
The TREEBAG Manual

– Version 1.6 –

Frank Drewes

Department of Computing Science, Umeå University,
S-901 87 Umeå, Sweden
e-mail: drewes@cs.umu.se

This document describes the use of TREEBAG, a system implemented in Java that allows to generate and transform objects using tree grammars and tree transducers. The basic ideas underlying TREEBAG are briefly summarised in Section 1. In Section 2 the TREEBAG worksheet—the main window of the system—is described. Section 3 explains the syntax of files needed to define worksheet configurations and TREEBAG components. In Section 4 the currently implemented classes of TREEBAG components are described in detail.

The current version of TREEBAG has been implemented using Java 2. It will probably not work if you use an interpreter based on an older version of Java. The latest TREEBAG version is always available at <http://www.cs.umu.se/~drewes/treebag>, the TREEBAG home page which is updated whenever the implementation is improved or extended. Changes usually concern small improvements or additions that are not reflected in the version number. Therefore, please make sure that you always use the version of this manual that corresponds to your TREEBAG version, in order to avoid inconsistencies.

Attention Users who are familiar with previous TREEBAG versions should notice an important (and, unfortunately, incompatible) change. Version 1.3 differs from all previous versions with respect to the way in which components are connected on the worksheet. See Section 1 for the new conventions.

Contents

1	What is TREEBAG?	1
2	The TREEBAG worksheet	3
2.1	The worksheet and component commands	3
2.2	The editor	4
2.3	Creating, redirecting, and deleting edges	4
2.4	Dragging nodes around	4
2.5	Opening nodes	4
2.6	Commands provided by all component types	5
3	Syntax	5
3.1	Configuration files	5
3.2	Component definitions	7
3.2.1	Numbers	8
3.2.2	Names	8
3.2.3	Sets and tuples	8
3.2.4	Symbols, signatures, and trees	8
3.2.5	File inclusion	8
4	Classes of TREEBAG components	9
4.1	Tree grammars	9
4.1.1	Regular tree grammars	9
4.1.2	Parallel deterministic total tree grammars	10
4.1.3	ETOL tree grammars	11
4.1.4	Branching tree grammars	13
4.2	Tree transducers	15
4.2.1	Top-down tree transducers	15
4.2.2	YIELD mappings	17
4.2.3	Macro tree transducers	17
4.2.4	Iterators	18
4.3	Algebras	18
4.3.1	The boolean algebra	19
4.3.2	The chain-code algebra	19
4.3.3	Collage algebras	19
4.3.4	The geometric-world algebra	25
4.3.5	Grid collage algebras	25
4.3.6	The integer algebra	27
4.3.7	The string algebra	27
4.3.8	The free term algebra	28
4.3.9	Turtle algebras	28
4.3.10	YIELD algebras	29
4.4	Displays	29

4.4.1	Collage displays	29
4.4.2	The geometric-world display	31
4.4.3	Line-drawing displays	31
4.4.4	The textual display	31
4.4.5	Tree displays	31
A	Lists of available classes of TREEBAG components	34
A.1	Tree Grammars	34
A.2	Tree Transducers	34
A.3	Algebras	34
A.4	Displays	34

1 What is TREEBAG?

TREEBAG is a system that allows to generate and transform objects of several types. This is accomplished by generating and transforming trees (= terms) using *tree grammars* and *tree transducers*, and interpreting the resulting trees as expressions that denote objects of the desired type. For the latter, *algebras* are used, each algebra defining an abstract data type (i.e., a set of objects together with a number of operations on them). Finally, there are *displays* whose purpose is to visualise the generated objects on the screen. Tree grammars, tree transducers, algebras, and displays are the four types of *TREEBAG components* everything relies on. The system has been designed as a Java application, but can since version 1.5 also be started from a web browser as an applet.

In order to explain the way in which TREEBAG works, some basic concepts are required. A *tree* in the sense discussed here is a rooted and ordered tree whose nodes are labelled with symbols from a ranked alphabet (or *signature*). A node which is labelled with a symbol of rank n must have exactly n children. As an example, consider the signature $\Sigma = \{+:2, fac:1\} \cup \{c:0 \mid c \in \mathbb{N}\}$ which contains the symbols $+$ and fac of rank 2 and 1, respectively, and all natural numbers, each considered as a symbol of rank 0. One of the trees over this signature is shown in Figure 1. In a natural way such a tree can be regarded

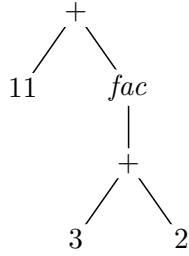


Figure 1: A tree over the signature $\Sigma = \{+:2, fac:1\} \cup \{c:0 \mid c \in \mathbb{N}\}$

as a term. As a term, the tree shown in Figure 1 would be denoted $+ [11, fac [+ [3, 2]]]$ or, using infix notation for the binary symbol $+$ in order to enhance readability, $11 + fac [3 + 2]$.

An algebra interprets every symbol of the considered signature as an operation on the domain of that algebra (where the arity of an operation equals the rank of the respective symbol). Thus, every tree may be considered as an expression that denotes an element of the domain. Taking the signature Σ from above as an example, one may, for instance, choose the domain \mathbb{N} and interpret $+$ as addition, fac as the faculty function, and $c \in \mathbb{N}$ as c . Then the number denoted by the tree which is depicted above is 131. It is important to notice that the signatures and trees themselves do not have a particular meaning—one can as well consider a totally different algebra to interpret the symbols in Σ . A tree as such is pure syntax without meaning.

A tree grammar is any device that generates a language of trees. A tree transducer transforms input trees into output trees according to some (possibly complex) rule. Now, the main observation is that one can interpret the output trees of a tree grammar or tree transducer by means of an appropriate algebra in order to obtain objects from the domain

one is interested in. Thus, one can deal with all kinds of objects in a tree-oriented way just by choosing the right algebra to interpret the trees. Many of the well-known systems studied in formal language theory can be simulated nicely in this way. For a more detailed and formal discussion see [DE98, Dre98a, Dre98b, Dre98c, Dre99].

Now, coming back to TREEBAG, the system allows to arrange instances of the four types of components as nodes of an acyclic graph and establish input/output relations between them. More precisely, the output of a tree grammar or tree transducer can be fed into tree transducers and algebras. The output of an algebra (i.e., the evaluation result) can be used as input for displays in order to visualise the respective objects (provided they are of the correct type).

Users who are familiar with earlier TREEBAG versions should notice the change! Instead of providing a display with input trees from a tree generator and with an algebra which determines the interpretation of these trees, it is now the algebra which receives the input trees and communicates the evaluation results to the display. The display itself does not any longer have a direct connection to the tree generator.

It must be stressed that each of the mentioned types of TREEBAG components consists of several classes. There is, for example, one class that implements top-down tree transformations and another one that implements the so-called YIELD mapping. Both are tree transformations, but of a different type. Technically, the latter means that they are implemented by different Java classes. In fact, one can add new types of tree transducers just by implementing a corresponding class.

Every class of TREEBAG components defines its own syntax. This makes it possible to load an instance of such a class—a concrete regular tree grammar or a top-down tree transducer, for example—from a file. Furthermore, every class provides a set of commands for interaction.

So far, the following classes of TREEBAG components are available:

- regular tree grammars, ET0L tree grammars and parallel deterministic total tree grammars (a special case of ET0L tree grammars), and branching tree grammars;
- top-down tree transducers, YIELD mappings, macro tree transducers, and a meta-class of tree transducers called iterator;
- algebras on truth values, integers, strings, trees, and two-dimensional collages, as well as algebras that correspond to the chain-code and turtle formalisms, yielding line drawings;
- displays for a textual representation of objects (which can be used to display truth values, numbers, strings, and trees), for a graphical representation of trees, and for collages and line drawings.

Because of the fact that arbitrary compositions of the available classes of tree grammars and tree transducers can be built, these classes open a large number of possibilities. For details see, e.g., [GS97, DE98, Dre98a, Dre98b, Dre98c].

2 The TREEBAG worksheet

The TREEBAG worksheet is the main window of the system. To start it, make sure that a java runtime environment is installed and knows where to search for your copy of the TREEBAG class files (i.e., include the directory or the jar-file in your set of class paths). Then, start TREEBAG using the command `java gui.worksheet` (to start with an empty worksheet) or `java gui.worksheet <file>` (to load an existing worksheet configuration).¹ Since version 1.5 TREEBAG can also be started from a web page as an applet. The applet class to be called is `gui.tbApplet.class`. It makes use of the (optional) parameter *file* – the file name of a local worksheet configuration to be loaded. (Note: Only one instance of the applet at a time may be started. If, e.g., a browser starts a second, the first will be terminated before the second starts its execution.)

Once the system has been started and the worksheet has appeared, the user can arrange instances of TREEBAG components as nodes of an acyclic graph and establish input/output relations by drawing edges between these nodes. More precisely, the worksheet allows to do the following.

2.1 The worksheet and component commands

These commands are accessible from the menu bar of the worksheet. Alternatively, one may press the right mouse button (or whichever button is the “popup button”) on a free area of the worksheet. Using the worksheet commands, the following can be done.

Saving a configuration The “Save...” item of the worksheet menu allows to save the current configuration. In addition to the structure of the configuration graph the information is saved which of the nodes are currently “open” (see below). Loading the configuration later on will open the respective nodes, too. Furthermore, if the size of the worksheet window has been changed (i.e., if it differs from the default size) that information is stored, too. Notice, however, that neither the current state of components nor available ReadMe information (see Section 3.1) is recorded.

Adding a configuration Using the “Add...” item one gets a file selector in order to select a configuration file. The corresponding configuration is then added to the worksheet. (For more details see the description of the syntax of configuration files in Section 3.)

Loading a configuration The “Load...” item works just like “add” except that the current configuration is removed from the worksheet when the new one is loaded.

Clearing the worksheet Selecting the item “Clear” will remove all nodes from the worksheet.

Exiting TREEBAG As one might expect, selecting the item “Quit” will terminate TREEBAG.

The component commands allow the user to do two additional things:

¹This is how it works on Unix, Linux, and Mac OS X systems. For Windows the procedure should essentially be the same.

Loading an instance of a TREEBAG component Selecting the item “Add component...” will open a file selector which can be used in order to select the file describing the instance to be loaded (see Section 3 for details on the syntax of these files). The same can, however, be accomplished in a quicker way by performing a double click somewhere on a free area of the worksheet. The new instance will in this case appear at the position determined by the double click.

