Writing and Porting Scripts in Cell Illustrator 5.0

Table of Contents:

1. Introduction 3
2. Scripting Languages 3
3. Porting Models created with CI 4.0 5
   a. Set Pnuts as Default Scripting Language for the Model 5
   b. Set Pnuts as Script Language for Selected Scripts 6
   c. Port Selected Scripts to Java 6
4. Porting Scripts 8
   a. Math Scripts 8
   b. Mathematical Functions 9
   c. Condition Statements 10
   d. Special predefined functions 11
   e. Complex/Generic Scripts 11
5. Appendix A. Definition of Simple Scripts 14
6. Appendix B. Definition of *Simplemath* Scripts 15

1. Introduction

Scripts are used in Cell Illustrator to define complex logic of the Petri Net and customize it.

Cell Illustrator (CI) 4.0 supports only one scripting language - *Pnuts*. Pnuts is a powerful language, quite similar to java, which allows defining classes and using java libraries. Nevertheless, in most cases scripts look like simple mathematical expressions (e.g. '*2+3*m1*'), sometimes enhanced by additional functions ('*getSimulationTime(simulator)*', '*beta(x)*') or expressions (such as conditions '*if(..) ...else ...*' or grouping instructions '*{...}*').

In CI 5.0 a more general scripting mechanism is used. It is taken from *Java Scripting API* and allows usage of many different languages in one model. CI 5.0 supports several scripting languages: *simplemath*, *Java*, *JavaScript* (*js*) and *Pnuts*. All these languages are quite similar, however each of them differs in details of their syntax. Scripts written in one language might not be correct in different language.

Models created and simulated in CI 4.0 included *Pnuts* scripts only. Simulation of these models in CI 5.0 might cause script execution errors, since *Pnuts* is NOT the *default scripting language* of CI 5.0.

The main goal of this document is to describe:
- How to port CSML models created with CI 4.0 and run simulations in CI 5.0?
- How to port *Pnuts* scripts written in CI 4.0 to script languages of CI 5.0?

Moreover, this document gives answers the following questions:
- What are the differences between *Pnuts* and the other scripting languages?
- What are the differences between scripting languages available in CIO 5.0?

2. Scripting Languages

CI 5.0 uses java scripting API for executing the scripts, this API comes with sample scripting language which is JavaScript. The scripting framework which executes the scripts can be quite slow. Fortunately, most of the scripts in Cell Illustrator models are simple mathematical expression, e.g. m1*m2, 0.1*m1. Thus, highly customized *simplemath* scripting language was developed for this project and added to the scripting framework. Additionally two types of scripting languages based on Java can be used: *java* and *java-bulk*. For those types the scripts are written in Java, then compiled into java byte-code and executed within the scripting framework as normal java method. For languages ported to the Java scripting framework refer to https://scripting.dev.java.net/.

Note: Not all ported languages are optimized to be used in the scripting framework. CI 5.0 compiles and executes the scripts in the standard way as recommended officially announced by Sun Inc.. Therefore script languages (e.g. *Pnuts*) that do not fully support the official rules might cause problems, e.g. a scripts might require a predefined object for its execution.

Note: Very *simple scripts* (such as '*1*','*true*','*2.434*' or '*m1*'), are not executed by evaluating the script in the selected language, but they simply return the value as it is written. The definition of *simple scripts* is summarized in Appendix A.

