Treebag—A Tree-Based Generator for Objects of Various Types*

Frank Drewes

Department of Computer Science, University of Bremen,
P.O. Box 33 04 40, D–28334 Bremen, Germany
e-mail: drewes@informatik.uni-bremen.de

Abstract. A software system called Treebag—*Tree-Ba*sed *G*enerator—is presented. The aim behind Treebag is to allow for the generation and visualization of objects of all kinds: pictures, trees, graphs, strings, numbers, etc. The basic principle is that *tree generators* like, for instance, regular tree grammars, generate terms over symbols that are interpreted by appropriate *algebras* as operations on the domain of objects under consideration. Thus, every term is viewed as an expression that denotes one of the objects of interest. These objects can be visualized using appropriate *displays*.

1 Introduction

In mathematics and computer science, one of the most natural and universal ways to denote elements of a given data space is to use expressions over a suitable set of operations, like $\sqrt{3/7} + 5$. The usefulness of this concept is even strengthened by the fact that such an expression may contain variables—in which case it does not denote a particular, single object, but a set of objects or a rule to obtain new objects (output objects) from given ones (input objects).

Formally, an expression is a *term* or, as it is often called, a *tree* over a ranked alphabet. A ranked alphabet is also called a *signature*. It consists of a set of symbols each of which has a finite *rank*. An *algebra* provides an *interpretation* of the symbols of a given signature by associating with every symbol an operation (on a certain domain) in such a way that the arity of the operation coincides with the rank of the symbol.

With these observations in mind it is clear that any device that produces terms (i.e., defines a language of terms in the sense of formal language theory) can be used in conjunction with a suitable algebra in order to generate objects of an arbitrary data space. Moreover, in addition to such *tree grammars* one may use *tree transductions* (also called *tree transformations*) in order to take further advantage of the representation of objects by terms. This is because a term t that denotes an object O may be transformed by a tree transduction τ into a term $\tau(t)$ that denotes another object O'—possibly with respect to a different algebra. Thus, a transduction of terms induces a transformation of objects between possibly different types of data (see also [Eng80, Eng94, Dre96a]). From a theoretical point of view maybe the most interesting aspect is that many properties of the generated (sets of) objects and of the induced transformations turn out to be decidable, provided that the chosen grammars,

*Partially supported by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems) through the University of Bremen.

transducers, and algebras are taken from suitable classes [Eng94, Dre96a, DE98].

In this paper, a software system called TREEBAG (*Tree-Based Generator*) is presented in which these facts are exploited in order to allow for the generation, manipulation, and visualization of objects of various kinds. From the user's point of view (which in many ways just reflects the internal structure of the system) it allows to build interactively a network of tree grammars, tree transducers, algebras, and displays. Tree grammars and tree transducers produce output trees that can be fed into tree transducers and into displays. Furthermore, an algebra can be attached to each display. Then the display will use the algebra in order to interpret its input terms and will visualize the objects obtained in this way. Of course, for different sorts of objects (i.e., of algebras) different sorts of displays are necessary.—A display which is designed to visualize graphs will usually not be able to display the values obtained from an algebra over integer numbers.

Presently, the main types of tree grammars, tree transducers, algebras, and displays that have been implemented are[1]

- regular tree grammars (cf. [GS97]),
- top-down tree transductions (cf., e.g. [Rou70, Tha70a, Eng75, GS84, GS97]) and the YIELD transduction (cf. [EV85]),
- algebras on truth values, strings, integer numbers, and terms (the free term algebra and the YIELD algebra), for chain-code pictures (cf. [MRW82]) and turtle graphics [PL90], and for the type of pictures generated by collage grammars [HK91] and iterated function systems [Bar88, PJS92], and
- displays for objects with string representation (strings, numbers, terms, etc.), for a tree-like visualization of terms, for line drawings (as produced by the chain-code and turtle algebras), and for collages.

As the user may interactively compose regular tree grammars with arbitrary sequences of top-down tree transductions and YIELD transductions one gets in fact a very large class of tree languages and tree transductions—the one which is denoted by R-TBY$^+$ in [DE98] and which equals the class of output languages of the composition closure of macro tree transductions (see also, e.g., [EV85, EV88]). In particular, the IO hierarchy of tree languages (whose first levels are the regular and the IO context-free tree languages) is obtained in this way. Furthermore, it follows that one can simulate several types of controlled rewriting (cf. [ERS80]). As a consequence, the current version of TREEBAG makes it possible to obtain—by a suitable combination of tree grammars, tree transductions, and algebras—regular, context-free, indexed, ET0L, and many other classes of string languages, the mentioned types of tree languages, context-free and other chain-code picture languages, picture languages generated by ET0L-systems with turtle interpretation, context-free collage languages, and picture languages generated by iterated function systems.

Furthermore, since one may use the output of a given tree grammar or tree transducer as input for several displays and tree transducers at the same time, one may for instance create a constellation as shown in Figure 1. Here, the term generated by a regular tree grammar is shown as a tree in the window on the left (i.e., by a display associated with the free term

[1] There are in fact more than these, but these are the most interesting ones.

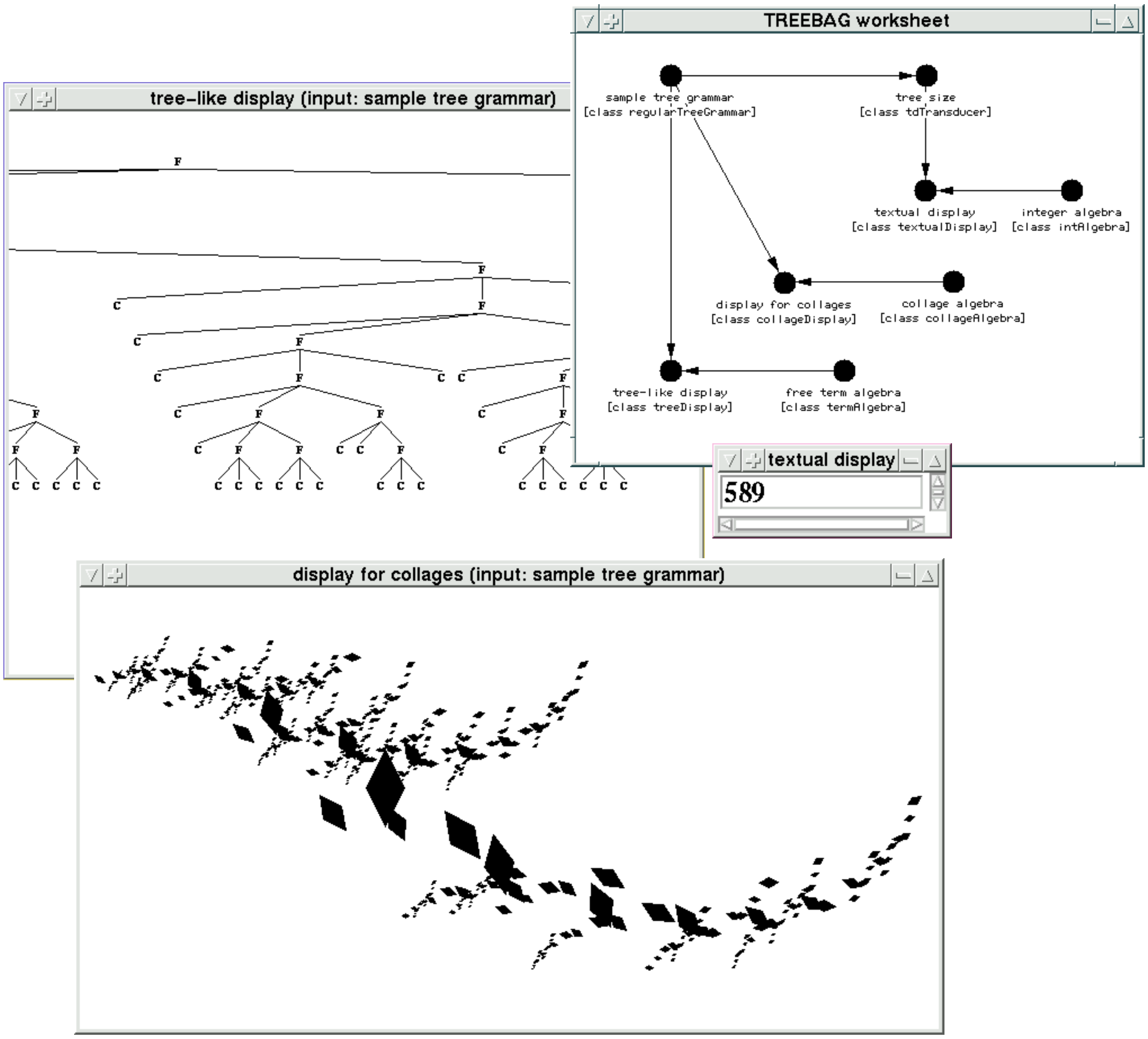

Figure 1: A snapshot of Treebag.

algebra). The second window shows the value of this term with respect to an algebra over collages, whereas in the third window the size of the tree is shown. The latter is obtained by taking the original term as input for a tree transducer that computes the size of its input term by transforming it into a term over operations on integer numbers (which can be evaluated by an integer algebra).

The motivation to implement a system like Treebag has been at least twofold. One of the aims has been to explore and demonstrate the usefulness of tree grammars and tree transducers in the framework discussed above, and to build a flexible open system that allows to visualize some of the basic concepts of the theory of computing by tree transductions. The second aim has been to provide some piece of software that can help teaching theoretical computer science (and in particular formal language theory) by demonstrating concepts like derivations, derivation trees, interpretation of symbols, transformation of objects, and their relation. For this reason, the implementation of regular tree grammars and top-down tree transducers allows to perform derivations step by step, which makes it possible to visualize in an easily understandable way how grammars work.

