ANIMALSCRIPT– The Reference

Dr. Guido Rößling
TU Darmstadt
Computer Science Department - RBG
roessling@acm.org[*]

September 27, 2008

Contents

[*]Check out the official ANIMAL WWW page at `http://algoanim.info/Animal2` for the most recent ANIMAL release including ANIMALSCRIPT, documentation including this reference, example animations and other stuff.

1 Notation

In the following sections, the structure for the supported types and effects is described. We use a very informal *BNF*[1] where components in brackets `[]` are *optional* and components in curly braces {} may be repeated as often as one wants, but must be present at least once. If a curly brace is supposed to be placed in the file, it will be quoted using single quotes (`'{'` and `'}'`, respectively).

Notation
[]
{}

Note that the combination, e.g., `[{component}]`, means that `component` is optional, but may appear as often as wanted: it is *optional* due to being set in brackets `[]`, but if given, may be given as often as wanted ({}).
Listing 1 provides an example definition for the notation used throughout this document.

Listing 1: Example definition

```
<illustrativeExample>:
  %example "<id>" \n
  { author "<text>" \n }
  [ comment "<text>" \n]
```

According to the definition in Listing 1, an entity called `illustrativeExample` is defined as follows:

- Each `illustrativeExample` *must* start with the characters `%example`, followed by an *ID* (defined elsewhere) enclosed in double quotes. This line *must* end with a linefeed character (do not worry about the notation "\n" here; pressing the *return* or *enter* key works exactly as intended).
- The header line must be followed by at least one author specification (due to the curly braces, which mean "as many as wanted, but at least one element"). Each author specification consists of the keyword `author`, followed by a *text* enclosed in double quotes, and followed by a linefeed.
- Finally, an optional comment may be given. This starts with the keyword `comment`, followed by a *text* in double quotes and a linefeed.

If we assume that *ID* is defined as an integer value, and text covers the usual text components (letters, digits, comma, space etc.), the following would be a legal instance of *illustrativeExample*:

Listing 2: Notation example

```
%example "17"
author "Dr. Guido Roessling <roessling@acm.org>"
author "Several volunteer testers"
```

As you can see, the optional comment has been left out.
In the following notation, keywords are **bold**; variables are placed in *italics*.
Mutually exclusive alternatives are separated by |. The following example illustrates *optional* arguments and *alternatives*:|

keywords
variables

Listing 3: timeOffset

```
<timeOffset>:
  after <nat> [ticks | ms]
```

This definition means that the last argument is optional (due to the brackets encapsulating them), but only one of them may be given. Thus, correct inputs are

- **after 500** (note: in this case, *ticks* is assumed)
- **after 50 ticks**
- **after 500 ms**

[1] *Backus-Naur-Form*, a formal notation for describing language grammars

2 Basic Definitions

In the following subsections, we examine basic definitions that will be used throughout the document. These include *numerical values* and *expressions*, the definition of *nodes, coordinates* and *locations*, and the list of available *colors, display options* and *fonts* in ANIMALSCRIPT.

2.1 Numbers and numerical expressions

ANIMALSCRIPT uses on *natural* or *integer* numbers in its BNF.
Natural numbers (symbol <nat> are positive whole numbers including **0**. nat
Integer numbers are whole numbers, i.e., without a decimal point. They can be specified in the following ways: int

- by giving their literal value, e.g., as **42**,
- as a *point* of a given object's bounding box, using the anchor points **x, y, width** and **height** *since 2.0*
- or as an *expression* using *double values* and the four operators **+, -,** $*$**, /**. *since 2.0*

Listing 4: Definition of natural and integer numbers

```
<nat>:
  0 | 1 | ... |

<int>:
  <nat> | -<nat> | ( <double> <operator> <double>)
  | "objectID" <objectPosition>

<operator>:
  + | - | * | /

<objectPosition>:
  x | y | width | height
```

2.2 Definition of nodes, coordinates and locations

ANIMALSCRIPT provides a very versatile way to define nodes or coordinates:

- Using absolute coordinates,
- Using an offset from a point of the *bounding box* of a given object,
- Using an offset from a numbered point of a fitting underlying structures, such as a polygon,
- Using an offset from a defined location,
- Using an offset from the baseline of a text.

These different approaches to define a coordinate can be used wherever the definition of a coordinate or node is expected. For example, a polygon can be defined by combining an arbitrary combination of the above approaches.

Listing 5: Coordinate definition

```
<nodeDefinition>:
  (<int>, <int>) | <offset> | move (<int>, <int>)

<offset>:
   offset (<int>, <int>) from "referenceID" node <nat+>
 | offset (<int>, <int>) from "referenceID" <direction>
 | offset (<int>, <int>) from "locationID"
 | offset (<int>, <int>) from "referenceID"
     baseline [start | end]

<direction>:
  NW | N | NE | W | C | E | SW | S | SE
   | Northwest | North | Northeast | West
   | Middle | Center | East | Southwest
   | South | Southeast
```

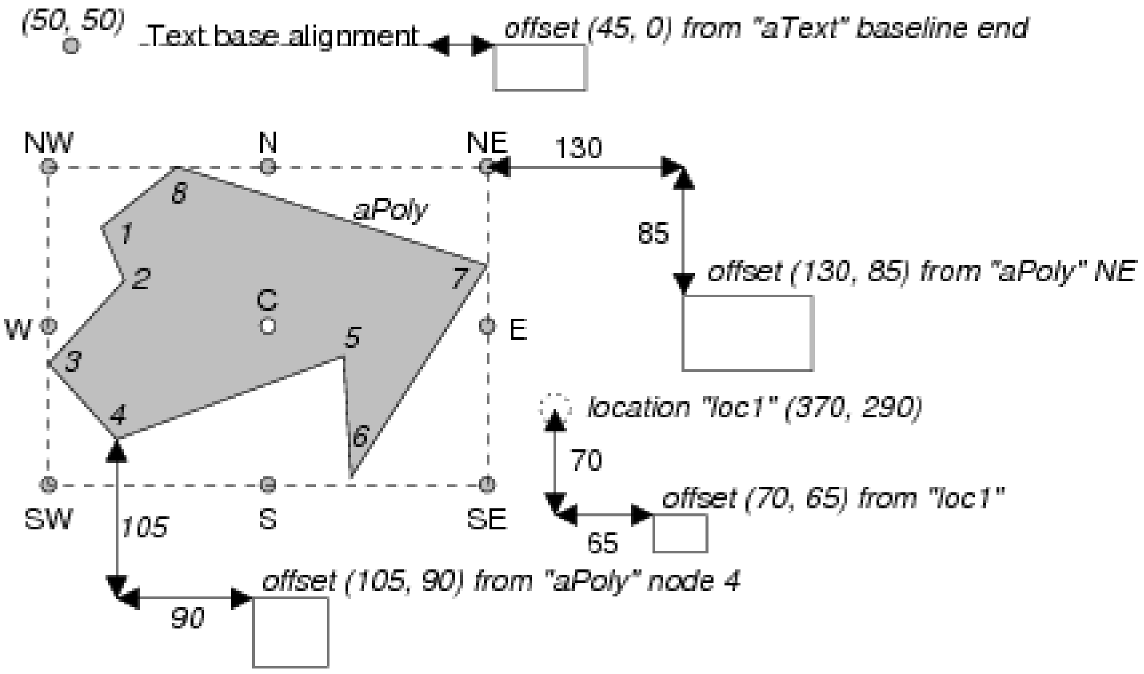
(50, 50)
Text base alignment
offset (45, 0) from "aText" baseline end
NW
N
NE
130
8
aPoly
1
85
2
7
offset (130, 85) from "aPoly" NE
C
W
E
5
3
location "loc1" (370, 290)
4
6
70
SW
S
SE
offset (70, 65) from "loc1"
105
65
offset (105, 90) from "aPoly" node 4
90

Figure 1: ANIMALSCRIPT node definitions: absolute, offset from object or node

Figure 1 illustrates the different variants for specifying coordinates in ANIMALSCRIPT. The circle at the top left is specified using absolute coordinates placed in a pair of parentheses. The shaded polygon is referred to as "aPoly", the user-assigned name of the object. The dashed box around the polygon represents the polygon's *bounding box*, the smallest rectangle that covers all nodes of the polygon. The rectangle to the right of the shaded polygon is specified at an offset of 130 pixels to the right and 85 pixels below the *northeast (NE)* corner of the bounding box. Note that the specification uses the polygon's name and that the *bounding box* is used implicitly, i.e. there is no explicit mentioning of it. The square below the shaded polygon was specified at an offset from one of the polygon's *nodes*.

bounding box

Finally, we have defined a *location* object called *loc1* at coordinates (370, 290), shown as a dashed circle. *since 2.0*

We can use this to specify a new object relative to the location. As the location is a single point, we do not have to specify a *bounding box coordinate* or *node number*.

2.3 Definition of locations

since 2.0

As of release 2.0, ANIMALSCRIPT allows the user to specify customized locations using the following notation:

Listing 6: Definition of locations

```
<location>:
  <locationKeyword> "locationID" [at] <nodeDefinition>
  | moveLocation "locationID" [to] <nodeDefinition>
```

Once a *location* is set, it can be used as the starting point for a relative object placement.

2.4 Color definitions

ANIMALSCRIPT provides 35 predefined color names and also allows the user to specify an arbitrary RBG color by giving the color values, as shown in Listing 7.

Listing 7: Specifying colors in ANIMALSCRIPT

```
<color>:
   black | blue | blue2 | blue3 | blue4 | brown2
   | brown3 | brown4 | cyan | cyan2 | cyan3 | cyan4
   | dark Gray | gold | green | green2 | green3
   | green4 | light Gray| light_blue | magenta
   | magenta2 | magenta3 | magenta4 | orange | pink
   | pink2 | pink3 | pink4 | red | red2 | red3
   | red4 | white | yellow | (<nat>, <nat>, <nat>)
```

All entries except for the last contain the *predefined* color names in ANIMALSCRIPT. The last entry allows the user to specify arbitrary RGB colors by assigning explicit values for *r (=red), g (=green)* and *b (=blue)*. Each color value must be an *integer* in the interval `[0, 255]`.
Note that there are two color consisting of *two* words: `light Gray` and `dark Gray`. This is not a mistake; we have adopted the notation of the colors from the Unix-based *xfig* graphics program.
For a mapping of the color names to *RGB* values, see section 6 on page 44.

2.5 Display options for graphical elements

ANIMALSCRIPT allows the user to specify that a given graphical object may be hidden (thus, invisible), or may only appear after a certain delay. The delay is measured either in *ticks*, where one tick equals one animation frame in the display, or in *ms*.

ticks
ms

Listing 8: Definition of display options

```
<displayOptions>:
  hidden | <timeOffset>

<timeOffset>:
  after <nat> [ticks | ms]

<timing>:
  [<timeOffset>] [within <nat> [ticks | ms]]
```

The `displayOptions` tag is usually employed for defining graphic objects, which may be specified as being invisible (`hidden`) or appear only after a certain delay. The `timing` tag is used for animation effects ("animators") which can use both an *offset* and a *duration*.

2.6 Font definitions

To keep maximum platform independence, ANIMALSCRIPT supports only the three basic font families *Serif, SansSerif* and *Monospaced*. If no font is specified for a given text element, ANIMAL uses

1. the last font name used for a text component;

2. `SansSerif` if no font name was given before.

Listing 9: Definition of a font entry

```
<font>:
  [font <fontNames>] [size fontSize] [bold] [italic]
```

Listing 10: Availabel font names in ANIMALSCRIPT

```
<fontName>:
  Serif | SansSerif | Monospaced
```

2.7 Specifying a list of graphic object IDs

ANIMALSCRIPT makes it very easy to specify a set of objects - one simply lists their IDs with a space between each element. Note that *each* object ID must be quoted *individually*. Other notations, for example"`a,b,c`", are invalid, as ANIMALSCRIPT would then look for an object with ID *"a,b,c"*.

Listing 11: Specifying a list of graphical object IDs

```
<oids>:
  { "targetOID" }
```

2.8 Specifying quoted text

Quoted text (“`xxx`”) must be enclosed in double quotes. All standard word characters (i.e., letters, digits, whitespace, punctuation etc.) is legal. For the sake of clarity, the BNF will usually place a *description* of the expected content inside the double quotes. This does *not* mean that you have to use this text verbatim!

2.9 Concluding comments

ANIMALSCRIPT is line-oriented: each line may contain *exactly* on command, comment, or completely consist of whitespace elements. Placing more than one command in a single line is syntactically incorrect. Spreading a command over several lines is also invalid.
Due to layout restrictions, some commands in this documentation are spread over multiple (document) lines. However, they have to be placed in one line per command in actual input files.
Mandatory line breaks – of which there are but a handful in ANIMALSCRIPT– are clearly indicated by **n**. **n**

3 File Header Format

ANIMALSCRIPT files always have the *header* shown in Listing 12, starting with the `animalScriptFile` tag.