Creating an instance of a TREEBAG component in an editor The command “Create component...” opens an editor window in which a new TREEBAG component can be created. (See below for a short description of the editor.) Once the file has been saved and could be successfully parsed, a corresponding node will appear on the worksheet.

2.2 The editor

The textual descriptions of TREEBAG components can be edited using any editor. However, TREEBAG also provides its own simple editor. It can be used to create new TREEBAG components (see above) or to edit existing ones. Per default, saving an edited file will automatically attempt to load resp. reload the component from the saved file. This behaviour can be changed using the menu item “Auto (re)load” provided by the menu “Options”.

2.3 Creating, redirecting, and deleting edges

New edges can be created by selecting (by single mouse clicks) first the source and then the target node of the desired edge. As mentioned above, if the source node represents a tree grammar or tree transducer then the target should be a tree transducer or an algebra. The other possibility is that the first node is an algebra and the second is a display. Tree grammars never accept ingoing edges whereas displays can never have outgoing ones. Any attempt to create a “wrong” edge will simply be ignored. In particular, one may cancel the creation of an edge by clicking somewhere on the worksheet instead of selecting a valid target node for this edge.

If a new edge is created, then the system removes any old edge with the same target. In addition, one can disconnect an edge from its target in order to redirect it to some other node. For this, just click at the edge. This can also be used to delete an edge by disconnecting it from its target and then clicking somewhere else on the worksheet.

2.4 Dragging nodes around

Nodes can be positioned by dragging them with the mouse in the usual way.

2.5 Opening nodes

A node v can be “opened” by a double-click. If v represents an instance of a display component the double-click opens its display window. If v is of another type or its display window has already been opened before, the double-click raises a small control pane containing a number of buttons. The buttons correspond to the commands offered by the component. As mentioned above, these commands differ from class to class (see Section 4 for details).

Since version 1.6 the control pane is not opened in its own top-level window any more.

Instead, it becomes a section of a larger window containing all opened control panes. This is intended to reduce the clutter on the screen.

The commands offered by a component may also be invoked by pressing the right mouse button (or, more correctly, the popup trigger of the window manager used) on a node. As a result, a popup menu containing the available commands becomes visible. Pressing the right mouse button on a free area of the worksheet is an alternative way of getting access to the general commands provided by the menu bar.

2.6 Commands provided by all component types

There are three standard commands which each and every TREEBAG component understands: *reload*, *delete*, and *edit*. The first updates a node when the file from which it was initially loaded has been changed. This operation will succeed only if the object described in the respective file still belongs to the same class. If not, or the file contains syntax errors, a message box appears and the operation is cancelled. The *delete* command deletes a node from the worksheet. Finally, *edit* opens the respective file in the TREEBAG editor.

3 Syntax

All TREEBAG input files are pure ASCII files that can be created and modified using any standard text editor. There is a common syntax for comments. A comment starts with a % and extends towards the end of the line (like in T_EX and L^AT_EX). Such comments can occur in all sensible places, except in the midst of an input token (which hardly anyone would consider a sensible place for a comment, I suppose).

As usual, the syntax descriptions below are given in the form of context-free productions (neglecting the possibility of comments), where ::= separates left- and right-hand sides. In addition, the following conventions are used.

- Nonterminals are always enclosed in angle brackets and are written in italics, like *<nonterminal>*.
- For terminal symbols a typewriter font is used, like `terminal`.
- The symbols `[`, `]`, `|`, `*`, `+`, `(`, and `)` are meta-symbols (unless a typewriter font is used): square brackets `[...]` enclose optional parts, `|` separates alternatives, *something*^{*} denotes zero or more occurrences of *something*, and *something*⁺ denotes one or more occurrences of *something*. Parentheses are used for grouping, in order to avoid ambiguities.

There are basically two different sorts of input files. The first one is the configuration file. Such a file describes a worksheet configuration that can be loaded using the “load” item of the worksheet menu. The second is the sort of file describing an instance of a TREEBAG component.

3.1 Configuration files

A configuration file mainly consists of a list of node definitions and a list of edges. Optionally, there may be a line defining the size of the worksheet (in pixels). Two further optional parts may be added at the very end of the configuration file. The first consists of

lists of commands that shall be executed automatically after loading. The second allows you to add a description of how to use this worksheet, or any other useful information. If you add such information, the lower right corner of the TREEBAG worksheet will display a **ReadMe** button that lets a corresponding text window pop up.

A node definition is mainly a file name together with the Cartesian coordinates of the node on the worksheet. In addition, a node definition says whether the node is to be opened and it provides the opportunity to execute a list of commands after loading (which is equivalent to the optional part mentioned above). The syntax is case insensitive with respect to keywords. It is defined as follows:

```

<configuration> ::=
  Worksheet configuration
    [ <worksheet size> ]
    <component>*
    [ Edges are <edge>* ]
  end
  [execute initially (<nat>: <commands>)* ]
  [info: <arbitrary text>]

<worksheet size> ::= Worksheet size is (<nat>, <nat>)

<component> ::=
  Component <nat> is
    [ <nat> up ] "<path name>" at (<nat>, <nat>) **
    [ with <commands> ]

<commands> ::= [<nat>] "<command>" (, [<nat>] "<command>")*

<edge> ::= <nat> -> <nat>

```

Here, *<nat>* is a natural number, *<path name>* is the path name of a file that defines a component. If a relative path name is used then it is regarded as being relative to the *n*th parent directory of the configuration file's directory. Here, *n* is the number in the optional part *<nat> up* if present, and 0 otherwise. (This may seem somewhat complicated, but it makes it possible to avoid the use of platform dependent notation for path names at least in those rather frequent cases in which a file resides in a parent directory. If you do not care about platform independence of your configuration files you may just use the usual path names.).

As mentioned above, the optional clause **with** *<commands>* lets you specify a list of commands to be executed initially by the component. In the definition for *<commands>*, the optional *<nat>* is a multiplicity and *<command>* is a command recognized by the component. An alternative place to specify such commands is the optional part starting with **execute initially**. Here, the addressed component is identified by its number, followed by a colon and the list of commands (again possibly with multiplicities). If both **with** clauses and **execute initially** are present, the commands specified in the former are executed first.

Note that, when a worksheet is saved, neither the commands to be executed initially nor the ReadMe information is stored. There is, however, an easy way to keep such information by using the include directive (see Section 3.2.5): Split the definition into two files and load a third one including both of them. Thus, the main file contains only the two lines

```
#include(worksheet.main)
#include(worksheet.opt)
```

while `worksheet.main` contains the “raw” worksheet and `worksheet.opt` contains everything following the keyword `end`. Then, whenever you wish to save your worksheet, save it to `worksheet.main`. However, inconsistencies may of course be created if components are deleted.

Components must be numbered consecutively, starting with 0. The sequence of stars following the coordinates determines how many times the component should receive opening requests (for displays 0, 1, and 2 are sensible, for other components usually only 0 and 1). Thus, if you load (on a UNIX machine, for instance) a configuration file like

Worksheet configuration

```
Component 0 is 2 up "display" at (324,161) **
Component 1 is      "grammar" at (163,86) * with 2 "advance"
Component 2 is 1 up "algebra" at (130,177)
```

Edges are

```
1 -> 2
2 -> 0
```

end

execute initially

```
0:  "auto zoom"
1:  3 "advance"
```

and this file is found in the directory `/home/someone/test` then TREEBAG will (try to) create new nodes representing instances of TREEBAG components which it loads from the files `/home/display`, `/home/someone/test/grammar`, and `/home/someone/algebra`. Afterwards, it will draw edges from the grammar to the algebra and from the algebra to the display and open the display as well as the control panels of the display and the grammar (provided these are indeed the types of the loaded components). Finally, the command “auto zoom” and five times the command “advance” are invoked on the respective components.

3.2 Component definitions

Instances of TREEBAG components are defined using the following syntax:

```
<component> ::= <class name> [ ( <name> ) ] : <instance>
```

Here, `<class name>` is the name of the (Java) class the component belongs to, `<name>` is an optional name one may provide (see below for the syntax of names), and `<instance>` is the actual description of the instance of `<class name>` to be created. The latter must use the syntax defined by this class, which is described separately for each class in Section 4.

(Internally, what happens is that the worksheet dynamically tries to find the implementation of *<class name>*, load it, create an instance, and then invoke an initialisation method of this instance, which parses the *<instance>* part of the file.)

In order to ensure as much syntactic consistency as possible the parsers have been implemented in a modular way, so that, for instance, all classes use the same notation for trees. These common syntactic figures are described in the following.

3.2.1 Numbers

Natural numbers (*<nat>*) are denoted in decimal notation, as usual (including 0). An integer (*<int>*) is a *<nat>*, optionally preceded by a minus sign. Non-negative rational numbers (*<rat₊>*) are denoted in decimal, too, with an optional decimal point. Rational numbers (*<rat>*) can be preceded by a minus sign. Finally, *<rat_[a,b]>* stands for all rational numbers between *a* and *b* (*a* and *b* included).

3.2.2 Names

A *<name>* is either a non-empty string entirely consisting of letters, digits, and the symbols +, -, *, /, ., -, <, >, and ;, or it is a string enclosed in double quotes which itself does not contain double quotes. (As a result, the double quote " cannot be part of a name. All other symbols, including newline characters, are allowed to occur within names.)

3.2.3 Sets and tuples

The usual mathematical notation is used for sets (which are always finite) and tuples. Thus, a set has the form { [*<element>* (, *<element>*)*] } and a *k*-tuple is written (*<item₁>*, ..., *<item_k>*).

3.2.4 Symbols, signatures, and trees

A (ranked) *<symbol>* has the form *<name>*:*<nat>*, the natural number being its *rank*. A *<signature>* is a set of symbols. Trees are denoted as terms in the usual way, direct subtrees being enclosed in square brackets and separated by commas. Furthermore, infix notation may be used for trees having exactly two direct subtrees. In this case the tree must be enclosed in parentheses. Altogether, this amounts to the following syntax for trees:

```
<tree> ::= <name>
          | <name>[ <tree> (, <tree>)* ]
          | ( <tree> <name> <tree> )
```

Thus, `f[(a + foo["s y m"]), g[0]]` and `f[+[a, foo["s y m"]], g[0]]` denote the same tree. For a signature Σ , a tree *t* is a *tree over Σ* if *f*:*k* is a symbol in Σ for every subtree *f*[*t*₁, ..., *t*_{*k*}] of *t*.

3.2.5 File inclusion

Sometimes it is useful to divide a component definition into several files, for example if several components have large parts in common that must be changed from time to time.