Language	**Properties**
simplemath	Default Scripting Language Quite restrictive syntax Allows writing mathematical expressions and some logical expressions Provides many mathematical functions (basically it allows doing operations as medium level calculators) Very fast (basically one java function is invoked for each operation), quite similar performance to java for not very complex scripts. Most of the scripts can be used later in SECG The syntax supported by *simplemath* script is summarized in Appendix B.
java	Allows for writing scripts in normal java code. The code is placed inside a java function from compilation and execution. Fast. Very fast during execution. However slow at startup, since creation of the simulation engine may take a lot of time - each of scripts is compiled separately. Most of the scripts can be used later in SECG. Requires JDK, JRE is not enough.
java-bulk	Identical to java, but all scripts are compiled at once, so the generation of engine is much faster than java. However there is one drawback: If the script has errors, the error report can only show the compile error but cannot point out the exact place where the error script is written, i.e. kinetic script in process p1. The recommended usage is to switch to *java* script to check the cause of error and switch back to *java-bulk* after the model compiles successfully.
js	Allows for writing scripts in js (*javascript*) language. Not fast, not so fast as java at execution time. More simple syntax than Java. Not compatible with SECG. Scripts created in js might not run with SECG. Not recommended for porting/conversion of CI 4.0 *Pnuts* scripts, complex scripts might be very difficult/impossible to port.
Pnuts	The *Pnuts* script is supported mainly for the backward compatibility to old Cell Illustrator. In CI5.0 the running speed of *Pnuts* is very slow, since the implementation of *Pnuts* is not customized to the java scripting framework. When loading the model created in CI4.0 to CI5.0, it is recommended to switch to other supported script language in CI5.0. In most cases the scripts will work just after switching the script from *Pnuts* to other language in the *Simulation Setting* dialog or *Element Setting* dialog. If a model contains a few complicated scripts, one solution is to set *Pnuts* as language for these complex scripts and set the default scripting language as *simplemath* in simulation settings dialog. In such a case the slow *Pnuts* executor will be used for the few complex scripts only, while rest of the scripts will be executed by using the fast *simplemath* executor.

Note: When creating the script leave the language field empty and use the default language of the model. The language should be set for very specific cases only, e.g. when you would like to add some condition processing or use outside feature.

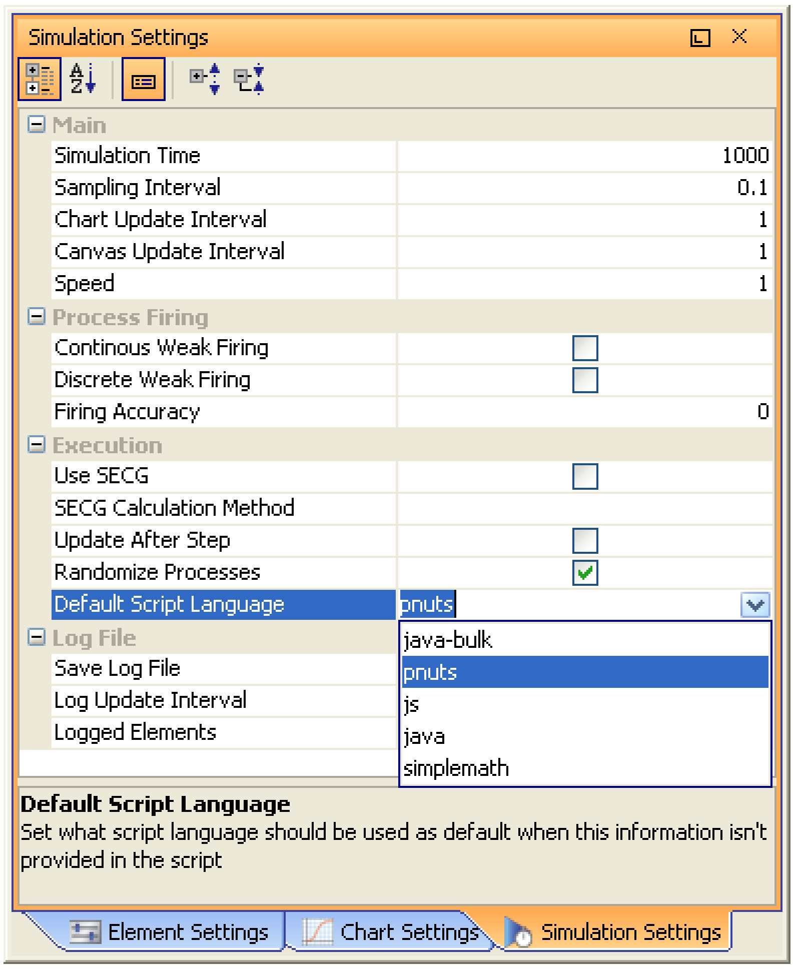
3. Porting Models created with CI 4.0

After loading models created with CI 4.0 into CI 5.0 workspace, it is recommended to run a simulation to verify whether the simulation can be run with no script execution errors. If the simulation run results in script execution errors, the user has to manually fix the problems choosing one of the methods described below

a. Set Pnuts as Default Scripting Language for the Model

This is the simplest solution:

- Open the *Simulation Settings* Frame by clicking its icon on the right toolbar or by choosing Window | Show Frame | Simulation Settings from the menu
- Set *Pnuts* in Default Scripting Language field by double clicking on the it and choosing *Pnuts* in the list.



