The SPiM Language

Version 0.05

Contents

1 Introduction

This document presents the SPiM Language Definition. The Language is defined in BNF notation, where optional elements are enclosed in braces as $\{Optional\}$.

2 Programs

Syntax A *Program* consists of one or more *Declarations*, together with optional top-level *Directives* for sampling and plotting simulation results.

$Program$	::=	$\{Directive_1 \ldots Directive_M\}$	Directives, $M \geq 1$
		$Declaration_1 \ldots Declaration_N$	Declarations, $N \geq 1$
$Directive$	::=	**directive sample** $Float$ $\{Integer\}$	Sample Directive
	|	**directive graph**	Graph Directive
	|	**directive plot** $Point_1 \ldots Point_N$	Plot Directive
$Point$	::=	!$Channel$ {**as** $String$}	Output Point
	|	?$Channel$ {**as** $String$}	Input Point
	|	$Name$ ($Value_1$, ..., $Value_N$) {**as** $String$}	Process Point, $N \geq 0$

Sample Directive

directive sample $Float$ $\{Integer\}$

Specifies the maximum duration of a simulation, together with the maximum number of plots.

The maximum duration of the simulation is specified by the given *Float*. The units of simulation time are not given explicitly, but are determined by the reaction rates in the program. For example, if the rates are given in $seconds^{-1}$ then the simulation time is assumed to be in *seconds*. The simulation is halted when the simulation time exceeds the maximum duration.

The maximum number of plots is specified by the given *Integer*. This is used to compute the minimum interval of time between plots. For example, if the duration is 10.0 and the maximum number of plots is 1000 then there will be a minimum time of 0.01 between plots. This has the effect of sampling plots at regular time intervals. If no maximum number of plots is given then all of the plots are used.

If no sample directive is given then the simulation continues until the program terminates.

Graph Directive

directive graph

Instructs the simulator to output a graphical representation of the program in *.dot* format. The resulting file can be viewed using the Graphviz DOT layout engine[1], which can also be used to export a directed graph in .ps or .png format. The graphical representation corresponds to the program code, where top-level value, type and channel declarations are omitted. The graph directive is disabled for the GUI version of the simulator, which uses a menu command to export a graphical representation of the program that was last simulated.

Plot Directive

directive plot $Point_1 \ldots Point_N$

Specifies one or more points to be recorded at each time step. If no plot directive is given then the number of possible inputs and outputs on all channels is recorded at each time step.

[1]Available from http://www.graphviz.org/ compatible with version 2.8

The simulation results for a given SPiM program are stored in a corresponding *.csv* file as a sequence of comma-separated values. The first line of the file contains the headers for the remaining lines. The first header denotes the simulation time, and each subsequent header denotes the number of possible inputs $?x$ or outputs $!x$ on a given channel x, or the number of running processes $P(n)$ with parameters n. Each subsequent line contains the current simulation time, followed by the number of inputs, outputs or processes. A new line is written to the result file for each simulation step.

The result file can be viewed using a suitable spreadsheet in order to plot a graph of the results. For example, Microsoft Excel can be used to open the result file in order to plot a scatter diagram of selected inputs, outputs and processes over time. The result file can also be imported into Microsoft Excel as external data, which can be refreshed whenever the result file is updated during a simulation.

Points A point can be an output or input on a *Channel* or the *Name* of a process, together with an optional *String* header.

$$!\mathit{Channel}\ \{\textbf{as}\ \mathit{String}\}$$

Records the number of possible outputs on the given *Channel* at each time step, using the given *String* header. If no string header is specified then the header "!*Channel*" is used by default. If more than one channel is declared with the same name then the sum of all the outputs on these channels is recorded.

$$?\mathit{Channel}\ \{\textbf{as}\ \mathit{String}\}$$

Records the number of possible inputs on the given *Channel* at each time step, using the given *String* header. If no string header is specified then the header "?*Channel*" is used by default. If more than one channel is declared with the same name then the sum of all the inputs on these channels is recorded.

$$\mathit{Name}\ (\mathit{Value}_1, \ldots, \mathit{Value}_N)\ \{\textbf{as}\ \mathit{String}\}$$

Records the number of processes with the given *Name* and parameters $(\mathit{Value}_1, \ldots, \mathit{Value}_N)$, using the given *String* header. If no string header is specified then the header "$\mathit{Name}(\mathit{Value}_1, \ldots, \mathit{Value}_N)$" is used by default. Only processes of a certain form can be recorded, i.e. those that are defined as a choice of actions preceded by zero or more channel, type or value declarations. If no parameters are specified then the sum of all the processes with the given *Name* is recorded.

