

Umeå University
Department of Computing Science
Master's Thesis
June 12, 2006

Branching Tree Grammars in TREEBAG

Author:
Gabriel Jonsson

Supervisor:
Frank Drewes

Examiner:
Per Lindström

Abstract

In this master thesis project a class of tree grammars called “Branching tree grammar” has been implemented into the TREEBAG system. A tree grammar is a device that generates a language of mathematical trees, according to rules specified by the user. These trees can then be transformed into for example pictures, by other components in TREEBAG. The branching tree grammar, which is a class of grammars known from literature, can generate more advanced languages than the grammars previously implemented in TREEBAG.

The implementation uses an approach of analyzing the branching tree grammar, and replacing it with a series of simpler components, namely regular tree grammars and top-down tree transducers. These components, when used together in a sequence transforming each others’ output, will generate exactly the same language as the specified branching tree grammar.

Regular tree grammars and top-down tree transducers were already implemented in TREEBAG. Therefore the core implementation of branching tree grammar consists of an algorithm that reads a file that specifies a branching tree grammar, translates it into the equivalent series of simpler components, and outputs files describing regular tree grammars and top-down tree transducers. These files are then processed by the existing TREEBAG-components.

As a basis for the translation an algorithm has been used, which was taken from a constructive proof, in which it was proved that for each branching tree grammar, there exists a sequence of a regular tree grammar and top-down tree transducers, which generates the same language.

However, in order to get an implementation that obeys feasible time requirements, and which is also fairly easy to use, the given algorithm to obtain the regular tree grammar and transducers had to be heavily reconstructed. This report describes at a relatively formal level the given translation algorithm as well as several improved versions of the algorithm. The advantages of the different algorithms are discussed, and some proofs and proof sketches for the correctness of the improved algorithms are also included.

Contents

1	Introduction	1
1.1	TREEBAG, tree grammars and algebras	1
1.2	Properties of the branching tree grammar	2
1.3	What algorithm to use, and what modifications are necessary . .	3
1.4	Why using the approach of translating branching tree grammar into other components?	3
1.5	Organization of the remaining chapters	4
2	Concepts and definitions	5
2.1	Signatures and trees	5
2.1.1	Definitions	5
2.1.2	Examples	6
2.1.3	More notation for trees	6
2.2	Collage algebras	6
2.2.1	Informal definition	7
2.2.2	Example	7
2.3	Regular tree grammars	9
2.3.1	Definition	9
2.3.2	Informal definition of derivations	9
2.3.3	Example derivations and pictures	9
2.4	ET0L tree grammars	12
2.4.1	Definition	12
2.4.2	Informal definition of derivations	12
2.4.3	Example derivations and pictures	12
2.4.4	ET0L tree grammars compared with regular tree grammars	15
2.5	Branching tree grammar	15
2.5.1	Definition of the simple branching tree grammar	15
2.5.2	Definition of the branching tree grammar	16
2.5.3	Informal definition of derivations for simple branching tree grammar	16
2.5.4	Informal definition of derivations for branching tree gram- mar	17
2.5.5	Example derivations and pictures	17
2.6	Top-down tree transducers	20
2.6.1	Definition	20
2.6.2	Example	20
2.7	Is the class of branching tree grammars needed?	21
3	The translation of branching tree grammars	23
3.1	The theorem of equality of branching tree grammars and regular tree grammars with td transducers	24
3.2	Notation	24
3.3	Original translation	25
3.3.1	Definitions	25
3.3.2	Example illustrating original translation	26
3.4	Plain translation	30
3.4.1	Definitions	30
3.4.2	Proof of correctness	30

3.5	Plain translation with terminating leaves	31
3.5.1	Definitions	31
3.5.2	Proof of correctness	32
3.6	Normal translation	34
3.6.1	Definitions	34
3.6.2	Proof sketch of correctness	35
3.6.3	Comparison with the original translation using an example	35
3.7	Extended translation	38
3.7.1	Definitions	39
3.7.2	An example where extended translation is useful	40
3.8	Turning a branching tree grammar into a simple branching tree grammar	48
3.9	Restricting top table sequences by regular expressions	49
3.10	Additions in order to allow nonterminal derivations	50
3.11	Obtaining synchronization strings for a derivation	50
3.11.1	An example of a translated grammar rewritten to show synchronization information	51
4	The implementation	55
4.1	Translation	55
4.2	Interaction with the branching tree grammar	55
4.3	Table enumeration	56
4.3.1	Results only	56
4.3.2	Derive stepwise	57
4.4	Random tables	57
4.5	Choose new subtables/rules	57
4.6	Hide/Show sync info	57
5	Results and conclusions	59
5.1	Some hints regarding efficiency for users of the implementation . .	59
5.2	Remaining problems and possible improvements	60
5.2.1	Extended translation for regular tree grammars	60
5.2.2	Undefined and repeated results	60
5.2.3	Enumerating every generated tree once and only once . .	61
5.2.4	Avoiding unwanted recomputation by the tree transducers	61
5.2.5	Regular expressions regulating subtables	62
5.3	Acknowledgement	63
	References	65

1 Introduction

This master thesis is a study within the area of tree based picture generation. A picture generator in this area usually consists of two parts: a tree generator and an algebra. The tree generator, which can be one of the standard tree grammars from formal language theory, generates a set of trees, called a tree language. The algebra then transforms trees into pictures, typically by interpreting leaves as certain pictures and internal nodes as graphical transformations and/or pictures.

TREEBAG is a framework in which a user can, among other things, create their own tree grammars and algebras to build picture generators. The subsection TREEBAG below gives an more detailed introduction to what TREEBAG is and how it works.

The objective of this master thesis is, briefly stated, to implement a new class of tree grammar into the TREEBAG system. The tree grammar in question is the branching synchronization tree grammar with nested tables, which is abbreviated as branching tree grammar.

The master thesis also involves some theoretical work. The implementation is to be based on a certain algorithm presented in [DE04], and this algorithm requires some modifications before it can be used in the implementation, in a reasonable manner. See Section 1.3 for details regarding this.

The background to the project is a book [Dre06] written by my master thesis supervisor Frank Drewes. This book introduces, at a formal level, many tree based picture generators, some of them using the branching tree grammar. All of the major generators presented in the book are implemented in TREEBAG, and the implementation of the branching tree grammar has been part of this master thesis project.

Section 1.2 will describe what kind of pictures the branching tree grammar is particularly good for, and compare its power with the grammars already existing in TREEBAG. Section 1.3 and 1.4 summarize further the objectives of the master thesis, and important choices made during the work. Section 1.5 explains the organization of the remaining chapters of this report.

1.1 TREEBAG, tree grammars and algebras

TREEBAG is a system written in Java, in which a user can create and combine different kinds of components to assemble, among other things, picture generators. Typically the user uses one of the predefined classes of grammars and algebras, in which case they write the rules in a text-file. One can also add new classes to TREEBAG, as has been done in this thesis, in which case Java code implementing the class is written.

Four major kinds of components are used:

- tree grammars, which create trees, often triggered by a user through the GUI of TREEBAG,
- tree transformations that take trees as input and output other trees,

- algebras that take trees as input and return pictures, which can be sent to a display, and
- displays that draw pictures on the screen.

The trees used here are rooted trees, which means that one particular node in the tree is specified as root. A tree also has the property that each node in the tree has a symbol assigned to it, which is taken from a finite set specified in the grammar. This symbol is what the typical algebra uses to select the picture or transformation to apply. For a more formal definition of trees, grammars and algebras, see Chapter 2.

For some examples of generated pictures, see Chapter 2. Most examples of pictures in this report are pictures resembling trees and plants, but this is far from everything such grammars are suitable for. Things like tiled patterns and mathematical objects like fractals are other types of pictures these devices are particularly appropriate for.

TREEBAG can be downloaded from <http://www.cs.umu.se/~drewes/treebag>, see also <http://www.cs.umu.se/~drewes/picgen>.

1.2 Properties of the branching tree grammar

The first thing one might ask is what the branching tree grammar is useful for. What does it do that makes it worthwhile to implement?

Different kinds of tree grammars can create different kinds of languages. A simple grammar like a regular tree grammar is easier to use than the more general ones, but on the other hand it is more limited in what languages it can generate.

Branching tree grammars are particularly noted for their ability to generate trees (and thus, with the help of algebras, pictures) with certain symmetries. For example, a tiled pattern resembling a carpet, with horizontal and vertical symmetry may be simple to create with a branching tree grammar, very complex to create with a ETOL-grammar and even impossible to create with a regular tree grammar. See the different sections of Chapter 2 for formal definitions of these classes of grammars, and Section 2.7 for further discussion comparing their generative power.

One way to increase the generative power, without creating new types of grammars, is to combine existing components. For example, one can combine a regular tree grammar with a top-down tree transducer, which is a kind of tree transformation. In this case the input of the transducer is the output from the regular tree grammar, and the output of the transducer is seen as the output of the composed component. The result is a tree grammar which can generate a more complex language than the regular tree grammar alone.

The languages that can be generated with a regular tree grammar, in composition with n top-down tree transducers, are actually the same as those that can be generated with branching tree grammars of table depth n . The concept of table depth will be defined along with the rest of the branching tree grammar in Section 2.5. The equality mentioned is a result from [DE04] and is stated

formally in Section 3.1. This equality is the background to the approach used in this thesis, as is explained in the next subsection.

Although the implementation of branching tree grammars does not add any generative power to TREEBAG, it is still worthwhile, since for most users it is likely to be a much simpler task to specify a branching tree grammar, than to specify the tree grammar and transducers otherwise needed.

1.3 What algorithm to use, and what modifications are necessary

The implementation of the branching tree grammar should be based on an algorithm described in [DE04] which transforms a branching tree grammar into a regular tree grammar and a series of top-down tree transducers, which when used in composition generate a language equal to that of the branching tree grammar. Regular tree grammars and top-down tree transducers are components that already exists in TREEBAG, and thus can be used directly in the implementation.

An objective of the master thesis is also to improve this given algorithm, in such a way that it transforms the branching tree grammar to more efficient and easier to use components, which still produce the same tree language. The improvement of the algorithm has been a major part of this master thesis.

The problem with using the given algorithm is that although the correct language will be generated, the resulting components will work way too slow, and many derivations will get stuck and not generate any tree. The latter is irrelevant in theory but unacceptable in practice. Another problem is that the same output tree might appear many times when enumerating all possibly derivations.

More specifically the problem lies in the trees that are sent between the different components in the composition emulating the branching tree grammar. These trees will often be very large, thought most of the branches will be ignored in later transformations.

Thus, for improving the algorithm, one has to identify when unnecessary branches are created, and modify the components so that such branches are replaced with leaves immediately.

1.4 Why using the approach of translating branching tree grammar into other components?

An alternative way of implementing the branching tree grammar in TREEBAG would be to ignore the transformation to other components altogether, and implement the branching tree grammar from scratch. This could be done using procedures directly derived from the formal definition of the derived language of a branching tree grammar. This approach has not been followed in this master thesis, for several reasons. The most important ones are listed below.

- Using the transformation approach does not only add the branching tree grammar component to TREEBAG. The feature of transforming a branching tree grammar to other components, which may be of interest for the more theoretically minded user, comes as a spin-off. As the components are actually generated as valid TREEBAG components the user may study them individually.
- The predefined TREEBAG components regular tree grammar and top-down tree transducer have gone through much testing, and optimization for speed and avoidance of derivations ending in undefined results, see for example [Vik04]. Thus, if the conversion of the branching tree grammar into practically feasible components is successful, the implementation automatically can benefit from the optimization already done to the regular tree grammar and top-down tree transducers.

The possible disadvantage of using the transformation approach is that it may perhaps never become as efficient as a direct implementation. Maybe there are some branching tree grammars which can be used to derive trees in polynomial time, with respect to tree size, while no equivalent composition of tree transducers can do it in less than exponential time. During the work on this master thesis, it has come to appear quite likely that this is not the case, and thus the method of transforming branching tree grammars to other components is to prefer.

The definition of branching tree grammars in [DE04] is slightly less general than the variant in [Dre06]. Still, both definitions are equivalent in the sense that a grammar of one kind can always be simulated by a grammar of the other kind, possibly with a larger table depth. Since the more general version is to be implemented, the algorithm given in [DE04] also has to be modified to handle this version. See the beginning of Chapter 3 for a more detailed explanation.

1.5 Organization of the remaining chapters

In the second chapter of this report, definitions are given for the different components, together with illustrating examples. These definitions are needed later in the report, in particular for Chapter 3 which describes formally the improvements done to the given algorithm. Chapter 3 also includes proofs and proof sketches showing that the improvements done are only improvements to efficiency and usability, and do not change the generated tree language.

Chapter 4 describes the implementation at a rather high level. It mainly describes how the user GUI commands are handled and how the interaction between the branching tree grammar component and the regular tree grammar and tree transducer components is done. This information is perhaps most useful for someone who wants to modify the implementation, but may be useful also for someone using it.

Chapter 5 summarize the results and discusses the efficiency of the implemented system. It also includes a list of limitations and possible improvements together with some discussion and suggestions of how these fixes and improvements could be made.

2 Concepts and definitions

This section contains a number of formal definitions that will be needed in later chapters, especially in Chapter 3, which formalizes the modified construction used in the implementation. Along with all definitions some illustrating examples are given. Except when specified otherwise, all definitions are taken from [Dre06].

Many things from the definitions in [Dre06], that are not necessary for any other formalism later in this report, are left out in the definitions here. In particular the definitions of derived languages, which are somewhat lengthy, are in this report replaced with informal explanations and examples.

In addition supplemental references will be given for some of the defined objects, some of them being texts available on the Internet, and some of them being older texts about theory which is the foundation of the theory mentioned here.

2.1 Signatures and trees

In TREEBAG, trees are the basic objects generated by grammars and interpreted by algebras. Signatures are the alphabets specifying the symbols the trees may consist of.

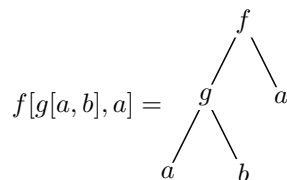
2.1.1 Definitions

A signature is a set Σ of ranked symbols. A ranked symbol is denoted $f : n$, f being its name and n its rank, with $n \in \mathbb{N}$. The rank is intuitively the exact number of subtrees that are allowed under a node of the corresponding name in a tree. Note that the same name f is allowed in more than one symbol of a signature, with different ranks. When there is no risk of confusion $f : n$ is denoted with just f . We let $\Sigma^{(n)}$ denote the subset of Σ consisting of all symbols with rank n .

Given a signature, we can build trees over it. The set T_Σ is intuitively the set of all trees that can be built using the signature Σ . Formally, T_Σ is defined as the smallest set such that $f \in \Sigma^{(n)}$ and $t_1, \dots, t_n \in T_\Sigma$ implies $f[t_1, \dots, t_n] \in T_\Sigma$.

The notation $f[t_1, \dots, t_n]$ means that the root of the tree has the name f , and its subtrees are the trees t_1, \dots, t_n . The symbols '[', ']', and ',' are reserved for this purpose, and will never occur in signatures. The tree $f[]$ is also denoted f .

Trees are also often drawn in a more tree-like way for illustrational purposes. An example is:



2.1.2 Examples

Here, a small signature is defined which will be used again in the following sections to exemplify grammars and algebras.

Let $\Sigma_{ex} = \{stem : 3, leaf : 0\}$.

Then $\Sigma_{ex}^{(0)} = \{leaf : 0\}$, and $\Sigma_{ex}^{(3)} = \{stem : 3\}$, and $\Sigma_{ex}^{(n)} = \emptyset$, for $n \in \mathbb{N} \setminus \{0, 3\}$.

$T_{\Sigma_{ex}} = \{leaf, stem[leaf, leaf, leaf], stem[stem[leaf, leaf, leaf], leaf, leaf], \dots\}$ is the infinite set of trees over Σ_{ex} .

2.1.3 More notation for trees

Let Σ be a signature. If R is a set of trees, $\Sigma(R)$ denotes the set of trees $f[t_1, \dots, t_n]$ such that $f : n \in \Sigma$ and $t_1, \dots, t_n \in R$.

The set $T_{\Sigma}(R)$ is intuitively the set of trees over Σ that have had some of its subtrees replaced by trees in R . Formally $T_{\Sigma}(R)$ is the smallest set of trees such that $R \subseteq T_{\Sigma}(R)$ and, for every symbol $f : n \in \Sigma$ and all trees $t_1, \dots, t_n \in T_{\Sigma}(R)$, the tree $f[t_1, \dots, t_n]$ is also in $T_{\Sigma}(R)$. In particular $T_{\Sigma}(\emptyset) = T_{\Sigma}$.

A set $L \subseteq T_{\Sigma}$ is called a tree language. A tree generator is any device g defining a tree language $L(g)$. $L(g)$ is said to be the language generated by g . Examples of tree generators are regular tree grammars, ETOL tree grammars and branching tree grammars, which are defined in Sections 2.3 to 2.5.

A notation useful for defining how the tree grammars work is substitution, which is defined as follows.

Let $X_n = \{x_1 : 0, \dots, x_n : 0\}$ be a signature whose symbols we call variables. The names x_1, \dots, x_n will from now on be reserved for this purpose only, and will not appear in ordinary signatures.

Let t, s_1, \dots, s_n be trees. Then $t[[s_1, \dots, s_n]]$ denotes the tree t' obtained by the simultaneous substitution of s_i for every occurrence of x_i in t . As an inductive definition,

$$t' = \begin{cases} s_i & \text{if } t = x_i \in X_n \\ f[t_1[[s_1, \dots, s_n]], \dots, t_k[[s_1, \dots, s_n]]] & \text{if } t = f[t_1, \dots, t_k] \notin X_n \end{cases}$$

As an example: if $t = f[g[x_2, a], x_1]$ then $t[[f[x_2], g[b, b]]] = f[g[g[b, b], a], f[x_2]]$.

2.2 Collage algebras

To be able to show how tree generators can be used to generate pictures, we now discuss collage grammars. A collage algebra takes a tree as input and returns a picture, build pretty much in the same manner as a collage known from traditional art. Each symbol in the tree is interpreted as a graphical object or a transformation on such an object, and the collage of all these sub-pictures makes the output picture.

A tree grammar combined with a collage algebra, makes a collage grammar. If the tree grammar is a regular tree grammar (defined in the next section) the collage grammar is called a context-free collage grammar. Originally, the name context-free collage grammar was introduced in [HK91], and this definition did not involve tree grammars. However, this version of the collage grammar has been shown equivalent to the one based on regular tree grammars [Dre01].