The paper is structured as follows. In the next section the theoretical concepts realized in TREEBAG are recalled and explained as far as needed. In Section 3 the main principles and the structure of TREEBAG are explained, whereas Section 4 describes the TREEBAG worksheet, a user interface that allows to interact with the system. Section 5 is devoted to the discussion of some examples, and in Section 6 some ideas for future extensions are discussed.

2 Some theory

In this section the theoretical concepts implemented in TREEBAG are explained as far as necessary. For further reading see [GS84, Dre96a, GS97, DE98] and the references cited there.

The set of natural numbers (including 0) is denoted by $\mathbb{N}$. As usual, A^* denotes the set of all finite strings over an alphabet A. This includes the empty string λ. The length of a string s is denoted by $|s|$. The reflexive and transitive closure of a binary relation $\rightarrow$ is denoted by $\rightarrow^*$.

A *signature* is a set Σ of pairs (f, n) called *ranked symbols* (or just *symbols*, for brevity), where f is the *name* of the symbol and $n \in \mathbb{N}$ is its rank. A ranked symbol (f, n) is also denoted as $f{:}\,n$ or just f if n is understood or of minor interest. A signature that does not contain any symbol of rank > 1 is called *monadic*.

A *term* (or *tree*) is a construct $f[t_1, \ldots, t_n]$, where f is a symbol name and $t_1, \ldots, t_n$ are terms. For $n = 0$ the brackets are usually omitted, i.e., the term $f[]$ is identified with the symbol name f. If Σ is a signature and T is a set of terms then $\mathrm{T}_\Sigma(T)$ denotes the smallest set S of terms containing T such that $f[t_1, \ldots, t_n] \in S$ for all $f{:}\,n \in \Sigma$ and $t_1, \ldots, t_n \in S$. Furthermore, T_Σ abbreviates $\mathrm{T}_\Sigma(\emptyset)$ and $\Sigma(T) = \{f[t_1, \ldots, t_n] \mid f{:}\,n \in \Sigma \text{ and } t_1, \ldots, t_n \in T\}$.

From now on, we shall fix an indexed set $X = \{x_1, x_2, x_3, \ldots\}$ of pairwise distinct *variables*. The set $\{x_1, \ldots, x_n\}$ is denoted by X_n. In the following, it is assumed that variables do not occur as symbols of rank 0 in signatures, i.e., $x_i{:}\,n \in \Sigma$ implies $n > 0$. The set of all terms $t \in \mathrm{T}_\Sigma(X_n)$ in which every variable in X_n occurs exactly once is denoted by $\widetilde{\mathrm{T}}_\Sigma(X_n)$. The term which is obtained from $t \in \mathrm{T}_\Sigma(X)$ by replacing every occurrence of x_i with t_i for $i = 1, \ldots, n$ is denoted by $t[\![t_1, \ldots, t_n]\!]$.

A *left-linear term rewrite system* R over Σ is a finite set of rules $s \rightarrow t$ such that $s, t \in \mathrm{T}_\Sigma(X)$, s is linear, and every variable in t occurs already in s. The term rewrite relation $\rightarrow_R$ is defined as usual, i.e., $u \rightarrow_R v$ holds for terms $u, v \in \mathrm{T}_\Sigma$ if there is a rule $(s \rightarrow t) \in R$ such that $u = u_0[\![s[\![s_1, \ldots, s_n]\!]]\!]$ and $v = u_0[\![t[\![s_1, \ldots, s_n]\!]]\!]$ for some $u_0 \in \widetilde{\mathrm{T}}_\Sigma(X_1)$ and $s_1, \ldots, s_n \in \mathrm{T}_\Sigma$.

A Σ-*algebra* (or just *algebra*) is a pair $\mathcal{A} = (\mathbb{A}, (f_\mathcal{A})_{f \in \Sigma})$, where $\mathbb{A}$ is a set, the *domain* of $\mathcal{A}$, and $f_\mathcal{A}$ is a function $f_\mathcal{A}\colon \mathbb{A}^n \rightarrow \mathbb{A}$ for every symbol $f{:}\,n \in \Sigma$. The *value* of a term $t = f[t_1, \ldots, t_n] \in \mathrm{T}_\Sigma$ with respect to $\mathcal{A}$ is given by $val_\mathcal{A}(t) = f_\mathcal{A}(val_\mathcal{A}(t_1), \ldots, val_\mathcal{A}(t_n))$.

A *tree generator* is any device with a finite description that yields terms as output. A *tree* *grammar* is a tree generator that does so without requiring input. Such a tree grammar always specifies a *tree language*—the set of terms being generated. Regular tree grammars, whose definition is recalled next, are a particularly useful sort of tree grammars. They generalize the concept of regular string grammars to trees in a very straightforward way.

2.1 Definition (*regular tree grammar*) A *regular tree grammar* is a tuple $g = (N, \Sigma, P, S)$, where N is a finite set of symbol names called *nonterminals*, Σ is a finite signature such that $A{:}\, 0 \notin \Sigma$ for all $A \in N$, P is a finite set of *productions* of the form $A \to t$ where $A \in N$ and $t \in \mathrm{T}_\Sigma(N)$, and $S \in N$ is the *initial nonterminal*. The tree language generated by g is given by $\mathrm{L}(g) = \{t \in \mathrm{T}_\Sigma \mid S \to_P^* t\}$. ◇

Note that, if $S \to_P^* t$ is a derivation in a regular tree grammar then it always holds that $t \in \mathrm{T}_\Sigma(N)$. That is, in a sentential form only leaves can be nonterminals—similar to regular string grammars, where nonterminals can only appear as the right-most symbols in sentential forms. As an example, consider the regular tree grammar

$$\begin{array}{rcl} g & = & (\ \{\mathit{even}, \mathit{odd}\}, \\ & & \ \ \{1{:}\,0, +{:}\,2\}, \\ & & \ \ \{\mathit{odd} \to 1,\ \ \mathit{odd} \to +[\mathit{odd}, \mathit{even}],\ \ \mathit{odd} \to +[\mathit{even}, \mathit{odd}], \\ & & \ \ \ \ \mathit{even} \to +[\mathit{even}, \mathit{even}],\ \ \mathit{even} \to +[\mathit{odd}, \mathit{odd}]\ \}, \\ & & \ \ \mathit{odd}\ ). \end{array}$$

It generates the set of all terms over $+{:}\,2$ and $1{:}\,0$ that denote odd numbers (under the usual interpretation of these two symbols).

The next concept to be recalled is that of a tree transduction, a tree generator which requires terms as input. Formally, a *tree transduction* is a binary relation $\tau \subseteq \mathrm{T}_\Sigma \times \mathrm{T}_{\Sigma'}$, where Σ and Σ' are signatures—the input and output signature, respectively. Such a tree transducer is viewed as a tree generator which, on input $s \in \mathrm{T}_\Sigma$, generates in a nondeterministic way one of the output terms t such that $(s, t) \in \tau$. If there is at most one output term t for every input term s, τ is considered as a partial function. One of the best-studied devices to compute tree transductions is the so-called top-down tree transducer, which was introduced by Rounds and Thatcher [Rou70, Tha70a, Tha70b, Tha73] as a natural generalization of finite-state transductions on strings.

2.2 Definition (*top-down tree transducer*) A *top-down tree transducer* is a tuple $\mathit{td} = (\Sigma, \Sigma', \Gamma, R, \gamma_0)$, where

- Σ and Σ' are finite signatures,
- Γ is a finite set of symbol names called *states*, such that $\gamma{:}\, 1 \notin \Sigma \cup \Sigma'$ for all $\gamma \in \Gamma$,
- $\gamma_0 \in \Gamma$ is the *initial state*, and
- $R \subseteq \Gamma(\Sigma(X)) \times \mathrm{T}_{\Sigma'}(\Gamma(X))$ is a left-linear term rewrite system.

The tree transduction computed by td (which is denoted by td as well) is the set of all pairs $(s, t) \in \mathrm{T}_\Sigma \times \mathrm{T}_{\Sigma'}$ such that $\gamma_0[s] \to_R^* t$. ◇

Due to the special form of the rules, a top-down tree transducer transforms its input tree into an output tree by reading it symbolwise from the top down. The intermediate trees in a derivation consist of an already generated part of the output tree at the top, a layer of states and, below the states, the subterms of the input tree which still have to be processed.

2.3 Example As an example (taken from [DE98]) consider the top-down tree transducer

$$
\begin{array}{rcll}
td & = & (& \{xor\colon 2, not\colon 1, true\colon 0\}, \\
 & & & \{and\colon 2, or\colon 2, true\colon 0, false\colon 0\}, \\
 & & & \{invert, copy\}, \\
 & & & \{invert[xor[x_1, x_2]] \;\rightarrow\; and[or[invert[x_1], copy[x_2]], or[copy[x_1], invert[x_2]]], \\
 & & & \;\; copy[xor[x_1, x_2]] \;\rightarrow\; or[and[invert[x_1], copy[x_2]], and[copy[x_1], invert[x_2]]], \\
 & & & \;\; invert[not[x_1]] \;\rightarrow\; copy[x_1], \\
 & & & \;\; copy[not[x_1]] \;\rightarrow\; invert[x_1], \\
 & & & \;\; invert[true] \;\rightarrow\; false, \\
 & & & \;\; copy[true] \;\rightarrow\; true\ \}, \\
 & & & invert\ ).
\end{array}
$$

Under the usual interpretation of the symbols as operations on truth values, *td* computes for every input term over *xor*, *not*, and *true* a term over *and*, *or*, *true*, and *false* whose value is the inverse of the value of the input term. The first two rules are obtained from the definition of the exclusive or by applying deMorgan's laws (which is necessary because the output signature contains no negation symbol). ◇

The *YIELD signature* Σ_Y is the infinite signature consisting of all symbols

- $const_{f:n}\colon 0$, where $f\colon n$ is a ranked symbol,
- $proj_i\colon 0$, where $i \geq 1$, and
- $subst\colon n+1$, where $n \in \mathbb{N}$.