Declarations

$$\mathit{Declaration}_1 \ldots \mathit{Declaration}_N$$

Specifies one or more top-level declarations to be executed.

3 Declarations

Syntax A *Declaration* can be a *Channel*, *Type*, *Value* or *Process* declaration, or one or more mutually recursive *Definitions*.

Declaration	::=	**new** *Name*{@*Value*}:*Type*	Channel Declaration
	|	**type** *Name* = *Type*	Type Declaration
	|	**val** *Pattern* = *Value*	Value Declaration
	|	**run** *Process*	Process Declaration
	|	**let** $\mathit{Definition}_1$ **and** … **and** $\mathit{Definition}_N$	Process Definitions, $N \geq 1$
Definition	::=	$\mathit{Name}\ (\mathit{Pattern}_1, \ldots, \mathit{Pattern}_N) = \mathit{Process}$	Process Definition, $N \geq 0$

Channel Declaration

$$\textbf{new } \mathit{Name}\{\texttt{@}\mathit{Value}\}\texttt{:}\mathit{Type}$$

Creates a new channel with rate *Value* and with the given *Type*, and assigns this channel to the given *Name*. The *Rate* is a floating point number that corresponds to the rate of an interaction on the channel. If no rate is specified then the channel is assumed to have an infinite rate.

Type Declaration

$$\textbf{type } \mathit{Name} \texttt{ = } \mathit{Type}$$

Assigns the given *Type* to the given *Name*. If the *Name* occurs in the *Type* then a recursive type is declared.

Value Declaration

$$\textbf{val } \mathit{Pattern} \texttt{ = } \mathit{Value}$$

Assigns the given *Value* to the given *Pattern*. If the *Pattern* is a *Name* then a single value is declared. If the *Pattern* is a tuple of N patterns $\mathit{Pattern}_1\texttt{,}\ldots\texttt{,}\mathit{Pattern}_N$ then N values are declared.

Process Declaration

$$\textbf{run } \mathit{Process}$$

Executes the given *Process*.

Process Definitions

$$\textbf{let } \mathit{Definition}_1 \textbf{ and } \ldots \textbf{ and } \mathit{Definition}_N$$

Declares one or more mutually recursive processes.

Process Definition

$$\mathit{Name}\ \texttt{(}\mathit{Pattern}_1\texttt{,}\ldots\texttt{,}\mathit{Pattern}_N\texttt{)} \texttt{ = } \mathit{Process}$$

Assigns the given *Process* to the given *Name*, parameterised by zero or more *Pattern* arguments. An instance of the given *Process* can be executed by invoking the given *Name* with corresponding *Value* arguments.

4 Processes

Syntax A *Process* can be Null, a Parallel Composition of processes, an Action, a Choice of actions, an Instance of a definition, a Replicated action, a Conditional process, a Pattern Matching process, a Repeated process or a collection of nested Declarations.

Process	::=	$\texttt{()}$	Null Process
	|	$\texttt{(}\mathit{Process}_1 \texttt{ \| } \ldots \texttt{ \| } \mathit{Process}_M\texttt{)}$	Parallel, $M \geq 2$
	|	$\mathit{Action}\{\texttt{;}\ \mathit{Process}\}$	Action Process
	|	$\textbf{do } \mathit{Action}_1\{\texttt{;}\mathit{Process}_1\} \texttt{ or} \ldots \texttt{or } \mathit{Action}_M\{\texttt{;}\mathit{Process}_M\}$	Choice, $M \geq 2$
	|	$\mathit{Name}\ \texttt{(}\mathit{Value}_1\texttt{,}\ldots\texttt{,}\mathit{Value}_N\texttt{)}\{\texttt{;}\ \mathit{Process}\}$	Instantiation, $N \geq 0$
	|	$\textbf{replicate } \mathit{Action}\{\texttt{;}\ \mathit{Process}\}$	Replicated Action
	|	$\textbf{if } \mathit{Value} \textbf{ then } \mathit{Process}\ \{\textbf{else } \mathit{Process}\}$	Conditional Process
	|	$\textbf{match } \mathit{Value} \textbf{ case } \mathit{Case}_1 \ldots \textbf{case } \mathit{Case}_N$	Matching, $N \geq 1$
	|	$\mathit{Integer} \textbf{ of } \mathit{Process}$	Repetition
	|	$\texttt{(}\mathit{Declaration}_1 \ldots \mathit{Declaration}_N\ \mathit{Process}\texttt{)}$	Nested Declarations, $N \geq 0$
Case	::=	$\mathit{Value} \texttt{ -> } \mathit{Process}$	Match Case