2.2.1 Informal definition

Intuitively a collage algebra is a device that takes as input a tree and returns a subset of some Euclidean space, usually \mathbb{R}^2 . This is done by assigning a picture (which may be empty) to each of the symbols in the input signature. This picture is simply a subset of the Euclidean space, in practice usually circles, rectangles and other polygons. In addition, each symbol of rank n is assigned n geometric transformations. All used transformations are affine transformations, that is those that preserve straight lines. Examples are rotation, scaling and translation.


For each node, the algebra will add the corresponding picture to the output set, after applying to it a chain of transformations taken from the path from the root to the current node in the tree. More precisely, suppose a node labeled $f \in \Sigma^{(n)}$ has subtrees t_1, \dots, t_n and that the algebra assigns the picture C and transformations τ_1, \dots, τ_n to f . Then the output pictures for the root nodes of t_1, \dots, t_n are transformed by τ_1, \dots, τ_n respectively.


The union of the resulting collages, together with C , forms the output picture for the node labeled f .

The definition can be extended to color collage algebras, by adding color attributes to each picture and adding a new kind of transformation that transforms colors. A formal definition of collage algebras can be found in section 3.1 of [Dre06] for the uncolored case, and section 7.1 in [Dre06] for the colored case. For a formal definition of algebras in general see [Dre06] page 21.

2.2.2 Example

Let the input signature be Σ_{ex} from the previous subsection. We define a corresponding collage grammar as follows:

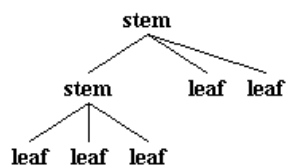
leaf is interpreted as the picture  .

stem is interpreted as the picture  plus the following triple of transformations:

- scale to 60%, rotate 30 degrees counterclockwise, translate 15 points – half the height of the stem – upward,
- scale to 80%, rotate 5 degrees clockwise, translate 30 points upward and
- scale to 60%, rotate 30 degrees clockwise, translate 20 points upward.

There are three transformations because *stem* has rank 3. Recall that this means it always has three subtrees. The first transformation in the triple is applied to the collage resulting from the evaluation of the first subtree. The collage obtained from the second and third subtrees are affected by the second and third transformations, respectively.

For example the following tree:



will be interpreted as the picture in Figure 1, which consists of two pictures of the stem and five of the leaf. The leaf to the right has been scaled down, moved upwards 20 points, and then rotated 30 degrees clockwise. Similarly the whole subtree on the left, consisting of a stem and leaves, has been scaled down, moved upwards 15 points and rotated counterclockwise. The leaves in this subtree have been transformed by compositions of two scalings, two rotations and two translations.

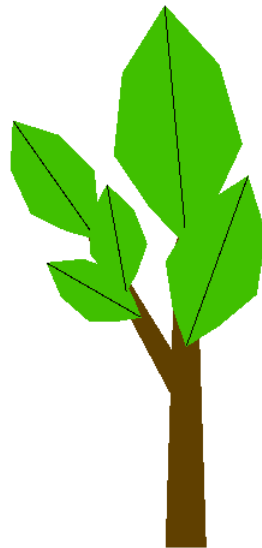


Figure 1: A picture obtained by interpreting a tree using a suitable algebra.

2.3 Regular tree grammars

One of the most simple devices to create trees over some signature, is the regular tree grammar. For the full definition see page 30 in [Dre06] or Chapter 2 in [CDG⁺97].

2.3.1 Definition

A regular tree grammar is a system $g = (N, \Sigma, R, S)$ consisting of:

- a finite signature N of symbols with rank 0, called nonterminals,
- a finite output signature Σ , disjoint with N , of symbols called terminals,
- a finite set of rules of the form $A \rightarrow t$ where $A \in N$ and $t \in T_{\Sigma}(N)$ and
- an initial nonterminal $S \in N$.

2.3.2 Informal definition of derivations

Let $g = (N, \Sigma, R, S)$ be a regular tree grammar as in the definition.

The principle by which it generates trees is the following:

The derivation starts with the initial nonterminal S , which is replaced with the right-hand side t of some rule $S \rightarrow t$ in R . The tree t may contain nonterminals as leaves. One of these nonterminals is then replaced, using again some rule of the grammar. It does not matter in which order the nonterminals are replaced.

If there are several rules with the same left hand side, the regular tree grammar is nondeterministic. Then one of these rules is chosen during this derivation step. The process goes on until no nonterminal remains. The resulting tree is then one in the language generated by g .

2.3.3 Example derivations and pictures

The regular tree grammar $REG_{ex} = (\{S\}, \Sigma_{ex}, \{S \rightarrow stem[S, S, S], S \rightarrow leaf\}, S)$ generates the whole language T_{Σ} . An example of a derivation is shown in Figure 2.

Using the collage algebra from Section 2.2.2 on this tree, results in the picture in Figure 3. One of the larger trees – too large to print in the tree-like form with the *stem* and *leaf* nodes – will be interpreted as the picture in Figure 4.

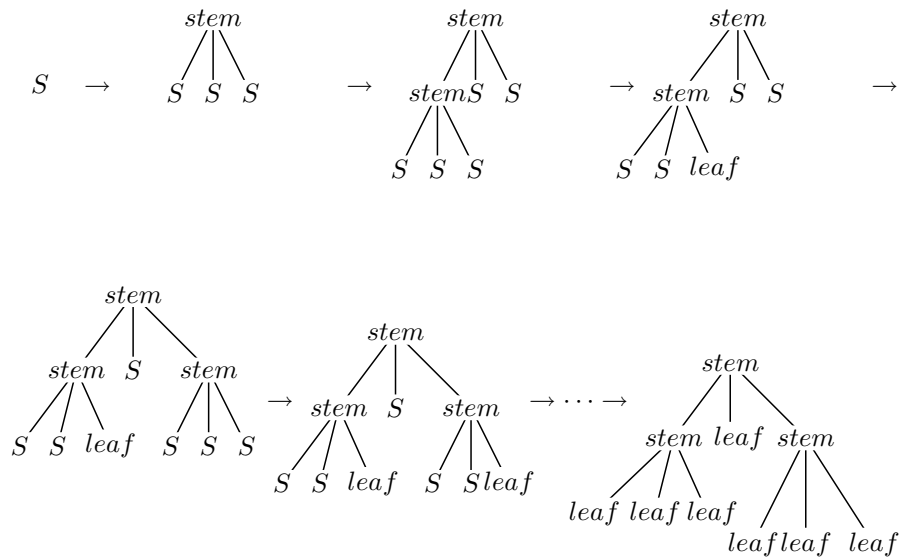


Figure 2: A derivation by the regular tree grammar in the example.

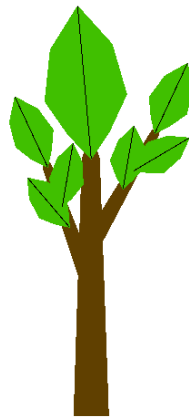


Figure 3: The picture obtained by interpreting the tree derived in Figure 2.

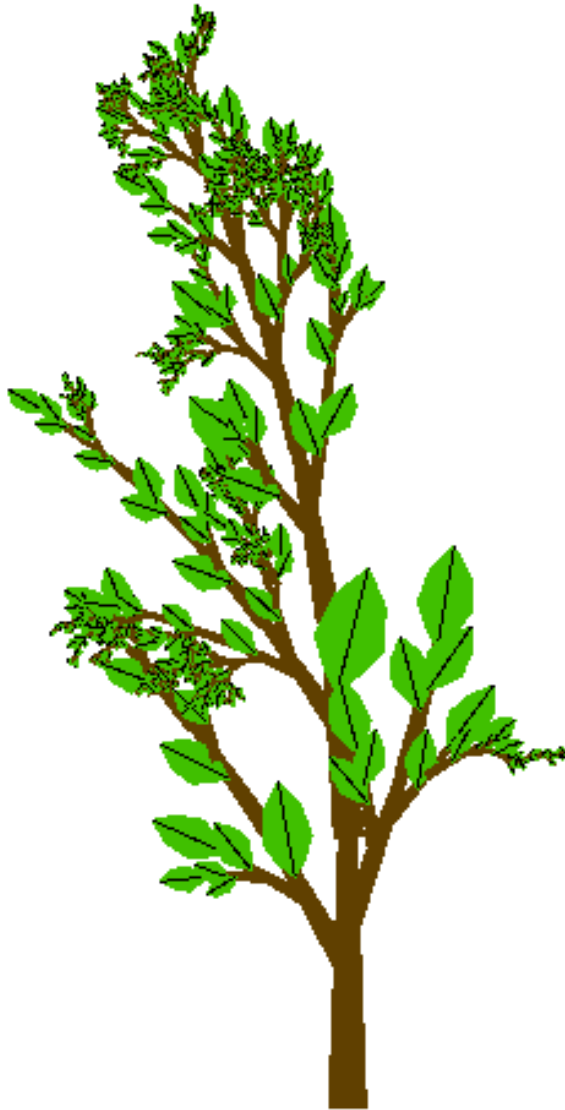


Figure 4: The picture obtained by interpreting a very large tree from the regular tree grammar.

2.4 ETOL tree grammars

The ETOL tree grammar (whose full definition can be found in page 74 in [Dre06], page 12 in [Dre04] or in [Eng76]) is a generalization of the regular tree grammar. In the regular tree grammar each nonterminal was derived completely independent of any other nonterminal. This makes it impossible to use that device to create for example the language $L \subset T_{\Sigma_{ex}}$, where all trees are balanced.

The concept of ETOL tree grammars originates from the Lindenmayer Systems [Lin68]. These systems are a large class of grammars that generates strings by repeatedly and in parallel replacing symbols according to the grammar rules. The ETOL grammar is one of those systems, and the ETOL tree grammar is a special case of the ETOL grammar, whose output language is a set of trees.

2.4.1 Definition

An ETOL tree grammar is a system $g = (N, \Sigma, R, t_0)$ consisting of:

- a finite nonterminal signature N of symbols with rank 0, called nonterminals,
- a finite output signature Σ (which may or may not be disjoint with N), of symbols called terminals,
- a finite set R of tables R_1, \dots, R_k for some integer k . Each table is a set of rules of the same form as for the regular tree grammar, and
- an initial tree $t_0 \in T_{\Sigma}(N)$, called the axiom.

2.4.2 Informal definition of derivations

The principle by which the trees are generated is the following:

The derivation starts with the axiom t_0 . If t_0 contains only symbols of the output signature then t_0 is a tree in the generated language of g , denoted $L(g)$. If t_0 contains nonterminals, one derivation step is carried out by *simultaneously* replacing all nonterminals, say A_1, \dots, A_i with trees t_1, \dots, t_i such that the rules $A_1 \rightarrow t_1, \dots, A_i \rightarrow t_i$ exist in *the same* table. Let t' be the tree one gets by this substitution. If t' contains only symbols of the output signature then $t' \in L(g)$. The process is repeated using t' as input for the next derivation step.

In other words $L(g)$ is the set of trees over the output signature that are possible to reach, by zero or more derivation steps starting out from the axiom t_0 , with the restriction that, in every derivation step, all nonterminals are replaced using rules from the same table.

2.4.3 Example derivations and pictures

Let $ETOL_{ex} = (\{S\}, \Sigma_{ex}, R, S)$ be the ETOL tree grammar with $R = \{table_1, table_2\}$, $table_1 = \{S \rightarrow stem[S, S, S]\}$, and $table_2 = \{S \rightarrow leaf\}$. This grammar generates the subset of T_{Σ} in which all leaves are at the same distance from the

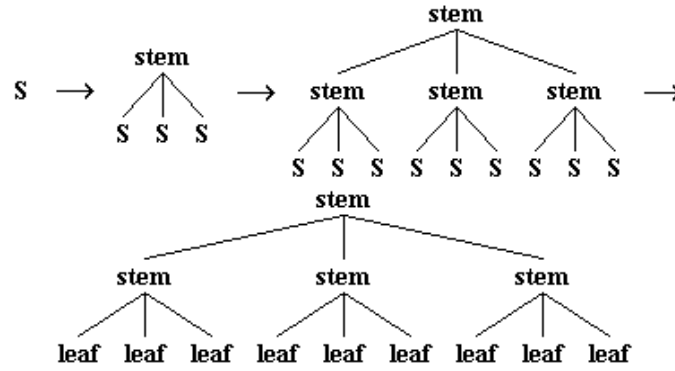


Figure 5: A derivation by the ETOL tree grammar in the example.

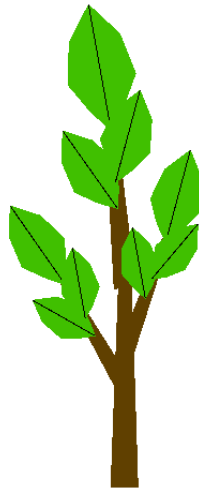


Figure 6: The picture obtained by interpreting the tree derived in Figure 5.

root. An example of a derivation is shown in Figure 5. Using the collage algebra from Section 2.2.2 to interpret this tree, yields the picture shown in Figure 6. An example of a larger tree generated by this language yields the one in Figure 7.

For an example where nonterminals and terminals are not disjoint, consider the following ETOL tree grammar, which generates exactly the same language. $ETOL_{ex2} = (\{leaf\}, \Sigma_{ex}, R, leaf)$, with $R = \{\{leaf \rightarrow stem[leaf, leaf, leaf]\}\}$.

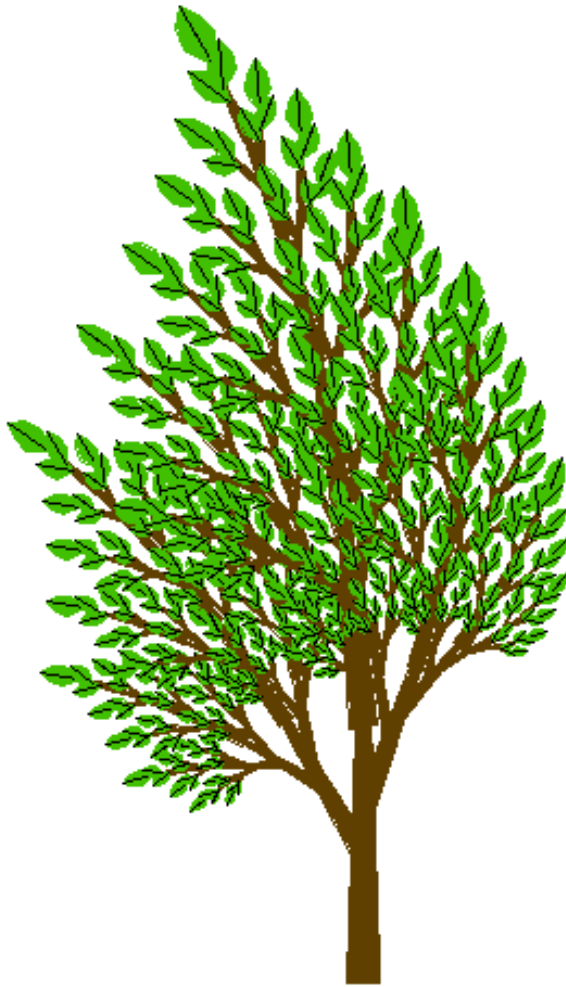


Figure 7: The picture obtained by interpreting a relatively large tree from the ETOL tree grammar.

2.4.4 ETOL tree grammars compared with regular tree grammars

It can be proved, by using a pumping lemma for regular tree grammars, that the language generated by $ETOL_{ex}$ in the previous subsection, cannot be generated by a regular tree grammar. See page 48 in [Dre06] for a statement of that pumping lemma.

This fact can also be argued for intuitively. If a regular tree grammar is used, all parts of the tree are built independently, so the only way to keep track of how long the tree should grow to be balanced, would be to let different nonterminals specify different distances from the root. With finitely many nonterminals it is, therefore, not possible to specify the infinite language of all balanced trees.

2.5 Branching tree grammar

The branching tree grammar is the kind of grammar that is to be implemented in this master thesis project. Branching tree grammars are a generalization of ETOL tree grammars. Suppose that we want to create the same types of tree pictures as before, but now with the requirement that the picture should be symmetric about the vertical line going through the root. This would be hard or even impossible to accomplish with ETOL. See further comments on this in Section 2.7.

This report will give definitions for two variants of branching tree grammars. The first, called “simple branching tree grammar”, uses the definitions in [DE04]. The second, called “branching tree grammar”, is a more general variant which is the one that should be implemented. In Chapter 3, we will see how a branching tree grammar can be rewritten to a simple branching tree grammar that generates the same language. This is necessary since the implementation is based on an algorithm from [DE04] that works for the simple branching tree grammar.

For the full definition of the branching tree grammar see page 150 and 323 in [Dre06].

2.5.1 Definition of the simple branching tree grammar

A simple branching tree grammar is a system $g = (N, \Sigma, I, J, R, t_0)$ consisting of:

- a finite signature N of symbols with rank 0, called nonterminals,
- a finite output signature Σ , disjoint with N , of symbols called terminals,
- an alphabet I of symbols called synchronization symbols,
- an alphabet J of symbols called table symbols,
- a mapping R which assigns to each $\tau \in J^n$, a set of rules $A \rightarrow t$, with $A \in N$ and $t \in T_\Sigma(N \times I^n)$, and
- an initial nonterminal $t_0 \in N$, called the axiom.

The number n is the nesting depth, or depth, of g , the sets $R(\tau)$ for $\tau \in J^n$ are the tables of g , and $SN_g = N \times (I^n)^*$ is the set of synchronized nonterminals. A string in $(I^n)^*$ will be called a synchronization string.

The intuition behind the definition of R is to create a hierarchical structure of tables. The tables at the level n are the sets $R(\tau)$ for $\tau \in J^n$. A table $R(\tau')$ at level $i < n$ is given by $\tau' \in J^i$. Intuitively τ' is the name or address of the table within the hierarchy. The table $R(\tau')$ is the union of all its subtables $\tau'.j$ where $j \in J$. Here $\tau'.j$ denotes the addition of the component j to τ' , i.e., $\tau'.j = (j_1, \dots, j_i, j)$ if $\tau' = (j_1, \dots, j_i)$. The table at level 0, which is also called the top table in the following, is the table $R() = \bigcup_{\tau \in J^n} R(\tau)$.

In order to make the derivations more readable, an alternative notation is used for synchronized nonterminals, which is shown by the following example. Let $I = \{1, 2, 3, 4\}$, $n = 2$ and $A \in N$. Suppose that during the derivation there is a synchronized nonterminal $\sigma = (A, S)$ with $S = (1, 2)(3, 4)(1, 2)$ (a string consisting of three pairs of symbols). Then σ is also denoted $A \langle \begin{smallmatrix} 1 & 3 \\ 2 & 4 \\ 1 & 2 \end{smallmatrix} \rangle$. The number of rows in the matrix of symbols always equals n , but the number of columns can be any number including zero.