The intuition behind the choice of these symbols is that they allow to denote terms using the operation of substitution. Roughly speaking, the symbol $const_{f:n}\colon 0$ is interpreted as a constant denoting the symbol $f\colon n$, $proj_i$ stands for the variable x_i, and $subst\colon n+1$ allows to construct a new term from $n+1$ terms $t_0, \ldots, t_n$ by substituting $t_1, \ldots, t_n$ for the variables $x_1, \ldots, x_n$ occurring in t_0. This is made precise by the definition of the so-called YIELD algebra.

2.4 Definition (*YIELD algebra* and *YIELD transduction*) The *YIELD algebra* $\mathcal{Y}$ is the Σ_Y-algebra whose domain is the set of all terms, and whose interpretation of symbols is defined as follows, for all $g \in \Sigma_Y$:

- $g_{\mathcal{Y}} = f[x_1, \ldots, x_n]$ if $g = const_{f:n}$,
- $g_{\mathcal{Y}} = x_i$ if $g = proj_i$,
- $g_{\mathcal{Y}}(t_0, \ldots, t_n) = t_0[\![t_1, \ldots, t_n]\!]$ for all terms $t_0, \ldots, t_n$, if $g = subst\colon n+1$.

The *YIELD transduction* is the tree transduction Y that consists of all pairs $(t, val_{\mathcal{Y}}(t))$ such that $t \in T_{\Sigma_Y}$ and $val_{\mathcal{Y}}(t)$ contains no variables. ◇

Remark. From a theoretical point of view it is sometimes more appropriate to speak of a *set of YIELD transductions* by saying that *YIELD* is the set of all tree transductions $y \subseteq \mathrm{T}_\Sigma \times \mathrm{T}_{\Sigma'}$ such that $\Sigma \subseteq \Sigma_\mathrm{Y}$ is finite and $y = \{(t, val_\mathcal{Y}(t)) \in Y \mid t \in \mathrm{T}_\Sigma\}$ (see [DE98]). ◇

It is shown in [EV85, Theorem 7.3] that the composition closure $(\mathit{TD} \cup \mathit{YIELD})^+$ of $\mathit{TD} \cup \mathit{YIELD}$ (where *TD* is the class of all top-down tree transductions and *YIELD* is defined as in the remark above) equals the composition closure of the set of all macro tree transductions. Furthermore, it is known from [EV88] that this class contains the tree transductions computed by the so-called n-level tree transducers. As a consequence, the class of all tree languages that can be generated by the tree generators defined in this section (i.e., as the class of images of regular tree languages under transductions in $(\mathit{TD} \cup \mathit{YIELD}^+)$) captures all tree languages from the IO and the OI hierarchies of tree languages (where the IO hierarchy starts with the regular and the IO context-free tree languages, and the OI hierarchy starts with the regular and the OI context-free tree languages (cf. [Mai74, ES77, ES78, Eng80, Dam82]).

3 Treebag

In this section the architecture of Treebag—which is independent of the currently available user interface that will be described in the next section—is explained.

Treebag is a system implemented in Java that allows to combine tree generators with appropriate algebras in order to obtain the desired interpretation of generated terms. There are four basic types of components that can be arranged in a network in order to generate and visualize objects, namely tree grammars and tree transducers, algebras, and displays. Their main functionality is very closely oriented at the theoretical definitions given in Section 2. Tree transducers require as input the terms generated by a tree generator (i.e., by another tree transducer or by a tree grammar), whereas tree grammars generate terms "from scratch". An algebra associates with every term a value by interpreting symbols as (possibly partial) operations on some domain. The value of a term will be undefined if the term contains symbols unknown to the algebra, or if an operation yields an undefined result. A display can visualize objects of a certain type. For this, an algebra must be attached to it and it must receive as input the terms generated by a tree generator. Then, the algebra determines the interpretation of input terms. In other words, if a display is given an input term it uses the attached algebra in order to evaluate the term and then visualizes the result, provided that the result is defined and is of the right type.

Every class of Treebag components can (but need not necessarily) provide a list of strings that it will interpret as commands controlling its behaviour. A typical example is the command `advance` of the class `regularTreeGrammar`, which causes the `regularTreeGrammar` to advance to the next term in the enumeration of the tree language it implements.

It is important to notice that a component only provides a list of command names rather than implementing a procedure for user interaction. In fact, even a display does not open a window by itself. Instead, it provides a raw "canvas" (or a similar window component) that cannot directly become visible, but must be put into a window by a part of the system that realizes a user interface. This has two advantages. Firstly, it makes it very easy to add new Treebag components (i.e., new classes of tree generators, algebras, or displays) because the implementor need not take care about questions concerning the user interface. Secondly, one of the ideas behind TREEBAG is to allow for the implementation of alternative user interfaces without the need to touch the rest of the system. There could, for instance, be a specialized user interface for context-free Chomsky grammars that combines a regular tree grammar generating "derivation trees", a string and a free term algebra, and a string as well as a tree display. The two displays could be placed in a window divided into two parts, and the commands to control the regular tree grammar could be arranged as items in a menu bar. In this way, one would obtain a specialized version of TREEBAG that allows to visualize context-free grammars by showing in the main window the currently generated string and its derivation tree. Since the existing implementations of tree grammars, algebras, and displays could be used without any modification, implementing such a system would not be much more than an easy exercise in writing user interfaces. In fact, the same interface could be used for other kinds of grammars like collage grammars (see Section 5.6) because the interface does not need to "know" the objects being displayed, i.e., the string algebra and string display could be replaced with a collage algebra and a collage display without doing any harm.

There is one more thing that every class of TREEBAG components is required to implement, namely a procedure that initializes an object by reading its definition from an ASCII file. In other words, every such class defines its own syntax. In case of tree generators, the syntax is quite exactly the same as their usual mathematical notation (see Section 2). Thus, the definition of a regular tree grammar needed to initialize an object of the class `regularTreeGrammar` in such a way that it behaves like the example in the previous section would be

$$\begin{array}{l}(\ \{\mathtt{even},\mathtt{odd}\},\\ \quad\{1{:}\,0,+{:}\,2\},\\ \quad\{\mathtt{odd}\to 1,\ \mathtt{odd}\to+[\mathtt{odd},\mathtt{even}],\ \mathtt{odd}\to+[\mathtt{even},\mathtt{odd}],\\ \quad\ \ \mathtt{even}\to+[\mathtt{even},\mathtt{even}],\ \mathtt{even}\to+[\mathtt{odd},\mathtt{odd}]\ \},\\ \quad\mathtt{odd}\ ).\end{array}$$

Of course, for some classes of TREEBAG components there is no need to initialize objects in this way because they do not depend on additional information like a set of rules. A good example is the class `YIELDTransduction` which implements the(!) YIELD transduction. Since the way in which this transduction works does not depend on additional parameters its syntax is trivial in the sense that the parsing method just reads the empty string.

4 The TREEBAG worksheet

The TREEBAG worksheet is a user interface that allows to build interactively a network of instances of TREEBAG components. These instances are represented as the nodes of a graph in the worksheet window. It will be explained below how a node can be created. For the moment, let us assume that some nodes are already present. Then the user can interact with them as follows.

- Nodes can be rearranged by dragging them with the mouse.

- By clicking on a node once, moving the mouse pointer onto another node, and clicking once on that node, too, it is possible to establish edges between nodes. Roughly speaking, the existence of such an edge from v to v' indicates that v' receives its input from v. If v is a tree generator then v' can be a tree transducer or a display (with the obvious consequences). If v is an algebra then v' must be a display. In this case the connection determines how the display interprets the terms it receives as input from a tree generator. Attempts to establish a connection will be refused if v' is a tree grammar or an algebra (the two types of nodes that never accept input), if the connection would result in a cycle, or if v' is a display and v is an algebra of a kind that is not accepted by the type of display v' represents. Nodes that represent displays can only be the target of two edges at a time, one that connects it with a tree generator and one whose source is an algebra node. Establishing a new edge automatically removes an old one.
- By clicking on a node using the right mouse button[2] (and holding it down) one can invoke a menu whose items are the commands that the object provides in order to control its behaviour.
- A double-click on a display node opens a window showing the visualized objects. If the window has already been opened before, or the node is not a display node, the double-click opens a control panel for that node. The panel contains a number of buttons that serve the same purpose as the items on the menu described above.

A new node can be created by reading the definition of an object from a file. A file selector is opened by a double-click on a free area of the worksheet window. The syntax of an object definition is

<classname>`(`*<name>*`):` *<definition>*

where `(`*<name>*`)` is an optional part and *<classname>* is the name of the Java class that implements the type of objects to be loaded. If present, *<name>* is used as a name for the object. The name must be enclosed in quotation marks unless it is non-empty and consists entirely of digits, letters, '-', '+', '*', '.', and underscore characters.

The *<definition>* part uses the syntax of the class of objects being loaded. Thus, a node representing the regular tree grammar to generate terms denoting odd numbers (see Sections 2 and 3) could be created by reading a file that contains the definition

```
generators.regularTreeGrammar("odd numbers"):
  ( {even, odd},
    {1:0, +:2},
    {odd → 1, odd → +[odd, even], odd → +[even, odd],
     even → +[even, even], even → +[odd, odd] },
    odd ),
```

[2]This can in fact vary, depending on the window manager and the personal settings used. To be precise, one should speak of "the button used to invoke pop-up menus".