For this, TREEBAG has the include directive `#include(<filename>)`. Here, `<filename>` may be any string whatsoever, where `\` is treated as an escape character. Thus, `\<char>` yields the character `<char>`. This is important only in two cases: `\\` yields a single backslash and `\)` yields a closing parenthesis. The include directive may occur in any place and the included file may contain any substring of the component definition. Recursion is allowed, i.e., included files may themselves contain include directives. Of course, it is not a good idea to create cyclic inclusions. Include directives that are split into pieces which are distributed over several (included) files are not recognized.

4 Classes of TREEBAG components

In this section the available classes of TREEBAG components are described briefly. It is not the purpose of this manual to explain the theoretical concepts behind these implementations in a very detailed manner. For this, the reader is referred to the literature (see the references given below). Here, the emphasis is laid on explaining how the implemented TREEBAG classes work. In particular, the available commands and the class-specific part of the syntax (i.e., the part denoted by `<instance>` in the last section) are described for each class.

The standard commands *reload*, *delete*, and *edit*, which every TREEBAG component accepts, have been described at the end of Section 2 and are therefore not explicitly mentioned below.

4.1 Tree grammars

4.1.1 Regular tree grammars (class *generators.regularTreeGrammar*)

Mathematically, a regular tree grammar is a 4-tuple (N, Σ, R, S) , where N is a set of *nonterminals*, Σ is a signature, R is a set of *productions* $A \rightarrow t$ consisting of a nonterminal $A \in N$ and a tree t over $\Sigma \cup N$ (where the nonterminals are considered as symbols of rank 0), and $S \in N$ is the *start symbol*. It is required that Σ does not contain any symbol $A:0$ for which $A \in N$. The *tree language* generated by such a grammar is the set of all trees over Σ that can be derived from S by repeatedly applying productions in the obvious way, i.e., replacing a nonterminal with the right-hand side of an appropriate production (see [GS97] for details). In addition, in TREEBAG one can assign a *weight* to every production.

Syntactically, in TREEBAG the set of nonterminals is a set of names. The syntax for productions is `<name> -> <tree> [weight <rat+>]`. For example, the following is a syntactically correct regular tree grammar:

```
( { left, somewhere },
  { left:2, .:2, 0:0 },
  { left -> left[left[left, somewhere], somewhere] weight 4,
    left -> 0,
    somewhere -> (somewhere . somewhere),
    somewhere -> 0 },
  left )
```

The grammar generates all trees with zeroes as leaves, such that the leftmost branch consists of an even number of `left` symbols and all the other symbols (except leaves) are dots.

The implementation of regular tree grammars basically provides two modes, called *enumeration* and *random generation* mode. The enumeration mode yields an enumeration of the tree language generated by the grammar. In this mode the main commands are *advance* (compute the next tree in the enumeration), *reset* (return to the first tree), and *random generation* (turn to random generation mode). In addition, there is the command *derive stepwise*, which yields a stepwise derivation of the current tree of the enumeration. An application of this command makes the grammar turn to the start of the derivation and brings up the additional commands *single step* (perform a single step of the derivation), *parallel step* (perform several steps of the derivation by applying productions to all nonterminals in parallel), *back* (undo the last single or parallel step), and *results only* (return to the enumeration mode).

The random generation mode iteratively applies productions to all nonterminals in parallel, using a random choice depending on the weight of productions. Here, the main commands are *refine* (replace all nonterminals in the current tree by the right-hand sides of randomly chosen productions), *back* (undo one refinement step), *enumeration* (apply terminating productions and return to enumeration mode), and *reset* (return to the initial nonterminal). As mentioned above, the random choice taken in the refinement step depends on the weight of productions (the default weight being 1). If w is the sum of the weights of all productions with left-hand side A and one of these productions has weight w_0 , then this production will be chosen with probability w_0/w (where $0/0 = 0$, by convention). In case of the sample grammar above, for instance, this would mean that the first production is applied to an occurrence of the nonterminal `left` with probability $4/5$.

4.1.2 Parallel deterministic total tree grammars (class `generators.pdtGrammar`)

Parallel deterministic total tree grammars are a very special, yet frequently useful class of tree grammars. They are similar to regular tree grammars, except that, for every nonterminal, there must be exactly two productions the second of which is terminating (i.e., contains no nonterminals). Derivations are maximum parallel, applying n times the first production to all nonterminals in parallel, and then, finally, the second production (which corresponds to the derivation mode of table-driven L-systems, with a fixed number of two tables). Thus, for every $n > 1$ there is exactly one tree t_n that can be derived by a derivation of length n . The generated language is the set $\{t_1, t_2, \dots\}$ of all these trees.

The syntax is like the one for regular tree grammars, the difference being that the two productions for each nonterminal are denoted as `<name> -> <tree> | <tree>` (where the left-hand side is the nonterminal and the two trees are the first and second right-hand side, separated by `|`). Because of the deterministic nature of these grammars, productions cannot be assigned a weight. An example is

```
( { start, left, right },
  { left:2, right:2, .:2, 0:0 },
  { start -> (left . right) | (0 . 0),
    left -> left[left, left] | left[0, 0],
    right -> right[right, right] | right[0, 0] },
  start )
```

which generates all trees $(t . t')$ such that t and t' are fully balanced trees of equal height, where t consists of the symbols `left` and `0`, and t' consists of the symbols `right` and `0`.

The implementation provides enumerations of the generated terminal and nonterminal trees, the n th tree in the enumeration being the one that can be derived by a parallel derivation of length n . The basic commands are *advance* (turn to the next tree in the enumeration), *back* (return to the previous tree in the enumeration), and *reset* (return to the first tree). In addition, the commands *terminal results* and *nonterminal results* allow to switch between terminal and nonterminal output terms.

4.1.3 ETOL tree grammars (class `generators.ETOLTreeGrammar`)

ETOL tree grammars are a tree-grammar version of the well-known ETOL-systems (extended context-free Lindenmayer systems with tables). They are parallel grammars which generalize the previously described class `generators.pdtGrammar`.

Syntactically, an ETOL tree grammar is a regular tree grammar except for the following differences. The set of rules is now a *set of sets of rules* $\{R_1, \dots, R_n\}$. Each set R_i is called a *table*. The output signature is allowed to intersect with the set of nonterminals and instead of the initial nonterminal one can, more generally, specify an initial tree called the *axiom*. An example is

```
( { c },
  { root:1, a:2, b:2, c:0 },
  { { c -> a[c, c] }, { c -> b[c, c] } },
  root[c] ).
```

Derivations start with the axiom. A derivation step is made by choosing a table and then replacing all nonterminals in parallel, using rules of the chosen table. If, for a given nonterminal A , the table does not contain an appropriate rule, then the implicit rule $A \rightarrow A$ is applied (i.e., A stays as it is). The example above is deterministic in the sense that every table contains only one rule for each nonterminal. In this case, the result of a derivation is uniquely determined by the chosen sequence of tables. (The grammar generates all trees of the form `root[t]` such that t is a fully balanced binary tree whose internal nodes are labelled with `a` respectively `b`, whose leaves are labelled with `c`, and where nodes at the same distance from the root have identical labels.)

The last syntactical deviation from regular tree grammars is that one may add, as a fifth component (i.e., after the specification of the initial nonterminal and separated by a comma) a regular expression which specifies the admissible table sequences. In such

a regular expression, the numbers $1, \dots, n$ denote the tables, juxtaposition denotes concatenation of table sequences, commas separate alternatives, E^* denotes arbitrarily many repetitions of E (i.e., the Kleene star), E^+ denotes at least one repetition of E (which is equivalent to EE^*), square brackets enclose optional parts, and parentheses (\dots) are used to override the precedence rules. Without parentheses, $*$ and $^+$ bind strongest, concatenation binds second strongest, and commas bind weakest. For instance, in a grammar with three tables, the expression $(1, 2\ 3)^*2^+$ requires admissible table sequences to start with a sequence of tables 1, 2, and 3 where tables 2 and 3 occur only in direct succession (and in this order). After this initial part, the sequence must end with at least one application of table 2 (note the space between 2 and 3; without it, this would refer to table number 23).

The implementation of ETOL tree grammars translates such a grammar into a regular tree grammar whose output signature is monadic, and a top-down tree transducer (see [ERS80] for the theory behind). Roughly speaking, an output tree of the regular tree grammar determines the table sequence (every symbol corresponds to a table) and the top-down tree transducer implements the actual rules, using the input symbols in order to choose the tables consistently. The parsing routine of the class `generators.ETOLTreeGrammar` stores the two components in files named *file.1* and *file.2*, where *file* refers to the parsed file. (The two files are syntactically correct descriptions of TREEBAG components and can therefore be studied or loaded onto the worksheet if someone wants to do so.)

The available commands are rather similar to those of regular tree grammars. There are two basic modes: *table enumeration* (the initial one) and *random tables*. The first implements an enumeration of the table sequences. In this mode, the following commands are available:

- *advance* turns to the next table sequence in the enumeration. A derivation based on this sequence is chosen nondeterministically, and the resulting tree is the output tree. If the grammar is deterministic, this yields an enumeration of the generated tree language;
- *reset* returns to the first table sequence;
- *derive stepwise* is similar to the corresponding command of regular tree grammars and allows to view the current derivation step by step, using the commands *derivation step* (perform one parallel step of the current derivation), *back* (back up one step), and *results only* (switch back to the state in which only the terminal results of derivations are shown);
- *random generation* switches to the second main mode.

If the grammar contains nondeterministic tables, an additional command *new derivation* is available. Upon invocation, it randomly chooses a new derivation based on the current table sequence (which is not affected by the command).

In the *random tables* mode a derivation based on a random table sequence is performed using the commands *refine*, *back*, and *reset*, similar to the corresponding mode of regular tree grammars. Note, however, that only the table sequence is chosen at random. If the

grammar is nondeterministic, the command *new derivation* should be used in order to get further output trees.

Pragmatically, if you do not restrict the table sequences by a regular expression, it is often a good idea to have one or more tables which contain terminating rules for all nonterminals. In this case, the translation into a regular tree grammar and a top-down tree transducer guarantees that the enumeration mode yields only defined results (which may otherwise fail to be the case because the tree transducer does not always turn a table sequence into a terminal tree).

The translation of the optional regular expression into a regular tree grammar does not perform state minimization. Therefore, it is not as efficient as one would like it to be. Moreover, depending on the expression used, certain table sequences may occur repeatedly in the table enumeration mode.