2.5.2 Definition of the branching tree grammar

A branching tree grammar is a system $g = (N, \Sigma, I, J, R, t_0)$, whose components are the same as in the simple branching tree grammar, except for the following:

- The output signature Σ need not be disjoint with the nonterminal signature N , and
- the axiom is now a tree $t_0 \in T_\Sigma(N)$, rather than a single nonterminal.

The first generalization makes sure that both versions of branching tree grammars studied in [Dre06] are covered. The second has been made for convenience.

2.5.3 Informal definition of derivations for simple branching tree grammar

For the simple branching tree grammar, the principle by which the trees are generated is the following:

The derivation starts with the synchronized nonterminal (t_0, λ) consisting of the axiom t_0 and the empty string (it is in the set $N \times (I^n)^0$ and has no synchronization information yet). A derivation step is carried out by *simultaneously* replacing all synchronized nonterminals, call them $(A_1, S_1), \dots, (A_i, S_i)$, with trees t_1, \dots, t_i such that the rules $A_1 \rightarrow t_1, \dots, A_i \rightarrow t_i$ exist in some tables, with the following limitation: The more rows S_i and S_j have in common, the more restricted is the choice of tables for A_i and A_j . More precisely, S_i and S_j will be members of the set $(I^n)^k$ which means they consist of k n -tuples of synchronization symbols. If the first l symbols, $l \leq n$, in every one of the k tuples are equal for S_i and S_j , then if A_i is to use table τ_i and A_j is to use table τ_j , the first l table symbols of τ_i and τ_j must agree.

Finally for each substitution $A_i \rightarrow t_i$, each synchronized nonterminal (N, S) in t_i is replaced by $(N, S_i.S)$ where $.$ denotes the concatenation of lists of n -tuples. This means that the synchronized nonterminals build up longer and longer series of synchronization information along the derivation.

Let t' be the tree one gets by this substitution. If t' contains only symbols of the output signature then $t' \in L(g)$. The process is repeated using t' as input for the next derivation step.

2.5.4 Informal definition of derivations for branching tree grammar

For the branching tree grammar, the principle by which the trees are generated is similar to the one for the simple branching tree grammar, except for the following:

The derivation starts with the tree obtained from the axiom by replacing every nonterminal A with $A(\langle \rangle)$. Let t' be the initial tree, or a tree obtained by using derivation steps as defined for the simple branching tree grammar case. Replace all synchronized nonterminals (A, S) with A in t' , yielding t'' . If this t'' contains only symbols of the output signature then $t'' \in L(g)$. Naturally the derivation then continues with t' rather than t'' .

2.5.5 Example derivations and pictures

Let $BST_{ex} = (\{S\}, \Sigma_{ex}, \{0, 1\}, \{gen, term, one, two, three\}, R, S)$ be a branching tree grammar with

$$R(gen.one) = \{S \rightarrow stem[S\langle 0 \rangle, S\langle 1 \rangle, S\langle 0 \rangle]\},$$

$$R(gen.two) = \{S \rightarrow S\langle 0 \rangle\},$$

$$R(gen.three) = \{S \rightarrow leaf\},$$

$$R(term.one) = \{S \rightarrow leaf\} \text{ and the remaining of } R(i.j) \text{ are empty.}$$

This grammar generates the subset of trees generated by REG_{ex} that satisfies the following condition: On every *stem*-node in the output tree, the left and the right of the three subtrees must be exact copies.

The addition of $R(gen.two)$ makes it possible for different paths from root to leaf to have different lengths. With only $R(gen.one)$ and $R(term.one)$ the language had been equivalent to the language generated by $ET0L_{ex}$.

The table $R(gen.three)$ does not add any power to the grammar, as its effects can also be obtained by combining $R(gen.two)$ and $R(term.one)$. However the table is convenient to have available. In particular the tree derived in the next example would have required a somewhat more lengthy derivation, if this table was not present.

An example of a derivation is:

$$S\langle \rangle \xrightarrow{(gen.one)} stem[S\langle 0 \rangle, S\langle 1 \rangle, S\langle 0 \rangle] \xrightarrow{(gen.one)}$$

```

stem[
  stem[S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>],
  stem[S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>],
  stem[S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>]
]
→
(gen.two) and (gen.three)
stem[
  stem[leaf, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, leaf]],
  stem[S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, leaf, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>],
  stem[leaf, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, leaf]
]
→
(gen.one)
stem[
  stem[leaf, stem[S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>], leaf]],
  stem[stem[S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>], leaf, stem[S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>, S<sup>0</sup><sub>1</sub><sup>0</sup></sub>]],
  stem[leaf, stem[S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>, S<sup>0</sup><sub>0</sub><sup>0</sup></sub>], leaf]
]
→
(term, one)
stem[
  stem[leaf, stem[leaf, leaf, leaf], leaf]],
  stem[stem[leaf, leaf, leaf], leaf, stem[leaf, leaf, leaf]],
  stem[leaf, stem[leaf, leaf, leaf], leaf]
]

```

The last tree is drawn in a tree-like form, in Figure 8. Using the collage algebra from Section 2.2.2 to interpret this tree, yields the picture shown in Figure 9.

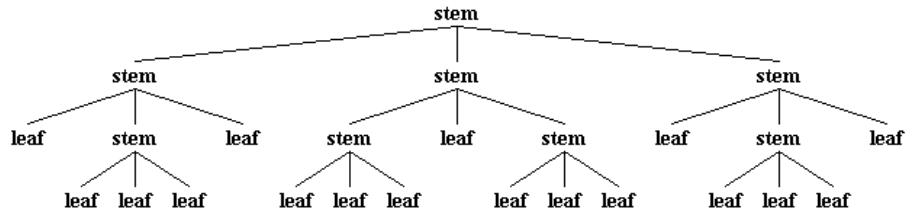


Figure 8: A tree derived by a branching tree grammar.

An example of a larger tree generated by this language yields the picture in Figure 10.

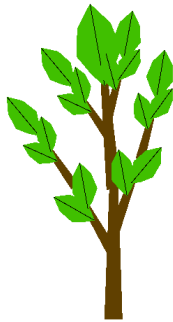


Figure 9: The picture obtained by interpreting the tree derived in Figure 8.

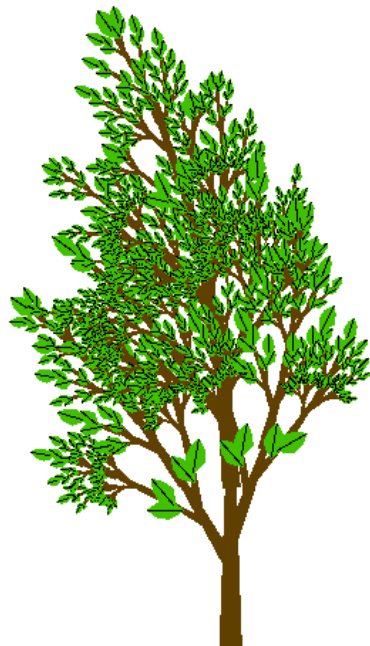


Figure 10: The picture obtained by interpreting a relatively large tree from the branching tree grammar.

2.6 Top-down tree transducers

A top-down tree transducer is a device that takes trees as input and transforms them into other trees. In the context of this thesis, the device is important since some of the more complicated tree grammars can be transformed into a series of a regular tree grammar and top-down tree transducers, that generates the same language. For the full definition of the top-down tree transducer see page 58 in [Dre06] or page 182, in Chapter 6 in [CDG⁺97].

2.6.1 Definition

A top-down tree transducer, also called td transducer, is a system $td = (\Sigma, \Sigma', \Gamma, R, \gamma_0)$ consisting of:

- a finite input signature Σ ,
- a finite output signature Σ' ,
- a finite signature Γ of symbols of rank 1, called states, such that Γ is disjoint with Σ and Σ' ,
- a finite set of rules R of the form $\gamma[f[x_1, \dots, x_k]] \rightarrow t[\gamma_1[x_{i_1}], \dots, \gamma_l[x_{i_l}]]$ where $\gamma, \gamma_1, \dots, \gamma_l \in \Gamma$, $f^{(k)} \in \Sigma$ and $t \in T_{\Sigma'}(X_l)$ with $i_1, \dots, i_l \in \{1, \dots, k\}$, and
- an initial state $\gamma_0 \in \Gamma$.

A derivation starts with a tree $\gamma_0[t_{in}]$ where $t_{in} \in T_{\Sigma}$. Rules from R are used repeatedly to make substitutions to this tree, until there are no more states left. If the final tree t_{out} is a tree in $T_{\Sigma'}$, ie only contains symbols of the output signature, we say that td can derive t_{out} from t_{in} .

For an input tree $t_{in} \in T_{\Sigma}$, the output of the td transducer td is a set $td(t_{in}) = \{t_{out} : t_{out} \text{ can be derived from } t_{in} \text{ by } R\}$.

2.6.2 Example

Here a regular tree grammar REG and a td transducer td will be presented, that when applied in a series generate exactly the same language as $ETOL_{ex}$ in Section 2.4.3. The terminals of the regular tree grammar are named $table_1$ and $table_2$, to indicate the interpretation that the regular tree grammar decides a series of table choices that the ETOL tree grammar would have used, and the td transducer simulates picking rules from these tables.

$$REG = (\{S\}, \{table_1 : 1, table_2 : 0\}, \{S \rightarrow table_1[S], S \rightarrow table_2\}, S)$$

$$td = (\{table_1, table_2\}, \{stem : 3, leaf : 0\}, \{\gamma\}, R, \gamma)$$

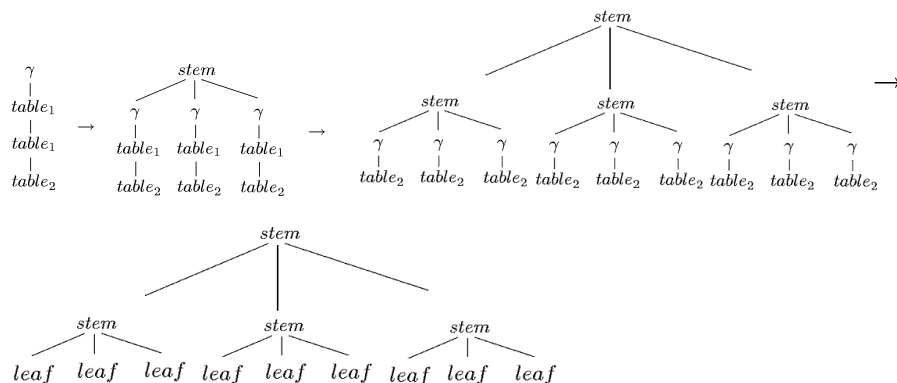
$$\text{with } R = \{ \gamma[table_1[x_1]] \rightarrow stem[\gamma[x_1], \gamma[x_1], \gamma[x_1]] , \gamma[table_2] \rightarrow leaf \} .$$

An example of a derivation is shown below.

First REG generates a monadic tree in the following derivation:

$$S \rightarrow table_1[S] \rightarrow table_1[table_1[S]] \rightarrow table_1[table_1[S]] \rightarrow table_1[table_1[table_2]] .$$

Then the transducer uses that tree as its input as below:



This is the same tree that was produced by $ETOL_{ex}$ in the example in Section 2.4.3. Note how the input tree to the tree transducer exactly corresponds to the tables used in that example. Using the collage algebra from Section 2.2.2 on this tree, results in the same picture as in Figure 6.

2.7 Is the class of branching tree grammars needed?

Could it be the case that the $ETOL$ tree grammar is really sufficient for picture generators of practical interest, and thus the branching tree grammar a quite uninteresting addition?

For $ETOL$ tree grammars there is no known suitable pumping lemma (as there is for regular tree grammars), which could be used to prove that a language cannot be generated by an $ETOL$ tree grammar [Dre06]. Thus it is hard to formally prove that a specific language generated by branching tree grammars could not be generated by $ETOL$ tree grammars as well. But there are many languages that can easily be specified with a branching tree grammar but seem at least very complicated to generate with $ETOL$ tree grammars [Dre06]. Examples (not directly related to picture generation) of languages which are provably not contained in the class of $ETOL$ tree languages can be found in [Eng82], [DE04].

Then when are branching tree grammars needed or preferred? Recall that $ETOL$ tree grammars are equally powerful as a regular tree grammar with monadic output signature and a top-down tree transducer, while branching tree grammars are equally powerful as a regular tree grammar and a series top-down tree transducers.

It is known from tree theory [Bak79] that two consecutive top-down tree transducers sometimes can be replaced with one single top-down tree transducer without changing the output. This can be done if both of the following two conditions are true:

- The first transducer is deterministic or the second one is linear.
- The first transducer is total or the second one is nondeleting.

The first part is of particular interest. It leads to the conclusion that if a picture language generated by a branching tree grammar uses synchronization to create

copies of nondeterministically (here nondeterminism can intuitively be thought of as randomness) generated parts, it is likely that the branching tree grammar cannot be replaced with an equivalent ETOL tree grammar.

3 The translation of branching tree grammars

This chapter formalizes the modified construction that is used in the implementation. The basis for the construction is a theorem from [DE04] which states that the languages one can generate with a branching tree grammar of depth n , are exactly those that one can generate composing a regular tree grammar and n top-down tree transducers. This theorem is stated in Section 3.1.

The proof of this theorem, also from [DE04], is a constructive proof describing how the regular tree grammar and td transducers can be constructed, and proving its correctness. This construction has been extracted from the proof and is presented in Section 3.3, where it is called “original translation”.

During the work on this master thesis, four alternative constructions have been studied. These four constructions are a chain of improvements, where each one is defined in principle by its difference from of the previous construction. These are in order called “plain translation”, “plain translation with terminating leaves”, “normal translation” and “extended translation”. Thus, for example, “plain translation” is the least sophisticated modification of “original translation”. The four translations, together with proof sketches for correctness, and illustrating examples are presented in Sections 3.4 to 3.7.

Section 3.8 describes how an arbitrary branching tree grammar is turned into a simple branching tree grammar generating the same language. This is necessary since the construction for translating a branching tree grammar to other components is designed for grammars of that kind.

Throughout this chapter, we consider an arbitrary but fixed branching tree grammar $BG = (N_{BG}, T_{BG}, I, J, R_{BG}, S_{BG})$, that is to be translated to a chain consisting of a regular tree grammar and top-down tree transducers. Let n be the depth of BG (ie the tables are all $R_{BG}(\tau)$ for $\tau \in J^n$), and let d be the number of elements in I . The letters d and n will be reserved for this purpose for this entire chapter.

Without loss of generality, we assume that $I = \{0, 1, 2, \dots, d - 1\}$. If BG is not a simple branching tree grammar, then it is replaced with a simple branching tree grammar generating the same language, as described in Section 3.8. After that one of the five translations from Sections 3.3 to 3.7 is used to turn it into a regular tree grammar and a sequence of td transducers. Then some additions are made to the transducers, as described in Sections 3.10 and 3.11, which do not change to output language, but gives the components functionality that can be used to obtain information about the derivation of an output tree.

There also exists a construction, in which the user is allowed to give a regular expression specifying a table sequence for the top tables. This construction is described in Section 3.9, where it is also argued that this does not add any generative power to the class of branching tree grammars.

In the case where BG is of depth 0, it is by definition a regular tree grammar, except for the synchronization information that is not used for anything. In this case the regular tree grammar is obtained by simply removing the corresponding components from the grammar.

3.1 The theorem of equality of branching tree grammars and regular tree grammars with td transducers

In [DE04] the correspondence between branching tree grammar on the one hand and regular tree grammars and top-down tree transducers on the other is studied. The main result is the following:

Theorem 1

Let BST be an arbitrary simple branching tree grammar of depth i . Then there exists a regular tree grammar REG and i td transducers td_1, td_2, \dots, td_i such that $L(BST) = td_i(\dots(td_2(td_1(L(REG))))\dots)$. Here REG and td_1, \dots, td_i can effectively be constructed from BST . In addition, this can be done in such a way that the td transducers td_1, td_2, \dots, td_i are total.

Theorem 2

Let REG be an arbitrary regular tree grammar, and let td_1, td_2, \dots, td_i be i arbitrary td transducers. Then there exists a branching tree grammar BST of depth i , such that $L(BST) = td_i(\dots(td_2(td_1(L(REG))))\dots)$.

3.2 Notation

In this section some notation and abbreviations are defined, that will be used for the rest of the chapter. The following terms are defined as stated:

The regular tree grammar - The regular tree grammar first in the chain of components simulating the branching tree grammar.

Final transducer - The last top-down tree transducer in the chain.

Intermediate transducer - Any top-down tree transducer in the chain except the final transducer.

Synchronization tree - Any output tree of the regular tree grammar or an intermediate transducer is a synchronization tree. The synchronization trees generated by the i :th component in the chain (where $i \leq n$) are trees over the signature $\Sigma_i = \{(\tau : d^i) \mid \tau \in J^i\} \cup \{\perp : 0\}$. Informally that means that Σ_i contains all table names for tables on depth i and the bottom symbol. Intuitively every path from the root to some node in a synchronization tree is a table sequence that some nonterminal will follow in the derivation (either all the way to the leaf, or just a part if it on the way uses a rule whose right hand side are only terminals). A more formal definition of what a synchronization tree is can be found in [DE04].

Pseudo leaf - Any node in a synchronization tree whose subtrees are only bottom symbols \perp is a pseudo leaf. (That means that if this node is used in a derivation the derivation for that nonterminal has to stop there. If this derivation step produces any nonterminal this particular derivation results in an undefined tree.)

Terminating rule - A rule with only terminals in its right hand side is said to be terminating.

Terminable table - A table is a terminable table, if it contains at least one terminable subtable, or if it contains rules, at least one terminating rule.

Terminal table - A table is a terminal table, if it contains only terminal subtables, or if it contains rules, only terminating rules.

Let $num_k(i_1, \dots, i_m) = 1 + \sum_{j=1}^m (i_j \cdot k^{m-j})$. That means the tuple (i_1, \dots, i_m) is interpreted as a number written on the base k , and then 1 is added to get a number in $\{1, \dots, k^m\}$ instead of $\{0, \dots, k^m - 1\}$.

As an additional notation, we let, for $j \leq k$, $first_j(i_1, \dots, i_k) = (i_1, \dots, i_j)$.

