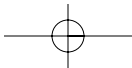
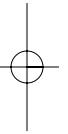
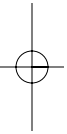


Do not print this
page

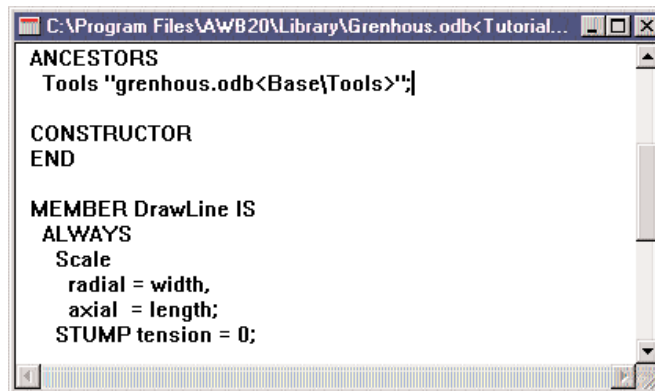




L-Systems

New Features	225
Making Plants	240
Generative Grammars	246
Modular Programming	252
Implementation	260
Program Environment	275

Most Aspects of L-Objects remain the same in World Builder 2.0. The changes reflect the new World Builder interface and include some new Functions and Terminals along with some syntax additions.



```
C:\Program Files\AWB20\Library\Grenhous.odb<Tutorial...
ANCESTORS
  Tools "grenhous.odb<Base\Tools>";

CONSTRUCTOR
END

MEMBER DrawLine IS
ALWAYS
  Scale
    radial = width,
    axial = length;
STUMP tension = 0;
```

NEW FEATURES IN VERSION 2.0

Syntax extension

L-Objects can now contain end-line comments in the C++ style. Such comments start with double slash "//" and stop at the end of the line.

Example:

```
Redwood age = 2000; //this is an end-line comment
```

All the text inside end-line comment is ignored. The only exception is a pseudo-comment with target directive.

Target Directive

```
//TARGET=RT_PLANT  
//TARGET=PLANT
```

Regular plants in the AWB 1.0 style and time-dependent (evolving) plants can be created using L-Objects. The default is a regular plant. To make evolving plants, load L-Object code into the editor, and place the target specification directive //TARGET=RT_PLANT at the very beginning of the L-Object code on a separate line. The directive //TARGET=PLANT will give the same effect as no directive at all, i.e. a static plant will be generated.

For evolving plants, compiling the code will create a special object that keeps a copy of the L-Object code in memory and rebuilds the plant for each new frame. The appearance and behavior of this time-dependent object does not differ from that of a regular plant. The only exception is that manual changes to plant geometry made by the local plant manipulator will be lost each time the frame changes. Wind, on the other hand, works with evolving plants.

Note: See the description of the pseudo-constant `_T` used to access the current frame number from L-Object code.

Warning: While making a plant evolve in time you must pay special attention to materials since they are loaded only once: after the plant was generated from L-Object code the very first time. If on different stages of evolution the plant uses different set of materials, you must force the L-Objects interpreter to load all of them regardless of the frame number. To implement this, use the following trick:



L-Systems

Declare a non-Terminal, such as RegisterAllMaterials in the MEMBERS section. Define it as follows:

MEMBERS

RegisterAllMaterials;

MEMBER RegisterAllMaterials IS

ALWAYS

MATERIAL file="my_materials.odt",
resource="sample1";

MATERIAL file="my_materials.odt",
resource="sample2";

```
//place here references to all materials used //in the
plant
END
END
```

Call RegisterAllMaterials at the very beginning of the constructor. It is also good to call RegisterAllMaterials for inherited objects, not just to copy code. This keeps everything automatically consistent.

The practice with registering materials is also useful for making several plants of the same kind, which nevertheless use different set of materials. With materials registration, all plants of this kind will have the same materials set and can be used in the Area Editor altogether.

References to object data-bases

World Builder can store and access L-Objects and their components in object data bases. In particular, ODB files can store L-Objects, and parts of L-Objects as plain text, as well as Functions defined by their plot. Also materials in libraries can be accessed from L-Objects. This means that you can inherit L-Objects from databases and easily access them relying on their graphical representation by icons rather than on their names.

In order to support this new feature, the syntax of string constants was extended.

Inheriting objects located in odt

In the section ANCESTORS you can specify a path name to .L files with the ancestor code; or a path to the object database accompanied by the complete path inside the database. The combined path is one string constant. The path inside database is enclosed by angular brackets and delimited by "\" characters:

```
"path_to_odt_file<path_inside_odt>"
```

The library Root node is not included in the inside-library path. In-libraries the path is case sensitive. The path to the file itself can be reduced to just a name, if the file is accessible via redirection (internal World Builder path).

Example:

```
ANCESTORS
  None "none.L",
  StartUp0 "grenhous.ocb<Base\StartUp>;
```

In the example, the object None is inherited as in the previous version. The object StartUp0 is inherited using the new method which uses a database object located at "grenhous.ocb". The name of the object in the .ODB is StartUp and it resides in the Base section.

The MATERIAL Terminal now also accepts an extended form of resource parameter. If the file parameter refers to an .ocb file, then the resource parameter must specify the complete path inside the .ODB file. The resource path in this case should not contain angular brackets

Example:

```
MATERIAL
  file="Grenhous.ocb", //its location must be on
                      //the redirection path
  resource="Materials\Bark\BrownTrunk0"; //no angular //brackets here
```

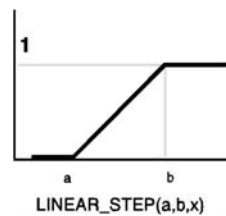
The same convention works for the IMPORT Terminal.

New built-in functions

The six new built-in functions for L-Objects are: LINEAR_STEP, SMOOTH_STEP, PULSE, STEP, MIN, MAX.

LINEAR_STEP(a,b,x)

Returns 0 if $x < a$, returns 1 if $x > b$, otherwise returns $(x-a)/(b-a)$.

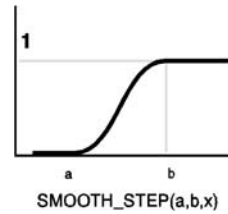




L-Systems

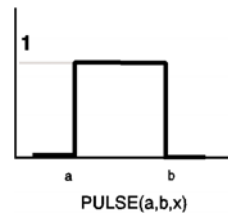
SMOOTH_STEP(a,b,x)

Same as linear step, except it grows from 0 to 1 non linearly. It is designed to make the derivative continuous and to make the curve smoother. Between a and b it returns $t*t*(3.0-2.0*t)$, where $t=(x-a)/(b-a)$:



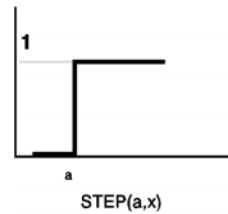
PULSE(a,b,x)

Returns 0 if $x < a$ or $x > b$; otherwise returns 1.



STEP(a,x)

Returns 0 if $x < a$ and 1 if $x >= a$.



MIN(a,b)

Returns the minimum of two values.

MAX(a,b)

Returns the maximum of two values.

Predefined parameter (pseudo-constant) `_T`

Always equals the current animation time in World Builder. For example, at frame 120 it is equal to 120, and for frame 0 it is equal to 0. The value of `_T` is used during building the plant, so if the L-Object code uses `_T`, then the results of its interpretation will be different for different animation frames. There can be no user-defined Object parameters with the name `_T`.

This parameter is used for making evolving plants see the Rose example in "greenhous.cdb".

New Terminals

FORCE

Simulates a force applied to the Turtle. The action of this Terminal is similar to the action of AXIS with non-zero rotations, since it also changes orientation of the Turtle. The difference is that AXIS rotation parameters are specified in the current local coordinates of the Turtle, and for the FORCE Terminal parameters are in the coordinates of the plant. Also AXIS specifies the rotation explicitly by angle, but the FORCE specifies some "strength" with which the Turtle is rotated.

Parameters:

`fieldType` specifies what kind of coordinates are used for force definition. Valid values are 0-cartesian coordinates (gives uniform isotropic force field), 1 cylindrical coordinates (gives axial symmetric force field) and 2 spherical coordinates (spherically symmetrical force field). Default value is 0.

`dirX`, `dirY`, `dirZ` are three coordinates of the force vector in the chosen coordinate system. Default value is 0.

`axis` selects the Turtle axis affected by the force. Valid values are 0-X, 1-Y, 2-Z. Default value is 0.

`strength` is the strength with which the Turtle is rotated. For higher values the selected axis tends to turn in the force direction closer and closer. Setting `strength` to, say, 100000 will make the Turtle axis turn almost exactly in the force direction. Default value is 0.

Example 1: Downward directed Cartesian vector field simulates gravity:

```
FORCE
```

```
    fieldType=0, dirX=0, dirY=0, dirZ= -1, strength=1;
```

Use this to simulate branches sagging down.

Example 2: Axial field giving constant clockwise force will twist/bend plant parts around upward direction:

```
FORCE //twists plant counter clockwise around vertical           //axis
    fieldType=1, dirX=0, dirY=1, dirZ=0,                       strength=1;
```

Example 3: A force applied to Y-axis will turn leaf to face direction of the force:



L-Systems

FORCE

```

    fieldType=0, axis=1, dirX=0, dirY=0, dirZ=1,
    strength=1;
    LEAF; //here leaf will be slightly turned to make its //surface face
    upward
    //The feature can be used together with LIGHT Terminal //to simulate the effect
    of light on plants
  
```

POS

A Terminal used to get information about the Turtle's location in space and, conversely, to set the new location of the Turtle.

Parameters:

x,y,z cartesian coordinates of the Turtle in plant's coordinate system. Coordinates do not have default values.

Since the Terminal can be used to set and to get the position, the meaning of its parameters depends on the context. To get the position of the Turtle just use POS.x, POS.y and POS.z as usual variables in the right side parts of assignments and in predicates.