Null

$$()$$

Terminates the execution of a process.

Parallel Composition

$$(Process_1 \mid \ldots \mid Process_M)$$

Executes two or more processes in parallel.

Action Process

$$Action\{;\ Process\}$$

Tries to perform the given *Action* and then execute the given *Process*. If no *Process* is specified then nothing is executed after the action is performed.

Choice

$$\textbf{do}\ Action_1\{;Process_1\}\ \texttt{or} \ldots \texttt{or}\ Action_M\{;Process_M\}$$

Tries to perform two or more competing actions simultaneously. Once a chosen $Action_i$ has been performed, any competing actions are discarded and $Process_i$ is executed. If no $Process_i$ is specified then nothing is executed after the action is performed.

Instantiation

$$Name\ (Value_1, \ldots, Value_N)\{;\ Process\}$$

Spawns a copy of the process defined by the given *Name*, instantiated with the given *Value* arguments, in parallel with the given *Process*. If no *Process* is specified then nothing is executed in parallel with the instantiated process. From version 0.042, the use of the optional $\{;\ Process\}$ is deprecated.

A number of predefined processes are available: $print(s)$ prints the given string s on the console, $println(s)$ prints the given string s on the console followed by a new line character and $break()$ halts the simulation until the user presses Enter.

Replicated Action

$$\textbf{replicate}\ Action\{;\ Process\}$$

Tries to perform the given *Action* and then spawn a copy of the given *Process*. This has the effect of repeatedly executing the given *Action* followed by the given *Process*.

Conditional

$$\textbf{if}\ Value\ \textbf{then}\ Process\ \{\textbf{else}\ Process\}$$

Executes the first *Process* if the given *Value* is true. Otherwise, the optional second process is executed.

Matching

$$\textbf{match}\ Value\ \textbf{case}\ Case_1 \ldots \textbf{case}\ Case_N$$

Tries to match the given *Value* with one or more *Cases*, where each $Case_i$ is of the form:

$$Value_i\ \texttt{->}\ Process_i$$

The cases are matched in order, from first to last. For each $Case_i$, if the given *Value* matches $Value_i$ then $Process_i$ is executed. Otherwise, the next case is examined. Note that any variables in $Value_i$ are bound during matching. If none of the cases match the given *Value* then nothing is executed.

Repetition

$$Integer \ \textbf{of} \ Process$$

Executes zero or more copies of the given *Process*, as specified by the given *Integer*.

Nested Declarations

$$(Declaration_1 \ldots Declaration_N \ Process)$$

Executes zero or more nested declarations, followed by the given *Process*. The syntax is further constrained so that nested declarations cannot contain process definitions.

5 Actions

Syntax An *Action* can be a stochastic Delay or an Output or Input on a *Channel*.

Action	::=	`delay@`$Value$	Delay
	|	$!Channel \ \{(Value_1, \ldots, Value_N)\} \ \{*Value\}$	Output, $N \geq 0$
	|	$?Channel \ \{(Pattern_1, \ldots, Pattern_N)\} \ \{*Value\}$	Input, $N \geq 0$

Delay

$$\texttt{delay@}Value\texttt{;} \ Process$$

Waits for a period of time stochastically determined by the given *Value*, and then executes the given *Process*.