Listing 12: ANIMALSCRIPT header format

```
<animalScriptFile>:
  <fileHeader> { <command> \n}

<fileHeader>:
  %Animal <double> [<nat>*<nat>] [\n <titleInfo>]
  [\n <authorInfo>]

<titleInfo>:
  title "title as a string"

<authorInfo>:
  author "author name including EMail address"
```

Note:

- The percentage sign **%** *must* be given!
- The *double* following the keyword **Animal** is the *version number* of ANIMALSCRIPT used to determine additional parsing options. You should *always* use the version number of the current ANIMAL release you use, e.g. **1.4** or **2.0**.
- The two optional *natural numbers* specify the display size of the animation (*width* ∗ *height*). Since ANIMAL 2.2, they will cause the animation window to have the proper width and height to show the animation.
- The texts for *author* and *title* will be shown in the *About this Animation*... window, if they are present.

A valid file header may thus look as shown in Listing 13.

Listing 13: Example ANIMALSCRIPT file header

```
%Animal 2.0
title "Demo of AnimalScript features"
author "Guido Roessling <roessling@acm.org>"
```

The file header is followed by a set of *single line commands*. Each command is defined as shown in Listing 14.

Listing 14: Definition of commands

```
<command>:
  <objectPrimitives> | <operations>
  | '{' | '}' | <languageSupport> | <extensionCommand>

<objectPrimitives>:
  <arcTypes> | <array> | <arrayMarker> | <codeTypes>
  | <listelement> | <point> | <polygonTypes>
  | <polyline> | <text> | <grid> | <graph>
```

Listing 15 provides an overview of the operations on graphical objects and other actions that ANIMALSCRIPT supports.

Listing 15: Definition of operations

```
<operations>:
  <arrayOp> | <clone> | <colorChangeTypes> | <delay>
   | <echo> | <label> | <link> | <location>
   | <merge> | <move> | <rotate> | <showTypes>
   | <swap> | <setText> | <setFont> | <variableSupport>
   | <assertion> | <gridOperations> | <graphOperations>
```

composite steps

Curly braces {} are used start or end a *composite step*. All commands between the braces will take place in the *same* step. Note that the braces **must** appear on a *single* line each, which must otherwise be *empty*.

#

To make scripting code easier to read, all lines starting with a *hash mark* **#** are considered *comments* and ignored during ANIMALSCRIPT parsing.

3.1 Language support

ANIMALSCRIPT is prepared for Internationalization using the built-in *language support*, which is used to support multiple languages inside ANIMALSCRIPT animations. If used wisely, it allows the user to generate animation to generate a single animation that incorporates separate language versions.
In order to harness the full power of this functionality, the user has to use *relative object placement* as illustrated in figure 1, as text objects tend to have different sizes in different languages. For example, German sentences are usually longer than their English counterpart. The built-in internationalization support of ANIMALSCRIPT provides the commands shown in Listing 16.

Listing 16: Internationalization support

```
<languageSupport>:
   supports { "languageKey" }
   [\n <resourceKey> "fileNameWithoutExtension"]

<resourceKey>:
   resource | bundle | resourceBundle

<intText>:
   "text" | key: "textResourceKey" | ( { key: "text" } )
```

The `supports` statement is followed by a list of *language keys*, as specified in the ISO standard. For further details, check the *Java API Documentation* for class java.util.Locale. For example, use en for *English*, fr for *French* (Française) and de for *German* (Deutsch). This variant is used when the translations are *embedded* into the animation.
For internationalized text, the user can choose between using the actual text (without translations), a key to the resource file, or a key followed by the translation into the target language. In the latter case, **key** should be a valid language key supported by the animation.
Once the ANIMALSCRIPT parser reaches a `supports` command, it will pop up a dialog that asks the user to choose the language for this animation. All internationalized elements will then be displayed in the language chosen by the user. Note that without a `supports` keyword, internationalization will *not work* in ANIMALSCRIPT.

resource files

Optionally, this statement may be followed by one of the `resourceKey` statements to define an *external resource file* that contains the translations of the text elements. *This is the preferred approach*. When the ANIMALSCRIPT parser reads the `support` statement, it queries the user for the language to use. If an external resource is specified, it will then try to open a file of the following name:

`fileNameWithoutExtension.languageKey`

Let us regard the brief example in Listing 17.

Listing 17: Internationalization example

```
1 supports "en" "de"
2 resourceBundle "demoBundle"
```

After the **supports** command is parsed (line 1), the user will be prompted to select one of the languages *en* or *de* (that is, English or German). Let us assume the user chooses *en*. ANIMALSCRIPT will then try to open the file `demoBundle.en` that contains the English translations of the animation elements.

4 Defining graphic primitives

4.1 Available types

The version of ANIMAL described in this document supports the following primitives:

- *point*, described in Section 4.2 on the next page
- *line* and *polyline*, described in Section 4.3 on the following page
- *polygon* with several ubtypes, described in Section 4.4 on page 14
- *text*, described in Section 4.5 on page 17
- *arcs* with several subtypes, described in Section 4.6 on page 19
- *list element*, described in Section 4.8 on page 24
- *arrays* and *array markers*, described in Section 4.7 on page 22
- *source code support*, described in Section 4.9 on page 25
- *grids*, usually in the form of a matrix, described in Section 4.10 on page 27
- *graphs*, described in Section 4.11 on page 28

Most graphical primitives share a set of common characteristics, as follows:

- The primitive's *ID* is always specified immediately after the first keyword. It contains the (user-chosen and thus arbitrary) name for the component, placed in double quotes. All operations later on have to use this ID if they want to modify the primitive.
- Each *nodeDefinition* entry represents the coordinates of a single point, or a relative placement according to the offset given - see Section 2 on page 4. The number of legal nodes depend on the specific primitive.
- At least one *color* is defined for each object. The "main" color–e.g., the color of a text primitive, and the outline of a polygon–is defined following the keyword `color`. Other colors are introduced by appropriate keywords.
- *depthValue* defines the depth of the object. *0* stands for *foreground*, 1 for the first layer *behind* the foreground etc. The higher the value, the farther to the "back" the object is places.

  The default value substituted is the value for the *background* $2^{31} - 1 = 2147483647$, equal to Java's `Integer.MAX_VALUE`.
- *displayOptions* determine if the object should be shown and if so, when. See Section 2 on page 4 for details.

Listing 18: Definition of the ANIMALSCRIPT primitives

```
<objectPrimitives >:
  <arcTypes> | <array> | <arrayMarker> | <codeTypes>
  | <listelement> | <point> | <polygonTypes>
  | <polyline> | <text> | <grid> | <graph>
```

4.2 Point primitive

The structure for a point is shown in Listing 19.

Listing 19: Notation for a *point* primitive

```
<point>:
  point "pointID" <nodeDefinition> [color <color>]
     [depth <nat>] [<displayOptions>]
```

Contrary to the formatting shown here, the *point* definition **must** appear in a single line which *starts* with the keyword **point** (disregarding whitespace) and does not contain any other statements.
An example command for generating a point is shown in Listing 20.

Listing 20: Example command for a *point* primitive

```
point "p" (20, 20) color blue3 depth 5
```

The output of this is shown in Figure 2.

Figure 2: Output for the example point primitive

4.3 Polyline type primitives

Listing 21 describes the notation for a *line* or *polyline* object. Listing 22 defines how the optional arrows for line objects can be defined.

Listing 21: Notation for a *line* or *polyline* object

```
<polyline>:
  <lineTag> "lineID" <nodeDefinition> { <nodeDefinition> }
    [color <color>] [depth <nat>]
    [<arrowOptions>] [<displayOptions>]

<lineTag>:
  polyline | line
```

Listing 22: Notation for the optional arrows used for several object types

```
<arrowOptions>:
  [fwArrow] [bwArrow]
```

Contrary to the formatting shown here, the definition **must** appear in a single line which *starts* with the keyword **line** or **polyline** (disregarding whitespace) and does not contain any other statements.
Further explanations:

- Each *line / polyline* must have at least *two* nodes (otherwise, it would simply be a point). On the other hand, the number of nodes is not limited by ANIMALSCRIPT, although there may be practical bounds imposed by handling speed, memory consumption etc. on your machine.
- If **fwArrow** is given, the object will have an arrowhead at its end, thus pointing forward.
- If **bwArrow** is given, the object will have an arrowhead at its start, thus pointing backward.

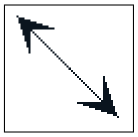

Figure 3: Example *line* primitive with both *fwArrow* and *bwArrow*

Figure 3 shows the output generated for the following code that generates a line with arrows at both ends, as it is defined in Listing 23

Listing 23: Example command for a *line* primitive

```
line "1" (25, 25) (75, 75) color (10, 20, 30) depth 1 fwArrow bwArrow
```

Figure 4 shows a polyline which specifies the first coordinate relative to the previous line edge, followed by two relative offsets to the previous node, respectively. The code for this is shown in Listing 24.

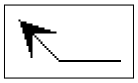

Figure 4: Example *polyline* with three relative nodes

Listing 24: Example command for a polyline with three relative nodes

```
polyline "pl" offset (10,10) from "1" SE move (20,20) move (30,0) bwArrow
```

4.4 Polygon primitives

The family of *polygon* primitives includes the following subprimitives:

- *triangle*,
- *square*,
- *rectangle*, either defined by the upper left point and width/height, or by its upper left and lower right corner, and
- arbitrary *polygon*.

As the definition of the subtypes are somewhat different, we describe them in separate subsections.
Apart from the common attributes described in Section 4.1 on page 12, each polygon object can possess *fill options*, specified as shown in Listing 25.

Listing 25: Notation for *fill* options

```
<fillOptions >:
  filled [fillColor <color >]
```

If *filled* is given, the object will be filled. *fillColor* may only be used if the keyword *filled* was also given; it defines the color the component will be filled with. See section 2.4 on page 6 for a list.
Listing 26 summarizes the elements of the *polygon* family.

Listing 26: Definition of the *polygon* family members

```
<polygonTypes>:
  <square> | <rect> | <triangle> | <polygon>
```

4.4.1 Square primitive

Listing 27 defines the notation for a *square* primitive.

Listing 27: Definition for the *square* primitive

```
<square>:
  square "ID" <nodeDefinition> <nat+>
      [color <color>] [depth <nat>]
      [<fillOptions>] [<displayOptions>]
```

A square is defined by its upper left corner (given using a *nodeDefinition*, as defined in Section 2 on page 4), and its *size* (identical for width and height) in pixels.
All other features are described in Section 4.4 on the preceding page.
Figure 5 shows the square generated from the code in Listing 28.

Listing 28: Example command for a *square* primitive

```
  square "s" (50, 50) 45 color grey depth 2 filled fillColor blue
```

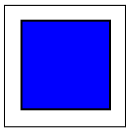

Figure 5: Example square filled with blue

4.4.2 Rectangle primitives

Listing 29 defines the notation for the *rectangle* primitives supported by AnimalScript.

Listing 29: Definition for the *rectangle* primitive types

```
<rect>:
  <absoluteRectangle> | <relativeRectangle>

<absoluteRectangle>:
  rectangle "ID" <nodeDefinition> <nodeDefinition>
      [color <color>] [depth <nat>]
      [<fillOptions>] [<displayOptions>]

<relativeRectangle>:
  <relRectName> "ID" <nodeDefinition> <nodeDefinition>
      [color <color>] [depth <nat>]
      [<fillOptions>] [<displayOptions>]

<relRectName>:
  relrect | relrectangle
```

The first *nodeDefinition* defines the upper left corner of the rectangle, as described in Section 2 on page 4.
For *rectangle* and its abbreviation *rect*, the second *nodeDefinition* contains the *coordinates* of the lower right rectangle corner.
For *relative rectangles*, the second node definition contains the *offset* from the upper left corner. Thus, the specified values represent the *width* and *height* of the rectangle.
Therefore, the following two definitions are equivalent–it is up to the user to decide which approach is easier to use in the given circumstances.

Listing 30: Equivalent definition of rectangles

```
# corners at coordinates (10, 10), (20,20)
rectangle "absoluteRect" (10, 10) (20, 20)
# corners at coordinates (10, 10), (10+10,10+10)
relrect   "relativeRect" (10, 10) (10, 10)
```

The output of these commands is shown in Figure 6.

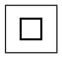

Figure 6: Example rectangle

4.4.3 Triangle primitive

Listing 31 defines the notation for *triangle* primitives.

