The AGAT manual

Olivier Arsac

Stéphane Dalmas

Marc Gaëtano

Stéphane Lavirotte

1 An overview of *agat*

agat is a tool for algorithm animation. It enables the programmer to display graphical animated views of the data that are manipulated by its program in "real-time" (i.e while its program is running). Compared with other tools (like *XTango* or *Zeus*), *agat* cannot perform very sophisticated animation but is much more easy to use. To animate a program, only a few lines of code need to be added to the original program. This extra code says that the values of certain expressions are sent on some streams (that are just identified by a name). The animation itself is described in a separate file in a little specialized programming language. This language describes how the values on the different streams are combined, transformed and displayed.

When the program is run under the control of the animation server, it sends its data...

A program, the animation server, is used to run the program with the animation. The program sends data to the server that executes the *agat* program to process these data.

Here is a simple picture which can help you to understand the mecanism of *agat*.

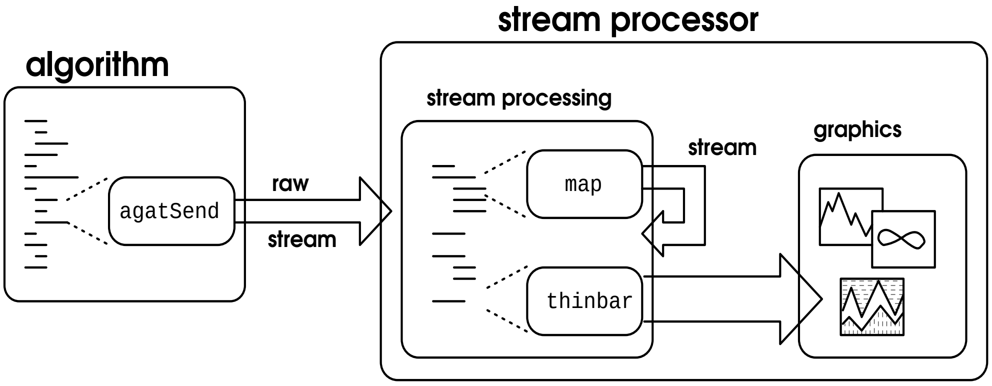

Figure 1: The *agat architecture*

2 Preparing a program for animation

To prepare a program for animation, the programmer uses the functions in the *agat* library to send the interesting data manipulated by the program on *streams*. He is free to use as many streams as he wants and to name them as suits him (a stream is just identified by a character string).

Here is a very simple example. The aim is to study the properties of a random number generator. The program loops, repeatedly sending a new random number on a stream that we choose to name `i`. This is done by calling the `agatSendDouble` function (as the random numbers are of type `double`). Here is the instrumented code :

```
/* aleat.c, a pseudo random number generator, instrumented for agat */
#include <stdio.h>
#include <math.h>
#include "libAGAT.h"

double          drand48(void);

/* modify this function to change generators */
double          myRand(void)
```

```
{
  return drand48();
}

int                main(int argc, char **argv)
{
  int                i, j, nbn;
  double             v;

  nbn = atoi(argv[1]);
  for (i = 1; i <= nbn; i++) {
    v = myRand();
    agatSendDouble("i", v);
  }
}
```

The user needs then to link its program with the *agat* library, where the function `agatSendDouble` is defined (note that we had also included the `libASAP.h` header file where prototypes of the *agat* functions can be found). It can be done with a command line like `gcc -o aleat aleat.c -lAGAT`.

Note that including calls to *agat* library functions in your program does not modify its behavior (at least if it is a correct program). It can be run without the control of the *agat* server and the results will be similar to what would be obtained without the calls to the *agat* library.

Some guidelines on how to instrument your code (to avoid recompilation) : you should just send raw data (processing can be done at the *agat* level), don't hesitate to use several streams (two streams can easily be combined into one in *agat*). Don't hesitate to send more data than actually needed (ignoring a stream is very easy : you just don't mention it in the *agat* program).

3 Animating a program

Then, the programmer has to write a file in the *agat* language to specify how the streams should be combined, what exactly should be displayed and how...

```
-- file animAleat.agt

include "stat.agt","misc.agt"; -- include some predefined agat functions

let ci=account(i);      -- count the number of values emitted on stream "i"
let avi=acaverage(i); -- average value on "i"
let inci=increase(i); -- increment between two values

-- we "delay" the streams to be able to have a value and the preceeding one together
let di=delay(i);
let dci=delay(ci);
let davi=delay(avi);
let dinci=delay(inci);

-- and then we visualize all the streams...

-- an histogram view of "i"
thinbar(i);
-- various plots (points in the plane)
plot(ci, i);
plot(ci, avi);
plot(di, i);
plot(davi, avi);
plot(dinci, inci);
```

Here, from the raw stream `i` of the random numbers, we choose to compute (construct) three other streams. `ci` is a stream counting the number of random values produced (it will thus contain 1, 2, 3, 4...). The stream `avi` will contain the successive averages of the generated numbers and `inci` the differences between two successive random numbers.

As we are interested in comparing the current value on these streams with the preceding one, we "delay" them, creating three other streams, `di`, `dinci` and `davi`. The values on the stream `di` are the values on `i` with a delay of one value. There is a natural notion of time in *agat*. A tick of the clock corresponds with the arrival of a value emitted by the program. The streams are therefore naturally synchronized through this clock. This means that, if `i` is 0.2 at time t_0, 0.3 at t_1, 0.1 at t_2 and so on, `di` will be 0.2 at time t_1, 0.3 at t_2... (and will contain no value at t_0).

The functions like `account`, and `delay` are written in *agat* and defined in library files (included with the `include` directive at the top of the *agat* program).