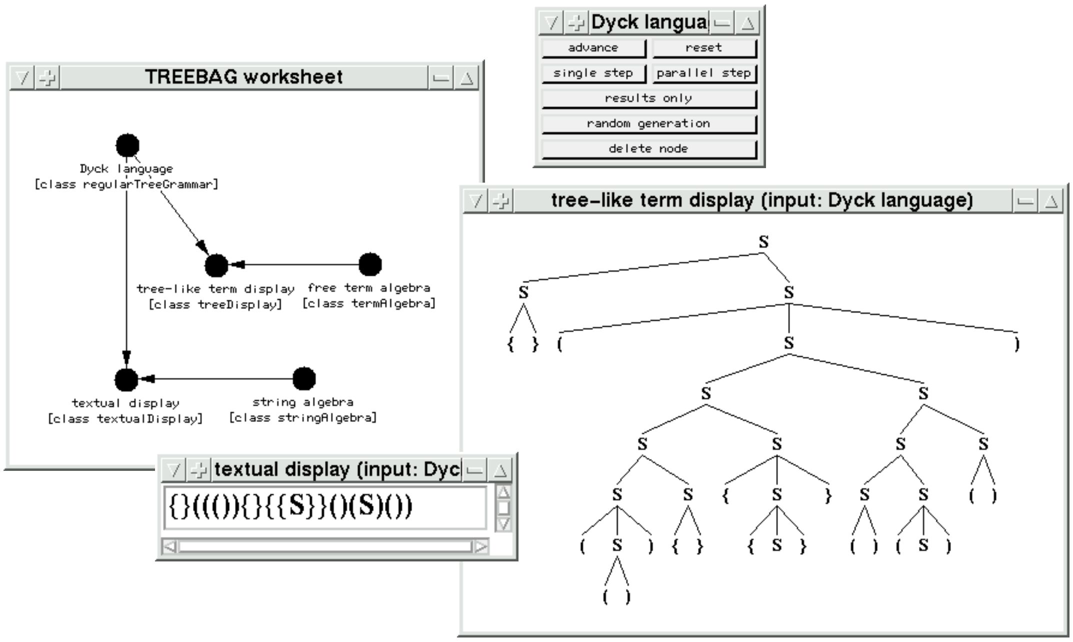

Figure 2: Generation of a Dyck language in TREEBAG

```
generators.regularTreeGrammar("Dyck language"):
  ( { S },
    { "(":0, ")":0, "{":0, "}":0, S:2, S:3 },
    { S -> S[S,S],
      S -> S["(",S,")"], S -> S["(",")"],
      S -> S["{",S,"}"], S -> S["{","}"] },
    S )
```

Figure 3: The definition of the regular tree grammar used in Figure 2

since `generators.regularTreeGrammar` is the full name of the class implementing regular tree grammars.

5 Some examples

In this section examples are discussed that show some of the possibilities TREEBAG provides. Of course, many other combinations of tree generators, algebras, and displays are possible.

5.1 Context-free string languages

By a combination of an appropriate regular tree grammar, a tree display in connection with a free term algebra, and a textual display together with a string algebra, one can use TREEBAG in order to visualize derivations in context-free string grammars. Figure 2 shows a snapshot of such a situation, where the language generated is the Dyck language with two different kinds of parenthesis (cf. [HU79]), namely '(', ')', and '{', '}'. The panel containing the

```
generators.tdTransducer("some function on strings"):
  ( { "(":0, ")":0, "{":0, "}":0, S:2, S:3 },
    { +:2, -:2, 0:0, 1:0 },
    { start, round, curly },
    { start[S[x1,x2]] -> -[+[round[x1],round[x2]],+[curly[x1],curly[x2]]],
      start[S[x1,x2,x3]] -> -[+[round[x1],round[x2]],+[curly[x1],curly[x2]]],
      round["("] -> 1,
      round["{"] -> 0, round[")"] -> 0, round["}"] -> 0,
      round[S[x1,x2]] -> +[round[x1],round[x2]],
      round[S[x1,x2,x3]] -> +[round[x1],round[x2]],
      curly["{"] -> 1,
      curly["("] -> 0, curly[")"] -> 0, curly["}"] -> 0,
      curly[S[x1,x2]] -> +[curly[x1],curly[x2]],
      curly[S[x1,x2,x3]] -> +[curly[x1],curly[x2]] },
    start )
```

Figure 4: The definition of the top-down tree transducer used in Figure 5

buttons `advance`, `reset`, etc. shows the commands one may use to control the regular tree grammar. The implementation of regular tree grammars allows to switch between different modes. The most basic choice is that between an enumeration of the tree language and a random derivation. In the enumeration mode one can furthermore choose between a mode that yields only the resulting (i.e., terminal) terms and a mode (shown in the figure) in which the current term in the enumeration is derived step by step.

In Figure 3 the definition of the regular tree grammar can be seen. Symbol names must be enclosed in quotation marks—except for non-empty strings consisting entirely of letters, digits, '+', '-', '*', '.', and underscore characters. The syntax of the algebras (the class names are `algebras.termAlgebra` and `algebras.stringAlgebra`) and of the displays (class names `displays.treeDisplay` and `displays.textualDisplay`) is trivial.

Of course, without the textual display and the attached string algebra this example can also be considered as a demonstration of regular tree grammars and their derivation process.

5.2 Top-down tree transductions

One can enrich the situation discussed in the previous example by adding a top-down tree transducer that computes some kind of function (together with an appropriate algebra and a display). In Figure 4 the definition of such a tree transducer is shown. It computes the difference between the number of occurrences of the two pairs of parentheses in the generated string. The algebra used is an integer algebra (`algebras.intAlgebra`) that provides some of the basic operations on integer numbers.

The tree transducer shown in Figure 5 has been switched to `stepwise` mode and the derivation is not yet finished. Therefore, the integer algebra cannot interpret the term, which means that there is no value the display could show. This is indicated by the string `<no value>`. As soon as the derivation has been completed (or the user has pressed `results only`) the display will show the value of the computed term.

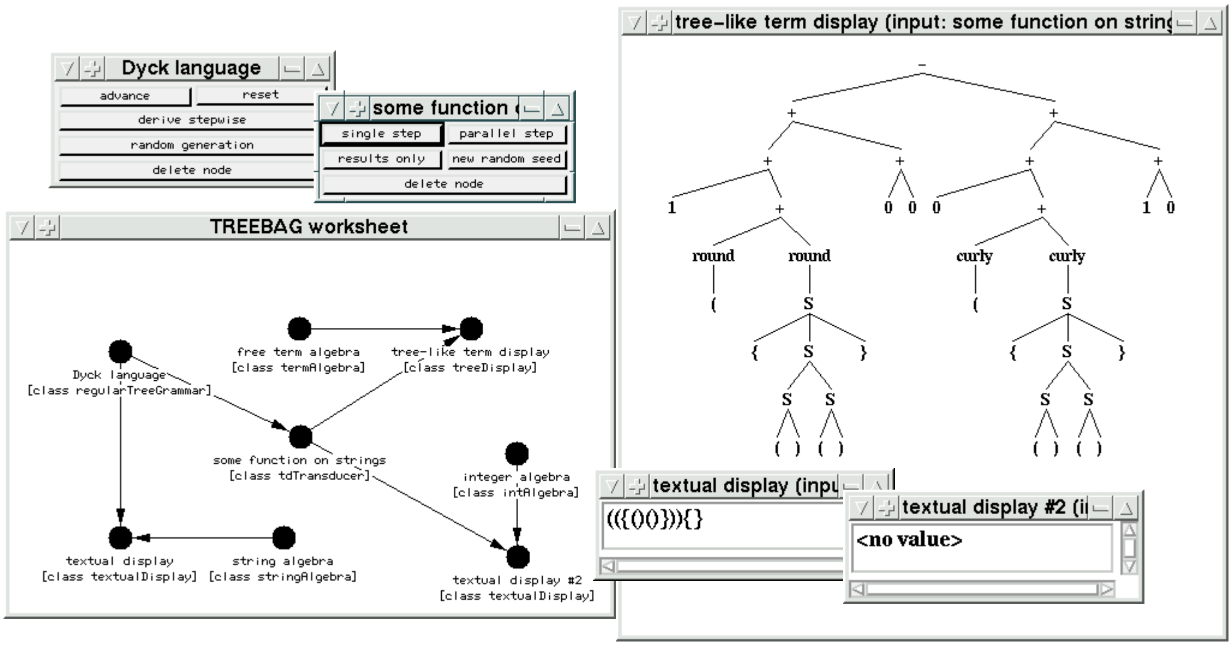

Figure 5: A top-down tree transducer at work

5.3 L systems

ET0L systems (cf., e.g., [Roz73, KRS97]) can be simulated by top-down tree transducers that take as input the terms generated by a regular tree grammar whose output signature is monadic (cf. [ERS80]). Roughly speaking, the symbols in an input term represent the tables used in the generation process. An example is given in Figure 6. It shows how to generate the set of all strings of the form $wc^{n_1}wc^{n_2}\cdots wc^{n_m}$, where $w \in \{a,b\}^*$ and $n_i \leq |w|$ for $1 \leq i \leq m$. The definitions of the regular tree grammar and of the top-down tree transducer are shown in Figure 7. The corresponding ET0L system consists of the terminal symbols a, b, nonterminals $<\mathit{start}>$, $<\mathit{AB}>$, and $<\mathit{AB'}>$, and tables

$$\begin{aligned}
&\{<\mathit{start}> \to <\mathit{AB}><\mathit{start}>,\ <\mathit{AB}> \to <\mathit{AB}>\},\\
&\{<\mathit{start}> \to \lambda,\ <\mathit{AB}> \to <\mathit{AB'}>\},\\
&\{<\mathit{AB'}> \to a<\mathit{AB'}>c,\ <\mathit{AB'}> \to a<\mathit{AB'}>\},\\
&\{<\mathit{AB'}> \to b<\mathit{AB'}>c,\ <\mathit{AB'}> \to b<\mathit{AB'}>\},\\
&\{<\mathit{AB'}> \to \lambda\}.
\end{aligned}$$

The current implementation of top-down tree transducers deals with non-determinism by choosing randomly, for a given redex, one of the matching rules.[3] Subsequent computations use the same initial value of the random number generator until the command `new random seed` is invoked. This turns out to be convenient in examples such as the present one, because it means that one can visualize a derivation of the L system by choosing one of the modes

[3]Admittedly, this way of dealing with non-determinism is somewhat inappropriate; a more sophisticated implementation should provide a mode that yields, for every input term, an enumeration of the derivable output terms.