Listing 31: Definition for *triangle* primitives

```

<triangle>:
  triangle "ID" <nodeDefinition> <nodeDefinition>
     <nodeDefinition> [color <color>] [depth depthVal]
```

The *nodeDefinition* entries define the three coordinates of the triangle, as described in Section 2 on page 4.
All other entries have been described in Section 4.4 on page 14.
The triangle generated from the example code in Listing 32 is shown in Figure 7.

Listing 32: Example command for a *triangle* primitive

```
  triangle "t" (100, 100) (50, 50) (150, 50) filled
```

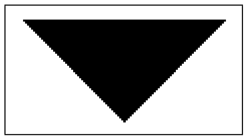

Figure 7: Example triangle output

4.4.4 Polygon primitive

Listing 33 describes the notation for a *polygon* primitive.

Listing 33: Definition for *polygon* primitives

```
<polygon>:
  polygon "ID" <nodeDefinition> <nodeDefinition>
     { <nodeDefinition> } [color <color>]
     [depth <nat>] [<fillOptions>]
     [<displayOptions>]
```

Each of the (arbitrarily many) *nodeDefinition* elements defines one edge of the polygon. All other entries have already been described in Section 4.4 on page 14.
Listing 34 defines an example polygon. The output of the code is shown in Figure 8.

Listing 34: Example command for a *polygon* primitive

```
  polygon "pp" (20,20) (75,80) (75,40) (90,20) filled fillColor (0,128,127)
```

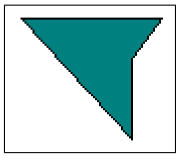

Figure 8: Example polygon

4.5 Text primitives

Components of this type always begin a new line with the keyword **text**, as shown in Listing 35.

Listing 35: Definition for *text* components

```

<text>:
  text "ID" <intText> [at] <nodeDefinition>
    [centered | right] [color <color>] [depth <nat>]
```

The *nodeDefinition* defines the *lower left corner* of the text, unless one of the keywords **centered** or **right** is given.

- The optional keyword *centered* places the *middle* of the text's baseline at the point calculated above. It has *no* effect on its vertical position. centered

  Similarly to `centered`, *right* places the *lower right corner* of the text's baseline at the point calculated above. right

- The text's font is specified inside the fontDef tag, as specified in Section 2.6 on page 7.

- The optional keyword **boxed** places a filled box below the text. This is especially useful if the text is supposed to be a header entry.

4.5.1 Internationalization for Text Primitives

The *internationalized text* is defined as follows:

Listing 36: Definition for internationalized text components

```
<intText>:
  "text" | key: "textResourceKey" | ( { key: "text" } )
```

As can be seen, there are three notations for specifying a text:

- as a fixed text enclosed in double quotes,
- as the keyword **key** followed by a colon (**:**) and the *name of the resource key*, or
- as a pair of parentheses containing a sequence of *languageKey* followed by a color (**:**) and the text component as a fixed text enclosed in double quotes

The first notation should be self-explanatory. The other two notations use internationalized texts. The second variant can only be used if a *resource file* is specified, as defined in Section 3.1 on page 10. If this is the case, ANIMALSCRIPT looks for a resource with the name as given inside the double quotes. *Make sure there are no typing mistakes in the resource name, especially concerning upper or lower case.*
The last variant uses an embedded definition of translations. The pair of parentheses contains an arbitrary number of *language keys* as specified by the `supports` command, followed by a colon (**:**) and the translation inside double quotes.
As an example, let us regard the specification provided in Listing 37.

Listing 37: Example of using internationalized text primitives

```
1  %Animal 2.0
2   supports "en" "es"
3   resource "demoResource"
4   # use direct encoded text, no internationalization
5   text "straight" "Straight English Text" (10, 10)
6   # use resource file for language
7   text "res" key: "header" (10, 40)
8   # use direct translations
9   text "trans" (en: "Hello" es: "Hola") (10, 70)
```

The first text will have the value Straight English Text regardless of the language choice by the viewer. The next text will take its translation from the specified resource. Note that the reader of the ANIMALSCRIPT file can *not* tell the final value of the text, as this is not embedded in the animation. Finally, the last text will be Hello or *Hola*, depending on whether the user chose *en* (English) or *es* (Spanish).
Figure 9 shows the output generated by line 5 of Listing 37.

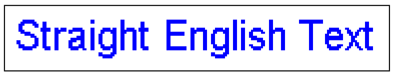
Straight English Text

Figure 9: Simple text example

The result of using internationalized text (taken from the above example) is as shown in Figure 10, assuming the user has chosen `en` as the output language. The first text primitive ("Hallo") comes from line 7 in Listing 37.

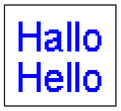

Figure 10: Internationalized text

4.6 Arc primitives

This type contains the following subprimitives:

1. *arc* and the synonymous *ellipseseg* for arcs (also called ellipse segments),
2. *ellipse*,
3. *circleseg* for segments of a circle, and
4. *circle*.

As the definition of the subtypes differ slightly, we describe them in separate subsections.
All arc types share the following attributes, beyond the standard attributes described before:

- *ID* is an arbitrary name for the component.
- The *first* node definition defines the *center* of the arc component.
- *displayOptions* determine if the object should be shown and if so, when. See Section 2 on page 4 for details.

Listing 38: Definition of the *arc* farmily or primitives

```
<arcTypes>:
  <arc> | <circleSeg> | <ellipse> | <circle>
```

4.6.1 Arc Primitive

Note: the name *ellipseSeg* is only provided for compatibility with *circleSeg*.

Listing 39: Definition of an *arc* primitive

```
<arc>:
  <arcName> "ID" <nodeDefinition>
    radius <nodeDefinition> [angle <int>]
    [starts <int>] [clockwise | counterclockwise]
    [color <color>] [depth <nat>]
    [<closedOptions> [<fillOptions>] | <arrowOptions>]
    [<displayOptions>]

<arcName>:
  arc | ellipseseg | ellipsesegment
```

- The *radius* specifies the *(x, y)* radius of the arc. Note that the width and height of the radius may be different.

- *arcAngle* specifies the total angle of the arc component; the value should lie in the interval `[0, 359]` or be `720` to handle a display problem in some Java Virtual Machines.
- *startAngle* is the starting angle of the arc; a value in the interval `[0, 359]` with 0 at the right (east) and counting clockwise.
- *clockwise* or the alternative *counterclockwise* determine whether the arc is oriented clockwise. If neither option is given, the default is *counterclockwise*, i.e., at a mathematically positive angle.
- The *closedOptions* described in Listing 40 currently only contain the keyword **closed**, which, if given, causes the segment to be closed (eg., a pie wedge).

Listing 40: Definition of the *closed options*

```
<closedOptions >:
  closed
```

- The *filledOptions* are described in detail in Section 4.3 on page 13.

The code in Listing 41 leads to the output shown in Figure 11.

Listing 41: Example definition of an *arc* primitive

```
  arc "ar2" (100, 100) radius (20, 40) angle 180 starts 45
    counterclockwise closed filled fillColor blue
```

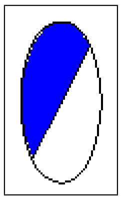

Figure 11: Example arc primitive

s

4.6.2 Circle Segment subtype

Circle segments are a special subtype of *arcs* where the radius width and height are identical. Thus, the definition replaces the node definition for the radius with an *integer* value. See Section 4.6.1 on the previous page for a specification of the other options.

Listing 42: Definition for *circle segment* primitives

```
<circleSeg >:
  <circleSegName> "ID" <nodeDefinition>
    radius <int> [angle <int>] [starts <int>]
    [clockwise | counterclockwise]
    [color <color>] [depth <nat>]
    [<closedOptions> [<fillOptions>] | <arrowOptions>]
    [<displayOptions>]

<circleSegName >:
  circleseg | circlesegment
```

The code in Listing 43 leads to the output shown in Figure 12.

Listing 43: Example definition of a *circle segment* primitive

```
circleSegment "cs" (100,100) radius 80 angle 270 starts 60 fwArrow bwArrow
```

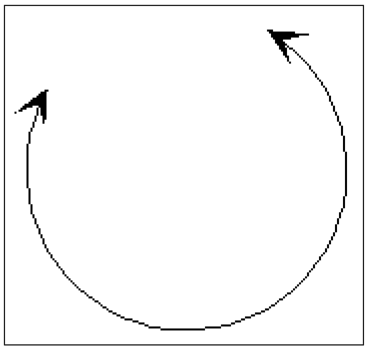

Figure 12: Circle Segment example output

4.6.3 Ellipse subtype

Ellipses, as defined in Listing 44, are a special subtype of *arcs* that describes a *closed* arc with an angle of 360°. Thus, the *angle, starts, clockwise, counterclockwise,* and *arrowOptions* parameters from the arc definition are obsolete. For an explanation of the other parameters, see Section 4.6.1 on page 19.

Listing 44: Definition of the *ellipse* primitive

```
<ellipse>:
  ellipse "ID" <nodeDefinition> radius <nodeDefinition>
    [color <color>] [depth <int>]
    [<fillOptions>] [<displayOptions>]
```

The output of the example *ellipse* command in Listing 45 is shown in Figure 13.

Listing 45: Example definition of an *ellipse* primitive

```
  ellipse "el" (50, 50) radius (30, 20) color blue filled fillColor red
```

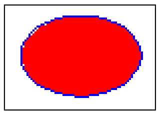

Figure 13: A filled ellipse with a blue outline and red fill color

4.6.4 Circle subtype

Circles are a special subtype of *arcs* describing a *closed* arc with an angle of 360°. Thus, the *angle, starts, clockwise, counterclockwise,* and *arrowOptions* parameters from the arc definition are obsolete. Additionally, the radius height and width are identical and thus specified by an *integer* value instead of a node definition. For an explanation of the parameters, see Section 4.6.1 on page 19.

Listing 46: Definition of the *circle* primitive

```
<circle >:
  circle "ID" <nodeDefinition> radius <int>
    [color <color>] [depth <nat>]
    [<fillOptions>] [<displayOptions>]
```

The example circle generated from the command in Listing 47 is shown in Figure 14.

Listing 47: Example definition of a *circle* primitive

```
  circle "cr" (70, 70) radius 35 filled fillColor red
```

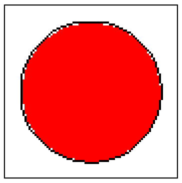

Figure 14: Circle primitive example

4.7 Array primitives

The *array* primitive supports the easy inclusion of arrays. Note that there are some specific operations on arrays described in Section 5.1 on page 30.

Listing 48: Definition of the *array* primitive

```
<array>:
  <arrayKey> "arrayID" [at] <nodeDefinition>
    [color <color>] [fillColor <color>]
    [elementColor <color>] [elemHighlight <color>]
    [cellHighlight <color>] [horizontal | vertical]
    length <nat+> { <intText> } [depth <nat>]
    [<timeOffset>] [cascaded [within <nat+> ticks | ms ]]
    [hidden] <font>

<arrayKey>:
  array | field
```

The *nodeDefinition* defines the *upper left corner* of the array.
The interpretation of the different *color* tags for arrays is listed in Table 4.7 on the next page.

- *horizontal* and *vertical* define the orientation of the array. The default orientation is *horizontal*.
- The *mandatory* keyword *length* is followed by the number of elements of the array.
- The number of *internationalText* components **must match** the array length. For "undefined" cells, use an *empty string* "" as a value.
- **cascaded** defines that the individual cells shall be displayed one after the other instead of all at the same time.
- the *font* specifies the base font to use for the array elements; see Section 2.6 on page 7.

Parameter	Visual effect
`color`	the color for the cell outlines
`fillColor`	the color used for filling the array cells (acting as a background color for the elements)
`elementColor`	the color in which the array elements are drawn.
`elemHighlight`	the color used for an element if it is highlighted, e.g. to indicate an upcoming *swap* operation. Highlighting is only performed at the explicit command by the animation author.
`cellHighlight`	the color used for filling a highlighted cell.

Table 1: Interpretation of the array colors

The code in Listing 49 illustrates the behaviour of array primitives. First, we generate a new array called "arr" with a red outline that contains five elements drawn in green. Then the array cell 1 (the second in the array - counting starts at 0) is highlighted, leading to a yellow fill color. Finally, the elements from position 2 to 4 are highlighted (set in blue). The output is shown in Figure 15. Remember that the complete code for the *array* primitive must span only one line; it is broken into three lines in Listing 49 only to make it more readable.