For the animation itself, we choose to plot an histogram of the random values produced (the `thinbar` primitive that looks a lot like the `xload` application) and several simple plots. `plot(ci, i)` produces a point for each pair of values arriving on the streams `ci` and `i` (a point for each pair of (`ci`, `i`) values). All these graphical operators dynamically choose the right scale and ranges and can display axes with smart tick marks. There also exists operators that link successive points with lines, display boxes of varying height and width and show the ratio of the values on different streams.

The animation is then obtained by running the `aleat` program under the control of the `agat` animation server. The simple command line `agat -f animSimple.agt aleat` do the job and six windows then appears on the user screen (as six graphical operators are called in the *agat* program), showing the evolution of the six streams. (Figure 2)

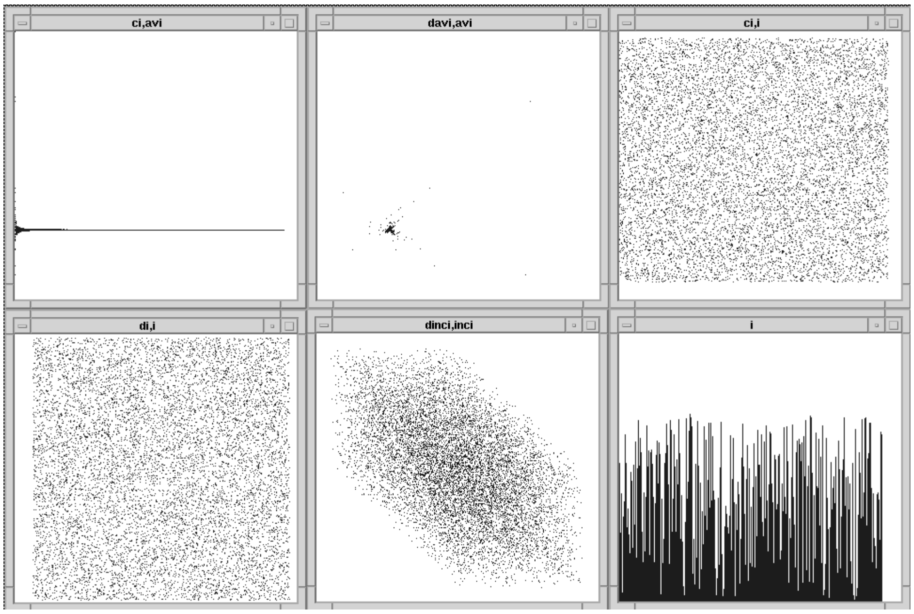

Figure 2: *agat* execution: agat -f animSimple.agt aleat. *drand48()*

4 Running an animation : the agat program

In the sequel *agat_file* is the name of a file containing the *agat* program describing the animation to be performed, *dump_file* a file containing (or that will contain) the recording of an *agat* animation, *dir* specifies a directory where the files included by *agat_file* can be found (the default is to use the current working directory), *cmd* is the name of the program to animate and $arg_1 \ldots arg_n$ will be given as command line arguments to *cmd*. The [...] notation indicates an optional part.

There are basically three ways to use the `agat` program :

- to run the user program with an animation
- to replay a previously recorded animation
- to run the user program with no animation but recording the emitted values

Here are the possible corresponding options :

`agat -f` *agat_file* [`-w`] [`-d` *dump_file*] [`-I` *dir*] *cmd* $arg_1 \ldots arg_n$

cmd is executed (with $arg_1 \ldots arg_n$ as arguments), and animation described by *agat_file* is performed. The emitted values are recorded in *dump_file*. Several `-I` can be given (specifying a search path). Warning messages are not printed if the `-w` flag is present.

`agat -f` *agat_file* [`-w`] `-r` dump_file [`-I` *dir*] [`-s` *speed*] [`-m`]

An animation described by *agat_file* is performed from the data found in *dump_file*. The animation is played at speed *speed*. This should be a positive integer, specifying a percentage of "real-time". So, 50 means "run at half speed" and 200 "run at double speed". If the `-m` argument is present, the animation is replayed as fast as possible. Warning messages are not printed if the `-w` flag is present.

`agat -t` *dump_file cmd* $arg_1 \ldots arg_n$

cmd is run with arguments $arg_1 \ldots arg_n$. The emitted values are recorded in *dump_file*. No animation is performed.

If an animation has been requested, when the program *cmd* terminates, the `agat` program waits for an interrupt (signal SIGTERM usually generated by typing CTRL-C or DEL) to terminate so that the windows are kept on the screen for as long as the user wishes. It is also possible to terminate by typing a `q` character in one of the animation windows.

There is others key and mouse commands (15.1 and 15.2).

5 The structure of an *agat* program

An *agat* program is a sequence of declarations. Declarations can be classified into five categories:

- *library* (`include`) declarations (5.1, page 5) to reuse the declarations contained in some other *agat* files.
- *constant* declarations (5.2, page 5). A constant declaration associates an identifier to a constant of a basic type (6, page 6).
- *stream* declarations (5.3, page 5). They build up new streams from existing ones, using the `map` construct.
- *function* declarations (5.4, page 5). They define functions on streams (functions acting on values cannot be defined in the present version of *agat*).
- *postprocessing* declarations (5.5, page 5). The effect of a postprocessing declaration is (normally) to create a graphical object that will show an animated view of the data on one or more streams.

5.1 Libraries and `include`

An *agat* program can include declarations defined in another *agat* program with the `include` directive. The general form of this declaration is :

$$\texttt{include}\ \mathit{file}_1, \ldots \mathit{file}_n$$

where file_i is a file name (pathname) between two double quote characters (" characters).
Example:

```
include "private.agt", "macros.agt", "/usr/local/lib/agat/include/stat.agt";
```

The effect of an include declaration is the one of the inclusion of the contents of the files in the order where they are listed.

