The Lustre V6 Reference Manual

Erwan Jahier, Pascal Raymond, Nicolas Halbwachs

Software Version: 6.101.4 (23-05-19)

Contents

How to read this manual

This reference manual is splitted in two parts. The first chapter presents and defines the Lustre basic concepts. This *Lustre Core* language corresponds more or less to the intersection of the various versions of the Lustre language (from V1 to V6). Advance features (structured types) that changed accross version versions are not presented here.

The second chapter deals with the V6 specific features. Arrays, that were introduced in V4, are processed quite differently, using iterators. But the main novelty resides in the introduction of a package mechanism. Readers already familiar with Lustre ougth to read directly this chapter.

Chapter 1

An Overview of the Lustre Language

1.1 Introduction

This manual presents the LUSTRE language, a synchronous language based on the dataflow model and designed for the description and verification of real-time systems. In this chapter, we present the general framework that forms the basis of the language: the synchronous model, the dataflow model, and the synchronous dataflow model. Then we introduce the main features of the language through some simple examples.

The end of the chapter gives some basic elements for reading the rest of the document: it makes precise the metalanguage used to describe the syntax throughout the document and describes the lexical rules of the language.

1.1.1 Synchronous Model

The synchronous model was introduced to provide abstract primitives assuming that a program reacts instantaneously to external events. Each output of the program is assigned a precise date in relation to the flow of input events.

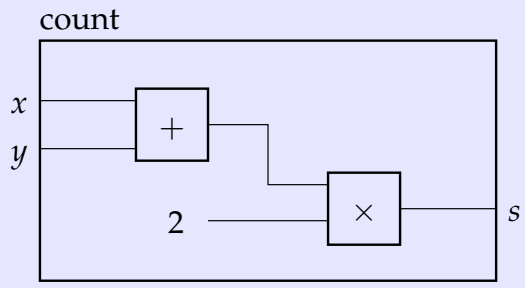
A discrete time scale is introduced. The time granularity is considered to be adapted a priori to the time constraints imposed by the dynamics of the environment on which the system is to react. It is verified a posteriori. Each instant on the time scale corresponds to a computation cycle, i.e., in the case of LUSTRE, to the arrival of new inputs. The synchrony hypothesis presumes that the means of computation are powerful enough for the level of granularity to be respected. In other words, the time to compute outputs in function of their inputs is less than the level of granularity on the discrete time scale. Consequently, outputs are computed and inputs are taken into account "at the same time" (with respect to the discrete time scale).

1.1.2 Dataflow Model

The dataflow model is based on a block diagram description. A block diagram can be described either graphically, or by a system of equations. A system is made up of a network of operators acting in parallel and in time with their input rate.

Example 1 A Textual and a graphical view of the same network

```
node count (x,y: int) returns (s: int);
let
 s = 2*(x+y);
tel
```



Graphic view of a network

This model provides the following advantages:

- maximal use made of parallelism (the only constraints are dependencies between data),
- mathematical formalization (formal verification methods),
- program construction and modification,
- ability to describe a system graphically.

1.1.3 Synchronous Dataflow Model

The synchronous dataflow approach consists in adding a time dimension to the dataflow model. A natural way of doing this is to associate time with the rate of dataflow. The entities manipulated can naturally be interpreted as functions of time. A basic entity (or flow) is a couple made up of:

- a sequence of values of a given type,
- a clock representing a suite of graduations (on the discrete time scale).

A flow takes the t^{th} value in its sequence at the t^{th} instant of its clock. For instance, the description given by the previous diagram expresses the following relation:

for any instant $t, s_t = 2 * (x_t + y_t)$

The time dimension is therefore an underlying feature in any description of this type of model. LUSTRE is a synchronous language based on the dataflow model. The synchronous aspect introduces constraints on the type of input/output relations that can be expressed: the output of a program at a given instant cannot depend on future inputs (causality) and can depend on only a bounded number of inputs (each cycle can memorize the value of the previous input).

1.1.4 Building a Description

A LUSTRE program describes the relations between the outputs and inputs of a system. These relations are expressed using operators, auxiliary variables, and constants. The operators can be:

- basic operators,
- more complex, user-defined, operators, called nodes.

Each description written in LUSTRE is built up of a network of nodes. A node describes the relation between its input and output parameters using a system of equations. Nodes correspond to the functions of the system and allow complex networks to be built simply by passing parameters.

The synchrony hypothesis presumes that each operator in the network responds to its inputs instantaneously.

A LUSTRE description is a list of type, constant and node declarations. The declarations can occur in any order.

The *functional behavior* of an application described in LUSTRE does not depend on the clock cycle. It is therefore possible to perform a functional validation of the application (ignoring the time validation) by testing it on a machine different from the target machine (on the development machine in particular).

Time validation is performed on the target machine. If the computation time is less than the time interval between two instants on the discrete time scale, it can be considered to be zero, and the synchrony hypothesis is satisfied. The interval between two instants on the scale is imposed by the requirements report. Computation time depends on software and hardware performance. LUSTRE is a language describing systems with a deterministic behavior from both a functional and a time point of view.

1.2 Basic Features

In this section, we present informally the main basic features of the language, through several simple examples.

A LUSTRE program or subprogram is called a *node*. LUSTRE is a functional language operating on *flows*. For the moment, let us consider that a flow is a finite or infinite sequence of values. All the values of a flow are of the same type, which is called the

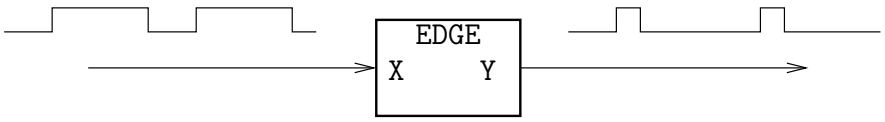


Figure 1.1: A Node

type of the flow. A program has a cyclic behavior. At the nth execution cycle of the program, all the involved flows take their nth value. A node defines one or several output parameters as functions of one or several input parameters. All these parameters are flows.

1.2.1 Simple control devices

As a very first example, let us consider a Boolean flow $X = (x_1, x_2, \ldots, x_n, \ldots)$. We want to define another Boolean flow $Y = (y_1, y_2, \ldots, y_n, \ldots)$ corresponding to the rising edge of X, i.e., such that y_{n+1} is true if and only if x_n is false and x_{n+1} is true (X raised from false to true at cycle $n+1$). The corresponding node (let us call it `EDGE`) will take X as an input parameter and return Y as an output parameter (see Fig. 1.1). The *interface* of the node is the following:

```
node EDGE (X: bool) returns (Y: bool);
```

The definition of the output `Y` is given by a single *equation*:

```
Y = X and not pre(X);
```

This equation defines "`Y`" (its left-hand side) to be *always equal* to the right-hand side *expression* "`X and not pre(X)`". This expression involves the input parameter `X` and three *operators*:

- "**and**" and "**not**" are usual Boolean operators, extended to operate pointwise on flows: if `A` $= (a_1, a_2, \ldots, a_n, \ldots)$ and `B` $= (b_1, b_2, \ldots, b_n, \ldots)$ are two Boolean flows, then "`A and B`" is the Boolean flow $(a_1 \wedge b_1, a_2 \wedge b_2, \ldots, a_n \wedge b_n, \ldots)$. Most usual operators are available in that way, and are called "*data-operators*".
- The "**pre**" (for "*previous*") operator allows one to refer at cycle n to the value of a flow at cycle $n-1$: if `A` $= (a_1, a_2, \ldots, a_n, \ldots)$ is a flow, **pre**`(A)` is the flow $(nil, a_1, a_2, \ldots, a_{n-1}, \ldots)$. Its first value is the undefined value *nil*, and for any $n > 1$, its nth value is the $(n-1)$th value of `A`

As a consequence, if `X` $= (x_1, x_2, \ldots, x_n, \ldots)$, the expression "`X and not pre(X)`" represents the flow $(nil, x_2 \wedge \neg x_1, \ldots, x_n \wedge \neg x_{n-1}, \ldots)$. Now, since its value at the first cycle is *nil* the program would be rejected[1] by the compiler: it indicates that the output lacks an initialization. A correct equation could be:

[1] Or, at least, a warning would be returned.

```
Y = false -> X and not pre(X);
```

Here, "**false**" denotes the *constant* flow, always equal to false. We have used the second specific LUSTRE operator, "->" (read "*followed by*") which defines initial values. If $A = (a_1, a_2, \ldots, a_n, \ldots)$ and $B = (b_1, b_2, \ldots, b_n, \ldots)$ are two flows of the same type, then "A -> B" is the flow $(a_1, b_2, \ldots, b_n, \ldots)$, equal to A at the first instant, and then forever equal to B.
So, the complete definition of the node EDGE is the following:

Example 2 The EDGE node

```
node EDGE (X: bool) returns (Y: bool);
let
 Y = false -> X and not pre(X);
tel
```

Once a node has been defined, it can be called from another node, using it as a new operator. For instance, let us write another node, computing the falling edge of its input parameter:

Example 3 The FALLING_EDGE node

```
node FALLING_EDGE (X: bool) returns (Y: bool);
let
 Y = EDGE(not X);
tel
```

The EDGE node is of very common usage for "deriving" a Boolean flow, i.e., transforming a "level" into a "signal". The converse operation is also very useful, it will be our second example: We want to implement a "switch", taking as input two signals "set" and "reset" and an initial value "initial", and returning a Boolean "level". Any occurrence of "set" rises the "level" to true, any occurrence of "reset" resets it to false. When neither "set" nor "reset" occurs, the "level" does not change. "initial" defines the initial value of "level". In LUSTRE, a signal is usually represented by a Boolean flow, whose value is true whenever the signal occurs. Below is a first version of the program:

Example 4 The SWITCH1 node

```
node SWITCH1 (set, reset, initial: bool) returns (level: bool);
let
  level = initial ->
    if set then true
    else if reset then false
    else pre(level);
tel
```

which specifies that the "level" is initially equal to "initial", and then forever,

- if "set" occurs, then it becomes true
- if "set" does not occur but "reset" does, then "level" becomes false
- if neither "set" nor "reset" occur, "level" keeps its previous value (notice that "level" is *recursively defined*: its current value is defined by means of its previous value).

Moreover, if this node is intended to be used only in contexts where inputs set and reset are never true together, such an *assertion* can be specified:

```
assert(not (set and reset));
```

Otherwise, this program has a flaw: It cannot be used as a "one-button" switch, whose level changes whenever its unique button is pushed. Let "change" be a Boolean flow representing a signal, then the call

```
state = SWITCH1(change,change,true);
```

will compute the always true flow: "state" is initialized to true, and never changes because the "set" formal parameter has been given priority. To get a node that can be used both as a "two-buttons" and a "one-button" switch, we have to make the program a bit more complex: the "set" signal must be considered only when the switch is turned off. We get the following program:

Example 5 The SWITCH node

```
node SWITCH (set, reset, initial: bool) returns (level: bool);
let
  level = initial  ->
    if set and not pre(level) then true
    else if reset then false
    else pre(level);
tel
```

1.2.2 Numerical examples

Recursive sequences are very easy to define in LUSTRE. For instance, the equation "N = 0 -> **pre** N + 1;" defines the sequence of natural numbers. Let us complexify this definition to build an integer sequence, whose value is, at each instant, the number of occurrences of the "true" value of a Boolean flow X:

```
N = 0 -> if X then pre N + 1 else pre N;
```

This definition does not exactly meet the specification, since it ignores the initial value of X. A well-initialized counter could be:

```
PN = 0 -> pre N;
N = if  X then PN + 1 else PN;
```

or, simply

```
N = if  X then (0 -> pre N) + 1 else  (0 -> pre N);
```

or even

```
N = (0 -> pre N) + if X then 0 else 1;
```

Let us write a more general operator, with additional inputs:

- an integer `init`, which is the initial value of the counter;
- an integer `incr`, which must be added to the counter when X is true;
- a Boolean `reset`, which reset the counter to the value `init`, whatever be the value of X.

The complete definition of this operator is the following:

Example 6 The COUNTER node

```
node COUNTER (init, incr: int; X, reset: bool) returns (N: int);
var PN: int;
let
  PN = init  -> pre N;
  N =
      if reset then init
      else if X then PN + incr
      else PN;
tel
```

This node can be used to define, e.g., the sequence of odd integers:

```
odds = COUNTER (0,2,true,false);
```

or the sequence of integers modulo 10:

```
mod10 = COUNTER (0,1,true,reset);
reset = true -> pre(mod10)=9;
```

Our next example involves real values. Let f be a real function of time, that we want to integrate using the trapezoid method. The program receives two real-valued flows F and STEP, such that

$$\mathtt{F}_n = f(x_n) \quad \text{and} \quad x_{n+1} = x_n + \mathtt{STEP}_{n+1}$$

It computes a real-valued flow Y, such that

$$\mathtt{Y}_{n+1} = \mathtt{Y}_n + (\mathtt{F}_n + \mathtt{F}_{n+1}) * \mathtt{STEP}_{n+1}/2$$

The initial value of Y is also an input parameter:

Example 7 The integrator node

```
node integrator(F,STEP,init: real) returns (Y: real);
let
 Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
tel
```

One can try to connect two such integrators in loop to compute the functions $\sin(\omega t)$ and $\cos(\omega t)$ in a simple-minded way:

Example 8 The buggy sincos node

```
-- there is a loop !
node sincos(omega:real) returns (sin, cos: real);
let
 sin = omega * integrator(cos,0.1,0.0);
 cos = omega * integrator(-sin,0.1,1.0);
tel
node integrator(F,STEP,init: real) returns (Y: real);
let
  Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
```

Called on this program, the compiler would complain that there is a *deadlock*. As a matter of fact, the variables sin and cos instantaneously depend on each other, i.e., the computation of the nth value of sin needs the nth value of cos, and conversely. We have to cut the dependence loop, introducing a "pre" operator:

Example 9 The sincos node

```
--
node sincos(omega : real) returns (sin, cos: real);
var pcos,psin: real;
let
  pcos = 1.0 fby(cos);
 psin = 0.0 fby sin;
 sin = omega * integrator(pcos,0.1,0.0);
 cos = omega * integrator(-psin,0.1,1.0);
tel
node integrator(F,STEP,init: real) returns (Y: real);
let
  Y = init -> pre(Y) + ((F + pre(F))*STEP)/2.0;
```

1.2.3 Multiple Equation

The node `sincos` above does not work very well, but it is interesting since it returns more than one output. To call such a node, LUSTRE allows *multiple definitions* to be written. Let `s`, `c`, `omega` be three real variables, then

```
(s, c) = sincos(omega);
```

is a correct LUSTRE equation, defining `s` and `c` to be, respectively, the first and the second result of the call.

So, the left-hand side of an equation can be a list of variables. The right hand side of such a multiple definition must denote a corresponding list of expressions, of suitable types. It can be

- a call to a node returning several outputs
- an explicit list
- the application of a *polymorphic* operator to a list

For instance, the equation

```
(min, max) = if a <b then (a,b) else (b,a);
```

directly defines `min` and `max` to be, respectively, the least and greatest value of `a` and `b`.

1.2.4 Clocks

Let us consider the following control device: it receives a signal "set", and returns a Boolean "level" that must be true during "delay" cycles after each reception of "set". The program is quite simple:

Example 10 The STABLE node

```
node STABLE (set: bool; delay: int) returns (level: bool);
var count: int;
let
  level = (count > 0);
  count =
      if set then delay
      else if false -> pre(level) then pre(count)-1
      else 0;
tel
```

Now, suppose we want the "level" to be high during "delay" seconds, instead of "delay" cycles. The "second" will be provided as a Boolean input "second", true whenever a second elapses. Of course, we can write a new program which freezes the counter whenever the "second" is not there:

Example 11 The TIME_STABLE1 node

```
node TIME_STABLE1(set,second:bool; delay:int) returns (level:bool);
var count: int;
let
  level = (count > 0);
  count =
      if set then delay
      else if second then
          if false -> pre(level) then pre(count)-1
          else 0
       else (0 -> pre(count));
tel
```

We can also reuse our node "STABLE", calling it at a suitable *clock*, by *filtering* its input parameters. It consists of changing the execution cycle of the node, activating it only at some cycles of the calling program. For the delay to be counted in seconds, the node "STABLE" must be activated only when either a "set" signal or a "second" signal occurs. Moreover, it must be activated at the initial instant, for initialization purposes. So the activation clock is

```
ck = true  ->  set or second;
```

Now a call "STABLE((set,delay) when ck)" will feed an instance of "STABLE" with rarefied inputs, as shown by the following table:

(set,delay)	(s_1,d_1)	(s_2,d_2)	(s_3,d_3)	(s_4,d_4)	(s_5,d_5)	(s_6,d_6)	(s_7,d_7)
ck	true	false	false	true	true	false	true
(set,delay) when ck	(s_1,d_1)			(s_4,d_4)	(s_5,d_5)		(s_7,d_7)

According to the data-flow philosophy of the language, this instance of "STABLE" will have a cycle only when getting input values, i.e., when ck is true. As a consequence, the inside counter will have the desired behavior, but the output will also be delivered at this rarefied rate. In order to use the result, we have first to *project* it onto the clock of the calling program. The resulting node is

Example 12 The TIME_STABLE node

```
node TIME_STABLE(set, second: bool; delay: int) returns (level: bool);
var ck: bool;
let
  level = current(STABLE((set,delay) when ck));
  ck = true  ->  set or second;
tel
node STABLE (set: bool; delay: int) returns (level: bool);
var count: int;
let
 level = (count > 0);
 count = if set then delay else if false -> pre(level) then pre(count)-1 else 0;
tel
```

Here is a simulation of this node:

(set,delay)	(tt,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(ff,2)	(tt,2)	(ff,2)
(second)	ff	ff	tt	ff	tt	ff	ff	ff	tt
ck	tt	ff	tt	ff	tt	ff	ff	tt	tt
(set,delay) when ck	(tt,2)		(ff,2)		(ff,2)			(tt,2)	(ff,2)
STABLE((set,delay) when ck)	tt		tt		ff			tt	tt
current(STABLE (set,delay) when ck))	tt	tt	tt	tt	ff	ff	ff	tt	tt

Chapter 2

Lustre Core

2.1 Notations

In the remaining of the document, we use the following notations: The wave arrow $\rightsquigarrow$ means that expression evaluates into. Grammar rule are given using an extended BNF notation, where non-terminals are written ⟨*like_this*⟩ and terminals "`like that`".