4.1.4 Branching tree grammars (class *generators.BSTGrammar*)

This class implements branching tree grammars introduced in [DE04] (with some slight deviations, some of them being motivated by the use of these tree grammars in [Dre05]). The following description assumes that the reader is familiar with the definition of this type of tree grammars. Very roughly speaking, branching tree grammars generalize ETOL tree grammars using the concepts of branching synchronization and nested tables, as follows. Rather than having only one set of tables, the tables of a branching tree grammar are organized into a hierarchy of supertables consisting of n levels (where n , the nesting depth, is fixed for every grammar). In addition, every nonterminal in the right-hand side of a rule is given n so-called synchronization symbols. These synchronization symbols are accumulated into n strings of synchronization symbols during a derivation. (Since derivations are fully parallel, all these strings will be of the same length.) Now, if the first k strings of synchronization symbols of two nonterminals are equal, rules from the same supertable at depth k must be applied to them.

Syntactically, the description of a branching tree grammar is similar to an ETOL tree grammar, with the following differences:

- The nesting depth n and the set of synchronization symbols are specified as the third and fourth components (i.e., after the set of nonterminals and the output signature).
- The table structure is denoted as a nested hierarchy of sets of rules. Thus, for $n = 0$ it is just a set of rules (as in a regular tree grammar), for $n = 1$ it is a set of sets of rules (as in an ETOL tree grammar), and so on.
- Each nonterminal in the axiom and in the right-hand side of a rule is followed by n synchronization symbols between angular brackets and separated by white space.

The definition of the actual grammar can be preceded by two optional statements (separated by commas if both are present):

1. `[no] implicit rules`
determines whether or not implicit rules (replacing a nonterminal with itself, by convention using the first synchronization symbol in all places) should be added if a

left-hand side is not present in a table. The default is that no such rules are added.

2. (plain | normal | extended) translation

allows to choose between three different types of translation. Internally, a branching tree grammar is translated into a regular tree grammar and a chain of n top-down tree transducers. Plain translation is the one given in [DE04], which is theoretically correct but practically not very useful as many of the simulated derivations will yield undefined and/or identical results. Normal translation (the default) tries to avoid this effect but shares with plain translation the unpleasant property that it is sometimes quite inefficient since unnecessarily large intermediate trees are generated internally. Extended translation often solves this problem, but may in certain (seldom?) cases result in exponentially large state sets. As a rule of thumb, try extended translation if normal translation seems to result in an inefficient generation process or many of its derivations yield undefined output trees.

Similar to ETOL tree grammars, the choice of tables at depth 1 can be restricted by a regular expression which is optionally added after the axiom and separated from it by a comma. The only difference is that the supertables at depth 1 must be given names (rather than using numbers), which is optional in the definition of (super)tables. For this, rather than simply using set notation $\{\dots\}$, the desired name is specified using the syntax `table "<string>" $\{\dots\}$` . The names defined in this way are then used in the regular expression (also in double quotes) in order to refer to tables. As another means to influence the generation process, tables can be given weights (the default being 1) in order to change their relative frequency in random tables mode.

Here is an example of nesting depth 2, where the first supertable at depth 1 is named but no regular expression is given (i.e., the name is superfluous):

extended translation

```
( { C, E },
  { F:4, G:3, H:1, a:0 },
  { 1, 2 },
  2,
  {
    table "xyz" {
      {
        C -> G[E<1 2>, C<1 1>, E<1 2>],
        E -> H[E<1 1>]
      },
      {
        C -> G[E<1 2>, C<1 1>, E<1 2>],
        E -> H[E<1 1>]
      }
    },
    {
      {
```

```

        C -> a,
        E -> a
    }
} weight 0
},
F[C<1 1>,C<1 1>,C<1 1>,C<1 1>] )

```

The fact that the second supertable at depth 1 has been given the weight 0 implies that it will never be chosen in a derivation in random tables mode (which may be intended because this table is terminating). Assigning weights to tables at nesting depth 1 has no effect if admissible table sequences are specified by means of a regular expression.

The available user commands are similar to those known from ETOL tree grammars, except that the command *new derivation* is replaced with *choose new supertables at depth i* for all $i \in \{2, \dots, n\}$ as well as *choose new rules*. Thus, applying the command *choose new supertables at depth i* will make new random choices for the supertables at depth i and below (but keep those above) and the command *choose new rules* will make a new random choice of rules (but keep the tables). The latter is only available if the grammar is nondeterministic. (Clearly, the command *choose new supertables at depth i* is useless if no supertable at depth $i - 1$ contains more than one subtable, but the command will be available in any case.)

To avoid confusion, it should be pointed out that the class allows nonterminals to be output symbols, as in [Dre05, Section 6.3], but in contrast to the original definition in [DE04]. As discussed in [Dre05, Section 6.3.1], such branching tree grammars have one “hidden” level of synchronization. To cope with this fact, the implementation internally inserts a new supertable above the supertable at depth 0. Externally, this becomes visible only in two ways. The table enumeration mode does not any more enumerate all sequences of tables at depth 1. It merely increases the length of derivations (because what it actually does now is enumerating all sequences of tables at depth 0 – and there exists only one of them). Correspondingly, the command *choose new supertables at depth i* is now also available for $i = 1$.

Finally, there is a pair of commands called *show sync info* and *hide sync info*. If the former is invoked, the grammar will generate trees in which the nonterminals are given monadic subtrees that represent the synchronization strings. Thus, to exploit this you should use a free term algebra and a tree display.

4.2 Tree transducers

4.2.1 Top-down tree transducers (class *generators.tdTransducer*)

A top-down tree transducer is a tuple consisting of an input signature Σ , an output signature Σ' , a set Γ of *states* which are considered as symbols of rank 1, a set of *rules*, and an *initial state* $\gamma_0 \in \Gamma$. The signatures Σ and Σ' must not contain any symbol $f:1$ such that $f \in \Gamma$. The rules are left-linear term rewrite rules of the form $\gamma[f[x_1, \dots, x_n]] \rightarrow t$, where $\gamma \in \Gamma$, $f \in \Sigma$, x_1, \dots, x_n are pairwise distinct variables, and t is a tree over Σ' , which may in addition contain subtrees of the form $\gamma'[x]$, where $\gamma' \in \Gamma$ and $x \in \{x_1, \dots, x_n\}$.

The transformation of an input tree t_0 is performed nondeterministically, starting with $\gamma_0[t_0]$ and applying rules until no state is left, i.e., a tree over Σ' is obtained. For a more thorough introduction to top-down tree transducers see [GS97, FV98] and the references cited there.

Syntactically, the set of states is a set of names. A variable consists of the letter x , followed by a decimal non-negative integer (without leading zeroes). As in the case of regular tree grammars, rules can be assigned a weight in order to influence the nondeterministic choice of rules. As an example, one may consider the following top-down tree transducer:

```
( { f:2, a:0, b:0 },
  { f:2, a:0, b:0 },
  { start, a, b },
  { start[f[x1, x2]] -> f[a[x2], a[x2]] weight 2,
    start[f[x1, x2]] -> f[b[x1], b[x1]],
    a[f[x1, x2]] -> f[a[x2], a[x2]],
    a[a] -> a,
    a[b] -> a,
    b[f[x1, x2]] -> f[b[x1], b[x1]],
    b[a] -> b,
    b[b] -> b },
  start )
```

The transformation computed by this top-down tree transducer is undefined for the input trees a and b (since no rule applies to `start[a]` or `start[b]`). For every other input tree there are two possible results, the first one being twice as probable: Either the tree is turned into a fully balanced binary tree over f and a whose height is the length of the rightmost branch of the input tree, or it is turned into a fully balanced binary tree over f and b whose height is the length of the leftmost branch of the input tree.

In its default mode a top-down tree transducer performs complete random derivations, producing an output tree every time it receives an input tree.² Similar to the implementation of regular tree grammars there is a stepwise mode that allows to view in a stepwise manner the derivation which yielded the current output tree. By default, subsequent computations use equal initialisation values for the random number generator that chooses between applicable rules. This turns out to be useful because it produces more comprehensible results in most of the typical situations.

The available commands are *derive stepwise* (switch to “stepwise” mode, yielding the additional commands *single step*, *parallel step*, *back*, and *results only*, similar to the respective commands of regular tree grammars). If the transducer is nondeterministic (i.e., at least one combination of a state and an input symbol occurs twice as a left-hand side), there are the additional commands *new random seed* (compute another output tree, using a new

²In fact, the result can also be undefined, namely if a derived term still contains states, but no rule applies. A nicer implementation of top-down tree transducers would produce an enumeration of all possible output trees, but this has not yet been implemented.

initialisation value for the random number generator), and *variable random seed* (always use a new random seed). In the latter case one can switch back using the command *fixed random seed*.

4.2.2 YIELD mappings (class `generators.YIELDTransduction`)

A YIELD mapping (see, e.g., [ES77, ES78, DE98]) is a YIELD algebra (see below), turned into a tree transformation: if the interpretation of an input term by the YIELD algebra results in a tree without variables, that tree is returned. Otherwise, the result is undefined. There are no commands (except the standard commands *delete node* and *reload file*, of course), and the syntax is the same as for YIELD algebras.

4.2.3 Macro tree transducers (class `generators.mtTransducer`)

This class implements the concept of a macro tree transducer as introduced in [Eng80, EV85] (see [FV98] for further references), which extends the top-down tree transducer. Besides the input tree to be processed, a state of a macro tree transducer can have a fixed number of parameter trees (but the initial state must be of rank 1, i.e., parameterless). As a consequence, states may be nested, since the parameters of a state may themselves contain states. A typical rule is

$$q[f[x_1, x_2], y_1, y_2] \rightarrow g[q[x_1, f[a, q'[x_2]], y_2], q''[x_1, a]].$$

Here, q , q' , and q'' are states with two parameters, zero parameters, and one parameter, respectively. In the rules, the variables x_1, x_2, \dots play the same role as in td transducers whereas y_1, y_2, \dots denote parameters. Note that the recursion goes over the first subtree of a state, which is the input tree. For more thorough introductions and examples see the references mentioned above.

Except for the more general form of rules, using the names y_1, y_2, \dots for the parameters, the syntax used by the class `mtTransducer` is similar to that of the class `tdTransducer`. Naturally, ranks must now be given in the signature of states. In addition, there is an option called **performance**, which can be given after the parenthesis enclosing the whole tuple (separated by a comma). In performance mode, the implementation will not produce the internal data structures needed to perform stepwise derivations (and, thus, stepwise derivations are unavailable in this case). This leads to a considerable increase in efficiency.

