and the meaning describes how these objects should be interpreted. Formal language theory studies the form of languages and certain techniques that exploit form in order to specify meaning (syntax-directed translation for example). These studies are an important field within computer science. Typical questions within formal language theory are how to describe the form of languages (commonly used methods are grammars and automata) and how to test if an object is part of a language (often called parsing). In the study of programming languages and the construction of compilers (see [ASU86]), which are areas within computer science where formal language theory is important, the objects that are part of a language typically are strings. This thesis focuses on languages that contain trees and how to generate trees that are elements of a given tree language.

Theories on tree languages and tree automata started to form in the 1960's (see [Rou69] and [Rou70]). An early application of tree automata was within circuit verification, and they are still used in that context. Recent works use tree automata in databases, term rewriting, automated theorem proving and program verification. Using tree automata has proven to be a powerful approach to simplify and extend previously known results, and also to find new ones. A current topic within computer science where trees and tree automata are frequently discussed is XML (the Extensible Markup Language).

A simple description of regular tree grammars is that trees are generated by applying one or more rules to a start state. The regular tree language generated by a specific regular tree grammar will contain zero or more unique trees depending on the rules in the grammar. A top-down tree transducer

also uses a set of rules, but in addition to starting with a specific state, an input tree also determines what trees will be generated. Thus, a regular tree grammar directly generates trees, while a top-down tree transducer produces output trees from input trees. Overviews of the theory and applications of tree languages and tree transductions can be found in [GS84], [GS97] and [FV98][1].

1.3 Run-Time Compilation

Generating code during the execution of a program allows compiled code to take advantage of variables and invariants that cannot be exploited when the program itself is compiled. Special-purpose code can be produced at run-time instead of general-purpose code compiled before the program is running. For example, code that implements a certain type of tree generator in general is bound to be slower than code that only implements the rules needed for one specific tree generator. Many optimizations will be simpler to use in run-time generated code, and some optimizations will not be possible for code generated prior to execution. For discussions on run-time compilation see [APC+96] and [LL96]. A particular area where code generation during execution currently receives a lot of attention is within Java virtual machines (see [NM00]).

1.4 Treebag

The basic idea behind Treebag is to combine tree generators with various interpretation components to generate pictures or other objects. Using trees instead of strings has the advantage that trees can be treated as expressions by interpreting symbols as operations (this is similar to the idea behind parse

[1] See http://www.grappa.univ-lille3.fr/tata/ as well.

trees). Depending on the interpretation chosen, different sets of objects can be created (pictures, strings, graphs and so on) while the actual generation parts remain the same.

1.5 Organization of this Thesis

The second chapter introduces concepts that are necessary to understand this thesis. Mathematical preliminaries, definitions of regular tree grammars and top-down tree transducers as well as an overview of the TREEBAG system are part of this chapter. Chapter three describes how the objective of this thesis has been achieved. The general approach, data structures and algorithms are described. The fourth chapter discusses the results of the work reported in this thesis. In particular, it will be discussed how the compiler parts change performance and how they could be improved. A short summary at the end of the chapter concludes the thesis.

Chapter 2

Background

In this chapter, concepts important to the understanding of this thesis are presented both mathematically and in a less formal manner. It is not necessary to read the mathematical definitions unless the reader finds it clearer or simply interesting. Some concepts that are necessary for definitions, but not for the general understanding, are only presented mathematically. The section about TREEBAG is intended to show the reader what TREEBAG is and how the work presented in this thesis relates to TREEBAG. The presentation of concepts given here is oriented at [Dre03].

2.1 Basic Concepts

Trees are one of the most basic concepts in this thesis. Anyone who has studied computer science is likely to have encountered trees within several areas. The trees used here are not different from the ones that are typically used. Nodes are labelled with symbols that have a certain number of child nodes organized in an ordered list. A number of definitions are needed to make this precise.

A signature is a set of ranked symbols. A ranked symbol has a name and

a number, equal to or larger than zero, associated to it. Trees can be built from such a signature using the symbols as nodes and the rank of a symbol to specify the number of children to the node. A symbol with rank zero would thus be a leaf. Symbols may be written as $f^{(n)}$, meaning a symbol named f with rank n. Also, a symbol $f^{(n)}$ may be denoted by its name f alone if n is understood from the context or is unimportant. A common textual representation of trees is to write the subtrees of a node within brackets. For example, $f[g[\ldots], g[\ldots]]$ denotes a tree with the root symbol f that has two subtrees that both have g as root symbol. Further examples of this are given below. The yield of a tree is the string obtained by reading the names of the leaves from left to right. Symbols with rank greater than zero are not part of the yield string since they never can be leaves.

Example (*Signatures and trees*) Consider the signature $\{f^{(2)}, c^{(0)}, d^{(0)}\}$ consisting of one symbol of rank 2 named f and two symbols of rank 0 named c and d. Figure 2.1 shows some of the trees that can be built from this signature. In such a drawing, the children of a node are ordered from left to right, as one would expect. For example, the root node of the rightmost tree in the figure has as its first child the one labelled f and as its second the one labelled c. The trees shown are just a subset of all the trees over the signature, over which an infinite number of trees can be constructed. Textual representations of the trees are $f[c, c]$, c and $f[f[c, d], c]$. The yields of the trees are `cc`, `c` and `cdc`.

Definition (*Ranked symbol and signature*) A *ranked symbol*, hereafter called symbol, is a pair (f, n) consisting of its name f and its rank $n \in \mathbb{N}$ and is denoted $f^{(n)}$. A *signature* Σ is a set of symbols. If Σ is a finite set of symbols, it is called a finite signature.

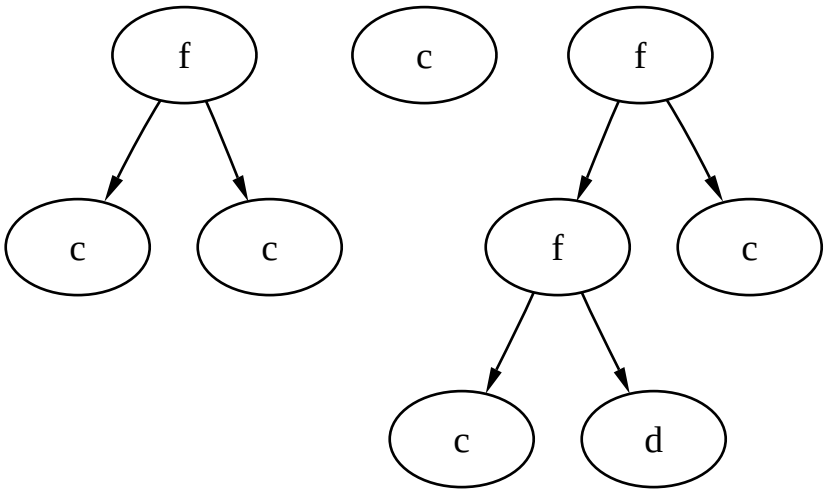

Figure 2.1: Some trees over the signature $\{f^{(2)}, c^{(0)}, d^{(0)}\}$.

Definition (*Tree and yield*) A *tree* is a pair consisting of a root symbol $f^{(n)}$ and a tuple of n direct subtrees. Trees are denoted $f[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are its direct subtrees; $f[]$ is abbreviated as f. The *yield* of a tree t, $yield(t)$, is $yield(f) = f$ for a root symbol of rank zero and $yield(f[t_1, \ldots, t_n]) = yield(t_1) \ldots yield(t_n)$ for a root symbol of rank greater than zero.

A tree language is a set of trees over a given signature. The set of trees in Figure 2.1 can be seen as a tree language over the signature $\{f^{(2)}, c^{(0)}, d^{(0)}\}$. However, while this tree language is finite, we will in the following mainly be interested in infinite ones. Tree grammars and tree transducers discussed in the following sections define tree languages.

Definition (*Tree language and trees over a signature*) A *tree language* is a set of trees. The set of all *trees over a signature* Σ, all trees that can be constructed from the symbols in Σ, is denoted T_Σ. Inductively, a tree $f[t_1, \ldots, t_n]$ is in T_Σ if and only if $f^{(n)} \in \Sigma$ and $t_1, \ldots, t_n \in T_\Sigma$.

An algebra assigns a meaning to the symbols in a signature. This can be used to interpret trees over that signature. With the previous example signature,

$\{f^{(2)}, c^{(0)}, d^{(0)}\}$, a simple algebra could state that c creates the string `C`, d creates the string `D` and f creates the string that is the concatenation of the strings that its children create. If this algebra is applied to the trees in Figure 2.1 the resulting strings would be `CC`, `C` and `CDC`. Other algebras might assign operations to the same signature that create picture drawing commands or perform computations. The example algebra is similar to taking the yield of a tree. In fact, yield can be seen as an algebra that treats all symbols of rank greater than zero as string concatenation.

Definition (*Algebra*) A Σ-*algebra* is a pair $A = (\mathcal{D}, (f_A)_{f \in \Sigma})$ where $\mathcal{D}$ is a set, the domain of A, and for every $f^{(n)} \in \Sigma$, $f_A : \mathcal{D}^n \to \mathcal{D}$ is an n-ary operation on $\mathcal{D}$, the interpretation of f in A. The value $val_A(t)$ of a tree $t \in T_\Sigma$ with respect to the Σ-algebra A is $val_A(f[t_1, \ldots, t_n]) = f_A(val_A(t_1), \ldots, val_A(t_n))$. In particular, the value of a symbol of rank 0 is a constant in $\mathcal{D}$.

2.2 Regular Tree Grammars

A tree grammar is a device which generates a tree language. Regular tree grammars are tree grammars consisting of two signatures called terminals and nonterminals, a finite set of rules called productions and a start symbol. All nonterminals must have rank 0 and no symbol may be both a terminal and a nonterminal. The start symbol is one of the nonterminals. A production defines a possible rewrite of a nonterminal to a tree over a combined signature of the terminals and the nonterminals. Trees are generated by, starting with the start symbol, rewriting nonterminals using the productions. All trees derived this way that only contain terminals are part of the tree language generated by the regular tree grammar. The resulting tree language is usually infinite (containing an infinite amount of trees), but may as well be finite or even empty.

Example (*Regular tree grammars*) Consider a regular tree grammar with the terminals $\{f^{(2)}, c^{(0)}\}$, the nonterminals $\{A^{(0)}, B^{(0)}\}$, the productions $\{A \to$

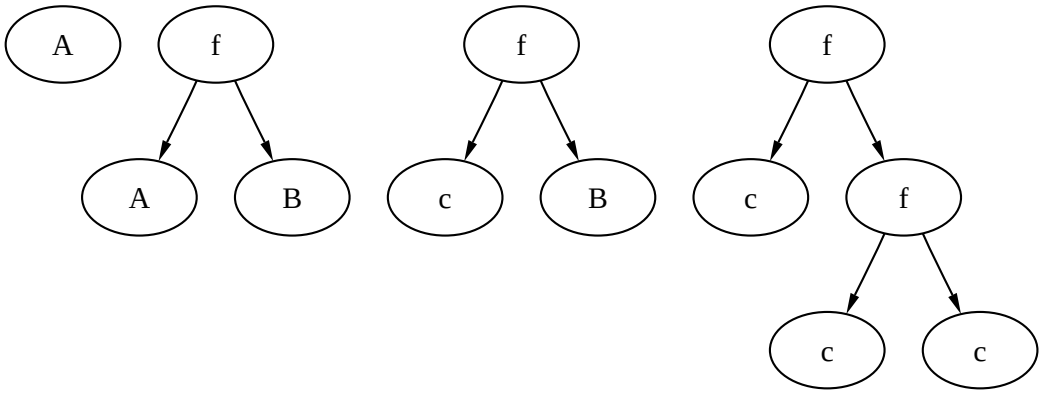

Figure 2.2: Regular tree grammar example.