2.2 Lexical aspects

- One-line comments start with `--` and stop at the the end of the line.
- Multi-line comments start with `(*` and end at the next following `*)`. Multi-line comments cannot be nested.
- *Ident* stands for identifier, following the C standard (`[_a-zA-Z][_a-zA-Z0-9]*`),
- *Floating* and *Integer* stands for decimal floating point and integer notations, following C standard,

⟨*Ident*⟩ ⟨*string*⟩ ⟨*Value*⟩ ⟨*comment*⟩

2.3 Pragmas

A pragma is either empty, or an arbitrary string between "%" (no "%" inside the string, or some escape to be defined), or a list of such things:

⟨$\mathcal{P}$⟩ ::= ("`%`" ⟨*string*⟩ "`%`")*

Example 13 Pragmas

```
% foo.lus:42:1%
```

2.4 Identifiers

Entities are generally referred to through identifiers, but they can also depend on a package instance (like in `BIN8::binary`). So we distinguish between ⟨*Ident*⟩, and ⟨*Identifier*⟩:

⟨*Identifier*⟩ ::= ⟨*Ident*⟩ | ⟨*Ident*⟩"**::**"⟨*Ident*⟩

2.5 Types

⟨*Type_Decl*⟩ ::= "**type**" ⟨*Ident*⟩$^+$ ⟨$\mathcal{P}$⟩ "**;**"
| "**type**" ⟨*Ident*⟩ "**=**" ⟨*Type*⟩ ⟨$\mathcal{P}$⟩ "**;**"
⟨*Type*⟩ ::= ⟨*Ident*⟩ | ⟨*Record_Type*⟩ | ⟨*Array_Type*⟩ | ⟨*Enum_Type*⟩
⟨*Record_Type*⟩ ::= "**struct**" "{" ⟨*Field_List*⟩ "}"
⟨*Field_List*⟩ ::= ⟨*Field*⟩ | ⟨*Field*⟩ "**,**" ⟨*Field_List*⟩
⟨*Field*⟩ ::= ⟨*Ident*⟩ "**:**" ⟨*Type*⟩
⟨*Array_Type*⟩ ::= ⟨*Type*⟩ "**^** " ⟨*Expression*⟩
⟨*Enum_Type*⟩ ::= "**enum**" "{" ⟨*Ident_List*⟩ "}"

Example 14 Type Declarations

```
type alias = int;
type pair = struct { a:int; b:int };
type color = enum { blue, white, black };

node type_decl(i1, i2: int) returns (x: pair);
let
  x= pair {a=i1; b=i2};
tel
```

2.6 Constants and Variables

⟨*Const_Decl*⟩ ::= "**const**" (⟨*One_Const_Decl*⟩)$^+$
⟨*One_Const_Decl*⟩ ::= ⟨*Ident_List*⟩ "**:**" ⟨*Type*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| ⟨*Ident*⟩ "**=**" ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| ⟨*Ident*⟩ "**:**" ⟨*Type*⟩ "**=**" ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩ "**;**"
⟨*Ident_List*⟩ ::= ⟨*Ident*⟩ | ⟨*Ident*⟩ "**,**" ⟨*Ident_List*⟩

Example 15 Constant Declarations

const x,y,z : int; verbose = true; pi:real = 3.14159265359;

2.7 Functions and Nodes

The main way of structuring Lustre equations is via *nodes*. A memoryless node can be declared a *function*. A Lustre node is made of an interface (input/output declarations) and a set of equations defining the outputs.

⟨*Node_Decl*⟩ ::= ⟨*Node_Header*⟩ [⟨*FN_Body*⟩]
⟨*Node_Header*⟩ ::= ["**unsafe**"] ["**extern**"] ("**node**" | "**function**") "**(**" ⟨*FN_Params*⟩ "**)**" "**returns**" "**(**"⟨*FN_Params*⟩ "**)**" ⟨$\mathcal{P}$⟩ "**;**"
⟨*FN_Params*⟩ ::= ⟨*Var_Decl_List*⟩
⟨*Var_Decl_List*⟩ ::= ⟨*Var_Decl*⟩ | ⟨*Var_Decl*⟩ "**;**" ⟨*Var_Decl_List*⟩
⟨*Var_Decl*⟩ ::= ⟨*Ident_List*⟩ "**:**" ⟨*Type*⟩ [⟨*Declared_Clock*⟩] ⟨$\mathcal{P}$⟩
⟨*Declared_Clock*⟩ ::= "**when**" ⟨*Clock*⟩
⟨*Clock*⟩ ::= ⟨*Identifier*⟩
⟨*FN_Body*⟩ ::= (⟨*Local_Decl*⟩)* "**let**" ⟨*Equation_List*⟩ "**tel**" ["**;**"]
⟨*Local_Decl*⟩ ::= ⟨*Local_Var_Decl*⟩ | ⟨*Local_Const_Decl*⟩
⟨*Local_Var_Decl*⟩ ::= "**var**" ⟨*Var_Decl_List*⟩ "**;**"
⟨*Local_Const_Decl*⟩ ::= "**const**" (⟨*Ident*⟩ ["**:**" ⟨*Type*⟩] "**=**" ⟨*Expression*⟩ "**;**")$^+$

Example 16 Node

```
node sum(A:int) returns (S:int)
let
   S=A+(0->pre(S));
tel
function plus(A,B:int) returns (X:int)
let
   X=A+B;
tel
```

Functions and nodes can be extern, in which case they should be preceeded by the **extern** keyword, and have an empty body. Of course if an extern entity is declared as a function while it has memory, the behavior of the whole program is unpredictable.

Example 17 Extern Nodes

```
extern node foo_with_mem(A:int, B:bool, C: real) returns (X:int, Y: real);
extern function sin(A:real) returns (sinx: real);
```

Extern nodes that performs side-effects should be declared as unsafe. A node that uses unsafe node is unsafe (a warning is emitted if a node is unsafe while it is not declared as such).

Example 18 Unsafe Nodes

```
unsafe extern node rand() returns (R: real);
unsafe node randr(r:real) returns (R: int);
let
  R = r*rand();
tel
```

2.8 Equations

⟨*Equation_List*⟩ ::= ⟨*Eq_or_Ast*⟩ | ⟨*Eq_or_Ast*⟩ ⟨*Equation_List*⟩
⟨*Eq_or_Ast*⟩ ::= ⟨*Equation*⟩ | ⟨*Assertion*⟩
⟨*Equation*⟩ ::= ⟨*Left_Part*⟩ "**=**" ⟨*Right_Part*⟩ ⟨$\mathcal{P}$⟩ "**;**"
⟨*Left_Part*⟩ ::= "**(**" ⟨*Left_List*⟩ "**)**" | ⟨*Left_List*⟩
⟨*Left_List*⟩ ::= ⟨*Left*⟩ ("**,**" ⟨*Left*⟩)*
⟨*Left*⟩ ::= ⟨*Identifier*⟩ | ⟨*Left*⟩ ⟨*Selector*⟩
⟨*Selector*⟩ ::= "**.**" ⟨*Ident*⟩ | "**[**" ⟨*Expression*⟩ [⟨*SelTrancheEnd*⟩] "**]**"
⟨*SelTrancheEnd*⟩ ::= "**..**" ⟨*Expression*⟩
⟨*Assertion*⟩ ::= "**assert**" ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩ "**;**"

Example 19 Equations

```
...
x = a[2];       -- accessing an array
slice = a[2..5] -- get an array slice (i.e., a sub array)
...
```

2.9 Assertions

Example 20 Assertions

```
node divide(i1,i2:int) returns (res:int);
let
  assert(i2<>0);
  o = i1/i2;
tel
```

Assertions takes boolean expressions. Tools that parse lustre program can use it (or ignore it). For instance, the Lesar model-checker uses them to cut some some paths in the state graph. Lustre interpreters generate a warning when an assertion is violated.