The `-I` option of the `agat` command can be used to specify a "search path" for include files (if they are given as relative pathnames, of course). For example, if `-I /usr/local/lib/agat` and `-I $HOME/lib/agat` are given as command line arguments, a relative file name (not beginning with `/`) is successively searched for in `/usr/local/lib/agat`, `$HOME/lib/agat` and the current working directory.

5.2 Constant declarations

A constant declaration associates a value of a basic type to an identifier. Its general form is:

$$\texttt{let}\ x\ \texttt{=}\ \mathit{expr}$$

Examples:

```
let Pi = 3.14159265358979323846;
let halfPi = Pi/2;
let Pi2 = 8 * atan(1);
```

5.3 Stream declarations

A stream declaration creates a new stream and gives it a name. This new stream is built up from some other streams thanks to the `map` construct.
Examples:

```
let double = map (x -> 2 * x) s;
let sum = map (x,y -> x + y) s1, s2;
```

Here, a new stream, `double` is built from `s` by multiplying by 2 each value on `s`. Another stream `sum` is obtained by summing the values on the two streams `s1` and `s2`.

Note that a stream declaration can be syntactically indistinguishable from a constant declaration because the `map` can be hidden by a function declaration.

5.4 Function declarations

The syntactic form of a function declaration is:

$$\texttt{let}\ f\,(x_1, \ldots x_n)\ \texttt{=}\ \mathit{expr}$$

where f is the name of the function being defined, x_1 to x_n its formal parameters and *expr* its body.
Example:

```
let filter_pos (s) = map (x -> if x >= 0 then x end) s;
let pos_st = filter_pos (st);
```

Here `filter_pos` is a function that, when given a stream (like `st`) will produce another stream that contains only its positive values.

Formal parameters are restricted to denote streams for the moment so that something like `let cube(x) = x * x * x;` is invalid.

5.5 Postprocessing declarations

These are the ultimate declarations of an *agat* program as they are the only one to have an observable effect. The general form of a postprocessing declaration is:

$$\mathit{operator}\,(x_1, \ldots x_n)$$

where *operator* is chosen among the list of available postprocessing operators (12, page 10) and x_i are stream names.

Most operators eventually create a window on the screen to show an animated view of the values arriving on some streams. But there exists "non graphical" operators like `print` (mostly used for debugging purpose).

6 Streams and their contents

Streams contain values. The values manipulated inside an *agat* program are of two basic types:
- integer
- floating point

Integer values can represent integers in the range $[-2^{31}, 2^{31}-1]$. Floating points values are basically IEEE double precision floating point numbers.

7 Building new streams

Existing streams can be combined to form new streams with the `map` construct. The simplest form of this construct is

$$\texttt{map(}\ c_1\,\texttt{|}\ \ldots c_n\texttt{)}\ s_1, \ldots s_n$$

A more complicated form enables the use of local variables to accumulate values

$$\texttt{map \{}x_1\ [\ \texttt{:=}\ expr_1], \ldots x_n\ [\ \texttt{:=}\ expr_n]\texttt{\}}\ \texttt{(}c_1\,\texttt{|}\ \ldots c_n\ \texttt{)}\ s_1, \ldots s_n$$

where s_i are stream names, $expr_i$ is an optional expression that should evaluate to a constant of a basic type that is used to initialize x_i and each c_i is a clause made of a pattern and an expression or a sequence of *actions*.

Example:

```
map {r := 0} (a,b -> r := r + a + b) f, g;
```

A `map` construct produces a new stream from some existing streams. This stream is built by combining the values of the input streams according to patterns describing the arrival of the data. The new values are produced as the result of evaluating expressions.

8 Expressions and Actions

We distinguish between *expressions* and *actions* :
- the evaluation of an expression always produces a value
- the evaluation of an action does not always produce a value. Usually it also does some side-effect (like assignment to local variables).

8.1 Basic expressions

Basic expressions are made up from identifiers, literal constants (floating-point and integer), arithmetic operators and predefined functions. See section 10.1, page 8 for a complete description of the available operations.

8.2 The conditional form

agat possesses only one conditional form, a very classical `if` construct. Its syntax is :

`if` *expr* `then` $actions_1$ `else` $actions_2$ `end`

or

`if` *expr* `then` *actions* `end`

where $actions_i$ and *actions* are sequences of actions.

The value of this `if` action is the value produced by $actions_1$ if the evaluation of the boolean expression *expr* returns *true* and the value produced by $actions_2$ if it is false. In any case, only one sequence of actions is evaluated. In the simplified form (without the `else` part) if *expr* is false, no value is produced. In fact, `if` *expr* `then` *act* `end` is equivalent to `if` *expr* `then` *act* `else null end`

8.3 Boolean expressions

Boolean expressions are used inside the `if` conditional construct. There is no boolean value that can be manipulated (assigned or put in a stream) in *agat*.

The available comparison operators are : `>`, `<`, `<=`, `>=`, `==` with their obvious meanings. They are all able to compare mixed floating-point and integer values.

9 Patterns

A pattern describes what is present on a stream and can give a name to that value (bind an identifier). Five different conditions can be tested :

1. no value has ever been present on the stream
2. no value at the moment
3. one value is present
4. at least one value was present on the stream but none is present now
5. the value present is the first one of the stream

on the stream:	*the pattern that matches:*
no value ever present	`#`
no new value	`*`
a new value	_ *or* *x*
at least one value arrived	`+`
the first value	_^ *or* *x*^

Patterns are combined to describe the state of a set of streams.

The ^ pattern can be used for initialization, as in the following example, that produces the successive maximum of the data on a stream :

```
let acmax(i) =
        map  {m}
             (x^ -> m := x
             |x  -> m := max(m,x)) i;
```

This avoids the need to initialize the maximum with the "minimum" possible value (with the problem of mixing floating-point and integers).