3.3 Original translation

This is the translation that was defined in [DE04]. The basic idea is that the regular tree grammar makes a series of top table choices, and outputs a tree consisting of top table names. After that the first intermediate transducer makes a series of choices of tables on level 2, restricted by the choices of top tables in the input tree. The second intermediate transducer chooses tables among the tables on level 3 in the branching tree grammar, and so on.

This goes on until the final transducer which takes as input a tree with the level- n tables, and chooses the rules to use from these tables for each step and each nonterminal in the derivation. This will all be specified more clearly in the definition in the next subsection, and further illustrated by an example.

3.3.1 Definitions

The regular tree grammar for this translation, is defined as

$$REG_{orig} = (\{S\}, \Sigma_1, R, S). \text{ Here } R \text{ consists of all rules } S \rightarrow \tau[\underbrace{S, S, \dots, S}_{d \text{ times}}]$$

where $\tau \in J$. R also contains the rule $S \rightarrow \perp$. Thus the grammar generates the whole language T_{Σ_1} .

The i :th intermediate transducer is defined as $TD_{orig}^{(i)} = (\Sigma_i, \Sigma_{i+1}, \{q\}, R, q)$, where R consists of all rules

$$q[\tau[x_1, \dots, x_{d^i}]] \rightarrow \tau.j[\underbrace{q[x_1], \dots, q[x_1]}_{d \text{ times}}, \dots, \underbrace{q[x_{d^i}], \dots, q[x_{d^i}]}_{d \text{ times}}]$$

and $q[\tau[\dots]] \rightarrow \perp$, where $\tau \in J^i$ and $j \in J$. R also contains the rule $q[\perp] \rightarrow \perp$.

The final transducer is defined as $TD_{orig}^{final} = (\Sigma_n, T_{BG}, N_{BG}, R, S_{BG})$,

where R consists of all rules $A[\tau[x_1, \dots, x_{d^n}]] \rightarrow t[[B_1[x_{num_d(\alpha_1)}], \dots, B_l[x_{num_d(\alpha_l)}]]$

such that $\tau \in J^n$ and $A \rightarrow t[[B_1, \alpha_1), \dots, (B_l, \alpha_l)]]$ is a rule in τ .

The formal proof that this construction gives the language $L(BG)$ can be found in [DE04].

3.3.2 Example illustrating original translation

In this section we will work with the example grammar BG_{ex} defined below, borrowed from [DE04]. Informally this grammar describes the language of trees whose leaves are the symbols \triangleleft and \triangleright , and the other nodes are \circ , \bullet and $|$. In addition there are two kinds of synchronization, which is achieved by having a nesting depth of 2 in the grammar. The synchronization on top table level causes the length of any path from root to leaf to be the same in the tree. The synchronization on level two causes the two subtrees below any $|$ to be mirror images of each other.

The table named *gen* as in “generate”, contains rules which are used to keep the derivation going on, making the tree larger. The table named *term* as in “terminate”, contains rules with only output symbols in the right hand side, and thus terminates the derivation.

$$BG_{ex} = (N_{ex}, T_{ex}, I_{ex}, J_{ex}, R_{ex}, S_{ex})$$

$$N_{ex} = \{S, S_r\}$$

$$T_{ex} = \{\circ : 2, \bullet : 1, \triangleleft : 0, \triangleright : 0, | : 1\}$$

$$I_{ex} = \{0, 1\}$$

$$J_{ex} = \{gen, term, one, two, three\}$$

$$S_{ex} = S$$

And R_{ex} is defined by:

$$\begin{aligned} R_{ex}(gen.one) &= \{S \rightarrow \circ[S\langle 0 \rangle], S_r \rightarrow \circ[S_r\langle 0 \rangle]\} \\ R_{ex}(gen.two) &= \{S \rightarrow \bullet[S\langle 0 \rangle, S\langle 1 \rangle], S_r \rightarrow \bullet[S_r\langle 1 \rangle, S_r\langle 0 \rangle]\} \\ R_{ex}(gen.three) &= \{S \rightarrow | [S\langle 0 \rangle, S_r\langle 0 \rangle], S_r \rightarrow | [S\langle 0 \rangle, S_r\langle 0 \rangle]\} \\ R_{ex}(term.one) &= \{S \rightarrow \triangleright, S_r \rightarrow \triangleleft\} \\ R_{ex}(term.two) &= \{S \rightarrow \triangleleft, S_r \rightarrow \triangleright\} \end{aligned}$$

and all other $R_{ex}(i, j) = \emptyset$.

Note that the upper digit is zero in all of the synchronized nonterminals. This causes the synchronization on top level to never be released. Thus in all derivation steps all nonterminals must either use the “generating” rules in the first table or the “terminating” rules in the second table, and this is the reason that all paths from the root to the leaves are of the same length.

An example of a derivation is: $S\langle \rangle \xrightarrow{(gen.two)}$

$$\bullet[S\langle 0 \rangle, S\langle 1 \rangle] \xrightarrow{(gen.one) \text{ and } (gen.three)} \bullet[\circ[S\langle 0 \ 0 \rangle], | [S\langle 1 \ 0 \rangle, S_r\langle 1 \ 0 \rangle]]$$

$$\xrightarrow{(term.one) \text{ and } (term.two)} \bullet[\circ[\triangleleft], | [\triangleright, \triangleleft]].$$

Figure 11 shows the same derivation in tree-like notation. Note that in the last step, the second and third nonterminal had to use the same table, in this case (*term.two*). This was because their sets of synchronization strings were completely identical. The first nonterminal on the other hand had the freedom to choose between (*term.one*) and (*term.two*) since the second row differed. However the first nonterminal was forbidden to use any of the *gen*-tables (unless the other two had done so, too) since the upper rows of digits were identical.

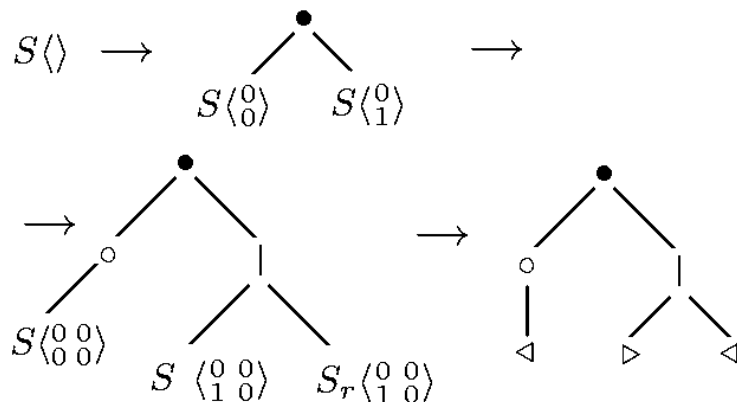


Figure 11: A derivation using the example branching tree grammar.

Let us now inspect how the same output tree could be generated using components retrieved by “original translation”.

Using the algorithm from the previous subsection on BG_{ex} results in the regular tree grammar $REG_{ex,orig}$ and td transducers $TD_{ex,orig}^{(1)}$ and $TD_{ex,orig}^{final}$. These look as follows:

$$REG_{ex,orig} = \{\{S\}, \{gen, term, \perp\}, R_{reg}, S\} \text{ with}$$

$$R_{reg} = \{S \rightarrow gen[S, S], S \rightarrow term[S, S], S \rightarrow \perp\} \text{ and}$$

$$TD_{ex,orig}^{(1)} = \{TD_{input}, TD_{output}, \{q\}, R_{td}, q\} \text{ with}$$

$$TD_{input} = \{gen, term\},$$

$$TD_{output} = \{(gen.one), (gen.two), (gen.three), (term.one), (term.two)\},$$

$$R_{td} = \left\{ \begin{array}{ll} q[gen[x_1, x_2]] & \rightarrow gen.one[q[x_1], q[x_2], q[x_3], q[x_4]], \\ q[gen[x_1, x_2]] & \rightarrow gen.two[q[x_1], q[x_2], q[x_3], q[x_4]], \\ q[gen[x_1, x_2]] & \rightarrow gen.three[q[x_1], q[x_2], q[x_3], q[x_4]], \\ q[term[x_1, x_2]] & \rightarrow term.one[q[x_1], q[x_2], q[x_3], q[x_4]], \\ q[term[x_1, x_2]] & \rightarrow term.two[q[x_1], q[x_2], q[x_3], q[x_4]], \\ q & \rightarrow \perp \end{array} \right\}$$

$$\text{and } TD_{ex,orig}^{final} = \{TD_{final-input}, TD_{final-output}, \{S, S_r\}, R_{final}, S\} \text{ with}$$

$$TD_{final-input} = \{(gen.one), (gen.two), (gen.three), (term.one), (term.two)\},$$

$$TD_{final-output} = \{\circ : 2, \bullet : 1, \triangleleft : 0, \triangleright : 0, | : 0\},$$

$$\begin{aligned}
 R_{final} = \{ & S[gen.one[x_1, x_2, x_3, x_4]] \rightarrow \circ[S[x_1]] , \\
 & S_r[gen.one[x_1, x_2, x_3, x_4]] \rightarrow \circ[S_r[x_1]] , \\
 & S[gen.two[x_1, x_2, x_3, x_4]] \rightarrow \bullet[S[x_1], S[x_2]] , \\
 & S_r[gen.two[x_1, x_2, x_3, x_4]] \rightarrow \bullet[S_r[x_2], S_r[x_1]] , \\
 & S[gen.three[x_1, x_2, x_3, x_4]] \rightarrow | [S[x_1], S_r[x_1]] , \\
 & S_r[gen.three[x_1, x_2, x_3, x_4]] \rightarrow | [S[x_1], S_r[x_1]] , \cdot \\
 & S[term.one[x_1, x_2, x_3, x_4]] \rightarrow \triangleright , \\
 & S_r[term.one[x_1, x_2, x_3, x_4]] \rightarrow \triangleleft , \\
 & S[term.two[x_1, x_2, x_3, x_4]] \rightarrow \triangleleft , \\
 & S_r[term.two[x_1, x_2, x_3, x_4]] \rightarrow \triangleright \\
 & \}.
 \end{aligned}$$

Next we will look at an example of a derivation which produces the same tree as in the example before, but with these components instead of the branching tree grammar.

REG_{ex} generates, in a number of steps, the tree in Figure 12. With this as input tree for $td_{ex,orig}^{(1)}$, one of the possible output trees is shown in Figure 13.

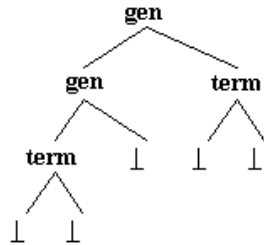


Figure 12: A tree generated by the regular tree grammar first in the chain.

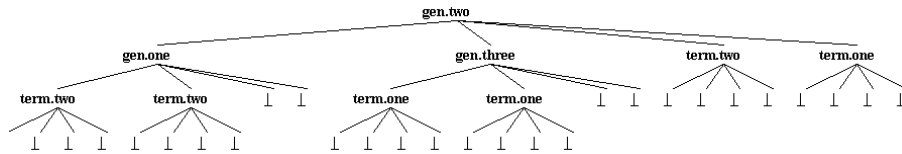


Figure 13: One possible output from the intermediate transducer, when using the tree from Figure 12 as input.

Now it is time for the final transducer $TD_{ex,orig}^{final}$ to transform this tree to the final output tree. This transformation is shown step in Figure 14, in order to give more insight into how a final transducer works. Recall that S and S_r are the states of this final transducer, which means that they exist only during the transformation and not in the output tree. All other symbols involved are input and output symbols for the transducer, which stand for table names and are output symbols of branching tree grammar, respectively.

The role of the regular tree grammar is to make all choices of which top tables to allow, and the first td transducer chooses the allowed subtables.

The input to the final transducer, which is the synchronization tree shown in

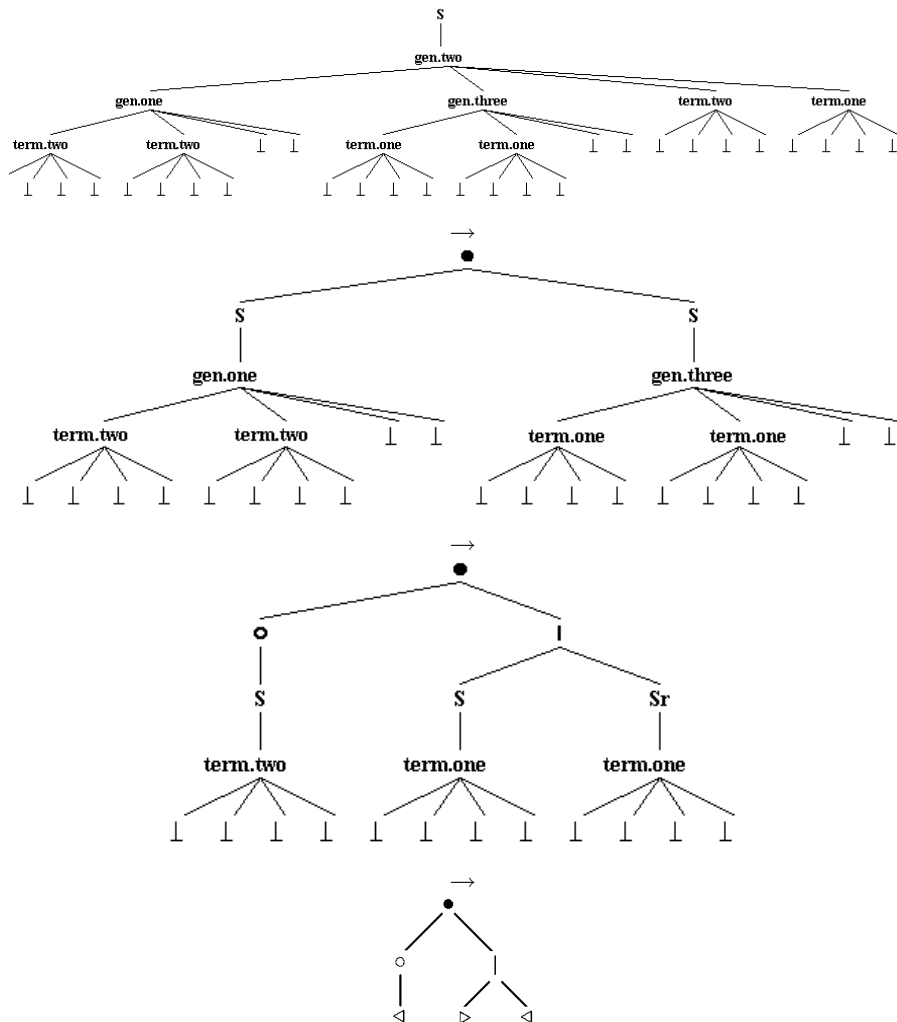


Figure 14: A derivation of the final transducer, when using the tree from Figure 13 as input.

Figure 13, determines which tables should be used, and the only thing left for the final transducer is to pick the rules from the specified tables.

The four steps in Figure 11 correspond directly to the four steps in Figure 14. Compare the pictures to see how the states of the final transducer exactly follow the pattern of the nonterminals of the branching tree grammar.

This example will be revisited and its sources of inefficiency will be discussed in Section 3.6.3, but before this we need to define some of the other translations.

3.4 Plain translation

The translation called “plain translation” is the same as “original translation” except that all rules $q[\tau[\dots]] \rightarrow \perp$ are removed from the intermediate transducers, where τ is any table in J^i . Intuitively these are the rules that cut random branches off the synchronization trees, and they are unnecessary because any application of them will either leave the end result unaffected if the derivations of the remaining transducers end above the cut, or give an undefined result otherwise.

The reason for this rule to be included in the first place, was that it was convenient to have in the proof of the correctness of “original translation”. Having a rule that causes undefined results when used does not matter theoretically, since the output language does not change.

However, in practice, the presence of this rule will make almost all tree derivations end with undefined trees. Therefore we want to remove it.

3.4.1 Definitions

The regular tree grammar REG_{plain} equals REG_{orig} .

The i :th intermediate transducer is defined as

$TD_{plain}^{(i)} = (\Sigma_i, \Sigma_{i+1}, \{q\}, R, q)$, where R consists of all rules

$$q[\tau[x_1, \dots, x_{d^i}]] \rightarrow \tau.j \left[\underbrace{q[x_1], \dots, q[x_1]}_{d \text{ times}}, \dots, \underbrace{q[x_{d^i}], \dots, q[x_{d^i}]}_{d \text{ times}} \right]$$

with $\tau \in J^i$ and $j \in J$. R also contains the rule $q[\perp] \rightarrow \perp$.

The final transducer TD_{plain}^{final} equals TD_{orig}^{final} .

3.4.2 Proof of correctness

Here we will start out from “plain translation” and analyze how the generated language is affected when turning to “original translation”. First we note that the language cannot become smaller. All we do is adding rules to the transducers, and adding rules to transducers can only enlarge the generated language, not remove elements. So what we need to do is to prove that the language does not become larger.

First we show that the generated language does not change if the modification is applied only to the last intermediate transducer. Consider an arbitrary derivation step, whose left hand side is not $q[\perp]$, in the last intermediate transducer, when using “plain translation”. The derivation step will be $q[\tau[\dots]] \rightarrow \tau'[\dots]$. What would happen if we added the rule $q[\tau[\dots]] \rightarrow \perp$ and that one was used instead? There are two cases. In the first case the final transducer would never look at this particular instance of τ' . In this case the modification does not matter. In the other case, the result would be undefined, because instead of

$\tau'[\dots]$ this particular node is \perp , and there is no rule in the final transducer, whose left-hand side contains \perp . In neither case the output language becomes larger.

Now we show that using the modification in two consecutive intermediate transducers td_1 and td_2 gives same result as if using the modification in only td_2 . If td_1 uses “plain translation” an arbitrary derivation step, whose left hand side is not $q[\perp]$, is $q[\tau[\dots]] \rightarrow \tau'[\dots]$. Now one possibility is that td_2 applies to this particular instance of τ' the rule $q[\tau'[\dots]] \rightarrow \perp$. What would happen if we added the rule $q[\tau[\dots]] \rightarrow \perp$ to the first transducer td_1 and that was used instead? Then td_2 would have to use the rule $q[\perp] \rightarrow \perp$ on this node, and we get exactly the same result as before, when td_2 used $q[\tau'[\dots]] \rightarrow \perp$.

We also have to take into account the possibility that a previous derivation step of td_2 could copy or delete the subtree for which τ' is a node. But in the first case the same argument is used on all copies and in the second case, since it was deleted it does not matter what rule td_1 applied to τ . We conclude that the output language does not increase if td_1 is made with “plain translation”.