Example:

```

MEMBER SampleOfPosUsage IS
  IF (POS.z>0) THEN //ask about current height of
    //the Turtle location
    STUMP length = POS.x; //use Turtle's X
    //coordinate as stump length
  END
END
  
```

Also you can use POS to set the new position for the Turtle. This Terminal does not change the Turtle orientation.

Example:

```

POS x=7, y=2, z=3; //moves the Turtle to the point
    //(7,2,3)
//after that operation POS.y would return a value of 2
  
```

PLOT

Allows the use of functions specified by their free-hand drawn plot. The function must reside in some object database. Creating and changing functions is explained in the next section.

Parameters:

`name` unique name of the function object with complete path to it. The name must obey rules of redirection and path conventions.

`x` specifies argument of the function

`y` returns function value at current `x`.

Example 1:

```
MEMBER ShapeTrunk IS
  ALWAYS
  PLOT
    name="Grenhous.odbc\Plots\SampleFunction", //selects function object
    x=height; //sets the argument to the
              //current height
  Scale
    radial = PLOT.y, //extracts the value and
                  //sets it to the radial scale
    axial = scale;
  STUMP;
  END
END
```

Example 2:

```
PLOT //select function object
  name="Grenhous.odbc\Plots\SampleFunction";
//once the function is specified, you can set the //argument and extract
value in a cycle without re- //setting the function name
SEQUENCE i=1,100 OF
  PLOT x=i; //sets the argument
  Dummy value = PLOT.y;
//But be careful about //changing
PLOT.name inside Dummy!
END;
```

LIGHT

Provides L-Objects with information about lighting conditions at the current Turtle position. It does not change either Light sources location or other parameters.

Parameters:

`name` is a case-sensitive string parameter. It selects the light source in



L-Systems

the scene by name. For unambiguous use, all light sources in the Scene must have unique names. Zero values are returned for other parameters if there is no light source with the specified name in the Scene.

dirX, dirY, dirZ three components of the unified vector of direction from the current Turtle location to the selected light source. The vector is presented in plant coordinate system.

red, green, blue components of the light intensity at the Turtle position. Shadow maps affect these values. Thus LIGHT Terminal can be used to model the effect of environment on the plant evolution: parts of a plant in shadow will grow slower. The range of red, green, blue components is typically 0 1; however for light sources with intensity higher than 1, these values can also be higher 1.

Example 1: The following code queries LIGHT about its position in order to turn the leaf to the light.

```

LIGHT name = "LightSource0"; //select light source. It
//must be present in the Scene.
FORCE
  fieldType = 0, //Cartesian coordinates
  //now turn the force vector in the direction of light
  dirX = LIGHT.dirX,
  dirY = LIGHT.dirY,
  dirZ = LIGHT.dirZ,
  strength = 1,
  axis = 2; //and apply force to the Y axis to turn
//the leaf surface to the light
HERMITLEAF;

```

Example 2: Assuming that the light source has (r,g,b) intensity (1,1,1) check for the shadow:

```

//SomeFoliage is non-Terminal for foliage which grows only //when not in shadow
MEMBER SomeFoliage IS
  IF (LIGHT.red =1) THEN //due to the assumption we can
//check only one component
    DoSomeFoliage; // make some unconditional foliage
  END
//do nothing if the Turtle is in shadow no foliage
END

//to use SomeFoliage, you must set LIGHT.name to a
//specific light
LIGHT name = "LightSource0";

```

```
SomeFoliage; // LightSource0 intensity is used in the
//predicate here
```

Function object

The Function Object is a special library-resident object currently used only for L-Objects. It represents functions $[0, 1] \rightarrow [0, 1]$ in a graphical way. For arguments less than 0, the value at 0 will be returned, for arguments greater than 1, the value at 1 will be returned.

Create a Function Object procedure

1. Open a library or make a new one. Make sure that you uncheck "read only" if you are opening a library. You can not edit libraries opened for read only. In the right panel of the library pane, click the right mouse button, and select "Create Objects|Function" in the pop up menu.

A new icon with question sign will appear and you will be prompted for a name of the new object.

2. Type in the name and double-click the icon.

The properties part of the Properties Editor will display three items: Icon, Animated Icon and Edit.

Warning: Do not use the Delete key while typing in the name, since this is interpreted as a command for deleting the object from a database.

3. Select Edit. Then press Edit Properties in the new property dialog.

A sub-node Function Plot will appear under the Edit property in the property tree.

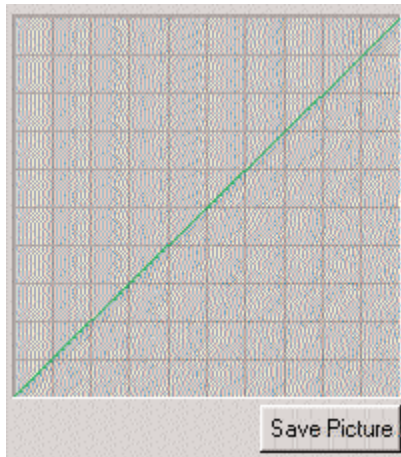


4. Select Function Plot.

A new property page will appear. The green line here represents the plot of default function $y=x$.



Function Plot
Property Page



5. click inside the grid area of the dialog.

Immediately the function value at the place where you clicked will change and the new value will be such that the plot will go through mouse cursor. While you move mouse, function will change. A pair of current X and Y values is shown under the grid area, when the mouse is inside this area. To stop editing, click again inside the dialog. You can enter edit mode and exit it as many times as required.

"Save picture" saves the picture to the file and the picture may be used for the function icon.

6. Click "Save Picture".

The Save file dialog box will appear.

7. Select a file and press OK. Go to the Icon property page and click the Load button. Select a file you just saved.

A new icon will be assigned to the function object. This icon will replace default icon with question sign.

8. To save editing, click Store Object in the Edit page.

Programming Environment

As in the previous version, all editing of L-Objects is performed in a text editor window. However, the new environment offers several features of visual programming, which simplifies and speeds up plant design.

Due to the ability to store L-Objects in .odb files you have two possible ways to start a new L-Object.

Traditional method:

1. Click File/New and select L-Object in the list. Click OK.

A window with text editor will appear.

Note: You must have up-to-date versions of two World Builder plug-ins LSYSTEM.BEM and PLN_TERM.BEM. Both of them must be located either in the World Builder executables directory or at any location specified in redirection as the bem section. If you don't have these plug-ins, then the list with L-Object options will not appear.

New odb-based method:

1. Open a new or existing object database for modification. In the right pane make a right mouse click. In the context menu select Create Objects/L-Object.

A new icon with question sign will appear and you will be prompted for a name for it.

2. Type a name for the new object. Under this name the object can be found and referenced in the library. (However the L-Object itself can have a different name.) Double click the icon. In the Property Tree of the Properties Editor select the Edit Property. It will display a new Property page. Press the Edit Properties button on this page.

A new window with text "OBJECT None END" will appear. You can add a new icon the object in the standard way.

Despite the fact that this way seems to be longer, it has certain advantages. First, all L-Objects related to an application can be placed in one database along with materials and other 3D stuff. Second, you can supply them with icons, which helps to speed searching for a required object. Last, but not least, you can inherit objects from the object database by a simple drag-and-drop procedure into the L-Objects editor.

Right after opening of the L-Object editor window there will be an extra item in the top dropdown menu. The item is called L-Objects and contains the following commands: Create L-Objects template, Expand name, and Compile L-Object.

Making an object template procedure

Once the L-Object editor window is open, you can place there a template for a new L-Object definition.



L-Systems

1. Press Ctrl T while in the L-Object editor window.

A new text with basic sections of L-Object definition will appear. You can undo this operation by pressing Alt+Backspace.

Compiling L-Objects

The compilation command will save all L-Object documents currently loaded in all editor windows (unlike in version 1.0, where you had to save each window manually). After that a compiler will be invoked. If there are no syntax errors, a plant or evolving plant will be created and placed into the scene.

Note: An evolving plant keeps its own copy of the L-Object definition, but it does not have copies of inherited or imported L-Objects.

If the definition of L-Object contains syntax errors, then a message will be displayed. The L-Object document with an error will be loaded if not already loaded by the user and the highlight will be placed at the error. Messages on syntax errors remain the same as in previous version.

A new error message was added:

```
"file is not l-system"
```

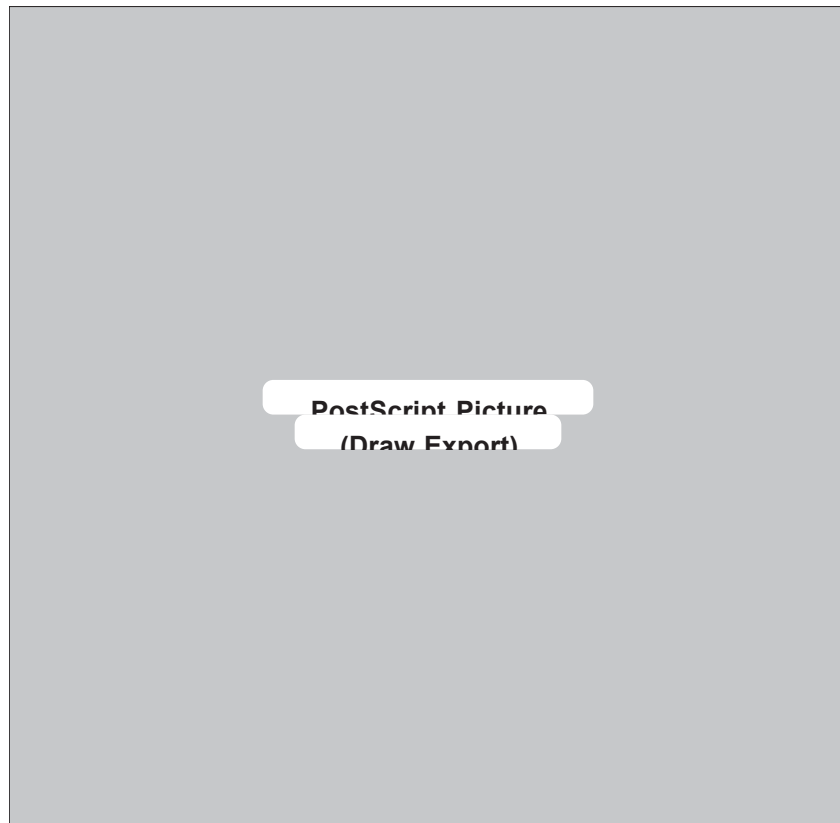
This means that referenced file or database object is not a valid L-Object. The most frequent reason of this message is wrong redirection settings. The next frequent reason is that the referenced file or object does not exist.

