Lola-2: A Logic Description Language

N. Wirth, 24.4.1994 / 14.4.2015

Lola is a notation (language) for specifying digital circuits (logic). In many ways it resembles a programming language. However, Lola texts describe static circuits rather than dynamic processes. Objects occurring in a description can be variables representing signals or registers. Their values are defined as expressions of other objects and operators representing gates.

0. Vocabulary

The vocabulary of Lola consists of special characters (or pairs), and so-called *reserved words* which cannot be chosen as identifiers.

Special characters:

```
~ & | ^ + - * = # < <= > >=
( ) [ ] { } -> . , ; : := ' !
```

Reserved words:

BEGIN CONST END IN INOUT MODULE OUT REG TYPE VAR

1. Identifiers, integers, and comments

Identifiers are used to denote constants, variables, and types.

```
identifier =    letter {letter | digit}.
integer =       digit {digit} ["H"].
```

Examples: Lola begin 100 100H 0FFH

Capital and lower case letters are considered as distinct. If an integer is followed by the capital letter H it is in hexadecimal form with digits 0, 1, ..., 9, and A ... F.

Comments are sequences of characters enclosed by the brackets (* and *), and they may occur between any two symbols within a Lola text.

2. Simple types and array types

Every variable in Lola has a type. The elementary type is BIT (BInary digiT), and its variables assume the values 0 or 1.

```
type =                  {"[" expression "]"} SimpleType.
SimpleType =            identifier |."MODULE" unit ";" identifier.
TypeDeclaration =       identifier "=" type .
```

An array type is specified by the form *[n]T*, where *T* is the type of the elements, and the integer *n* specifies the number of elements. In expressions, individual elements are designated by an index. Indices range from 0 to the array's length minus 1.

Examples: [10] BIT [8][16] BIT

The identifiers BIT, BYTE and WORD are predeclared. The latter denote arrays of 8 and 32 bits respectively.

3. Constant declarations

Constant declarations serve to introduce identifiers denoting a constant, numeric value.

```
ConstDeclaration  = identifier "=" integer ";".
```

4. Variable and register declarations

Variable declarations introduce registers and variables and associate a type with them. All variables declared in an identifier list have the same type. Variables of type *[n] BIT* are called *bitstrings*.

varlist = identifier {"," identifier} ":" type.

Examples: x, y: BIT a: [32] BIT R: BIT Q (clk50): [4] BIT

If a register declaration contains an expression, this denotes the register's clock. The default is a variable *clk*, which must be declared (see also Section 7).

5. Expressions

Expressions serve to combine variables with operators to define new values. The operators are logical negation, conjunction, disjunction, and difference, or arithmetic sum and difference. Elements of an array are selected by an index. (e.g. a.5, a[10]).

|	logical disjunction (or)
^	logical difference (exclusive or)
&	logical conjunction (and)
~	logical negation (not)
+	arithmetic sum
-	arithmetic difference

Operands are registers, variables, and constants. Whereas variables denote the value given by the assigned expression, registers denote the value of the expression in the preceding clock cycle. Registers are instrumental in sequential circuits. The operands of diadic operators must be of the same type.

selector =	{"." factor | "[" expression [":" expression] "]"}.
element =	expression ["!" integer].
constructor =	"{" element {"," element} "}".
factor =	identifier selector | integer | "~" factor | constructor | "(" expression ")".
term =	factor {"&" factor}.
simpleExpr =	["+" | "-"] term {("|" | "^" | "+" | "-") term}.
uncondExpr =	simpleExpr [("=" | "#" | "<" | "<=" | ">" | ">=") simpleExpr].
expression =	uncondExpr ["->" expression ":" expression].

The form *a[m : n]* denotes the *range* of indexed elements *a[m], ... , a[n].*

Constructors denote bitstrings and are sequences of elements. The length (number of bits) of every element must be known. This length is specified in a declaration, and in this case of constants by an explicit integer. For example 10'8 denotes the number 10 represented by 8 bits. The length of the bitstring is the sum of the lengths of its elements. Also, an element can be followed by a replication factor of the form *!n*.

A conditional expression of the form *z := cond -> x : y* expresses represents a multiplexer. *Cond* must be of type BIT. If *cond* yields 1, *z* is equal to *x*, otherwise to *y*.

Examples:		
	x count 1 100 a.4 a[20]	operands
	a[15 : 8]	range
	{u, a.4, a[25:20], 0'8, 15'4}	constructor (20 bit)
	(x & y) | (z & w) (m + n) + 10	simple expressions
	a.5 -> b : c	expression

6. Assignments and statements

Assignments serve to define a register's or a variable's value, which is specified by an expression. An assignment must be understood as a variable's definition (in contrast to an identifier's declaration). In an assignment *v := x, v* and *x* do not have the same roles, and this asymmetry is

emphasized by the use of the symbol := instead of the symmetric equal sign. *v* and *x* must be of the same type.

assignment = operand ":=" expression..

Examples: x := y & z a := {x, y, z} R := rst -> 0 : enb -> x : R

Every variable and register can be assigned in only one single assignment statement, and the assignment must be to the entire variable (not to elements). The only exception is the indexed assignment to an array of registers (e.g. R[n] := x).

For example, instead of

a.0 := x; a.1 := x+y; a.2 := x-y; a.3 := x*y

the single assignment

a := {x, x+y, x-y, x*y)

must be used.

Statements are either assignments or instantiations of modules (see next section).

statement = [assignment | instantiation].
StatementSequence = statement {";" statement}.

7. Modules

A module specifies constants, types, variables, registers, and assignments to variables and registers. Modules are specified as types, and variables can be declared of this type. This implies that modules can be replicated.

```
ModuleType = "MODULE" ["*"] unit ";".
paramlist = ("IN" | "OUT" | "INOUT") varlist.
unit = "(" paramlist {";" paramlist} ")" ("^" | ";"
    ["CONST" {ConstDeclaration}]
    ["TYPE" {TypeDeclaration}]
    {("VAR" | "REG" "(" expression ")") {varlist ";"}}
    ["BEGIN" StatementSequence] "END").
```

The expression after the symbol REG specifies the register's clock.

Example of a module type:

```
TYPE Counter = MODULE (IN clk, rst, enb: BIT; OUT data: WORD);
  REG (clk) R: Word;
BEGIN data := R;
  R := ~rst -> 0 : enb -> R + 1 : R
END Counter
```

Given a variable of this type (*cnt: Counter*), its instantiation could be

cnt (clock, reset, enable, val)

where *clock, reset, enable* and *data* are variables called *actual parameters.* They must be expressions of the corresponding types.

instantiation = identifier selector "(" expression {"," expression} ")".

A "main program" has the form

module = "MODULE" identifier unit identifier ".".

Evidently, a "main program" is the combined declaration of an (anonymous) module type and of a single instance. The identifier at the end of the module's declaration must be the same as the one following the symbol MODULE.