These statements together imply that the modification can be done to all intermediate transducers without changing the output language of the final transducer, which completes the proof.

3.5 Plain translation with terminating leaves

The “plain translation” gives an improvement in practical usability, but still many derivations will end in undefined trees. The reason for this is that usually not all of the top tables contain, somewhere in their subtables, rules with only terminals in the right hand sides. If there is not one such table on every path from the root to a leaf in the synchronization tree, the derivation can not terminate. The original regular tree grammar generates all trees over the input signature, and therefore the chance for a derivation to terminate is small.

Recall that a pseudo leaf is a node with only bottom symbols as subtrees. The next step in modification of the translation, is to only allow terminable tables to be pseudo leaves. Another limitation is that terminating tables may only be pseudo leaves. That is because any tables below can never be used since the derivation has already terminated.

3.5.1 Definitions

The regular tree grammar for this translation is defined as

$$REG_{term} = (\{S\}, \Sigma_1, R, S) . \text{ Here } R \text{ consists of all rules } S \rightarrow \tau[\underbrace{S, S, \dots, S}_{d \text{ times}}]$$

where $\tau \in J$ and τ is not terminating, and all rules $S \rightarrow \tau[\underbrace{\perp, \perp, \dots, \perp}_{d \text{ times}}]$ where

$\tau \in J$ and τ is terminable.

Note that R does not contain the rule $S \rightarrow \perp$ so the derivation of the regular tree grammar cannot terminate unless there are terminable tables. Note also that a table τ can be both terminable and not terminating, and then τ will appear in the right hand side of two rules.

The i :th intermediate transducer is given by $TD_{term}^{(i)} = TD_{plain}^{(n)}$.

The final transducer TD_{term}^{final} equals TD_{orig}^{final} .

3.5.2 Proof of correctness

Let $L_{term} = TD_{term}^{final}(TD_{term}^{(n-1)}(\dots(TD_{term}^{(1)}(L(REG_{term}))))\dots)$.

We have to show that $L_{term} = L(BG)$.

Note first that the output language of the regular tree grammar does not become larger (in most cases it becomes much smaller), that is $L(REG_{term}) \subseteq L(REG_{plain})$. Since the output language of the tree transducers do not become larger when some of the trees in the input language are removed, this implies that $L_{term} \subseteq L(BG)$. What we need to show is that also $L(BG) \subseteq L_{term}$, ie no trees are missing in L_{term} .

We will show that by proving that for any tree $t' \in L(BG)$, there is a tree $t \in L(REG_{term})$, such that $t' \in TD_{term}^{final}(\dots(TD_{term}^{(1)}(\{t\}))\dots)$. This implies that REG_{term} is indeed able to generate all trees that are required.

First look at the special case when $L(REG_{term}) = \emptyset$. Since S is the only nonterminal of REG_{term} , it can only happen if there is no rule $S \rightarrow \tau[\perp, \dots, \perp]$ for any τ . But that happens if no table of BG is terminable, and in that case $L(BG) = \emptyset$, since no derivation can terminate without terminating rules. Therefore we only need to consider the case when REG_{term} generates trees. The following lemma summarizes these conditions, and completes the proof.

Lemma

Let $t' \in L(BG)$ and let $L(REG_{term}) \neq \emptyset$. Then there exists a tree $t \in L(REG_{term})$ such that $t' \in TD_{term}^{final}(\dots(TD_{term}^{(1)}(\{t\}))\dots)$.

Proof of Lemma

Since $t' \in L(BG)$ we know from Section 3.4 that there is at least one derivation using the composition of REG_{plain} , $TD_{plain}^{(i)}$ for $i = 1, \dots, n-1$, and TD_{plain}^{final} that generates t' . Let Tr be the tree generated by REG_{plain} in one of those derivations.

The proof is done by induction over the tree Tr . If $Tr \in L(REG_{term})$ we are done. If not, Tr contains a finite nonzero number of internal nodes that either

- case 1) are terminal tables but are not pseudo leaves, or
- case 2) are not terminable tables but are pseudo leaves.

Let Ti be one of these nodes.

In *case 1*, remove all subtrees from Ti and replace with \perp .

In *case 2*, replace all symbols \perp right below Ti , with $Tj[\perp, \perp, \dots, \perp]$ where Tj is a terminable table. By the assumption $L(REG_{term}) \neq \emptyset$ there exists at least one such Tj .

In both cases the number of nodes that have any of these two properties decreases. By induction we can repeat these actions and eventually end up with a tree in $L(REG_{term})$. What is left to prove is that these actions does not affect the output language of the branching tree grammar.

Note that the proof is about showing that the set of trees one can generate does not become smaller when doing the modifications mentioned in case 1 and 2. We are switching from something “plain translation with terminating leaves” can do to something “plain translation” can do, and we concluded in the beginning of Section 3.5.2 that the language does not increase when doing so.

Proof for case 1

In this case a subtree $Ti[t_1, \dots, t_k]$ where Ti represents a terminal table, was replaced by $Ti[\perp, \dots, \perp]$. We will prove that it does not matter since t_1, \dots, t_k are never looked at by the tree transducers.

For any application of an intermediate transducer the rule looks like $q[Ti[\dots]] \rightarrow Tii[\perp, \dots, \perp]$. That is because since Ti represents a terminal table, then all its subtables are also terminal, in particular the one represented by Tii . The tree $Ti[\perp, \dots, \perp]$ may also have been copied when handling some node above Ti , but that does not matter since this argument can be used on all of these copies.

Now, this argument is repeated for all intermediate transducers. For the final transducer, let $Tiii$ be a symbol that Ti has been (indirectly or directly) replaced with. Since Ti was terminal, and thus also $Tiii$, all rules of the final transducer whose left-hand side contains $Tiii$ must have a terminal right hand side. Therefore the subtrees of $Tiii$ are not looked at either, and the proof for case 1 is done.

Proof for case 2

What we have to show is that the set of trees one can generate does not become smaller when the modification is used, that replaces the children of Ti , all of which are \perp , with tables Tj .

Here we will consider two sub-cases.

Sub-case 2.1: The depth of BG is > 1 .

We proved in the previous section that one can replace all transducers obtained by “original translation” in the translation of BG , with those obtained with the “plain translation with terminating leaves”, without changing the output language. Thus one can also go the other way and switch back to those of the

“original translation”. Let us therefore assume here that the transducers from “original translation” are used.

Recall that $Ti[\perp, \dots, \perp]$ was replaced by $Ti[Tj[\perp, \dots, \perp], \dots, Tj[\perp, \dots, \perp]]$. But by using the rule $q[Tj[X_1, \dots, X_d]] \rightarrow \perp$ in the first intermediate transducer, on all such subtrees the effect can be undone.

Note that the modification might enlarge the sets of trees the intermediate transducer can generate on this particular input tree. It can not decrease it, however, and that was all we needed to prove here.

Sub-case 2.2: The depth of BG is 1.

In this case the tree Tr is the input to the final transducer TD_{term}^{final} . The nodes are the names of the applied tables, and the table named Ti did by assumption not contain any rules with only terminals in their right-hand side. Thus the final transducer does not contain any rule with $Ti[\dots]$ as left-hand side, and at the same time only terminals in its right-hand side.

Since Ti is pseudo leaf and a final transducer does not contain rules whose left-hand side contains the symbol \perp , we must conclude that all derivations of TD_{term}^{final} resulting in output trees terminated before reaching the node Ti .

Therefore the output language does not change with the change of the particular node Ti . Thus it does not depend on any subtrees of Ti either, so the modification was harmless.

3.6 Normal translation

Here the tree grammar and the intermediate transducers are modified so that even more of the unnecessary subtrees of the synchronization trees will not be generated. This will at many occasions allow a derivation that took exponential space and time with “plain translation” to take only polynomial space and time. Here space and time counted are with respect to the size of the output tree.

3.6.1 Definitions

The regular tree grammar REG_{normal} is given by $(\{S\}, T, R, S)$ where R consists of all rules $S \rightarrow \tau[Y(1), Y(2), \dots, Y(d)]$ where $\tau \in J$ and τ is not terminating, and all rules $S \rightarrow \tau[\underbrace{\perp, \perp, \dots, \perp}_{d \text{ times}}]$ where $\tau \in J$ and τ is terminable.

For $z = 1, \dots, d$, $Y(z)$ is defined as follows: Let N be the set of all synchronized nonterminals that appear in some right hand side in some rule r in the top table τ or its subtables. If $z \notin \{num_d(first_1(w)) \mid (A, w) \in N\}$ then $Y(z) = \perp$; otherwise $Y(z) = S$.

The i :th intermediate transducer is defined as $TD_{normal}^{(i)} = (\Sigma_i, \Sigma_{i+1}, \{q\}, R, q)$, where R consists if all rules $q[\tau[x_1, \dots, x_{d^i}]] \rightarrow$

$$\tau.j[\underbrace{Y(1, x_1), \dots, Y(d, x_1)}_{d \text{ terms}}, \dots, \underbrace{Y(d^{i+1} - d + 1, x_{d^i}), \dots, Y(d^{i+1}, x_{d^i})}_{d \text{ terms}}]$$

where $\tau \in J^i$ and $j \in J$. R also contains the rule $q[\perp] \rightarrow \perp$. For $z = 1, \dots, d^{i+1}$ and $x = x_1, \dots, x_{d^i}$, $Y(z, x)$ is defined as follows: Let N be the set of all synchronized nonterminals that appear in some right hand side in some rule r in table τ or some of its subtables. If $z \notin \{num_d(first_i(w)) \mid (A, w) \in N\}$ then $Y(z, x) = \perp$; otherwise $Y(z, x) = q[x]$.

The final transducer TD_{normal}^{final} equals TD_{orig}^{final} .

3.6.2 Proof sketch of correctness

No proof of the correctness of this construction will be presented here. Instead, the discussion in the next subsection should give an intuitive idea of why the construction works. However, a sketchy idea for a proof could be the following:

The only difference between “normal translation” and “plain translation with terminating leaves”, is that some subtrees in the rules are replaced by \perp in “normal translation”.

These subtrees can never be used by later transducers, since those transducers only look at subtrees, whose number when numbering all nodes directly below its parent, match the synchronization numbers (when transformed from tuples to integers) in some right-hand side of a rule.

Later transducers in the chain look at rules from a smaller set of tables, and thus the available synchronization numbers must be more restrictive. Thus the earlier transducer can safely remove subtrees not matching any of the synchronization symbols in the table, whose name appear in the rule.

3.6.3 Comparison with the original translation using an example

We will now analyze the sources of inefficiency when translating the example in Subsection 3.3.2. As it turns out, “original translation” is not suitable for the example, whereas on the other hand “normal translation” will work well.

Using the algorithm of the “normal translation” from the previous subsection on BG_{ex} results in the regular tree grammar $REG_{ex,normal}$ and td transducers $TD_{ex,normal}^{(1)}$ and TD_{ex}^{final} . These look as follows:

$$REG_{ex,normal} = \{\{S\}, \{gen, term, \perp\}, R_{reg}, S\}$$

with $R_{reg} = \{S \rightarrow gen[S, \perp], S \rightarrow term[\perp, \perp]\}$ and

$$TD_{ex,normal}^{(1)} = \{TD_{input}, TD_{output}, \{q\}, R_{td}, q\} \text{ with } TD_{input} = \{gen, term\},$$

$TD_{output} = \{(gen.one), (gen.two), (gen.three), (term.one), (term.two)\}$ and

$$R_{td} = \begin{array}{ll} q[gen[x_1, x_2]] & \rightarrow gen.one[q[x_1], \perp, \perp, \perp], \\ q[gen[x_1, x_2]] & \rightarrow gen.two[q[x_1], q[x_2], \perp, \perp] \\ q[gen[x_1, x_2]] & \rightarrow gen.three[q[x_1], \perp, \perp, \perp] \\ q[term[x_1, x_2]] & \rightarrow term.one[\perp, \perp, \perp, \perp] \\ q[term[x_1, x_2]] & \rightarrow term.two[\perp, \perp, \perp, \perp] \\ q & \rightarrow \perp \end{array}$$

and the final transducer $TD_{ex,normal}^{final}$ is equal to the one of “original translation”.

First we will focus on how the intermediate transducers work more efficiently, by comparing the transducers of original and “normal translation”. We will not at this point compare “original translation” with the other translations that are less sophisticated than “normal translation”, since there is no obvious gain in efficiency for this particular example.

We let the transducer $TD_{ex,normal}^{(1)}$ take as its input tree, the tree in Figure 12. One possible output tree would be the one in Figure 15, which is indeed smaller than the tree in Figure 13 which is the corresponding tree when using “original translation”.

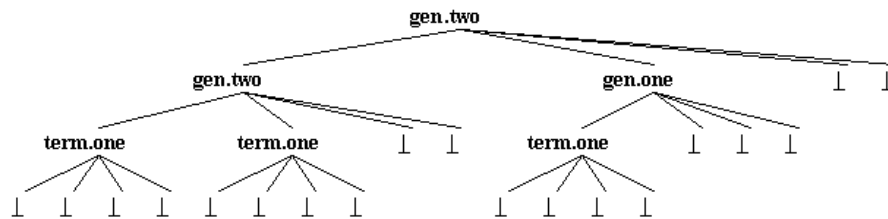


Figure 15: An output tree from the intermediate transducer from the “normal translation”.

When the final transducer ($TD_{ex,orig}^{final} = TD_{ex,normal}^{final}$) is applied, the same output tree as before, shown as the last tree in Figure 14, results. The gain may not seem so big in this example, but this is only because of the smallness of the derivation. It will become quite different if we consider a slightly larger output tree from the regular tree grammar $REG_{ex,orig}$, namely the one in Figure 16.

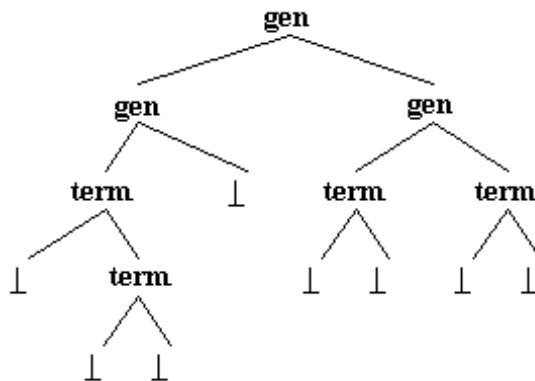


Figure 16: A larger output tree from the regular tree grammar first in the chain, that still will lead to the same end result.

With this as an input tree, $TD_{ex,normal}^{(1)}$ will generate the tree in Figure 15 once again. On the other hand $TD_{ex,orig}^{(1)}$ would generate a tree with approximately 100 symbols (\perp 's included). Yet the final transducer $TD_{ex,orig}^{final}$ would have

generated the same old final tree in Figure 14 from this huge input tree. In fact, when using “original translation”, this output tree will come up repeatedly many times, and sometimes after very long computations.

This illustrates how the unwanted exponential space requirement enters the computation of components from “original translation”. If we for example consider a binary synchronization tree, produced by a regular tree grammar from “original translation”, it will have a number of nodes depending exponentially on the height of the tree. Apparent from the regular tree grammar from “normal translation” (note that the right subtree is the leaf \perp in every node) only the left-most path from the root to a leaf will actually be kept in the end of the derivation, and the size of this part of the tree depends only linearly on the height of the tree.

Next we will discuss how to detect which nodes in the trees are actually unnecessary. Now the focus is more on the efficiency of the regular tree grammar, and in particular on how to cause it to not create subtrees that will necessarily be ignored by the transducers.

Consider the trees from Figure 12 and Figure 16. Using them will lead to exactly the same results (even though this fact may not completely appear from the discussion above, it can be shown), so most of the nodes in the second tree are unnecessary. Here “unnecessary” means that the nodes created from such a node – by means of the first transducer – will never be looked at by the final transducer.

First note that the node “term” at the bottom of the second tree can never be used. All derivations of the final transducer that reach the node “term” right above, will end there, since the subtables of table “term” only contain terminating rules. The translation “plain translation with terminating leaves” contains functionality to make the regular tree grammar avoid creating exactly this type of nodes. To be more specific, it would make the symbol “term” occur at every pseudo leaf, and never anywhere where it is not a pseudo leaf. The latter is because that if a “gen” occurs at a pseudo leaf, and it is used, the derivation will get stuck, since the rules are not terminal, and there are no more tables in the tree to use to get rid of the remaining nonterminals.

Furthermore, the whole right half of the second tree is unnecessary, since the third and fourth subtree of any given node are only used when a table is chosen, that contains synchronization symbols $\langle \frac{1}{0} \rangle$ or $\langle \frac{1}{1} \rangle$, and no such rules exists in the whole branching tree grammar. Note that the right half of the tree will be transformed into an even larger subtree by the transducers, if “original translation” is used, but at the end it will always be ignored by the final transducer. The “normal translation” aims at removing such branches as early as possible. Optimally they are never created at all, but sometimes with more complicated grammars these branches will have to be created by the regular tree grammar, but can then sometimes be removed by the intermediate transducers.

There is actually an even smaller tree that the regular tree grammar $REG_{ex,orig}$ could produce, that works as well, namely the one in Figure 17.

In fact, the only trees that the regular tree grammar ever needs to generate are the trees of the form shown in Figure 18. These are exactly the trees that

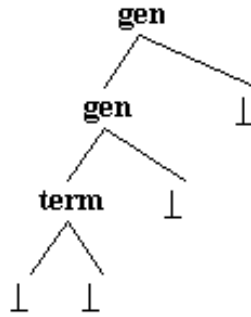


Figure 17: A smaller output tree from the regular tree grammar, that will also lead to the same end result.

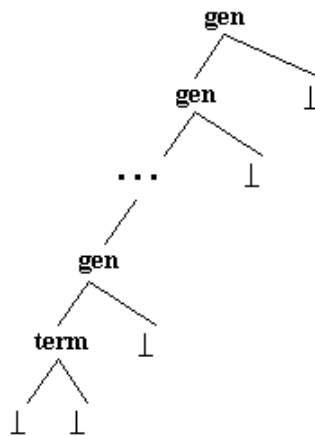


Figure 18: For this particular example, the only necessary trees in the language of the regular tree grammar, follow this pattern.

will be generated by $REG_{ex,normal}$, the regular tree grammar obtained using “normal translation”. The transducers created with “normal translation” for this particular branching tree grammar example, will be optimal as well. The last construction “extended translation” contains nothing that will improve this particular example. In Section 3.7.2 another example will be presented where “extended translation” will be the best choice.

3.7 Extended translation

There are still many cases where the space and time complexity with respect to the size of output trees are exponential. The modification discussed now will turn this into polynomial time for some of these cases.

