

A Language for Abstract Geometrical Computation

Jérôme DURAND-LOSE*

May 11, 2020

Abstract

To manipulate signal machine, we design a new language to have special syntax for collision rules, meta-signals. . . To encode modifications and generation of parameterized signal machine, a whole computing language has been implemented. This document presents the language in general and then the libraries specific to AGC.

This is classical imperative language with variables, expressions, conditional, loops, functions. . . It is un-typed. It provides functional/lambda traits.

This language is targeted to Abstract Geometrical Computation, in particular:

- to define signal machine, configuration. . . and run them
- to manipulate them

It the future, it is imagined to:

- implements exceptions,
- implements inductive structure definition and filtering (as in Caml),
- implements object oriented programming, and
- delayed evaluation.

Although this is not meant to be a technical documentation, main Java classes are indicated in order to provide entries into the javadoc for the language and AGC.

Somethings you have to be warned about

The only numbers are integers (-157) and rational numbers (-45/7) with exact precision and without limitations. There is no approximation nor decimal numbers.

There are some hardwired *personal best practices* that relate only to my own personal experience and opinion:

No=`directive` To avoid misspelling, test is `==`, affectation is `:=`. Operator `=` is not provided and cannot be used nor defined, and

< and `≤` only directive Comparisons are usually easier and faster to read and manipulate when they are always written in the same direction (here increasing order). Operators `>`, `>=`, and `≥` are not provided and cannot be used nor defined.

*Contact: Jerome.Durand-Lose@univ-orleans.fr.

You may agree or not with these, may also consider this outrageous, fell free not to use this language.

This language has been developed especially to cover some needs. This explains why usual operators might not be available yet.

Relating to the technical documentation

Although this is not meant to be a technical documentation, main Java classes are indicated in order to provide entries into the javadoc.

The language is implemented in the package `fr.univ_orleans.jdl.language`. The top class of the inheritance hierarchy for the language is `Chunk` that mostly divides into values (`Value`) and operators (`Op`) plus library definitions (`Library_Abstract`).

There are also a bunch of classes that the language refers through. The core AGC java library is the package `fr.univ_orleans.jdl.agc`.

Contents

1	Types (values, basic operators and expressions)	3
1.1	Basic / Atomic Types (<code>Value_Atomic</code>)	4
1.1.1	Boolean (<code>Value_Boolean</code>)	4
1.1.2	Numbers: Integers (<code>Value_Integer</code>) and Rationals (<code>Value_Rational</code>) . .	4
1.1.3	Glyphs (<code>Value_l_Glyph</code>): Characters (<code>Value_Character</code>) and Strings (<code>Value_String</code>)	5
1.1.4	Special values (<code>Value_Special</code>)	5
1.2	Test on Basic / Atomic Types	6
1.2.1	Equality and Comparison (<code>OP_Test_Sequence</code>)	6
1.3	Lists (used as arrays)	6
1.3.1	Creating	6
1.3.2	Accessing a value (as a variable)	6
1.3.3	Size	6
1.3.4	Concatenating	6
1.4	Aggregation type	6
2	Flow control	7
2.1	if Conditional execution	7
2.2	foreach Loop	8
2.2.1	foreach Loop with Range	8
2.2.2	foreach Loop with Iterable	8
2.3	while Loop	8
3	Context/block	9
3.1	Variable	9
3.2	Constants	9
3.3	Associations	9
3.4	Access to specific contexts	9
3.5	Contexts can be reopened	10