2.10 Expressions

Lustre is a data-flow language: each variable or expression denotes a infinite sequence of values, i.e., a *stream*. All values in a stream are of the same data type, which is simply called the type of the stream. A variable `X` of type τ represents a sequence of values $\mathtt{X}_i \in \tau$ with $i \in \mathbb{N}$.

For instance, the predefined constant `true` denotes the infinite sequence of Boolean values $(true, true, \cdots)$, and the integer constant `42` denotes the infinite sequence $(42, 42, \cdots)$.

Three predefined types are provided: Boolean, integer and real. All the classical arithmetic and logic operators over those types are also predefined. We say that hey are *combinationnal* in the sense that they are operating pointwise on streams.

Example 21 Expressions

`X + Y` denotes the stream $(\mathtt{X}_i + \mathtt{Y}_i)_i$ with $i \in \mathbb{N}$.
`Z = X + Y` defines the stream `Z` from the streams `X` and `Y`

⟨*Expression*⟩ ::= ⟨*Identifier*⟩
| ⟨*Value*⟩
| "**(**" ⟨*Expression_List*⟩ "**)**"
| ⟨*Record_Exp*⟩
| ⟨*Array_Exp*⟩
| ⟨*Unary*⟩ ⟨*Expression*⟩
| ⟨*Expression*⟩ ⟨*Binary*⟩ ⟨*Expression*⟩
| ⟨*Nary*⟩ ⟨*Expression*⟩
| "**if**" ⟨*Expression*⟩ "**then**" ⟨*Expression*⟩ "**else**" ⟨*Expression*⟩
| ⟨*Call*⟩
| ⟨*Expression*⟩ ⟨*Selector*⟩

⟨*Expression_List*⟩ ::= ⟨*Expression*⟩ | ⟨*Expression*⟩ "**,**" ⟨*Expression_List*⟩

⟨*Record_Exp*⟩ ::= ⟨*Ident*⟩ "**{**" ⟨*Field_Exp_List*⟩ "**}**"

⟨*Field_Exp_List*⟩ ::= ⟨*Field_Exp*⟩ | ⟨*Field_Exp*⟩ "**;**" ⟨*Field_Exp_List*⟩

⟨*Field_Exp*⟩ ::= ⟨*Ident*⟩ "**=**" ⟨*Expression*⟩

⟨*Array_Exp*⟩ ::= "**[**" ⟨*Expression_List*⟩ "**]**" | ⟨*Expression*⟩ "**^** " ⟨*Expression*⟩

⟨*Call*⟩ ::= ⟨*User_Op*⟩ ⟨*P*⟩ "**(**" ⟨*Expression_List*⟩ "**)**"

⟨*User_Op*⟩ ::= ⟨*Identifier*⟩
| ⟨*Iterator*⟩ << ⟨*User_Op*⟩ "**,**" ⟨*Expression*⟩ >>

⟨*Iterator*⟩ ::= "**map**" | "**red**" | "**fill**" | "**fillred**" | "**boolred**"

Example 22 Array Expressions

```
array2 = [1,2];
array10 = 42^10;
array12 = array2 | array10; -- concat
slice = array12[1..10]; -- slice
array_sum = map<<+, 10>>(array10,slice);
max_elt = red<<max, 10>>(array_sum)
```

Example 23 Struct Expressions

```
type Toto = struct
x : int = 1;
y : int = 2
;
[...]
 s = Toto  x = 12; y = 13 ;
 ns = Toto  s with x = 42 ;
 x = s.x + ns.y;
```

2.11 Combinational operators

An operator is a predefined Lustre node.

⟨*Unary*⟩	::=	"-" | "not"
⟨*Binary*⟩	::=	"+" | "-" | "*" | "/" | "div" | "mod"
	|	">" | "<" | ">=" | "<=" | "<>" | "="
	|	"or" | "and" | "xor" | "=>"
⟨*Nary*⟩	::=	"♯" | "nor"

2.12 Temporal operators

In addition to the combinationnal operators, Lustre provides a delay (`pre`) and an initialization operator (`->`).

⟨*Unary*⟩	::=	"pre" | "current"
⟨*Binary*⟩	::=	
	|	"->" | "when" | "fby"

Example 24 Temporal operators

The equation

```
pX = 0 -> pre(X) + 1; -- or pre X + 1
pY = 0 fby Y + 1; -- or 0 fby(y)+1
```

defines X and Y as the stream (0,1,2,3, ...)

Example 25 Operators

```
X_on_c = X when C;
curr_X_on_base = current(X_on_C);
```

2.13 Operators Priority

The list below shows the relative precedences and associativity of operators. The constructions with lower precedence come first.

- "`else`"
- "`->`"
- "`=>`" (right associative)
- "`or`" "`xor`"
- "`and`"
- "`<`" "`<=`" "`=`" "`>=`" "`>`" "`<>`"
- "`not`"
- "`+`" "`-`" (left associative)
- "`*`" "`/`" "`%`" "`mod`" "`div`" (left associative)
- "`when`"
- "`-`" (unary minus) "`pre`" "`current`"

2.14 Clocks

It also provides a notion of clock, with a sampling operator (`when`) and a dual projection operator `current`.

Example 26 An example illustrating the use of clocks (cf Section 1.2.4)

```
node TIME_STABLE(set, second: bool; delay: int) returns (level: bool);
var ck: bool;
let
  level = current(STABLE((set,delay) when ck));
  ck = true  ->  set or second;
tel
node STABLE (set: bool; delay: int) returns (level: bool);
var count: int;
let
 level = (count > 0);
 count = if set then delay else if false -> pre(level) then pre(count)-1 else 0;
tel
```

2.15 Abstract types

At last, complex data types and functions are handled via a mechanism of *abstract types* (also called *imported types*). An imported type is defined as a simple name. Abstract constants and function manipulating such types can be declared. The way those external items are effectively launched from a Lustre program depends on the back-ends of the compiler.

2.16 Programs

A Lustre-core program is a set of constant, types, function and node Declarations.

Chapter 3

Lustre V6

In this chapter, we present the Lustre V6 specific features, that are not part of the basic Lustre. In Section 3.1 we introduce the Lustre V6 Structured data types (records, enumerations, arrays). In Section 3.2 we introduce array iterators. In Section 3.4 we introduce The Lustre V6 package system which aims at introduced a new level of structuration and modularity as well as namespace facilities. In Section 3.5 we provide the predefined entities (constant, type, operator and package) of Lustre V6. In Section A.1 we provide the Lustre V6 syntax rules. In Section 3.7 we provide a complete and commented program example.

3.1 User-defined data types

Structured data type are introduced in Lustre V6. We give an informal description of them in this Section. The syntax for their declaration and used is provided in Section A.1.

Enumerations. Enumerations are similar to enumerations in other languages.

Example 27 Enumerations

```
type color1 = enum { blue, white, black };
type color2 = enum { green, orange, yellow };

node enum0(x: color1) returns (y: color2);
let
  y = if x = blue then green else if x = white then orange else yellow;
tel
```

Records. The declaration of a structured type is (semantically) equivalent to the declaration of an abstract type, a collection of field-access functions, and a constructor function.

Example 28 Records

```
type complex = { re : real ; im : real };

const j = { re = -sqrt(3)/2; im = sqrt(3)/2 }; -- a complex constant

node get_im(c:complex) returns (x:real) ;
let
  x = c.im;
tel
```

Arrays. Here are a few examples of array declarations and definitions.

Example 29 Arrays

```
type matrix_3_3 = int ^ 3 ^ 3 ; -- to define a type matrix of integers
const m1 = 0 ^ 3 ^ 3;           -- a constant of type matrix_3_3
const m2 = [1,2,3] ^ 3;         -- another constant
const sm1 = m2[2]               -- a constant of type int^3 (⇝ [1,2,3])
```

⟨*Type_Decl*⟩	::=	"**type**" ⟨*Ident*⟩$^+$ ⟨$\mathcal{P}$⟩ "**;**"
	|	"**type**" ⟨*Ident*⟩ "**=**" ⟨*Type*⟩ ⟨$\mathcal{P}$⟩ "**;**"
⟨*Type*⟩	::=	⟨*Ident*⟩ | ⟨*Record_Type*⟩ | ⟨*Array_Type*⟩ | ⟨*Enum_Type*⟩
⟨*Record_Type*⟩	::=	"{" ⟨*Field_List*⟩ "}"
⟨*Field_List*⟩	::=	⟨*Field*⟩ | ⟨*Field*⟩ "**;**" ⟨*Field_List*⟩
⟨*Field*⟩	::=	⟨*Ident*⟩ "**:**" ⟨*Type*⟩
⟨*Array_Type*⟩	::=	⟨*Type*⟩ "**^** " ⟨*Expression*⟩
⟨*Enum_Type*⟩	::=	"**enum**" "{" ⟨*Ident_List*⟩ "}"

TO DO !!!slices

3.2 Array iterators

One the main novelty of Lustre-V6 is to provide a (restricted) notion of higher-order programming by defining *array iterators* to operate over arrays. Iterators replace the use of Lustre V4 homomorphic extension [**?**].