Besides the commands known from the case of td transducers (which have a similar semantics as in that case), some additional commands are provided.

- In macro tree transducers which are neither deterministic nor total, one can switch between safe and blind derivations. The former is the default. It means that the transducer will only make derivations that lead to an output tree, thus avoiding undefined results if possible (i.e., if there exists at least one output tree for the given input tree). Blind derivations skip the respective checks, which may be useful in connection with stepwise mode if arbitrary derivations shall be considered.
- It is possible to switch between derivations in IO (inside-out or innermost-outmost) and OI (outside-in or outermost-innermost) modes. In the case of nested states, rewriting always continues with the innermost states in the IO case, and with the

outermost states in the OI case. It is a well-known fact that these evaluation strategies result in different tree transformations. More precisely, the set of output trees obtained in IO mode may be strictly smaller than the set of output trees obtained in OI mode (which is equivalent to unrestricted mode). The reason for this is that, in IO mode, parameters are evaluated before applying rules that possibly duplicate them, yielding identical subtrees, while evaluation after copying may result in different subtrees (in the presence of nondeterminism).

4.2.4 Iterators (class *generators.iterator*)

An iterator is a tree transducer that encapsulates another tree transducer and works like the encapsulated object, except that it adds the command “iterate”, which can be used to iterate the application of the tree transducer. When the iterator receives an input term, the output is the term obtained by applying the tree transducer as if it was not encapsulated by an iterator. The iterator keeps the output term, however, so that it can apply the tree transducer to this term when the command *iterate* is invoked. Note that, usually, wrapping a tree transducer by an iterator makes sense only if its output signature is a subset of its input signature.

The syntax equals the syntax of the encapsulated tree transducer, including the initial part *<class name> [(<name>)] :.* Thus, an iterator that encapsulates the sample top-down tree transducer given above could be denoted as follows:

```
generators.tdTransducer:
( { f:2, a:0, b:0 },
  { f:2, a:0, b:0 },
  { start, a, b },
  { start[f[x1, x2]] -> f[a[x2], a[x2]] weight 2,
    start[f[x1, x2]] -> f[b[x1], b[x1]],
    a[f[x1, x2]] -> f[a[x2], a[x2]],
    a[a] -> a,
    a[b] -> a,
    b[f[x1, x2]] -> f[b[x1], b[x1]],
    b[a] -> b,
    b[b] -> b },
  start )
```

(Note that the first line belongs to the class-specific part of the definition of an iterator. As always, this has to be preceded by something like “*generators.iterator:*” or “*generators.iterator("iterated top-down tree transducer"):*” in order to tell the system that this is meant to define an iterator.)

4.3 Algebras

So far, none of the available classes of algebras provides a command except the standard commands *delete node* and *reload file*. Furthermore, in most classes all instances behave

alike, so that the syntax just consists of the empty string. Below, in these cases syntax is not mentioned explicitly.

4.3.1 The boolean algebra (class *algebras.booleanAlgebra*)

The boolean algebra interprets the following symbols in the standard way: `true:0`, `false:0`, `not:1`, `and:2`, `or:2`, and `xor:2`.

4.3.2 The chain-code algebra (class *applications.lineDrawings.chainCodeAlgebra*)

The chain-code algebra (cf. [Dre98b]) is one of the two types of line-drawing algebras available so far, the other type being the turtle algebra described below. Such an algebra yields line drawings, two-dimensional pictures consisting of a sequence of unit lines and an endpoint. The concatenation of two line drawings works in the obvious way. First, the second argument is translated, moving its origin to the endpoint of the first (including a corresponding translation of its endpoint). Then, the two sequences are concatenated, the new endpoint being the (translated) endpoint of the second argument. Concatenation of n line drawings works alike (notice that the concatenation operation is associative).

The chain-code algebra interprets *all* symbols of rank $n \in \mathbb{N}$ as n -ary concatenation operations (in particular, $n = 0$ yields a constant denoting the empty line drawing whose endpoint is the origin), except a few symbols which have a special meaning:

- The symbols `l:0`, `r:0`, `u:0`, and `d:0` are interpreted as lines starting at the origin and extending one unit to the left, to the right, up, and down, respectively (which also determines their endpoints). Synonyms for these symbols are `w:0`, `e:0`, `n:0`, and `s:0` standing for *west*, *east*, *north*, and *south*, respectively.
- The symbol `hide:n` is interpreted like an n -ary concatenation operation, the difference being that all lines of the resulting line drawing are discarded. In other words, this operation yields the empty line drawing whose endpoint is obtained by summing up the endpoints of its arguments. (One may alternatively say that the lines are made transparent, so that they become invisible.)
- Finally, symbols of the form `h-scaler:n`, `v-scaler:n`, and `scaler:n`, where r is a $\langle \text{rat}_+ \rangle$, are interpreted as n -ary concatenation operations which, in addition, scale their result by the factor r horizontally, vertically, or in both directions, respectively.

4.3.3 Collage algebras (class *applications.collages.collageAlgebra*)

Collage algebras [Dre96, Dre98b] deal with objects called collages. In the current implementation a collage is a finite list of two-dimensional geometric parts, which may be coloured. The colour of a part is either static (which means that it is not affected by the colour operations described below) or determined by the value of so-called colour attributes. In any case, the actual colour is given by three numbers between 0 and 1 which are interpreted in the rgb colour model. Symbols of rank greater than 0 can denote two different kinds of operations on collages, so-called collage operations and the mentioned colour operations.

The most basic variant of a collage operation $\langle f_1 \cdots f_n \rangle$ is given by n affine transformations,

as follows. For argument collages C_1, \dots, C_n

$$\langle f_1 \cdots f_n \rangle(C_1, \dots, C_n) = \bigcup_{i=1, \dots, n} f_i(C_i).$$

Thus, the i th argument C_i is transformed using the i th transformation f_i . Afterwards, the union of the resulting collages is taken. (Here, the union is acutally list concatenation, which makes a difference when collages with overlapping parts of different colours are drawn. In this case the resulting picture consists of the overlay of all parts in the order in which they occur in the list. Consequently, if parts overlap the colour of the last one will prevail.) In a slightly more general form of collage operations, the transformations f_1, \dots, f_n in its definition are interspersed with an arbitrary number of constant collages (which does not affect the arity of the operation). These collages will then, in addition to the transformed argument collages $f_i(C_i)$, be added to the resulting collage in the given order.

A colour operation modifies colour attributes, thus affecting the rgb values of those parts which use these attributes to determine their colour. More precisely, a colour operation $F = \text{colourOperation}(\text{name}|f:d)$ is determined by the respective attribute name and two real numbers f, r between 0 and 1. Here, ‘ f ’ stands for ‘fraction’ or ‘force’ and ‘ d ’ stands for ‘destination’. For every part, if the value of the given attribute currently is x , then its new value will be $x + (d - x) * f$. Intuitively, this means that the value is drawn towards d by the fraction f (with respect to the distance between x and the destination value d). For example, if $f = 2/3$, $d = 1/4$, and $x = 1/2$ then the new value will be $1/3$, which is $2/3$ nearer to $1/4$ than $1/2$ is. The variant $F' = \text{colourOperation}(\text{name}||f:d)$ is defined in the same way, but substituting $1 - x$ for x . In other words, F' first mirrors x at $1/2$ and acts like F afterwards. For convenience, several triples of the form $f:d(\text{name})$ or $f::d(\text{name})$ may be combined in one colour operation, separated by commas, if the respective names are pairwise distinct.

Colour operations of the form just described can also be *higher order*. In this case, f and/or d are replaced by names of attributes, say a_f and a_d . For each particular application of such an operation to a given part, f and d will then be given by the current values of a_f and a_d .

The colour operations described above are badly disguised affine transformations. By rewriting the definition one easily checks that the transformations that can be expressed by operations of the forms $\text{colourOperation}(\text{name}|f:d)$ and $\text{colourOperation}(\text{name}||f:d)$ are exactly the one-dimensional affine transformations that map the interval $[0, 1]$ to itself. In addition to these, collage algebras provide a much more general class of colour operations, namely all transformations of the n -dimensional space of attribute values, which can be expressed by a certain set of arithmetic operations (see below). This type of colour operation need not even assign values in the interval $[0, 1]$ to all attributes. However, whenever some $x \notin [0, 1]$ is to be interpreted as a red, green, or blue value this is done by continuously “folding” \mathbb{R} into $[0, 1]$: If $\lfloor x \rfloor$ is even then the resulting value is $x - \lfloor x \rfloor$, otherwise it is $\lfloor x \rfloor + 1 - x$. (This operation, called *fold*, is also among the primitive operations that can be used to modify attribute values.)

Syntactically, the definition of a collage algebra starts with an optional list of attribute names according to the following syntax:

attributes *<name>* (, *<name>*)*;

If the list of attributes is omitted then the attributes **r**, **g**, **b** are automatically defined (which is *not* done if attributes are defined explicitly).

The remaining part of the syntax is a set of definitions, each definition assigning a meaning to a name. The syntax of these definitions is the following (where *<expr>* denotes an expression yielding a *<rat>*; the exact syntax of expressions is described below).

```

<definition> ::=
  <name> = ( <const> | <collage-union> | <transformation> | <operation> )

<const> ::= const ( <expr> )

<collage-union> ::= <collage> (+ <collage>)*

<collage> ::= <part-set> ((transformed | coloured) by <name>)*
  (where <name> refers to a previously defined transformation or colour operation)

<part-set> ::= { [ <part> ( , <part>)* ] } (i.e., a <part-set> is a set of parts)

<part> ::= <part-def> [ <colour> ]
  [ width <expr> ] ((transformed | coloured) by <name>)*

<part-def> ::=
  (curve | polyline) ( <bezier point> ( , <bezier point>)* )
  | (closedCurve | polygon) ( <bezier point> ( , <bezier point>)* )
  | (filledCurve | filledPolygon) ( <bezier point> ( , <bezier point>)* )
  (the polygon/polyline versions are obsolete but exist for reasons of backwards
  compatility)
  | ellipse( <expr>, <expr>, <expr>, <expr> )
  | filledEllipse( <expr>, <expr>, <expr>, <expr> )
  (parameters of ellipses: coordinates of centre, horizontal radius, vertical radius)
  | arc( <expr>, <expr>, <expr>, <expr>, <expr> )
  | closedArc( <expr>, <expr>, <expr>, <expr>, <expr> )
  | filledArc( <expr>, <expr>, <expr>, <expr>, <expr> )
  | pie( <expr>, <expr>, <expr>, <expr>, <expr> )
  | filledPie( <expr>, <expr>, <expr>, <expr>, <expr> )
  (parameters of pies and arcs: coordinates of centre, radius, start angle, angle)
  | "<string>" [ [ <font options>* ] ] size <expr> at <coordinates>
  (string in default or specified font with given font size and centered at given point)

<bezier point> ::= <coordinates> [ left <coordinates> ] [ right <coordinates> ]

<font options> ::= ( "<string>" | bold | italic | outlined )*
  (any combination of font name and attributes bold, italic, outlined)

<coordinates> ::= ( <expr>, <expr> )

```

$\langle \text{colour} \rangle ::=$
 $[\langle \text{expr} \rangle [, \langle \text{expr} \rangle, \langle \text{expr} \rangle]]$
 (static rgb values between 0 and 1, will not be affected by colour operations)
 $| \langle \langle \text{expr} \rangle [, \langle \text{expr} \rangle, \langle \text{expr} \rangle] \rangle$
 (use default attributes **r,g,b**; assign them values between 0 and 1. Only available if no user-defined attributes are used!)
 $| \langle \langle \text{name} \rangle, \langle \text{name} \rangle, \langle \text{name} \rangle [; \langle \text{name} \rangle = \langle \text{expr} \rangle (, \langle \text{name} \rangle = \langle \text{expr} \rangle)^*]$
 (attributes determining the part's colour; optional assignment of values between 0 and 1 to attributes (default is 0). Only available if user-defined attributes are used!)