4	Libraries	10
4.1	Code inclusion	10
4.2	Precompiled/java libraries	10
4.3	Naming conventions	10
4.3.1	Global function	10
4.3.2	Method	11
5	Functions	11
5.1	Calling	11
5.2	Return :=	11
6	Library AGC	11
6.1	Signal Machine	11
6.1.1	Creation	11
6.1.2	Query for elements	12
6.2	Meta-Signal	12
6.2.1	Creation	12
6.2.2	Queries on a Meta-Signal	12
6.3	Collision Rule	13
6.3.1	Creation	13
6.3.2	Undefined rules	13
6.3.3	Queries on a Rule	13
6.4	Configuration	13
6.4.1	Creation	13
6.4.2	Queries on a Configuration	14
6.5	Signal	14
6.5.1	Creation	14
6.5.2	Queries on a Signal	14
6.6	Run	14
6.6.1	Creation	14
6.6.2	Act on a Run	15
6.7	To String	15

1 Types (values, basic operators and expressions)

If an operator is specific to a type, it is presented in the type. Otherwise it is listed after the basic types. Almost all are usual and have the usual priority. Priorities are gathered in Fig. 1.

There might be some added types in the future¹

Parenthesis have their usual meaning for sub-expressions as well as for function calling.

¹We are thinking about: objects, exceptions and exceptions.

1.1 Basic / Atomic Types (`Value_Atomic`)

They correspond to non-mutable/constant/final values: boolean, numbers, characters, string and some special values. These are single values as opposed to compounds (list...), contexts (hash tables...), function and labels (see corresponding sections).

1.1.1 Boolean (`Value_Boolean`)

There are only two values: `false` (`Value_Boolean.Value_False`) and `true` (`Value_Boolean.Value_True`).

The operators are:

- ! Boolean negation (`OP_Not`)
- && Boolean conjunction (`OP_And`)
- || Boolean disjunction (`OP_Or`)

Lazy evaluation. By default, the last two operators are lazy: the right operand is evaluated only if needed. This means that although this operand has to be syntactically valid, if it is not evaluated, it is not tested for semantic validity (i.e. variable may be not defined, wrong type used for operators...).

It is possible turn on or off the lazy evaluation (e.g. for semantic test or for side effect) with the method `make_boolean_evaluation_lazy` in `Eval_Context`. Yet it is not possible to switch from the program.

There are a few examples of expressions:

- `! true`
- `true && false`
- `true || false`
- `true || false && true`
- `(true || false) && true`

1.1.2 Numbers: Integers (`Value_Integer`) and Rationals (`Value_Rational`)

Numbers are unbounded. Operations are exact and belongs to two compatible kinds:

Integers. They can be positive, null or negative. There is no limit value (`BigInteger` is used for internal encoding).

Rationals. They can be positive, null or negative. There is no limit to the values of Rational numbers (i.e. `BigInteger` are used both for numerator and denominator).

A rational number is denoted using a slash if necessary: e.g. `3/5`, `-153/5`, `35`. There is no special way to input a rational number but as the division of integers.

The predefined values, `plus_infinity` and `minus_infinity` are use respectively for $-\infty$ and $-\infty$ as rational values.

The operators are:

- `+`, `-` Addition (`OP_Addition`) and subtraction (`OP_Substraction`)
- `*`, `/` Multiplication (`OP_Product`) and exact division (`OP_Division`)

`/%`, `%` Integral division (`OP_Division_Integral`) and remainder (its rest, `OP_Remainder`). The integral division ensures that, for all x and y , $y \neq 0$, $x = (x/\%y) * y + (x\%y)$ and $|x\%y| < |y|$ and $0 \leq x\%y$ if $0 < x$ and $x\%y \leq 0$ otherwise.

- Unary minus, change sign (`OP_Change_Sign`).

These operators returns rational or integers (when most appropriate). The operation is always exact: there is no approximation nor overflow. Yet division (exact or integral) and remainder by zero results in an undefined (or infinite value) and the rest of the computation is not defined. As a matter of fact, a java exception is raised by the interpreter².

N.B. The division is exact, which means that the division of two numbers may be an integer or a rational.