Simulation Time	1000
Sampling Interval	0.1
Chart Update Interval	1
Canvas Update Interval	1
Speed	1
Continous Weak Firing	☐
Discrete Weak Firing	☐
Firing Accuracy	0
Use SECG	☐
SECG Calculation Method	
Update After Step	☐
Randomize Processes	✓
Default Script Language	pnuts
Save Log File	
Log Update Interval	
Logged Elements	

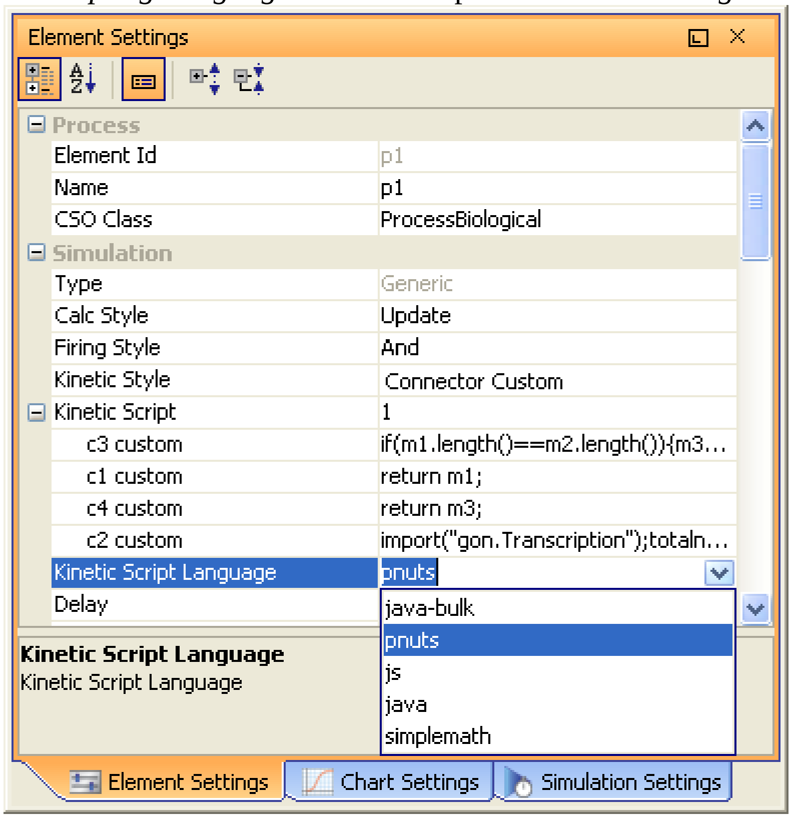
Default Script Language
Set what script language should be used as default when this information isn't provided in the script

This solution is straight forward and recommended in most of the cases. However *Pnuts* language is not recommended for simulation of large models since it is slower than the other scripting languages *Java*, *simplemath*. Also models converted in that way cannot be simulated with SECG (Simulation Engine Code Generator), since the code generated with this engine must be compatible with Java.

b. Set Pnuts as Script Language for Selected Scripts

Execution of Pnuts scripts is much slower than of java or simplemath scripts. Therefore, if the simulation speed matters, it recommended to minimize the number of Pnuts scripts. This can be done in the following way:

- Open the CSML model created with CI 4.0 in CI 5.0
- Run the simulation.
- Find the scripts that cause execution errors using the *Simulation Errors* frame
- Change the *Scripting Language* of these scripts in *Element Settings*



Element Settings

Process	
Element Id	p1
Name	p1
CSO Class	ProcessBiological
Simulation	
Type	Generic
Calc Style	Update
Firing Style	And
Kinetic Style	Connector Custom
Kinetic Script	1
c3 custom	if(m1.length()==m2.length()){m3...
c1 custom	return m1;
c4 custom	return m3;
c2 custom	import("gon.Transcription");totaln...
Kinetic Script Language	pnuts
Delay	

Kinetic Script Language
Kinetic Script Language

Please note: For some script you will be not able to set a script language or the language property will be common for many scripts (this is the case for kinetic parameters for process, for example if process allows you specifying coefficient1 and coefficient2 than they will share the language property). The default language for the whole model can be set in simulation settings. This default language will be used whenever the script doesn't have the language set or its language cannot be set.

c. Port Selected Scripts to Java

An alternative way to changing the Script Language to Pnuts, is to port the Pnuts script to the recommended scripting language *Java* or *simplemath*. Using this approach the number of *Pnuts* scripts can be minimized to 0, which should make the simulation much faster. To do this:

- Open the CSML model created with CI 4.0 in CI 5.0
- Run the simulation.
- Find the scripts that cause execution errors using the *Simulation Errors* frame

- Open the problematic scripts in *Script Editor* and convert/port the script to Java syntax.