Listing 49: Example commands for animating an *array* primitive

```
array "arr" (30, 20) color red fillColor grey elementColor green
  elemHighlight blue cellHighlight yellow length 5 "2" "7" "11" "19"
  "25" font Monospaced size 20
highlightArrayCell on "arr" position 1
highlightArrayElem on "arr" from 2 to 4
```

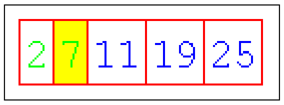

Figure 15: Array example, including cell and element highlight

4.7.1 Array marker

Array markers can be used to set a marker (an arrow pointing to an element) on an array. Note that there are some specific operations on array markers, described in Section 5.1 on page 30.

Listing 50: Definition of the *arrayMarker* primitive

```
<arrayMarker>:
  <markKey> "indexID" on "arrayID" atIndex <nat>
     [label <intText>] [short | normal | long] [color <color>]
     [depth <nat>] <font> [<displayOptions>]

<markKey>:
  arrayMarker | arrayPointer | arrayIndex
```

indexID is the name under which the array marker can be used in later commands. In contrast, *arrayID* must be the valid ID of a previously defined array.

The keyword **atIndex** must be followed by the index of the element to be pointed to. The value must be valid by being in the interval [0, arrayID length - 1]. As in C, Java and several other languages, ANIMALSCRIPT arrays *always* start at index 0.
The optional *label* can be an arbitrary text, as specified in Section 3.1 on page 10.
The three optional keywords *short, normal* and *long* offer three different lengths of the arrow. This can help in making the text of arrows more readable, especially if more than one array marker is installed on the same array.

4.8 List elements

ANIMALSCRIPT supports list elements consisting of a value and an arbitrary number of pointers to other list elements (or other graphical elements). Additionally, the position of the pointer box–above, below, to the left or to the right of the value–can be defined, as shown in Listing 51.

Listing 51: Definition of the *listElement* primitive

```
<listelement>:
  listelement "ID" <nodeDefinition> [text <intText>]
    pointers <nat> [position <pointerPos>]
    [ { <ptrLocation> } ] [prev "prevID"] [next "nextID"]
    [color <color>] [boxFillColor <color>]
    [pointerAreaColor <color>]
    [pointerAreaFillColor <color>]
    [textColor <color>] [depth <nat>]
    [<displayOptions>]

<pointerPos>:
  top | left | right | bottom | none

<ptrLocation>:
  ptr<nat> <nodeDefinition> | ptr<nat> to "targetID"
```

- The *nodeDefinition* defines the *upper left border* of the element.
- The definition of the optional *intText* value is given in Section 4.5.1 on page 18.
- The **mandatory** keyword *pointers* introduces the number of pointers, which must be in the interval [0, 255].
- The default pointer *position* is *bottom*.
- for *ptrLocation*, the text `ptr` with the appended `pointer number` is followed by the target coordinates of the pointer. Coordinates may be given in absolute coordinates or using relative placement.

  Note: `ptr<nat>` is written *as one word*, e.g. as `ptr1` or `ptr3`.
- The optional statements *prev* and *next* introduce the predecessor and predecessor, respectively, of the current list element, which must be a valid ID of another list element. *since* ANIMAL *2.3.19*
- The color definitions specify the color of the object:
  - the outline of the box containing the element value (*color*)
  - the fill color of the box containing the element's value (*boxFillColor*)
  - the outline color of the box containing the pointers (*pointerAreaColor*)

  - the fill color of the box containing the pointers (*pointerAreaFillColor*)
  - and the color of the element value (*textColor*).

  Note that the color of the *pointers* is identical to the *color* attribute.

- The *depth* and *displayOptions* are described in Section 2 on page 4.

The example command in Listing 52 leads to the list element primitive shown in Figure 16.

Listing 52: Example definition of a *listElement* primitive

```
listElement "le" (100, 100) text "Value" pointers 2 position bottom
  ptr1 (200, 200) ptr2 to "arr" boxFillColor grey
```

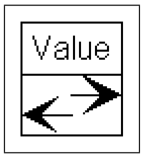

Figure 16: Example list element

4.9 Source Code embedding

The support for source or pseudo code in ANIMALSCRIPT is split into two different aspects:

- Declaring source code groups, including the setting of standard attributes,
- insertion of source code entries.

Listing 53: Declaration of the code types

```
<codeTypes>:
  <codeGroup> | <codeLine> | <codeElem>
```

Before a code component can be inserted, the user must first declare a *code group* as described in Listing 54.

Listing 54: Definition of a *codeGroup* primitive

```
<codeGroup>:
  codegroup "groupID" [at] <nodeDefinition>
    [color <color>] [highlightColor <color>]
    [contextColor <color>] <font>
    [depth <nat>] [<timeOffset>]
```

The components of the *code group* have the following meaning:

- *groupID* is the name of the code group. The user **must** keep track of this name, as code entries can only be shown in the context of a *code group ID*.
- *nodeDefinition* is the position of the upper left corner of the code group entries, thus defining the starting point of all entries.

- The colors of the code group follow the specification in Section 2.4 on page 6. They have the following meanings:
  - *color* - the standard color of all elements
  - *highlightColor* - the color used when a code element is highlighted
  - *contextColor* - the color used when a code element is highlighted in "context" mode. This can be used to highlight the environment or context of a current command. For example, an animation author may want to highlight the *current* command in one color, and show the enclosing statement—such as `if, for` or *method declaration*—in a different color.

    The *context color* is used whenever the **highlightCode** command introduced in Section 5.4 on page 33 is used with parameter **context**.
- All other entries have been defined in Section 2.

There are two different ways for entering a new code entry, as shown in Listing 55

Listing 55: Definition for adding *code lines* to a code group

```
<codeLine>:
  addCodeLine "code" [name "ID"] to "codeGroupID"
    [indentation <nat>] [<timeOffset>]
```

The *addCodeLine* command will insert a new code entry as a whole line of code. Note that the *code group ID* **must** be given for entering elements!
indentation can be used to cause automatic indentation based on the font size used. An indentation level of 0 means "no indentation". Each higher value will indent the text further to the right. *timeOffset* toggles the delayed display of individual lines.
The second form of code input is shown in Listing 56.

Listing 56: Definition for adding *code elements* to a code group

```
<codeElem>:
  addCodeElem "code" [name "ID"] to "codeGroupID"
    [column <nat>] [indentation <nat>] [<timeOffset>]
```

The only difference between the two notations is the optional entry *column*, which allows the specification of the entry in the appropriate position in the *current* code line. This is especially useful for multi-part statements such as *for* that shall highlight the currently executed individual parts of the statement separately.
The *code* commands *codeGroup, addCodeLine,* and *addCodeElem* using the same ID belong together, as they define the same code group, but span several lines. Therefore, they should **always** be placed in a composite step using {}.

4.10 Grid Primitives

since 2.3

A *grid* represents different three different visualizations of a two-dimensional data set: a plain ordering, a matrice and a table. Grids are among the most versatile, but also the most complex data structures supported in ANIMALSCRIPT.
Listing 57 shows the definition of a grid primitive in ANIMALSCRIPT. Please see Section 5.16 for details on how to modify a grid.

Listing 57: Definition of the *grid* primitive

```
<grid>:
  grid "id" <nodeDefinition> lines <nat+> columns <nat+>
  [style <gridStyle>]
  [cellWidth <nat+>] [maxCellWidth <nat+>]
  [cellHeight <nat+>] [maxCellHeight <nat+>] [fixedCellSize]
  [color <color>] [textColor <color>] [borderColor <color>
  [fillColor <color>] [highlightTextColor <color>]
  [highlightFillColor <color>] [highlightBorderColor <color>]
  [<font>] [align <gridAlign>] <depth> <timeOffset>

<gridStyle>:
  plain | matrix | table

<gridAlign>:
  left | center | right
```

nodeDefinition describes the upper left corner of the grid.

lines, columns define the number of rows and columns of the grid. Both values must be at least 1.

style is a choice of the three default values *plain, matrix* and *table*. If no option is specified, *plain* is assumed. A *plain* grid aligns only the elements, but draws no border or grid lines. A *matrix* places the matrix lines with curved edges around the elements. Finally, a *table* will place all elements into rectangular boxes. See Figure 17 for an example of the three different styles.

cellWidth, maxCellWidth, cellHeight, maxCellHeight allow the user to specify the size of the cells. Both *cellWidth* and *cellHeight* define the initial size, while *maxCellWidth* and *maxCellHeight* define the maximum size of the cells. *fixedCellSize* can be used to define that the value used for *cellWidth* and *cellHeight* are used for *maxCellWidth* and *maxCellHeight*, respectively.

color, textColor, borderColor, fillColor define the basic colors for the grid. *color* is used as a fall-back value for the colors that the user has not specified. Intuitively, *textColor* defines the color for the lements, *borderColor* the color for the border (in a *matrix* or *table*), and *fillColor* represents the fill color of the box containing a given grid element.

highlightTextColor, highlightFillColor, highlightBorderColor are the colors used if one or more elements (*highlightTextColor*), one or more cells (*highlightBorderColor*) or the border of the grid are highlighted (*highlightBorderColor*).

font defines the font for the elements.

align uses the three values *left, center* and *right* to specify the alignment of the elements inside their cells. the default alignment is *left*.

Listing 58 shows three example grids that will result in the display shown in Figure 17. The grid on the left is created with the *plain* style, the center grid is a *matrix* and the grid to the right uses the *table* style.

Listing 58: Example of three *grid* definitions

```
grid "grid1" (10,10) lines 5 columns 5 style plain
setGridValue "grid1[][]" "10" refresh
grid "grid2" (185,10) lines 5 columns 5 style matrix
setGridValue "grid2[][]" "8" refresh
grid "grid3" (360,10) lines 5 columns 5 style table
setGridValue "grid3[][]" "6" refresh
```

Note that a grid defined as above appears "empty" and thus potentially invisible unless values are assigned to the grid. See also Section 5.16 for details on working with grids and especially Section 5.16.1 for addressing grid elements.

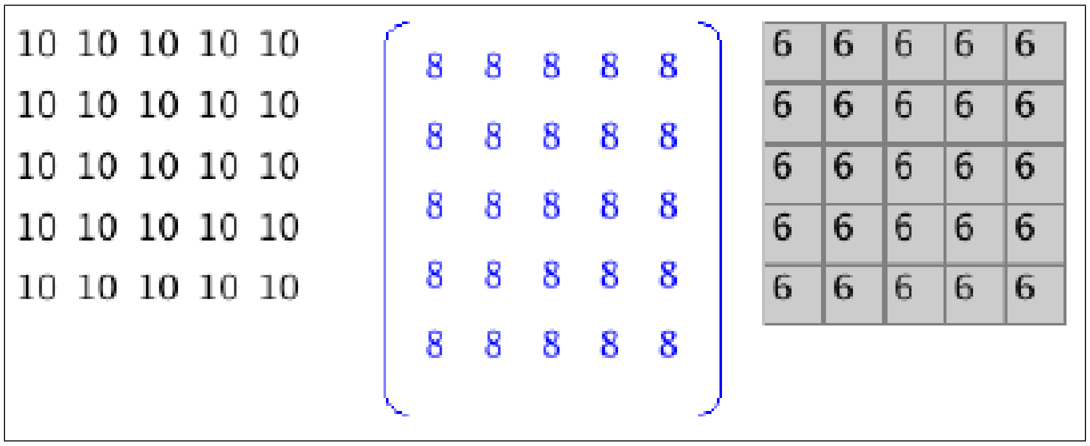

Figure 17: Grid primitive examples

4.11 Graphs

since 2.3

Listing 59 shows the definition of the *graph* primitive, which together with the *grid* primitive described in Section 4.10 on the previous page is one of the most complex and expressive primitives in ANIMALSCRIPT.

Listing 59: Definition of the *graph* primitive

```
<graph>:
 graph "ID" size <nat+> [color <color>] [bgColor <color>]
  [outlineColor <color>] [highlightColor <color>]
  [elemHighlightColor <color>] [nodeFontColor <color>]
  [edgeFontColor <color>] [directed] [weighted]
  nodes "{" [<font>] { "value" [at] <nodeDefinition> [","] } "}"
  edges "{" [<font> { "(" <nat+> "," <nat+> ["," <intText>] ")" [","]"}"
  [origin <nodeDefinition>] [showIndices] [depth <nat>] <timeOffset>
```

Please note that similar to the *array* and *grid* primitives, the size of a graph is *fixed* once it was defined.

color, bgColor, outlineColor define the basic colors of the graph. *color* is the basic color for a graph. *bgColor* is the background color of each node. *outlineColor* is the color used for the edges and the borders of the nodes.

highlightColor, elemHighlightColor are the colors for highlighting the nodes and edges (highlightColor) and the values inside the nodes (elemHighlightColor).

nodeFontColor, edgeFontColor are the colors for the value of the nodes and the edge weight (if any), respectively.

directed, weighted allow the user to specify that the graph is *directed* (i.e., an edge from A to B does not mean that there is an edge from B to A) and / or *weighted* (i.e., each edge has a weight or “cost” associated with it).