There are a few examples of expressions and the associated evaluations:

- $5 + 7 \rightarrow 12$
- $2 + 3 * 5 \rightarrow 17$
- $3 / 5 + 1 \rightarrow 8 / 5$
- $50 / 6 \rightarrow 25 / 3$
- $50 /\% 6 \rightarrow 8$
- $50 \% 6 \rightarrow 2$
- $50 /\% (6 / 5) \rightarrow 41$
- $50 \% (6 / 5) \rightarrow 4/5$
- $(13 / 5) /\% (2 / 5) \rightarrow 6$
- $(13 / 5) \% (2 / 5) \rightarrow 1 / 5$

N.B. There is no decimal nor approximation numbers.

1.1.3 Glyphs (`Value_I_Glyph`): Characters (`Value_Character`) and Strings (`Value_String`)

Character represent a single character, exactly as a Java char (which is used for storage). They are separated by '.

String are a sequence of characters, exactly as a Java String (which is used for storage). They are separated by ".

Character are understood as graphical representation like String and not as “small integers”. This implies that, on the one hand, the conversion forth and back to char (using unicode) is not automatic, and on the other hand, they are treated as length one string.

The operators are:

_ Concatenation: if any operand is not a glyph, it is converted.

This types lacks many usual operators that should be provided in the future³

1.1.4 Special values (`Value_Special`)

These are unique values that account for special cases⁴:

- error, an error,
- undef, the absence of a definition for a variable, and

²In the future, a language exception should be raised, but exceptions are not implemented yet.

³Typically, character access, sub-strings, regular expressions, cast. We think to use the python syntax for extracting.

⁴Should the need arise, possible additions are: unit (like in caml, one value type), null (the absence of a reference, but references are addressed in the language).

- `void`, the absence of a value.

Basically these values can only be tested for equality or printer, otherwise it mostly provokes an error/exception/end of program.

1.2 Test on Basic / Atomic Types

1.2.1 Equality and Comparison (`OP_Test_Sequence`)

The `<` and `≤` methodology is enforced: there is no greater than comparator.

Shortcut `3<5<=7` correspond to `(3<5) && (5<=7)`. It is evaluated left to right and it is lazy (as explained above for booleans).

Lazy evaluation. By default, the last two operators are lazy: the right operand is evaluated only if needed and is never re-evaluated. This means that although this operand has to be syntactically valid, if it is not evaluated, it is not tested for semantic validity (i.e. variable may be not defined, wrong type used for operators...).

1.3 Lists (used as arrays)

1.3.1 Creating

Usage: `[]` the empty list

Usage: `[<expr>]` one element list

Usage: `[<expr> , <expr> *]` many element list

1.3.2 Accessing a value (as a variable)

Usage: `<list>[<expr>]` The expression must evaluate into an integer.

1.3.3 Size

Usage: `# <expr>`

If the expression does not evaluate to a list, then an error is raised.

1.3.4 Concatenating

Usage: `<expr> _ <expr>`

Any expression not evaluated to a list is considered as a one element list.

N.B. It is the same symbol as the string concatenation.

1.4 Aggregation type

The only record/structure-like type is associative arrays that are also used as environment/context blocks. They are also presented in Chapter 3.

Highest priority		Associativity
!	Boolean negation	(unary, on left)
#	Unary list size	(unary, on left)
*	Multiplication	left
/	Division (exact)	left
/%	Integral division	left
%	Modulo	left
+	Addition of numbers	left
-	Subtraction of numbers	left
_	Concatenation (list and strings)	left
-	Unary minus, change sign	(unary, on left)
==	Equality test	left, lazy
!=	Inequality test	left, lazy
<	(strict) inferiority test between comparable values	left, lazy
<=	Inferiority or equality test between comparable values	left, lazy
&&	Boolean conjunction (by default lazy)	left, lazy
	Boolean disjunction (by default lazy)	left, lazy
Lowest priority		Associativity

Figure 1: Priority of operators.

2 Flow control

Please note that ‘;’ at the end of instruction are mandatory since braces also denotes contexts (see Chap.3)⁵ So that the instruction will provide it as seen in the examples.