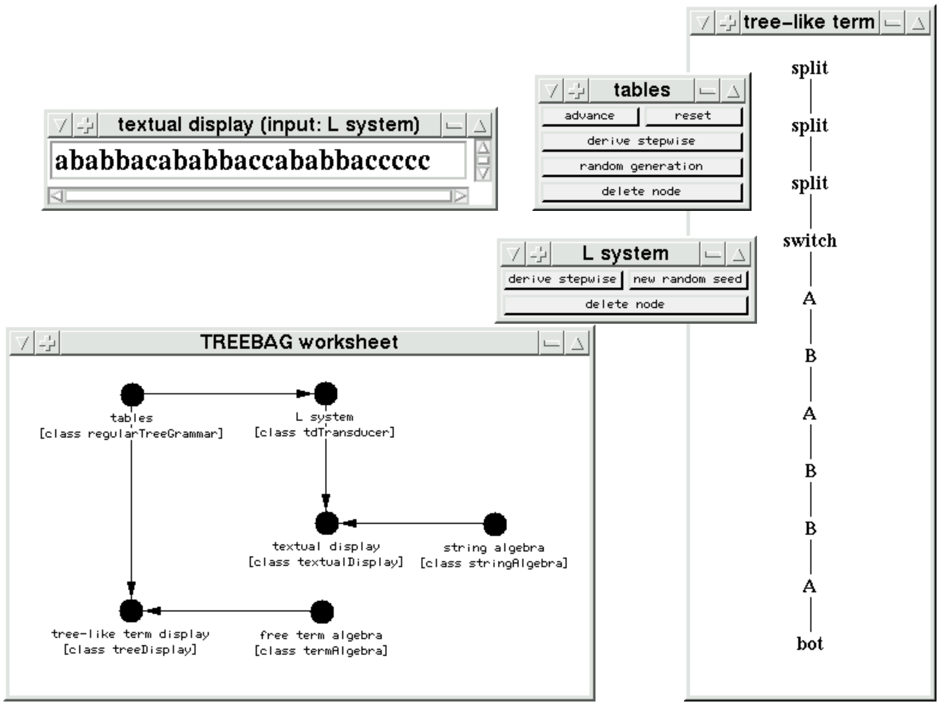

Figure 6: Simulating an L system

```
generators.regularTreeGrammar("tables"):
  ( { split, gen },
    { split:1, switch:1, A:1, B:1, bot:0 },
    { split -> split[split], split -> switch[gen],
      gen -> A[gen], gen -> B[gen], gen -> bot },
    div )

generators.tdTransducer("L system"):
  ( { split:1, switch:1, A:1, B:1, bot:0 },
    { .:2, a:0, b:0, c:0, "":0 },
    { start, AB },
    { start[split[x1]] -> (AB[x1] . start[x1]), start[switch[x1]] -> "",
      AB[split[x1]] -> AB[x1], AB[switch[x1]] -> AB[x1],
      AB[A[x1]] -> (a . (AB[x1] . c)), AB[A[x1]] -> (a . AB[x1]),
      AB[B[x1]] -> (b . (AB[x1] . c)), AB[B[x1]] -> (b . AB[x1]),
      AB[bot] -> "" },
    start )
```

Figure 7: The regular tree grammar and the top-down tree transducer used in Figure 6. Note that, as one can see in the definition of the top-down tree transducer, the TREEBAG syntax of terms allows to use infix notation for binary symbols: $(t_1\, op\, t_2)$ is equivalent to op[t_1 , t_2].

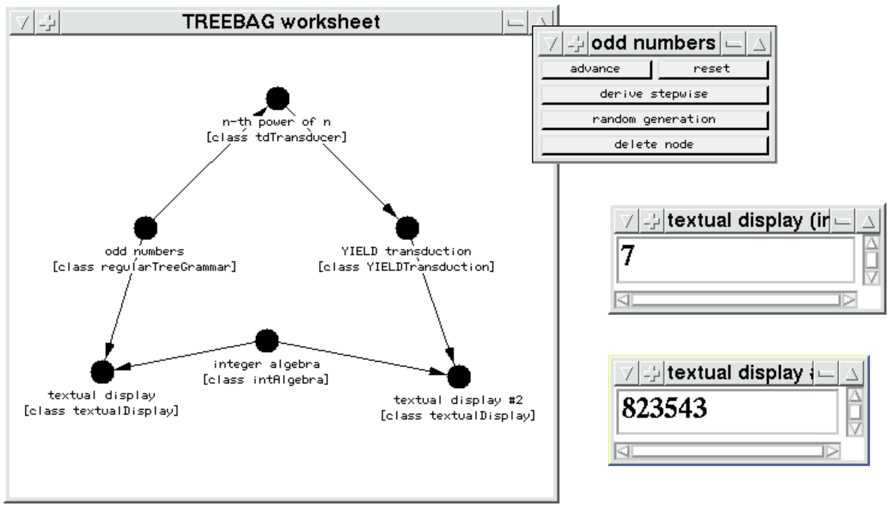

Figure 8: Computing n^n using the YIELD transduction

`derive stepwise` or `random generation` of the regular tree grammar and extending the top-down tree transducer by additional rules

$$\begin{array}{lcl} \mathit{start}[\mathit{split}] & \to & <\mathit{start}>, \\ AB[\mathit{split}] & \to & <AB>, \text{ and} \\ AB[\mathit{gen}] & \to & <AB'>. \end{array}$$

Here, $\mathit{split}\colon 0$, $\mathit{gen}\colon 0$, $<\mathit{start}>\colon 0$, $<AB>\colon 0$, and $<AB'>\colon 0$ are new input and output symbols, respectively, corresponding to the nonterminals of the regular tree grammar and of the simulated L system. Now, using the (parallel) stepwise mode of the regular tree grammar (or the random derivation mode), the textual display shows the derivations of the ET0L system given above.

5.4 The YIELD transduction

By composing top-down tree transductions with the YIELD transduction one can express many transductions that cannot be expressed by (compositions of) top-down tree transductions alone (cf. [EV85]). As an example, consider Figures 8 and 9. The regular tree grammar generates (under the usual interpretation of symbols) all odd natural numbers, denoted by terms over $+\colon 2$ and $1\colon 0$, as discussed in Section 2. Now, the composition of the top-down tree transduction whose definition is shown in Figure 9 with the YIELD transduction (`generators.YIELDTransduction`) makes it possible to transform a term denoting the number n into a term denoting n^n using only addition and multiplication. In particular, no exponentiation symbols are required. The idea is that, on input t, the composed transduction yields the term $t_*[\![t]\!]$, where t_* is obtained from t by replacing all occurrences of $+$ with $*$ and all occurrences of 1 with x_1. Thus, the value of $t_*[\![t]\!]$ is n^n if t denotes n.

```
generators.tdTransducer("n-th power of n"):
  ( { +:2, 1:0 },
    { subst:2, subst:3, proj-1:0, const-+-2:0, const-*-2:0, const-1-0:0 },
    { start, prod, copy },
    { start[1] -> const-1-0,
      start[+[x1,x2]] -> subst[subst[const-*-2,prod[x1],prod[x2]],
                               subst[const-+-2,copy[x1],copy[x2]]],
      prod[1] -> proj-1,
      copy[1] -> const-1-0,
      prod[+[x1,x2]] -> subst[const-*-2,prod[x1],prod[x2]],
      copy[+[x1,x2]] -> subst[const-+-2,copy[x1],copy[x2]] },
    start )
```

Figure 9: The definition of the top-down tree transducer used in Figure 8

5.5 Turtle and chain-code pictures

The turtle and the chain-code algebras (`applications.lineDrawings.turtleAlgebra` and `applications.lineDrawings.chainCodeAlgebra`) allow to generate line drawings as is usually done by the use of ET0L systems with turtle interpretation (cf. [PL90]), or by using context-free grammars with the chain-code interpretation of strings [MRW82]. In both cases, if the interpretation of strings is turned into a set of operations the central operation turns out to be the concatenation of line drawings, where a line drawing is a picture composed of unit lines that has a distinguished end point (and, implicitly, also a starting point, namely the origin). Therefore, except for the symbols which have a special meaning, all symbols are interpreted as n-ary concatenation of line drawings (where n is the rank of the respective symbol). In case of the chain-code algebra, which has a trivial syntax, the additional operations are given by the interpretation of the symbols $n\colon 0$, $e\colon 0$, $s\colon 0$, and $w\colon 0$. These symbols are interpreted as single lines extending from the origin one unit up, to the right, down, and to the left, respectively.

Turtle algebras provide somewhat more flexible operations. The symbols that have a special meaning (i.e., are not interpreted as concatenation operations) are the following ones. $F\colon 0$ is interpreted like $n\colon 0$ above, whereas $f\colon 0$ is its "invisible" counterpart, i.e., the empty picture whose end point is $(0,1)$. The operations associated with the symbols $+\colon 1$ and $-\colon 1$ turn their argument about the origin by α and $-\alpha$ degrees, respectively, and $enc\colon 1$ ("encapsulate") returns its argument unchanged, except that the end point is replaced by $(0,0)$. Furthermore, $+branch\colon 1$ and $-branch\colon 1$ rotate the picture like $+$ and $-$ do, but replace the end point by $(0,0)$, in addition. Thus, $+branch[t]$ and $-branch[t]$ can be seen as shorthands for $enc[+[t]]$ and $enc[-[t]]$, respectively).

For convenience, turtle algebras also allow to use the symbols $n{+}\colon 1$, $n{-}\colon 1$, $n{+}branch\colon 1$, and $n{-}branch\colon 1$, where n is a natural number. These symbols are interpreted like $+\colon 1$, $-\colon 1$, $+branch\colon 1$, and $-branch\colon 1$, except that the angle $n\alpha$ is used instead of α.