It is possible, although it seems hard to prove, that with this translation the generated synchronization trees will have no unnecessary subtrees. If that is the case, this is about the best that can be done if implementing branching tree

grammars by composing regular tree grammar and tree transducers.

Here we only discuss how to modify the transducers, but a very similar optimization could be applied to the regular tree grammar, too. This is left as one of the possible extensions of the system.

The downside of this translation is a drastic increase in the number of states of the transducers; in the worst case it will depend exponentially on the number of nonterminals in the branching tree grammar. That means that with 10 nonterminals the transducers will become 1024 times as large as with “normal translation”, and thus take about 1000 times longer to apply. However, if that takes the space and time complexity with respect to output tree size from an exponential down to a polynomial, this will always be an improvement, as soon as the output trees we generate become sufficiently large.

Intuitively, the state q of the intermediate transducers means “any nonterminal”. When the transducer rule $q[Ti[...]] \rightarrow Tii[q[x_1], ...]$ is used, it intuitively means the following: It has previously been decided that a particular nonterminal (or several if synchronization demands them to use the same tables) will be derived further, by using some rule in the table named Ti . Now the choice is refined such that the rule must come from a table named Tii , which is a subtable to that table. At this stage it is completely unknown which nonterminal this might be, which the q in the left-hand side notifies. It is also unknown what nonterminals that might appear in the result tree, which is notified by the q in the right-hand side.

If we instead use as states all subsets of the set of nonterminals in the branching tree grammar, we can keep track of exactly which nonterminals can possibly be used on each table node in the synchronization tree.

As an example, if C and E are the nonterminals of the branching tree grammar, and all no rule with left-hand side E contains a C in its right-hand side, the intermediate transducer could, instead of the rule above, contain the following two rules:

$$\{C, E\}[Ti[...]] \rightarrow Tii[\{C, E\}[x_1], ...] \text{ and } \{E\}[Ti[...]] \rightarrow Tii[\{E\}[x_1], ...].$$

Note that $\{C, E\}$ has been replaced with $\{E\}$, in the second right-hand side since a C can not be created by using rules from table named Tii to replace E . A more elaborate example will be given in the next subsection.

Initially the algorithm to create the intermediate transducer uses all combinations of nonterminals as states, but in the end all unreachable states are removed from the transducer. In many cases almost all states will be unreachable. This means that in practice, the size of the resulting transducer does not always depend exponentially on the number of nonterminals.

3.7.1 Definitions

The regular tree grammar $REG_{extended}$ equals REG_{normal} .

The final transducer $TD_{extended}^{final}$ equals D_{normal}^{final} .

The i :th intermediate transducer is defined as $TD_{extended}^{(i)} = (\Sigma_i, \Sigma_{i+1}, Q, R, \{S_{BG}\})$ with $Q = \{q \in \wp(N_{BG})\}$.

R is constructed as follows:

1. For every $q \in \wp(N_{BG})$:
 - (a) For every $\tau \in J^i$ and for every $j \in J$:
 - i. Add to R the rule:

$$q[\tau[x_1, \dots, x_{d^i}]] \rightarrow$$

$$\tau.j[\underbrace{Y(1, x_1), \dots, Y(d, x_1)}_{d \text{ terms}}, \dots,$$

$$\underbrace{Y(d^{i+1} - d + 1, x_{d^i}), \dots, Y(d^{i+1}, x_{d^i})}_{d \text{ terms}}]$$
 - ii. Let ς be the set of all synchronized nonterminals that appear in some right hand side in some rule r in the table τ or its subtables, such that the left-hand side of r is in q .
 - iii. Now $Y(z, x)$ is defined, for $z = 1, \dots, d^{i+1}$ and $x = x_1, \dots, x_{d^i}$, as follows:

Let $q' = \{A \mid (A, w) \in \varsigma \text{ and } num_d(first_i(w)) = z\}$.

If $q' = \emptyset$, then $Y(z, x) = \perp$; otherwise $Y(z, x) = q'[x]$.
 - (b) In addition, R contains the rule $q[\perp] \rightarrow \perp$.
2. Finally, all states in Q that are not reachable are removed, as well as all rules in R with such states in their left-hand side.

Here is an informal explanation to clarify the idea:

In the construction ς contains all nonterminals that can possibly be derived when this occurrence of τ is handled by the final transducer. Now different terms in the right hand side of the rule in the intermediate transducer will get different subsets of ς . That is because those nonterminals in that do not have synchronization symbols that refer to that subtree do not need to be in the set which becomes the state that handles the next node.

3.7.2 An example where extended translation is useful

In this section we will explore how the “extended translation” affects the performance of a picture generating branching tree grammar. It is a slightly modified version of an example from page 149 of [Dre06].

The branching tree grammar is defined by $BG_{carpet} =$

$(N_{carpet}, T_{carpet}, I_{carpet}, J_{carpet}, R_{carpet}, S_{carpet})$ where $N_{carpet} = \{S, E, C\}$,

$T_{carpet} = \{quarters : 4, corner : 4, edge : 2, a\text{-part} : 0, b\text{-part} : 0, empty : 0\}$,

$I_{carpet} = \{0, 1\}$, $J_{carpet} = \{gen, term, one, two\}$ and $S_{carpet} = S$.

R_{carpet} is defined by:

$$R_{carpet}(gen.one) = \{ \\ C \rightarrow corner[a-part, E\langle 1 \rangle^0, C\langle 0 \rangle^0, E\langle 1 \rangle^0], \\ E \rightarrow edge[a-part, E\langle 0 \rangle^0], S \rightarrow quarters[C\langle 0 \rangle^0, C\langle 0 \rangle^0, C\langle 0 \rangle^0, C\langle 0 \rangle^0] \},$$

$$R_{carpet}(gen.two) = \{ \\ C \rightarrow corner[b-part, E\langle 1 \rangle^0, C\langle 0 \rangle^0, E\langle 1 \rangle^0], \\ E \rightarrow edge[b-part, E\langle 0 \rangle^0], S \rightarrow quarters[C\langle 0 \rangle^0, C\langle 0 \rangle^0, C\langle 0 \rangle^0, C\langle 0 \rangle^0] \},$$

$$R_{ex}(term.one) = \{C \rightarrow empty, S \rightarrow empty\},$$

and all other $R_{ex}(i.j)$ are empty.

The collage algebra that interprets the output trees of BG_{carpet} will only briefly be described here. For more details, see pages 149 to 153 of [Dre06]. The symbol *a-part* is interpreted as a black square, and the symbol *b-part* is interpreted as a square with a cross inside. The symbol *quarters* is only present in the root, and implies a rotation by $i \cdot 90$ degrees for each of its four subtree. The symbol *edge* yields a geometrical transformation which translates its argument horizontally one unit away from the center of the picture. Similarly *corner* corresponds to three geometrical transformations which translate one argument horizontally, one vertically and one both horizontally and vertically. Finally *empty* is interpreted as the empty picture.

The components when using “normal translation” are, the regular tree grammar REG_{carpet} and td transducers $TD_{carpet,normal}^{(1)}$ and TD_{carpet}^{final} which are given as follows:

$$REG_{carpet} = \{\{S\}, \{gen, term, \perp\}, R_{reg}, S\} \text{ with } R_{reg} =$$

$$\{S \rightarrow gen[S, \perp], S \rightarrow term[\perp, \perp]\}.$$

$$TD_{carpet,normal}^{(1)} = \{TD_{input}, TD_{output}, \{q\}, q\} \text{ with } TD_{input} = \{gen, term\},$$

$$TD_{output} = \{(gen.one), (gen.two), (term.one)\}, \text{ and } R_{td} = \{$$

$$\begin{aligned} q[gen[x_1, x_2]] &\rightarrow gen.one[q[x_1], q[x_1], \perp, \perp], \\ q[gen[x_1, x_2]] &\rightarrow gen.two[q[x_1], q[x_1], \perp, \perp], \\ q[term[x_1, x_2]] &\rightarrow term.one[\perp, \perp, \perp, \perp], \\ q[\perp] &\rightarrow \perp \end{aligned}$$

$$TD_{carpet}^{final} = \{TD_{final-input}, TD_{final-output}, \{S, C, E\}, R_{final}, S\} \text{ with}$$

$$TD_{final-input} = \{(gen.one), (gen.two), (term.one)\}, \text{ and } TD_{final-output} =$$

$$\{quarters : 4, corner : 4, edge : 2, a-part : 0, b-part : 0, empty : 0\}$$

$$\text{and } R_{final} = \{$$

$S[gen.one[x_1, x_2, x_3, x_4]]$	→	$quarters[C[x_1], C[x_1], C[x_1], C[x_1]],$
$C[gen.one[x_1, x_2, x_3, x_4]]$	→	$corner[a-part, E[x_2], C[x_1], E[x_2]],$
$E[gen.one[x_1, x_2, x_3, x_4]]$	→	$edge[a-part, E[x_1]],$
$S[gen.two[x_1, x_2, x_3, x_4]]$	→	$quarters[C[x_1], C[x_1], C[x_1], C[x_1]],$
$C[gen.two[x_1, x_2, x_3, x_4]]$	→	$corner[b-part, E[x_2], C[x_1], E[x_2]],$
$E[gen.two[x_1, x_2, x_3, x_4]]$	→	$edge[b-part, E[x_1]],$
$E[term.one[x_1, x_2, x_3, x_4]]$	→	$empty,$
$C[term.one[x_1, x_2, x_3, x_4]]$	→	$empty\}$

The components when using “extended translation” are REG_{carpet} , $TD_{carpet,extended}^{(1)}$ and TD_{carpet}^{final} . Both “normal translation” and “extended translation” yield the same regular tree grammar and final transducer, so only $TD_{carpet,extended}^{(1)}$ is defined below.

$TD_{carpet,extended}^{(1)} = \{TD_{input}, TD_{output}, \Gamma, R_{td}, \{S\}\}$ with

$TD_{input} = \{gen, term\}$, $TD_{output} = \{(gen.one), (gen.two), (term.one)\}$ and

$\Gamma = \{\{S\}, \{C\}, \{E\}\}$

When constructing the transducer according to the algorithm, the states $\{S, C\}$, $\{S, C, E\}$, $\{E, S\}$, \dots are also present, but they are all removed at the end as they turned out to be unreachable.

$R_{td} = \{$

$\{S\}[gen[x_1, x_2]]$	→	$gen.one[\{C\}[x_1], \perp, \perp, \perp],$
$\{S\}[gen[x_1, x_2]]$	→	$gen.two[\{C\}[x_1], \perp, \perp, \perp],$
$\{S\}[term[x_1, x_2]]$	→	$term.one[\perp, \perp, \perp, \perp],$
$\{S\}[\perp]$	→	$\perp,$
$\{E\}[gen[x_1, x_2]]$	→	$gen.one[\{E\}[x_1], \perp, \perp, \perp],$
$\{E\}[gen[x_1, x_2]]$	→	$gen.two[\{E\}[x_1], \perp, \perp, \perp],$
$\{E\}[term[x_1, x_2]]$	→	$term.one[\perp, \perp, \perp, \perp],$
$\{E\}[\perp]$	→	$\perp,$
$\{C\}[gen[x_1, x_2]]$	→	$gen.one[\{C\}[x_1], \{E\}[x_1], \perp, \perp],$
$\{C\}[gen[x_1, x_2]]$	→	$gen.two[\{C\}[x_1], \{E\}[x_1], \perp, \perp],$
$\{C\}[term[x_1, x_2]]$	→	$term.one[\perp, \perp, \perp, \perp],$
$\{C\}[\perp]$	→	$\perp\}$

The regular tree grammar is the same as for “normal translation” and “extended translation”. Therefore we will only consider the output tree from the intermediate transducer in this section.

A tree generated by $TD_{carpet,extended}^{(1)}$ is shown in Figure 19. The corresponding tree, generated by $TD_{carpet,normal}^{(1)}$ is shown in Figure 20. In both cases the output tree of the final transducer TD_{carpet}^{final} is the tree in Figure 21. This output tree is transformed by the collage grammar to the picture in Figure 22, consisting of four black squares.

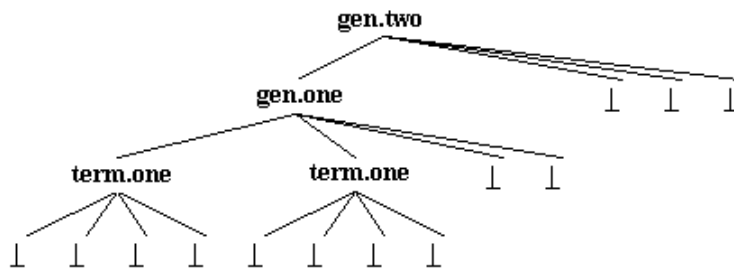


Figure 19: An output tree from the intermediate transducer from the “extended translation”.

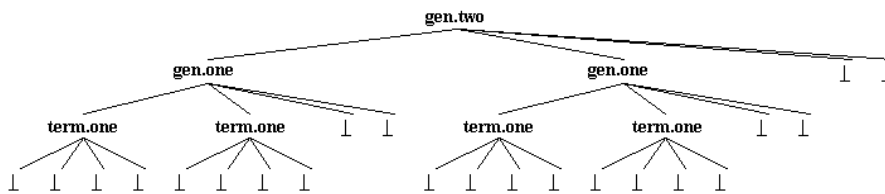


Figure 20: An output tree from the intermediate transducer from the “normal translation”, when using the same input as in Figure 19.

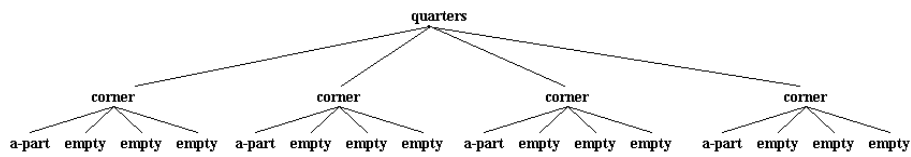


Figure 21: The output from the final transducer, when using as input the tree in Figure 19 as well as the one in Figure 20.



Figure 22: A picture obtained by interpreting the tree in Figure 21 tree using the example algebra. It consists of four black squares (corresponding to the four *a-part* in the tree), which are aligned so that they form a larger black square.

The argument that the subtrees that are missing of Figure 19 (but present in Figure 20) could indeed safely be removed without affecting the end result, is the following: Assume that the right *gen.one* in Figure 20 had been used. It is the root of the second subtree of its parent, and thus is this context labeled as the "1":th subtree, as we for technical reasons choose to count from zero. Therefore it corresponds to the binary string 01, because $d = 2$ and $num_2(0, 1) = 2$. Note that if d had been three, a number written in base 3 had been used instead of binary base (ie num_3 had been used). By inspection of the construction of "extended translation" (as well as for "normal translation"), one can see that this means that if we reach this node, a rule has been used with the synchronization tuple $\langle 1^0 \rangle$ in its right-hand side.

But such a right-hand side exists only for the nonterminal E . We also see that the derivation has only gone one step, so the initial nonterminal S can have become a C , but an E cannot yet have been created.

We can now conclude that the right *gen.one* can never be used for anything, and thus it is safe to remove it. This far it does not seem that we gain much by using "extended translation". To compute which nonterminals may have been reached only works for the first few depths, and cannot remove the exponential growth of a deep tree. But the "extended translation" also contains functionality that can be used to optimize at any depth, as the following example shows.

Let us now consider a slightly larger tree. We will still have to stick to as small trees as possible due to the exponential increase in width of the intermediate trees, when the height is increased.

A tree generated by $TD_{carpet,extended}^{(1)}$ is shown in Figure 23. The corresponding tree, generated by $TD_{carpet,normal}^{(1)}$ is shown in Figure 24 (only a part of it is shown).

The tree that both of these intermediate trees will give rise to is shown in Figure 25. Due to its size, only half the tree is shown. The picture for this tree is shown in Figure 26.

The unnecessary branch has been marked with a ring in Figure 24. Now why is it unnecessary? By the same argument as earlier, the nonterminal has to be an E when/if one reaches the node *gen.one* above the ring, since that node is the second subtree of its parent. But if we look at table *gen.one* we see that E only has the rule $E \rightarrow edge[a-part, E \langle 0^0 \rangle]$. Since $num_2(0, 0) = 0$ the only subtree to be used is the one numbered zero, ie the one to the left of the ring. Thus the subtree marked with the ring can be replaced with \perp .

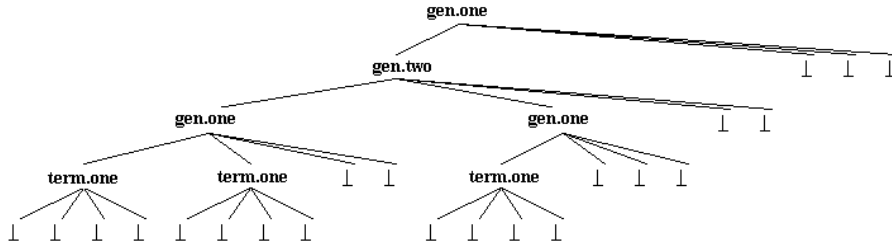


Figure 23: An output tree from the intermediate transducer from the “extended translation”.

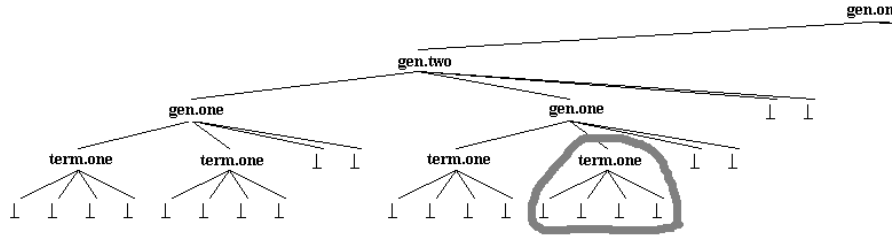


Figure 24: An output tree from the intermediate transducer from the “normal translation”, when using the same input as in Figure 23. The “unnecessary subtree” is marked with a ring.

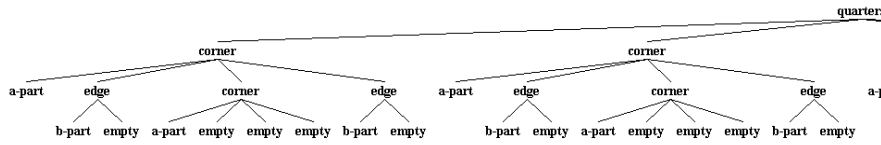


Figure 25: The output from the final transducer, when using as input the tree in Figure 23 as well as the one in Figure 24.