Expand Name

Some of words or their parts can be used in the editor to speed up entering code. Select the word or its part and press Ctrl E. If there is no expansion for the selection, the message will appear instead of the selection:

```
<The selection is not extendible. Press Alt+Bksp for undo.>  
Press Alt Backspace for undo.
```

Words and parts that can be expanded are the following.



Some Terminals support visual editing. If the Terminal supports visual editing, then after pressing Ctrl E on the selection containing the Terminal name or its abbreviation, specified in a table, there will appear a preview window and a new page in the Properties Editor. The preview window will display a plant element corresponding to the Terminal. You can adjust projection and resolution of the preview window.

For FILL Terminal the property page will contain Terminal parameters:



Fill Type Property
Page

Any changes in parameters will be at once displayed in the preview window. You can easily adjust parameters by using spinners or typing in new values. Dialog parameters have the same names and meaning as FILL Terminal parameters. After finishing editing, press "Finish and place data into clipboard!". The text in the editor will be updated. Instead of the previous selection, there will be pasted a text from the right column of the table with new parameters values.

The same procedure works for HERMITLEAF Terminal. When you press Ctrl E on one of the left column entries for HERMITLEAF Terminal, the preview window with sample leaf will appear and the property page will contain the dialog:

Hemit Leaf
Property Page

As in the case of FILL Terminal, the dialog contains fields of HERMITLEAF Terminal and allows you to change them while viewing the leaf updated in the preview window. The "Finish and place data into clipboard" button also finishes editing and places text with new parameters into the L-Object code.

If you move the mouse/keyboard focus to some other object in the Scene while using one of these two dialogs, it will disappear since the new selected object will display its own Property Page. You can return to the editor window (click on its top to preserve selection) and press Ctrl E again. The dialog with Terminal parameters will return.

Also you can select an existing Terminal (FILL or HERMITLEAF) and call visual editing for it. Old parameters in the selection will be replaced by new.

Drag and drop

Another feature that adds to the visual aspect of L-Objects programming is drag and drop support. There are two types of items that can be dragged and dropped: L-Object and L-Object Component.

Both items can be created directly at the object database. Click right button at the right pane and select Create Objects menu item. But typically these items represent reusable parts of L-Objects code and are created initially in the editor window.

L-Object This is text containing a syntactically valid definition of an L-Object. It must have the key word OBJECT and the key word END. After selecting text that meets these requirements, you can drag and drop it into the object database. (The library must be open for modification.) The L-Object item is used for inheriting objects. To drop it into the editor you need to have in the editor some L-Object definition with an ANCESTORS section already present. The reference to the library object is placed into the ANCESTORS section after dropping, so the current object will inherit the object from the database. Making a plant that just inherits some basic features from the library objects will be very easy with this drag-and-drop support.

L-Object component In the drag and drop context this is just a fragment of L-Object code. It does not have to contain a complete L-Object definition. This piece of code can be selected, placed into the library and later dropped back into the editor. For example, grenhous.odc contains some useful pieces of code in the Terminals section.



MAKING PLANTS IN WORLD BUILDER

This section explains how to make plants in World Builder. The process of making plants looks very much like programming. It is not assumed that you have any programming experience, though such experience can definitely help.

We introduce the concepts of L-Objects starting from a simple version of LOGO-style 2D graphics and then moving up to the object-oriented version of L-Systems, called L-Objects. In theory you do not need any other books on the subject to understand L-Systems. There is a list of references at the end of the chapter which can help you during your work with L-Objects in World Builder.

As already mentioned, L-Objects are an object oriented version of L-Systems. L-systems are a very powerful tool for mathematical modeling and computer graphics. Among the major applications of L-systems are static models of plants and other objects (see references 1,2, & 3) along with dynamic models of development (see reference 4).

L-systems are already widely used in Computer Graphics. Some examples are commercial plug-ins for 3D Studio and some public domain products. Despite this L-Systems remain a tool mainly for computer science professionals. The reasons become obvious if we compare the evolution of L-systems design with conventional programming.

The evolution of programming languages started with very technical and complex low-level languages which evolved into more understandable and comprehensive high-level languages. Recently, visual and object oriented languages have been replacing the procedural languages for many applications. The modern era of L-systems seems to correspond to the procedural era in programming. According to the logic of this evolution, we can expect the appearance of object-oriented and visual programming techniques for L-systems which will simplify the modeling process.

The current implementation of L-Systems in World Builder corresponds to the beginning of the object-oriented stage. The main goal was to develop a programming paradigm for coding L-systems which supports modularity and incorporates the advantages of inheritance. Both of these new features are aimed at incremental design and code reusability.

Keeping these aims in mind, we extensively used Occam's razor to minimize the number of new notions added to L-systems. This resulted in quite a small

programming language called L-Objects. It has a simple syntax and is more readable than most of its predecessors.

This section is organized as follows. We start with a brief introduction to L-systems. We introduce some non-classical features, which evolves into the object-oriented version of L-Systems. We illustrate the discussion with a set of examples. The last section is devoted to the L-Objects programming environment and describes how this is implemented in the current version of World Builder.

The best way to read this section is to start with a brief look at examples in the Example 2D Turtle Graphics and L-Objects section. Examples present in the text are also aimed to help you in understanding what are sometimes rather abstract notions.

L-systems and L-objects

L-systems are a mathematical modeling tool proposed by A.Lindenmayer in 1968. A comprehensive introduction to classical L-systems can be found in works by P.Prusinkiewicz (see references 1 & 2). For detailed discussion see reference 3. Here we briefly present only the key concepts in a context suitable for further discussion.

L-systems are closely related to turtle graphics and generative grammars.

Turtle graphics LOGO-style graphics

LOGO is a language that produces graphics from a simple set of commands that move a pen or turtle around on the screen. A LOGO-style turtle is a tool for drawing on a plane. It responds to commands like move forward and draw line segment or turn right by angle 90 degrees etc.

Between execution of two sequential commands the turtle preserves its state. A state of the turtle includes Cartesian coordinates of the turtle's position and the heading angle - the direction in which the turtle is facing. We can call the turtle a graphics interpreter or just an interpreter.

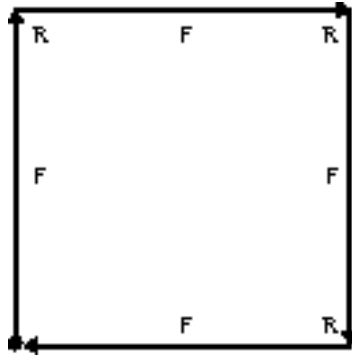
Usually commands are represented by symbols. For example, we can denote line drawing by the symbol F and the turn command by the symbol R. A sequence of commands (a program) corresponds to a word using this alphabet.

Example 1 : The turtle interpretation of the word FRRFRFRF is a square (see figure 1).



L-Systems

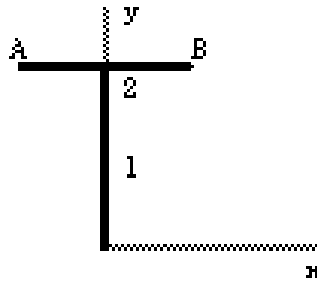
Results of turtle
command:
FRFRFRF



We add two new symbols to the input alphabet of the turtle: [and]. The symbol [makes the turtle push the current state onto a pushdown stack. The right bracket] makes the turtle pop a state from the stack and make it the current state of the turtle.

Example 2 : If initially the turtle was headed along axis Y and was located at (0,0) then the turtle interpretation of the word FFR[F]RRF is a T shape (see figure 2).

The result of the
word: FFR[F]RRF



We can assemble valid input words for the turtle if we use the following two rules:

- There is a one-to-one correspondence between left and right brackets. This means that brackets always come in pairs.
- Each bracket [appears in the word before the corresponding bracket]. You must store the state with a push before you can restore it with a pop.

The depth of bracket embedding is not constrained. You can have as many levels of brackets as you need.

A generalized version of the turtle with structures modeled by words consisting of turtle commands is used in World Builder for modeling trees. A mathematical

tree is a graph (a set of nodes connected by edges) without cycles. Cycles are connections that produce circular references where the connections between nodes will never refer back to a previous node. Words of commands with brackets can be used for building tree-like structures because the space of valid words can be mapped onto the space of mathematical trees in a natural way. (You can imagine a mathematical tree as a tree where you do not care about geometry of its branches. The only concern is the layout of forks.)

We can build an abstract mathematical tree corresponding to a valid word using the following simple procedure.

1. We start from a node, called root.
2. While parsing the word w from left to right we add a new edge line for each symbol if it is not `[` or `]`. We attribute to the edge the corresponding symbol. Thus each edge is marked by a symbol.
3. For each bracket `[` we generate at the current node a branch point. The subword between `[` and corresponding `]` defines a subtree with its root at the branch point.

This procedure defines an abstract tree because it does not specify orientation, length and other parameters of edges.