$f[A, B], A \to c, B \to f[c, c]\}$ and $A^{(0)}$ as start symbol. $A \to f[A, B]$ means that A can be replaced by a tree with f as root node, having A and B as children. In Figure 2.2, the steps in generating a tree with this grammar can be seen. The first tree includes only the start symbol. In the second tree, A is replaced with $f[A, B]$ (the production $A \to c$ could have been used as well, directly ending up with a tree with only terminals). In the last two trees, A is replaced with c and B is replaced by $f[c, c]$. Note that only the last tree in the figure is a tree generated by the grammar since the others still have nonterminals in them (but it is not the only tree generated by the grammar).

Definition (*Regular tree grammar*) A *regular tree grammar* is a quadruple (N, Σ, R, S) consisting of a finite output signature Σ, a finite signature N of nonterminals of rank 0 disjoint with Σ, an initial nonterminal $S \in N$ and a finite set R of rules having the form $A \to t$ where $A \in N$ and $t \in T_{\Sigma \cup N}$. The application of a rule $A \to t$ means replacing a node labelled A with the tree t.

Definition (*Regular tree language*) Let $g = (N, \Sigma, R, S)$ be a regular tree grammar. The notation $t \underset{R}{\rightarrow} t'$ is used to denote that t' is obtained from t by applying some rule in R to a single nonterminal in t. A sequence $t_1 \underset{R}{\rightarrow} t_2 \underset{R}{\rightarrow} \ldots \underset{R}{\rightarrow} t_n$ may be written as $t_1 \overset{*}{\underset{R}{\rightarrow}} t_n$. The *regular tree language* generated by g is $L(g) = \{t \in T_\Sigma \mid S \overset{*}{\underset{R}{\rightarrow}} t\}$.

Regular tree languages are a class of tree languages that is related to the class of context-free string languages. If L is a regular tree language, then $L' = \{yield(t) \mid t \in L\}$ will be a context-free string language. The set of all derivation trees for a context-free string language L' will be a regular tree language whose yield is L'. Note that every nonempty context-free string language can be obtained as the yield of infinitely many different regular tree languages. This fact is closely related to the observation that every context-free string language is generated by infinitely many different context-free Chomsky grammars.

2.3 Top-Down Tree Transducers

A tree transducer is a device that transforms trees. The particular type of tree transducers considered here, top-down tree transducers, will be described and defined further down. To understand how they work we will first consider how finite-state string transductions work.

Example (*Finite-state string transduction*) Consider an automaton that transforms each letter in an input string one after another with two independent sets of rules, corresponding to two different states. The rules that the automaton uses are shown in Figure 2.3. The change at any position in the string depends both on the letter at that position and the current state. The transformation of the string `abaabb` can be seen in tabular form in Figure 2.4. Figure 2.5 shows the same transformation as if the string was a tree. Every letter in the string is treated as a node that has the next letter

State	Input	Action
Q	a	Transform into b.
Q	b	Transform into a and switch to state R.
R	a	Transform into c.
R	b	Transform into a and switch to state Q.

Figure 2.3: Rules for a simple finite-state string transduction.

Output	State	Input	Action
Empty	Q	abaabb	Transform into b.
b	Q	baabb	Transform into a and switch to state R.
ba	R	aabb	Transform into c.
bac	R	abb	Transform into c.
bacc	R	bb	Transform into a and switch to state Q.
bacca	Q	b	Transform into a and switch to state R.
baccaa	R	Empty	

Figure 2.4: Finite-state string transduction example.

as child. A top-down tree transduction works very similarly to this, with the major difference that each node may have more than one child.

A tree transducer indirectly generates a tree language, the set of output trees, from another tree language, the set of input trees. With a top-down tree transducer, the transformation starts from the root node, the top, and continues downwards through the tree. Associated with a top-down tree transducer are three signatures, one each for the input and output trees and one with symbols of rank 1 called states. The states may not have any symbols in common with the input and output signatures, but the input and output signatures themselves may contain one or more identical symbols. An initial state, which is one of the states, as well as a set of rules (similar to productions in regular tree grammars) are also needed.

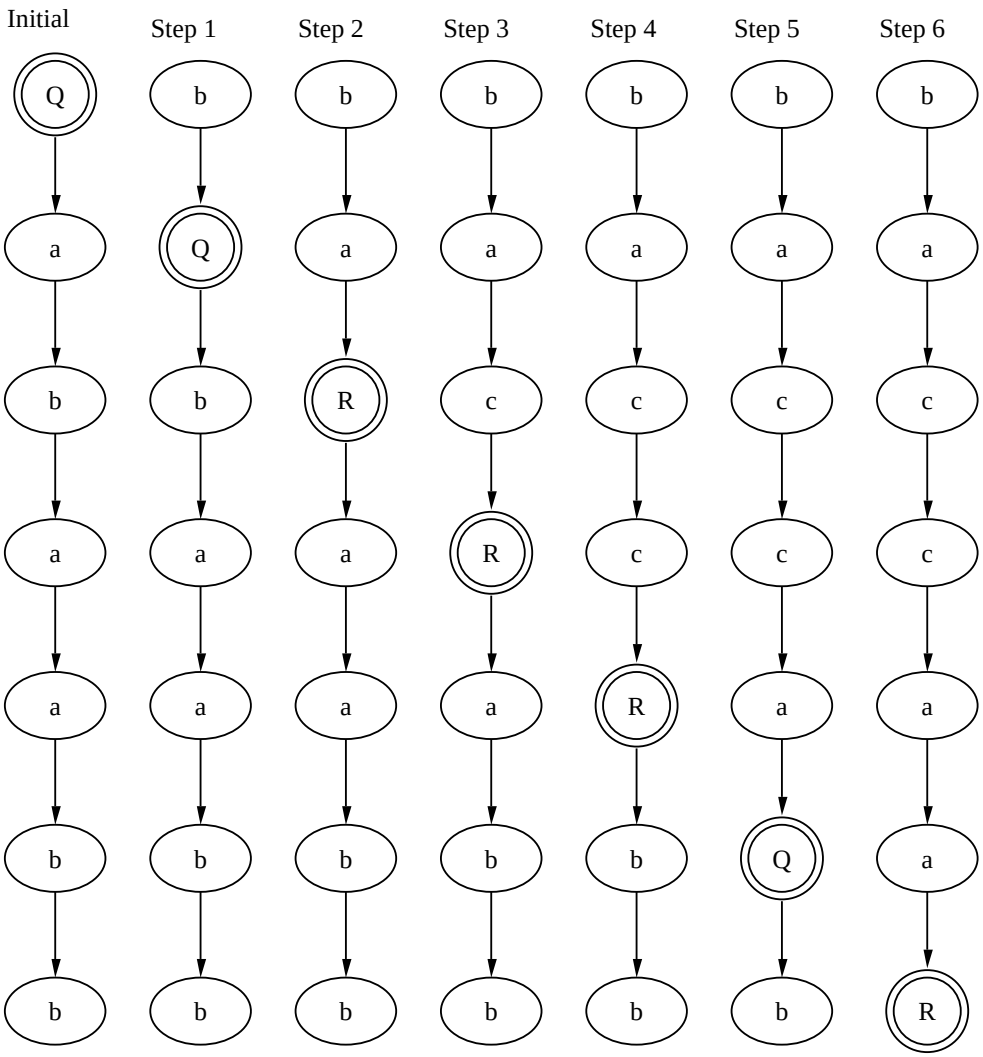

Figure 2.5: Representing the string transduction with trees.

The output tree is constructed by applying the rules on the input tree. Which rule to apply depends on the currently encountered symbol and the current state. Initially the current state is the initial state, which is placed on top of the root of the input tree. A rule basically says that in a specific state a certain symbol in the input tree should result in a certain subtree in the output tree. The output subtree can contain symbols from the output signature as well as states, which will result in rules being applied on subtrees of the current subtree. In this way, the input tree is consumed symbolwise from the top to the leaves. A tree generated by the top-down tree transducer is reached when an output tree only contains symbols from the output signature.

Example (*Top-down tree transducers*) Consider a top-down tree transducer with the input signature $\{f^{(2)}, c^{(0)}\}$, the output signature $\{g^{(3)}, h^{(1)}, d^{(0)}\}$, the states $\{A^{(1)}, B^{(1)}\}$, the rules $\{A[c] \to d, A[f[x_1, x_2]] \to g[A[x_1], B[x_2], A[x_1]]$, $B[f[x_1, x_2]] \to h[d]\}$ and $A^{(1)}$ being the initial state. $A[f[x_1, x_2]] \to g[A[x_1]$, $B[x_2], A[x_1]]$ means that if the state is A and the subtree that currently is being worked on looks like $f[\ldots, \ldots]$ a tree should be produced that looks like $g[\ldots, \ldots, \ldots]$. The x_n on the right-hand side are replaced by the subtrees matching x_n on the left-hand side. Figure 2.6 shows an output tree being generated. The first tree is the input tree with the initial state attached on top of if, and in the next two trees the changes being made can be seen.

Definition (*Variables and Substitution*) Let $X = \{x_1, x_2, \ldots\}$ be a signature of symbols of rank 0 called *variables*. These symbols are considered reserved for their use as variables and must not occur in ordinary signatures. X_n is used to denote $\{x_1, \ldots, x_n\}$. The *substitution* of the variables $x_1, \ldots, x_n$ with the trees $t_1, \ldots, t_n$ within the tree t is denoted $t[\![t_1, \ldots, t_n]\!]$.

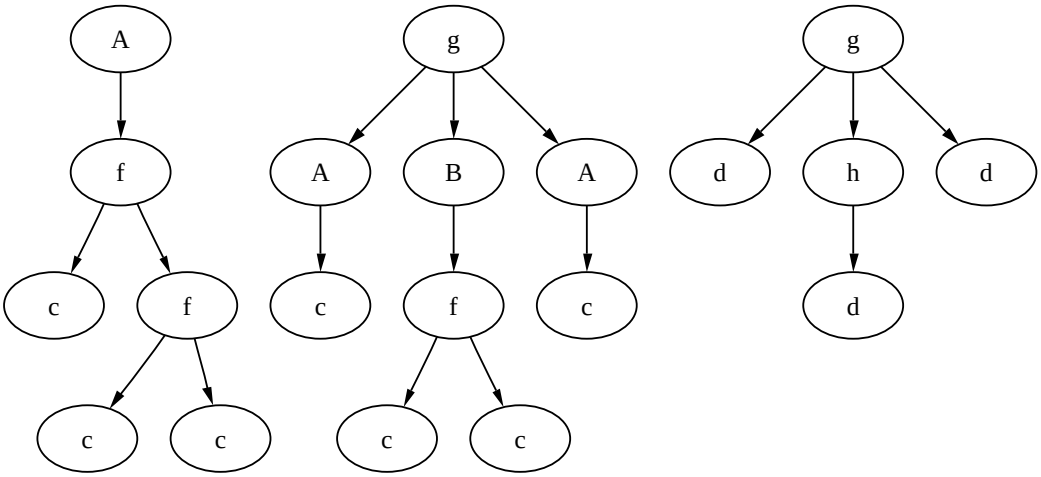

Figure 2.6: Top-down tree transducer example.