Using node expressions. In Lustre V6, a node denotation is not necessarily a simple identifier, since a node can be "built" by instantiating an iterator with static arguments. A node expression is then defined by:

node-exp ::= *ident*
| *meta-op* **<<** *static-arg*{ **;***static-arg* }$^+$ **>>**
static-arg ::=
meta-op ::= **map** |**fill** |**red** |**fillred** |**boolred**
val-exp |*node-exp* |*usual-op*

A static argument may be a statically evaluable expression (with the restriction that it can be statically evaluated), or a node expression as defined below. With some restrictions, it is also possible to use the "usual denotation" of the predefined operators (like **+**, **>=** etc). See **??** for a complete discusion on the use of predefined operators.

The semantics of iterators are presented in the sequel.

Using node expressions. The rules presented here complete the basic ones (chapter **??**).

Node expressions can be used as static parameters (see above), in value expressions:

val-exp ::= *node-exp***(** *val-exp*{ **,***val-exp* }$^+$ **)**

Node expressions can also be used to define a node:

⟨*node-def*⟩ ::= ⟨*node*⟩ ⟨*ident*⟩ **=** ⟨*node-exp*⟩ **;**

3.2.1 From scalars to arrays: `fill`

The `fill` iterator transforms a scalar-to-scalar node into a scalar-to-array node. The node argument must have a single input (input accumulator), a first output of the same type (output accumulator), and at least one another output.

The figure 3.1 shows the data-flow scheme of the fill iterator.

Definition 1: `fill`

For any integer constant `n` and any node `N` of type:

$$\tau \rightarrow \tau \times \theta_1 \times \ldots \times \theta_\ell,$$

`fill<<N; n>>` denotes a node of type:

$$\tau \rightarrow \tau \times \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$$

such that

$$(a_{out}, Y_1, \cdots, Y_\ell) = \texttt{fill<<N; n>>}(a_{in})$$

if and only if, $\exists a_0, \cdots a_n$ such that $a_0 = a_{in}$, $a_n = a_{out}$ and

$$\forall i = 0 \cdots n-1, (a_{i+1}, Y_1[i], \cdots, Y_\ell[i]) = N(a_i)$$

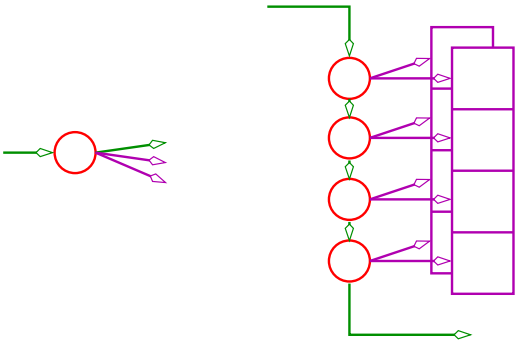


Figure 3.1: A node `N` (1 input, 1+2 outputs), and the node `fill<<N; 4>>`

Example 30 `fill`

```
fill<<incr; 4>>(0) ⇝ (4, [0,1,2,3])
```

with:

```
node incr(ain : int) returns (aout, z : int);
let
    z = ain; aout = ain + 1;
tel
```

3.2.2 From arrays to scalars: `red`

The `red` iteretor transforms a scalar-to-scalar node into an array-to-scalar node. The node argument must have a single output, a first input of the same type, and at least another input.

The figure 3.2 shows the data-flow scheme of the reduce iterator.

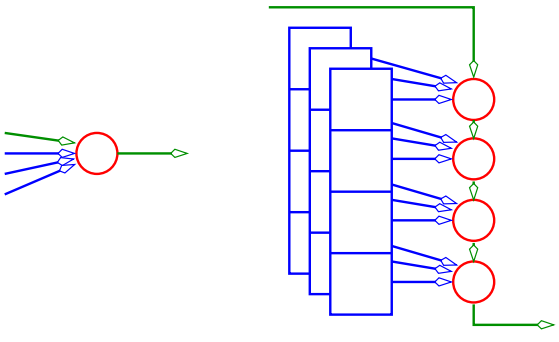


Figure 3.2: A node `N` (1+3 inputs, 1 output), and the node `red<<N; 4>>`

Definition 2: `red`

For any integer constant `n` and any node `N` of type:

$$\tau \times \tau_1 \times \ldots \times \tau_k \to \tau,$$

`red<<N; n>>` denotes a node of type:

$$\tau \times \tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \to \tau$$

such that

$$\mathtt{a}_{out} \texttt{ = red<<N; n>>(}\mathtt{a}_{in}\texttt{,}\mathtt{X}_1\texttt{,}\cdots\texttt{,}\mathtt{X}_k\texttt{)}$$

if and only if, $\exists \mathtt{a}_0, \cdots \mathtt{a}_n$ such that $\mathtt{a}_0 = \mathtt{a}_{in}$, $\mathtt{a}_n = \mathtt{a}_{out}$ and

$$\forall i = 0 \cdots \mathtt{n} - 1,\ \mathtt{a}_{i+1} \texttt{ = N(}\mathtt{a}_i\texttt{,}\mathtt{X}_1\texttt{[}i\texttt{]}\texttt{,}\cdots\texttt{,}\mathtt{X}_k\texttt{[}i\texttt{]}\texttt{)}$$

Example 31 `red`

```
red<<+; 3>>(0, [1,2,3]) ⇝ 6
```

3.2.3 From arrays to arrays: `fillred`

The `fillred` iterator generalizes the `fill` and the `red` ones. It maps a scalar-to-scalar node into a "scalar and array"-to-"scalar and array" node. The node argument must have a (first) input and a (first) output of the same type, and at least one more input and one more output. The degenerated case with no other input (resp. output) corresponds to the `fill` (resp. `red`) iterators.

The Figure 3.3 shows the data-flow scheme of the `fillred` iterator.

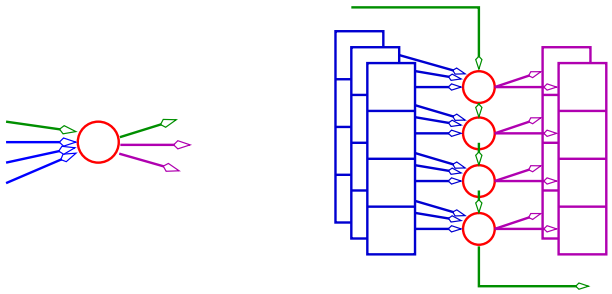


Figure 3.3: A node `N` (1+3 inputs, 1+2 outputs), and the node `+fillred<<N; 4>>`

Definition 3: `fillred`

For any integer constant `n` and any node `N` of type:

$$\tau \times \tau_1 \times \ldots \times \tau_k \to \tau \times \theta_1 \times \ldots \times \theta_\ell,$$

where k and $\ell \geq 0$; `fillred<<N; n>>` denotes a node of type:

$$\tau \times \tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \to \tau \times \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$$

such that

$$\texttt{(}\mathtt{a}_{out}\texttt{,}\mathtt{Y}_1\texttt{,}\cdots\texttt{,}\mathtt{Y}_\ell\texttt{)} \texttt{ = fillred<<N; n>>(}\mathtt{a}_{in}\texttt{,}\mathtt{X}_1\texttt{,}\cdots\texttt{,}\mathtt{X}_k\texttt{)}$$

if and only if, $\exists \mathtt{a}_0, \cdots, \mathtt{a}_n$ such that $\mathtt{a}_0 = \mathtt{a}_{in}$, $\mathtt{a}_n = \mathtt{a}_{out}$, and

$$\forall i = 0 \cdots \mathtt{n} - 1,\ \texttt{(}\mathtt{a}_{i+1}\texttt{,}\mathtt{Y}_1\texttt{[}i\texttt{]}\texttt{,}\cdots\texttt{,}\mathtt{Y}_\ell\texttt{[}i\texttt{]}\texttt{)} \texttt{ = N(}\mathtt{a}_i\texttt{,}\mathtt{X}_1\texttt{[}i\texttt{]}\texttt{,}\cdots\texttt{,}\mathtt{X}_k\texttt{[}i\texttt{]}\texttt{)}$$

Example 32 `fillred`

A classical exemple is the binary adder, obtained by mapping the "full-adder". The unsigned sum `Z` of two bytes `X` and `Y`, and the corresponding overflow flag can be obtained by:

```
(over, Z) = fillred<<fulladd, 8>>(false, X, Y)
```

where:

```
node fulladd(cin, x, y : bool) returns (cout, z : bool);
let
    z = cin xor x xor y;
    cout = if cin then x or y else x and y;
tel
```

3.2.4 From arrays to arrays, without an accumulator: `map`

The `map` iterator transforms a scalar-to-scalar node into an array-to-array node. The figure 3.4 shows the data-flow scheme of the map iterator.