Node positions start with the keyword *nodes* followed by a list of definitions placed inside curly braces. Before the first entry—and only there—the font used for all nodes can be given. Each node definition starts with the *value*, followed by the optional keyword *at* and the *nodeDefinition* as described in Section 2.2 on page 4. The individual entries are separated by a comma.

Edges are defined by the keyword *edges* followed by an opening brace and an optional definition of the font for edge weigths. Each edge is a tuple or triplet of values, depending on whether the graph is *weighted*. The first value represents the source node index of the edge (starting at 0), the second value represent the index of the target node of the edge. If the graph is *weighted*, the third element is the weight of the edge. The definition of the edges

origin can be used to translate the graph to the position given by the *nodeDefinition* following the keyword.

showIndices displays the index of each node inside the node. This feature is not primarily mreant for productive uses, but can be used to help during the generation and validation of an animation.

Figure 18 shows an example graph generated by the code in Listing 60. Again, the line breaks in Listing 60 are only used for layout reasons; the full command must occupy exactly one line.

Listing 60: Example code for a *graph* primitive

```
graph "graph" size 5 directed weighted nodes { "A" (40, 100),
  "B" (160, 100), "C" (40, 220), "D" (160, 220), "E" (100, 160) }
  edges { (0, 1, "15"), (0, 2, "3"), (1,3, "7") (4,2,"8"), (2,0,"11") }
  showIndices
```

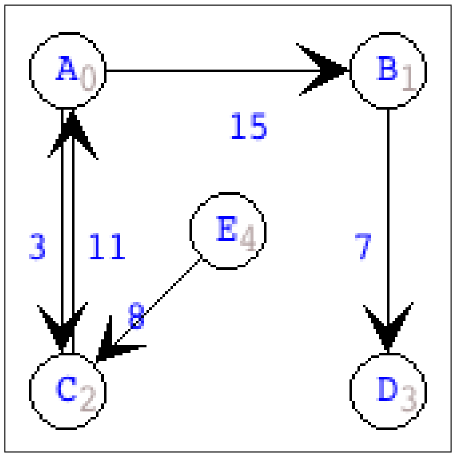

Figure 18: Graph primitive examples

ANIMALSCRIPT also supports several operations on graphs. These will be described in detail in Section 5.17 on page 43.

5 Supported Operations

ANIMALSCRIPT supports a wide selection of operations. The overview definition in Listing 61 gives a first impression. In the course of this Cection, we will explore each type of operation in detail.
All operations allow the specification of an optional *timing*, which can define both an *offset* and a *duration* for the operation - see Section 2.5 on page 7.

Listing 61: Definition of the animation effects

```
<operations >:
  <arrayOp> | <clone> | <colorChangeTypes> | <delay>
   | <echo> | <label> | <link> | <location>
   | <merge> | <move> | <rotate> | <showTypes>
   | <swap> | <setText> | <setFont> | <variableSupport>
   | <assertion> | <gridOperations> | <graphOperations>
```

.

5.1 Array Operations

The array operations defined in Listing 62 are supported:

Listing 62: Array operations overview

```
<arrayOp >:
  <arrayPut> | <arraySwap> | <moveArrayMarker>
   | <highlightArrayCell> | <highlightArrayElem>
```

5.1.1 Putting Elements in an Array

Listing 63 defines the notation for the *arrayPut* command.

Listing 63: Definition of the *arrayPut* command

```
<arrayPut >:
  arrayPut "value" on "arrayID" position <nat>
    [<timing >]
```

Using the *arrayPut* command, one can easily cause elements to be put in a given array (see Section 4.7 on page 22 for information on how to define an array in ANIMALSCRIPT). The user only has to give the new *value* for the element, the *arrayID* and the *targetPosition*. The *timing* is optional and can provide both *offset* and *duration*.
ANIMALSCRIPT keeps track of the arrays and prints a warning if the specified position is invalid, for example because the *array ID* was mistyped, or the user entered a position which is too small (< 0) or too large (greater or equal to the array length as defined in the array declaration - see Section 4.7 on page 22).
Please note that this operations usually requires *moving* all components following the insertion position and will thus cause a slightly changed layout of the whole array including its bounding box. Due to a possibly large number of position lookups, the operation can become slow when using large arrays or many objects.

5.1.2 Swapping Array Elements

The *arraySwap* defined in Listing 64 supports easy swapping of array elements in any array defined as described in Section 4.7 on page 22. The command requires the *array ID* and the two swap *positions*.

Listing 64: Definition of the *arraySwap* command

```
<arraySwap>:
  arraySwap on "arrayName" position <nat> with <nat>
    [<timing>]
```

Please note that this operations usually requires *moving* all components lying between the two swapped components and may thus cause a slightly changed layout of the whole array, but will *not* change the array's bounding box. Due to a possibly large number of position lookups, the operation can become slow when using large arrays or many objects.

5.1.3 Moving Array Markers

Listing 65 presents the notation for updating array markers.

Listing 65: Definition of the commands for moving array markers

```
<moveArrayMarker>:
  <moveMarkerKeyword> "markerID" to
    [position <nat> | arrayEnd | outside] [<timing>]

<moveMarkerKeyword>:
  moveArrayIndex | moveArrayMarker | moveArrayPointer
  | moveIndex | moveMarker | movePointer
  | jumpArrayIndex | jumpArrayMarker | jumpArrayPointer
  | jumpIndex | jumpMarker | jumpPointer
```

The large set of (equivalent) commands stems from the diverse notations found in related systems, and when authors write about array indices. We recommend that you simply choose one tag and stick to it.
Using these commands, moving reference pointers to array cells is easily accomplished. The underlying array and array marker must be defined as described in section 4.7. The command expects the *array ID* and the *target position*.
There are three different operations for moving the array marker:

- To a given *position* in the array, i.e. an integer value in the interval `[0, arrayID.length - 1]`,
- to the end of the array (the last array element),
- to the "outside" (slightly to the right of the last array element).

Timing is optional and can provide both *offset* and *duration*.
The *jumpXXX* commands are reserved for *instanteous* movements having *a duration of 0*. While ANIMALSCRIPT is not very strict about this, it *will* reset the duration to 0 whenever *jumpXXX* is used.

5.1.4 Highlighting Array Cells or Elements

Listing 66 defines the notation for highlighting and unhighlighting array cells or elements. The highlighting operation is usually employed to direct the end user's attention towards the current elements, for example, the two elements of the array just being compared.
ANIMALSCRIPT does not make a semantic difference between highlighting a "cell" (changing its fill color to the *cellHighlight* color) and highlighting an "element" (changing its color to the *elemHighlight* color, as defined in Section 4.7 on page 22. It is recommended to use the *element*-based operation when the attention of the end user shall be drawn to the *value* of the element, e.g., because this is used in the current calculation. Conversely, the cell should be highlighted it the focus is on the cell (position) rather than on the value inside the cell.

Additionally, by using a color such as *grey* as the cell highlight color, parts of the array that are currently of no relevance can be "shadowed out". This can be useful for recursive algorithms, such as Quicksort or Binary Searching, to help the end user focus on the remaining "relevant" elements.

Listing 66: Definition of the commands for highlighting array cells or elements

```
<highlightArrayCell >:
  <hlACellKeyword> on "arrayID" <aHighlightRange>
  <timing>

<hlACellKeyword >:
  highlightArrayCell | unhighlightArrayCell

<aHighlightRange >:
  position <nat> | [from <nat>] [to <nat>]

<hilightArrayElem >:
  <hlAElemKeyword> on "arrayID" <aHighlightRange>
  <timing>

<hlAElemKeyword >:
  highlightArrayElem | unhighlightArrayElem
```

These two very similar elements allow the highlighting or unhighlighting of array cells or elements. Apart from the *array ID*, the user has to define the target of the effect as well as an optional timing. The affected array elements may be either a fixed *position* within `[0, arrayID.length - 1]` or a *range* with the two optional keywords *from* and *to*. If *neither* is given, the range is assumed to cover the full array.

5.2 Object Cloning

Listing 67 defines the notation for the *clone* operation. This command supports the easy cloning of a given object. The user has to provide a new location for the cloned element, as well as a new reference ID. The standard display options apply.

Listing 67: Definition of the *clone* operation

```
<clone >:
  clone "originalID" as "cloneID" [at] <nodeDefinition>
    [<displayOptions >]
```

5.3 Color Change Operations

ANIMALSCRIPT supports the standard *color change* operations defined in Listing 68.

Listing 68: Definition of the different color change operations

```
<colorChangeTypes >:
  <colorChange> | <codeColorChange>

<colorChange >:
  color <oids> [type <colorType >] <color> [<timing >]

<colorType >:
  "color" | "fillColor" | "textColor" | "colorSetting"
```

Note that the effect can operation simultaneously on an *arbitrarily long (non-empty)* list of target IDs for this operation. Some object types may provide *additional* color change types which are *not* listed here. Additionally, some of the color types are *not* available for all objects, following common sense. For example, applying a *fillColor* animator to a *polyline* is semantically incorrect.
The "*colorSetting*" entry can specify an arbitrary (but existing) color setting to be changed.

5.4 Code Operations

The color changing operations for code elemetns supported in ANIMALSCRIPT are defined in Listing 69.

Listing 69: Definition of the color change effects for *code* elements

```
<codeColorChange>:
  <codeColorType> on "baseCodeGroup" line <nat>
      [column <nat>] [context | region] [<timing>]

<codeColorType>:
  highlightCode | unhighlightCode
```

Note that the user *must* provide the *code group ID* for changing the code color – even if it is for only a single line! The optional **column** *c* statement is used for *code elements*, as opposed to whole *code lines*. The **context** command will use the *code context color* instead of the *code highlight color* – see section 4.9 on page 25.
Note: in previous versions of ANIMALSCRIPT, the keyword "row" was used, although "column" was meant. This has now been corrected in the parser; the older notation is still parsed correctly, but now regarded as deprecated.

Listing 70: Definition of the operation for showing or hiding a code group

```
<codeHide>:
  hideCode "codeID" [<timeOffset>]
```

The **hideCode** command defined in Listing 70 is used to hide the display of a whole code group after an optional offset. It is far more comfortable to use than manually hiding all elements of the code group, and should thus be the preferred choice.

5.5 Delay Between Steps

For installing a *customized delay* between individual steps on a *millisecond (ms)* base, the *delay* command defined in Listing 71 can be given in a separate line which must be *outside* composite steps:

Listing 71: Definition of delays between steps using *delay*

```
<delay>:
  delay <nat> [ms] | delay click on "elemID"
```

This will cause the animation to wait *i* milliseconds before continuing with the next step. The notation using *click* 2.0 is new in ANIMALSCRIPT 2.0. It will pause the animation until the user clicks on the object specified by the ID. Note that this operation is *not yet properly implemented and left for further implementation*. *since 2.0*
Caution: if you mistype the ID, the object is currently invisible, or it is completely occluded by other objects, this will effectively freeze the animation.

5.6 Printing Status Information

For printing status information, ANIMALSCRIPT provides the very useful *echo* command defined in Listing 72.

Listing 72: Definition of the *echo* command for debug output

```
<echo>:
  echo location: <nodeDefinition>
   | echo <boundsKeyword>: { <oids> }
   | echo text: "text"
   | echo value: { "ID" }
   | echo ids: { <oids> }
   | echo visible
   | echo rule: "keyword"
   | echo unquotedText

<boundsKeyword>:
  boundingBox | bounds
```

echo supports the following operations:

echo location: can be used to print out the location of any given node, with full support for *relative placement*, for example **echo location**: **offset** (20, −30) **from** "A" **SE**

echo bounds: is used to print the *bounding box* of any object at the current animation state. Alternatively, the keyword *boundingBox:* can be used. This can also be done for multiple objects at the same time: **echo bounds**: "A" "B" *since 2.0*

echo text: prints the text passed in, for example to explain the next output, or for debugging just when an error occurs. For example, you can use **echo text**: "All OK till now, Bounding Box **of** 'B' is ..."

echo value: prints the value of a variable that has been assigned a value . This is currently only supported by the optional Assertion extension package. *since 2.0*

echo ids: prints the ID(s) belonging to a certain identifier. This is especially helpful during animation development when one is dealing with a *group* of objects merged under a common label. *since 2.0*

echo visible display the list of IDs of all currently visible objects.

echo rule: prints the rule syntax for an arbitrary element. The user has to provide the keyword starting the rule, for example *echo*, within double quotes. Note that this only works if the rule database is up to date and has been properly maintained by all authors. *since 2.0*

echo followed by unquoted text will simply print the text following the *echo* keyword.