Definition (*Top-down tree transducer*) A *top-down tree transducer* is a quintuple $(\Sigma, \Sigma', S, R, S_0)$ consisting of finite input and output signatures Σ and Σ', a finite signature S of states of rank 1 disjoint with $\Sigma \cup \Sigma'$, an initial state $S_0 \in S$ and a finite set R of rules. The rules have the form $A[f[x_1, \ldots, x_n]] \rightarrow t[\![x_{i_1}, \ldots, x_{i_m}]\!]$ where $A \in S$, $f^{(n)} \in \Sigma$ and $t \in T_{\Sigma' \cup S \cup X_m}$ with the limitation that any state only may have a variable from X_m as child and variables may only be children to states. The application of a rule $A[f[x_1, \ldots, x_n]] \rightarrow t[\![x_{i_1}, \ldots, x_{i_m}]\!]$ means replacing a tree $A[f[t_1, \ldots, t_n]]$ with the tree $t[\![t_{i_1}, \ldots, t_{i_m}]\!]$.

Definition (*Top-down tree transduction*) Let $td = (\Sigma, \Sigma', S, R, S_0)$ be a top-down tree transducer. The notation $t \underset{R}{\rightarrow} t'$ is used to denote that t' is obtained from t by applying some rule in R to a single state in t. A sequence $t_1 \underset{R}{\rightarrow} t_2 \underset{R}{\rightarrow} \ldots \underset{R}{\rightarrow} t_n$ may be written as $t_1 \overset{*}{\underset{R}{\rightarrow}} t_n$. The *top-down tree transduction* computed by td is $td(t) = \{t' \in T_{\Sigma'} \mid S_0[t] \overset{*}{\underset{R}{\rightarrow}} t'\}$ for all $t \in T_\Sigma$. Top-down tree transducers may be non-deterministic in general (having two or more rules that apply to the same situation). For this reason, $td(t)$ is a set rather than a single output tree.

A regular tree grammar combined with a top-down tree transducer can generate tree languages that cannot be generated by a regular tree grammar alone. For example, the yield of a regular tree language will always be a context-free string language, but the yield of a regular tree grammar combined with a top-down tree transducer is not always a context-free string language. Consider a regular tree grammar with the rules $\{A \to f[A], A \to c\}$. The yield of the regular tree language generated by this grammar is the context-free string language that only contains the string `c`. Combining this with a top-down tree transducer with the rules $\{Q[f[t_1]] \to g[Q[t_1], Q[t_1]], Q[c] \to c\}$ will create a tree language that yields the string language $\{\texttt{c}, \texttt{cc}, \texttt{cccc}, \ldots\} = \{\texttt{c}^{2^n} \mid n \in \mathbb{N}\}$ (every `f` doubles the amount of leaves), which is not context-free.

2.4 Treebag

Treebag is a system written in Java that allows the user to generate and display objects. The generation is handled by various tree generators which includes, but is not limited to, regular tree grammars and top-down tree transducers. To create a displayable interpretation of the generated trees, the concept of algebras is used. Displaying the resulting interpretation is handled by a display component. Modularization is one of the important concepts in Treebag - tree generation, interpretation of the trees and the displaying of the interpretation all take place in different and isolated components. The functionality of Treebag can also be extended by the user by implementing further classes of tree generators, algebras or displays.

2.4.1 Example Usage

A simple usage session of Treebag might include the definition of a tree grammar and then connecting the grammar to an algebra and a display in the worksheet in Treebag. A tree transducer might also be added to

transform the trees generated by the grammar before they are interpreted by the algebra. Defining a tree grammar is done through external files with a syntax depending on the type of grammar (and the same is true for tree transducers). The example given here will use the components most relevant for this thesis, regular tree grammars and top-down tree transducers.

Below is the contents of a file that defines the regular tree grammar used as example in the previous section about regular tree grammars. The first line declares that this specific component is a regular tree grammar and the lines that follow define nonterminals, output signature (the syntax `name:rank` is used for the symbols), rules and the start symbol. The syntax should be easy to understand with basic knowledge of what constitutes a regular tree grammar.

```
generators.regularTreeGrammar:
(
    { A, B },
    { f:2, c:0 },
    {
        A -> f[A, B],
        A -> c,
        B -> f[c, c]
    },
    A
)
```

The definition of the top-down tree transducer uses the example from the section about top-down tree transducers. After the first line of component type declaration comes the definition of input and output signatures, states, rules and the initial state.

```
generators.tdTransducer:
(
    { f:2, c:0 },
    { g:3, h:1, d:0 },
    { A, B },
    {
        A[c] -> d,
        A[f[x1, x2]] -> g[A[x1], B[x2], A[x1]],
        B[f[x1, x2]] -> h[d]
    },
    A
)
```

In TREEBAG, an algebra component takes input trees from some sort of tree generator (such as a regular tree grammar or a top-down tree transducer) and returns as output the resulting object. The type of the resulting object depends on the algebra, in this example the free term algebra is used which simply returns the input tree. A display component takes input objects from an algebra and displays them in a specific way. Since the free term algebra returns trees, a display that draws trees is used. The choice of algebra and display in this case are thus made so that the output picture will be the tree that is generated by the transducer. In the same way as the generator components, algebras and displays need files defining their types. The component definitions in these files simply state `algebra.termAlgebra:` and `displays.treeDisplay: {}`.

In Figure 2.7 the four created components have been added to the TREEBAG worksheet and connected appropriately. One of the possible output trees is shown in the display-window. Every component that has been added to the worksheet provides a set of commands depending on its type and internal state. The user can select any of the commands for an individual component to control it. The control panel labelled "file input" in Figure 2.7 shows the commands that currently are available for the regular tree grammar in this worksheet.

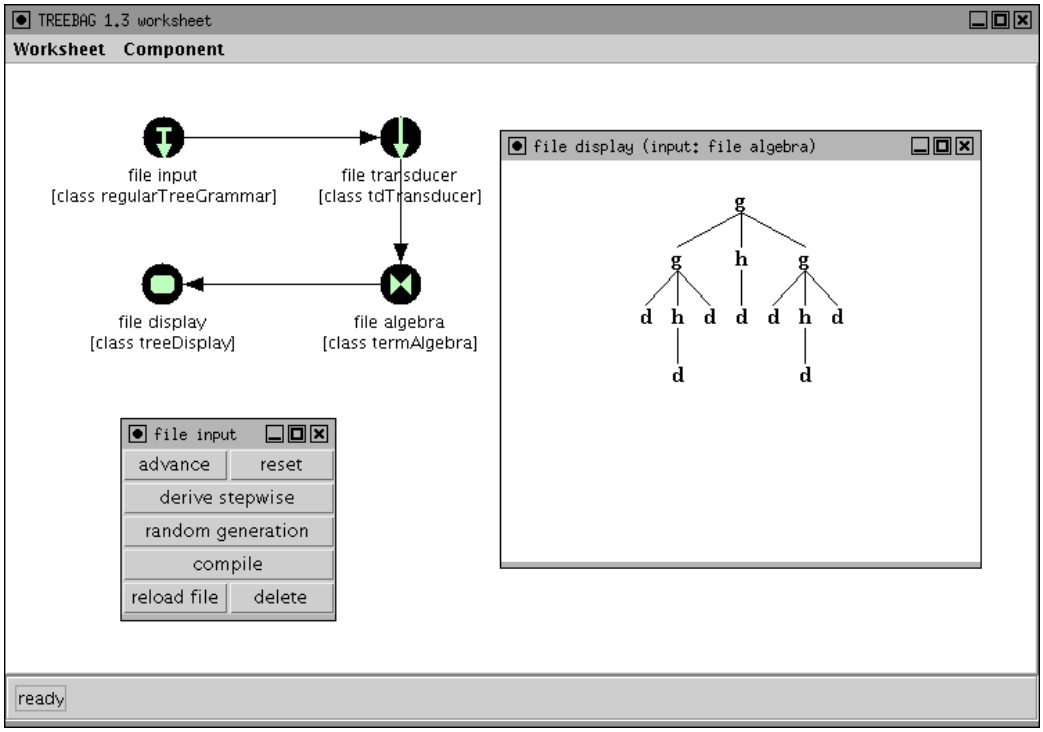

Figure 2.7: TREEBAG with the example components added.

2.4.2 Provided Functionality

The concepts deterministic and non-deterministic are of importance both to regular tree grammars and top-down tree transducers. In this context, deterministic means that there is at most one rule for each situation, and non-deterministic that in some (or all) situations a choice has to be made between two or more rules. In the usage example, the regular tree grammar is non-deterministic and the top-down tree transducer is deterministic. Worth noting is that a deterministic regular tree grammar is of little interest as it generates a tree language containing either exactly one tree or no trees at all.

Regular tree grammars in Treebag allows the user to generate trees in two ways, enumeration and random generation. Enumeration steps through the whole set of trees generated by the grammar in an order that in general can be described as generating the smallest trees first. The exact order may however depend on both the grammar and the order in which rules are defined, Treebag does not strictly ensure a smaller-to-larger ordering. Random generation picks rules randomly between those that are possible for a given nonterminal. It always works in a stepwise manner, thus generating both nonterminal trees (that are not in the generated tree language) and terminal trees (that are in the generated tree language). Enumeration may also generate nonterminal trees in its stepwise-mode which shows step-by-step how a tree is generated.

Top-down tree transducers in Treebag normally transform the entire tree according to its rules, but also have a stepwise-mode to show the transformation step-by-step. In the case of a non-deterministic top-down tree transducer it picks rules randomly where there is a choice. If the top-down tree transducer fails to generate an output tree, the result is null. Failure is unavoidable if there is no output tree, $td(t) = \emptyset$ for the input tree t, but Treebag currently has further limitations; see Chapter 4 for a discussion of this issue.

The possibilities provided by TREEBAG are much greater than those presented here, where only aspects relevant to this thesis are described. If the reader is interested in tree-based picture generation it would be worthwhile to play around a bit with TREEBAG[1]. It is fairly straightforward to use and comes with a set of examples. Some interesting pictures generated with TREEBAG can be seen in Figure 2.8.

2.5 Command Descriptions

The commands available in regular tree grammars and top-down tree transducers within TREEBAG will be described in detail, since one of the fundamental requirements for this thesis was that a compiled component should have the same functionality and behavior as an uncompiled component (except perhaps for minor details such as the enumeration order of trees generated by a regular tree grammar). Readers who only want to get an overview may skip this section.

2.5.1 Regular Tree Grammars

A regular tree grammar is at any point in time in one of three different modes called random generation, enumeration and stepwise. The enumeration and stepwise modes are similar in functionality. When a regular tree grammar is loaded it starts in enumeration mode.

Random Generation

Random generation mode creates trees by replacing nonterminals in the current output tree with rules selected at random. Initially, a regular tree grammar in random generation mode returns an output tree that only contains

[1] TREEBAG can be found at http://www.informatik.uni-bremen.de/theorie/treebag/.