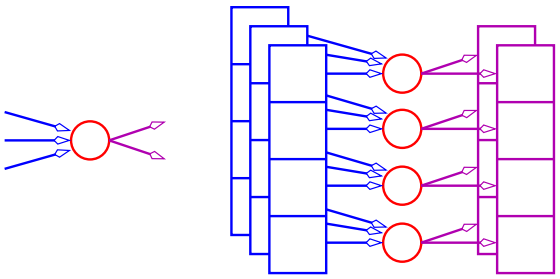


Figure 3.4: A node `N` (3 inputs, 2 outputs), and the node `map<<N; 4>>`

Definition 4: `map`

For any integer constant `n` and any node `N` of type:

$$\tau_1 \times \ldots \times \tau_k \to \theta_1 \times \ldots \times \theta_\ell,$$

`map<<N; n>>` denotes a node of type:

$$\tau_1\hat{}n \times \ldots \times \tau_k\hat{}n \to \theta_1\hat{}n \times \ldots \times \theta_\ell\hat{}n$$

such that

$$(Y_1,\cdots,Y_\ell) = \texttt{map<<N; n>>}(X_1,\cdots,X_k)$$

if and only if

$$\forall i = 0 \cdots n-1,\ (Y_1[i],\cdots,Y_\ell[i]) = N(X_1[i],\cdots,X_k[i])$$

Example 33 `map`

```
map <<+; 3>>([1,0,2],[3,6,-1]) ⇝ [4,6,1]
```

3.2.5 From Boolean arrays to Boolean scalar: `boolred`

Definition 5: `boolred`

This iterator has 3 integer static input arguments:

```
boolred<<i; j; k>>
```

such that $0 \leq i \leq j \leq k$ and $k > 0$.
It denotes a combinational node whose profile is `bool^k→bool`, and whose semantics is given by: the output is true if and only if at least i and at most j elements are true in the input array.

Note that this iterator can be used to implement efficiently the diese and the nor operators :

Example 34 boolred

```
#(a1, ..., an) ⇝ boolred<<0,1,n>>(a1, ..., an)
nor(a1, ..., an) ⇝ boolred<<0,0,n>>(a1, ..., an)
```

3.2.6 Lustre iterators versus usual functional languages ones.

Note that those iterators are more general than the ones usually provided in functional language librairies. Indeed, the arity of the node is not fixed. For example, in a usual functional language, you would have `map` and `map2` with the following profile:

`map : ('a -> 'b) -> (a' array) -> (b' array)`

`map2 : ('a -> 'b -> 'c) -> (a' array) -> (b' array) -> (c' array)`

whereas the `map` iterator we define here would have the following profile in the functional programming world :

`mapn : ('a1 -> 'a2 -> ... -> 'an) -> (a1' array) -> (a2' array) -> ... -> (an−1' array) -> (an' array)`

Note that it even note possible to give a milner-style type to describe this iterator. Indeed, the type of the node depends on the size of the array; it would therefore require a dependant-type system.

3.3 Parametric nodes

node can be parametrised by constants, types, and nodes.

Example 35 Parametric Node

```
node mk_tab<<type t; const init: t; const size: int>>
       (a:t) returns (res: t^size);
let
  res =  init ^ size;
tel
node tab_int3 = mk_tab<<int, 0, 3>>;
node param_node2 = mk_tab<<bool, true, 4>>;
```

Example 36 Parametric Node

```
node toto_n<<
   node f(a, b: int) returns (x: int);
   const n : int
  >>(a: int) returns (x: int^n);
var v : int;
let
  v = f(a, 1);
  x =  v ^ n;
tel
node param_node = toto_n<<Lustre::iplus, 3>>;
```

nodes can even be defined recursively using the "with" construct

Example 37 Recursive Node

```
node consensus<<const n : int>>(T: bool^n)
returns (a: bool);
let
  a = with (n = 1) then T[0]
      else T[0] and consensus << n-1 >> (T[1 .. n-1]);
tel

node consensus2 = consensus<<8>>;
```

3.4 Packages and models

A lustre V6 *program* is a list of packages, models (generic packages), and model instances.

Basic lustre programs are still accepted by the lustre V6 compiler, which consider implicitely that a program without package annotations :

- uses no other package
- provides all the package parameters it defines
- is part of a package that is made of the file name

⟨*Program*⟩ ::= (⟨*Package*⟩ | ⟨*Model*⟩ | ⟨*Model_Instance*⟩)*

A *package* is made of:

- a header, which gives the name of the package, the entities exported by the package, and the packages and models used by the package;
- and an optional body which consists of the declarations of the entities defined by the package. When the body is not given, the package is external.

⟨*Package*⟩ ::= ⟨*Package_Header*⟩ [⟨*Package_Body*⟩] "**end**"
⟨*Package_Header*⟩ ::= "**package**" ⟨*Ident*⟩ ⟨$\mathcal{P}$⟩
["**uses**" ⟨*Ident_List*⟩]
"**provides**" ⟨*Package_Params*⟩
⟨*Package_Params*⟩ ::= (⟨*Package_Param*⟩)$^+$
⟨*Package_Param*⟩ ::= "**const**" ⟨*Ident*⟩ "**:**" ⟨*Type_Identifier*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| "**type**" ⟨*Type_Ident_List*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| ⟨*Function_Header*⟩
| ⟨*Node_header*⟩
⟨*Type_Identifier*⟩ ::= ⟨*Identifier*⟩
⟨*Type_Ident_List*⟩ ::= ⟨*Ident*⟩ "**;**" | ⟨*Ident*⟩ "**,**" ⟨*Type_Ident_List*⟩

The output parameters of packages can be constants, types, nodes, or functions.

Example 38 Package

```
package pack
  uses pack1, pack2;
  provides
    const pi,e:real;
    type t1,t2;
    function cos(x:real) returns (y:real);
    node rising_edge(x:bool) returns (re:bool);
body
   ...
end
```

Example 39

```
package complex
provides
  type t; -- Encapsulation
  const i:t;
  node re(c: t) returns (r:real);
body
  type t = struct { re :  real ; im :  real };
  const i:t = t { re = 0. ; im = 1. };
  node re(c: t) returns (re:real);
  let re = c.re; tel;
  node complex = re;
end
```

A *model* has an additional section (`needs` ...) in its header which declares the formal parameters of the model. A model is somehow a parametric package.

⟨*Model*⟩ ::= ⟨*Model_Header*⟩ [⟨*Body*⟩] "**end**"
⟨*Model_Header*⟩ ::= "**model**" ⟨*Ident*⟩ ⟨$\mathcal{P}$⟩
["**uses**" ⟨*Ident_List*⟩]
"**needs**" ⟨*Package_Params*⟩
"**provides**" ⟨*Package_Params*⟩

Example 40 Model

```
model model_example
  needs
     type t;
     const pi;
  provides
    node n(init, in : t ) returns (res : t);
        body
           node n(init, in: t) returns (res: t);
           let
              res = init -> pre in;
           tel
        end
```

A *model instance* defines a package as an instance of a model by providing input parameters. It declares the list of packages it uses. It provides all objects exported by the model and its effective parameters.

⟨*Model_Instance*⟩ ::= "**package**" ⟨*Ident*⟩
["**uses**" ⟨*Ident_List*⟩]
"**is**" ⟨*Ident*⟩ "**(**" ⟨*Model_Actual_List*⟩ "**)**" ⟨$\mathcal{P}$⟩ "**;**"
⟨*Model_Actual_List*⟩ ::= ⟨*Model_Actual*⟩ | ⟨*Model_Actual*⟩ "**,**" ⟨*Model_Actual_List*⟩
⟨*Model_Actual*⟩ ::= ⟨*Identifier*⟩ ⟨$\mathcal{P}$⟩ | ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩

The user decide which node is the main one at compile time, following the Lustre V4 tradition. For example the node `bar` of package `p` in file `foo.lus` will be used as main node if the following command is launched: `lv6 foo.lus -main p::bar`.