The issue of *Porting Scripts* is described in the next chapter in detail.

4. Porting Scripts

The *Pnuts* in CI 4.0 isn't very restrictive to the form of the script (generally scripting languages aren't), so it allowed omission of brackets, not using return statement or even not returning a value. Generally the scripts can be divided into groups depending on their complexity:

a. Math Scripts

Math Scripts are pure mathematical formulas (algebraic expressions) that are contained in one statement.

Examples
*M1*m2/4*
*sin(5)*cos((4+m1)/2)*(m1/(m1+m2))*
return m1;

Math scripts are simple, one line scripts. They can be written in two ways:
- *Simplemath* style – pure algebraic expression
- *Java* style – mathematical formula surrounded with 'return' and ';'

In general both styles are supported; CI 5.0 automatically converts between *simplemath* and *Java* style by adding or removing the surrounding 'return' and ';', if necessary.

It is recommended to use *simplemath* style when writing the scripts. Such scripts should neither include 'return' nor semicolon ';',

In CI 4.0 the scripts were written in Puts language. The porting of math scripts from CI 4.0 to CI 5.0 is summarized in the table below.

CI 4.0 Expression *Pnuts* style	**Conversion from CI 4.0 to CI 5.0**	**CI 5.0 Expression *simplemath* Style**	**CI 5.0 Expression *Java* Style**

m1
return m1
return m1;
	Automatic	m1	return m1;

m2+m3 return m2+m3;	Automatic	m2+m3	return m2+m3;

m1=m2+m3;
m1=m2+m3
	User check/fix required.	m2+m3	return m2+m3

m1=m2+m3;return m3;	User check/fix required.	m2+m3	return m2+m3

b. Mathematical Functions

As for the math functions there are two cases:

- *Standard* functions which include common math functions (sin, pow, exp) defined in the scripting languages.
- *CI-Specific* math functions written in *Pnuts* specially for CI 4.0 (like LSMass, median, beta)

Java, *simplemath* and *js* scripts may use all the functions and constants defined in Math class - http://java.sun.com/javase/6/docs/api/java/lang/Math.html. *Pnuts* has a very similar set of functions.

CI-Specific functions were ported to *simplemath* and *java* – through static import of simulation functions.

CI 4.0 Expression ***Pnuts* style**	***CI 5.0 Expression*** ***Java*** ***Simplemath*** ***Pnuts***	***CI 5.0 Expression*** ***Js***
	Standard Math Functions	

sin(x)
cos(x)
max(x,y)
etc.
	Identical to CI 4.0 Pnuts style	Math.sin(x)
		Math.cos(x)
		Math.max(x,y)
		etc.

	Standard Math Constants	

PI()
E()
	Identical to CI 4.0 Pnuts style	Math.PI
		Math.E

	CI-Specific Math Functions	

median(x,y,z)
beta(x,y)
gamma(x)
rand()
SMass(x,y)
LSMass(x,y)
LSMass2(x,y)
pow(x,y)
	Identical to CI 4.0 Pnuts style	Not available

Moreover *simplemath* operates on list of doubles, so it is acceptable to write '*exp(m1,m2,m3,…)*' or '*min(m1,m2,m3,…)*', simply the additional arguments will be omitted (please note that it may happen in the future that for functions such as min or median will operate on all provided arguments (also there is a simple way of adding additional math function to simplemath).

c. Condition Statements

Examples

```
m1>3?34:m1*0.1
if(m1>3) 34; else m1*0.1
if(m1>3)
{
	return 34;
}
else
{
	return m1*0.1;
}
```

Above scripts generally check some condition and return a result depending on the outcome. Basically there are two ways of defining conditions:

- by ‘?:’ operator - available in all languages, and
- by ‘if/else’ operator (*java, Pnuts, js*)

Both ways are supported in SECG. The ‘?:’ is better in the sense that the script can be written in one statement and is similar to math script case described before. The if/else statement generates more problems with the usage of ‘return’, ’;’ and ‘{..}’. For instance Pnuts has some problems with return statement at runtime. Also please note that if the script is complex (if/else) you must use return statement in Java. And please also take care that all paths return a value (this may otherwise generate null pointer exceptions at runtime.