These commands are helpful both for debugging an animation if entries seem to be *invisible* (but actually are only placed outside the animation window), and for better *tuning* of especially generated animations.

5.7 Element Grouping

To perform several operations on the same set of elements, it quickly becomes tedious to enumerate all elements each time. Therefore, ANIMALSCRIPT offers grouping and ungrouping commands as shown in Listing 73.

Listing 73: Definition of the commands for grouping and ungrouping elements

```
<merge>:
  <mergeKeyWord> "targetID" { "ID" }

<mergeKeyWord>:
  group | merge | set | ungroup | remove
```

Simply choose between the *merge / group / set* operations for gathering elements under a new ID, or *remove / ungroup* to remove elements from a collection ID. Apart from the collection ID – which *must not* be in use for *merge, group, set*, but *must* already be used for *remove / ungroup* –, the user only has to specify the required object IDs.

5.8 Generating Navigation Labels

The **label** statement defined in Listing 74 can be used for generating *navigation labels* in a separate window.

Listing 74: Definition of the *label* operation

```
<label>:
  label "labelEntry"
```

Place this statement either *within* a composite step or *after* the statement you want to label. The *labelEntry* will be added to ANIMAL's *Timeline Window* and can be used for jumping directly to the referenced step.

5.9 List Operations

To manipulate the pointers or *links* used in a list, use the commands specified in Listing 75:

Listing 75: Definition of the *setLink* and *clearLink* commands

```
<link>:
  setLink "elemID" [link <nat>] to "targetID" [<timing>]
  | setLink "elemID" [link <nat>] <nodeDefinition> [<timing>]
  | clearLink "elemID" [link <nat>] [<timing>]
```

setLink will set the link of the given object to the target object, while *clearLink* will reset the link. Note that if the list elements have more than only one link, the user should *always* give the optional statement *link* followed by the number of the link to be used. ANIMALSCRIPT does not enforce this and will always assume the user meant the *first* pointer. However, for consistency and readability purposes, it is better to explicitly state the link number in all cases.
Link numbers are counted starting with 1.
ANIMALSCRIPT 2.0 also allows the user to set the pointer to an arbitrary location. *since 2.0*

5.10 Move Operations

One of the most versatile operations, **move** commands come in several subtypes in ANIMALSCRIPT, including the synonymous keywords **move, translate**. See Listing 76 for an overview of the move types.

Listing 76: Overview of the move types

```
<move>:
  <moveVia> | <moveAlong> | <moveTo>

<moveKeyword>:
  move | translate
```

Two common parts used for all *move* operations are specified in Listing 77: *type* and *corner*.

Listing 77: Definition of the method subtype and the move corner

```
<methodSpec>:
  type "typeName"

<corner>:
  corner <direction>
```

Many primitives offer special *move types*. For a complete list, see Appendix A.1 on page 45 or consult the ANIMAL Home Page at `http://www.algoanim.info/Animal2`.

5.10.1 Move Via ...

Listing 78 defines the notation for moving an object *via* another object.

Listing 78: Definition of the *move via* command

```
<moveVia>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
   via "oid" [<timing>]
```

The basic **move** operation, this will simply move the given objects via the object with ID `oid`. *oid* must be a valid *MoveBase*, that is, it must be any of the various subtypes of *polyline*, *polygon* or *arc*.

5.10.2 Move Along ...

Listing 79 defines the notation for the *move along* command. This **move** subtype allows the user to inline a component definition along which the move is to take place. Simply use the command by choosing the *keyword, target IDs, optional move corner* and possibly *move method*, and provide the component.

Listing 79: Definition of the *move along* command

```
<moveAlong>:
  <moveAlongPolyline> | <moveAlongArc>

<moveAlongPolyline>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
      along <lineTag> <nodeDefinition> { <nodeDefinition> }
      [<timing>]

<moveAlongArc>:
  <moveAlongArcType> | <moveAlongCircleType>

<moveAlongArcType>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
     along <arcType> <nodeDefinition> <int> <int>
      <int> <int> [<timing>]

<moveAlongCircleType>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
     along <circleType> <nodeDefinition> <int> <int>
     <int> [<timing>]
```

ANIMALSCRIPT allows moves along either *polyline* or *arc-based* objects, supporting both *ellipse* and *circle segments*. See Listing 80 for the valid type names for the associated *move along* command subtypes. Note that the notation for providing the move object *inlined* is very similar to the specification as described in the object sections.

Listing 80: Definition of the keywords for the *move along* move object type

```
<arcType>:
  <arcName> | ellipse

<circleType>:
  circle | <circleSegName>
```

5.10.3 Move To ...

Using the *move to* notation defined in Listing 81, the user can specify that all object IDs passed are to be moved to the given target node. Of course, the *node definition* can also use *relative coordinates*.

Listing 81: Definition of the *move to* command

```
<moveTo>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
   to <nodeDefinition> [<timing>]
```

The user should keep two aspects in mind about this command:

- all objects will have their left upper point of the bounding box at *precisely the same spot* after the operation. This usually also means that the space below and to the right of this spot may become cluttered with objects, and that many objects may be at least partially obstructed by others.
- as the objects to be moved can lie far apart, this operation is transformed into *several* move operations—usually *as many* as the number of given objects IDs. These operations will take place using the *same timing*.

5.11 Rotation Operation

The *rotate* command specified in Listing 82 allows the user to *rotate* entries either around a **point** passed via its *oid*, or around an *inclined rotation center*. Keep in mind that rotations are performed in mathematical order, i.e. *counterclockwise*.

Listing 82: Definition of the *rotate* command

```
<rotate>:
  rotate <oids> around "id" [degrees <int>] [<timing>]
 | rotate <oids> center <nodeDefinition> [degrees <int>]
   [<timing>]
```

The optional **degrees** statement expects values between `0` and `360`, but is rather lax in its parsing.
Please note that ANIMALSCRIPT only supports the rotation of *polyline-* or *polygon*-based components including *all* subtypes. Also note that the *oid* must be an ID of a **point** object—all other objects will lead to an error message.

5.12 Show / Hide Operation

The *show* and *hide* commands and their counterparts, as defined in Listing 83, modify the visibility of objects. Somewhat unsurprisingly, **show** makes components visible, and **hides** causes them to "disappear". Note that the operations for hiding code have already been defined in Section 5.4 on page 33.

Listing 83: Definition of the *show / hide* commands

```
<showTypes>:
  <simpleShow> | <codeHide> | <selectiveHide>

<simpleShow>:
  <showMode> <oids> [<timing>]

<codeHide>:
  hideCode "codeID" [<timeOffset>]

<selectiveHide>:
  hideAll [<timing>]
  | hideAllBut { <oids> } [<timing>]

<showMode>:
  show | hide
```

As of ANIMAL version 2.0, ANIMALSCRIPT also offers operations for hiding *all* objects or all objects *except* for a list of IDs. This is very helpful if the user wants to start the next step with a "clean plate", except for a few objects. *since 2.0*

5.13 Swap Operation

The **swap** command defined in Listing 84 exchanges the *object IDs* of two elements. This will **not** have any visible effect, but can prove very helpful if the animation is generated by programs. Note that one should **never** use **swap** on *array* or *code elements* – the IDs are updated automatically whenever an operation takes place, for example **arraySwap**.

Listing 84: Definition of the *swap* command

```
<swap>:
  <swapKeyword> "oid1" "oid2"

<swapKeyword>:
  swap | exchange
```

5.14 Changing the Text or Font of a Text Component *since 2.0*

Listing 85 defines the syntax for the *setText* and *setFont* commands. They can be used to change the text or font, respectively, of a text component, and should be self-explanatory.

Listing 85: Definition of the *setText* command

```
<setText>:
  setText [of] "oid" [<methodSpec>] to <intText> <timing>

<setFont>:
  setFont [of] "oid" [to] <font> <timing>
```

5.15 Operations on Variables

since 2.2

Listing 86 defines the notation for operations on variables supported in ANIMAL 2.2+.

Listing 86: Operations on variables

```
<variableSupport>:
  variable "id" [type <varType>]
  | assign "id" = <int>
  | <assertion>

<varType>:
  int | double | String

<assertion>:
  <assertKeyword> <assertionPart>

<assertKeyword>:
  assert | check

<assertionPart>:
  variable <comparator> <int>
  | <assertionPart> <boolOperator> <assertionPart>

<comparator>:
  == | != | < | <= | >= | >

<boolOperator>:
  && | ||
```

Variables are declared with the keyword *variable*, followed by the name of the variable in double quotes. The optional type can be defined as *int, double* or *String*; by default, all values are regarded as *String*.
Values are assigned using the *assign* keyword. Additionally, rudimentary conditions can be checked using the *assert* or *check* keywords (which are identical; pick the one you prefer). If the condition specified in *assert* is not evaluated to *true*, an error will be printed.
Note: at the moment, the support for variables as outlined here is under revision and may change in future versions. Additionally, the current implementation is not bug-free.

5.16 Grid Operations

since 2.3

5.16.1 Addressing Grid Positions

As already shown in Listing 58 on page 28, ANIMALSCRIPT offers powerful ways to address a grid. In contrast to most other data structures, a *grid* is initially empty (no visible elements), until elements are assigned to the positions.
Elements in a grid can be address in a set of different ways. The following ways to address a grid are supported by ANIMALSCRIPT, where we assume that *"grid"* is the name for a valid grid and that all addressed positions actually exist. *Keep in mind that rows and columns both start at 0.* Listing 87 provides the BNF for the notation.

Listing 87: Definition of the *cellIdentifier* for accessing grid elements

```
<cellIdentifier>:
  "id[<nat>][<nat>]"
```

A single cell is addressed by giving the name of the basic grid, followed by the index of the cell enclosed in one pair of brackets each. As with any other name in ANIMALSCRIPT, the full name has to be placed in double quotes, for example as "`grid[3][2]`" to address the cell in the fourth row and third column.

A complete column is addressed by giving the name of the basic grid, followed by an empty pair of brackets (in order to match all rows) and finally the chosen column number placed in a pair of brackets. For example, "`grid[][2]`' will access all elements in column 2 of the matrix, independent of their row number.

A complete row is addressed by giving the name of the basic grid, followed by the chosen row number in a pair of brackets and ending with an empty pair of brackets (in order to match all columns). For example, "`grid[3][]`" will access all elements in row 3 of the matrix, independent of their column number.

The complete grid can be address by adding *two* pairs of empty brackets to the grid name. Thus, "`grid[][]`" will address the complete grid.

Listing 88 shows the definition and access for four example grids, all of which are defined using the *table* style. The values *t1-t4* illustrate the four ways to address a grid in the order presented above. Figure 19 shows the results of the operations.

Listing 88: Example of addressing grid positions

```
{
  grid "tb1" (20 , 20 ) lines 4 columns 5 style table
  grid "tb2" (180, 20 ) lines 4 columns 5 style table
  grid "tb3" (20 , 150) lines 4 columns 5 style table
  grid "tb4" (180, 150) lines 4 columns 5 style table
  setGridValue "tb1[3][0]" "t1" refresh
  setGridValue "tb2[] [0]" "t2" refresh
  setGridValue "tb3[3][ ]" "t3" refresh
  setGridValue "tb4[ ][ ]" "t4" refresh
}
```

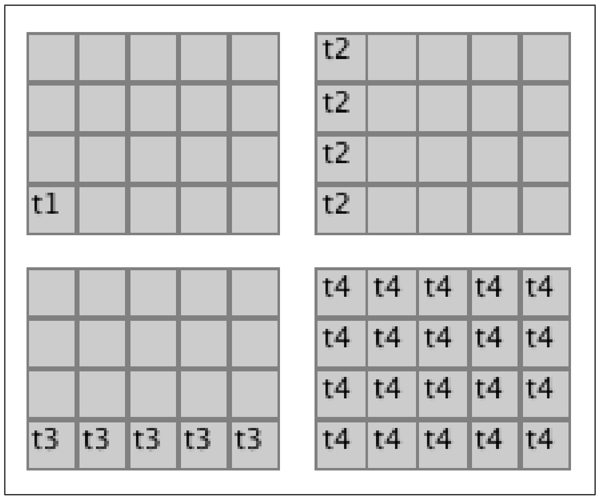

Figure 19: Grid addressing examples

5.16.2 Refreshing the Display of a Grid

All operations that may result in a redisplay of a grid, such as assigning new values to a grid, have a parameter *refresh*. If this parameter is not given, the operation will be executed *without re-layouting the grid*. Thus, the newly inserted value–if somewhat larger than the previous value–may overlap with a neighboring cell.
If multiple operations are used in the same animation step, e.g. by initializing a grid cell by cell, it makes sense to provide the *refresh* parameter only with the last command that modifies the grid. Each refresh operation requires a recalculation of the grid borders and layout and thus is costly in terms of runtime; avoiding unneccesary refreshs can therefore reuce the load time for an animation.
In Listing 88, the *refresh* parameter was given each time. However, this was necessary because we were working on four different grids: each *refresh* command affects only the grid that was associated with the current command.