Example 41 Model instance

Here is how to obtain packages by instanciating the model given in Example 40:

```
package model_instance_examble_bool is model_example(t=bool,pi=3.14);
package model_instance_examble_int is model_example(t=int,pi=3.14);
```

In this way, `model_instance_examble_bool` is a package that provides the node:

```
n(init, in : bool) returns (res : bool)
```

3.4.1 Package body

⟨*Package_Body*⟩ ::= ["**body**"] ⟨*Entity_Decl*⟩$^+$
⟨*Entity_Decl*⟩ ::= ⟨*Const_Decl*⟩
| ⟨*Type_Decl*⟩
| ⟨*Model_Instance*⟩
| ⟨*Function_Decl*⟩
| ⟨*Node_Decl*⟩

Example 42 Package body

3.5 Predefined entities

a package is a set of definitions of entities: types, constants and operators (nodes or functions).

a model can have as parameters a type, a constant, or a node.

3.6 The Merge operator

Example 43 The Merge operator

```
type piece = enum { Pile, Face, Tranche };
node test_merge(clk: piece; i1, i2, i3 : int)
returns (y: int);
let
  y = test_merge_clk(clk, i1 when Pile(clk),
                          i2 when Face(clk),
                          i3 when Tranche(clk));
tel
node test_merge_clk(clk: piece;
   i1 : int when Pile(clk) ;
   i2 : int when Face(clk);
   i3 : int when Tranche(clk))
returns (y: int);
let
    y = merge clk
        ( Pile    -> (0->i1))
        ( Face    -> i2)
        ( Tranche -> i3);
tel
node merge_bool_alt(clk : bool ;
                    i1  : int when clk ;
                    i2  : int when not clk)
returns (y: int);
let
    y = merge clk (true -> i1) (false-> i2);
tel
node merge_bool_ter(clk : bool ;
                    i1  : int when clk ;
                    i2 : int when not clk)
returns (y: int);
let
    y = merge clk (false-> i2) (true -> i1) ;
tel
```

clk	Pile	Pile	Face	Tranche	Pile	Face
i1 i2 i3	1	2	 1 	 1	3	 2
y	1	2	1	1	3	2

3.7 A complete example

Example 44 Detecting the stability of a flow

```
-- Time-stamp: <modified the 18/12/2017 (at 15:20) by Erwan Jahier>
-- Computes the speed (of some vehicle with wheels) out of 2 sampled inputs:
-- + Rot, true iff the wheel has performed a complete rotation
-- + Tic, true iff some external clock has emitted a signal indicating that
--        some constant amount of time elapsed (e.g., 100 ms)
--
-- This example was inspired from a real program in a train regulating system
const period = 0.1;  -- in seconds
const wheel_girth = 1.4;  -- in meter
const size = 20; -- size of the sliding window used to compute the speed
node compute_speed(Rot, Tic: bool) returns (Speed:real);
var  d,t,dx,tx:real;
let
  dx = if Rot then wheel_girth else 0.0;
  tx = if Tic then period else 0.0;
  d = sum<<size,0.0>>(dx);
  t = sum<<size,period>>(tx);
  -- the speed is actually the average speed during the last "size*period" seconds
  Speed = (d/t);
  -- nb : yes there can be some division by zero! For instance if the vehicle
  -- overtakes the speed of size*wheel_girth/period
  -- (i.e., with size=20, period=0.1, wheel_girth=1.4, if the speed is > 1008km/h)
  -- This means that for high-speed vehicle, one needs to increase "size".
tel
-- The idea is to call the node that do the computation only when needed, i.e.,
-- when Tic or Rot is true.
node speed(Rot, Tic: bool) returns (Speed:real);
var
  TicOrRot : bool;
  NewSpeed : real when TicOrRot;
let
  TicOrRot = Tic or Rot;
  NewSpeed = compute_speed(Rot when TicOrRot, Tic when TicOrRot);
  Speed = current(NewSpeed);
tel
-- computes the sum of the last d values taken by s
node sum<<const d: int; const init:real>>(s: real) returns (res:real);
var
  a,pre_a: real^d; -- circular array
  i: int;
let
  i = 0 fby i + 1;
  pre_a = (init^d) fby a;
  a = assign<<d>>(s, i mod d, pre_a);
  res =red<<+; d>>(0.0, a);
tel
-- assign the jth element of an array to a value. v.(j) <- i
type update_acc = { i: int; j: int; v: real };
function update_cell_do<<const d: int>>(acc: update_acc; cell: real)
returns (nacc: update_acc; ncell: real);
```

Appendix A

Appendix

A.1 The syntax rules summary

$\langle \mathcal{P} \rangle$ ::= ("**%**" $\langle$ *string* $\rangle$ "**%**")*

$\langle$ *Identifier* $\rangle$ ::= $\langle$ *Ident* $\rangle$ | $\langle$ *Ident* $\rangle$ "**::**" $\langle$ *Ident* $\rangle$

$\langle$ *Type_Decl* $\rangle$	::=	"**type**" $\langle$ *Ident* $\rangle^+$ $\langle \mathcal{P} \rangle$ "**;**"
	|	"**type**" $\langle$ *Ident* $\rangle$ "**=**" $\langle$ *Type* $\rangle$ $\langle \mathcal{P} \rangle$ "**;**"
$\langle$ *Type* $\rangle$	::=	$\langle$ *Ident* $\rangle$ | $\langle$ *Record_Type* $\rangle$ | $\langle$ *Array_Type* $\rangle$ | $\langle$ *Enum_Type* $\rangle$
$\langle$ *Record_Type* $\rangle$	::=	"**struct**" "{" $\langle$ *Field_List* $\rangle$ "}"
$\langle$ *Field_List* $\rangle$	::=	$\langle$ *Field* $\rangle$ | $\langle$ *Field* $\rangle$ "**,**" $\langle$ *Field_List* $\rangle$
$\langle$ *Field* $\rangle$	::=	$\langle$ *Ident* $\rangle$ "**:**" $\langle$ *Type* $\rangle$
$\langle$ *Array_Type* $\rangle$	::=	$\langle$ *Type* $\rangle$ "**^** " $\langle$ *Expression* $\rangle$
$\langle$ *Enum_Type* $\rangle$	::=	"**enum**" "{" $\langle$ *Ident_List* $\rangle$ "}"

$\langle$ *Const_Decl* $\rangle$	::=	"**const**" ($\langle$ *One_Const_Decl* $\rangle$)$^+$
$\langle$ *One_Const_Decl* $\rangle$	::=	$\langle$ *Ident_List* $\rangle$ "**:**" $\langle$ *Type* $\rangle$ $\langle \mathcal{P} \rangle$ "**;**"
	|	$\langle$ *Ident* $\rangle$ "**=**" $\langle$ *Expression* $\rangle$ $\langle \mathcal{P} \rangle$ "**;**"
	|	$\langle$ *Ident* $\rangle$ "**:**" $\langle$ *Type* $\rangle$ "**=**" $\langle$ *Expression* $\rangle$ $\langle \mathcal{P} \rangle$ "**;**"
$\langle$ *Ident_List* $\rangle$	::=	$\langle$ *Ident* $\rangle$ | $\langle$ *Ident* $\rangle$ "**,**" $\langle$ *Ident_List* $\rangle$

⟨*Node_Decl*⟩ ::= ⟨*Node_Header*⟩ [⟨*FN_Body*⟩]
⟨*Node_Header*⟩ ::= ["**unsafe**"] ["**extern**"] ("**node**" | "**function**") "**(**" ⟨*FN_Params*⟩ "**)**"
"**returns**" "**(**"⟨*FN_Params*⟩ "**)**" ⟨$\mathcal{P}$⟩ "**;**"
⟨*FN_Params*⟩ ::= ⟨*Var_Decl_List*⟩
⟨*Var_Decl_List*⟩ ::= ⟨*Var_Decl*⟩ | ⟨*Var_Decl*⟩ "**;**" ⟨*Var_Decl_List*⟩
⟨*Var_Decl*⟩ ::= ⟨*Ident_List*⟩ "**:**" ⟨*Type*⟩ [⟨*Declared_Clock*⟩] ⟨$\mathcal{P}$⟩
⟨*Declared_Clock*⟩ ::= "**when**" ⟨*Clock*⟩
⟨*Clock*⟩ ::= ⟨*Identifier*⟩
⟨*FN_Body*⟩ ::= (⟨*Local_Decl*⟩)* "**let**" ⟨*Equation_List*⟩ "**tel**" ["**;**"]
⟨*Local_Decl*⟩ ::= ⟨*Local_Var_Decl*⟩ | ⟨*Local_Const_Decl*⟩
⟨*Local_Var_Decl*⟩ ::= "**var**" ⟨*Var_Decl_List*⟩ "**;**"
⟨*Local_Const_Decl*⟩ ::= "**const**" (⟨*Ident*⟩ ["**:**" ⟨*Type*⟩] "**=**" ⟨*Expression*⟩ "**;**")+