$\langle \text{transformation} \rangle ::= \langle \text{primitive transformation} \rangle (. \langle \text{primitive transformation} \rangle)^*$

$\langle \text{primitive transformation} \rangle ::=$
translate ($\langle \text{expr} \rangle, \langle \text{expr} \rangle$) (a translation)
 $|$ **scale** ($\langle \text{expr} \rangle [, \langle \text{expr} \rangle]$) (a uniform/non-uniform scaling)
 $|$ **matrix** ($\langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle$)
 (**matrix**(a,b,c,d) denotes the linear transformation given by the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$)
 $|$ **map** ($\langle \text{coordinates} \rangle \rightarrow \langle \text{coordinates} \rangle,$
 $\langle \text{coordinates} \rangle \rightarrow \langle \text{coordinates} \rangle$
 $[, \langle \text{coordinates} \rangle \rightarrow \langle \text{coordinates} \rangle]$)
 (**map**($c_1 \rightarrow c'_1, c_2 \rightarrow c'_2, c_3 \rightarrow c'_3$) denotes the affine transformation that maps c_1 to c'_1 , c_2 to c'_2 , and c_3 to c'_3 ; for this, $c_1 - c_3$ and $c_2 - c_3$ must be linearly independent. In other words, the points c_1, c_2, c_3 must not lie on the same straight line. If $c_3 = (0,0) = c'_3$ then the last pair of points can be omitted.)
 $|$ **similarity** ($\langle \text{coordinates} \rangle \rightarrow \langle \text{coordinates} \rangle,$
 $\langle \text{coordinates} \rangle \rightarrow \langle \text{coordinates} \rangle$)
 (**similarity**($c_1 \rightarrow c'_1, c_2 \rightarrow c'_2$) denotes the similarity transformation that maps c_1 to c'_1 and c_2 to c'_2 without changing orientation; for this, c_1 and c_2 must be distinct.)
 $|$ **use** ($\langle \text{name} \rangle$) (use a previously defined transformation)

$\langle \text{operation} \rangle ::=$
 $\langle [(\langle \text{name} \rangle | \langle \text{collage-union} \rangle) (, (\langle \text{name} \rangle | \langle \text{collage-union} \rangle))^*] \rangle$
 $|$ **colourOperation**($\langle \text{spec} \rangle (, \langle \text{spec} \rangle)^*$)

$\langle \text{spec} \rangle ::= \langle \text{simple spec} \rangle | \langle \text{generalized spec} \rangle$

$\langle \text{simple spec} \rangle ::=$
 $\langle \text{name} \rangle (| | |) (\langle \text{expr} \rangle | \text{val}(\langle \text{name} \rangle)) : (\langle \text{expr} \rangle | \text{val}(\langle \text{name} \rangle))$

$\langle \text{generalized spec} \rangle ::= \langle \text{name} \rangle := \langle \text{attr expr} \rangle$
 (see below for the definition of $\langle \text{attr expr} \rangle$)

The four types of definitions have the following meaning.

A definition $c = \text{const}(\langle \text{expr} \rangle)$ defines a numerical constant c which can be used in

subsequent expressions. Primitive expressions are numbers (i.e., $\langle rat \rangle$'s), the constant e , and expressions of the form $\#c$, where c is (the name of) a constant defined earlier. More complex expressions can be built using the binary operators $+$, $-$, $*$, and $/$ (which are left associative and follow the usual precedence rules), a unary $-$, parentheses, and the functions

- $\min(x, y)$ (the minimum of x and y)
- $\max(x, y)$ (the maximum of x and y)
- $\sin(x)$ (sine of x degrees)
- $\cos(x)$ (cosine of x degrees)
- $\tan(x)$ (tangent of x degrees)
- $\text{sqrt}(x)$ (square root of x)
- $\max(x, y)$ (the maximum of x and y)
- $\text{power}(x, y)$ (yielding x^y)
- $\ln(x)$ (natural logarithm of x)
- $\text{ifneg}(x, y, z)$ (y if $x < 0$, otherwise z)

Note that spaces are necessary to separate $\#c$ from a subsequent operator since, e.g., length-5 is a valid name for a constant. Thus, $\#\text{length-5}$ refers to that constant whereas $\#\text{length} - 5$ denotes the expression which subtracts 5 from the value of the constant length .

A definition $f = \langle \text{collage-union} \rangle$ means that the collage algebra will interpret the symbol $f:0$ as a collage. A collage may be defined as a union (using the symbol $+$) of sets of parts and other, already defined collages (referred to as $\#C$, where C is the name of the collage). Note that there are several way to define the colour of a part:

1. Rgb values in square brackets are static colours that are not affected by colour operations. Specifying no colour at all is equivalent to specifying $[0,0,0]$ (which, in turn, is equivalent to $[0]$), i.e., black.
2. Colours specified between angel brackets are determined by attribute values and are thus subject to colour operations. If no explicit attributes have been defined, one simply specifies the numerical values of the three default attributes r, g, b . If a user-defined set of attributes is used, one first specifies the names of the three attributes that determine the colour, and then, separated by a semicolon, the values of nonzero attributes.

If present, $\text{width } x$ determines the line width used in drawing the part (which has no effect on filled polygons and curves). Curves (possibly closed or filled) are Bézier curves (see [FDF⁺97]). Each of the main points p_0, \dots, p_{n-1} of the curve can optionally be assigned a left and a right control point. The segment between p_i and $p_{i+1 \bmod n}$ is a curve that leaves p_i into the direction given by the right control point of p_i and approaches $p_{i+1 \bmod n}$ from the direction given by the left control point of $p_{i+1 \bmod n}$. (This implies that the left control point of p_0 and the right one of p_{n-1} have no effect unless the curve is closed

or filled.) *The coordinates of control points are given relative to the coordinates of the corresponding main point* (which makes it easy to move points around without having to change the control points all the time). If only one control point is given, the other one is obtained by mirroring at the respective main point (which yields a smooth curve unless the control points coincide with the main one). If neither of the two control points are given explicitly, they are assumed to be identical to the main point, so leaving out both `left` and `right` is equivalent to `left (0,0) right (0,0)`. Note that a curve becomes a polyline if all control points coincide with their respective main points.

An ellipse is specified by the coordinates of its centre, the horizontal radius, and the vertical radius. (The axes of an ellipse are always parallel to the coordinate axes. To obtain other ellipses, a transformation must be applied.) An arc is a section of a circle given by the coordinates of the centre of the circle, its radius, the angle where the section begins, and the angle of the section itself. Closed arcs are obtained by adding a chord connecting the two end points. A pie is similar to an arc except that it is always closed, namely by lines connecting the two endpoints with the centre of the circle (rather than by a chord).

A part can be a string, in a default or explicitly given font, possibly with one or more of the attributes `bold`, `italic`, and `outlined`. Available fonts depend on the Java installation, but the standard Java fonts `Dialog`, `DialogInput`, `Monospaced`, `Serif`, `SansSerif`, and `Symbol` are guaranteed to be available. Note: Internally, a string is immediately converted into the corresponding set of curves using methods from the respective Java libraries. If the font size is small, the result may not be very nice. An easy remedy is to choose a larger font size and to scale down afterwards, e.g.,

```
scaleDown = scale(1/12),
"This is nicer than font size 1" size 12 at (0,0) transformed by scaleDown
```

A definition of the form $f = \langle \text{transformation} \rangle$ makes f denote an affine transformation $f_1 \cdot \dots \cdot f_n$ composed of primitive transformations f_1, \dots, f_n , which are applied from left to right. Such a definition is not directly visible from the outside.

A definition $f = \langle f_1, \dots, f_n \rangle$, where f_1, \dots, f_n are the names of transformations defined earlier in the file, means that the collage algebra will interpret the symbol $f:n$ as the collage operation $\langle f_1 \dots f_n \rangle$. The case $n = 0$ yields a constant denoting the empty collage. Thus, $f = \langle \rangle$ is equivalent to $f = \{ \}$. Any f_i can in fact also be a *collage-union* rather than a transformation. These collages are then added to the result whenever the operation is applied. For instance, if the definition reads $f = \langle f_1, \#col, f_2 \rangle$, where `col` is the name of a collage defined earlier, then f is the operation of arity 2 given by $f(C_1, C_2) = f_1(C_1) \cup col \cup f_2(C_2)$. (Notice that collage displays will draw the three sub-collages one after another. Therefore, it can make a difference where `col` is placed if they overlap.)

Finally, $f = \text{colourOperation}(\dots)$ defines a symbol f of rank 1 that changes the respective colour attributes in the way explained earlier. In its generalized form, $\langle \text{attr expr} \rangle$ is a superset of $\langle \text{expr} \rangle$ in the sense that folding (as explained above) is available as an

additional operation `fold(<rat>)` and `val(<name>)` can be used to refer to the value of an attribute.