CI 4.0 Expression *Pnuts* style	***CI 5.0 Expression Java***	***CI 5.0 Expression Simplemath***	***CI 5.0 Expression Js***
?: condition operator			
m2 > 0.5 ? 1 : 0;	Identical to CI 4.0 Pnuts style	Identical to CI 4.0 Pnuts style	Identical to CI 4.0 Pnuts style
If/else condition operator			

if (m2 > 0.5)
{
return 1;
}
else
{
return 0;
}
	Identical to CI 4.0 Pnuts style	NOT SUPPORTED	if (m2 > 0.5)
			{
			result = 1;
			}
			else
			{
			result = 0;
			}

All languages provide support for conditional operators ‘>’,’<’,’==’,’!=’ and for logical operators ‘||’, ‘&&’. Thus, you can generally write conditions such as ‘*((m1>3||m4+3<m1) && m2!=m1)*’.

Note: ‘!’ operator is not supported in *simplemath*. To use that, need to use other scripting language.

d. Special predefined functions

In CI 4.0 user could use predefined functions:
getElapsedTime(simulator),
getSamplingInterval(simulator) and
IfTime(simulator,time).

In CI 5.0 there are corresponding variables *time*, *samplingInterval* and *simulationTime*. The predefined functions are replaced by the corresponding variables or constructs before compilation of all scripting languages. **Thanks to that the CI4.0 model with those functions will run without problem.**
The four functions should be treated as deprecated.

Conversion rules of CI4.0 predefined time functions to CI5.0 constants is show in the table below:

CI 4.0 Expression ***Pnuts* style**	***Converted CI 5.0 Expression*** ***Java*** ***Simplemath*** ***Pnuts*** ***Js***
getElapsedTime(simulator)	time
getSamplingInterval(simulator)	samplingIntreval
IfTime(simulator,timePoint).	IfTime(simulator,time) is equivalent to: samplingInterval/2>abs(timePoint-time)

Also additional *sequence* manipulation functions exist in CI 4.0, like transition, transcription, and are defined in gon.jar. All these functions from gon.jar are also available in CI 5.0 in both simulation engines: standard and SECG.

CI 4.0 Function ***Pnuts* style**	***CI 5.0 Function Java***	***CI 5.0 Function Js***
transcription or *Transcription::Trans*	*gon.Transcription.Trans*	*Packages.gon.Transcription.Trans*
translation or *Translation::Trans*	*gon.Translation.Trans*	*Packages.gon.Translation.Trans*

Note: The proper function call has to be used for each language – it means the script code has to be updated after changing the script language.

e. Complex/Generic Scripts

Despite similarity between different languages, complex scripts are very dependent on the language used. These differences may lead to compilation or execution problems when changing from one scripting language to another. Many of complex *Pnuts* scripts will fail to compile in *Java*. The table below summarizes the differences between the script languages.

	CI 4.0 Expression *Pnuts* style	**CI 5.0 Expression *Java* Style**	**CI 5.0 Expression *js* Style**
Return Value	Optional	Required	Required
'return' statement	Optional	Required	Not allowed
Type Declaration	Optional	Required	Optional
Imports	Supported	Not Allowed	Supported
Language Specific Operators or Functions	*operator* '::'	*operator* '.'	*operator* '.'

Note: *Simplemath* is not intended for handling complex scripts.

Here are general guidelines for writing/porting complex scripts:

1. Return Value.
   In CI 4.0, in some cases, *Pnuts* scripts did not return any value; their execution did not make any change. Such scripts will cause compilation problems in CI 5.0 and have to be fixed by the user.
   In CI 5.0, scripts must return a value, because the general execution rule is newValue=script(values).

CI 4.0 Expression *Pnuts* style	**CI 5.0 Expression *Java* Style**

if (rand() > 0.5)
{
return 1;
}
	if (rand() > 0.5)
	{
	return 1;
	}
	return m1; // return value required!!