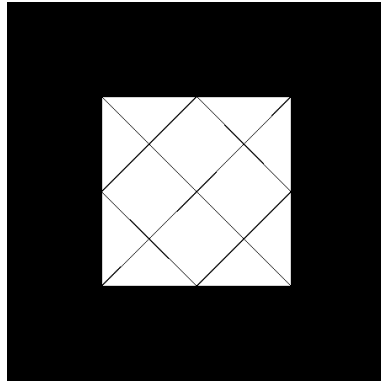


Figure 26: A picture obtained by interpreting the tree in Figure 25 tree using the example algebra.

Had we not known that E was the only possible nonterminal, we could not have removed the branch. All nodes where a C is possible need to keep both its first and second subtree, since the right hand sides of C contain both the tuple $\langle 0_0^0 \rangle$ and $\langle 1_1^0 \rangle$. This is a way to see why “normal translation” cannot be optimal. In this construction the transducers never keep track of which nonterminals are possible at any given node, or more precisely what nonterminal-named state the final transducer can possibly have when reaching this node.

If we consider a larger tree, built by adding more branches to the tree discussed above, we can see another way to detect unnecessary branches. Since E can never go over to a C in the grammar (no C occurs in any right-hand side of an E), for all nodes below the node to the left of the ring, only the nonterminal E is relevant, and thus only the first subtree needs to be saved in all nodes in this branch.

Finally, in order to further illustrate how an extended intermediate transducer works, the derivation to create the tree in Figure 23 is shown step by step in Figure 27. The last step of the derivation is omitted, as it would have been a copy of Figure 23.

Note that the states named $\{E, C\}, \{E, C, S\}, \dots$ that potentially could have been present, never had to be used. This is because it was always possible to deduce whether an E , S or C was a possible nonterminal, and it was never the case that several of them were possible. This, in turn, is because no right-hand side of any rule contained a C and an E with the same synchronization tuple, in the same table (and S did not occur in any right hand side at all). It can generally be worthwhile to try to rewrite a branching tree grammar such that it gets this property, in order to get efficient components when using “extended translation”.

In Figure 28 a large picture is shown, yielded by a tree from the same grammar. With “extended translation” this took less than a second with a 2GHz computer. With “normal translation” the time needed is unknown but smaller pictures take several minutes to generate.

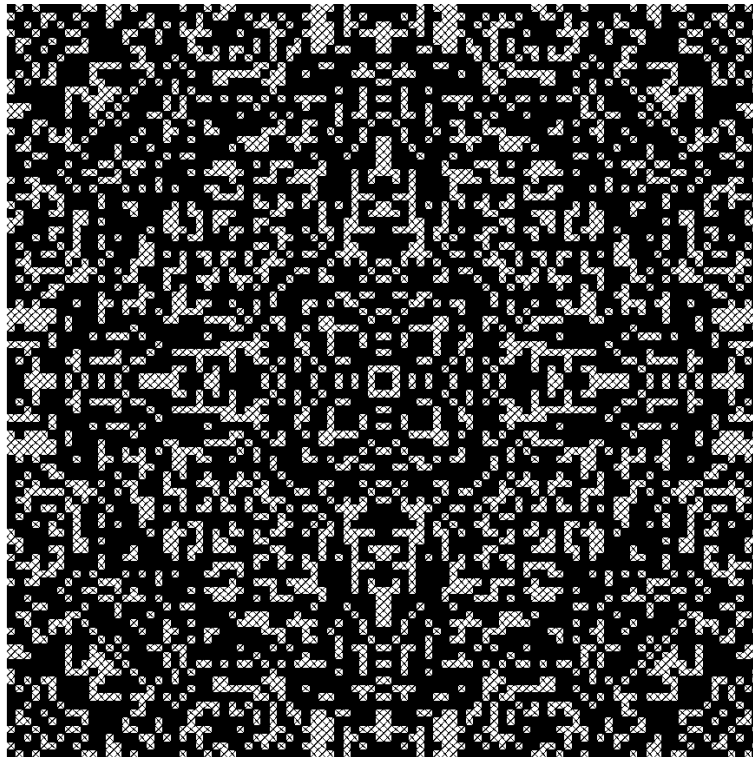


Figure 28: The picture obtained by interpreting a very large tree from the example grammar.

3.8 Turning a branching tree grammar into a simple branching tree grammar

All of the algorithms in Sections 3.3 to 3.7 are designed to work on simple branching tree grammars. Therefore, a branching tree grammar $BST_{nonsimple}$ that is to be translated is first transformed to a simple branching tree grammar BST_{simple} , as follows.

If $BST_{nonsimple} = (N, T, I, J, R, S)$ is of depth n , then the simple branching tree grammar $BST_{simple} = (N_{simple}, T, I, J_{simple}, R_{simple}, S_{simple})$ constructed is of depth $n + 1$.

We let $N_{simple} = \{A' : A \in N\}$, where $A' \notin T$ for all $A \in N$, and

$$J_{simple} = J \cup \{axiomtable, derivationtable, terminatetable\}.$$

For every $\tau \in J^n$, we define the following tables: $R_{simple}(derivationtable . \tau)$ equals $R(\tau)$, with every nonterminal A therein replaced by A' .

Similarly, $R_{simple}(axiomtable . \tau)$ equals $\{S_{simple} \rightarrow S\}$ with the nonterminals A in S replaced by A' . Finally $R_{simple}(terminatetable . \tau)$ is the table $\{A' \rightarrow A : A \in N \cap T\}$. All other tables of BST_{simple} are empty.

It is clear from the construction above that all derivations using BST_{simple} that generate output trees will start by using the single table within *axiomtable*, then use tables within the table *derivationtable* a number of times, and then finish by using the table *terminatetable* once.

Consider derivations using $BST_{nonsimple}$ and BST_{simple} , respectively. If the choices of tables and rules taken from the table *derivationtable* in BST_{simple} follows exactly the choices in $BST_{nonsimple}$, the same output tree will be generated. To be precise, the derivation $BST_{nonsimple}$ may generate more than one tree on the way, but all of them can be generated by BST_{simple} as well by choosing *terminatetable* in an earlier step.

The fact that the languages $L(BST_{nonsimple})$ and $L(BST_{simple})$ are equal is not formally proven here, but should be intuitively apparent because of the argument mentioned above.

3.9 Restricting top table sequences by regular expressions

We now discuss how table sequences at top level can be restricted by regular expressions. The objective of this functionality is to limit the derivation of a branching tree grammar such that the choices of tables at top level seen as strings over the table names, are members of some regular language specified by the user. Note that, in one and the same derivation, unsynchronized nonterminals may be replaced using different choices of tables even at top level. However, the resulting strings must all be members of the specified regular language.

First the branching tree grammar is translated to a regular tree grammar and a sequence of transducers according to one of the algorithms. Then the regular tree grammar is replaced by another more restricted version of the same regular tree grammar, where all paths from the root to a leaf in the output tree yield a string in the specified regular language.

The detailed construction will not be presented here, but the idea is that every nonterminal in the regular tree grammar corresponds to a state in a NFA (non-deterministic finite automaton, see [ASU86]) that accepts the specified regular language. Only nonterminals corresponding to accepting states have rules that turn them into the terminal \perp .

We can conclude that this modification does not add any generative power to the branching tree grammar. This can be seen from the second theorem in Section 3.1, which states that this modified regular tree grammar, together with the sequence of transducers, can be turned back into a branching tree grammar with the same depth as the original one. On the other hand, restricting a grammar with a regular expression may greatly reduce the complexity of the grammar. Another benefit for the user is that it can be exploited to exclude a large number of derivations that do not generate any tree.

3.10 Additions in order to allow nonterminal derivations

Some additional symbols and rules have to be added to the regular tree grammar and the tree transducers, in order to simulate stepwise derivations of the branching tree grammar.

First the branching tree grammar is translated to a regular tree grammar REG and a sequence of transducers $td^{(1)}, \dots, td^{(n-1)}$, and td^{final} according to one of the algorithms in Sections 3.3 to 3.7.

Now, for every nonterminal A in REG do the following:

- To the input signatures and output signatures of $td^{(i)}$ we add $A : 0$.
- To the rules of $td^{(i)}$ we add $\gamma[A] \rightarrow A$, for each state γ .
- To the input signature of td^{final} we add $A : 0$.
- To the output signature of td^{final} we add $\gamma : 0$ for each state γ .
- To the rules of td^{final} we add $\gamma[A] \rightarrow \gamma$, for each state γ .

Note that this does not conflict with the requirement that the set of states of a transducer is disjoint with its output signature, as $\gamma : 0$ and $\gamma : 1$ are distinct symbols.

In TREEBAG, when using the branching tree grammar in the “stepwise derivation” mode, the instruction “stepwise derivation” is passed to REG , which will then generate trees containing nonterminals. These nonterminals are left unchanged by the intermediate tree transducers, and the final transducers replaces them by the actual nonterminals of the branching tree grammar. Recall that the state of the final transducer is equal to the nonterminal the branching tree grammar would have used at this time, which explains the rules of the form $\gamma[A] \rightarrow \gamma$ in td^{final} .

3.11 Obtaining synchronization strings for a derivation

The functionality of showing synchronization strings gives the user more insight in how the derivations of the simulated branching tree grammar BG behave, by adding the synchronization information to the output tree when using the “stepwise derivation” mode. First the changes described in Section 3.10 are made. Then the final transducer td^{final} is changed in the following way:

All synchronization symbols in I_{BG} are added to the output signature of td^{final} , seen as symbols of rank one.

If n is the depth of BG , the symbols $- : 0$, $proj-1 : 0$, \dots , $proj-n : 0$,

$subst : (n+1)$ and $syncinfo : (n+1)$ are added to the output signature of td^{final} .

Furthermore, a new initial state γ'_0 is added, as well as the rule $\gamma'_0 \rightarrow subst[\gamma_0[x_1], -, -]$ where γ_0 was the old initial state.

Finally, recall that the final transducer contains some rules of the form

$$A[\tau[x_1, \dots, x_{d^n}]] \rightarrow t[[B_1[x_{num_d(\alpha_1)}], \dots, B_l[x_{num_d(\alpha_l)}]]$$

In these rules we replace each $B_i[x_{num_d(\alpha_i)}]$ by

$$subst[B_i[x_{num_d(\alpha_i)}], \alpha_{i1}[proj-1], \dots, \alpha_{in}[proj-n]] \text{ where } \alpha_{i1}\alpha_{i2}\dots\alpha_{in} = \alpha_i.$$

Now, another kind of tree transducer, the YIELD transducer, is added to the end of the series of components, such that it uses the output of the final transducer as input, and give its output as the output of the simulated branching tree grammar.

The symbols $- : 0, proj-1 : 0, \dots, proj-n : 0, subst : (n+1)$ are special symbols that are interpreted by the YIELD transducer. What will happen is that the yield transducer collects all synchronization information that the final transducer left all over the output tree, and attaches these to the nonterminals.

Slightly informally, the YIELD-transducer can be described as a mapping from the set of all trees, to trees which are the input tree with some substitutions made. The substitutions to be made are specified by the special symbols $subst : n+1$ and $proj_i, i = 1, \dots, n$ in the input tree. Applying $subst$ to $n+1$ trees is done by evaluating them recursively yielding t_1, \dots, t_n , and then substituting t_i for every occurrence of $proj_i$ in t_0 . The t_0 with these substitutions made is then returned.

For formal definition of YIELD transducer see page 446 in [Dre06] or [FV98]. In the following subsection an example illustrating all of this will be given.

3.11.1 An example of a translated grammar rewritten to show synchronization information

We will once again look at the example from Section 3.3.2.

Let N be the nonterminal used by the regular tree grammar.

The final transducer from Section 3.3.2, will after the modification in the previous subsection, look like $TD_{ex}^{final} = \{TD_{final-input}, TD_{final-output}, \{S, S_r, \gamma\}, R_{final}, \gamma\}$, with

$$TD_{final-input} = \{(gen.one), (gen.two), (gen.three), (term.one), (term.two), N\},$$

$$TD_{final-output} = \{\circ : 2, \bullet : 1, \triangleleft : 0, \triangleright : 0, | : 0, S : 0, S_r : 0, - : 0, 0 : 1, 1 : 1, proj - 1 : 0, proj - 2 : 0, subst : 3, syncinfo : 3\}, \text{ and}$$

$$R_{final} = \{$$

$$\gamma[x_1] \rightarrow subst[S[x_1], -, -],$$

$$S[gen.one[x_1, x_2, x_3, x_4]]$$

$$\rightarrow \circ[subst[S[x_1], 0[proj-1], 0[proj-2]]],$$

$$S_r[gen.one[x_1, x_2, x_3, x_4]]$$

$$\begin{aligned}
 &\rightarrow \circ[\text{subst}[S_r[x_1], 0[\text{proj-1}], 0[\text{proj-2}]]], \\
 &S[\text{gen.two}[x_1, x_2, x_3, x_4]] \\
 &\rightarrow \bullet[\text{subst}[S[x_1], 0[\text{proj-1}], 0[\text{proj-2}]], \text{subst}[S[x_2], 0[\text{proj-1}], 1[\text{proj-2}]]], \\
 &S_r[\text{gen.two}[x_1, x_2, x_3, x_4]] \\
 &\rightarrow \bullet[\text{subst}[S_r[x_2], 0[\text{proj-1}], 1[\text{proj-2}]], \text{subst}[S_r[x_1], 0[\text{proj-1}], 0[\text{proj-2}]]], \\
 &S[\text{gen.three}[x_1, x_2, x_3, x_4]] \\
 &\rightarrow |[\text{subst}[S[x_1], 0[\text{proj-1}], 0[\text{proj-2}]], \text{subst}[S_r[x_1], 0[\text{proj-1}], 0[\text{proj-2}]]], \\
 &S_r[\text{gen.three}[x_1, x_2, x_3, x_4]] \\
 &\rightarrow |[\text{subst}[S[x_1], 0[\text{proj-1}], 0[\text{proj-2}]], \text{subst}[S_r[x_1], 0[\text{proj-1}], 0[\text{proj-2}]]], \\
 &S[\text{term.one}[x_1, x_2, x_3, x_4]] \quad \rightarrow \quad \triangleright, \\
 &S_r[\text{term.one}[x_1, x_2, x_3, x_4]] \quad \rightarrow \quad \triangleleft, \\
 &S[\text{term.two}[x_1, x_2, x_3, x_4]] \quad \rightarrow \quad \triangleleft, \\
 &S_r[\text{term.two}[x_1, x_2, x_3, x_4]] \quad \rightarrow \quad \triangleright, \\
 &S[N] \quad \rightarrow \quad \text{syncinfo}[S, \text{proj-1}, \text{proj-2}], \\
 &S_r[N] \quad \rightarrow \quad \text{syncinfo}[S_r, \text{proj-1}, \text{proj-2}] \\
 & \}
 \end{aligned}$$

A derivation by the final transducer is shown in Figures 29 to 33. The last one is used as input to the YIELD transducer, and the output from that is shown in Figure 34. For comparison, see the corresponding part of the derivation in Figure 35, or for the whole derivation Figure 11 in Section 3.3.2.

In the output tree of the YIELD transducer, the synchronization strings corresponding to a certain table depth are read vertically from bottom to top. Thus the vertical strings of digits in Figure 34 correspond to the rows of the matrices of digits in Figure 35.

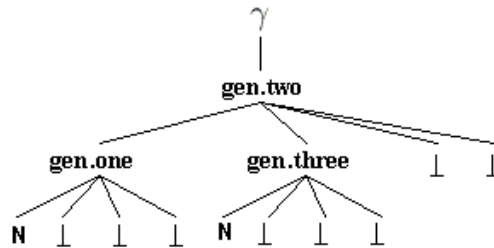


Figure 29: The first step in a derivation using the alternative final transducer.

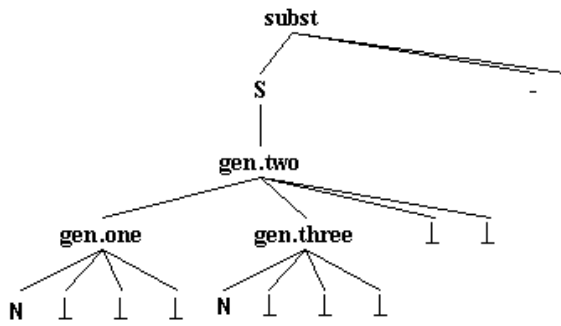


Figure 30: The second step in the derivation.

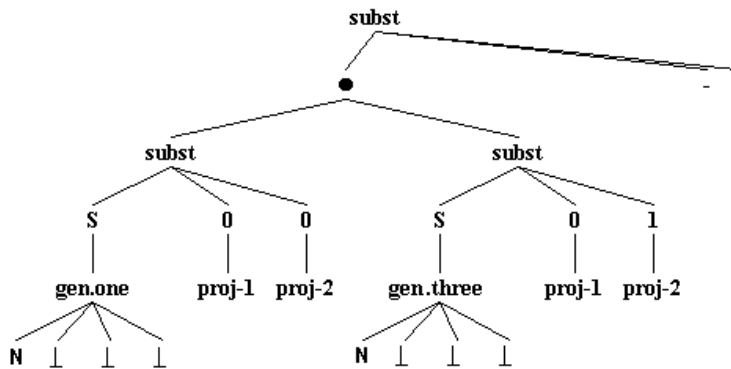


Figure 31: The third step in the derivation.

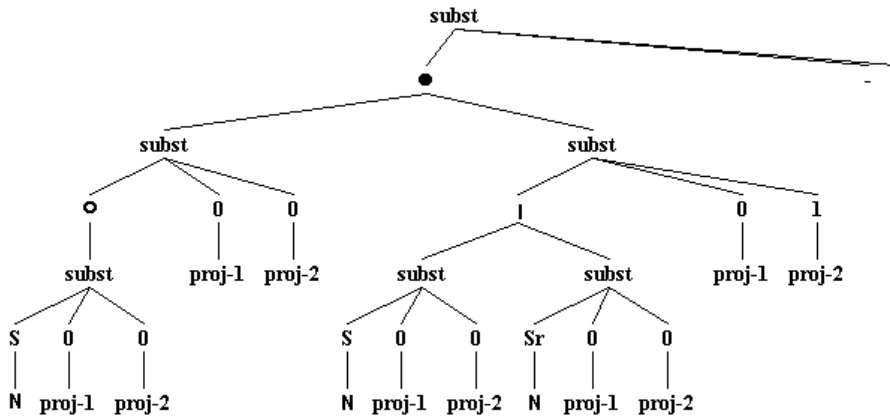


Figure 32: The fourth step in the derivation.