It is allowed to use equal names for different operations as long as their arities differ. The name space used for transformations is a separate one. Of course, the names of transformations should be pairwise distinct. For example, the following would be a correct definition of a collage algebra:

```
{ f1 = scale(.5),
  f2 = scale(.8) . rotate(12) . translate(3,-2),
  f3 = map((1,0) -> (1,0), (0,1) -> (.5,1)),
  flip = scale(-1,1),
  C = { filledCurve((0,-2), (-1,0), (0,2), (1,0))<.2,.3,.4> },
        % a polygon, in fact
  F    = <f1, f2, f3>,
  flip = <flip>,
  darken = colourOperation(r|.2:0, g|.2:0, b|.2:0) }
```

Every symbol $s:n$ which is not defined explicitly is interpreted as a union of collages, i.e., as $\langle id \cdots id \rangle$, where id is the identity.

4.3.4 The geometric-world algebra (class *applications.geoWorld.geoWorldAlgebra*)

This is a simple class implementing an algebra whose domain consists of geometric objects. There are two primitive objects. Given two objects, they can be arranged horizontally or vertically. Moreover, given an arbitrary arrangement of objects, they can be composed into one object by capturing them by a rectangular frame. The algebra interprets

- the symbols **a:0** and **b:0** as the two primitive geometric objects (a squarish one resp. an octagonal one),
- the symbols **h:2** and **v:2** as operations that take as input two arrangements x, y of objects and place x to the left of y resp. x below y (separated by a small gap and centered vertically resp. horizontally),
- the symbol **c:1** that captures an arrangement by means of a rectangular shape, thus creating a composed object.

In addition, every symbol $A:0$ with $A \notin \{\mathbf{a}, \mathbf{b}\}$ is interpreted as a square with the string A in its centre. The font size is appropriate for symbol names of length 1 but can be adjusted interactively by the commands `smaller font`, `larger font`, and `reset font`.

4.3.5 Grid collage algebras (class *applications.collages.gridAlgebra*)

This class implements the two-dimensional case of the algebras investigated in [Dre96]. A grid collage algebra (grid algebra, for short) is a special collage algebra. Such an algebra interprets a predefined set of symbols, which means that the syntax gets much simpler as there is no need to define the meaning of symbols. Basically, the defining parameter of a

grid algebra is simply a positive integer, its *grid size*. A grid algebra of grid size n is also called an n -grid algebra.

The only non-empty collage available as a content is the unit square sq (i.e., the collage having as its only part the black unit square). The available collage operations, which are called *grid collage operations* are defined as follows. Divide the unit square by a regular $n \times n$ -grid, so that n^2 smaller squares of equal size are obtained. Now, the n -grid algebra contains all collage operations F consisting of m similarity transformations f_1, \dots, f_m , such that each f_i maps sq to one of the small squares in the grid.

Remark. In [Dre96] it is also required that $f_i(sq) \neq f_j(sq)$ for all $i \neq j$, i.e., that an operation addresses every subsquare at most once. This restriction is dropped, here. Moreover, instead of considering the unit square, in the implementation everything is enlarged and centred at the origin, because this fits better with the default area shown by collage displays.

In an n -grid algebra each of the aforementioned operations can be denoted using a standard symbol, but the syntax also provides the opportunity to introduce more concise synonyms. The standard symbols for an n -grid algebra are determined as follows.

One of the standard symbols is $\mathbf{sq:0}$, which denotes the unit square. The other standard symbols denote grid collage operations. For this, number the n^2 subsquares of the grid from the bottom left one to the top right one, starting with 0, and denote the corners of the unit square by **a**, **b**, **c**, and **d** (counterclockwise, starting at the lower-left corner). For example, for $n = 3$ we get the following picture:

d				c
	6	7	8	
	3	4	5	
	0	1	2	
a				b

Now, consider one of the admissible transformations of an n -grid operation. Such a transformation is composed of a rotation and/or reflexion f about the centre of the square that maps sq to itself, and a scaling and translation that maps sq to one of its subsquares. Thus, the whole transformation is uniquely determined by the string ixy , where

- (1) i is the index of the addressed subsquare and
- (2) $x, y \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$ are the images of **a** and **b** under f .

For example, if $n = 3$ then $\mathbf{5bc}$ denotes the transformation that rotates sq by 90 degrees and maps it to square number 5 in the figure above. Let F be an m -ary operation of the n -grid algebra and let s_1, \dots, s_m be the strings denoting the individual transformations that define F . Then, every symbol of rank m whose name has the syntactic form

$$-*s_1-* \dots -*s_m-*$$

is interpreted as F . Thus, the name of the symbol is composed of the strings s_i , optionally separated by some $-$'s. In particular, all symbols which solely consist of $-$'s denote the

empty collage. As another example, the 3-grid algebra interprets `1ab-5dc-8cd:3` as the collage operation whose first transformation maps *sq* to square number 1 without rotating or reflecting it, whose second transformation maps *sq* to square number 5, reflecting it at the horizontal axis, and whose third transformation maps *sq* to square number 8 while rotating it by 180 degrees.

For reasons of backward compatibility, transformations may also be denoted by a number followed by three rather than two symbols in $\{a, b, c, d\}$, these three being the images of *a*, *b*, and *c* (where the last one provides redundant information).

In addition, symbols of rank 1 or 0 which are not standard (and are not used as synonyms either, see below) are interpreted as follows. Symbols of rank 1 are simply interpreted as the identity. Symbols of rank 0 yield a unit grey square with an inscribed arrow indicating the order $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$ of corners. This yields a graphical representation of nonterminals similar to the one employed in [Dre96].

The syntax of a grid collage algebra either consists of a single non-zero $\langle nat \rangle$, the grid size, or is a pair consisting of the grid size and a set of definitions of two kinds. Definitions of the form $\langle name \rangle = \langle name \rangle$ allow the user to introduce a synonym (the left-hand side) for some standard symbol (the right-hand side). Note that ranks are not given explicitly because they are uniquely determined by the name of the given standard symbol. The synonyms introduced in this way should be pairwise distinct, and they must not themselves be standard symbols. Thus, it is not allowed to define `1ab-3bc = 2ab-1ab` (since `1ab-3bc:2` is a standard symbol), but it *is* allowed to define `1ab-3bc = 2ab` (since `1ab-3bc:1` is not standard). Defining both $F = 1ab$ and $F = -$ is allowed (because it results in symbols of different ranks), but having both $F = sq$ and $F = -$ is incorrect.

The second type of definitions has the form $\langle name \rangle += \langle part-set \rangle$, where $\langle part-set \rangle$ is defined as for general collage algebras (but with static *rgb* values only). It changes the appearance of non-standard symbols of rank 0, i.e., those which are normally represented as grey squares with inscribed arrows, by replacing the arrow with user-defined parts. The definition `exa += C` thus results in `exa` being interpreted as a grey square with *C* on top of it. For this to work as expected, it is important to know that grid collage algebras actually work with squares whose corners are $(-20, -20)$, $(20, -20)$, $(20, 20)$, $(-20, 20)$. For instance, to interpret `exa` as a grey square with an inscribed red circle, one should use the definition `exa += { filledEllipse(0,0,20,20) [1,0,0] }`.

4.3.6 The integer algebra (class *algebras.intAlgebra*)

The integer algebra interprets every symbol of the form $\langle int \rangle : 0$ as the respective integer. Furthermore, it interprets the following symbols as operations on integers in the usual way: `neg:1`, `abs:1`, `s:1` (successor), `max:2`, `min:2`, `+:2`, `-:2`, `*:2`, `div:2`, and `mod:2`.

4.3.7 The string algebra (class *algebras.stringAlgebra*)

The string algebra interprets every symbol $f:0$ as the string given by *f*. Furthermore, every symbol of rank $n \geq 1$ is interpreted as *n*-ary string concatenation. (Thus, using common terminology, the interpretation of a tree with respect to the string algebra is its *yield*—the string of leaf symbols read from left to right.)

4.3.8 The free term algebra (class `algebras.termAlgebra`)

As usual, the free term algebra interprets every symbol $f:n$ as the n -ary operation on trees (i.e., terms) that maps t_1, \dots, t_n to $f[t_1, \dots, t_n]$. Thus, the interpretation of a tree is the tree itself.

4.3.9 Turtle algebras (class `applications.lineDrawings.turtleAlgebra`)

Besides the chain-code algebra, the turtle algebra is the second sort of line-drawing algebra available so far (see also [Dre98b]). The syntax of turtle algebras is either of the form

$$(\alpha_0, \alpha \ [\ <synonyms> \] \)$$

where α_0, α are positive angles given in degrees (two $\langle rat \rangle$'s), or it is simply given by the angle α (without the parantheses). In the latter case, α_0 is set to 90 degrees. The optional part $\langle synonyms \rangle$ is described later.

Like the chain-code algebra, a turtle algebra interprets every n -ary symbol as n -ary concatenation of line drawings. However, there are the following exceptions of symbols with a special meaning:

- **F:0** is interpreted as the line drawing consisting of a single line extending from the origin one unit into the direction given by α_0 (i.e., upward by default); **f:0** is its “invisible” counterpart. It denotes the empty line drawing having the same endpoint as **F:0**.
- **hide:n** is interpreted as in the chain-code algebra. It concatenates the arguments and makes all lines of the resulting line drawing invisible.
- **h-scaler:n**, **v-scaler:n**, and **scaler:n**, where r is a $\langle rat_+ \rangle$, are interpreted in the same way as in the case of chain-code algebras.
- **enc:n** concatenates its arguments and then sets the endpoint to $(0,0)$.
- **+:n** concatenates its arguments and rotates the resulting line drawing by α degrees counterclockwise; **-:n** is interpreted alike, but rotates clockwise.
- **enc+:n** combines the two above. Thus, it is interpreted like **+:n**, but sets the endpoint of the resulting line drawing to $(0,0)$, in addition. Again, **enc -:n** is similar, but yields a clockwise rotation.
- **+branch:n** and **-branch:n** are synonyms for **enc+:n** and **enc -:n**, respectively.
- For convenience, each ‘+’ or ‘-’ in one of the aforementioned operations may be preceded by a factor $k > 0$, a $\langle nat \rangle$, which affects the angle α used. The symbol is interpreted in the same way as if it was not preceded by k , except that the angle $k\alpha$ is used.

There is, in fact, another type of operation which allows to go beyond pure line drawings:

$$\text{fill } r-g-b:n$$

(where r, g, b are rgb values between 0 and 1) concatenates its arguments and fills the resulting polygon (taking into account both visible and invisible lines) with the colour determined by r, g, b . The symbol **fill v:n** abbreviates **fill v-v-v:n**.