5.16.3 Updating a Grid Value

Grid values can be updated with the *setGridValue* command defined in Listing 89. Please see Section 5.16.1 for the different ways how a grid can be addressed using the *cellIdentifier*. The *intText* entry is a standard ANIMALSCRIPT text. Please also see Section 5.16.2 for a description of the optional *refresh* command and Figure 19 on the previous page for an example of the *setGridValue* command.
As a special feature, the text to be inserted will be inserted character by character if the animation effect has been specified with a non-zero duration. If the *refresh* command is given, the borders of the cells will also be animated during the transition (if necessary).

Listing 89: Definition for setting a grid value

```
<setGridValue>:
  setGridValue <cellIdentifier> <intText> [refresh] <timing>
```

5.16.4 Highlighting Grid Cells or Elements

Grid cells and elements can be highlighted or unhighlighted using the *(un-)highlightGridCell* and *(un-)highlightGridElem* commands. These operations only change the color of the cell or element and thus do not affect the size of the grid cells. Therefore, this operation does not contain the optional *refresh* command found in many other grid operations.

Listing 90: Definition of the commands for (un-)highlighting grid elements/cells

```
<highlightGridComponent>:
  highlightGridCell <cellIdentifier> <timing>
  | highlightGridElem <cellIdentifier> <timing>
  | unhighlightGridCell <cellIdentifier> <timing>
  | unhighlightGridElem <cellIdentifier> <timing>
```

5.16.5 Changing the Color of Grid Cells or Elements

In contrast to arrays, each grid cell stores its own text, border and fill color, as well as the associated highlight colors. These colors can be updated individually using the *setGridColor* command defined in Listing 91. If a duration is specified, the color will be linearly interpolated between the start and the end color. Changing the highlight color (no matter of which aspect) will only become visible if the associated cell or element is highlighted.
For a definition of the meaning of the different color names, please see Section 4.10 on page 27.

Listing 91: Definition for setting the color of a grid cell or element

```
<setGridColor>:
  setGridColor <cellIdentifier>  [color <color>]
  [textColor <color>] [fillColor <color>]
  [highlightTextColor <color>] [highlightBackColor <color>]
```

5.16.6 Aligning Grid Values

Listing 92 defines the command for aligning the contents of a given cell. This operation requires the recalculation of the positions of the element. If the *refresh* command is give, the complete display will be re-layoutet for consistency reasons (see Section 5.16.2 on the previous page for more information about the *refresh* command).

Listing 92: Definition for aligning grid values

```
<alignGridValue>:
  alignGridValue <cellIdentifier> <gridAlign> [refresh] <timing>
```

If a duration is specified and the *refresh* command was given, the realignment of the element(s) will be animated.

5.16.7 Changing the Font of a Grid Element

Listing 5.16.7 defines the notation for changing the font of a given grid element. Please see Section 5.16.1 on page 39 for information on how to specify the target element(s) using the *cellIdentifier*.
Changing the font will often require a recalculation of the grid cell size, so that the command also supports the *refresh* command. If *refresh* is not specified, the font will be changed, but the size of the cells will remain as before—and may thus become too small or too large.

Listing 93: Definition for changing the font of a grid element

```
<setGridFont>:
  setGridFont <cellIdentifier>  <font> [refresh]
  [within <nat> [ticks | ms]]
```

If a duration is specified and the *refresh* command was given, the font change will be animated by the update of the grid cell. At the end of the duration, the font will change to the target value; there will be no size interpolation or "font interpolation".

5.16.8 Swapping Grid Values

The *swapGridValues* command defined in Listing 94 supports swapping individual cells, columns rows or even complete grids. The elements to be swapped "fly" to their respective target position; the border and background elements and colors are not affected. All other cell properties, such as the element font, remained tied to the grid position and thus do not "follow" the swapped value.

Listing 94: Definition of the *swapGridValues* command

```
<swapGridValues>:
  swapGridValues <cellIdentifier> and
    <cellIdentifier> [refresh] <timing>
```

The two *cellIdentifiers* may refer to different grids. However, they must affect the same number and type of elements, so that only the following swaps are possible:

- individual cell swapped with an individual cell,
- a single column of length *n* is swapped with a different column with the same length,
- a row of length *n* is swapped with another row of the same length.

If a timing is given, the move of the elements is animated. The affected cells will be adapted while the new values are still "on their way", to ensure that the cell sizes will fit the elements when they arrive.

5.17 Graph Operations

Listing 95 presents the operations available on graphs.

Listing 95: Operations on *graph* primitives

```
<graphOperations>:
  <edgeTransformation> | <changeWeight> | <nodeTransformation>

<edgeTransformation>:
   highlightEdge on "ID" [<methodSpec>] { "(" <int> "," <int> ")" }
   | unhighlightEdge on "ID" [<methodSpec>] { "(" <int> "," <int> ")" }

<changeWeight>:
   setEdgeWeight [of] "ID" edge "(" <int> "," <int> ")" to <int>

<nodeTransformation>:
   highlightNode on "ID" [<methodSpec>] nodes { <int> } <timing>
   | unhighlightNode on "ID" [<methodSpec>] nodes { <int> } <timing>
```

Edges can be highlighted and unhighlighted using the *(un-)highlightEdge* command, giving the graph ID and the set of edges to be (un-)highlighted. Each edge is given as a tuple consisting of the index of the start and target node, placed in parentheses and separated by a comma.
To adapt the weight of an edge, use *setEdgeWeight*. The command uses the same notation for specifying the edge as described in the previous paragraph.
Additionally, nodes can be highlighted or unhighlighted. Here, the list of nodes to be highlighted is given as a sequence of the node indices.

6 Supported Color Names

Page 6 contains a list of the predefined color names. The color names gained by dissolving the pure colors are not given in bold and usually end in a number from 2 to 4, with 4 being the darkest shade.

Color Name	Color Value		
	R	G	B
black	0	0	0
blue	0	0	255
blue2	0	0	208
blue3	0	0	176
blue4	0	0	144
brown2	192	96	0
brown3	160	64	0
brown4	128	48	0
cyan	0	255	255
cyan2	0	208	208
cyan3	0	176	176
cyan4	0	144	144
dark Gray	64	64	64
gold	255	215	0
gray	128	128	128
green	0	255	0
green2	0	208	0
green3	0	176	0

Color Name	Color Value		
	R	G	B
green4	0	144	0
light Gray	192	192	192
light_blue	135	206	255
magenta	255	0	255
magenta2	208	0	208
magenta3	176	0	176
magenta4	144	0	144
orange	255	200	0
pink	255	175	175
pink2	255	192	192
pink3	255	160	160
pink4	255	128	128
red	255	0	0
red2	208	0	0
red3	176	0	0
red4	144	0	0
white	255	255	255
yellow	255	255	0

A Specific Methods

A.1 Move Methods

The following operations are available for the *move* command. The *default* method name if none is specified is **always** `translate`.

Type	Method	Description
Arc	`translate`	moves the object "as is"
Arc	`translateRadius`	changes the arc's radius
Arc	`translateAngle`	changes the arc's angle
Arc	`translateStartAngle`	changes the arc's start angle
ListElement	`translate`	moves the object "as is"
ListElement	`setTip`	sets the tip of the first pointer
ListElement	`translateWithFixed-Tip`	move the object without changing the location of the pointer(s)
ListElement	`translateWithFixed-Tip #i`	move the object without changing the pointer number `i` (substitute a value in `[1, nrPointers]` for `i`)
ListElement	`setTip #i`	Set the pointer `i` (with 1 being the first pointer)
ListElement	`translateWithFixed-Tips i j k...`	move the object without changing the location of the pointers given (separated by a single space)
ListElement	`setTips i j k...`	Set the pointers numbered `i, j, k` - specify as many as wanted, separated with a single space
Point	`translate`	move the point
Polyline or Polygon	`translate`	move the complete object "as is"
Polyline or Polygon	`translate #i`	translate only node `i` of the polyline or polygon. The number count starts at **1**
Polyline or Polygon	`translateNodes i j`	translate only the given nodes. The user may specify as many as wanted, separated by a single space. Note that the node count starts with **1**.
Polyline or Polygon	`translateWithFixed-Nodes i j k`	translate the objects while leaving the specified nodes at their current location. The user may specify as many as wanted, separated by a single space. Note that the node count starts with **1**.
Text	`translate`	move the component "as is".

A.2 Color Change Methods

The following operations are available for the *color* command. The *default* method name if none is specified is **always** `color`.

Type	Method	Description
Arc	`color`	Change the color of the arc outline
Arc	`textColor`	Change the color of the (optional) arc text
Arc	`fillColor`	Change the (optional) fill color of a closed and filled arc
Arc	`colors: color, textColor`	Change both color and (optional) text color
Arc	`colors: color, fillColor`	Change both color and (optional) fill color
Arc	`colors: textColor, fillColor`	Change both optional text and fill color
Arc	`colors: color, textColor, fillColor`	Change the arc's outline, text and fill color
Polyline / Polygon	`color`	Change the object's outline color
Polygon	`fillColor`	Change the object's fill color (only effective for filled polygons!)
Polygon	`colors: color, fillColor`	Change the object's outline and fill color, if filled