2. 'return' statement.
   In *Pnuts* scripts return statements are often omitted, *e.g. if (m2>m3) {m1=1} else {m1=2};* Such scripts will cause compilation problems in Java. To fix them please use the 'return' statement to point the return value explicitly, e.g. *e.g. if (m2>m3) {return 1} else {return 2};*

CI 4.0 Expression *Pnuts* style	**CI 5.0 Expression *Java* Style**
sin(m2);	**// return statement required!!** return sin(m2);

3. Type Declaration
   In *Pnuts* and *js*, the variable type is not declared in most cases. In *java* you must use full type name (with packages). Therefore, you will always have to define the type for each variable used in the script, when porting the script code from *Pnuts* to *Java*. Alternatively, an entity could be added with the same variable name as the local variable, e.g. ,

CI 4.0 Expression *Pnuts* style	**CI 5.0 Expression *Java* Style**

code = "A";
return code.length();
	// type decelaration required
	String code = "A";
	return code.length;

4. Imports
   Imports cannot be used in Java code. If imports were used in *Pnuts*, they must be removed and the absolute package name has to be used in the code, e.g.:

CI 4.0 Expression ***Pnuts* style**	**CI 5.0 Expression** ***Java* Style**

import("gon.Transcription");
Transcription::Trans("A");
	// import forbidden
	return gon.Transcription.Trans(code);.

5. Operators, e.g. Static functions
   Some advanced operators might be different in each scripting language. In *Pnuts* static functions are accessed by '::' operator, e.g. *Transcription::Trans(code)*. In *Java* these functions are accessed by '.' Operator, e.g. *gon.Transcription.Trans(code*);. Therefore, the operators have to be replaced when changing the scripting language.

CI 4.0 Expression ***Pnuts* style**	**CI 5.0 Expression** ***Java* Style**

import("gon.Transcription");
Transcription::Trans("A");
	// different static function operator
	return gon.Transcription.Trans(code);

5. Appendix A. Definition of Simple Scripts

Simple scripts are scripts that are treated as plain text by the simulation engine. Such scripts are neither compiled nor executed, but the simulation engine replaces the script by a value. Thank to that *simple scripts* are independent from *scripting language* selection, they will always return the same value regardless of the selected language. The simple scripts can be divided into several types, which are defined by *regular expressions*:

Type	**Regular Expression**	**Example**
Number	`"[+-]?([0-9]*\\.?[0-9]+|[0-9]+\\.?[0-9]*)([eE][+-]?[0-9]+)?"`	-98.76 1.234E-56
Boolean	“true” “false”	true false
String	`"\"[^\"]*\""`	“abc”
Variable	`"[_a-zA-Z][_a-zA-Z0-9]*"`	m22

The simple scripts might be surrounded by “return” and “;”. In such a case the program will detect this and neither compile nor execute the script, but replace it with the proper value

Examples of simple scripts surrounded with “return” and “;”.

return 1.23;
return 4.56e-789
true;
return false;
return “abc”
m22;

6. Appendix B. Definition of *Simplemath* Scripts

The *simplemath* language is dedicated for writing math scripts, which will be executed very fast by the simulation engine. A *simplemath* script consist of exactly **one expression** that may include tokens listed in the table below:

***Simplemath* tokens**			
TOKEN	**Description**	**Definition**	**Example**
VARIABLE	Variable Label identifying an entity value	('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*	m1, m2, m3, etc
VALUE	Number	('0'..'9')+('.'('0'..'9')*)?| '.'('0'..'9')+	123.456
OPERATOR	Algebraic operator:	ADDITIVEOP : '+' | '-' MULTIPLICATIVEOP : '*' | '/' | '%'	m1+2 m2*m3 m2/m3 m1%m2
UNARY	Unary sign operator:	ADDITIVEOP : '+' | '-'	-m1 -6.5
COMPARATOR	Comparator operators: equal, not equal, less than, greater than	EQUALOP : '==' | '!=' NOTEQUALOP : '<' | '>' | '<=' | '>='	m1 == m2 m1 >= 2
LOGICAL	Logical operators	OROP : '||' ANDOP : '&&'	m1 > 1 || m2 > 1 m1 > 1 && m2 > 1
FUNC	Functions and Constants, which are defined in enum EFunction in class FuncMathScript	`rand` `random` `gauss` `gaussian` `PI` `E` `sin` `cos` `tan` `asin` `acos` `atan` `ceil` `floor` `round` `exp` `log` `abs` `sgn` `signum` `min` `max`	rand() sin(m1) PI E pow(m1,2) hill(1,2,3,4)

		pow atan2 beta gamma median SMass LSMass LSMass2 hill err	

IF	Question mark operator	QMARK
		  : '?'
		ELSE
		  : ':'
			m1 > 0.5 ? 0 : 1

Parentheses	Brackets)	LPAREN
		  : '('
		  ;
		RPAREN
		  : ')'
		  ;
			(1+m2)/2

WHITESPACE	Spaces	WHITESPACE
		  : ('\t' | ' ' | '\r' | '\n'| '\u000C')+
		    { $channel = 99; }
		  ;	

The precise *simplemath* grammar is shown below. It can be also found in the ifElse.g file in SVN (NCI\trunk\Simulation\src\org\csml\nci\simulation\script\simplemath\gramma).

