LCSL Reference Manual

Sachin Nene, Chaue Shen, Bogdan Caprita, Julian Maller

1.1 Lexical Conventions

1.1.1 Comments
Comments begin with “(*” and end with “*)”

1.1.2 Variables
Variables consist solely of alphanumeric characters and underscores. The first character must always be an uppercase letter (A-Z) and the last character can never be an underscore (“_”).

1.1.3 Keywords
All keywords start with lowercase letters in order to avoid any overlap with variable names:

```
true           false          fileloc
comp           end            component
if             then           else
ef             head           tail
length         width
```

1.1.4 Operators
Operators in LCSL include:

```
.      @      `      !      ~      **     *      /      %      +
-      <<     >>     <      >      <=     >=     #      ::     ++
\      &      ^      |      &&     ||     ?  :   <-
```

1.2 Types

1.2.1 Fundamental Data Types
The following list includes all fundamental data types in LCSL:
Integers- arbitrary precision and can be expressed in decimal, binary, octal, and hex form
Floats- differentiated with integers by using decimal point; can also have exponent using “e”
Booleans- take on keyword values true or false

1.2.2 Lists and Strings
Lists are data structures consisting of fundamental or more complex data types (including other lists). The elements of a list are separated by whitespace and enclosed within ‘[‘ and ‘]’.

Strings are implemented in LCSL as lists of ASCII-encoded integers.

1.2.3 Components
Components are similar to functions in other languages that can pass into and return from other components. This is a primary feature of LCSL and is a characteristic of it’s functional programming nature. Components can either be declared without a name and using the comp and end keywords, or they can be declared with a name using the component keyword. Parameters passed to and from a component are known as ports. The ports of a component are preceded with either a ‘+’ or ‘-‘ which mean an input or output port respectively. The inputs must be introduced before any output port.

1.2.4 Systems

Systems represent an actual instance of a component. A variable becomes a `System` type when it is connected to an instance of a component with specific input and output port values given.

1.2.5 Vectors

Vectors are similar to bit-vectors in hardware design languages and consist of basic logical bit values (0 and 1). Vectors are created in two different ways. First, the vector can be created through conversion of an integer using the `VectorConst` component. Vectors can also be created by writing them directly between single quotes and connecting this string to a `Vector` variable. All operations involving vectors must use two vectors with the exact same width.

1.3 Type Operators

1.3.1 Integer Operators

The following is a table for operations of integers.

Operator	Function
`- X`	bitwise negation
`X ** Y`	power
`X * Y`	multiplication
`X / Y`	division
`X % Y`	modulo
`X + Y`	addition
`X - Y`	subtraction
`X << Y`	shift left
`X >> Y`	shift right
`X < Y`	less than
`X > Y`	greater than
`X <= Y`	less than or equal
`X >= Y`	greater than or equal
`X == Y`	equal
`X != Y`	not equal
`X & Y`	bitwise AND
`X ^ Y`	bitwise XOR
`X | Y`	bitwise OR

1.3.2 Float Operators

The following is a table for operations of floats.

Operator	Function
`X ** Y`	power
`X * Y`	multiplication
`X / Y`	division
`X + Y`	addition
`X - Y`	subtraction
`X < Y`	less than
`X > Y`	greater than
`X <= Y`	less than or equal
`X >= Y`	greater than or equal
`X == Y`	equal
`X != Y`	not equal

1.3.3 Boolean Operators

The following is a table for operations of Booleans.

Operator	Function
! X	logical NOT
X && Y	short circuit logical AND
X || Y	short circuit logical OR
X ? expr1 : expr2	conditional

1.3.4 List Operators

The following is a table for operations of lists.

Operator	Function
length X	list length
X # <int>	repeated list concatenation (<int> times)
<elements> :: []	list construction
X ++ Y	list concatenation
X \ Y	list concatenation with new-line

1.3.5 Vector Operators

The following is a table for operations of vectors.

Operator	Function
X ` <intList>	bit selection
~ X	bitwise negation
width X	vector width
msb X	most significant bit
msbs X	all bits except LSB
lsb X	less significant bit
lsbs X	all bits except MSB
X * Y	unsigned multiplication
X *+ Y	signed multiplication
X + Y	addition
X - Y	subtraction
X << <int>	shift left
X >> <int>	shift right
X < Y	unsigned less than
X > Y	unsigned greater than
X <= Y	unsigned less than or equal
X >= Y	unsigned greater than or equal
X <+ Y	signed less than
X >+ Y	signed greater than
X <=+ Y	signed less than or equal
X >=+ Y	signed greater than or equal
X # <int>	repeated vector concatenation
X ++ Y	vector concatenation
X == Y	equal

1.4 Expressions

1.4.1 Expression Syntax

Expressions in LSCL are based on the conceptual notion of source expressions and sink expressions. Source expressions produce values whereas sink expressions consume values..