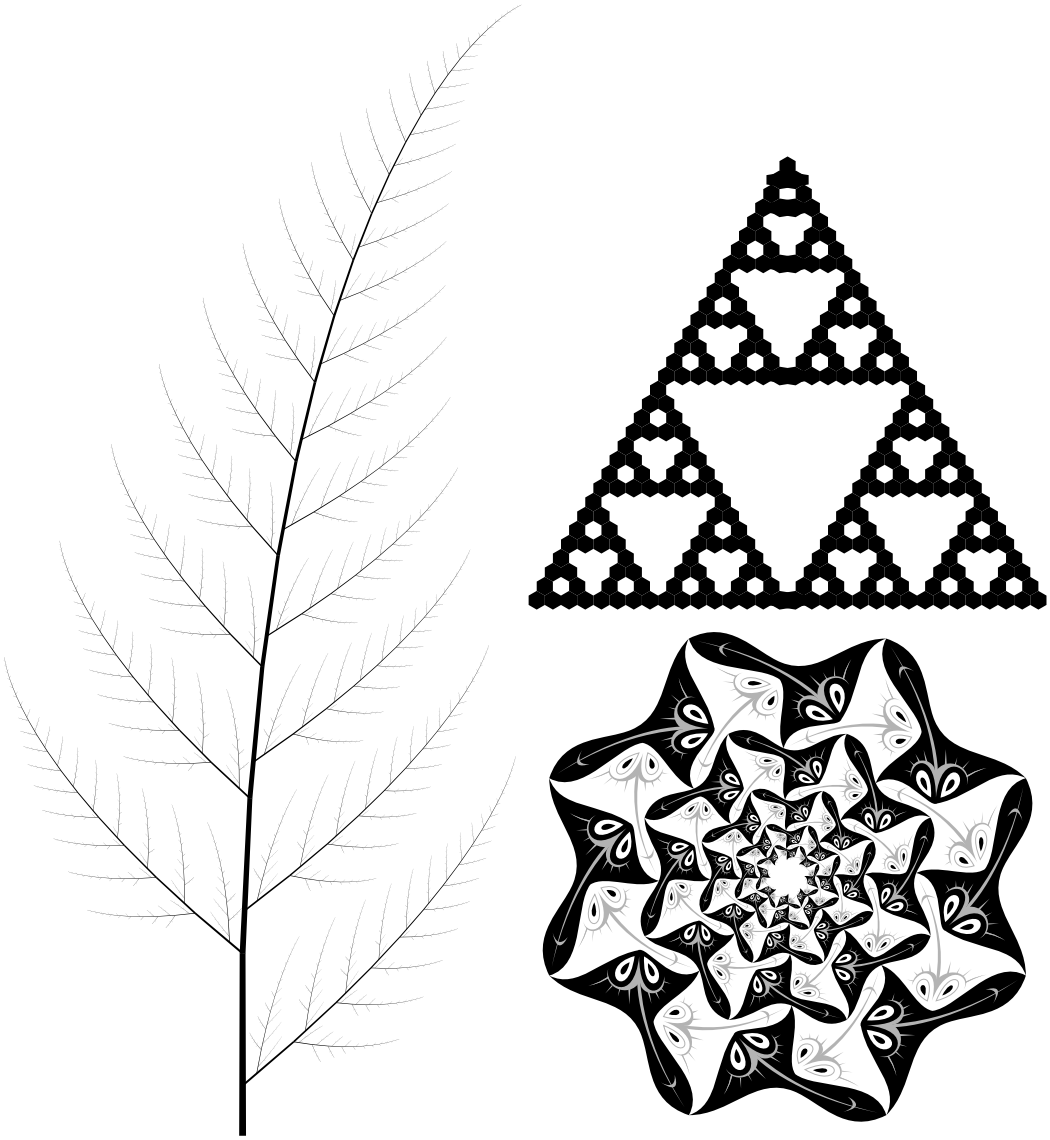

Figure 2.8: TREEBAG generated pictures.

the start symbol. Commands available in random generation mode are given below.

Refine. Replaces all nonterminals of the current output tree with rules selected at random. This may make branches terminal or create new nonterminal leaves.

Back. This operation is the reverse of *Refine.* The rules in the current output tree that were refined last are changed back to the nonterminals they were derived from.

Reset. Sets the current output tree to a tree that only contains the start symbol.

Enumeration. Switches to enumeration mode. Each nonterminal in the current output tree is replaced with a terminal tree derived from it.

Enumeration

Suppose that a given regular tree grammar generates the tree language L. In enumeration mode the component will compute the trees $t_0, t_1, t_2, \ldots$ such that $L = \{t_0, t_1, t_2, \ldots\}$. The initial state of a regular tree grammar in enumeration mode is to return the tree t_0 as output (or null if no such tree exists). As mentioned earlier, no guarantees are made regarding the order in which the trees appear. A specific tree t_i may even appear more than once if it can be generated in more than one way with the rules of the grammar. Commands available in enumeration mode are given below.

Advance. With a current output tree t_i, sets the current output tree to t_{i+1} or null if there is no tree t_{i+1}.

Reset. Sets the current output tree to t_0 or null if no such tree exists.

Derive Stepwise. Switches to stepwise mode.

Random Generation. Switches to random generation mode. The rules at the bottom of the current output tree are changed to the nonterminal from which the rule was derived.

Stepwise

The stepwise mode outputs the same trees as the enumeration mode, but instead of outputting the entire tree at once it allows the user to step-by-step review how the final output tree is built from the rules in the grammar. If no step commands have been made by the user, the tree actually outputted only contains the start symbol. When a step command is issued, one or several nonterminals are replaced using the rules that are used to derive the tree that would be directly outputted in enumeration mode.

Advance, *Reset* and *Random Generation.* Same as in enumeration mode. Using *Advance* or *Reset* also resets the stepping state, the output tree will only show the start symbol until *Single Step* or *Parallel Step* is used.

Single Step. Replaces one nonterminal, in depth-first order, using the relevant rule.

Parallel Step. Replaces all nonterminals at the bottom of the tree using their relevant rules.

Back. Takes back the last single or parallel step.

Results Only. Switches to enumeration mode.

2.5.2 Deterministic Top-Down Tree Transducers

A deterministic top-down tree transducer has two modes of operation, normal and stepwise mode. When a top-down tree transducer is loaded it starts in normal mode.

Normal

In normal mode, tree transductions are made as soon as an input tree is available and the resulting output tree is returned as result.

Derive Stepwise. Switches to stepwise mode.

Stepwise

Stepwise mode shows the current tree transduction step-by-step. Before any steps have been made, the output tree is a tree with the initial state as root and the input tree as its direct subtree.

Single Step. Applies a rule to exactly one state, in depth-first order.

Parallel Step. Applies rules to all states at the bottom of the tree.

Back. Takes back the last single or parallel step.

Results Only. Switches to normal mode.

2.5.3 Non-Deterministic Top-Down Tree Transducers

A non-deterministic top-down tree transducer has additional commands that alters how random choices are made (but otherwise has the same commands as a deterministic top-down tree transducer).

Variable Seed

In variable seed mode, a new random seed is picked whenever a new input tree is received.

Fixed Seed. Switches to fixed seed mode.

New Random Seed. Picks a new random seed and recomputes the last tree transduction using the new seed.

Fixed Seed

In fixed seed mode, a new random seed is only picked when the user issues the *New Random Seed* command.

Variable Seed. Switches to variable seed mode.

New Random Seed. Picks a new random seed and recomputes the last tree transduction using the new seed.

Chapter 3

Approach

This chapter describes how the compilation of the regular tree grammars and top-down tree transducers within TREEBAG works. The structure of the output as well as the design of the compiler extensions and their interaction with TREEBAG is discussed. Focus is on how the result of the compilation works, not how the compilation itself is performed. Some concepts are intentionally described in a sketchy manner, where a more accurate description would make things harder to understand without providing important information.

3.1 General Structure

The compiler extensions are handled within a new Java package that is separated from TREEBAG as much as possible. To the rest of TREEBAG, the visible parts of this package are one exception class for reporting failures and one class each for the tasks of compiling a regular tree grammar and a top down tree transducer. TREEBAG has a class that represents a regular tree grammar and supplies the operations that are supported with regular tree grammars. Another class serves the same purpose for top-down tree transducers. When the compilers compile a specific regular tree grammar

or top-down tree transducer, a new class is created that inherits from one of these classes. Most methods are overloaded in the inheriting classes since they only implement the supported operations for a specific regular tree grammar or top-down tree transducer.

Integration of the extensions into TREEBAG requires some minor changes to the original regular tree grammars and top-down tree transducers. The actual parsing of input files is not handled by the compiler extensions. Modifications have been made to the already existing parsing process so that nonterminals, states and rules are passed on to the compiler extensions. Regular tree grammars and top-down tree transducers have a new function, "compile", that compiles the generator and starts using the compiled implementation instead of the uncompiled.

3.1.1 Output Classes

The classes that the compilers create use the same general design idea. Any nonterminal or state is represented as an internal class within the output class. All rules are also represented as internal classes that inherit from the nonterminal or state on their left-hand side. An internal class that represents a rule has any nonterminals or states that are on the right-hand side of the rule as attributes. The internal classes have methods that are similar in functionality to those operations that the grammar or transducer supports. Instantiated objects of the internal classes represent the current tree within the grammar or transducer. Replacing a nonterminal or state within the tree is thus done by replacing objects. This internal representation is not itself the resulting tree, an internal class that represents a rule may correspond to more than one node in the resulting tree. An example that illustrates this difference will be given further down.

In Figure 3.1 the internal classes that would be constructed from the regular tree grammar used in previous examples can be seen. All the rules

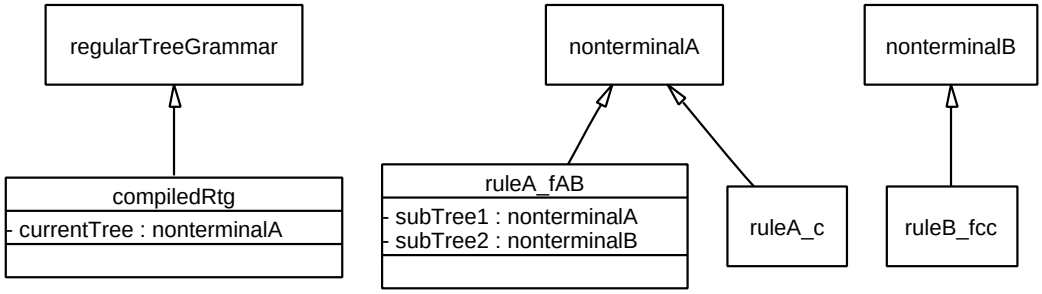

Figure 3.1: Overview of a compiled regular tree grammar.

in the grammar, $\{A \to f[A, B], A \to c, B \to f[c, c]\}$, as well as the nonterminals, have their own classes. The class `regularTreeGrammar` is the class that TREEBAG uses for normal operation on regular tree grammars and `compiledRtg` is the compiled output class that contains the internal classes. Names for the internal classes in the figure are chosen to make understanding easier and are not consistent with the names actually generated by the compiler. At runtime, an object of the class `compiledRtg` contains an object of the class that corresponds to the start symbol, in this case `nonTerminalA`, to represent the current tree. When the current tree is something other than a tree containing just the start symbol, the actual object will be from one of the rule classes that inherit from `nonTerminalA` (which due to inheritance also qualifies as type `nonTerminalA`). If the object is of type `ruleA_fAB` it will contain two objects of type `nonTerminalA` and `nonTerminalB` to represent the subtrees of that rule. To make it less confusing that a rule contains subtrees it might be helpful to consider an object of a rule class as the tree that results from the application of that rule. Any operation that modifies the current tree will change what objects are stored at the modified locations (which might include a change of the root node, `currentTree`).