The syntax of turtle algebras is not trivial, but is nevertheless very simple. I just consists in a rational number to the base of ten (i.e., a non-empty string over digits, possibly containing

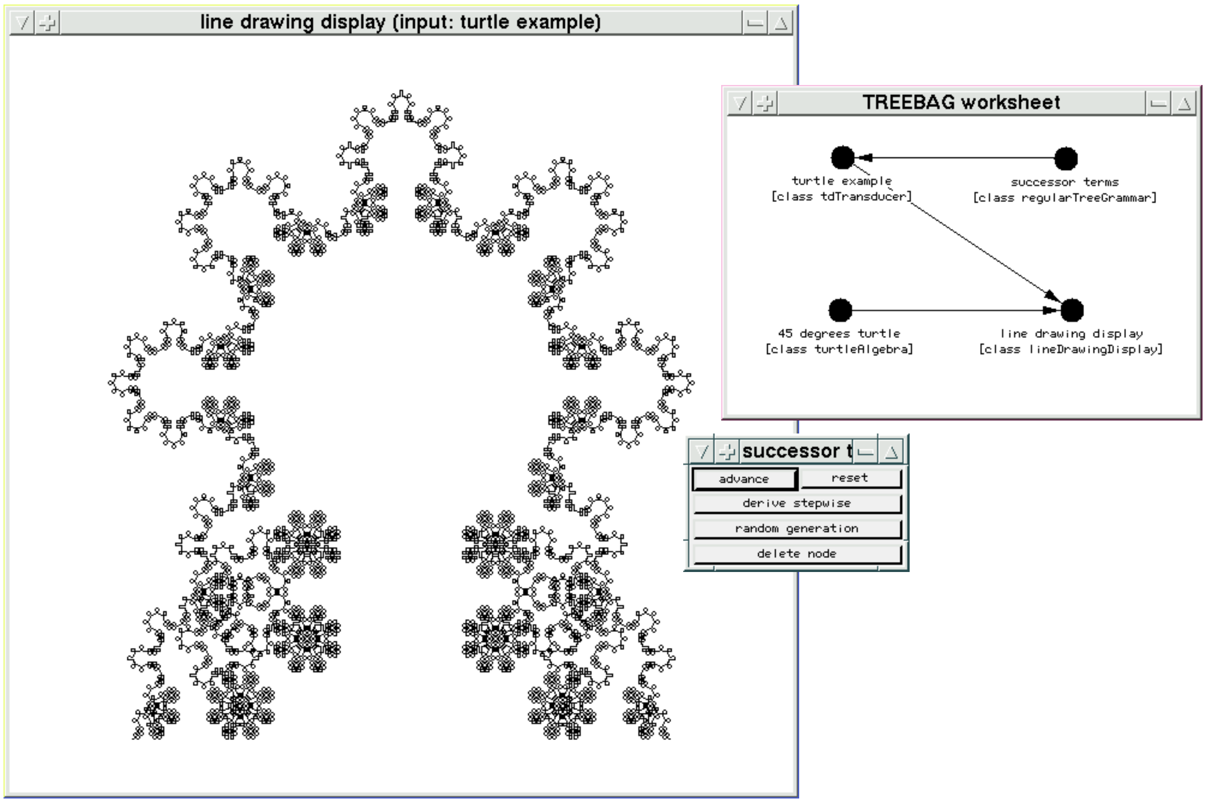

Figure 10: A picture produced by the use of turtle operations

a decimal point) which yields the angle α by which $+$, $-$, $+branch$, and $-branch$ turn their arguments.

An example is shown in Figure 10. It should be noticed that, as discussed in Example 5.3, in order to mimic the parallel mode of rewriting known from L systems the terms that yield the line drawings are not directly generated by a regular tree grammar. Instead, the top-down tree transducer is used whose definition is shown in Figure 11 and whose input terms are those in $\mathrm{T}_{\Sigma_\mathrm{s}}$, where $\Sigma_\mathrm{s} = \{s\colon 1, 0\colon 0\}$. Thus, an input term to the tree transducer is just a chain of s symbols and the length of this chain determines the degree of refinement of the picture (which is the same in all places). The resulting drawings are visualized by a line-drawing display (`applications.lineDrawings.lineDrawingDisplay`). These displays take care for an automatic scaling and positioning of displayed pictures, so that they always fit into the window (which is useful because line drawings tend to become very large).

5.6 Pictures generated by collage grammars and by iterated function systems

A collage grammar is a context-free device that generates pictures, each consisting of a finite number of d-dimensional geometric objects (see [HK91] and, for the formulation using regular tree grammars, [Dre96a, Dre96b]). For $d = 2$, this can be visualized in Treebag using collage algebras (`applications.collages.collageAlgebra`) and collage displays (`applications.collages.collageDisplay`), together with a regular tree grammar. A collage algebra provides symbols of rank 0 that denote collages, and symbols $F\colon n$ that denote

```
generators.tdTransducer("turtle example"):
  ( { s:1, 0:0 },
    { F:0, empty:0, conc:13, +:1, 2+:1, 3+:1, -:1, 2-:1, 3-:1 },
    { start, A },
    { start[0] -> empty,
      start[s[x1]] -> -[-[A[x1]]],
      A[0] -> F,
      A[s[x1]] ->
        conc[A[x1],+[A[x1]],2+[A[x1]],3+[A[x1]],2+[A[x1]],+[A[x1]],
             A[x1],-[A[x1]],2-[A[x1]],3-[A[x1]],2-[A[x1]],-[A[x1]],A[x1]] },
    start )
```

Figure 11: The definition of the top-down tree transducer used in Figure 10

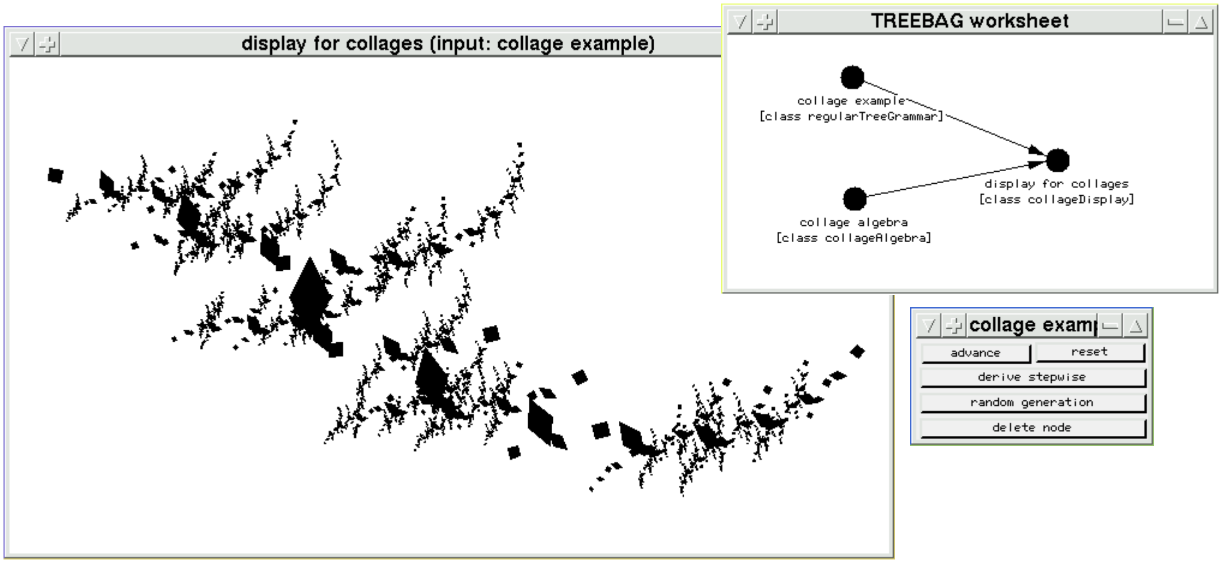

Figure 12: A collage grammar in Treebag

operations $<f_1,\ldots,f_n>$ which are given by n affine transformations $f_1,\ldots,f_n$ in such a way that $<f_1,\ldots,f_n>(C_1,\ldots,C_n) = f_1(C_1) \cup \cdots \cup f_n(C_n)$ for collages $C_1,\ldots,C_n$ (where $f_1,\ldots,f_n$ are extended to collages in the canonical way).

The syntactic definition of a collage algebra will not be discussed in detail, here, although it is not very complicated (see Figure 13 for an example). Roughly speaking, it consists of a list of definitions that associate symbol names with collages, transformations, and operations $<f_1,\ldots,f_n>$, where $f_1,\ldots,f_n$ are the names of previously defined transformations. Figures 12 and 13 show an example (the one used also in the introduction) and the corresponding definitions of the regular tree grammar and the collage algebra. As opposed to line-drawing displays, collage displays do not automatically scale or translate displayed pictures. Instead, the user can zoom in and out, and can drag the picture using the mouse.

It should be noticed that an easy extension of the collage algebra allows to visualize the derivations in this collage grammar in a nice pictorial way. For this, one just has to assign a

```
generators.regularTreeGrammar("collage example"):
  ( { S },
    { F:3, C:0 },
    { S -> C, S -> F[C,S,S] },
    S )

applications.collages.collageAlgebra("collage algebra"):
  { reduce-x = scale(.5,1),
    right    = scale(.8).rotate(12).translate(3,-2),
    left     = scale(.6).rotate(12).translate(-3,2),
    C = { filledPolygon((0,-1),(-1,0),(0,1),(1,0)) },
    F = <reduce-x,right,left> }
```

Figure 13: The regular tree grammar and the collage algebra used in Figure 12

meaning to the nonterminal S of the regular tree grammar in order to make it visible. One could, for instance, add to the definition of the algebra the following lines:

```
S = { polygon((-1,-1),(1,-1),(1,1),(-1,1)),
      polyline((-1,-1),(1,1)),
      polyline((1,-1),(-1,1)) },
```

so that the non-terminal trees generated by the regular tree grammar are interpreted as shown in Figure 14.