2.1 if Conditional execution

Usage:

```
if ( <expression> )
  <instruction>
```

Example:

```
if ( v < 3 ) {
  println ( v * "ok" );
} ;
```

Usage:

```
if ( <expression> )
  <instruction>
else
  <instruction>
```

Example:

⁵This might changed, but this would be a major modification not backward compatible.

```
if ( v < 3 )
    println ( v _ "ok" ) ;
```

```
if ( v < 3 ) {
    println ( v _ "ok" ) ;
} ;
```

```
if ( v < 3 ) {
    println ( v _ "ok" ) ;
} else
    print ( "otherwise" ) ;
```

Please note that `if` returns a value as in:

```
v := if ( ! true || 24 - 23 / 7 <= 300 ) 45 else 65 ;
```

2.2 foreach Loop

2.2.1 foreach Loop with Range

Usage:

```
foreach ( <variable> : <expression> ... <expression> )
    <instruction>
```

Example:

```
foreach ( v : 0 ... 7 )
    println ( v * 2 ) ;
```

```
foreach ( v : 0 ... 0 ) {
    println ( "Done one for 0" ) ;
} ;
```

2.2.2 foreach Loop with Iterable

Usage:

```
foreach ( <variable> : <iterable> )
    <instruction>
```

Example:

```
foreach ( v : [ 0 , 2 , 5 ] ) {
    println ( v * 2 );
} ;
```

2.3 while Loop

Usage:

```
while ( <condition> )
    <instruction>
```

Example:


```

while ( true )
  println ( v * 2 ) ;

while ( 0 < n ) {
  println ( "Done one for " _ n ) ;
  n := n + 1
} ;

```

3 Context/block

They amount for locality and works as associative arrays/maps (variable to value).

Each block is a context and can be refereed by a variable.

Block are disposed of by Java garbage collector. Thus only when not accessible anymore, but may remain for some time.

Some constructs like signal machines and configurations are a specialization of context.

Context can be re-open with operator `::`.

This is static since label are constants.

Context are nested up to the global context. If a label is not found, the context right above is searched for.

3.1 Variable

Variable are type-less references to values. But each value have a type.

There is no pointer: variable cannot refer to variable.

Affectation is done with `:=`

3.2 Constants

Constants are type-less fixed references to values. But each value have a type.

There is no pointer: variable cannot refer to variable.

Constant affectation is done with `.=`

3.3 Associations

Usage: to define `<expr> -> <expr>`

Usage: to get `<expr> >`

It remains valid in above and below context (but not siblings)

This is dynamical since it relies on expressions.

3.4 Access to specific contexts

Usage: `<ctx> :: <label>` refer to variable/constant label in context *ctx*.

There are special access:

Usage: `. <label>` means here, exactly in the current context.

Usage: `.. <label>` means the above context.

In a function, it is possible to access the calling context⁶

⁶This is very special and not an advised feature.

Usage: ? <label> means the calling context in a function (it can have to go up levels to find it).

There can be some combination of these:

Usage: . :: .. <label> means exactly the above context.

3.5 Contexts can be reopened

Usage: <ctx> :: { <instruction>* } ; means instructions in this context.

Please be careful and make a lot of test to see exactly what is the meaning of, e.g. .. inside it!

4 Libraries

4.1 Code inclusion

The `load` primitive is used to include another file.

Usage: `load <string> ;`

<string> is the name of the file to include.

4.2 Precompiled/java libraries

The `use` primitive is used to access a libraries in java.

Usage: `use <string> ;`

<string> is the name of the library. It must correspond to the name of a java class.

Libraries are search, first in package then on top as a full qualified name.

4.3 Naming conventions

Remember, a function/method is an action. It should correspond to a verb (i.e. `create` and not `creation`).

Functions can either be at global level or inserted into some environment. When the environment can be considered as an object (as in Sect. 6) then the method convention is used otherwise it is the ones of global functions.