For the node a we can define a path from the tree root to the node. It is the connected sequence of edges without repetitions starting at the root and terminating at the node. For such path we can write symbols associated with edges in a word. Let us denote it $W(a)$. This word defines a program for the turtle which moves the turtle from its initial location at the tree root to the node a .

Example 3 : The path $W(A) = \text{FFRRRF}$ corresponds to the node A in figure 2 from and $W(B) = \text{FFRF}$ corresponds to B . Remember that R is the turn right by 90 degrees command.

Parameters

For a more accurate (but also more formal) discussion see reference 6.

It is more suitable to control the turtle using commands with numeric parameters. The length of a line segment is a natural parameter for the command F . For the turn command R we can choose an angle as a parameter. In traditional notation parameters are added in round brackets after a command symbol.

Example 4 : The parametric word $F(1)R(120)F(1)R(120)F(1)$ causes the turtle to draw a triangle.

A symbol can have more than one parameter. For example, in addition to the line segment length it is natural to supply the line with a color defined by an index to a color palette or by RGB components of the color. Usually parameters



L-Systems

are identified by their position in the parameter list.

It is more suitable to assign names (identifiers) to parameters. This leads to a new version of parametric symbol definition.

In this new version the parametric symbol is defined by its name and by the ordered set of its parameters names. All parameters are real numbers (floating point numbers).

The new definition of parametric symbol offers a new syntax for the turtle command notation. This new syntax looks more like the values assignment to structure members in C++ than the classic turtle command syntax.

In the new form the command starts with a symbol name, then parameters names follow with values assigned in the format `parameter_name = value`. We use a comma between two consecutive parameters and a semicolon to terminate the record.

Example 5 : Let the command F (move forward and draw colored line segment) have four parameters: `l` is segment length and `r`, `g`, & `b` are color components. Then the command

```
F l=1, r=1, g=0, b=0;
```

draws a red line segment of unit length.

There are several advantages to using indexed parameters. The first advantage is the possibility of listing command parameters in any order without confusion.

The second advantage is a natural rule for default values of parameters. It allows the dropping of optionally parameters in the symbol `s` records.

The rule for the default values is defined as follows.

Suppose that in the word made of parametric symbols (parametric word) we have an occurrence of the symbol `S`. Suppose that `S` has a parameter with a name `p` and this parameter has no explicitly assigned value at the given occurrence. Let us build a tree for the word. Let `W(S)` be the path from the root of the tree to the node `S`. In the word `W(S)` we look for the last occurrence of `S` with a value `x` explicitly assigned to the parameter `p`. By definition the value `x` is the default value for the parameter `p`. If there is no such occurrence then the default value equals 0 (or any other predefined value, specified in advance).

In other words, the last explicitly assigned value is the default value for the parameter.

We can illustrate this convention in the terms of the turtle state. Let us store in

Making Plants in World Builder

the turtle stack states for each symbol of the input alphabet. While interpreting input word we update each component of the alphabet state each time its value was assigned explicitly. Brackets affect the alphabet state in the same way as other turtle parameters. Thus at any step of interpretation the current alphabet state keeps default values for all parameters. At the initial moment we can assume that the alphabet state is a zero vector.

Another advantage of the new parametric symbol definition is the possibility to treat parametric words in declarative fashion as definitions of some (not necessary graphical) objects. Previously parametric words were mainly considered as turtle programs, i.e. list of executable instructions.



GENERATIVE GRAMMARS AND L-SYSTEMS

Basic concepts

Manual coding of turtle s programs is too laborious. The so called generative grammars offer a method for automatic generation of such programs.

Let us call symbols which are executable commands for the turtle terminal symbols (or just terminals). We need to extend the set of terminals by nonterminal symbols, which have the same nature except they are just ignored by the Turtle. Terminals and nonterminals have the same syntax and can be equally used in input words.

Non-terminals were introduced to define productions.

A production maps nonterminal (predecessor) into a valid word (successor):

$n \rightarrow w$,

where: the leftpart is nonterminal, and the right part is a valid word. The right part can contain terminals and non-terminals. The leftpart must be always a non-terminal symbol.

There could be one, few or no productions defined for each nonterminal. All productions are organized into the ordered list.

Sets of terminals, nonterminals and productions together with a starting word, called initiator, define the generative grammar.

Let us define the rewriting process for the generative grammar.

The rewriting process is a series of single steps. On each step we scan the current word from left to right and for each nonterminal n look for the first production in the list with the predecessor n . After that we replace the nonterminal with the right part of the production. The process starts from the initiator.

The process is infinite if for each step there exists a production applicable to the current word.

Note: Classical L-systems have a parallel mechanism of rewriting. This is an important feature for a variety of models and for program implementation for parallel computers. In World Builder we use only a

sequential mechanism of rewriting.

Example 6 : Let the grammar G have terminals R, F , nonterminals a, b and the productions set:

$$\begin{aligned} a &\rightarrow RFb \\ b &\rightarrow RRRFa \end{aligned}$$

If the initiator is a word of a single symbol a , then the rewriting process will never stop. The turtle interpretation of intermediate steps looks like a staircase. An application of $a \rightarrow RFb$ gives horizontal lines and $b \rightarrow RRRFa$ gives vertical lines.

This example shows that there is no natural mechanism to terminate infinite rewriting. The common solution is to provide with a grammar a set of numbers limiting the depth of recursion for each production. (A more natural solution to this problem is offered by parametric L-systems discussed later)

L-Systems

We define an L-system as a pair consisting of the interpreter (the turtle) and the generative grammar.

The alphabet of terminals must coincide with the set of symbols with nontrivial interpretations. Nonterminals are valid input symbols but have no such interpretation, i.e. are ignored by the turtle.

We will call the generative grammar in this definition an L-grammar.

Note: There are obvious methodological reasons to decompose the notion of L-systems into two weakly connected parts. One can use different interpreters (not only LOGO-style turtle) to obtain different representations of the language generated by L-grammar.

Now we will summarize the main differences from the classical version of L-systems. The interpreter and the generative grammar are almost independent parts of an L-system. Their only common element is the alphabet of terminals. All nonterminals are valid input symbols for the interpreter but are ignored by it. The rewriting process is finite, sequential and deterministic.

Parametric L-Systems

It is natural to use a parametric alphabets in the grammar definition. This generalization is called a parametric grammar. It was introduced by P.Prusinkiewicz (see reference 3).

Here we will discuss the notion of a production in a parametric grammar.

The new element of a production is a predicate. It is a Boolean function which



L-Systems

defines conditions of the production application. The usual notation for a parametric production is:

$$n \text{ IF } \pi \rightarrow w$$

The predicate π here is defined on the space of all symbols parameters. It means that not only parameters of the predecessor can be used as arguments. This is possible because for each position of the symbol in a word we can define default values of parameters, so we can use parameters of any symbol among arguments of the predicate.

Production $n \text{ IF } \pi \rightarrow w$ rewrites the symbol n in the word only if π has value True. Sometimes we skip the predicate for the sake of brevity.

In the right part of the production we can use functions instead of just constant numeric values for parameters in the successor.

Values of these functions are calculated and assigned to the corresponding parameters at the moment of production application. The order of parameter evaluation in the successor is from left to right. Square brackets act in the usual manner. With the occurrence of the left bracket, all parameter values are pushed onto a stack and then are restored after the corresponding right bracket. Within any single symbol the order of parameter evaluation is induced by the order on the parameters names set. (So the order of parameters in the symbols record is not significant.)

These conventions eliminate ambiguity in cases like the following. The symbol

$$F \ r=1, \ g=r+1;$$

in the word

$$F \ r=0, \ b=0; \ F \ r=1, \ g=r+1;$$

will get different parameters values depending on the order of parameter evaluation.

Example 7 : Let the symbol n have three parameters a, b, c ordered as written. Then the production

$$n \text{ IF } a + b > c \ F \ l=a + b - c; \ F \ l=a;$$

can be applied to the symbol

$$n \ a=3, \ b=4, \ c=2;$$

It will be replaced with $F \ l=5; \ F \ l=3;$

Scope resolutions

Different parametric symbols can have parameters of the same name. To solve this common problem of naming conflicts we introduce a scope resolution for parameter names. There are only a few scopes for parameter names. We will describe these scopes later.

As usual names with prefixes serve to identify the scope of names explicitly. If the symbol has the name S and its parameter has name p then the pair $S.p$ identifies this parameter in all contexts. Here S is a prefix for the name p .

Two names with prefixes are different if those names differ as pairs. The usage of names with prefixes is the same as that of ordinary names.

Example 8 : Let a denote an angle of rotation and let it be the name of the symbol sR parameter. Let also the symbol n have parameters a, b and c . Then the production

$$n \text{ IF } a + b > c \text{ R } a = a;$$

does not change the value of $R.a$ because the name a defined in the symbol R hides the same name defined in the symbol n . To access the value of the parameter a from the symbol n inside the scope defined by R one must explicitly specify this scope:

$$n \text{ IF } a + b > c \text{ R } a = n.a;$$

Now n will be rewritten by the rotation symbol with the angle equal to $n.a$.

This example illustrates the scope defined by the production predecessor. The predicate and the successor belong to this scope.

Now the consistency of the production definition with the definitions of the predicate and the parameters as functions on the alphabet state is obvious. This is implied by the fact that at any step of rewriting any component of the alphabet state vector could be accessed. This distinguishes our definition from the classic where the predicate and the successor depends only on the predecessor's parameters.

Example : The successor in the production

$$n \text{ IF } a + b > c \text{ R } a = FL$$

defines the rotation by an angle equal to the current length FL of a line segment.

Initiator SELF

Another enhancement is a special nonterminal: $SELF$. It is the starting symbol and there is only one production for it:



L-Systems

$\text{SELF IF True} \rightarrow a$,

where a is the initiator of the grammar.