In the same way as in Section 5.5 one can simulate a more controlled type of rewriting than the context-free one of collage grammars by taking, instead of a regular tree grammar, a top-down tree transducer with input terms in T_{Σ_s}. In fact, this yields a class of systems equivalent to hierarchical (or: mutually recursive) function systems with condensation (cf. [Bar88, PJS92]). The result of applying this modification to the situation of Figure 12 is shown in Figure 15; the definition of the tree transducer can be seen in Figure 16. In addition, a textual display with an attached integer algebra (which interprets s as the successor function) has been added in order to display the depth of refinement.

5.7 Generation of tilings

Collage algebras in connection with top-down tree transducers can also be used in order to tile the plane (i.e., to generate a sequence of collages whose limit is a tiling). Of course, periodic tilings are easy to produce, but non-periodic ones may also be generated. A particularly beautiful[4] example from [GS87, Figure 9.5.5] (see also [Pen89]) with an apparently spiral structure is shown in Figure 17. The top-down tree transducer and the collage algebra are given in Figure 18 and Figure 19. (In the definition of collage algebras the triples in square brackets that follow the definitions of filled polygons are optional and denote colours in rgb notation. The default colour is black.)

[4] a matter of taste, of course ...

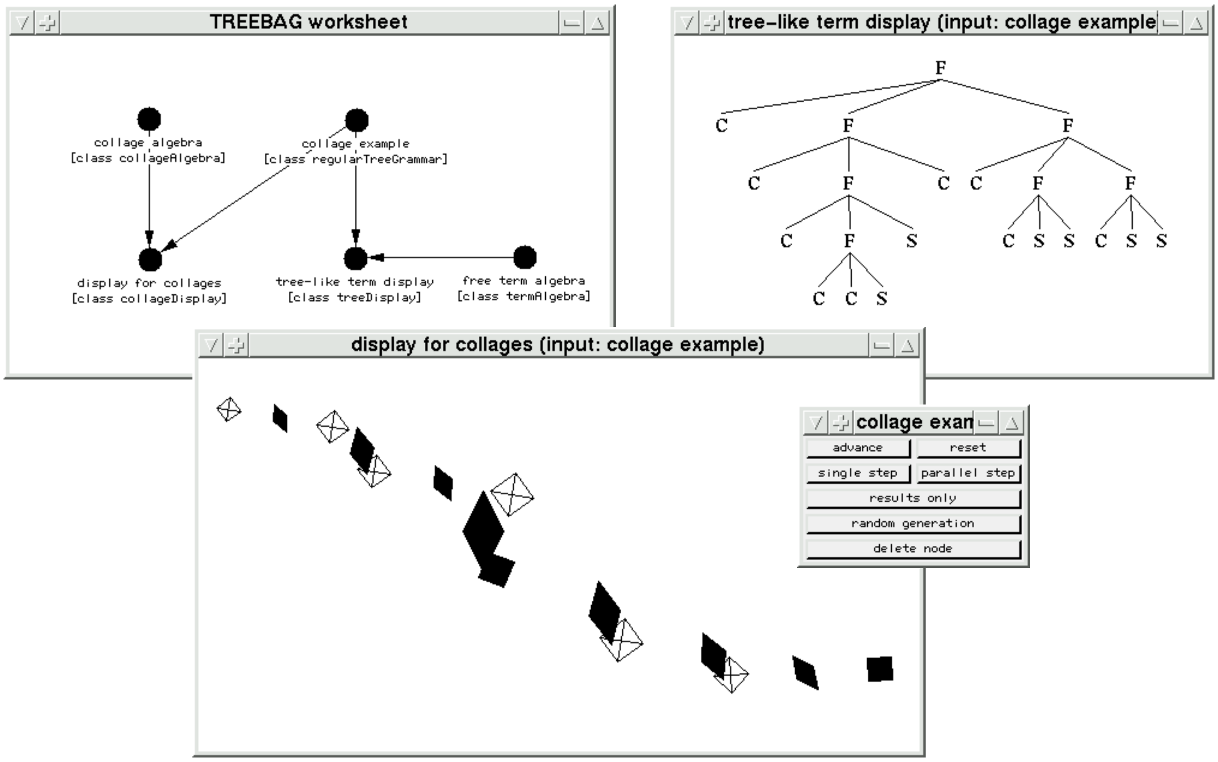

Figure 14: The example of Figure 12, where an interpretation of nonterminals has been added

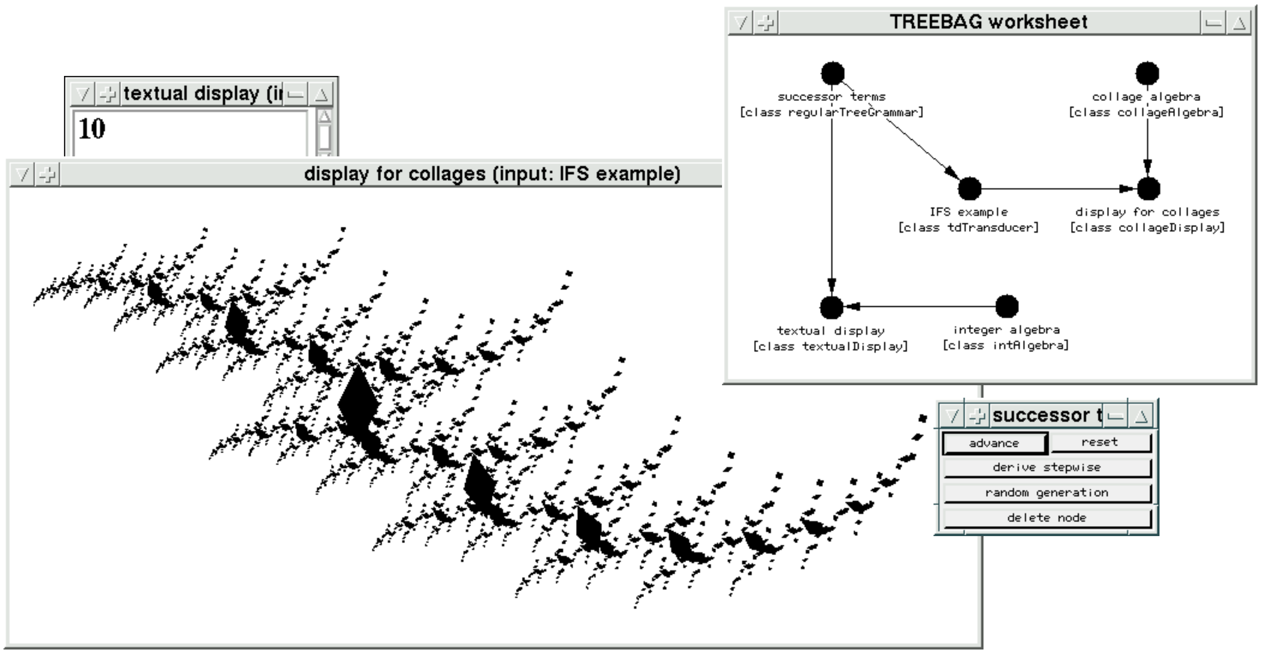

Figure 15: The example of Figure 12, turned into an iterated function system

```
generators.tdTransducer("IFS example"):
  ( { s:1, 0:0 },
    { F:3, C:0 },
    { S },
    { S[0] -> C,
      S[s[x1]] -> F[C,S[x1],S[x1]] },
    S )
```

Figure 16: The definition of the top-down tree transducer used in Figure 15

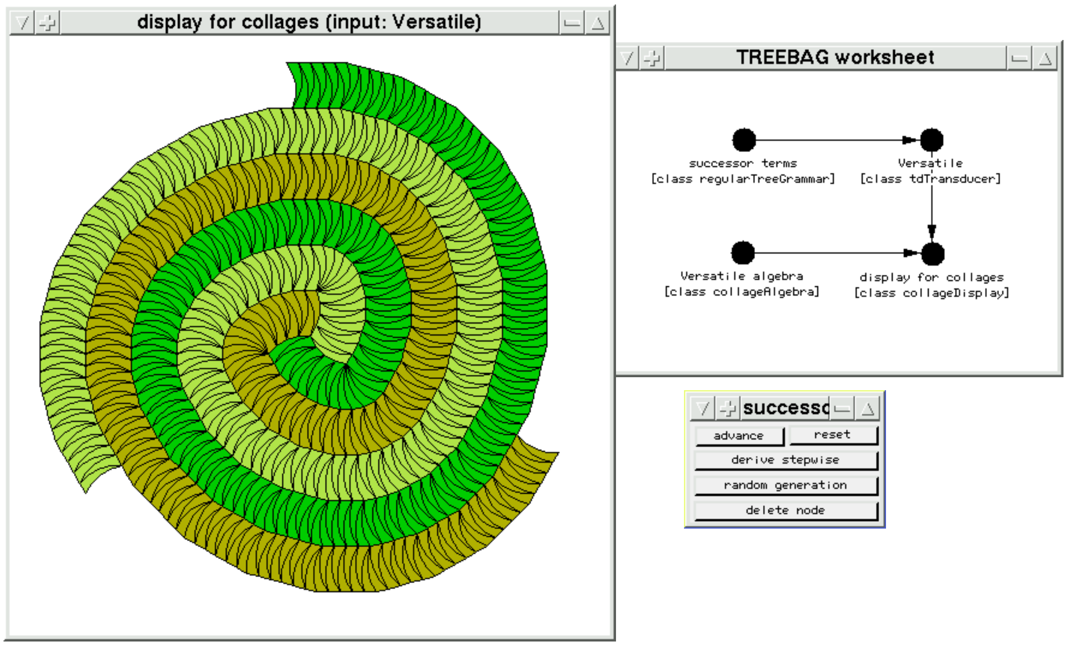

Figure 17: Generating a tiling

6 Future work

Due to its open architecture TREEBAG is an easily extensible system. Thus, there are some rather obvious things that could be done in the future.