4.3.1 Global function

There is (yet?) no namespace-like (as in C++) mechanism, so that function names have to be long and non ambiguous.

The convention is to form names by concatenating the name of the type of the first argument and then the verb (and qualification). This is thought after the `class::method` of C++.

Examples: `list_sorted_unique_insert` to insert an element in list that are supposed sorted without repetition. This also indicate that the first element is expected to be such a list.

4.3.2 Method

It is expected to be a verb (plus qualification).

and object oriented features are very limited. Nevertheless, the terminology "instance" is used for a better understanding.

There is no `new` (it is reserved in case). Please use `create` to generate new instance.

The `get_` and `set_` should be reserved to query and modification of the state of the instance but have no border effect what-so-ever.

If there is a border/side effect (like the creation of a new instance of some other type) or the modification is non-trivial or semantics imposes it, please use another verb.

5 Functions

5.1 Calling

It is done by either providing an argument list (eventually empty) or by opening the environment after. Not doing any of this just yields the function as a value.

5.2 Return :=

This means: end of the function all and report the value after as the result of the function.

Usage: `:= <expr>`

For example `:= 3 * 4` ; returns the value 12.

Anything that can be considered as value can be returned.

6 Library AGC

Signal machines are not presented in this document. Please refer to this document for an introduction.

This library is presented like it should be used so that some query function appear only when all concerned types are presented.

The library is loaded by:

```
use AGC ;
```

6.1 Signal Machine

6.1.1 Creation

Creating an empty signal machine is done by:

```
sig_mach := create_signal_machine {} ; // or  
sig_mach := create_signal_machine () {} ; // or  
sig_mach := create_signal_machine () ;
```

if nothing is to be done in the context.

This context is the right place to create meta-signal, configuration, etc.

6.1.2 Query for elements

Meta-Signal (list or individual). This can only be done inside/referring to a signal machine.

To get the meta-signal corresponding to a specific id:

```
sig_mach . get_meta_signal ( "id" ) ;
```

If there is no such meta-signal then void is returned.

To get a table containing all the meta-signals:

```
sig_mach . get_meta_signal_list () ;
```

Rules (list or individual). This can only be done inside/referring to a signal machine.

To get the list of all the collision rules:

```
sig_mach . get_rule_list () ;
```

To get the output associated to a set of in-coming signals according to defined rules:

```
sig_mach . get_rule_output ( in ) ;
```

where in is a list of incoming meta-signals. If there is no corresponding rule then void is returned.

6.2 Meta-Signal

6.2.1 Creation

This can only be done inside/referring to a signal machine.

```
meta_sig := add_meta_signal ( "id" , 3/5 ) { color => "DarkRed" ; } ;  
// or  
meta_sig := sig_mach.add_meta_signal ( "id" , 3/5 ) { color => "DarkRed" ; } ;
```

It is also possible to provide a line style:

```
line_style => "dashed" ;
```

The different possible values are: densely dashed, loosely dashed, densely dotted, dotted and loosely dotted. Beware that drivers might not implement these (or implement others more specific).

6.2.2 Queries on a Meta-Signal

To get the id from a meta-signal:

```
meta_sig . id ;
```

To get the speed from a meta-signal:

```
meta_sig . speed ;
```

To get any property from a meta-signal:

```
meta_sig . get ( "color" ) ;
```

Please note that the property is always returned as a **String** (i.e. not a **Color** nor an **Object**)

6.3 Collision Rule

6.3.1 Creation

This can only be done inside/referring to a signal machine.

```
rule := [ sm1 , sm2 ] --> [ sm2 , "m3" ] ;
```

of from outside of the target machine, it can be indicated on the operator:

```
rule := [ sm1 , sm2 ] sig_mach . --> [ sm2 , "m3" ] ;
```

6.3.2 Undefined rules

If rule is not defined, by default the signal machine consider it blank: the same meta-signals are regenerated. This corresponds to signals “crossing” each other and avoid the long listing of such rules.