Figure 3.2 shows the difference between the internal representation and the resulting tree for a tree in the example grammar. As mentioned earlier, rule classes can represent more than one node. Another less obvious difference is

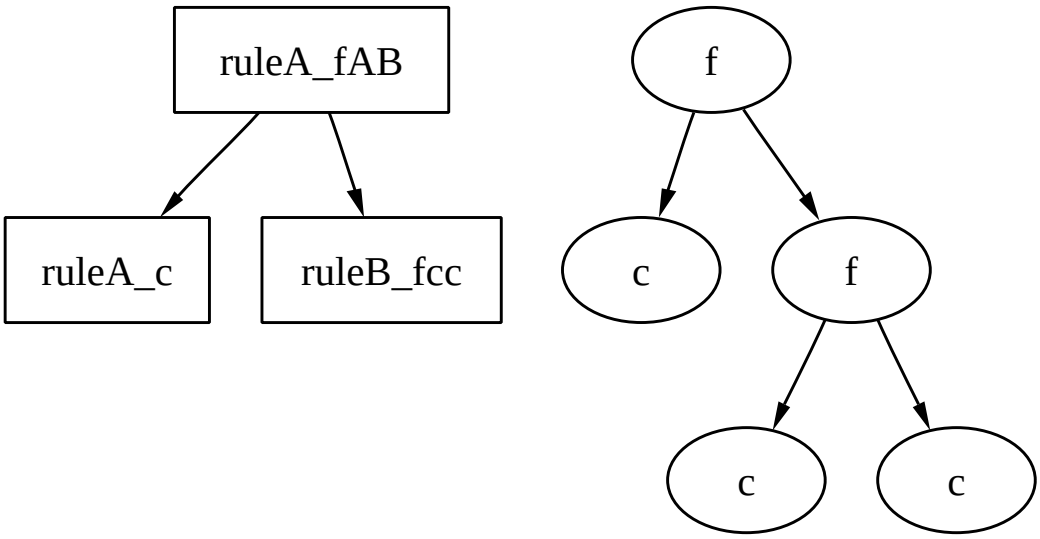

Figure 3.2: Internal representation and the resulting tree.

that the internal representation keeps track of how a tree has been generated and not just how the tree looks like. Since the rule classes know which nonterminal they are generated from it is also known how an entire tree is generated. This is necessary in several of the operations supported by Treebag. A good way to look at the internal representation is as a data structure (which happens to be a tree itself) that shows which rules have been applied to generate the resulting tree. It is thus very similar to a derivation tree. There will exist several unique internal representations of a single resulting tree if it can be generated in several ways within the grammar or transducer. In particular, this implies that output trees from a regular tree grammar may occur several times in the enumeration order even though the implementation makes sure that every internal representation is generated exactly once.

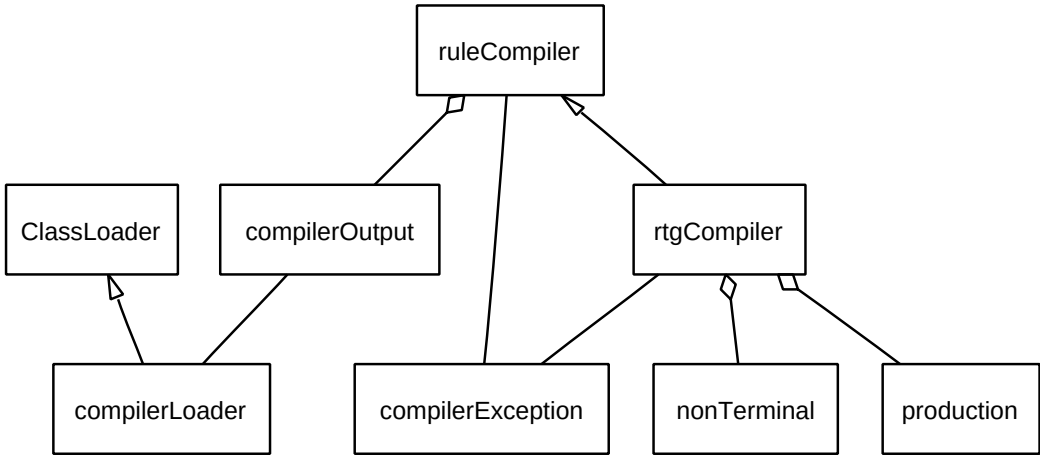

Figure 3.3: Class overview for the compiler.

3.2 Compiler Classes

In Figure 3.3, the classes used by the compiler can be seen. There is also a class `tdtCompiler`, as well as some special helper classes that are excluded from the figure to reduce its size. The `tdtCompiler` class and the `rtgCompiler` class are very similar, they both inherit from `ruleCompiler` and use the same classes. Compilation of regular tree grammars is handled by `rtgCompiler` and compilation of top-down tree transducers by `tdtCompiler`. Basic functionality such as adding rules and setting the start symbol is implemented by `ruleCompiler`. It also provides methods for performing parts of the compilation process, `rtgCompiler` and `tdtCompiler` only implement methods that create strings with Java code for nonterminals and rules. Output of the strings to a file and the compilation of that file is handled by methods in `ruleCompiler` (which uses `compilerOutput` for those tasks). Any error during compilation is reported by throwing a `compilerException`.

For every rule in the grammar or transducer being compiled, there is an instance of the class `production`, and for every nonterminal or state there is an instance of the class `nonTerminal`. Every `nonTerminal` keeps track of

what rules it is related to, and every `production` keeps track of which nonterminals or states it contains. Both of these classes provide several methods that compute useful information for the compilation process. Compilation of Java code is handled by the `compilerOutput` class, which currently outputs the Java code to a file and executes `javac`. The resulting class is then loaded into the virtual machine with `compilerLoader`. If another approach for the Java compilation would become desired, changes only need to be made in `compilerOutput`.

3.2.1 Helper Classes

The special helper classes earlier mentioned are string building classes and are not implemented directly in Java. A script is used to translate template files into string generation code. The string generation code becomes a part of the compiler that during the execution of Treebag generates the compiled grammars and transducers. Special control sequences within the template files are used to modify the actual output depending on the specific grammar or transducer being compiled. The template files can be seen as a general description of how the output classes should look. This approach was chosen since the strings with Java code needed for a compiled grammar or transducer become quite large, so that it is impractical to create the string generation code manually. Within the compiler extensions there are three helper classes each for `rtgCompiler` and `tdtCompiler`. They create strings with Java code for the internal nonterminal classes, the internal rule classes and the class that contains the internal classes.

Each template file builds methods that execute a series of string concatenations. Figure 3.4 shows an example of how a section that builds one method can look. The first line declares a new string building method named `createArrayAdd` that takes one argument. Anything between the first and last line is the string that the new method will create except for the parts with `#` characters. Lines starting with `#` are inserted into the string building

```
### createArrayAdd( int arraySize)
void arrayAdd( int add, int[] array)
{
    if ( array.length != #arraySize# ) { throw X; }
# for ( int i = 0; i < arraySize; i++ ) {
    array[ #i# ] = array[ #i# ] + add;
# }
}
###
```

Figure 3.4: Example contents of a template file.

method as code. Expressions enclosed within # characters are added to the string as the value they evaluate to (which thus must be a string). In Figure 3.5 the string building method created from this template can be seen. As the size of the generated strings grows bigger (when the result is the code for a regular tree grammar class for instance) the advantage of not having to create the string generation code manually also becomes larger.

3.3 Regular Tree Grammars

This section describes how the operations supported by the Treebag realization of regular tree grammars work in a compiled regular tree grammar. In Figure 3.6 the most important methods within an internal class (`rtgInternalClass`, which may be either a nonterminal or a rule) can be seen.

3.3.1 Random Generation

As discussed in the previous chapter, the random generation mode has the operations refine, back and reset. Refine replaces all nonterminals in the

```
public String createArrayAdd( int arraySize)
{
    String s = "void arrayAdd( int add, int[] array)\n";
    s = s + "{\n";
    s = s + = "if ( array.length != " + arraySize +
        " ) { throw X; }\n";
    for ( int i = 0; i < arraySize; i++ ) {
        s = s + "array[ " + i + " ] = array[ " + i +
            " ] + add + "\n";";
    }
    s = s + "}\n";
    return s;
}
```

Figure 3.5: Resulting method from the template in Figure 3.4.

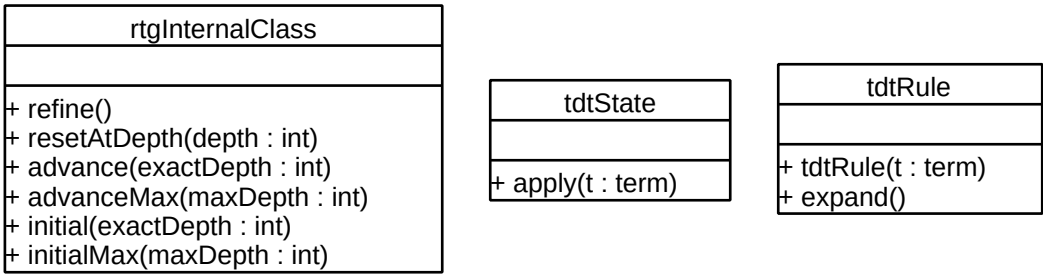

Figure 3.6: Important methods in the internal classes.

current tree with valid rules. The rules are chosen at random where several choices exist, and they can be given weights to affect how likely it will be that a specific rule is chosen. Back does the inverse of refine and reset sets the current tree to its initial state, a tree containing only the start symbol.

The internal classes handle these operations with two methods, `refine()` and `resetAtDepth()`. Reset simply creates a new object of the internal class that represents the start symbol and sets it as the current tree. Back uses the method `resetAtDepth()`, which changes all rules at a given depth to their parent nonterminals. To step back with the correct rules the current tree depth must be kept track of. The `refine()` method, used by the refine operation, calls `refine()` on all its subtrees if invoked on a rule. If invoked on a nonterminal it changes the nonterminal to one of its rules. When several rules are possible choices, one of them is picked at random.

3.3.2 Enumeration

Enumeration has the operations advance and reset. Advance sets the current tree to the next tree in the enumeration order. Reset sets the current tree to the first tree in the enumeration order. The advance operation is the most complicated part of the compilers and will be discussed in more detail than other parts. An important concept for this discussion is the depth of a tree. The depth of a tree is the maximum length of direct paths from the root to a leaf in that tree. In particular, the depth of a tree consisting only of its root is zero. As an example, the depth of the tree $f[f[c, c], c]$ is two.

The algorithm used by advance can be divided into two parts, one algorithm that creates an initial terminal tree for a nonterminal (the first tree in the order) and one algorithm that creates the next tree in the order from another tree. These two algorithms are referred to as initial and advance. Both algorithms are used in two slightly different variants, one where the resulting tree must be of an exact depth (*Initial* and *Advance*) and one where the

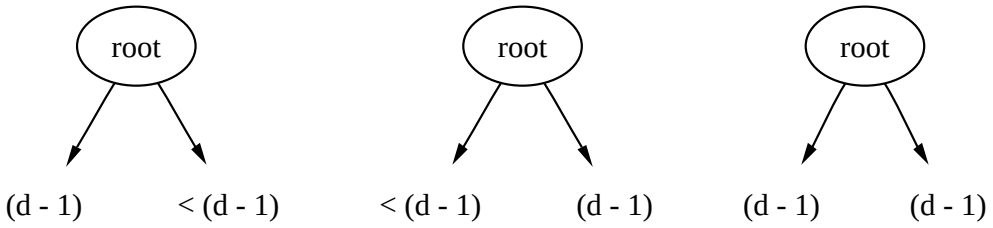

Figure 3.7: Branch selection algorithm.

resulting tree must be smaller or equal to a given maximum depth (*InitialMax* and *AdvanceMax*). The algorithms that target an exact depth both use the same technique to ensure that the depth requirement is satisfied. This algorithm is referred to as *Branch selection*. Two smaller algorithms that increase performance are also described, *Branch skipping* and *Result caching*.

Algorithm (*Branch selection*) Consider a target tree depth d and a tree t where the root has n branches. In order for t to be of depth d at least one of the n branches must be of depth $d-1$. A binary counter consisting of n bits is used to determine which branches are less deep. The branches are numbered from the left and a branch must be of depth $d-1$ if there is a one in the binary counter at the position that corresponds to its number. The binary counter is initially 1_2 and is increased by one when a new branch setup is needed (if there are no further trees that satisfy the current setup). When all branches have zeroes in the binary counter all possible setups have been tried. To avoid creating the same tree several times branches that have a zero at its position in the binary counter are required to be of depth strictly less than $d-1$.