Simplemath Grammar

```
grammar IfElse;

/*
 * use antlr 3 to generate the parser and grama files
 * from this file, it should generate IfElse.java and IfElseLexer.java
 *
 * With regard to the generated output tree there are two issues:
 * - it generates nodes which are solely used for presedence i.e.:
 *   AddExpresion
 *    |-MultiplyExpresion
 *       |-*
 *       |-SomeExprForParam1
 *       |-SomeExprForParam2
 * Here the AddExpr is empty and only references proper subexpresion
 * - it generates empty 'if' nodes, which don't hold if statements
 * but similarly to the previous issue reference some other nodes
 * The solution is to simply ignore this two cases:
 * if (node.childCount==1) return parse(node.child(0));
 *
 */

options
{
        output = AST;
}

tokens {
    VARIABLE;
    VALUE;
    OPERATOR;
    UNARY;
    COMPARATOR;
    LOGICAL;
    FUNC;
    IF;
}

series
  : expr
  ;

expr
  : or_conditional_expr (QMARK expr ELSE expr )? -> ^(IF or_conditional_expr expr*)
  ;

or_conditional_expr
  : and_conditional_expr (OROP and_conditional_expr)* -> ^(LOGICAL and_conditional_expr (OROP
and_conditional_expr)*)
  ;
and_conditional_expr
  : notequal_conditional_expr (ANDOP notequal_conditional_expr)* -> ^(LOGICAL notequal_conditional_expr
(ANDOP notequal_conditional_expr)*)
  ;

notequal_conditional_expr
  : equal_conditional_expr (NOTEQUALOP equal_conditional_expr)* -> ^(COMPARATOR
equal_conditional_expr (NOTEQUALOP equal_conditional_expr)*)
  ;
equal_conditional_expr
  : additive_expr (EQUALOP additive_expr)* -> ^(COMPARATOR additive_expr (EQUALOP additive_expr)*)
  ;

additive_expr
  : multiplicative_expr (ADDITIVEOP multiplicative_expr)* -> ^(OPERATOR multiplicative_expr (ADDITIVEOP
multiplicative_expr)*)
  ;
multiplicative_expr
  : item (MULTIPLICATIVEOP item)* -> ^(OPERATOR item (MULTIPLICATIVEOP item)*)
  ;

item
  : LPAREN expr RPAREN -> ^(expr)
  | LABEL -> ^(VARIABLE LABEL)
  | LABEL LPAREN (expr (',' expr)*)?  RPAREN -> ^(FUNC LABEL expr*)
  | NUMBER -> ^(VALUE NUMBER)
  | ADDITIVEOP item -> ^(UNARY ADDITIVEOP item)
  ;


COMMA
            :           ','
            ;
LABEL
  : ( 'a'..'z'|'A'..'Z'|'_' )( 'a'..'z'|'A'..'Z'|'_'|'0'..'9' )*
  ;
NUMBER
  : ('0'..'9')+('.'('0'..'9')*)?
  | '.'('0'..'9')+
  ;
NEWLINE
  :   '\r' '\n'   // DOS
  |   '\n'        // UNIX
  ;
ADDITIVEOP
```

```
  : '+'
  | '-'
  ;
MULTIPLICATIVEOP
  : '*'
  | '/'
  | '%'
  ;
NOTEQUALOP
  : '<'
  | '>'
  | '<='
  | '>='
  ;
OROP
  : '||'
  ;
ANDOP
  : '&&'
  ;
EQUALOP
  : '=='
  | '!='
  ;
NOTOP
  : '!'
  ;
LPAREN
  : '('
  ;
RPAREN
  : ')'
  ;
QMARK
  : '?'
  ;
ELSE
  : ':'
  ;
WHITESPACE
  : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+        { $channel = 99; }
  ;
```