It is possible to change this behaviour for:

- used a java define class that must implements
fr.univ_orleans.jdl.agc.kernel.I_undefined_Collision_Rule:

```
undefined_rule => "fr.univ_orleans.jdl.agc.kernel.Undefined_Collision_Rule" ;
```

- provide a function:

```
undefined_rule => ( args ) -> {  
    println ( args ) ;  
    := [ sm1 , "m3" ] ;  
} ;
```

The last one can be used for implicit declaration of rules.

6.3.3 Queries on a Rule

To get the in-coming part of a rule:

```
rule . in ;
```

To get the out-going part of a rule:

```
rule . out ;
```

6.4 Configuration

6.4.1 Creation

They are created inside signal machines:

```
configuration := sig_mach . create_configuration ( ) ;
```

6.4.2 Queries on a Configuration

The following function returns the list of existing signals:

```
configuration . get_signal_list ( ) ;
```

The field machine is used to retrieve the signal machine.

The function `print_signal_list()` can be used to print all the signal in the configuration according to the Java AGC library. It provides a lot of information but the output cannot be used as input to the language.

6.5 Signal

This can only be done inside/referring a configuration.

6.5.1 Creation

Inside a configuration

```
a @ 110 / 3 ;
```

Where `a` is either a string (meta-signal's id) or a meta-signal.

Or referring to a configuration at the operator:

```
"id" configuration . @ 110 / 3 ;
```

6.5.2 Queries on a Signal

To get the associated meta signal:

```
print ( signal . meta_signal ) ;
```

To get the date and position at birth:

```
print ( signal . birth_date ) ;  
print ( signal . birth_position ) ;
```

6.6 Run

6.6.1 Creation

A run is an execution of a signal machine on a configuration. Since the signal machine is known to the configuration, it is enough to call it from a configuration.

```
run := configuration.run ( ) ;
```

6.6.2 Act on a Run

A run can run for a given number of collision ticks (times where there is at least one collision).

```
run.step ( 200 ) ;
```

This is to advance up to some given time. Please note the due to the continuous nature of time, there could be an infinite number of collision before that time and the computation may not finish.

```
run.until ( 3 / 7 ) ;
```

An optional argument can be given, it is the number of collision times to do it the time bound is not reach.

```
run.until ( 3 / 7 , 2000 ) ;
```

The last and more useful is the function to export a space-time diagram in various formats:

```
run.export ( "PDF" , "file.pdf" , {  
  scale := 1 ;  
  clip_left := 41 ;  
  clip_right := 110 ;  
  clip_end := 4 - 1/2 + 1/8 ;  
  clip_start := 2 + 1 - 1/4 ;  
} ) ;
```

The first argument is a string indicating the output type (PDF, EPS, LATEX, LATEX_PIC_ONLY, SVG). The second is the name of the output file. Then comes an environment adding more information on the desire output. Basically one can scale and clipping.

If you want to use different scales on x and y you can use `scale_x` and `scale_y`. Please note that the order of treatment of parameters is not defined, so that using both `scale_x` and `scale` might not generate the expected result.

The L^AT_EX drivers, LATEX, generates a single autonomous file that can be compile directly while LATEX_PIC_ONLY generate only the picture so that it can be included by some main file. In the latter case, the main file is supposed to include needed package (mostly `tikz`).

For these drivers the name of the command to use can be defined by a the property `latex_command` in the meta-signal:

```
meta_sig := add_meta_signal ( "id" , 3/5 ) {  
  latex_command => "\DrawSigDarkRed" ;  
} ;  
// or  
meta_sig . { latex_command => "\DrawSigDarkRed" ; } ;
```

The command is then used with the parenthesis syntax in latex:

```
\DrawSigDarkRed(0,0)(1.5,2.7)
```

6.7 To String

Signal machines, meta-signals, configurations and runs have a `to_string` function defined.

Check