Going on with our maximum computations, we will illustrate the use of * and # patterns with the following functions that computes the maximum of the values occuring on two streams.

```
let acmax2(i,j) =
        map  {m}
             (a^,#  -> m:=a
             |# ,b^ -> m:=b
             |a^,b^ -> m:=max(a,b)
             |a ,*  -> m:=max(a,m)
             |* ,b  -> m:=max(b,m)
             |a ,b  -> m:=max(max(a,b),m)
             )i ,j;
```

The apparent complexity comes from the fact that we want the maximum to be updated each time a value arrives on one of the streams.

10 Operations on values

Partial operations (operations not defined everywhere, like `sqrt` of `log`) are treated in a special way. If the argument is out of the domain (like a negative number for `sqrt`) an error is written on the "standard error" file descriptor and an "almost random" (implementation dependent) value is produced as the result. This behavior was chosen for two reasons :

- we don't want to stop an animation unnecessarily
- some graphical operators (like the plotting ones) will be confused if no value are produced when they expect one (the y coordinate for a given x, for example).

Note that the `NaN` value is correctly handled. (C.f : Norme IEEE) The following convention is used throughout the description of the operations : an argument named n is restricted to be an integer. An argument named x or y can be a floating point or an integer (except where explicitly stated).

10.1 Basic operations

x + y
: the sum of x and y.

x * y
: the product of x and y.

abs(x)
: returns the absolute value of x.

sgn(x)
: returns the "sign" of x i.e. 0 if x is equal to 0, -1 if x is negative and 1 if x is positive.

max(x_1, x_2), max(x_1, x_2, x_3) , max(x_1, x_2, x_3, x_4)
: returns the maximum of its argument (each x_i can be integer or floating point). `max` can accept from 2 to 4 arguments.

min(x_1, x_2), min(x_1, x_2, x_3) , min(x_1, x_2, x_3, x_4)
: returns the minimum of its arguments (each x_i can be integer or floating point). `min` can accept from 2 to 4 arguments.

ceil(x)
: returns the smallest integer which is not less than x.

`floor(`x`)`
: returns the largest integer which is not greater than x.

`fmod(`x, y`)`
: returns the floating point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $\mid f \mid < \mid y \mid$.

`sqrt(`x`)`
: returns the square root of x.

10.2 Transcendental operations

The most common transcendental functions are available in *agat*. Note that the inverse of the hyperbolic trigonometric functions are not available. All trigonometric functions assume that their argument is measured in radians.

`acos(`x`)`
: returns the arc-cosine of x (`acos` is thus the inverse of the `cos` function).

`asin(`x`)`
: returns the arc-sine of x (`asin` is thus the inverse of the `sin` function).

`atan(`x`)`
: returns the arc-tangent of x (`atan` is thus the inverse of the `tan` function).

`atan2(`x, y`)`
: is just $\operatorname{atan}(y/x)$.

`cos(`x`)`
: returns the cosine of x.

`cosh(`x`)`
: returns the hyperbolic cosine of x.

`exp(`x`)`
: is the exponential function.

`log(`x`)`
: returns the natural logarithm (base e) of x.

`log10(`x`)`
: returns the common logarithm (base 10) of x.

`pow(`x, y`)`
: returns x to the power y. This is could be computed as $e^{y \ln x}$.

`sin(`x`)`
: returns the sine of x

`sinh(`x`)`
: returns the hyperbolic sine of x

`tan(`x`)`
: returns the tangent of x.

`tanh(`x`)`
: returns the hyperbolic tangent of x.

10.3 Miscellaneous operations

The following three functions operate on integers. Implementation dependent results are produced if they are applied to floating-point values.

`nbdigit(`n`)`
: the number of digits of n written in decimal.

`lmostdigit(`n`)`
: the lefmost digit (most significant) of n (in decimal).

`rmostdigit(`n`)`
: the rightmost digit (least significant) of n (in decimal).

`odd(`n`)`
: returns 1 if n is odd, 0 otherwise.

11 Scoping in *agat*

An identifier can denote either :

1. a stream
2. a basic value (integer, floating point)
3. a function
4. a basic operation
5. a register (or the content of)

There is overloading. It is possible to use `cos` as the basic operation and a function acting on a stream and giving the cosine of its value as

```
let cos f = map (x -> cos(x)) f;
```

12 Postprocessing and graphic objects

A graphic object shows an animated view of the values arriving on one or several streams. Graphic objects are created by postprocessing operators.

Some graphic object show the values arriving on the streams others only show the relative proportions between the values arriving on the streams.

Some operators can block. This means that they wait for new values to appear on their stream arguments to continue the animation.

All operators whose names contain mixed lower case and upper case characters have synonyms with only lower case characters.

12.1 Not graphic operators

`print(`s`)`
: causes the values arriving on the stream s to be printed on standard output.

`prints(`$s_1, \ldots s_n$`)`
: causes the values arriving on the streams $s_1 \ldots s_n$ to be printed on standard output. Be more precise please...

`sleep(`s`)`
: causes the animation server to sleep 1 second each time a value is available on the stream s. It is the *agat* server that sleeps and it cannot be awaken. Use of this operator is strongly discouraged (except for particular debugging purposes). Storing the animation in a file and replying it is a much better way...

12.2 2D graphic operators

12.2.1 Histogram-like operators

`thinBar(`$s_1, \ldots s_n$`)`
: causes the values arriving on $s_1 \ldots s_n$ to be displayed...

`thinBarAxe(`$s_1, \ldots s_n$`)`
: similar to `thinBar` but graded axis are printed.

`plotBar(`$s_1, \ldots s_n$`)`
: similar to `thinBar` but only the tops of the bars are printed.

`plotBarAxe(`$s_1, \ldots s_n$`)`
: similar to `plotBar` but with graded axis.