We replace the old initiator with the new initiator which is the word SELF . The symbol SELF could not be used in any successor in any production. The language generated by this new version of the grammar is the same as before the introduction of the starting symbol.

The symbol SELF can have parameters. These parameters play the role of global constants and can not be modified because there are no productions with SELF occurring in the successors. It distinguishes these parameters from all other symbols parameters. Obviously all other parameters can be modified.

Further we consider only grammars reduced to the form with the starting symbol SELF . Thus we actually eliminate the initiator from the grammar definition.

Example 9 : Consider the grammar with terminals R, F and nonterminals a, b, SELF . Let w be the only parameter of the terminal R . It denotes the clockwise rotation angle measured in degrees. Let F and b have no parameters. Assume that a and SELF each have one parameter of the same name d (denoting recursion depth).

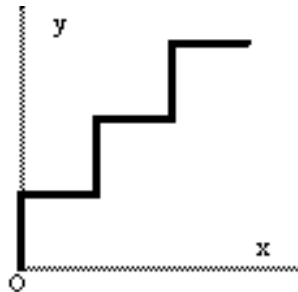
The grammar productions are the following (compare with):

$\text{SELF IF } d > 0 \rightarrow a \ d = \text{SELF } d$
 $a \text{ IF } d > 0 \rightarrow R \ w = -90; F; b$;
 $b \text{ IF True} \rightarrow R \ w = -w; F; a \ d = d - 1$;

If the initial value of the parameter d of the starting symbol equals 3 then the rewriting process is finite and results in the parametric word:

$R \ w = -90; F; R \ w = 90; F; R \ w = -90; F; R \ w = 90; F; R \ w = -90; F; R \ w = 90; F;$

If initially the turtle direction was horizontal then the turtle interpretation of this word is a three segment staircase.



Summary of nonclassic features

To help you in reading additional literature we briefly list non-classic features which were introduced.

All parameters have names and their values are referenced by names. This eliminates the dependence on the order of parameters.

Scopes resolutions for names were introduced. Prefixes are used to avoid ambiguity when parameters of different symbols have the same name.

The starting symbol SELF substitutes the classical initiator. It is also used for storing the values of global constants.

The rewriting process is sequential and deterministic.



MODULAR PROGRAMMING AND L-SYSTEMS

The L-systems models design resembles conventional programming. It is well known that the coding process is much more efficient if it is supported by a library of standard procedures and functions. This is the motivation for introducing into L-systems a mechanism similar to a call of an external procedure from a library or from another module.

Suppose that we have a parametric grammar. We can organize the call of another grammar from inside the right part of a production of the first grammar in the following way. We can use the symbol SELF of the second grammar in the first grammar in the same way as we use other nonterminals. To distinguish it from the SELF of the first grammar we can assign the name for the second grammar and use it with nonterminals of the main grammar. This new symbol we will call the interface symbol of the grammar. It has same parameters as the symbol SELF. We will call the occurrence of the interface symbol a grammar call.

The call mechanism works in the following way:

When the right part with a grammar call overwrites the predecessor, then all parameters of the grammar call (i.e. parameters of the interface symbol) are calculated in a regular way. Then the rewriting process of the external grammar is started. The result of the process is substituted on the place of the grammar call.

This mechanism works correctly with the following assumptions:

Terminals of both (main and external) grammar must be the same.

Nonterminals of the external grammar must all differ from nonterminals of the main grammar.

The last condition protects the side effects of the call. If the external grammar has common nonterminals with the main grammar, then after imbedding the result of the call into the word we can later apply productions from the main grammar to nonterminals which come from the external grammar. This leads to side effects of the call.

It is very difficult to satisfy the last condition in actual applications. The simplest way to avoid side effects in this case is to eliminate all nonterminals from the word derived in the external grammar before it replaces the interface symbol.

This solution is used in L-Objects compiler, so in practice you can use any set of nonterminals.

Example 10 : A simple example of a grammar call. Let the grammar from be the main grammar. We modify the production for the symbol b as follows:

$$b \text{ IF True} \rightarrow h \text{ w=R.w}; a \text{ d=d-1};$$

Here h is an interface symbol of the external grammar H with terminals F, R . The grammar H has no nonterminals except $SELF$ and has only one production. The only parameter of $SELF$ is w . And the only production is:

$$SELF \text{ IF True} \rightarrow F; R \text{ w= -SELF.w}$$

One can check that the new version of the grammar generates the same language.

Note: In the terminology of reference 6 an external grammar is called a subordinate grammar.

The mechanism of a grammar call helps to support modularity in L-systems design. An external grammar is similar to the module which exports only one object (a word in the language defined by grammar). This modularity allows one to decompose L-systems models easier and build libraries of reusable components

L-objects as L-systems with inheritance

Single Inheritance

Following the logic of the evolution of conventional programming languages, we introduce L-systems with inheritance.

Productions will play the role of virtual functions. An explicit non-virtual call of the inherited production could be performed through the explicit type cast to the grammar ancestor. The inherited parameters of symbols behave like data members of classes in C++.

The interested user can find all the technical details in reference 6. Here we will place just a draft of the main idea.

Suppose that you have a grammar and you want to extend it and modify just few things. The usual way is to create a new copy and edit it to meet your new goals. This is a traditional way which leads to duplication of the code. Moreover, the difference between the prototype and the new version is hidden among common details.

Object-oriented design proposes a new approach. We can create a new grammar and place a reference on its prototype - the first grammar. The reference



L-Systems

will mean that we use all features of the prototype except those which are redefined in the new grammar. The new grammar can have a set of required additional features.

The prototype is called the ancestor (or super-grammar, if following C++ name styles) and the derived grammar is called the descendant (or sub-grammar). The relation between the ancestor and the descendant is called inheritance.

L-Systems with inheritance we will call L-Objects

Let us describe in more details how the inheritance is organized and works. First we list all extensions allowed in the descendant:

The descendant can have more nonterminals with respect to the ancestor and it inherits all nonterminals from the ancestor.

In the descendant's inherited nonterminals can have additional parameters, along with all inherited parameters. All additional parameters have default value 0.

A descendant can redefine productions for inherited nonterminals.

The rewriting process is modified in a natural way.

To rewrite the nonterminal we search for the proper production first in the list of productions in the descendant. If nothing found, then we search in the ancestor.

To call the production form of the ancestor explicitly we use a prefix - an interface name of the grammar - before the nonterminal name. In this case the search of the production starts from the list of the ancestor specified by the prefix.

An important feature of the new rewriting process is the following. Suppose we have a nonterminal N in the right part of the production in the ancestor. We can add productions with N as predecessor in the descendant. It means that the behavior of N was redefined. The ancestor knows nothing about it, but when we expand the symbol N occurred in the right part of the ancestor's production, we will use the new production for N . So we add new properties to productions of the ancestor without changing them explicitly.

Note that the starting symbol $SELF$ of the extended grammar is also an extension of the starting symbol from the ancestor. It means that the ancestor's constants are contained among constants of the extended grammar and constitutes the tail of the constants list. This behavior resembles the behavior of an inherited data member in C++.

Grammar inheritance have all the features of inheritance in modern program-

ming languages. Productions applications look like calls of virtual functions. The search of the appropriate production starts among the descendant s productions. Then the search continues on the inherited productions. Inherited productions could be rewritten by descendant s productions. This corresponds to the usual for OOP polymorphism. Here it is supported by rewriting-time search of the actual call address . The address is defined according to the type of the grammar . Due to this, the ancestor s productions can evoke productions of descendants

Non-virtual calls are also available. An explicit type cast suppresses the virtual call mechanism. In this case the production from the particular ancestor is evoked and the call address is known a priori.

The relation of inheritance is transitive. You can use the descendant as an ancestor for another grammar.

We described a single inheritance. A grammar can have only one direct ancestor in the case of single inheritance. But it can be inherited by multiple grammars.

Example 11: Consider the grammar G (see also) with terminals F and R , nonterminals a , b , $SELF$ and productions:

```
SELF IF True → a;
a IF True → b;
b IF True → a;
```

(Here the symbols have no parameters.)

This grammar generates only infinite words which are templates for periodic words. The grammar is a counterpart of the abstract class of C++ in the respect that it does not generate finite words with nontrivial turtle interpretation (abstract classes could not be instantiated in C++ terminology).

Consider the extension of this grammar, where the symbols $SELF$ and a each have one parameter named d . This parameter corresponds to the recursion depth. Productions in this extended grammar are the following:

```
SELF IF True → a d=SELF d;
a IF d > 0 → R w= -90; F; G.a d=d-1;
b IF True → R w= -w; F; G b
```

It is easy to check that this grammar is equivalent to the grammars in and .

Below we list words derived at the first six steps in the extended grammar. We start with $d=3$:



L-Systems

- 0 SELF d=3;
- 1 a d=3;
- 2 R w= -90; F; G.a d=2;
- 3 R w= -90; F; b;
- 4 R w= -90; F; R w=90; F; G b
- 5 R w= -90; F; R w=90; F; a d=2;
- 6 R w= -90; F; R w=90; F; R w= -90; F; G.a d=1;
- 7 ..

Multiple Inheritance

Multiple inheritance of L-objects is a straight forward generalization of the previous section. Instead of just one reference to one ancestor we can use references to several grammars.

We need to order these references, and also be careful of different names for ancestors. The procedure of searching the proper production in the case of multiple inheritance has two variants

It is convenient to compare these variants using an inheritance graph.

As usual an inheritance graph is a graph whose nodes correspond to grammars. Directed edges of the graph represent the inheritance relation. We identify nodes attributed to the same grammar (just like virtual classes in C++ inheritance). Thus the graph is not necessarily a tree but it is a DAG (Directed Acyclic Graph). A directed acyclic graph is a graph of connections whose arrangement and direction do not produce circular references.