Output

$$!Channel \ \{(Value_1, \ldots, Value_N)\} \ \{*Value\}; \ Process$$

Tries to send zero or more values on the given *Channel* and then execute the given *Process*. If there is a parallel input on the same channel then the values are sent over the *Channel* and the *Process* is executed. The rate of the reaction is multiplied by the given *Value*.

Input

$$?Channel \ \{(Pattern_1, \ldots, Pattern_N)\} \ \{*Value\}; \ Process$$

Tries to receive zero or more values on the given *Channel*, assign them to the given patterns and then execute the given *Process*. If there is a parallel output on the same channel then values are received over the *Channel* and assigned to the patterns, and the *Process* is executed. The rate of the reaction is multiplied by the given *Value*.

6 Patterns

Syntax A *Pattern* can be a Wildcard, a Name with an optional type annotation, or a sequence of zero or more patterns enclosed in parentheses:

Pattern	::=	_	Wildcard Pattern
	|	$Name\{:Type\}$	Name Pattern
	|	$(Pattern_1, \ldots, Pattern_N)$	Patterns, $N \geq 0$

Assignment A *Value* can be assigned to a given *Pattern* inside a given *Process*, written

$$Process\{Pattern{:=}Value\}$$

7 Values

Syntax A $Value$ can be a Constant, a Constructed value or an Expression:

$Value$	::=	$String$	String Value
	|	$Integer$	Integer Value
	|	$Float$	Float Value
	|	$Character$	Character Value
	|	`true`	Boolean True
	|	`false`	Boolean False
	|	`int_of_float`	Float to Integer
	|	`float_of_int`	Integer to Float
	|	`sqrt`	Square Root
	|	$Name$	Variable
	|	`show` $Value$	String Representation
	|	`-`$Value$	Negation
	|	$Value$`+`$Value$	Addition
	|	$Value$`-`$Value$	Subtraction
	|	$Value$`*`$Value$	Multiplication
	|	$Value$`/`$Value$	Division
	|	$Value$`=`$Value$	Equal
	|	$Value$`<>`$Value$	Different
	|	$Value$`<`$Value$	Less Than
	|	$Value$`>`$Value$	Greater Than
	|	$Value$`<=`$Value$	Less Than or Equal
	|	$Value$`>=`$Value$	Greater Than or Equal
	|	$Name$ `(`$Value_1$`,` ... `,`$Value_N$`)`	Constructor Value, $N \geq 0$
	|	`[]`	Empty List
	|	$Value$`::`$Value$	List Value
	|	`(`$Value_1$`,` ... `,`$Value_N$`)`	Values, $N \geq 0$

Constant Values A $String$, $Integer$, $Float$ or $Character$ constant, boolean `true`, or boolean `false`.

Constructed Values A sequence of zero or more values, enclosed in parentheses:

$$(Value_1, \ldots, Value_N)$$

A data constructor consisting of a $Name$ and a sequence of zero or more $Value$ arguments:

$$Name\ (Value_1, \ldots, Value_N)$$

A list, which can be either empty [] or of the form $Value_1$`::`$Value_2$, where $Value_1$ is the first element of the list and $Value_2$ is the remainder of the list. Note that all values in a list must be of the same type:

$$Value::Value$$

Expressions A value expression can be a variable $Name$ representing a predefined value, a prefix operator followed by a $Value$ argument, or an infix operator between two $Value$ arguments.

The prefix operator `show` $Value_1$ converts $Value_1$ to a string value. By definition, every value has a corresponding string representation. The prefix operators `int_of_float` and `float_of_int` perform float and integer conversions, respectively. The prefix operator `-` $Value$ is defined in Table 1.

Infix operators take two arguments of any type, provided both types are the same. The comparison operators $(=, <>, >, >=, <=)$ return a result of boolean type and rely on an ordering to compare both arguments. The arithmetic operators (+,-,*,/) return a result of the same type as their arguments. Table 1 describes the behaviour of the operators for each corresponding type of arguments. The _ symbol means that the behaviour of the operator is unspecified, although the result will always be of the correct type.