`ratThinBar(`$s_1, \ldots s_n$`)`
: shows the .... This operator can block.

`histo(`i, v`)`
: draws a rectangle at i on the abscissa and with a v height.

`histoAxe(`i, v`)`
: similar to `histoAxe` with graded axis.

12.2.2 Plotting operators

`plot(`x, y`)`
: draws the point of coordinate (x, y) on the plane for each pair or values on the two streams. This operator can block.

`plotAxe(`x, y`)`
: similar to `plot` but the graded axis are printed.

`plotLine(`x, y`)`
: similar to `plot` but two consecutive points are linked with a straight line.

`plotAxeLine(`x, y`)`
: similar to `plotLine` with graded axis.

`plotRestart(`x, y, r`)`
: similar to `plot`. The third parameter is a boolean value to specifie that you want forget all the old values and to restart with a clear draw.

`plotAxeRestart(`x, y, r`)`
: similar to `plotRestart` with graded axis.

`plotLineRestart(`x, y, r`)`
: similar to `plotLine`. The thrird parameter is a boolean value to specifie that you want to forget all the old values and to restart with a clear draw.

`plotAxeLineRestart(`x, y, r`)`
: similar to `plotLineRestart` with graded axis.

`plotLineInt(`x, y, i`)`
: similar to `plot`. With the third parameter (boolean value))you specifie you want to link the new point with the last one.

`plotAxeLineInt(`x, y, i`)`
: similar to `plotLineInt` with graded axis.

12.2.3 Box operators

The following operators are very similar to the plotting operators but instead of points they draw rectangles.

`box(`ulx, uly, lrx, lry`)`
: draws the rectangle whose upper left and lower right corners are given by (ulx, uly) and (lrx, lry) respectively. This operator can block.

`boxFill(`ulx, uly, lrx, lry`)`
: draws the rectangle whose upper left and lower right corners are given by (ulx, uly) and (lrx, lry) respectively. The rectangle is filled with a color. This operator can block.

`boxAxeFill(`ulx, uly, lrx, lry`)`
: draws the rectangle whose upper left and lower right corners are given by (ulx, uly) and (lrx, lry) respectively, filled with a color. The axes are plotted. This operator can block.

`boxRestart(`ulx, uly, lrx, lry, r`)`
: similar to `box`. The third parameter is a boolean value to specifie that you want forget all the old values and to restart with a clear draw.

`boxAxeRestart(`ulx, uly, lrx, lry, r`)`
: similar to `boxRestart` with graded axis.

`boxFillRestart(`ulx, uly, lrx, lry, r`)`
: similar to `boxFill`. The third parameter is a boolean value to specifie that you want forget all the old values and to restart with a clear draw.

`boxAxeFillRestart(`ulx, uly, lrx, lry, r`)`
: similar to `boxFillRestart` with graded axis.

12.2.4 Matrix operators

`matrix(`i, j`)`
: draws a rectangle at the i, j position of a matrix. i correspond to the x axis and j to the y axis.

`matrixGrid(`i, j`)`
: similar to `matrix` but the displayed rectangle has a border.

`matrixAxe(`i, j`)`
: similar to `matrix` with graded axis.

`matrixAxeGrid(`i, j`)`
: similar to `matrixGrid` with graded axis.

`matrixCol(`i, j, c`)`
: similar to `matrix`. c is the color of the rectangle (the value must be in 0..14).

`matrixColGrid(`i, j, c`)`
: similar to `matrixCol` but the displayed rectangle has a border.

`matrixColAxe(`i, j, c`)`
: similar to `matrixCol` with graded axis.

`matrixColAxeGrid(`i, j, c`)`
: similar to `matrixColGrid` with graded axis.

`matrixVal(`i, j, v`)`
: similar to `matrix`. The height of the rectangle is proportional to v.

`matrixValGrid(`i, j, v`)`
: similar to `matrixVal` but the displayed rectangle has a border.

`matrixValAxe(`i, j, v`)`
: similar to `matrixVal` with graded axis.

`matrixValAxeGrid(`i, j, v`)`
: similar to `matrixGrid` with graded axis.

12.3 3D graphic operators

12.3.1 Plotting operators

`plot3D(`x, y, z`)`
: draws the point of coordinate (x, y, z) for each group of values on the three streams. This operator can block.

`plot3DLine`(x, y, z)
: similar to `plot` but consecutives points are linked with a straight line.

`plot3DLineInt`(x, y, z, r)
: similar to `plot`. With the fourth parameter (boolean value) you specifie you want to link the new point with the last one.

13 Lexical issues

13.1 Literal constants

13.2 Identifiers

Identifiers are defined by the following regular expression :

```
[a-zA-Z][_a-zA-Z0-9]*
```

13.3 Keywords

The keywords of the *agat* language are :

```
map let true false if then else end endif null include
```

13.4 Comments

Three forms of comments exist in the *agat* language

- ada-like (the rest of the line from `--`),
- C-like (between `/*` and `*/` and
- C++-like comments (the rest of the line from `//`).

C-like comments can be nested, but keep in mind that this is may lead to conclusions.