Different variants of the search procedure differ in the search orders on the productions set. Consequently nodes of the inheritance graph are also traversed in different order.

First we discuss the simplest generalization (from the implementation point of view). It is linearized multiple inheritance.

Let us draw directed edges of the graph as arrows. So, an arrow denotes the inheritance relation. For example, $A \rightarrow B$ stands for B inherits A.

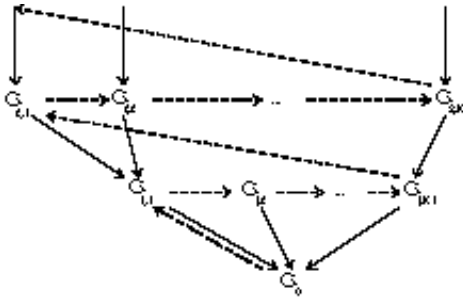
We attribute to each node in the graph to numeric parameters d and n - depth and number respectively. Let the grammar G_0 be the maximal element. This implies that no arrows exit from the node G_0 . This node has depth d=0 by defini-

dition. All nodes connected to G_0 by a single arrow have the depth $d=1$. All nodes with yet undefined depth connected with depth 1 nodes have depth $d=2$. In general the depth of G_i is equal to the length of the shortest path from G_i to G_0 . Define a linear order on each set of grammars with the same depth in some way (usually this is the order of references to the ancestor in the ancestors list). The parameter n denotes the number produced by this order.

The lexicographic order of the set of pairs (d,n) defines the order by which we traverse the graph and collect all productions in one linear list.

We use this list to search for productions as in the previous sections. In particular, the search for an occurrence of the symbol cast to some type starts from the first production of the grammar of this type. From this point the whole tail of the list is scanned and prefixes are ignored.

Graphing an Inheritance Network



In figure 4 the indices of grammars correspond to pairs (d,n) . The search is performed along dashed arrows. Solid arrows denote inheritance. The linear list of productions constructed for this graph looks like:

$$\begin{array}{ccccccc}
 P_0, & P_{1,1}, \dots, & P_{1,k_1}, & \dots, & P_{M,1}, \dots, & P_{M,k_M} \\
 d = 0 & d = 1 & \dots & & d = M
 \end{array}$$

This corresponds to the linearized inheritance graph:

$$G_0 \leftarrow G_{1,1} \leftarrow \dots \leftarrow G_{1,k_1} \leftarrow \dots \leftarrow G_{M,1} \leftarrow \dots \leftarrow G_{M,k_M}$$

There are several advantages to the described mechanism which are useful for implementation. But also there is an obvious disadvantage. It is possible to



L-Systems

apply to a symbol cast to the type h a production unreachable from the node H in the original nonlinearized graph. For example such application is possible for productions from the grammar with the same depth as H but with higher n .

The second approach eliminates such possibility. The search is allowed only against solid arrows in the inheritance graph. It is more close to the regular multiple inheritance, like in C++.

We will call the first approach linear multiple inheritance and the second strict multiple inheritance.

Example 12: In assumptions of the consider another extension of G - the grammar H . New nonterminals x and y are added. The set of productions is the following:

```
SELF IF True → a d=SELF d;
a IF d > 0 → x; G.a d=d-1;
b IF True → y; G.b
```

Symbols x and y have no parameters. This grammar H also could be treated as an abstract class because a word generated by this grammar has no nontrivial turtle interpretation.

Consider another grammar Q with terminals F and R , nonterminals x and y , and productions.

```
x IF True → R w= -90; F;
y IF True → R w= -w; F;
```

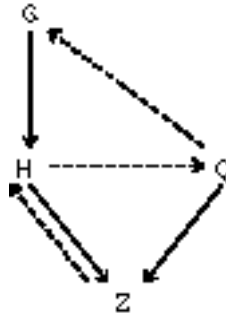
This grammar generates a trivial language because there is no production for the starting symbol.

Grammar H and Q linearly inherited by the grammar Z . The grammar Z adds nothing and contains nothing except references to H and Q .

The grammar Z generates a language equivalent to the language in Examples 10 and 11.

The list of productions for the grammar Z consists of lists from H , Q and G concatenated in the order: H , Q , G . The inheritance graph for Z is depicted in figure 5:

The Inheritance
Network of Four
Grammars



Obviously in the case of strict inheritance, productions of the grammar Q are not reachable from the grammar H against solid arrows. But productions from Q are reachable along dashed arrows. Thus in the linear and strict cases we obtain different languages.

It is possible to use both versions of inheritance simultaneously. We choose linear inheritance as the base mechanism. But we leave an opportunity to specify productions searched in the strict sense. Names with prefixes cast to the type in the strict inheritance manner are denoted as $a:b$.

Such a choice of a base inheritance model is inspired by the realities of software application. We use L-objects for modeling plants and it has turned out that linear inheritance better suits simulating cross breeding of different kinds of plants. This is illustrated by Example 14 from part II.



PROGRAM IMPLEMENTATION OF L-OBJECTS

Due to the small number of basic notions used in defining L-Objects, their program implementation is compact and simple. Actually the language description consists of the list of differences in notation of L-Objects, the description of several extensions and the definition of the interface with a file operating system.

We start with the description of general elements leaving aside specification of a terminals alphabet and its turtle interpretation. As an example of L-objects applications, we consider first 2D turtle graphics and show a simple example of an L-objects program. Then we describe the interpreter implemented in World Builder and discuss its commands.

A set of these examples goes with World Builder in files. An additional L-object file is used to convert regular World Builder L-objects into 2D turtle graphics. See the files in the directory LSYSTEMS\TUTORIAL.

General definitions

The description of each L-object is stored in a separate file with the extension .L. Names of symbols and parameters are the usual identifiers. The length of identifiers is not restricted. The language is case sensitive. Delimiters are tab,

space, carriage return and line feed symbols. Comments are enclosed by pairs of `*` and `*`.

For grammars calls and inheritance all L-objects are referenced by file name and by the name of the grammar. We introduce parameters of the string type to store file names. Actually all string parameters store only a numeric index of the corresponding string constant. It is an index into the string table which exists during a compilation stage. It means that names of string parameters can appear in arithmetic expressions. But a string constant itself cannot appear in mathematical expressions. String constants are enclosed by quotes: `" "`.

For an external grammar call we reserve the name `IMPORT`. A symbol with this name has a string parameter named `file`. The number of other parameters and their names must correspond to the number and names of parameters of the external grammar specified by the parameter `file`. An occurrence of the symbol `IMPORT` evokes rewriting process in the external grammar with parameters specified in the main grammar.

For example if the file `EXAMPLE.L` contains the grammar with parameters `recursionDepth` and `branchingAngle` then the symbol

```
IMPORT file= example1.1 , recursionDepth = 2;
```

evokes the grammar stored in file `example1.1`. The parameter `branchingAngle` during this call has the default value specified in the file `example1.1`.

We assume that any terminals alphabet contains brackets `[` and `]`. Their action was defined in Part I.

Reserved Keywords:

ALWAYS	ANCESTORS	AND
CONSTRUCTOR	END	IF
IS	MEMBER	MEMBERS
OBJECT	OF	OR
PARAMETERS	SEQUENCE	THEN

Specific terminals are added to this set of reserved words each time the new field of application is studied. Later we will describe terminals used in `World Builder`.

Arithmetic expressions are the same as in C++ and use a conventional set of operations denoted in a standard way. Atomic factors in expressions are parameters names, parameters names with prefixes, and floating point numbers. All numbers are converted to the floating point form.



L-Systems

Logical operations are the following: AND, OR and NOT. Comparison operations are $>$, \geq , \neq , $=$, \leq , $<$ (greater, greater or equal, not equal, equal, less or equal, less respectively). The priority of operations is the same as in C++.

We use the keyword THEN instead of an arrow in the notation $n \text{ IF } \pi \rightarrow w$.

The keyword END terminates the production record.

The keyword ALWAYS is used for brevity instead of the sequence IF True THEN.

L-objects syntax

Informal description

An L-objects description consists of two parts. The first part is the declarations and the second is the definitions.

Declarations start from the interface symbol name. Then follows the list of its parameter names with their default values. Then follows a list of nonterminals with their parameters. Default values for user-defined nonterminals are assumed to be 0. Nonterminals are followed by the list of inherited grammars.

The definition section of the grammar contains productions. The production for the starting symbol is indented by syntax and is called CONSTRUCTOR.

All sections are labeled with corresponding keywords and are terminated with a semicolon or END.

Formal definition

A more formal and detailed definition of the language syntax follows:

L-object:

```
OBJECT declarations definitions END
```

declarations:

```
obj_name opt params_decl opt nonterms_decl
      opt ancestors_decl
```

obj_name:

```
identifier
```

params_decl:

```
PARAMETERS obj_par_list;
```

obj_par_list:

```
obj_par_decl
obj_par_list, obj_par_decl
```

obj_par_decl:

```
par_name_decl = constant_expression
```

par_name_decl:

```
identifier
```

nonterms_decl:

```

MEMBERS nonterms_list;
nonterms_list:
  nonterm_decl
  nonterms_list, nonterm_decl
nonterm_decl:
  nonterm_name_decl opt nonterm_params_decl
nonterm_name_decl:
  identifier
nonterm_params_decl:
  (params_list)
params_list:
  par_name_decl
  params_list , par_name_decl
ancestors_decl:
  ANCESTORS ancestors_list;
ancestors_list:
  ancestor_decl
  ancestors_list, ancestor_decl
ancestor_decl:
  object_name constant_string
definitions:
  opt constructor opt productions_list
constructor:
  CONSTRUCTOR parametric_word END
productions_list:
  production_def
  productions_list, production_def
production_def:
  MEMBER nonterm_name IS alternatives_list END
alternatives_list:
  alternative
  alternatives_list, alternative
alternative:
  IF predicate THEN parametric_word END
  ALWAYS parametric_word END
predicate:
  boolean_function
parametric_word:
  parametric_symbol;
  parametric_word parametric_symbol;
  [parametric_word]
parametric_symbol:
  symbol_name opt params
  cycle_operator
params:
  parameter
  params, parameter
parameter:

```



L-Systems

```

    param_name = arithmetic_expression
param_name:
    modifier.nonterm_name
    modifier.nonterm_name
    nonterm_name
    term_name
nonterm_name:
term_name:
    identifier
modifier:
    obj_name
cycle_operator:
    SEQUENCE cycle_var=low_lim, hi_lim OF parametric_word END
cycle_var:
    identifier
low_lim:
hi_lim:
    arithmetic_expression

```

Actually in World Builder the declaration of parameters is more extended:

```
parameter (= default_value, min_value: max_value, text comment)
```

Both `min_value` and `max_value` are parsed by the compiler but are ignored. The `text comment` field is optional, but the comma before it must be present. No semicolons are required after the parameter declaration, but the semicolon is required at the end of the `PARAMETERS` section.