Type	+	-	*	/	- (prefix)	= ,<>,>,>=,<=
String	Concatenate	_	_	_	_	Lexicographic Order
Integer	Add	Subtract	Multiply	Divide	Minus	Integer Order
Float	Add	Subtract	Multiply	Divide	Minus	Float Order
Character	_	_	_	_	_	ASCII Code Order
Boolean	Or	_	And	_	Not	Lexicographic Order
List	Append	_	_	_	_	Order of Elements
Data	_	_	_	_	_	Lexicographic Order
Other	_	_	_	_	_	_

Table 1: Operator Definitions

8 Types

Syntax A *Type* can be a Constant type, a Constructed type or a type Expression:

Type	::=	`string`	String Type
	|	`int`	Integer Type
	|	`float`	Float Type
	|	`char`	Character Type
	|	`bool`	Boolean Type
	|	*Name*	Type Variable
	|	`'`*Name*	Polymorphic Type
	|	`chan`{ `(`$Type_1$`,` ... `,`$Type_N$`)`}	Channel Type, $N \geq 0$
	|	`proc` `(`$Type_1$`,` ... `,`$Type_N$`)`	Process Type, $N \geq 0$
	|	*Name* `(`$Type_1$`,` ... `,`$Type_N$`)`	Constructor Type, $N \geq 0$
	|	$Type_1$ `|` ... `|` $Type_M$	Data Type, $M \geq 2$
	|	`list(`*Type*`)`	List Type
	|	`(`$Type_1$`,` ... `,`$Type_N$`)`	Types, $N \geq 0$

Constant Types A *string*, *int*, *float*, *char* or *bool* type.

Constructed Types A channel that can carry zero or more values of given types:

`chan`{ `(`$Type_1$`,` ... `,`$Type_N$`)`}

A process that can be instantiated with zero or more values of given types:

`proc` `(`$Type_1$`,` ... `,`$Type_N$`)`

A sequence of zero or more types, enclosed in parentheses:

`(`$Type_1$`,` ... `,`$Type_N$`)`

A data constructor consisting of a *Name* and a sequence of zero or more arguments of given types:

Name `(`$Type_1$`,` ... `,`$Type_N$`)`

A data type consisting of a choice between two or more types:

$$Type_1 \mid \ldots \mid Type_M$$

A *list* that can contain values of a given type:

$$\texttt{list}(Type)$$

Type Expressions A type expression can be a *Name* representing a predefined type, or a polymorphic type variable that can be instantiated with an arbitrary type.

Type-checking Before executing a given program, the simulator checks if the program is well-typed and reports any type errors. The type system for the SPiM Language is based on the type system for the Pict Language, available from http://www.cis.upenn.edu/~bcpierce/papers/pict/

9 Lexical Syntax

Regular Expressions Regular Expressions (*Regexp*) are used to describe the syntax of Constants and Variables in the SPiM Language:

Regexp	::=	c	Character
	|	c···c	Character Range
	|	¬c	Character Complement
	|	*Regexp Regexp*	Concatenation of Expressions
	|	*Regexp* | *Regexp*	Alternative Expressions
	|	$Regexp^?$	Optional Expression
	|	$Regexp^*$	Repetition of Expression
	|	$Regexp^+$	Strict Repetition of Expression
	|	(*Regexp*)	Nested Expression

Constants An *Integer* constant consists of an optional negative sign followed by one or more digits:

$$Integer ::= (\text{-})^?(0\cdots9)^+$$

A *String* constant consists of a sequence of zero or more characters enclosed in double quotes. The sequence can only contain a double quote if it is preceded by a backslash:

$$String ::= \text{"}((\neg\text{"}) \mid (\backslash\text{"}))^*\text{"}$$

A *Float* constant consists of an *Integer*, followed by a decimal point and one or more digits, followed by an optional exponent. The exponent consists of e or E, followed by $+$ or $-$, followed by one or more digits:

$$Float ::= Integer.(0\cdots9)^+((\text{e} \mid \text{E})(+ \mid \text{-})(0\cdots9)^+)^?$$

A *Character* constant consists of any character enclosed in single quotes, apart from the single quote character. It can also consist of a backslash, followed by a special *escaped* character or a three-digit decimal number, enclosed in single quotes:

Character ::=		'(¬')'	Regular Character
	|	'\''	Single Quote
	|	'\\'	Backslash
	|	'\n'	Linefeed
	|	'\r'	Carriage Return
	|	'\t'	Horizontal Tabulation
	|	'\b'	Backspace
	|	'\(0···9)(0···9)(0···9)'	ASCII Character Code