Example (*Branch selection*) Figure 3.7 shows the three different setups that will be tried when creating a tree of depth d for a root node with two children. Initially the state of the binary counter is 01_2 which means that the leftmost branch must be of depth $d-1$ and the rightmost of depth less than $d-1$.

The next setup of the binary counter, 10_2, has the opposite configuration and the final setup, 11_2, requires that both branches are exactly $d-1$ deep.

Algorithm (*Branch skipping*) Any of the algorithms that uses the *Branch selection* algorithm skips branch setups if they are known to fail. When a branch cannot be created for a certain position in the binary counter, the next branch setup tried is one where the bit of that branch has been changed. The next setup would thus have zeroes for all branches to the left of the one that failed and if there was a one at the failing position there would also be a "carry effect" to branches to the right. If the binary counter is 0101_2 and the third branch failed the next setup tried would be 1000_2 instead of 0110_2.

Algorithm (*InitialMax*) Consider a nonterminal A to build an initial tree of maximum depth d from, and assume that there are n rules $R_1, \ldots, R_n$ with A on their left-hand side (and there are no further rules with A on their left-hand side in the grammar).
If d is less than zero the algorithm fails.
If d is zero the first R_i (meaning that i should be as small as possible) that is terminal (no nonterminals on its right-hand side) is chosen as the result. If no such R_i exists the algorithm fails.
If d is greater than zero every R_i is tried until a rule is found that can create a tree of maximum depth d. In particular, this will succeed if R_i is terminal. In general, it is necessary that a recursive call to *InitialMax* with $d-1$ as maximum depth is successful for all nonterminals on the right-hand side of R_i. The result of the algorithm for such an R_i has the nonterminals replaced by the results from the recursive calls. If no appropriate rule can be found, the algorithm fails.

Algorithm (*Initial*) Consider a nonterminal A to build an initial tree of exact depth d from, and assume that there are n rules $R_1, \ldots, R_n$ with A on their left-hand side (and there are no further rules with A on their left-hand side in the grammar).
If d is less than zero the algorithm fails.
If d is zero the first R_i that is terminal is chosen as the result. If no such R_i exists the algorithm fails.
If d is greater than zero the *Initial* and *InitialMax* algorithms are used recursively. The *Branch selection* algorithm is used to determine if a specific branch (corresponding to a nonterminal on the right-hand side of a rule) should be of exact depth $d-1$ (resulting in a recursive call to *Initial* with $d-1$ as target depth) or of maximum depth $d-2$ (resulting in a call to *InitialMax* with $d-2$ as maximum depth). The first rule that has a branch setup where all the calls succeed is chosen as result (with the nonterminals replaced by the results of the calls). If there was no such rule *Initial* fails.

Example (*Initial and InitialMax*) Consider a grammar with the rules $\{R_1 = A \to c, R_2 = A \to f[A, A], R_3 = A \to d\}$. Using *InitialMax* on A with any depth equal to or greater than zero will result in R_1 (R_1 is always picked instead of R_2 or R_3 since $1 < 2 < 3$). With *Initial* the result is the same for depth zero. For depth one R_2 is the only candidate for the root node since using R_1 or R_3 would give a tree of depth zero. The first branch setup given by the *Branch selection* algorithm is 01_2, so the leftmost branch should be of depth one and the rightmost should be of depth minus one. This fails since no tree can have negative depth. Similarly, the next branch setup fails since the first branch would need a negative depth. The third and last branch setup requires that both branches have depth zero which is possible for R_1 or R_3 as children to the A branches (resulting in $R_2[R_1, R_1]$).

Algorithm (*Result caching*) The *Initial* and *InitialMax* algorithms remember failures to increase performance. Every nonterminal has a set of bits that initially are zero (one set each for *Initial* and *InitialMax*). When an initial tree is needed for a nonterminal for a depth d, bit number d in the set of bits is checked. If the bit is zero the algorithms try to create an initial tree the way described above. If no initial tree was possible for depth d, bit number d is set to one. If the bit already is one the algorithms directly report that they cannot create an initial tree of that depth. In the current implementation, the size of the bit set is limited to 50 bits. Any request for a tree of depth greater than 50 does not directly use *Result caching* (subtrees to the requested tree that are of depth 50 or lower will however take advantage of it).

Algorithm (*AdvanceMax*) Consider a tree t whose root is the rule R_i, a maximum depth d and n rules $R_1, \ldots, R_n$ that have the same parent nonterminal as R_i (with R_i being one of these). If applied to t *AdvanceMax* works as follows:
If d is less than zero the algorithm fails.
If d is zero then the result is the first R_j that is terminal where $j > i$. If there is no such R_j the algorithm fails.
If d is greater than zero the algorithm is used recursively on the children to the rule R_i in t with a maximum depth $d - 1$. The basic idea behind advancing the children is to advance the first child until the *AdvanceMax* algorithm fails. When the algorithm fails the second child is advanced one step and the first child is reset to its first tree (using the *InitialMax* algorithm). When the algorithm fails for the second child the third child is advanced one step while the first two are reset using *InitialMax* and so on. As soon as this has succeeded for one of the children the whole tree is returned as result. If the algorithm fails for all children the result is the first R_j (terminal or nonterminal) where $j > i$ and *InitialMax* (again with a maximum depth of $d - 1$) is successful on all nonterminals on the right-hand side of R_j. If there is no such R_j the algorithm fails.

Example (*AdvanceMax*) Consider the same grammar as in the *Initial* example, a tree $R_2[R_1, R_1]$ to advance and a maximum depth of one. First the algorithm tries to advance the children of R_2 with a maximum depth of zero. This is successful for the first child giving $R_2[R_3, R_1]$ as result. Advancing this again fails for the first child (resetting it to R_1) and the second child gets advanced from R_1 to R_3 (giving $R_2[R_1, R_3]$). The next advance gives $R_2[R_3, R_3]$. Trying to advance once more fails for both children and the rule R_3 is tried as root instead of R_2. This succeeds since a terminal rule satisfies a maximum depth of one, giving the result R_3.

Algorithm (*Advance*) Consider a tree t whose root is the rule R_i, an exact depth d and n rules $R_1, \ldots, R_n$ that have the same parent nonterminal as R_i. If applied to t, *Advance* works as follows:
If d is less than zero the algorithm fails.
If d is zero then the result is the first R_j that is terminal where $j > i$. If there is no such R_j the algorithm fails.
If d is greater than zero the algorithm is used recursively on the children of the rule R_i in t with an exact depth $d - 1$ or a maximum depth $d - 2$ as determined by the *Branch selection* algorithm. A child is either advanced using an exact depth $d-1$ with the *Advance* algorithm or using a maximum depth $d - 2$ with the *AdvanceMax* algorithm and reset with either *Initial* with $d - 1$ or *InitialMax* with $d - 2$. Instead of trying the next rule as in *AdvanceMax* when no child could be advanced further the next *Branch selection* setup is tried. When the setups have all been tried the result is the first R_j where $j > i$, such that there is a *Branch selection* setup that makes the algorithm succeed for the exact depth d. If there is no such R_j the algorithm fails.

Example (*Advance*) Consider the same grammar as in the *Initial* example, a tree $R_2[R_1, R_1]$ and an exact depth of one. This tree gets advanced to the same trees as with *AdvanceMax* until R_3 is tried as root. Instead of succeeding with R_3 as result it fails since R_3 is less deep than the required

depth. Consider another tree $R_2[R_1, R_2[R_3, R_3]]$ having depth two and a binary counter at 10_2 (the first branch is less than depth one, while the second has depth one). *AdvanceMax* will be called for the first child with a maximum depth of zero which yields R_3, giving $R_2[R_3, R_2[R_3, R_3]]$ as result. Advancing this tree further fails on the first branch and a call to *Advance* with exact depth one on the second branch also fails. Note that there is a binary counter for this R_2 node as well, which must be at 11_2 since both branches are of depth one. It fails since neither of the R_3 children could be advanced and the binary counter is increased to 100_2 putting zeroes on both branches. The algorithm does however not fail for the root node since its binary counter is increased to 11_2 which gives the tree $R_2[R_2[R_1, R_1], R_2[R_1, R_1]]$ as result ($R_2[R_1, R_1]$ being the result returned by *Initial* with depth one on A).

In a compiled grammar, four methods handle the enumeration mode in the internal classes, `advance()`, `advanceMax()`, `initial()` and `initialMax()`. These methods perform the algorithms described above. The *Branch selection* algorithm is used inside the `advance()` and `initial()` methods. The reset operation uses the `initialMax()` method to create the first tree in the enumeration and the `advance()` method to create the trees that follow. If the first tree created by `initialMax()` was of depth 0, then the advance operation will successively create all trees of exact depth 0 until that fails. After that `initial()` is used to create the first tree of exact depth 1 (if possible) and advance will create all the trees of depth 1 until that fails. This procedure is repeated until there are no more trees (or forever if there is an infinite number of trees). The fact that there are no more trees in the language is discovered by knowing how many nonterminals there are in the grammar. In the worst case a terminal tree is reached after using every nonterminal once (with each nonterminal increasing the depth of the internal tree by one). For a grammar with n nonterminals, there is no need to try deeper trees after n depths have failed in a row.

3.4 Top-Down Tree Transducers

Let us now turn to the discussion of compiled top-down tree transducers. Important methods within an internal class of a compiled top-down tree transducer can be seen in Figure 3.6. States (`tdtState` in the figure) only have an `apply()` method. A transduction is initiated by invoking this method with the input tree as argument. Rules (`tdtRule` in the figure) have an `expand()` method and a constructor that takes a tree as argument. As an overview, the `apply()` method determines which rule should be chosen for that state and input tree, the constructor in rules is used to pass on the input tree to the rule and the `expand()` method activates `apply()` on any new states within that rule.

When the `apply()` method is invoked on a state, it chooses an appropriate rule to replace the state (a rule that accepts the root node of the input tree). In a non-deterministic transducer, the rule is picked randomly. As with regular tree grammars, the random choices can be affected by setting weights on rules. A new rule object is constructed using the input tree as argument and is put in a queue. New rules are queued to ensure that transductions take place in a breadth-first order. It is sometimes desirable that transductions make the same random choices on similar trees (when a fixed random seed is used) and a breadth-first transduction makes this easier since changes far down in the tree do not change the transduction in the upper parts of the tree. An entire transduction starts by a call to `apply()` on an object that represents the initial state. After the rule created by `apply()` has been queued, `expand()` is called on that rule. New states will be created by the `expand()` method if the rule contains states on its right-hand side. On those new states `apply()` will be invoked with the subtrees of the input tree that the states are linked to in the rule. These calls to `apply()` will queue more rules that later will receive `expand()` calls.

Example (*Top-down tree transduction*) Consider a top-down tree transducer with one state, S, and two rules, $R = S[f[x_1]] \rightarrow g[S[x_1], S[x_1]]$ and $Q =$

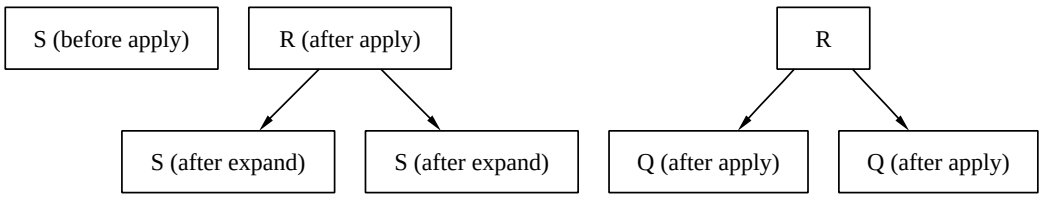

Figure 3.8: Top-down tree transducer operation.