14 A grammar of *agat*

definitions →
    definition   definitions
  | definition

definition →
    defconst   ;
  | defmacro   ;
  | defstream   ;
  | defpostproc   ;
  | include   ;
  | ;

include →
    include files

files →
    file , files
    | file

file →
    " string "

defconst →
    let ident := expression

defmacro →
    let ident (names) = construct

defstream →
    let ident = construct

defpostproc →
    ident (names)

construct →
    map

map →
    map registers (clauses) names
    | **map** (clauses) names

registers →
    { regdecls }

regdecls →
    regdecl , regdecls
    | regdecl

regdecl →
    ident
    | ident := expression

names →
    name , names
    | name

name →
    ident

clauses →
    clause | clauses
    | clause

clause →
    patterns -> actions

patterns →
pattern , patterns
| pattern

pattern →
ident state
| state ident
| ident
| state

state →
#
| ^
| _
| *
| +

actions →
action , actions
| action

action →
expression
| assignation
| ifthenelse
| **null**

ifthenelse →
if expression **then** actions **end**
| **if** expression **then** actions **else** actions **end**

assignation →
ident := expression

expression →
sexpression
| sexpression == sexpression
| sexpression <> sexpression
| sexpression > sexpression
| sexpression < sexpression
| sexpression >= sexpression
| sexpression <= sexpression

sexpression →
factors
| sexpression + factors
| sexpression - factors

factors →
factor
| factors * factor
| factors / factor

factor →
fact

| + fact
| - fact

fact →
number
| ident
| **true**
| **false**
| funcall
| (expression)

funcall →
ident ( )
| ident (params)

params →
params , expression
| expression

15 *agat* commands

15.1 The key commands

- **b,B**
  To have a *"big"* window size. See 16.
- **k,K**
  Kill a zoom window.
- **n,N**
  To have a *"normal"* window size. See 16.
- **p,P**
  Create a PostScript file in the current directory. The name of the created file is the name of the window in which you press **p,P**.
- **q,Q**
  Quit Agat.
- **r,R**
  Force to refresh a window. (Specialy for zoom windows)
- **s,S**
  Stop the animation. Press any key to restart.
- **t,T**
  To have a *"tiny"* window size. See 16.

15.2 The mouse commands

- **Button1**
  Display the coordinates of the clicked point. Stopped when button released.
- **Button2**
  Move the view. Only for 3D operators.

– **Button3**

It's the zoom function. When you click on this button, you can select the area you want to zoom in. Release the button to activate the zoom. You can zoom in a zoom window. But there is no automatic refresh in a zoom window. To have a refresh, use the **r,R** key.

16 *agat* resources

With this new version of *agat*, you can configure the resources if you don't like the "look" of the application.

Here is the list of the resources which can be redefined :

– **background** (Class Background)

Sets the background color of a window.

– **colorxx** (Class Colorxx)

Sets the color number xx for drawing.

– **font** (Class Font)

Sets the font to draw the labels and axes.

– **foreground** (Class Foreground)

Sets the color of the text on the axes.

– **organized** (Class Organized)

Opens the windows with a "beautiful" organization.

– **origx** ()Class Origx)

Specifies the place of the first window for the X coordinate. Used only when organized is enable.

– **origy** (Class Origy)

Specifies the place of the first window for the Y coordinate. Used only when organized is enable.

– **pdx** (Class Pdx)

Specifies the space between two windows for the X coordinate. Used only when organized is enable.

– **pdy** (Class Pdy)

Specifies the space between two windows for the Y coordinate. Used only when organized is enable.

– **iconic** (Class Iconic)

Opens the window in an iconic form at the beginning of the animation.

– **xm** (Class Xm)

Specifies the margin in the window for the X coordinate.

– **xsz** (Class Xsz)

Specifies the size of the window for the X coordinate.

– **ym** (Class Ym)

Specifies the margin in the window for the X coordinate.

– **ysz** (Class Ysz)

Specifies the size of the window for the Y coordinate.

– **bigx** (Class Bigx)

Specifie the X size of a *"big"* window. See 15.1.

– **bigy** (Class Bigy)

Specifie the Y size of a *"big"* window. See 15.1.

- **tinyx** (Class Tinyx)

  Specifie the X size of a *"tiny"* window. See 15.1.

- **tinyy** (Class Tinyy)

  Specifie the Y size of a *"tiny"* window. See 15.1.

You can specify resources for:

- all the application
  `agat*<resource>: <val>`
- a class of graphic operator
  `agat.<op_name>*<resource>: <val>`
- a specified window
  `agat.<op_name>.<window_name>.<resource>: <val>\\`

For example (the default ressources):

```
agat.organized:      True
agat.origx:          15
agat.origy:          15
agat.pdx:            10
agat.pdy:            10
agat*background:     White
agat*foreground:     Black
agat*xsz:            300
agat*ysz:            300
agat*bigx:           600
agat*bigy:           600
agat*tinyx:          100
agat*tinyy:          100
agat*xm:             5
agat*ym:             5
agat*markcolor:      Red
agat*font:           -adobe-helvetica-medium-r-normal-*-18-180-75-75-p-98-iso8859-1
agat*color0:         Blue
agat*color1:         Magenta
agat*color2:         Red
agat*color3:         Sienna
agat*color4:         DimGray
agat*color5:         Springgreen
agat*color6:         Goldenrod
agat*color7:         Darkorange
agat*color8:         OrangeRed
agat*color9:         Maroon
agat*color10:        Saddlebrown
agat*color11:        Slategrey
agat*color12:        Seagreen
agat*color13:        Navyblue
agat*color14:        Pink
```

17 The *agat* C library

Here are the prototypes of all the functions that send data to the *agat* animation server in the *agat* library as defined in the include file `libAGAT.h`. All these functions accept as their first argument a string that is the name of the stream on which to send the value and as the second argument the value itself. Note that, as *agat* only knows two basic types the integer and character values are viewed as integers and the floating-point ones as *agat* floating point numbers.

```
void agatSendChar(char *f, char v)
void agatSendUChar(char *f, unsigned char v)
void agatSendShort(char *f, short int v)
void agatSendUShort(char *f, short unsigned int v)
void agatSendInt(char *f, int v)
void agatSendUInt(char *f, unsigned int v)
void agatSendLong(char *f, long int v)
void agatSendFloat(char *f, float v)
void agatSendDouble(char *f, double v)
```

18 The *agat* FORTRAN library

It is possible to use *agat* from a FORTRAN program (i.e to animate FORTRAN programs).