Variables A *Name* variable consists of a letter followed by zero or more letters, digits, underscores or single quotes:

$$Name ::= \quad (\mathrm{A}\cdots\mathrm{Z} \mid \mathrm{a}\cdots\mathrm{z})(\mathrm{A}\cdots\mathrm{Z} \mid \mathrm{a}\cdots\mathrm{z} \mid 0\cdots9 \mid \_ \mid ')^*$$

A *Channel* variable is a *Name* representing a *Channel* value:

$$Channel ::= \quad Name$$

The following variable names are reserved keywords of the language:

`and`	`as`	`bool`	`chan`	`char`	`delay`	`directive`
`do`	`else`	`float`	`float_to_int`	`if`	`in`	`int`
`int_to_float`	`false`	`let`	`list`	`new`	`out`	`or`
`of`	`plot`	`proc`	`replicate`	`run`	`sample`	`show`
`sqrt`	`string`	`then`	`true`	`type`	`val`	

Comments A comment starts with the sequence of characters (* and ends with the sequence of characters *). Comments can be nested, but they cannot occur inside single or double quotes.

10 Language Summary

This section presents a summary of the SPiM language definition, where optional elements are enclosed in braces as {*Optional*}. The syntax is further constrained so that nested declarations cannot contain process definitions.

Program	::=	$\{Directive_1 \ldots Directive_M\}$	Directives, $M \geq 1$
		$Declaration_1 \ldots Declaration_N$	Declarations, $N \geq 1$
Directive	::=	**directive sample** *Float* {*Integer*}	Sample Directive
	$\mid$	**directive graph**	Graph Directive
	$\mid$	**directive plot** $Point_1 \ldots Point_N$	Plot Directive
Point	::=	`!`*Channel* {**as** *String*}	Output Point
	$\mid$	`?`*Channel* {**as** *String*}	Input Point
	$\mid$	*Name* `(`$Value_1$`,`$\ldots$`,`$Value_N$`)` {**as** *String*}	Process Point, $N \geq 0$
Declaration	::=	**new** *Name*{`@`*Value*}`:`*Type*	Channel Declaration
	$\mid$	**type** *Name* `=` *Type*	Type Declaration
	$\mid$	**val** *Pattern* `=` *Value*	Value Declaration
	$\mid$	**run** *Process*	Process Declaration
	$\mid$	**let** $Definition_1$ **and** $\ldots$ **and** $Definition_N$	Definitions, $N \geq 1$
Definition	::=	*Name* `(`$Pattern_1$`,`$\ldots$`,`$Pattern_N$`)` `=` *Process*	Definition, $N \geq 0$
Process	::=	`()`	Null Process
	$\mid$	`(`$Process_1$ $\mid$ $\ldots$ $\mid$ $Process_M$`)`	Parallel, $M \geq 2$
	$\mid$	*Name* `(`$Value_1$`,`$\ldots$`,`$Value_N$`)`{`;` *Process*}	Instantiation, $N \geq 0$
	$\mid$	*ActionProcess*	Action Process
	$\mid$	**do** $ActionProcess_1$ `or` $\ldots$ `or` $ActionProcess_M$	Choice, $M \geq 2$
	$\mid$	**replicate** *ActionProcess*	Replicated Action
	$\mid$	**if** *Value* **then** *Process* {**else** *Process*}	Conditional Process
	$\mid$	**match** *Value* **case** $Case_1 \ldots$ **case** $Case_N$	Matching, $N \geq 1$
	$\mid$	*Integer* **of** *Process*	Repetition
	$\mid$	`(`$Declaration_1 \ldots Declaration_N$ *Process*`)`	Nested Declarations, $N \geq 0$
Case	::=	*Value* `->` *Process*	Match Case
ActionProcess	::=	*Action*{`;` *Process*}	Action Process
Action	::=	`delay@`*Value*	Delay
	$\mid$	`!`*Channel* {`(`$Value_1$`,`$\ldots$`,`$Value_N$`)`} {`*`*Value*}	Output, $N \geq 0$
	$\mid$	`?`*Channel* {`(`$Pattern_1$`,`$\ldots$`,`$Pattern_N$`)`} {`*`*Value*}	Input, $N \geq 0$
Pattern	::=	`_`	Wildcard Pattern
	$\mid$	*Name*{`:`*Type*}	Name Pattern
	$\mid$	`(`$Pattern_1$`,`$\ldots$`,`$Pattern_N$`)`	Patterns, $N \geq 0$