One more extension is the `‡` comment which you can add right after declaration of the object name. This is a special comment which (along with extended definition of parameters) was planned to use for automatic generation of a dialog box with parameters and object description. For now you need to open a file with L-Object and change parameters in a text editor, which is not ideal. This feature will appear in future versions of World Builder.

Cycles

The cycle operator `SEQUENCE OF END` is an extension of an abstract scheme from Part I. It was introduced to eliminate unnecessary recursion.

The cycle is expanded each time when the production with the cycle in its successor rewrites its predecessor occurrence. The cycle body is substituted as many times as there are integer numbers between the low and the high cycle boundaries. The order of substitution is natural: it starts from the low limit of the cycle variable and runs to the high limit. The cycle variable is used inside the cycle body as all other parameters. The cycle defines the scope where this variable is defined. The cycle itself is identified by the name of its cycle variable. Thus, imbedded cycles with the same name for the cycle variable are illegal. However different cycles can be imbedded without any restrictions.

The keyword `SEQUENCE` (instead of `CYCLE`) emphasizes the declarative nature of the L-objects language.

Boolean functions

Boolean functions used for the predicate definition should be always of the form:

$$P_1 \text{ OR } P_2 \text{ OR } \dots \text{ OR } P_N$$

where P_i is a Boolean product:

$$C_1 \text{ AND } C_2 \dots \text{ AND } C_N$$

and C_i is an expression of the form:

$$(\text{arithm_expr} \text{ comparison_sign } \text{arithm_expr})$$

Comparison signs are quite standard comparisons: `<`, `>`, `>=`, `<=`, `=`, `<>`.

The simplest example of Boolean function is:

$$(a > b).$$

Note, that brackets must present in this case too.

More examples:

$$(a > 0) \text{ AND } (b > a)$$

$$(a > 0) \text{ AND } (b > a) \text{ OR } (b > 5)$$

In the last example `AND` (the Boolean product) is computed first, then `OR` (the Boolean addition).

Functions

There are several quite traditional built-in functions in the L-objects language. Random number generator can be called by the function `RANDOM(low high)`. The seed for the random number generator is defined by the special parameter `seed`. This is a parameter of the interface symbol. If the parameter is absent, then the random seed has some unpredictable starting value.

Note: There are no user-defined functions in the language. Nevertheless it is possible to use side effects of inheritance and production application to simulate user-defined functions.

Consider a trivial L-object (this is the minimal syntactically valid L-object):

```
OBJECT None END
```



L-Systems

Let another object inherit the object None. One can introduce a new nonterminal in this ancestor:

```
MEMBERS ... , UserFunction(par1, par2, returnValue);
```

and define the following production for this nonterminal:

```
MEMBER UserFunction IS
  ALWAYS None:UserFunction returnValue=f(par1,par2); END
END
```

where $f(\text{par1}, \text{par2})$ is a user-defined arithmetic expression. When this production rewrites an occurrence of the symbol `UserFunction par1=a, par2=b`; the value $x=f(a,b)$ is calculated and assigned to the parameter `returnValue`. Recursion is suppressed here by strict type cast to the type `None`. The object `None` has no productions. Thus the word rewriting the successor's occurrence is

```
UserFunction par1=a, par2=b, returnValue=x;
```

After that the value $x=f(a,b)$ can be accessed from all productions by the name `UserFunction.returnValue`. This trick allows one to organize procedures and function of any complexity with any number of input and output parameters.

Note that we use here strict type casting. Linear type casting in some cases can evoke productions of ancestors visible in the linearized inheritance graph. The result is not the same as in the case of strict casting. Linear casting does not guarantee that nothing more at this point will be called. The result of `UserFunction` call will be unexpected.

Described trick is extensively used in examples supplied with `World Builder`.

Examples 2D turtle graphics and L-objects

Consider the set of terminals

```
[ , ], DrawLine(length,width), TurnBy(angle).
```

The meaning of these terminals is obvious. We will use this set for drawing on the plane in the following example.

Example 1

The file `EXAMPLE1.L` contains the definition of a simple 2D tree. The turtle interpretation of this definition appears in Figure 5 first to the left

```
OBJECT Tree
  PARAMETERS recursionDepth =5, branchingAngle =60;
  MEMBERS Init, Branch(depth), Fork(side);
  ANCESTORS None none.l ;
```

```

CONSTRUCTOR Init; END

MEMBER Init IS
ALWAYS
  TurnBy angle = -90; /* turns the turtle up */
  Branch;
END
END

MEMBER Branch IS
IF (depth < recursionDepth) THEN
  DrawLine
  length = 5/(depth*depth+1),
  width = 0.2/(depth+1);
  SEQUENCE i=-1,1 OF Fork side = i; END;
END
END

MEMBER Fork IS
ALWAYS
  [ TurnBy angle = side*branchingAngle;
  Branch depth = depth+1; ]
END
END

```

END

The tree defined by this object is too symmetric and artificial. This is usual for simple computer graphics models. We can improve the tree appearance by changing nothing in the original definition but rather by adding new features to the descendants of this object.

Example 2

```

OBJECT RandomForkTree
  PARAMETERS probability=0.85; // branching probability

  ANCESTORS Tree example1.1 ;

  CONSTRUCTOR Init; END

  MEMBER Fork IS
    // The first alternative: //
    IF (RANDOM(0,1) < probability) THEN Tree.Fork; END
    // If the first alternative fails then there is no //branch at this
    point and one must suppress the //search of productions in the
    ancestor: //
    ALWAYS END
  END
END

```



L-Systems

There are less branches in this new version of the tree. Here we define a new version of the production for the symbol Fork. The new production hides the old one. But the old production could be evoked explicitly. It happens with probability 0.85 when the first alternative in the production works. If the check (RANDOM(0,1)<0.85) fails then an empty word rewrites the symbol Fork. This is due to the second alternative.

The turtle interpretation of the new version appears next in the same figure.

Curved branches further improve the tree appearance.

Example 3

```

OBJECT CurvedBranchTree
PARAMETERS
  nSections =2; // the number of straight
                //segments of branches

MEMBERS BranchSection(depthSquare);
ANCESTORS Tree example1.1 ;

CONSTRUCTOR Tree.Init; END

MEMBER BranchSection IS
ALWAYS
  TurnBy angle = RANDOM(-36,36);
  DrawLine
    length = 5/(nSections*depthSquare+2),
    width = 0.2/(depthSquare+1);
  END
END

MEMBER Branch IS
  IF (depth<recursionDepth) THEN
    SEQUENCE i=1,nSections OF
      BranchSection
        depthSquare=depth*depth;
      END;
    SEQUENCE i=-1,1 OF Fork side = i; END;
  END
END
END

```

We introduce a new symbol BranchSection in this version of the tree. This symbol is responsible for drawing straight segments of branches. The turtle interpretation of this version appears third in Figure 6.

And, finally, we combine all improvements in one object. This is a tree with curved branches and irregular forks.

Example 4

```

OBJECT RandomTree
  PARAMETERS seed=5249; /* just a number */
  ANCESTORS RandomForkTree example2.1 , CurvedBranchTree exam-
ple3.1 ;
  CONSTRUCTOR Init; END
END

```

The turtle interpretation of the final version is the last in Figure 6.

Turtle Graphic
Trees - Tree,
Randomforktree,
Curvedbranchtree,
and Randomtree.



The series of these L-Objects illustrates almost all features of the L-Systems language built into World Builder (except grammar call). Also this L-Object provides an example of Incremental Design approach in making trees (Incremental Design in World Builder originates from Object Oriented Programming).

3D terminals and 3D Turtle

Now we will discuss 3D specifics of L-Objects as they are implemented in World Builder .

3D Turtle is quite similar to its 2D counterpart.

The orientation of the Turtle in 3D space is defined by three parameters. So, to change it, we need also specify three parameters. The terminal symbol `AXIS` serves this purpose.

Another difference is that instead of actual drawing 3D Turtle creates 3D primitives which are organized in a tree. This tree is a Plant Object in a World Builder scene. This trees can be rendered and edited (see the Plants section in the World Builder On-Line Tutorials). A Skeleton Line representation of a Plant Object actually draws this tree.

Here is a list of terminals with description of primitives or actions defined by them.

`IMPORT`: calls the external grammar.

Parameters:



L-Systems

`file`: string parameter. Contains a file name in quotes with the external grammar. For example `file= TES.L`. You need to specify a full path if the file is not accessible by redirection.

Other parameters are optional. Only parameters with the same names as in the external grammar are allowed. Parameters without explicitly defined values has values specified in the definition of the external grammar.

`NOISE` : assigns a value to the random seed during the rewriting process.

You can specify the value of the random seed in the `PARAMETERS` section of the l-object definition also. In this case the value of the seed parameter is assigned before the rewriting process starts.

Parameters:

`seed`: a value of the seed.