There are three subroutines available :

- `AGTDBL`(s, d) to send the double precision floating-point number d on the stream s
- `AGTFLT`(s, f) to send the floating-point number f on the stream s
- `AGTINT`(s, i) to send the integer i on the stream s

You should then link your FORTRAN program with the *agat* library and...

We use a package written by Burkhard Burow (see the file `cfortran.doc` somewhere in the *agat* distribution).

Note that even if you plan to only animate FORTRAN programs, you should have a C compiler available to compile the *agat* server, the *agat* C library and the small piece of code that make the correspondance between the C and FORTRAN functions of the library.

19 Errors and troubleshooting

List here the errors and warning given by the *agat* compiler. What can happen when running animations etc....

19.1 The *agat* program

- **server not connected :**

  Can't manage to establish connection with server. Check if your system is not overloaded (no space left in `/tmp` or how much process already running.)

- **Speed option -s discarded. MaxSpeed option -m used instead :**

  Maxspeed and speed options are exclusives.
- **Can't open** *display* :

  Check your $DISPLAY and X authorisations. Agat cant' connect to your X server.
- **Agat: bg color specification** *color-name* **invalid**

  The *color name* is not in the Host Portable Character Encoding. Change colorname in your resources.
- **Agat: couldn't allocate color:** *color name.*

  The X server can't allocate the *color name.* Change this color in your resources.
- **Agat: display** *display* **doesn't know font** *font name*

  The X server doesn't know the *font name.* Change your resources and give a valid X font name.
- **Bad parameter** *param* **for function** *func. func* **don't match prototypes ,**
- **not compiled for so much parameters** *nb-param...* **look at function ppCall :**

  Type or number of parameters don't match the prototype given with function declaration.
- **not compiled for so much parameters** *nb-param* ... **look at function applyFunc:**

  Sorry, agat is not compiled to deal with functions with so much parameters. But you can take a look at applyFunc to modify this.
- **uncompatible type for test expression (***param* **found).**

  Expressions used as tests in conditions (like if *exp* then) must be compatible with boolean. For instance, a floating point value can't be used.

19.2 The *agat* compiler

- **can't find such a stream:**

  You refer to a stream wich is unknow.
  Ex:
  ```
  let wrong(s1) = map(...) s2;
  ```
  s2 is unknown.
- **can't find such a register** *reg-name*

  You refer to an unknown register.
  Ex:
  ```
  map {r1} (* -> r2 = 0) s;
  ```
- **Variable** *var-name* **is unknown.**

  Ex:
  ```
  let c1 = 1;
  let c3 = c2;
  ```
  c2 is unknown.
- **Aborting: nothing to parse**

  Your animation file seems to be empty.
- **Constant** *const-name* **already defined in** *name* **:** *num* **first declaration of** *name*

  You try to redefine a constant.
  Ex :
  ```
  let c1 = 0;
  let c1 = 1;
  ```
- **Macro** *macro name* **already defined in** *name* **: : first declaration of** *name*

  i.e. for macro.
- **Stream** *stream name* **already defined in...** *name* **:** *num* **: first declaration of** *name*

  i.e. for stream.

– *name* **is taken to be a function call instead of a macro call because parameters are not stream,**
– *name* **is taken to be a macro call (hiding a function call)**

You may overload a function with a macro. Ex :

Remember **atan** is a predefined function which take a value as parameter.

```
(1) let PI = atan(1) * 4;
(2) let atan(s) = map (x -> atan x) s;
(3) let s2 = atan (s1);
(4) let PIQ2 = atan(1) * 2;
```

In line (1) the predefined function is used.
In line (2) a macro named **atan** is defined (which use the function **atan**)
In line (3) a dynamic check is done to choose between a function call or a macro call. In this case if s1 is a stream, *agat* choose to call the macro (which take stream as parameter). (generate warning for the second form)
In line (4) *agat* choose to call the function (outputing warning for the first form).

– **bad kind of parameters for a macro call :**

You try to give constant parameters to a macro. Macro use only streams.

– **no such constant** *const-name,*
– **macro** *macro-name* **undefined,**
– **macro** *macro-name* **defined with** *num* **argument,**
– **no such function** *func-name* **:**

unknown constant, macro or function.

– **pattern** *name* **will hide register of same name,**
– **pattern** *name* **will hide constant of same name** *name* **:** it num **: declaration of constant** *name*,
– **register** *name* **will hide constant of same name** *name* **:** *num* **: declaration of constant** *name* **:**

When writing an expression in a map, you can use three types of values :

- constant, which are defined outside the map.
- registers which are a kind of local variables to a map.
- patterns which are ways to access to values comming on a stream.

In case of name conflict between these values ambiguties are solved :
Patterns hide registers and constant. Registers hide constants.
Ex :

```
let a = 1;
let s2 = map { a = 2 } ( a -> a * 10) s1);
```

If a value (let say 3) arrive on the stream s1, the value emmited on s2 is 30.

– **a 'no value' pattern can't be named,**
– **a 'no new value' pattern can't be named :**

A pattern like * or ^ can't be named (they matche only when there's no value on stream).

)
– **there is** *(more, less)* **pattern than stream (** *num* **pattern for** *num* **stream)**

A group of pattern don't matche a group of streams.
Ex :

```
map (i, j -> i + j) s1, s2, s3;
```

s3 is not matched.

20 The predefined library

Here is a listing of the basic libraries shipped with the *agat* distribution.