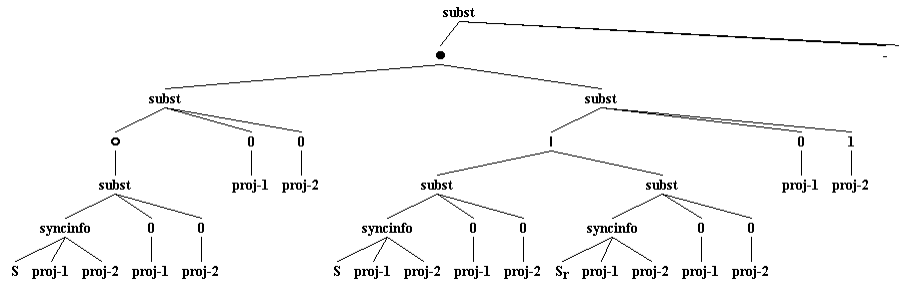


Figure 33: The last step in the derivation.

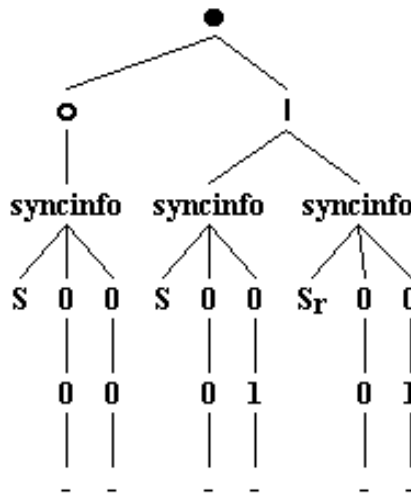


Figure 34: The output from the YIELD transducer, when using the tree in Figure 33 as input.

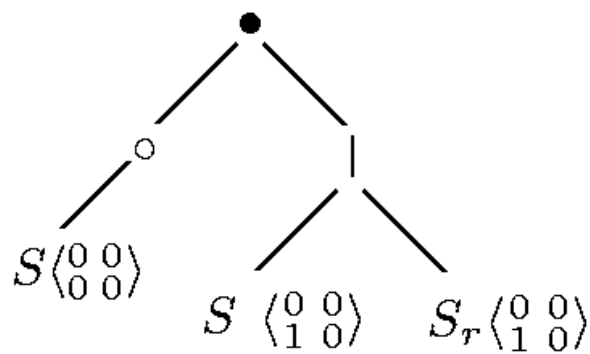


Figure 35: A step during the derivation of the equivalent untranslated branching tree grammar, that corresponds to the tree in Figure 34.

4 The implementation

This section describes some aspects of the implementation of branching tree grammars in TREEBAG. This implementation consists the Java-files `generators/BSTGrammar.java`, `generators/SuperTable.java`, and `generators/SyncedRule.java`. The main class for the branching tree grammar is named `BSTGrammar`.

There will not be any in-depth description of the Java code used, but rather comments that can be useful for someone who wants to modify the branching tree grammar, and to some extent for users of the grammar. The advice for using the implementation efficiently is also summarized in Section 5.1.

4.1 Translation

The implementation reads the file, in which the user has specified the branching tree grammar. This information is put into an internal data structure. Then the implementation proceeds by creating several components, in the way explained in Chapter 3. One field in the input file specifies whether to use “plain”, “normal” or “extended” translation. Depending on this field the definitions from either Section 3.4, 3.6 or 3.7 are chosen.

The additional modifications from Sections 3.10 and 3.11, which allow a user to derive stepwise as well as inspect the synchronization information, are always made. The modifications of Section 3.8, which transforms the branching tree grammar into a simple branching tree grammar is used if necessary. If a regular expression restricting the allowed table sequences at top level is specified in the input file the regular tree grammar is created as specified in Section 3.9.

This translation is remade every time a branching tree grammar-component is loaded in TREEBAG, even if the translated files where up to date.

The resulting components are written to files in the same folder as the branching tree grammar, with the suffix `.1`, `.2`, `.3` and so on, added to the file name. The alternative final transducer, used to show synchronization information, which is specified in Section 3.11, gets the suffix `.Y`.

The resulting files are then read and handled using code of the existing implementations of the corresponding classes. This process is invoked by the code of the branching tree grammar.

For the TREEBAG user, all of this is invisible, unless they choose to check the file directory for the translated files. The only component that is visible in the TREEBAG GUI, is the branching tree grammar, even though the other components are actually used internally.

4.2 Interaction with the branching tree grammar

There are a number of ways a user can interact with the branching tree grammar component through the TREEBAG GUI. Most of these interactions are handled

by the branching tree grammar by sending appropriate commands to its internal components. The different commands are described in the corresponding sections in the rest of this chapter.

In general the tree transducers always transform their input trees all the way, while the regular tree grammar is set to different modes (eg “results only” or “derive stepwise”) depending on the mode of the branching tree grammar.

If a user feels this functionality is not enough, for example if the user wants to derive the transducers stepwise, he or she can always open the branching tree grammar, then close it and instead load the translated components separately, to be able to interact with them.

The two main modes are “table enumeration” as described in Section 4.3, and “random tables” as described in Section 4.4. The “table enumeration” can be done either in the “results only” mode, which enumerates trees in a particular sequence, or the “derive stepwise” mode, which shows derivations in a stepwise manner, in which one can see which nonterminals are used.

4.3 Table enumeration

Unlike the case of regular tree grammars the command “results only” does not enumerate all trees in the branching tree grammar language. It rather enumerates all possible choices of tables at top-level, and then for each such choice chooses nondeterministically among the subtables in those tables.

This is due to the fact that the regular tree grammar, whose language is enumerated, can be seen as the device choosing the table sequences at top level, while the tree transducers choose subtables and rules. Instead, if a user wants to see all trees, they can, for each enumeration, use commands of the kind “choose new subtables at depth 2”, which is described below.

For both “results only” and “derive stepwise” there is an interesting special case, which is when the top table has only two tables, such that one is not terminable and the other is terminal. In this case, the sole effect of enumerating a new top table choice, is increasing the (maximum) height of the output tree by one. This behavior makes this type of branching tree grammar particularly easy to use. A user who wants that functionality is encouraged to rewrite the branching tree grammar to have exactly this property.

4.3.1 Results only

When the branching tree grammar is in the “results only” mode, the regular tree grammar is instructed to work in this mode too. In this mode the user of the branching tree grammar component never sees any nonterminals, but only complete trees.

The commands “advance” and “reset” are forwarded to the regular tree grammar, “advance” causing the next top table choice to be enumerated, and “reset” causing the branching tree grammar to reset to the first choice in the enumeration.

4.3.2 Derive stepwise

The mode “derive stepwise” causes the regular tree grammar to be set to “derive stepwise”. This causes the output tree of the grammar, which specifies table choices at top level, to generate nonterminal trees. The final transducer contains rules to recognize these nonterminals and replace them with the actual branching tree grammar nonterminals, exactly as described in Section 3.10.

By default, the synchronization strings are not present in the output tree. See Section 4.6 for the mode that reveals these strings.

4.4 Random tables

The mode “random tables” is to be used when a user wants to derive the output trees step by step, and have the ability to go backwards in the derivation in order to try new nondeterministic choices for some of the latest steps. Internally the regular tree grammar is set to the mode “random tables” in order to accomplish this.

The commands ”refine”, ”back” and ”reset” are handled by sending them to the regular tree grammar.

When using “refine”, all table choices at top level are kept, and one new choice is added, causing the length of the derivation to be increased by one step.

4.5 Choose new subtables/rules

The command “choose new subtables at depth i ”, where $i \geq 2$, causes the $(i-1)$ th transducer and all transducers after to be reinvoked with a new random seed. The command “choose new rules” similarly reinvokes the final transducer only. For instance, new subtables at depth 3, means that all choices of top tables and their direct subtables are kept, but all choices of tables at higher depths are remade.

4.6 Hide/Show sync info

The commands “Hide sync info” and “Show sync info” toggle whether to use the final transducer as the last component in the chain or the YIELD-transduction together with the transducer (whose file gets suffix .Y) described in Section 3.11. The standard final transducer is the one that does not include the synchronization information in its output tree.

As described in Section 3.11, the synchronization information is added to the nonterminals in the form of monadic subtrees. To use this feature in a picture generating device, the user may extend the algebra, so that it interprets the synchronization symbols in some meaningful way. If a free term algebra is used nothing has to be done, since the output is simply passed on to the display component which draws the tree in a treelike form, interpreting any names of the nodes, simply as text strings to be drawn.

5 Results and conclusions

The theoretical work and the implementation have turned out to be successful, in that the objectives stated in the master thesis specification have been fulfilled, and all of the branching tree grammar examples from [Dre06] now work with the implementation.

All of these grammars also work quite efficiently, ie there does not seem to be any exponential time dependence on the tree size in the grammars tested. The grammars also do not generate a lot of undefined or repeated trees, though it happens sometimes.

However it should be noted that some of these grammars have been slightly rewritten in order to make them become efficient. Thus, the implementation can not be said to handle all cases efficiently, but still it seems to handle most cases without requiring too much rewriting. Some detailed tips about how to rewrite the grammars are described in the next subsection.

At the beginning of this work, there were quite some doubts about whether it was even possible to, by using the approach of translating branching tree grammars to regular tree grammars and top-down tree transducers, make an implementation that handled all examples from the book with reasonable efficiency. Taking this into account, the results of the master thesis are quite positive.

The work on the proof sketches have been quite illuminating and instructive, and because of the proof sketches I find the construction reliable. There could of course still be some mistakes in the construction, as not many persons have checked the proofs.

In fact during the search for a proof sketch for the correctness of the “plain translation with terminating leaves”, I could not immediately find a convincing argument for its correctness, and after some thinking I instead ended up with a counter example showing that the construction did not work. As a consequence, the construction was modified to its current state, which led to the proof sketch in Section 3.5.2.

5.1 Some hints regarding efficiency for users of the implementation

The mode “plain translation” is very inefficient, and should be seen as having been implemented mainly for the theoretically interested who wants to inspect the components involved or the synchronization trees. For all cases of picture generation “normal translation” should be tried first. If it works too slowly, try “extended translation”. If the grammar still works slowly, or yields many undefined results, consider the following:

The translation works best if the branching tree grammar has two tables at the top level, one with only non-terminable tables (recall that they have no rules with only terminals in right hand side) and one with only terminal tables (recall that they only have rules with only terminals in right hand side). Alternatively

several non-terminable tables and/or several terminal tables can be used with good results, but then a regular expression should be used to specify suitable top table sequences. A rule of thumb could be that the more terminal and non-terminable rules are mixed, within the same tables at top level, the more undefined results one will get when using the grammar.

Note that if the branching tree grammar has non-disjoint sets of nonterminals and terminals, the grammar is in the implementation rewritten to a simple branching tree grammar, having two tables at top level, with only terminal rules in the second, as described in Section 3.8. In this case there might not be any gain in rewriting the grammar, as described above. It has not been studied very much what properties make a grammar of this kind efficient, but it appears to work better if all of the nonterminals are also terminals.

5.2 Remaining problems and possible improvements

The implementation is of course not perfect, and there are many possible improvements. Some of these are described in the following subsections, together with hints regarding workarounds for the problems.

5.2.1 Extended translation for regular tree grammars

The “extended translation” described in Section 3.7 is applied to the transducers only. It is not implemented for regular tree grammars, nor is it described in this report how this could be done. It should however be rather straightforward to convert the “extended translation” algorithm, to one that improves the regular tree grammar as well. However, this modification would not improve anything if the sets of nonterminals and terminals intersect, and cannot in any obvious manner be used if a regular expression for top-tables is specified.

It is likely that some grammars, though none of the ones tested, could benefit largely if this was implemented. There is, however, a workaround yielding a similar effect, namely to simply add a new single table at the top, containing all tables at top level. Thus the depth is increased by one, and the regular tree grammar created will be very simple. There will be one additional transducer now, but since transducers are optimized more thoroughly than tree grammars it will hopefully work more efficiently anyway.

5.2.2 Undefined and repeated results

Depending on the structure of the branching tree grammar at hand, the derivations may lead to many undefined results, and the same tree may appear in the output many times. The reason for the first is basically that there are sometimes combinations of table choices that can lead to an output tree, but does not do so in a particular situation, due to unfortunate choices of rules within them. The reason for the second is that the derivation may terminate before using all the tables in the given list of choices, and thus two different lists may result in the same output tree.

As mentioned above, each table at top level should preferably be either non-terminable or terminal. Many branching tree grammars can be rewritten to this form without too much trouble, and if so, the user is encouraged to do so. However there may be ways to improve the implementation in order to handle grammars not complying with this in a smarter way.

A possible improvement to the implementation could be an algorithm that takes the branching tree grammar as input and tries to rewrite it to a branching tree grammar generating the same language, using the guidelines discussed above. This algorithm could then be used before doing the translation described in Chapter 3.

However it is probably quite difficult to design this algorithm, unless one limits it to handle certain special cases and simpler tasks, like recognizing all terminal tables and moving them out to the second top-table, leaving the rest in the first top-table. Cases when terminal right hand sides and nonterminal ones are heavily mixed will be more difficult to handle.

5.2.3 Enumerating every generated tree once and only once

As stated in Section 4.3, the enumeration function does not enumerate all trees, but rather enumerates all choices of tables at top level. At deeper levels, random choices are made. In order to see all trees one has to use the command “choose new tables” several times, for each enumerated choice of tables at top level. Thus results will be repeated, and because of the randomness a user can never be sure that they has seen them all.

One way to accomplish this could be to make the transducers, for every new input, enumerate all possible computations. There is currently no support for this implemented in the tree transducers in TREEBAG, so that would have to be added to the implementation. It does not seem to be clear whether this modification alone would remove all cases of duplicate trees. Theoretically the problem could be solved using a vector listing all enumerated trees so far. Then when a tree comes up a second time, it can immediately be discarded, turning to the next. However, such an approach is of course not feasible since it requires a vast amount of memory and has an unacceptable time complexity.

5.2.4 Avoiding unwanted recomputation by the tree transducers

There are some points where the implementation causes the tree transducers to recompute their output, using a new random seed, when this may neither be necessary nor appropriate.

Perhaps the most important example is the case when toggling the “show sync info” mode. As described in Section 4.1, there are two variants of the final transducer, one used when showing, and one used when hiding synchronization information. Since these two transducer components do not currently have access to each other’s random seed, they must choose the rules of the tables independently. Thus, if the user has used the branching tree grammar to generate a certain tree and wants to see the synchronization information of the

nonterminals, it does not work to select “show sync info” since then a new tree is generated, and the synchronization information for this tree is displayed.

Of course, if the final transducer is deterministic (ie the branching tree grammar is deterministic), then the problem disappears. This is because there is no need to change the synchronization tree from the last intermediate transducer at all.

5.2.5 Regular expressions regulating subtables

As described earlier a regular expression can be used to regulate the choice of tables at top level, by restricting the regular tree grammar to generate only trees in which all paths, from the root to a leaf, correspond to a string in the language of the regular expression. This raises the questions whether a similar construction can be done for the deeper levels of tables.

A nearby problem is that the use of a regular expression is not supported when the sets of nonterminals and terminals intersect. This can be explained with the transformation of the branching tree grammar into a simple one. Since a new table is inserted above the old top level, we are back to the problem of using a regular expression on subtables.

It is also not obvious how to properly define what the regular expression would mean in this case. Consider the case when the set of terminals equals the set of nonterminals. In this case all paths from the root to a leaf in the output tree must always have the same length. Consider further two nonterminals in a generated tree of this kind, and the two series of table choices that led to them respectively. Now there are suddenly two requirements of these series - one that they both should be a string in the regular language and the other that the strings have equal length.

In this case, is it clear that the resulting construction does not have a generative power exceeding the class of branching tree grammar? After that, there is also the more general case when nonterminals and terminals only partially intersect, which is probably even more difficult to study.

A possible solution could be changing the tree transducers in such a way that they use many states instead of just q . The states are intuitively the states of a NFA accepting the language given by the corresponding regular expression. Only the transducer state(s) f with the same name as a final state of the NFA have the rule with left hand side $f[\perp]$ defined, and thus any table sequence not complying with the regular expression will yield an undefined result instead of an output tree.

The fact that this construction can be done with transducers, means that the language generated is still one that can be generated with a branching tree grammar, according to the theorem in Section 3.1. However this solution appears to have some severe problems, discussed below.

The most important one seems to be the very high probability of receiving an undefined result. What in principle happens is that the regular tree grammar constructs a strings of table choices, that later are checked for acceptance in

all the transducers, using a more restrictive pattern. Thus it seems that most derivations will fail to create a tree.

What about combining this idea with the “extended translation”? It already has multiple states instead of the single state q . While it could be possible to make this work by using as states, the set of all combinations of a state from the NFA and a state from the transducer as it looks now, it would include a potentially very large number of states. No further work regarding this matter has been made in this master thesis.

5.3 Acknowledgement

I would like to thank my thesis supervisor Frank Drewes, who has always given tips and quick answers to questions that emerged during the work on the theory and implementation, as well as given a lot of valuable feedback on this report.

References

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
- [Bak79] Brenda S. Baker. Composition of top-down and bottom-up tree transductions. *Info. and Control*, 41(2):186–213, May 1979.
- [CDG⁺97] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 1997. release October, 1st 2002.
- [DE04] Frank Drewes and Joost Engelfriet. Branching synchronization grammars with nested tables. *Journal of Computer and System Sciences*, 68:611–656, 2004.
- [Dre01] Frank Drewes. Tree-based generation of languages of fractals. *Theoretical Computer Science*, 262:377–414, 2001.
- [Dre04] Frank Drewes. Introduction to tree languages. Available on: <http://www.cs.umu.se/kurser/TDBC92/VT04/trees.pdf>, 2004. Part of the course material for “Formal languages”.
- [Dre06] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.
- [Eng76] Joost Engelfriet. Surface tree languages and parallel derivation trees. *Theor. Comput. Sci.*, 2(1):9–27, 1976.
- [Eng82] Joost Engelfriet. Three hierarchies of transducers. *Mathematical Systems Theory*, 15(2):95–125, 1982.
- [FV98] Zoltan Fulop and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.
- [HK91] Annegret Habel and Hans-Jörg Kreowski. Collage Grammars. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. 4th. Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 411–429. Springer-Verlag, 1991.
- [Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18:280–315, 1968.
- [Vik04] Lars Vikberg. Compilers for tree grammars and tree transducers. Available on: <http://www.cs.umu.se/~c971vg/exjobb/thesis.ps>, 2004.