B Complete BNF

```
<animalScriptFile >:
  <fileHeader> { <command> \n}

<fileHeader >:
  %Animal <double> [<nat>*<nat>] [\n <titleInfo >]
  [\n <authorInfo >]

<titleInfo >:
  title "title as a string"

<authorInfo >:
  author "author name including EMail address"

<command>:
  <objectPrimitives> | <operations>
  | '{' | '}' | <languageSupport> | <extensionCommand>

<objectPrimitives >:
  <arcTypes> | <array> | <arrayMarker> | <codeTypes>
  | <listelement> | <point> | <polygonTypes>
  | <polyline> | <text> | <grid> | <graph>

<operations >:
  <arrayOp> | <clone> | <colorChangeTypes> | <delay>
  | <echo> | <label> | <link> | <location>
  | <merge> | <move> | <rotate> | <showTypes>
  | <swap> | <setText> | <setFont> | <variableSupport>
  | <assertion> | <gridOperations> | <graphOperations>

<arcTypes >:
  <arc> | <circleSeg> | <ellipse> | <circle>

<arc >:
  <arcName> "ID" <nodeDefinition>
    radius <nodeDefinition> [angle <int >]
    [starts <int >] [clockwise | counterclockwise]
    [color <color >] [depth <nat >]
    [<closedOptions> [<fillOptions >] | <arrowOptions >]
    [<displayOptions >]

<arcName >:
  arc | ellipseseg | ellipsesegment

<circleSeg >:
  <circleSegName> "ID" <nodeDefinition>
    radius <int> [angle <int >] [starts <int >]
    [clockwise | counterclockwise]
    [color <color >] [depth <nat >]
    [<closedOptions> [<fillOptions >] | <arrowOptions >]
    [<displayOptions>]

<circleSegName>:
  circleseg | circlesegment

<ellipse>:
  ellipse "ID" <nodeDefinition> radius <nodeDefinition>
    [color <color>] [depth <int>]
    [<fillOptions>] [<displayOptions>]

<circle>:
  circle "ID" <nodeDefinition> radius <int>
    [color <color>] [depth <nat>]
    [<fillOptions>] [<displayOptions>]

<array>:
  <arrayKey> "arrayID" [at] <nodeDefinition>
    [color <color>] [fillColor <color>]
    [elementColor <color>] [elemHighlight <color>]
    [cellHighlight <color>] [horizontal | vertical]
    length <nat+> { <intText> } [depth <nat>]
    [<timeOffset>] [cascaded [within <nat+> ticks | ms ]]
    [hidden] <font>

<arrayKey>:
  array | field

<arrayMarker>:
  <markKey> "indexID" on "arrayID" atIndex <nat>
    [label <intText>] [short | normal | long] [color <color>]
    [depth <nat>] <font> [<displayOptions>]

<markKey>:
  arrayMarker | arrayPointer | arrayIndex

<codeTypes>:
  <codeGroup> | <codeLine> | <codeElem>

<codeGroup>:
  codegroup "groupID" [at] <nodeDefinition>
    [color <color>] [highlightColor <color>]
    [contextColor <color>] <font>
    [depth <nat>] [<timeOffset>]

<codeLine>:
  addCodeLine "code" [name "ID"] to "codeGroupID"
    [indentation <nat>] [<timeOffset>]

<codeElem>:
  addCodeElem "code" [name "ID"] to "codeGroupID"
    [column <nat>] [indentation <nat>] [<timeOffset>]
```

```
<listelement>:
  listelement "ID" <nodeDefinition> [text <intText>]
    pointers <nat> [position <pointerPos>]
    [ { <ptrLocation> } ] [prev "prevID"] [next "nextID"]
    [color <color>] [boxFillColor <color>]
    [pointerAreaColor <color>]
    [pointerAreaFillColor <color>]
    [textColor <color>] [depth <nat>]
    [<displayOptions>]

<pointerPos>:
  top | left | right | bottom | none

<ptrLocation>:
  ptr<nat> <nodeDefinition> | ptr<nat> to "targetID"

<point>:
  point "pointID" <nodeDefinition> [color <color>]
     [depth <nat>] [<displayOptions>]

<polygonTypes>:
  <square> | <rect> | <triangle> | <polygon>

<square>:
  square "ID" <nodeDefinition> <nat+>
     [color <color>] [depth <nat>]
     [<fillOptions>] [<displayOptions>]

<rect>:
  <absoluteRectangle> | <relativeRectangle>

<absoluteRectangle>:
  rectangle "ID" <nodeDefinition> <nodeDefinition>
     [color <color>] [depth <nat>]
     [<fillOptions>] [<displayOptions>]

<relativeRectangle>:
  <relRectName> "ID" <nodeDefinition> <nodeDefinition>
     [color <color>] [depth <nat>]
     [<fillOptions>] [<displayOptions>]

<relRectName>:
  relrect | relrectangle

<triangle>:
  triangle "ID" <nodeDefinition> <nodeDefinition>
     <nodeDefinition> [color <color>] [depth depthVal]
     [<fillOptions>] [<displayOptions>]

<polygon>:
  polygon "ID" <nodeDefinition> <nodeDefinition>
     { <nodeDefinition> } [color <color>]
```

```
      [depth <nat>] [<fillOptions>]
      [<displayOptions>]

<polyline>:
  <lineTag> "lineID" <nodeDefinition> { <nodeDefinition> }
    [color <color>] [depth <nat>]
    [<arrowOptions>] [<displayOptions>]

<lineTag>:
  polyline | line

<text>:
  text "ID" <intText> [at] <nodeDefinition>
    [centered | right] [color <color>] [depth <nat>]
    <font> [boxed] [<displayOptions>]

<arrayOp>:
  <arrayPut> | <arraySwap> | <moveArrayMarker>
  | <highlightArrayCell> | <highlightArrayElem>

<arrayPut>:
  arrayPut "value" on "arrayID" position <nat>
    [<timing>]

<arraySwap>:
  arraySwap on "arrayName" position <nat> with <nat>
    [<timing>]

<moveArrayMarker>:
  <moveMarkerKeyword> "markerID" to
    [position <nat> | arrayEnd | outside] [<timing>]

<moveMarkerKeyword>:
  moveArrayIndex | moveArrayMarker | moveArrayPointer
  | moveIndex | moveMarker | movePointer
  | jumpArrayIndex | jumpArrayMarker | jumpArrayPointer
  | jumpIndex | jumpMarker | jumpPointer

<highlightArrayCell>:
  <hlACellKeyword> on "arrayID" <aHighlightRange>
  <timing>

<hlACellKeyword>:
  highlightArrayCell | unhighlightArrayCell

<aHighlightRange>:
  position <nat> | [from <nat>] [to <nat>]

<hilightArrayElem>:
  <hlAElemKeyword> on "arrayID" <aHighlightRange>
  <timing>
```

```
<hlAElemKeyword>:
  highlightArrayElem | unhighlightArrayElem

<clone>:
  clone "originalID" as "cloneID" [at] <nodeDefinition>
    [<displayOptions>]

<colorChangeTypes>:
  <colorChange> | <codeColorChange>

<colorChange>:
  color <oids> [type <colorType>] <color> [<timing>]

<colorType>:
  "color" | "fillColor" | "textColor" | "colorSetting"

<codeColorChange>:
  <codeColorType> on "baseCodeGroup" line <nat>
      [column <nat>] [context | region] [<timing>]

<codeColorType>:
  highlightCode | unhighlightCode

<delay>:
  delay <nat> [ms] | delay click on "elemID"

<echo>:
  echo location: <nodeDefinition>
  | echo <boundsKeyword>: { <oids> }
  | echo text: "text"
  | echo value: { "ID" }
  | echo ids: { <oids> }
  | echo visible
  | echo rule: "keyword"
  | echo unquotedText

<boundsKeyword>:
  boundingBox | bounds

<label>:
  label "labelEntry"

<link>:
  setLink "elemID" [link <nat>] to "targetID" [<timing>]
  | setLink "elemID" [link <nat>] <nodeDefinition> [<timing>]
  | clearLink "elemID" [link <nat>] [<timing>]

<location>:
  <locationKeyword> "locationID" [at] <nodeDefinition>
  | moveLocation "locationID" [to] <nodeDefinition>

<locationKeyword>:
  location | defineLocation | defLocation

<merge>:
  <mergeKeyWord> "targetID" { "ID" }

<mergeKeyWord>:
  group | merge | set | ungroup | remove

<move>:
  <moveVia> | <moveAlong> | <moveTo>

<moveKeyword>:
  move | translate

<moveVia>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
   via "oid" [<timing>]

<moveAlong>:
  <moveAlongPolyline> | <moveAlongArc>

<moveAlongPolyline>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
      along <lineTag> <nodeDefinition> { <nodeDefinition> }
      [<timing>]

<moveAlongArc>:
  <moveAlongArcType> | <moveAlongCircleType>

<moveAlongArcType>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
     along <arcType> <nodeDefinition> <int> <int>
      <int> <int> [<timing>]

<moveAlongCircleType>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
     along <circleType> <nodeDefinition> <int> <int>
     <int> [<timing>]

<moveTo>:
  <moveKeyword> <oids> [<corner>] [<methodSpec>]
   to <nodeDefinition> [<timing>]

<methodSpec>:
  type "typeName"

<corner>:
  corner <direction>

<arcType>:
  <arcName> | ellipse
```

```
<circleType >:
  circle | <circleSegName>

<rotate >:
  rotate <oids> around "id" [degrees <int >] [<timing >]
 | rotate <oids> center <nodeDefinition> [degrees <int >]
   [<timing >]

<showTypes >:
  <simpleShow> | <codeHide> | <selectiveHide >

<simpleShow >:
  <showMode> <oids > [<timing >]

<codeHide >:
  hideCode "codeID" [<timeOffset >]

<selectiveHide >:
  hideAll [<timing >]
  | hideAllBut { <oids> } [<timing >]

<showMode >:
  show | hide

<swap >:
  <swapKeyword> "oid1" "oid2"

<swapKeyword >:
  swap | exchange

<languageSupport >:
  supports { "languageKey" }
  [\n <resourceKey> "fileNameWithoutExtension"]

<resourceKey >:
  resource | bundle | resourceBundle

<intText >:
  "text" | key: "textResourceKey" | ( { key: "text" } )

<color >:
  black | blue | blue2 | blue3 | blue4 | brown2
  | brown3 | brown4 | cyan | cyan2 | cyan3 | cyan4
  | dark Gray | gold | green | green2 | green3
  | green4 | light Gray| light_blue | magenta
  | magenta2 | magenta3 | magenta4 | orange | pink
  | pink2 | pink3 | pink4 | red | red2 | red3
  | red4 | white | yellow | (<nat >, <nat >, <nat >)

<displayOptions >:
  hidden | <timeOffset>
```

```
<timeOffset>:
   after <nat> [ticks | ms]

<timing>:
  [<timeOffset>] [within <nat> [ticks | ms]]

<fontName>:
   Serif | SansSerif | Monospaced

<nodeDefinition>:
  (<int>, <int>) | <offset> | move (<int>, <int>)

<offset>:
   offset (<int>, <int>) from ”referenceID” node <nat+>
|  offset (<int>, <int>) from ”referenceID” <direction>
|  offset (<int>, <int>) from ”locationID”
|  offset (<int>, <int>) from ”referenceID”
     baseline [start | end]

<direction>:
  NW | N | NE | W | C | E | SW | S | SE
  | Northwest | North | Northeast | West
  | Middle | Center | East | Southwest
  | South | Southeast

<arrowOptions>:
  [fwArrow] [bwArrow]

<closedOptions>:
  closed

<fillOptions>:
 filled [fillColor <color>]

<font>:
  [font <fontNames>] [size fontSize] [bold] [italic]

<oids>:
  { ”targetOID” }

<nat>:
  0 | 1 | ... |

<int>:
  <nat> | –<nat> | ( <double> <operator> <double>)
  | ”objectID” <objectPosition>

<operator>:
  + | − | ∗ | /

<objectPosition>:
  x | y | width | height
```

<extensionCommand>:

<**setText**>:
  setText [**of**] "oid" [<methodSpec>] **to** <intText> <timing>

<**setFont**>:
  setFont [**of**] "oid" [**to**] <**font**> <timing>

<variableSupport>:
  variable "id" [**type** <varType>]
  | **assign** "id" = <**int**>
  | <assertion>

<varType>:
  int | **double** | **String**

<assertion>:
  <assertKeyword> <assertionPart>

<assertKeyword>:
  assert | **check**

<assertionPart>:
  variable <comparator> <**int**>
  | <assertionPart> <boolOperator> <assertionPart>

<comparator>:
  == | != | < | <= | >= | >

<boolOperator>:
  && | ||

<**grid**>:
  grid "id" <nodeDefinition> **lines** <nat+> **columns** <nat+>
  [**style** <gridStyle>]
  [**cellWidth** <nat+>] [**maxCellWidth** <nat+>]
  [**cellHeight** <nat+>] [**maxCellHeight** <nat+>] [**fixedCellSize**]
  [**color** <**color**>] [**textColor** <**color**>] [**borderColor** <**color**>
  [**fillColor** <**color**>] [**highlightTextColor** <**color**>]
  [**highlightFillColor** <**color**>] [**highlightBorderColor** <**color**>]
  [<**font**>] [**align** <gridAlign>] <**depth**> <timeOffset>

<gridStyle>:
  plain | **matrix** | **table**

<gridAlign>:
  left | **center** | **right**

<gridOperations>:
  <**setGridValue**> | <**setGridColor**>
  | <**setGridFont**> | <**swapGridValues**>

```
  | <highlightGridComponent> | <alignGridValue>

<cellIdentifier>:
  "id[<nat>][<nat>]"

<setGridValue>:
  setGridValue <cellIdentifier> <intText> [refresh] <timing>

<setGridColor>:
  setGridColor <cellIdentifier>  [color <color>]
  [textColor <color>] [fillColor <color>]
  [highlightTextColor <color>] [highlightBackColor <color>]

<setGridFont>:
  setGridFont <cellIdentifier>  <font> [refresh]
  [within <nat> [ticks | ms]]

<swapGridValues>:
  swapGridValues <cellIdentifier> and
    <cellIdentifier> [refresh] <timing>

<highlightGridComponent>:
  highlightGridCell <cellIdentifier> <timing>
  | highlightGridElem <cellIdentifier> <timing>
  | unhighlightGridCell <cellIdentifier> <timing>
  | unhighlightGridElem <cellIdentifier> <timing>

<alignGridValue>:
  alignGridValue <cellIdentifier> <gridAlign> [refresh] <timing>

<graph>:
 graph "ID" size <nat+> [color <color>] [bgColor <color>]
  [outlineColor <color>] [highlightColor <color>]
  [elemHighlightColor <color>] [nodeFontColor <color>]
  [edgeFontColor <color>] [directed] [weighted]
  nodes "{" [<font>] { "value" [at] <nodeDefinition> [","] } "}"
  edges "{" [<font> { "(" <nat+> "," <nat+> ["," <intText>] ")" [","]"}"
  [origin <nodeDefinition>] [showIndices] [depth <nat>] <timeOffset>

<graphOperations>:
  <edgeTransformation> | <changeWeight> | <nodeTransformation>

<edgeTransformation>:
  highlightEdge on "ID" [<methodSpec>] { "(" <int> "," <int> ")" }
  | unhighlightEdge on "ID" [<methodSpec>] { "(" <int> "," <int> ")" }

<changeWeight>:
  setEdgeWeight [of] "ID" edge "(" <int> "," <int> ")" to <int>

<nodeTransformation>:
  highlightNode on "ID" [<methodSpec>] nodes { <int> } <timing>
  | unhighlightNode on "ID" [<methodSpec>] nodes { <int> } <timing>
```

Listings

Index