20.1 The math library

```
let PI    = atan(1.0)*4.0;
let PI_2 = PI/2;
let PI_3 = PI/3;
let PI_4 = PI/4;
let PI_5 = PI/5;
let PI_6 = PI/6;
let PI_8 = PI/8;
let PI_12 = PI/12;

let E = exp(1);


let opp(i) = map(x->-x)i;
let abs(i) = map(x->abs(x))i;
let sgn(i) = map(x->sgn(x))i;


let acos(i) = map(x->acos(x))i;
let asin(i) = map(x->asin(x))i;
let atan(i) = map(x->atan(x))i;
let atan2(i,j) = map(x,y->atan2(x,y))i,j;
let ceil(i) = map(x->ceil(x))i;
let cos(i) = map(x->cos(x))i;
let cosh(i) = map(x->cosh(x))i;
let exp(i) = map(x->exp(x))i;
let fabs(i) = map(x->fabs(x))i;
let floor(i) = map(x->floor(x))i;
let fmod(i,j) = map(x,y->fmod(x,y))i,j;
let log(i) = map(x->log(x))i;
let log10(i) = map(x->log10(x))i;
let pow(i,j) = map(x,y->pow(x,y))i,j;
let ssin(i) = map(x->sin(x))i;
let sinh(i) = map(x->sinh(x))i;
let sqrt(i) = map(x->sqrt(x))i;
let tan(i) = map(x->tan(x))i;
let tanh(i) = map(x->tanh(x))i;
```

20.2 The statistical library

```
-----------------
-- statistic library -
-----------------




---------------------------------
-- accumulator
-- count the number of values on stream i
--
let account(i) =
        map    {c:=0}
               (_ -> c:=c+1)
                i;


--------------------------------
```

```
-- accumulator
-- compute the summ of values on stream i
--
let acsumm(i) =
        map     {s:=0}
                (x -> s:=s+x)
                 i;


-----------------------------
-- accumulator
-- compute the product of i values
--
let acprod(i) =
        map     {p:=1}
                (x -> p:=p*x)
                 i;


-----------------------------
-- accumulator
-- the average of i values
--
let acaverage(i) =
        map     {s:=0, n:=0}
                (x -> s:=s+x, n:=n+1, s/n)
                 i;


----------------------
-- map
-- these macros use the last value of each stream (i,j,k...)
-- to compute the average value.
-- warning: doesn't work like acaverage
--
let average2(i,j) =
        map(a,b -> (a+b)/2)i,j;
let average3(i,j,k) =
        map(a,b,c -> (a+b+c)/3)i,j,k;
let average4(i,j,k,l) =
        map(a,b,c,d -> (a+b+c+d)/4)i,j,k,l;
let average5(i,j,k,l,m) =
        map(a,b,c,d,e -> (a+b+c+d+e)/5)i,j,k,l,m;



-----------------------------
-- accumulator
-- the maximum of i values
--
let acmax(i) =
        map     {m}
                (x^ -> m:=x
                |x  -> m:=max(m,x))
                 i;


-----------------------------
-- accumulator
-- the minimum of i values
--
let acmin(i) =
        map     {m}
                (x^ -> m:=x
                |x  -> m:=min(m,x))
                 i;
```

```
-----------------------
-- accumulator
-- compute the maximum of 2 streams
-- warning: it doesn't work like max2 !!!
-- here, this is the maximum of all values of the
-- two streams which is returned
--
let acmax2(i,j) =
        map     {m}
                (a^,#   -> m:=a
                |# ,b^  -> m:=b
                |a^,b^  -> m:=max(a,b)
                |a ,*   -> m:=max(a,m)
                |* ,b   -> m:=max(b,m)
                |a ,b   -> m:=max(max(a,b),m)
                )i  ,j;



-------------------------------
-- filter
-- only the local extrema go through this filter
-- (ie : local max and local min)
-- for example: if i contains values given by a sin,
-- only -1 and 1 values go through
--
let extremafilter(i) =
         map    {slope,last,tmp}
                (x^  -> last:=x, slope:=0, null
                |x   -> if sgn(x-last) != slope
                          then tmp:=last, slope:=sgn(x-last), last:=x, tmp
                          else last:=x, null
                        end)
                 i;


-------------------------------
-- map
-- return the increase between successive values of i
--
let increase(i) =
        map     {last,tmp}
                (x^  -> last:=x, null
                |x   -> tmp:=last, last:=x, x-tmp)
                 i;
```

20.3 Miscellaneous functions

```
----------------
-- misc library -
----------------



-----------------------
-- map
-- create a delayed copy of stream i
--
let delay(i)=
        map     {last,tmp}
                ( x^  -> last:=x, null
                | x   -> tmp:=last, last:=x, tmp)
                  i;
```

21 The content of an *agat* dump file

Here is an example of an *agat* dump file :

```
T1 74 -1 1
T2 74 0 4
T3 74 1 1
T4 74 -4.71238898038469 1
T1 74 -0.974927912181824 1
T2 74 -0.222520933956314 1
T3 74 1 1
T4 74 -4.37578976750008 1
...
```

The format of the file is very simple :

```
<stream-name> <code> <value> <time>
```

First, this is the name of the stream.
The second parameter is a code to know the format of the numbers (integer, real...).
Third, this is the value itself.
And the last parameter is the time in micro-seconds between the arrival of the last and the current value. This is useful to replay animation.

22 Installing and porting *agat*

22.1 Customisation

Edit Makefile which is in directory .../Agat/Src. Modify the customisable part to install *agat* and the *agat*'s man in the directory you want.

```
INSTDIR=$(HOME)/Agat
MANPATH=$(INSTDIR)/Man
```

22.2 Requirements

Compiling *agat* requires a machine with a UNIX-like operating system, an ANSI C compiler and the X11 library (release 4 or higher).

22.3 Bugs

agat was tested on Sun4 and Dec. If you find bugs, please report them.
There are known bugs on DecAlpha and Linux sytems. If you find bugs on other systems, please report them to :

Stéphane Lavirotte (lavirott@essi.essi.fr)
or
Olivier Arsac (oarsac@sophia.inria.fr)

Contents