$S[c] \to c$. The transduction of the tree $f[c]$ is illustrated in Figure 3.8. First there exists an object that represents the state S. After `apply()` is invoked with $f[c]$ as argument the rule R is chosen to replace S. When `expand()` is invoked on R two new objects representing the two states S in the right-hand side are created. The `apply()` method is invoked on these objects one after another with the tree c as argument, causing them to be replaced by objects representing the rule Q.

3.5 Stepwise Operations

Both regular tree grammars and top-down tree transducers offer the possibility to review how trees are created from rules step-by-step. In regular tree grammars these operations are only available in enumeration mode. The stepwise operations are single step, parallel step and back. They are handled in a very similar way in regular tree grammars and top-down tree transducers. Initially every rule object in stepwise mode is hidden. As single or parallel step operations are performed rules lose their hidden status, and as back operations are performed rules regain their hidden status. The output tree differs a little between regular tree grammars and top-down tree transducers. A hidden rule in a regular tree grammar shows its parent nonterminal instead of the actual tree that the rule represents. A hidden rule in a top-down tree transducer shows its parent state with the input tree as child instead of the actual tree.

The internal classes handle the stepwise operations with the four methods `singleStep()`, `parallelStep()`, `singleBack()` and `parallelBack()`. Every rule keeps track of whether it is hidden or not and returns a tree that represents this when asked. The compiled regular tree grammar or top-down tree transducer keeps track of at what depth in the tree changes should be made (and this is passed as argument to the operations). If no change could be made at the given depth a failure code is returned and the compiled class will try to perform the step at a greater depth (up to the depth of the tree). For the back operation to work correctly, the compiled class also keeps track of which single and parallel operations have been made (and uses `singleBack()` or `parallelBack()` accordingly). All the operations work recursively until they have gone deeper than the given depth. If a hidden rule is found within `singleStep()` or `parallelStep()` before the given depth has been exceeded, the rule is marked as unhidden. The `singleStep()` method terminates its search as soon as one rule is marked unhidden, while the `parallelStep()` method tries to mark one rule as unhidden in all branches within the given depth. The `singleBack()` and `parallelBack()` methods do the opposite, marking unhidden rules as hidden. In order for `parallelBack()` to not change back rules that actually were single stepped, all rules also have a flag whether they were unhidden by `singleStep()` or `parallelStep()`. These measures ensure that the user can mix single and parallel steps as desired, still getting back to exactly the previous states by means of back commands.

Chapter 4

Results

The results chapter discusses how the compiler and its results actually change the performance of Treebag and how further improvements might be made. Finally there is a summary that discusses some less formal aspects of this thesis.

4.1 Performance

Here it is shown how the performance of compiled regular tree grammars and top-down tree transducers have changed compared to their uncompiled counterparts. The refine and advance operations within regular tree grammars have improved significantly, but unfortunately the compiled top-down tree transducers give no performance increase.

4.1.1 Regular Tree Grammars

The most significant improvement with the compiled generators is the advance operation within regular tree grammars. Since it incorporates some algorithmic differences compared to the uncompiled advance operation it

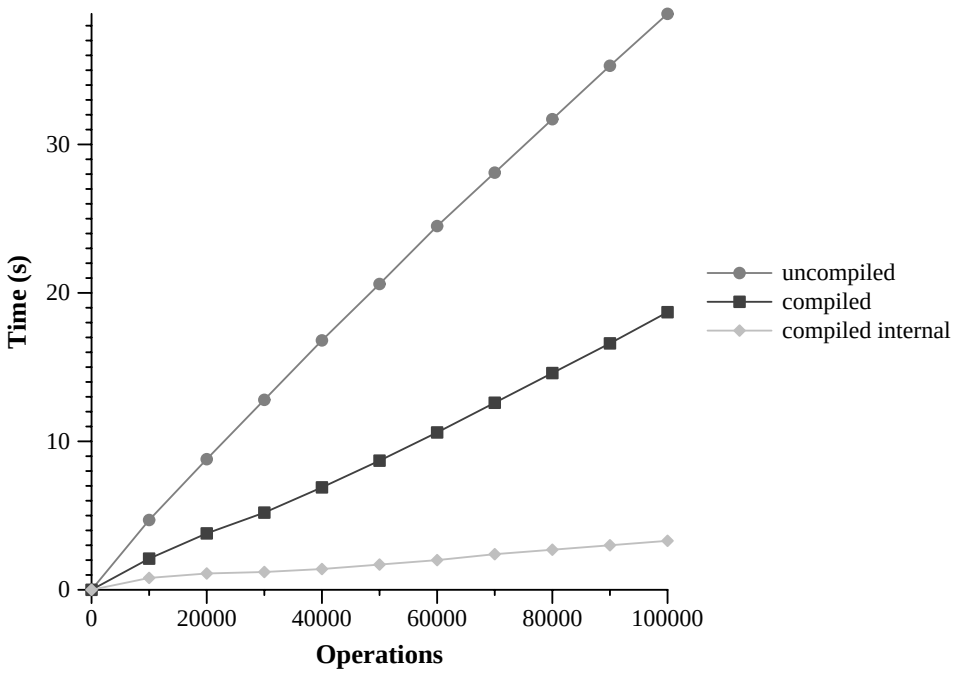

Figure 4.1: Performance of the advance operation.

achieves much better performance for some grammars (to the extent of being usable where the uncompiled advance operation is too slow for practical use). In Figure 4.1 the difference between a compiled and an uncompiled advance operation can be seen. A total of 100 000 advance operations were executed on a fairly simple grammar and the total time spent was measured every 10 000 operations. The topmost line in the graph labelled uncompiled shows the time spent for the original implementation and both the lower lines are measured using the compiled regular tree grammar. The lower line shows the time spent executing the advance operations and creating the trees that are internally used by the compiled classes. The middle line also includes the time spent to convert these internal trees to the actual output trees. As can be seen, the conversion of the internal trees is expensive. The grammar used in the test was $\{A \to f[A, B], A \to c, B \to f[B, A], B \to d\}$ where f, c and d are terminal symbols.

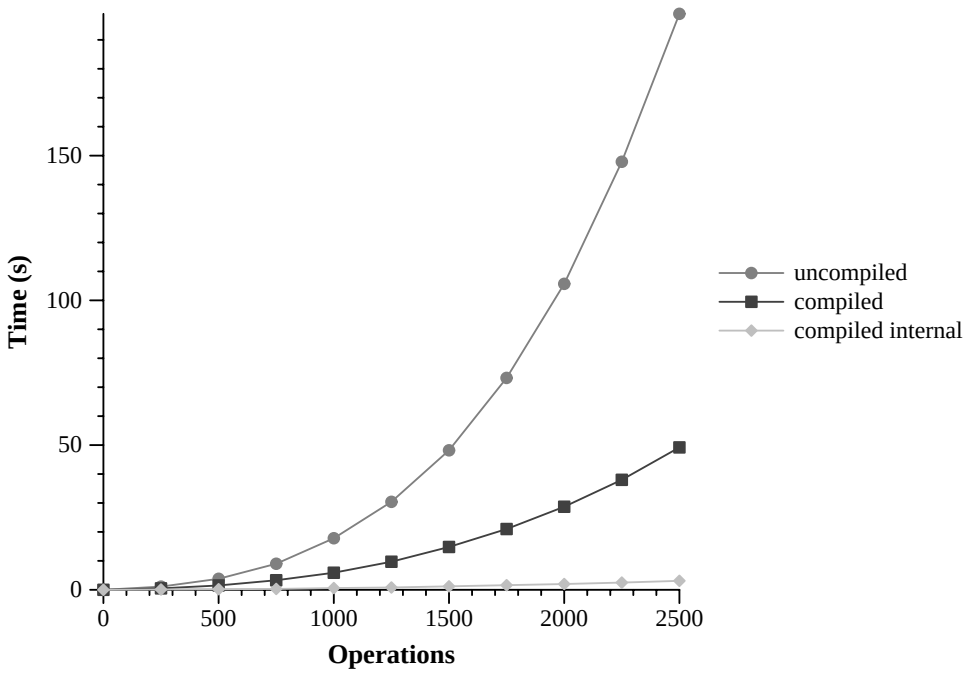

Figure 4.2: Performance of the refine operation.

The performance of the refine operation can be seen in Figure 4.2. A deterministic grammar, $\{A \to f[A, B], B \to c\}$ with f and c as terminal symbols, was used since the test gets entirely random otherwise. A total of 2 500 refine operations were executed and the total time spent was measured every 250 operations. As in the graph showing the performance of the advance operation, there are three lines. One line showing the time spent for an uncompiled grammar and two lines showing the time spent for a compiled grammar with and without the conversion of the internal trees. The lines are superlinear since the tree that refine is applied to grows larger for every operation.

The graphs may give the impression that the performance increase is greater for the refine operation than the advance operation. For simple grammars this can often be the case, but as grammars grow in complexity the performance advantage of the compiled advance operation also becomes larger. Replacing a nonterminal with a random rule does not get that much more

expensive due to having many rules to choose from. Having many rules to choose from when advancing a tree can on the other hand become very expensive. Since a compiled regular tree grammar already makes some decisions at compile time it has greater potential to be improved from compilation.

A simple example grammar where a compiled advance operation performs much better is $\{A \to f[A, A], A \to B, B \to a\}$. Figure 4.3 shows the difference between a compiled and uncompiled advance operation for this grammar. Time spent for the uncompiled operation increases a lot every time the next depth level is to be searched. The time needed for the uncompiled operation to complete 700 operations and above is not shown in the figure. Testing was canceled after 30 minutes and it is unknown how long it would take for the uncompiled operation to finish (thus, the compiled operation is far superior on this grammar). The example grammar shows this behavior when quite many advance operations have been made, grammars that are more complex can behave like this already after a few advances (or even when searching for the first tree in the ordering).

4.1.2 Top-Down Tree Transducers

Results from the top-down tree transducer compiler are unfortunately not as nice as the results from the regular tree grammar compiler. The performance of a compiled top-down tree transducer is generally worse than that of an uncompiled one. In Figure 4.4 the time spent on a total of 1 000 transductions can be seen. Input trees are generated from the deterministic regular tree grammar used in the performance test for refine. The top-down tree transducer used has the rules $\{S[c] \to d, S[A] \to d, S[B] \to d, S[f[x_1, x_2]] \to g[S[x_1], S[x_2]]\}$. With this specific transducer the compiled version is faster at creating an internal tree than the uncompiled one, but even this is not always the case. In fact, the compiled version tends to perform worse as the complexity of the transducer increases.