Value	::=	*String*	String Value
	$\mid$	*Integer*	Integer Value
	$\mid$	*Float*	Float Value
	$\mid$	*Character*	Character Value
	$\mid$	true	Boolean True
	$\mid$	false	Boolean False
	$\mid$	int_ of_ float	Float to Integer
	$\mid$	float_ of_ int	Integer to Float
	$\mid$	sqrt	Square Root
	$\mid$	*Name*	Variable
	$\mid$	show *Value*	String Representation
	$\mid$	-*Value*	Negation
	$\mid$	*Value*+*Value*	Addition
	$\mid$	*Value*-*Value*	Subtraction
	$\mid$	*Value***Value*	Multiplication
	$\mid$	*Value*/*Value*	Division
	$\mid$	*Value*=*Value*	Equal
	$\mid$	$Value<>Value$	Different
	$\mid$	$Value<Value$	Less Than
	$\mid$	$Value>Value$	Greater Than
	$\mid$	$Value<=Value$	Less Than or Equal
	$\mid$	$Value>=Value$	Greater Than or Equal
	$\mid$	*Name* $(Value_1, \ldots, Value_N)$	Constructor Value, $N \geq 0$
	$\mid$	[]	Empty List
	$\mid$	*Value*::*Value*	List Value
	$\mid$	$(Value_1, \ldots, Value_N)$	Values, $N \geq 0$
Type	::=	string	String Type
	$\mid$	int	Integer Type
	$\mid$	float	Float Type
	$\mid$	char	Character Type
	$\mid$	bool	Boolean Type
	$\mid$	*Name*	Type Variable
	$\mid$	'*Name*	Polymorphic Type
	$\mid$	chan{ $(Type_1, \ldots, Type_N)$}	Channel Type, $N \geq 0$
	$\mid$	proc $(Type_1, \ldots, Type_N)$	Process Type, $N \geq 0$
	$\mid$	*Name* $(Type_1, \ldots, Type_N)$	Constructor Type, $N \geq 0$
	$\mid$	$Type_1 \mid \ldots \mid Type_M$	Data Type, $M \geq 2$
	$\mid$	list(*Type*)	List Type
	$\mid$	$(Type_1, \ldots, Type_N)$	Types, $N \geq 0$

Regexp ::= c — Character
| c···c — Character Range
| ¬c — Character Complement
| *Regexp* *Regexp* — Concatenation of Expressions
| *Regexp* | *Regexp* — Alternative Expressions
| *Regexp*? — Optional Expression
| *Regexp** — Repetition of Expression
| *Regexp*+ — Strict Repetition of Expression
| (*Regexp*) — Nested Expression

Character ::= '(¬')' — Regular Character
| '\'' — Single Quote
| '\\' — Backslash
| '\n' — Linefeed
| '\r' — Carriage Return
| '\t' — Horizontal Tabulation
| '\b' — Backspace
| '\(0···9)(0···9)(0···9)' — ASCII Character Code

Channel ::= *Name*
Integer ::= (-)?(0···9)+
String ::= "((¬") | (\"))*"
Float ::= *Integer*.(0···9)+((e | E)(+ | -)(0···9)+)?
Name ::= (A···Z | a···z)(A···Z | a···z | 0···9 | _ | ')*
Keywords ::=

`and`	`as`	`bool`	`chan`	`char`	`delay`	`directive`
`do`	`else`	`float`	`float_to_int`	`if`	`in`	`int`
`int_to_float`	`false`	`let`	`list`	`new`	`out`	`or`
`of`	`plot`	`proc`	`replicate`	`run`	`sample`	`show`
`sqrt`	`string`	`then`	`true`	`type`	`val`	

A comment starts with the sequence of characters (* and ends with the sequence of characters *). Comments can be nested, but they cannot occur inside single or double quotes.