1.4.2 Basic Expressions

1.4.2.1 Constants
Constants are the most fundamental of values and can be of any of the fundamental data type forms (integer, float, Boolean).

1.4.2.2 Variables
Variables are bounded to a specific (possibly dynamic) value based on another expression.

1.4.2.3 Connections
Connections bound variables to other expressions using the '`<-`' operation.

1.4.2.4 Instantiations
Instantiations create instances of components with certain input and output port values based on other expressions.

1.4.2.5 Conditionals
Expression conditionals use the "`<predicate> ? <expr1> : <expr2>`" where `<predicate>` is a source expression. The expression returns `<expr1>` if `<predicate>` is true and `<expr2>` if it is false.

1.4.2.6 Operational expressions
Depending on the data type of the value of variables and sub-expressions used, there are several operations that create expressions. Short circuit logical AND (`&&`) and OR (`||`) can be used with several sub-expressions. The prefix unary operator `!` and infix binary operators like '`+`' are also ways to create expressions.

1.4.3 Source Expressions
As mentioned above, source expressions produce values. The following are valid source expressions:

Variables
Constants (integer, Boolean, or float)
Lists
Connections
Instantiations
Conditionals
Anonymous component definitions
Operational expressions
expression groupings of any combination of above

1.4.3 Sink Expressions
As mentioned above, sink expressions consume values. The following are valid sink expressions:

Variables
Instantiations
Connections

1.5 Statements

1.5.1 Component Definitions
Component definitions both define a component and bound it to a variable (the component name) using the `component` keyword. Anonymous component definitions use the `comp` keyword and are not necessarily bounded to any variable. For instance, a normal component looks like the following:

```
component True +X -X
```

```
  X ? X <- true : X <- false
end
```

1.5.2 Connections
Connections using the '<-' , defined above as a type of expression, are also simple statements.

1.5.3 Conditionals
Conditionals as statements provide the same functionality as if they were basic expressions but also provide extra functionality through the `else` and `ef` keywords. For instance:

```
if (<predicate>)
  (* other statements *)
ef (<otherPredicate>)
  (* even more statements *)
else
  (* default statements *)
end
```

1.6 Syntax Grammar

```
statements:
  EMPTY |
  statements statement

statement:
  ifelse |
  component_named |
  instantiation_system |
  connection

ifelse:
  "if" expression ifelse_then statements ifelse_else |
  "if" expression ifelse_then namespace_sugar ifelse_else

ifelse_then:
  EMPTY |
  "then"

ifelse_else:
  "end" |
  "else" statements "end" |
  "else" namespace_sugar "end" |
  "ef" expression ifelse_then statements ifelse_else |
  "ef" expression ifelse_then namespace_sugar ifelse_else

component_named:
 "component" name component_inputs component_outputs statements "end" |
 "component" name component_inputs component_outputs "end"

component_inputs:
  EMPTY |
  component_inputs component_inputs "+" name

component_outputs:
  EMPTY |
  component_outputs component_outputs "-" name

STRING:
   "`" ( ("a”..”z”) | ("A" .. "Z") | ("0" .. "9") | "_"

INTEGERorVECTOR:
   digits |
   “0x” + digits |
   “1x” + digits |
   “0b” + digits |
   “1b” + digits


digits:
   ("0” .. “9")+

exponent:
   “E” digits

FLOAT:
   (digits)* "." (digits)* (exponent)?

expression:
  "{" expression ")" |
  STRING |
  INTEGERorVECTOR |
  FLOAT |
  "true" |
  "false" |
  "[]"|
  "fileloc" |
  name |
  "[" list_sugar_items "]" |
  "import" STRING |
  component_anonymous |
  instantiation_system |
  connection |
  expression "." expression |
  expression "@" expression |
  expression "&&" expression |
  expression "||" expression |
  expression "?" expression ":" expression |
  prefix_operator expression |
  expression infix_operator expression |

name:
  NAME |
  "`" prefix_operator |
  "`" infix_operator

prefix_operator:
   (all prefix operations listed in data type operation tables in 1.3)

infix_operator:
   (all infix operations listed in data type operation tables in 1.3)

component_anonymous:
  "comp" component_inputs component_outputs statements "end" |
  "comp" component_inputs component_outputs namespace_sugar "end" |

instantiation_system:
  "{" expression "}" |
  "{" expression "," port_expressions "}" |
  "{" expression "," port_expressions "," port_expressions "}" |

port_expressions:
  EMPTY |
  port_expressions expression |
  port_expressions "_"

connection:
  expression "<-" expression
```