The optional part $\langle \text{synonyms} \rangle$ of the syntactic definition makes it possible to specify two signatures of symbols of rank 0 that the algebra interprets like **F** and **f**, respectively. The syntax is

```

<synonyms> ::=
  with
  F-synonyms { [ <name> (, <name>)* ] },
  f-synonyms { [ <name> (, <name>)* ] }

```

where either the **F-synonyms** part or the **f-synonyms** part can be left out.

4.3.10 YIELD algebras (class `algebras.YIELDAlgebra`)

A YIELD algebra (see, e.g., [ES77, ES78, DE98]) provides certain operations on trees, substitution being the most important one. The following are the standard symbols that all YIELD algebras interpret in a special way:

- **const- f - k :0**, where f is a $\langle \text{name} \rangle$ and k a $\langle \text{nat} \rangle$,
- **proj- i :0**, where i is a $\langle \text{nat} \rangle$, and
- **subst: n** , where $n > 0$ is a $\langle \text{nat} \rangle$.

The interpretation of **const- f - k :0** is $f[x_1, \dots, x_k]$, **proj- i :0** is interpreted as x_i , and the interpretation of **subst: n** yields the function g on trees such that $g(t_0, t_1, \dots, t_{n-1})$ is obtained from t_0 by substituting t_i for x_i , $i = 1, \dots, n - 1$.

As in the case of grid algebras the syntax allows to introduce synonyms for the symbols above. Thus, syntactically, a YIELD algebra is a set of definitions of the form $\langle \text{name} \rangle = \langle \text{symbol} \rangle$. The meaning of these definitions as well as the restrictions that apply are similar to the case of grid algebras. Notice, however, that the right-hand side is a symbol in this case. This is necessary because, unlike the case of grid algebras, the name of a symbol does not determine its rank in a unique way.

For convenience, all symbols $f:n$ which do not have a special meaning (i.e., which do not belong to the three types above and have not been defined as a synonym) are interpreted as in the free term algebra. In other words, the application of f to terms t_1, \dots, t_n yields $f[t_1, \dots, t_n]$. As an example, the evaluation of

```
f [subst [g [f [proj-2], proj-2], const-h1-2, const-h2-2]]
```

yields $f[g[f[h2[x1, x2]], h2[x1, x2]]]$, discarding the evaluation result of **const-h1-2** since the first argument of **subst** does not contain **proj-1**. Note that a symbol **const- f - n** can always be replaced with $f[\text{proj-1}, \dots, \text{proj-}n]$ unless $f:n$ has a special meaning or is used as a synonym.

4.4 Displays

4.4.1 Collage displays (class `applications.collages.collageDisplay`)

A collage display can either automatically zoom the displayed collage to an appropriate size and centre it in its window, or let the user resize and position the picture manually.

The latter is the default mode, providing the commands *zoom in*, *zoom out*, *auto zoom*, and *reset* (reset the view). Furthermore, in this mode the user can

- use horizontal and vertical scroll bars if the picture is larger than the visible part,
- drag the picture by holding down the first mouse button and moving the mouse (instead of using the scroll bars), and
- zoom into a desired area by holding down the second or third mouse button and dragging the mouse over the area in question.

The area shown initially is the square with lower left corner $(-25, -25)$ and side length 50. In *auto zoom* mode one can switch back to the default mode using the command *manual zoom*.

In addition to producing a picture on the screen a collage display can be set up in order to produce PostScript files which describe the displayed collages. In such a PostScript picture, one centimetre corresponds to one unit in the collage.

The syntax of collage displays is

```
<collage display> ::=
  [background <rat>,<rat>,<rat>]
  (postscript disabled | <PostScript settings>)

<PostScript settings> ::=
  pathname "<directory name>", filename "<file name>"
  [, linewidth <rat> (cm | mm | pt)]
```

The optional part at the beginning, signified by the keyword **background**, defines the background colour. The colour model used is the rgb model; the numbers are required to belong to the interval $[0, 1]$. If no background colour is given, the default background colour of the Java runtime environment is used (normally white).

In the remainder of the syntax, the keywords **postscript disabled** will disable PostScript output. Otherwise, the path name (which is, unless it is an absolute path, interpreted relative to the directory in which TREEBAG was started) determines the directory into which the PostScript files will be written. Here, the separator character for directory names is always ‘/’, regardless of the platform on which TREEBAG is running. In the file name the character ‘#’ is regarded as a special symbol. Every time a PostScript file is produced, each occurrence of this character is replaced by a running number. Thus, for example, *out-#* would result in output files named *out-1*, *out-2*, etc. Optionally, one can define the width of lines in centimetres, millimetres, or points. The default line width is “as narrow as possible”.

If PostScript output is enabled, the following additional commands are available: *ps-output/no ps-output* (start/stop producing an output file whenever an input tree with a defined value is received), *reset ps-output* (reset the number of the next file to 1), and *clipping on/clipping off*. Without clipping, the whole collage is written to the PostScript file and the bounding box is set up accordingly. When clipping is active, only those parts are written to the file of which at least one point lies in the display window. In this

case, the bounding box will correspond to the window area. (For L^AT_EX users: Note that packages for the inclusion of PostScript pictures, like `graphicx`, usually have a `clip` key that allows you to clip off parts lying on the boundary of the viewing window.)

4.4.2 The geometric-world display (class `applications.geoWorld.geoWorldDisplay`)

This display class is intended to display the geometric arrangements produced by the algebra `applications.geoWorld.geoWorldAlgebra`. It works in a similar way as a collage display (see above) since it is actually a subclass of that class.

4.4.3 Line-drawing displays

(class `applications.lineDrawings.lineDrawingDisplay`)

Line drawing displays are able to visualise the line drawings produced by chain-code and turtle algebras. A line drawing display behaves similar to a collage display in *auto zoom* mode. Thus, the picture is always centred and scaled to an appropriate size. Line drawing displays are able to produce PostScript in the same way as collage displays do. The syntax of line drawing displays is the same as the one for collage displays; the available commands are the commands to control PostScript output as described above for the collage display.

4.4.4 The textual display (class `displays.textualDisplay`)

Textual displays can visualise every kind of object, using its string representation. This is sensible for strings, numbers, truth values, and trees. If the character sequence `\\` occurs in a displayed string it is interpreted as a line break (but it is also possible to include newline characters in the string, directly.) The syntax for textual displays is the empty string. The available commands are *larger* (enlarge the font size), *smaller* (reduce the font size), and *adjust window size* (adapt the window size to the size of the displayed text).

4.4.5 Tree displays (class `displays.treeDisplay`)

A tree display visualises trees in the usual way, applying a simple layout algorithm. These displays are not really suitable for very large trees (more than 2^{12} nodes, say), because the standard Java classes that do the actual painting seem to deal incorrectly with large coordinates. In these cases it usually helps to reduce the font size.

The available commands are *larger* (enlarge the font size), *smaller* (reduce the font size), *smallest* (reduce the font size as much as possible, which is useful in order to view the structure of large trees), and *reset view* (return to the default view and font size). In addition to these commands the user can drag the picture using the mouse or use scroll bars to achieve a similar effect (if parts of the tree lie outside the visible area).

By default, every symbol in a displayed tree is shown in black. It is possible, however, to assign individual colours to certain symbols. For this, the syntax of tree displays requires a set of colour definitions of the form `<symbol> [<rat>, <rat>, <rat>]`, where the three numbers are rgb values between 0 and 1. For example, if the definition of a tree display contains the expression `foo:2[0,0,.3]`, then the symbol `foo:2` will be shown in dark blue colour.

References

- [DE98] Frank Drewes and Joost Engelfriet. Decidability of the finiteness of ranges of tree transductions. *Information and Computation*, 145:1–50, 1998.
- [DE04] Frank Drewes and Joost Engelfriet. Branching synchronization grammars with nested tables. *Journal of Computer and System Sciences*, 68:611–656, 2004.
- [Dre96] 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.
- [Dre98a] Frank Drewes. TREEBAG—a tree-based generator for objects of various types. Report 1/98, Univ. Bremen, 1998.
- [Dre98b] Frank Drewes. Tree-based picture generation. Report 7/98, Univ. Bremen, 1998. Revised version appeared in *Theoretical Computer Science* 246:1–51.
- [Dre98c] Frank Drewes. TREEBAG – Baum-basierte Generierung und Transformation von Objekten. In J. Dassow and R. Kruse, editors, *Proceedings of the Informatik '98*, Informatik Aktuell, pages 47–56, 1998. In German.
- [Dre99] Frank Drewes. Tree-based generation of languages of fractals. Report 2/99, Univ. Bremen, 1999. Revised version to appear in *Theoretical Computer Science*.
- [Dre05] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005. To appear.
- [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.
- [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.
- [FDF⁺97] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1997.

- [FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer, Berlin, Heidelberg, 1998.
- [GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*. Vol. 3: *Beyond Words*, chapter 1, pages 1–68. Springer, Berlin, Heidelberg, 1997.

A Lists of available classes of TREEBAG components

A.1 Tree Grammars

Component	Class name
Regular tree grammar	<code>generators.regularTreeGrammar</code>
Parallel det. total tree grammar	<code>generators.pdtGrammar</code>
ETOL tree grammar	<code>generators.ETOLTreeGrammar</code>
branching tree grammar	<code>generators.BSTGrammar</code>

A.2 Tree Transducers

Component	Class name
Top-down tree transducer	<code>generators.tdTransducer</code>
YIELD transduction	<code>generators.YIELDTransduction</code>
Macro tree transducer	<code>generators.mtTransducer</code>
Iterator	<code>generators.iterator</code>

A.3 Algebras

Component	Class name
Boolean algebra	<code>algebras.booleanAlgebra</code>
Chain-code algebra	<code>applications.lineDrawings.chainCodeAlgebra</code>
Collage algebra	<code>applications.collages.collageAlgebra</code>
Geometric-world algebra	<code>applications.geoWorld.geoWorldAlgebra</code>
Grid collage algebra	<code>applications.collages.gridAlgebra</code>
Integer algebra	<code>algebras.intAlgebra</code>
String algebra	<code>algebras.stringAlgebra</code>
Free term algebra	<code>algebras.termAlgebra</code>
Turtle algebra	<code>applications.lineDrawings.turtleAlgebra</code>
YIELD algebra	<code>algebras.YIELDAAlgebra</code>

A.4 Displays

Component	Class name
Collage display	<code>applications.collages.collageDisplay</code>
Geometric-world display	<code>applications.geoWorld.geoWorldDisplay</code>
Line-drawing display	<code>applications.lineDrawings.lineDrawingDisplay</code>
Textual display	<code>displays.textualDisplay</code>
Tree display	<code>displays.treeDisplay</code>