1. Other classes of tree generators and improved implementations of the presently existing ones should be provided. For example, bottom-up tree transductions and macro tree transductions are still missing. In particular, it would be nice to have macro tree transducers [EV85] available, despite the fact that a macro tree transduction is more or less the same as the composition of a suitable top-down tree transducer with the YIELD transduction, and the composition closures of these two classes are equal (see the discussion at the end of Section 2). This is because macro tree transducers provide a more direct and intuitive way to define certain tree transductions (whereas, for a theoretical investigation, it is sometimes more convenient to decompose them into top-down and YIELD transductions). Another advantage would of course be that such

```
generators.tdTransducer("Versatile"):
  ( { s:1, 0:0 },
    { tileR:0, tileG:0, tileB:0,
      +:2, rot:1, *:2, mir:1, up:2, up2:3, rot120:1, comp:4 },
    { start, red, green, blue, redA, redB, greenA, greenB, blueA, blueB },
    { start[s[x1]] -> +[red[x1],rot120[+[green[x1],rot120[blue[x1]]]]],
      red[0] -> mir[tileR],
      red[s[x1]] ->
       comp[redA[x1],tileR,redB[x1],
        rot[comp[redA[x1],tileR,redB[x1],
         rot[comp[redA[x1],tileR,redB[x1],
          rot[comp[redA[x1],tileR,redB[x1],
           (redB[x1]*(redB[x1]*(redB[x1]*(redB[x1]*rot[redA[x1]]))))
       ]]]]]]],
      green[0] -> mir[tileG],
      green[s[x1]] ->
       comp[greenA[x1],tileG,greenB[x1],
        rot[comp[greenA[x1],tileG,greenB[x1],
         rot[comp[greenA[x1],tileG,greenB[x1],
          rot[comp[greenA[x1],tileG,greenB[x1],
           (greenB[x1]*(greenB[x1]*(greenB[x1]*(greenB[x1]*rot[greenA[x1]]))))
       ]]]]]]],
      blue[0] -> mir[tileB],
      blue[s[x1]] ->
       comp[blueA[x1],tileB,blueB[x1],
        rot[comp[blueA[x1],tileB,blueB[x1],
         rot[comp[blueA[x1],tileB,blueB[x1],
          rot[comp[blueA[x1],tileB,blueB[x1],
           (blueB[x1]*(blueB[x1]*(blueB[x1]*(blueB[x1]*rot[blueA[x1]]))))
       ]]]]]]],
      redA[0] -> mir[tileR],
      redB[0] -> mir[tileR],
      greenA[0] -> mir[tileG],
      greenB[0] -> mir[tileG],
      blueA[0] -> mir[tileB],
      blueB[0] -> mir[tileB],
      redA[s[x1]] -> (mir[tileR]+up[tileB,blueA[x1]]),
      redB[s[x1]] -> (mir[tileR]+up2[tileB,blueB[x1],blueA[x1]]),
      greenA[s[x1]] -> (mir[tileG]+up[tileR,redA[x1]]),
      greenB[s[x1]] -> (mir[tileG]+up2[tileR,redB[x1],redA[x1]]),
      blueA[s[x1]] -> (mir[tileB]+up[tileG,greenA[x1]]),
      blueB[s[x1]] -> (mir[tileB]+up2[tileG,greenB[x1],greenA[x1]]) },
    start )
```

Figure 18: The top-down tree transducer used in Figure 17

```
applications.collages.collageAlgebra("Versatile algebra"):
 {tileR
   = {filledPolygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866))[.7,.7,0],
      polygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866)) },
  tileG
   = {filledPolygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866))[0,.8,0],
      polygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866)) },
  tileB
   = {filledPolygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866))[.7,.9,.3],
      polygon( (0,0), (1,0), (1.2588,0.9659), (1.2588,1.9659), (1,2.932),
        (0.5,3.7979), (0.7588,2.832), (0.7588,1.832), (0.5,0.866)) },

  rot    = rotate(15),
  rot    = <rot>,
  *      = <rot,rot>,

  mir    = translate(-.5,-3.7979).scale(1,-1),
  mir    = <mir>,

  up     = translate(-.5,3.7979),
  up2    = rotate(15).translate(-.5,3.7979),
  up     = <up,up>,,
  up2    = <up,up,up2>,

  id     = scale(1),
  left   = translate(-1,0),
  comp   = <id,left,left,left>,

  rot120 = rotate(120),
  rot120 = <rot120> }
```

Figure 19: The top-down tree transducer used in Figure 17

classes could provide ways to execute derivations step by step in order to demonstrate how macro tree transductions work.

2. One of the things that should be quite easy is the extension of collage algebras to three dimensions. However, one also needs a corresponding class of displays. Therefore, it seems a good idea to wait until there exist suitable Java class libraries for three-dimensional graphics that can be used to build such displays without too much effort.
3. Another aim is the implementation of context-free graph grammars. For this, algebras like those corresponding to hyperedge or node replacement (cf. [Cou91, Eng94, Dre96a]) as well as a class of displays for graphs would be needed. While the implementation of the mentioned graph algebras is a rather easy task it is somewhat difficult to provide good displays for graphs because this involves computing layouts. However, since quite powerful graph layout tools exist it may be possible to avoid the difficult part, at least in a first step, by building displays that make use of such a tool.

Later on, it may nevertheless be worthwhile to think about layout algorithms tailored towards the special cases of graphs produced by hyperedge replacement or node replacement operations since these are very special ways to generate graphs, that enjoy several nice properties (cf. [DHK97, Eng97, ER97]). In fact, it may be a good idea to think about such algorithms in connection with algebras that yield as values graphs containing some sort of initial layout information.

4. Finally, one should aim at more sophisticated ways of user interaction in TREEBAG. In particular, future versions of the system should allow to control derivations interactively. The challenging question is to find a general way to let the user select parts of a displayed object and to relate the selected parts to subterms of the term from which the object was obtained. If a display shows a term as a tree, users should, for instance, have the possibility to select a nonterminal symbol in order to apply a rule, whereas in a collage display it should be possible to select a geometric part that represents a nonterminal in the corresponding term.

   Of course, such an extension should avoid to violate the TREEBAG structure by adding new concepts in an ad-hoc manner. A way to satisfy this requirement could be to use algebras that yield objects in which certain parts are annotated with occurrences $o \in \mathbb{N}^*$ indicating their relation to subterms of the term from which the object was obtained. Then, a display could provide means to select such parts. Similar to the concept of commands as it is available in the current version it would be left to the user interface to take care for such selections by, for example, opening a menu that shows the commands the respective tree generator would recognize (like, e.g., a set of applicable rules).

References

[Bar88] Michael Barnsley. *Fractals Everywhere*. Academic Press, Boston, 1988.

[Cou91] Bruno Courcelle. Recursive queries and context-free graph grammars. *Theoretical Computer Science*, 78:217–244, 1991.

[Dam82] Werner Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20:95–208, 1982.

[DE98] Frank Drewes and Joost Engelfriet. Decidability of the finiteness of ranges of tree transductions. *Information and Computation*, 1998. To appear.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

[Dre96a] Frank Drewes. Computation by tree transductions. Doctoral dissertation, University of Bremen, Germany, 1996.

[Dre96b] Frank Drewes. Language theoretic and algorithmic properties of d-dimensional collages and patterns in a grid. *Journal of Computer and System Sciences*, 53:33–60, 1996.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations—a comparison. *Mathematical Systems Theory*, 9(3):198–231, 1975.

[Eng80] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 241–286. Academic Press, New York, 1980.

[Eng94] Joost Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Proc. CAAP 94*, volume 787 of *Lecture Notes in Computer Science*, pages 15–37. Springer, 1994.

[Eng97] Joost Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 3, pages 125–213. Springer, 1997.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 1, pages 1–94. World Scientific, 1997.

[ERS80] Joost Engelfriet, Grzegorz Rozenberg, and Giora Slutzki. Tree transducers, L systems, and two-way machines. *Journal of Computer and System Sciences*, 20:150–202, 1980.

[ES77] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. I. *Journal of Computer and System Sciences*, 15:328–353, 1977.

[ES78] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. II. *Journal of Computer and System Sciences*, 16:67–99, 1978.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

[EV88] Joost Engelfriet and Heiko Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26:131–192, 1988.

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

[GS87] Branko Grünbaum and G.C. Shephard. *Tilings and Patterns.* Freeman, New York, 1987.

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

[HK91] Annegret Habel and Hans-Jörg Kreowski. Collage grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 532 of *Lecture Notes in Computer Science*, pages 411–429. Springer, 1991.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison–Wesley, Reading, Massachusetts, 1979.

[KRS97] Lila Kari, Grzegorz Rozenberg, and Arto Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. I: Word, Language, Grammar*, chapter 5, pages 253–328. Springer, 1997.

[Mai74] Tom S.E. Maibaum. A generalized approach to formal languages. *Journal of Computer and System Sciences*, 8:409–502, 1974.

[MRW82] H.A. Maurer, G. Rozenberg, and E. Welzl. Using string languages to describe picture languages. *Information and Control*, 54:155–185, 1982.

[Pen89] Roger Penrose. *The Emperor's New Mind. Concerning Computers, Minds, and the Laws of Physics.* Oxford University Press, Oxford, 1989.

[PJS92] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals. New Frontiers of Science.* Springer-Verlag, New York, 1992.

[PL90] Przemyslaw Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, New York, 1990.

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

[Roz73] Grzegorz Rozenberg. Extensions of tabled 0L systems and languages. *International Journal of Computer and Information Sciences*, 2:311–334, 1973.

[Tha70a] James W. Thatcher. Generalized^2 sequential machine maps. *Journal of Computer and System Sciences*, 4:339–367, 1970.

[Tha70b] James W. Thatcher. There's a lot more to finite automata theory than you would have thought. In *Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems*, pages 263–276, 1970. Revised version published as [Tha73].

[Tha73] James W. Thatcher. Tree automata: an informal survey. In A.V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172. Prentice Hall, 1973.