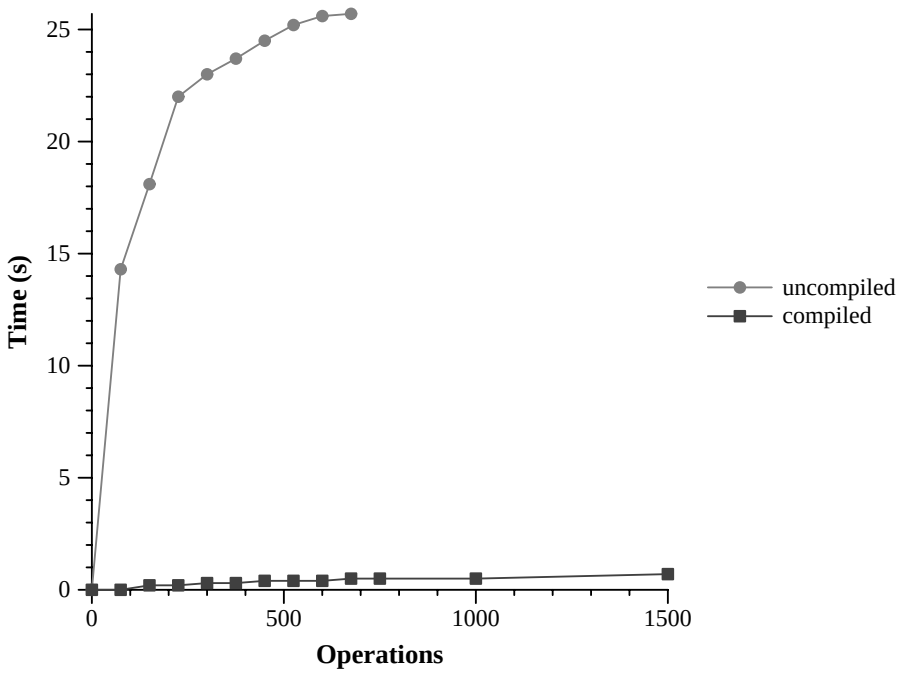

Figure 4.3: Nasty example for advance.

Large amounts of time within a compiled transducer is spent on creating new objects for every state and rule that is processed. Whenever a state is used or a rule is chosen a new object for that state or rule is created. This serves the purpose of creating the internal tree, but with top-down tree transducers it would probably be better to not bother with the creation of an internal tree. Instead classes for states and rules could be used statically, acting as tree transformers rather than building blocks for an internal tree. This will be described in more detail in the next section.

4.2 Future Improvements

This section discusses how the work in this thesis could be improved. Some of the suggested improvements are also applicable to the uncompiled operations in Treebag.

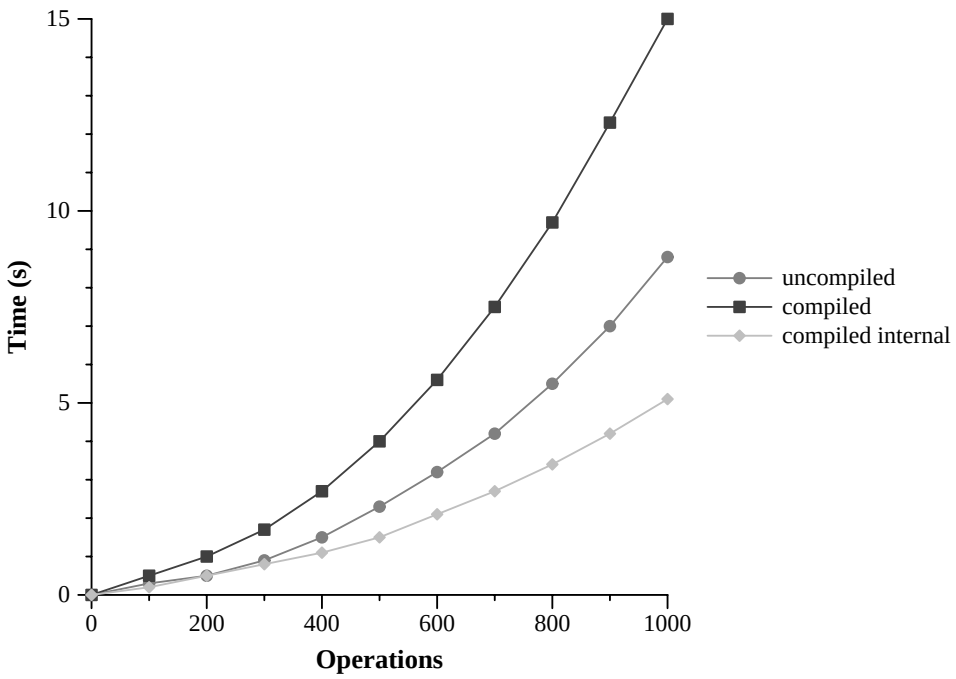

Figure 4.4: Top-down tree transducer performance.

4.2.1 Tree Conversion

Conversion from the internal trees to the actual output trees is expensive. Currently the entire output tree is created from scratch whenever asked for, even if only one node was changed in the internal tree. If every internal node always kept track of when changes were needed instead of creating entirely new trees all the time it would become a lot faster. This does however introduce the problem that code outside the compiler might modify the returned output tree and cause faulty behavior due to this. Thus, the tree might need to be cloned before it is returned and the performance gain would be lost. However, currently no TREEBAG class does actually modify trees obtained from a tree grammar without making copies. Hence, if all future additions to TREEBAG are required to adhere to this style of defensive programming, no problem would arise.

4.2.2 Unnecessary Rules

Both regular tree grammars and top-down tree transducers would benefit from an algorithm that removes rules that never produces terminal trees (and also any nonterminal or state for which this is true). The refine operation in regular tree grammars needs to keep such rules since it returns both trees that are terminal and nonterminal. The only removal that takes place in the compiler currently is of nonterminals or states that are never reached. Result caching counters this problem to some extent. It would also be desirable if warnings where issued when unnecessary rules are removed since the existence of such rules usually indicates a mistake by the user.

4.2.3 Regular Tree Grammars

The Refine Operation

A lot of time in the refine operation is spent searching for the nonterminal nodes. For every refine operation the entire tree is searched starting from the root node. This could be improved a lot by keeping a list of the nonterminal nodes and changing them directly.

4.2.4 Top-Down Tree Transducers

Implementation Approach

The current approach to improving top-down tree transducers worked poorly. One of the main problems is that a compiled top-down tree transducer creates a new class for every state and production that is used. Another approach that should work a lot better would be to use static classes that transform the tree without creating a costly internal tree. The stepwise operations would however be more troublesome to realize with this approach. Most likely they

would have to be taken care of by really transducing the input tree step-by-step (and not by step-by-step showing different parts of the already computed output tree). This is due to the fact that information on how the tree was transduced would be unavailable. This problem is particularly difficult to handle if the top-down tree transducer in question is non-deterministic.

Improvement of Functionality

Non-deterministic partial top-down tree transducers would benefit from an algorithm that ensures a successful transduction whenever there are one or more valid output trees. Currently both the compiled and uncompiled implementations may fail when they randomly pick a rule that further down will encounter a state that has no valid rules for the input tree. This could be solved by creating a list of possible successful states for every node in the input tree before starting the actual transduction. It would have to be done starting from the leaves of the tree and working upwards.

Since it would reduce performance to check for possible states prior to a transduction, a compiled top-down tree transducer should ideally also determine when it would be needed. Deterministic or total transducers should of course not bother with it, but it would also be beneficial to examine if a non-deterministic transducer has states and rules that may cause this problem (since it is not necessarily so).

Using Determinism

Deterministic top-down tree transducers that have the same state with the same variable in more than one place could use sharing in order to compute that transduction only once. Care must however be taken so as this does not cause problems due to several branches in the tree actually being the same. If some code modifies the tree it might be modified in more than one place, which probably is undesirable. However, as mentioned before, this would not

be a problem with the current version of TREEBAG. In fact, a certain class of tree grammars in TREEBAG, namely `pdtGrammar`, already uses sharing in order to gain efficiency.

4.2.5 Stepwise

The stepwise mode of uncompiled regular tree grammars in TREEBAG was improved during the work on this thesis. This change would be interesting for the compiled classes as well, since it is a simpler and cleaner approach. It would not increase the performance of the stepwise operations, but it would make the code nicer. Every internal node would have an integer number associated to it that initially is zero. A single step operation goes through nodes one by one in breadth-first order until a node is encountered which contains an integer number of zero. All nodes on the way are increased by one, including the node with the number zero. A parallel step operation does the same but works on all nodes at the same depth at once. A back operation decreases the number of all nodes by one unless they are already zero. Nodes whose associated number is zero correspond to the hidden ones in the current compiled implementation.

4.3 Summary

The summary reviews the work done in a less formal manner. How the improvements are perceived when actually using TREEBAG and where run-time compilation might be useful is discussed.

4.3.1 Perceived Improvement

When actually using TREEBAG, the improvement that compiled regular tree grammars introduces is mostly experienced with grammars that are a bit

more complex (or with badly written grammars). The outstanding difference is that advance is usable on grammars where the uncompiled operation takes almost infinite time. Top-down tree transducers are not meaningful to compile since they only provide performance gains in rare cases (and in those cases it is not really a difference that matters).

4.3.2 Usefulness of Run-Time Compilation

It is the writers opinion that run-time compilation only is desirable in rare cases. In the case of Treebag it would be more interesting to improve the performance through preprocessing of grammars and algorithmic improvements or optimizations. The parts of the compiled regular tree grammars that make the largest difference could be introduced into uncompiled operations as well, or handled by algorithmic improvements. Examples would be inserting result caching into the uncompiled advance operation and using a list of nonterminals in refine to remove the time spent in tree traversal. A compiled operation would of course always have the potential of performing better, but the work involved is only worthwhile in applications where performance is very important. Some preprocessing and optimization might be cleaner to implement and more efficient with compiled operations, but the difference is not likely to be all that large.

Another aspect in the case of Treebag concerning run-time compilation is where time will be spent if the grammars and transducers perform really well. Without improvements to other parts, such as the drawing and displaying, the performance gain might not be that large during actual usage. As always, it is most relevant to improve parts which use a lot of computing resources. Run-time compilation is however not useless in a system such as Treebag, but other ways to improve efficiency should be explored first.

For most applications, run-time compilation is probably not the best way to improve performance. It should be one of the last options to explore, and

then only if performance is a critical issue. If it was simple to provide a good estimation of how much that can be gained in general from run-time compilation it would have been an interesting addition to this thesis. A specific area where run-time compilation might be particularly interesting would be interpreters for scripting languages or virtual machines such as the one used by Java.

Also worth noting is that the run-time compilation added to TREEBAG uses the Java compiler to actually produce byte code (which in turn might be translated to native machine code during execution by some Java virtual machines). Adding run-time compilation to other languages such as C or C++ would include a lot more work and also not be usable across different architectures (though there exists at least one cross-platform library to achieve this, GNU lightning).

4.3.3 Final Thoughts

Although one might question the actual usefulness of adding run-time compilation to TREEBAG, it has been very interesting to implement. Among the aspects that made the compiled regular tree grammars work better than the compiled top-down tree transducers is the fact that they were the focus early on. They received more thought and redesigning to properly handle new ideas. As it is in general, it is sometimes needed to rewrite larger parts of existing code from scratch to introduce significant improvements that are discovered during the implementation. Top-down tree transducers were implemented without any major rewrites (and also in much shorter time), and more thought should have been given to whether using exactly the same approach as with regular tree grammars really was fitting. The implementation time spent for top-down tree transducers should however be less than the time required for regular tree grammars since they are less complicated. Regular tree grammars become fairly complex due to their advance operation.

The most tedious and troublesome part of this thesis has most certainly been writing the report. Finding the motivation and inspiration to write about the work done and improve already written parts has not been easy. Without all the helpful advice and suggestions from my tutor it would have been even harder. Both the implementation and report writing have been a good learning experience in completing a larger project by myself.

Bibliography

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[Dre03] Frank Drewes. Picture generation, a tree-based approach. Unpublished manuscript, 2003.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer, 1998.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 1, pages 1–68. Springer, 1997.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[NM00] Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled Java executions. *Concurrency: Practice and Experience*, 12(6):343–362, 2000.

[Rou69] William C. Rounds. Context-free grammars on trees. In *Proc. 1st Annual ACM Symposium on Theory of Computing (STOC)*, pages 143–148, 1969.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4:257–287, 1970.