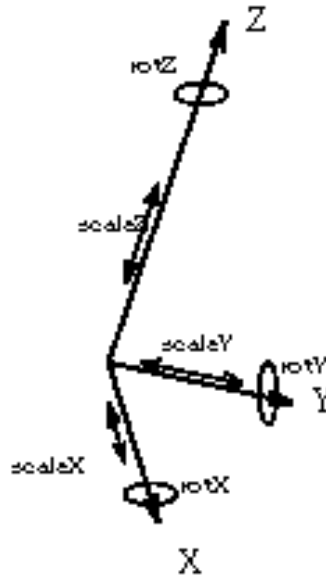
`AXIS`: controls orientation of the Turtle in 3D space.

Parameters:

`rotX`, `rotY`, `rotZ`: define the rotation angle around the corresponding axis. The angle is specified in radians. (There is a constant `PI` in a language.) First rotation is performed around X, then Y and, finally, Z axis. The turtle is headed along the X axis.

`scaleX`, `scaleY`, `scaleZ`: changes the scale along corresponding axis. The size and proportions of primitives depends on the current scale. The initial scale is 1 for each dimension.

3D Turtle Controls



JUMP: moves the turtle position forward (along its X axis) on the specified distance. Does not affect the orientation and scale.

Parameters:

length: specifies the distance which the Turtle moves. The plant contains a primitive corresponding to this symbol. It has length and is not rendered, so it is invisible.

LINE: draws a spline segment of the length length.

Parameters:

length: defines the length of the segment.

tension: controls the spline parameter. For zero tension the spline segment degenerates into a line.

STUMP: defines a segment of the trunk. It is a figure lofted along the spline path with elliptic section. The radii of the ellipse are equal to 1, and the ellipse appears because of possible different scales along the Y and Z axis. Two consequent STUMP primitives are blended automatically.

Parameters:



L-Systems

length: defines the length of the segment.

tension: controls spline parameter. If tension equals 0 then the spline degenerates into a line segment.

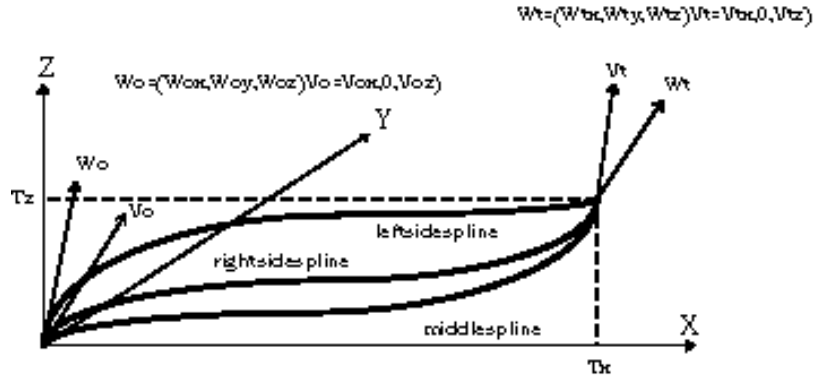
twoSided: if this parameter equals 0 then only those parts of the segment are drawn, which have normal directed to the viewport. Back side of the primitive surface is not drawn. If it has non-zero value, then both sides are rendered. It is switched automatically on if the Profile Shader is attached to the primitive.

HERMITLEAF: defines a primitive consisting of Hermit splines. Used as a model for leaves.

Parameters:

Tz, Tx, Vox, Voz, Vtx, Vtz, Wox, Woy, Woz, Wtx, Wty, Wtz: these parameters are better understandable on the scheme:

HERMITLEAF parameters



The spline patch is mirror symmetric with respect to the $Y=0$ plane. It is defined by three spline curves: the middle and two side splines (which are symmetric). The surface is a smooth blend between left and middle, and between right and middle splines.

Because of the symmetry we can define only the middle and the left side spline. Each spline requires two tangent vectors to be defined. The starting point is always $(0,0,0)$, the terminal point has coordinates $(Tx, 0, Tz)$. The Y component = 0 because of the symmetry.

Left side spline tangent vectors are denoted W_o and W_t (starting point and the terminal point respectively). We have: $W_o = (W_{ox}, W_{oy}, W_{oz})$, and $W_t = (W_{tx}, W_{ty}, W_{tz})$.

For the middle spline the tangent vectors are V_o and V_t . Because of the sym-

metry: $V_0 = (V_{0x}, 0, V_{0z})$, and $V_t = (V_{tx}, 0, V_{tz})$.

This primitive allows to define quite wide range of leaves. The Profile Shader allows to add new details to the original surface.

FILL: creates a volume-fill effects like 3D air brush or needles of pines and spruces.

Parameters:

fillType: selects the type of effect:

- 0 switches all effects of f
- 1 air brush
- 2 spruce needles
- 3 pine needles.

air brush: voxels are randomly placed in the elliptic volume defined by scales along the Turtle axis.

spruce needles: lines are drawn. These lines radiates from the spline segment, the angle between the line and the spline segment is defined by the angle parameter. The length of the segment is defined by the length parameter. The tension affects the spline segment as in the LINE primitive. The length of needles is 1 in the scale along the X axis.

pine needles: lines are radiating from one point. The angle of the cone raster is defined by the angle parameter. The length parameter controls the length of needles.

density: defines the number of volume fill elements (voxels or lines).

angle: works only with pine and spruce needles see above.

seed: assigns the value to the random seed of random numbers generator.

length: works only with pine and spruce needles see above.

tension: works only with spruce needles see above.

TAPE: a primitive similar to the segment of the film tape. The starting edge of the tape is the line cut from $(0, 1, 0)$ to $(0, 1, 0)$ in the local coordinates of the turtle. The ending edge is the same line segment for the next position of the turtle. Side edges are splines.



L-Systems

Parameters:

length: the length of the primitive.

tension: controls splines.

twoSided: if non-zero then forced to be two-sided. Default is 1.

MATERIAL: assigns the material to all primitives which follow the MATERIAL instruction. The next MATERIAL overrides the previous.

Parameters:

file: the name of the resource file (see binary *.shd files in MATERIALS directory). You can create material resource file from the L-Objects editor.

resource: the name of the resource in the resource file.

Both names must be enclosed in quotes.

BACK_MATERIAL: assigns material for back sides of HERMITLEAF and TAPE primitives. Works as the MATERIAL instruction.

Parameters:

file, resource: are same as in the MATERIAL case.

Built-in functions and Constants

Constants

PI = 3.14...,

LONG_MAX = maximal LONG in the system.

Functions

RANDOM(min_value, max_value): returns random numbers uniformly distributed in the range (min_value, max_value).

Unary functions are quite standard:

ABS: returns the absolute value of the argument

SIGN: returns -1 if the argument is negative, +1 if it is positive and 0 if it is zero.

SQRT: returns square root of the argument if it is positive and 0 otherwise.

SIN: returns sine of the argument

L-Objects Programming Environment

`COS`: returns cosine of the argument

`ASIN`: returns arcsine of the argument

`ACOS`: returns arccosinus of the argument

`TAN`: returns tangent of the argument

`ATAN`: returns arctangent of the argument

`COTAN`: returns cotangent of the argument

`EXP`: returns exponent of the argument

`LN`: returns a natural logarithm of the argument

If the argument does not belong to the definition domain of the function, then 0 value is returned, as in the case of negative argument for `SQRT`.

Warning: no run time error reports are generated!



L-OBJECTS PROGRAMMING ENVIRONMENT

Debugging L-objects

Debugging features of L-Objects are controlled by the AWB.INI file.

The first feature available in the current version is the possibility to create a dump file with the parametric word derived during the rewriting process. It has the same name as the main file with the current L-Object and the extension .rep.

Change the GenerateLObjectReport parameter to YES in AWB.INI to switch on the reporting function.

Warning: Report files are rather big! Do not forget to switch the option off later, otherwise your disk will fill up.

Also you can control the maximal depth of recursion. This can help to find the erroneous recursive calls in L-objects productions.

Assign a small value, like 4 or 5 to the MaxRecursionDepth parameter in AWB.INI. If the specified depth will be reached, the message will appear with the name of production (the MEMBER name) where it happened.

Because of exponential growth of data volume in recursive algorithms you need to be careful while using recursive definitions. Otherwise sometimes you can occasionally create monstrously big objects.

You are encouraged to gain experience modifying examples which come on the World Builder CD ROM.

L-Objects compiler reports

The L-Objects compiler reports on the following errors:

- (expected: Left bracket expected.
-) expected: No pairing right bracket in the arithmetic expression.
- , expected: A comma expected.
- : expected: A colon expected.
- } expected: The { } comment after the object name declaration is not

closed properly.

= expected: Must be assignment here.

file= ... expected: The IMPORT symbol requires file parameter.

OF expected: The key word OF is missing.

" expected: String expression enclosed in quotations must appear here.

can not evaluate const: You can not use arithmetic expressions in the PARAMETERS section

circular inheritance: The inheritance graph can not have cycles. For example, A can not inherit B if B already inherits A.

comparison expected: Comparison missed after IF key word.

CONSTRUCTOR expected: Must never occur, since a constructor is optional.

duplicated name: You used the same name for another parameter inside one symbol.

END expected: The object or member definition is not closed properly.

ENDIF expected: You did not close IF section properly.

identifier expected: Here must be an identifier.

IF or ALWAYS expected: Predicate is missing.

invalid expression: Invalid arithmetical expression.

invalid field name: The name of parameter with a prefix is invalid.

invalid function name: No such built-in function.

invalid string index: You are addressing to the string which is not present in the string table. It can happen only if you calculate the value of the string parameter and can never occur if you use string constants

IS expected: The key word IS is missing.

is not an l-value here: The leftpart of assignment is wrong.

MEMBER expected: Members definitions must follow each other and must be opened with the MEMBER key word.