⟨*Equation_List*⟩ ::= ⟨*Eq_or_Ast*⟩ | ⟨*Eq_or_Ast*⟩ ⟨*Equation_List*⟩
⟨*Eq_or_Ast*⟩ ::= ⟨*Equation*⟩ | ⟨*Assertion*⟩
⟨*Equation*⟩ ::= ⟨*Left_Part*⟩ "**=**" ⟨*Right_Part*⟩ ⟨$\mathcal{P}$⟩ "**;**"
⟨*Left_Part*⟩ ::= "**(**" ⟨*Left_List*⟩ "**)**" | ⟨*Left_List*⟩
⟨*Left_List*⟩ ::= ⟨*Left*⟩ ("**,**" ⟨*Left*⟩)*
⟨*Left*⟩ ::= ⟨*Identifier*⟩ | ⟨*Left*⟩ ⟨*Selector*⟩
⟨*Selector*⟩ ::= "**.**" ⟨*Ident*⟩ | "**[**" ⟨*Expression*⟩ [⟨*SelTrancheEnd*⟩] "**]**"
⟨*SelTrancheEnd*⟩ ::= "**..**" ⟨*Expression*⟩
⟨*Assertion*⟩ ::= "**assert**" ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩ "**;**"

```
⟨Expression⟩        ::= ⟨Identifier⟩
                      | ⟨Value⟩
                      | "(" ⟨Expression_List⟩ ")"
                      | ⟨Record_Exp⟩
                      | ⟨Array_Exp⟩
                      | ⟨Unary⟩ ⟨Expression⟩
                      | ⟨Expression⟩ ⟨Binary⟩ ⟨Expression⟩
                      | ⟨Nary⟩ ⟨Expression⟩
                      | "if" ⟨Expression⟩ "then" ⟨Expression⟩ "else" ⟨Expression⟩
                      | ⟨Call⟩
                      | ⟨Expression⟩ ⟨Selector⟩
⟨Expression_List⟩   ::= ⟨Expression⟩ | ⟨Expression⟩ "," ⟨Expression_List⟩
⟨Record_Exp⟩        ::= ⟨Ident⟩ "{" ⟨Field_Exp_List⟩ "}"
⟨Field_Exp_List⟩    ::= ⟨Field_Exp⟩ | ⟨Field_Exp⟩ ";" ⟨Field_Exp_List⟩
⟨Field_Exp⟩         ::= ⟨Ident⟩ "=" ⟨Expression⟩
⟨Array_Exp⟩         ::= "[" ⟨Expression_List⟩ "]" | ⟨Expression⟩ "^ " ⟨Expression⟩
⟨Call⟩              ::= ⟨User_Op⟩ ⟨P⟩ "(" ⟨Expression_List⟩ ")"
⟨User_Op⟩           ::= ⟨Identifier⟩
                      | ⟨Iterator⟩ << ⟨User_Op⟩ "," ⟨Expression⟩ >>
⟨Iterator⟩          ::= "map" | "red" | "fill" | "fillred" | "boolred"
```

```
⟨Unary⟩   ::= "-" | "not"
⟨Binary⟩  ::= "+" | "-" | "*" | "/" | "div" | "mod"
            | ">" | "<" | ">=" | "<=" | "<>" | "="
            | "or" | "and" | "xor" | "=>"
⟨Nary⟩    ::= "#" | "nor"
```

```
⟨Unary⟩   ::= "pre" | "current"
⟨Binary⟩  ::=
            | "->" | "when" | "fby"
```

```
⟨Type_Decl⟩     ::= "type" ⟨Ident⟩+ ⟨P⟩ ";"
                  | "type" ⟨Ident⟩ "=" ⟨Type⟩ ⟨P⟩ ";"
⟨Type⟩          ::= ⟨Ident⟩ | ⟨Record_Type⟩ | ⟨Array_Type⟩ | ⟨Enum_Type⟩
⟨Record_Type⟩   ::= "{" ⟨Field_List⟩ "}"
⟨Field_List⟩    ::= ⟨Field⟩ | ⟨Field⟩ ";" ⟨Field_List⟩
⟨Field⟩         ::= ⟨Ident⟩ ":" ⟨Type⟩
⟨Array_Type⟩    ::= ⟨Type⟩ "^ " ⟨Expression⟩
⟨Enum_Type⟩     ::= "enum" "{" ⟨Ident_List⟩ "}"
```

node-exp ::= *ident*
| *meta-op* **<<** *static-arg*{ **;***static-arg* }$^{+}$ **>>**
static-arg ::=
meta-op ::= **map** | **fill** | **red** | **fillred** | **boolred**
val-exp | *node-exp* | *usual-op*

⟨*node-def*⟩ ::= ⟨*node*⟩ ⟨*ident*⟩ **=** ⟨*node-exp*⟩ **;**

⟨*Program*⟩ ::= (⟨*Package*⟩ | ⟨*Model*⟩ | ⟨*Model_Instance*⟩)*

⟨*Package*⟩ ::= ⟨*Package_Header*⟩ [⟨*Package_Body*⟩] "**end**"
⟨*Package_Header*⟩ ::= "**package**" ⟨*Ident*⟩ ⟨$\mathcal{P}$⟩
["**uses**" ⟨*Ident_List*⟩]
"**provides**" ⟨*Package_Params*⟩
⟨*Package_Params*⟩ ::= (⟨*Package_Param*⟩)$^{+}$
⟨*Package_Param*⟩ ::= "**const**" ⟨*Ident*⟩ "**:**" ⟨*Type_Identifier*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| "**type**" ⟨*Type_Ident_List*⟩ ⟨$\mathcal{P}$⟩ "**;**"
| ⟨*Function_Header*⟩
| ⟨*Node_header*⟩
⟨*Type_Identifier*⟩ ::= ⟨*Identifier*⟩
⟨*Type_Ident_List*⟩ ::= ⟨*Ident*⟩ "**;**" | ⟨*Ident*⟩ "**,**" ⟨*Type_Ident_List*⟩

⟨*Model*⟩ ::= ⟨*Model_Header*⟩ [⟨*Body*⟩] "**end**"
⟨*Model_Header*⟩ ::= "**model**" ⟨*Ident*⟩ ⟨$\mathcal{P}$⟩
["**uses**" ⟨*Ident_List*⟩]
"**needs**" ⟨*Package_Params*⟩
"**provides**" ⟨*Package_Params*⟩

⟨*Model_Instance*⟩ ::= "**package**" ⟨*Ident*⟩
["**uses**" ⟨*Ident_List*⟩]
"**is**" ⟨*Ident*⟩ "**(**" ⟨*Model_Actual_List*⟩ "**)**" ⟨$\mathcal{P}$⟩ "**;**"
⟨*Model_Actual_List*⟩ ::= ⟨*Model_Actual*⟩ | ⟨*Model_Actual*⟩ "**,**" ⟨*Model_Actual_List*⟩
⟨*Model_Actual*⟩ ::= ⟨*Identifier*⟩ ⟨$\mathcal{P}$⟩ | ⟨*Expression*⟩ ⟨$\mathcal{P}$⟩

⟨*Package_Body*⟩ ::= ["**body**"] ⟨*Entity_Decl*⟩$^+$
⟨*Entity_Decl*⟩ ::= ⟨*Const_Decl*⟩
| ⟨*Type_Decl*⟩
| ⟨*Model_Instance*⟩
| ⟨*Function_Decl*⟩
| ⟨*Node_Decl*⟩

A.2 The syntax rules (automatically generated)

Lexical rules:

- *Ident* is an identifier, following the C standard.
- *IdentRef* is either an identifier, or a long identifier, that is an two identifiers separated by a double colon (*Ident*`::`*Ident*).
- *IntConst* is a integer notation, following the C standard.
- *RealConst* is a floating-point notation, following the C standard.

program ::= { *Include* } (*PackBody* | *PackList*)
PackList ::=
OnePack { *OnePack* }
OnePack ::=
ModelDecl | *PackDecl* | *PackEq*
Include ::= **include "<string>"**
Provides ::= [**provides** *Provide* **;** { *Provide* **;** }]
Provide ::= **const** *Lv6Id* **:** *Type* [**=** *Expression*]
| **unsafe node** *Lv6Id StaticParams Params* **returns** *Params*
| **node** *Lv6Id StaticParams Params* **returns** *Params*
| **unsafe function** *Lv6Id StaticParams Params* **returns** *Params*
| **function** *Lv6Id StaticParams Params* **returns** *Params*
| **type** *OneTypeDecl*
ModelDecl ::= **model** *Lv6Id Uses* **needs** *StaticParamList* **;** *Provides* **body** *PackBody* **end**
PackDecl ::= **package** *Lv6Id Uses Provides* **body** *PackBody* **end**
Uses ::= [**uses** *Lv6Id* { **,** *Lv6Id* } **;**]
Eq_or_Is ::= **=**
| **is**
PackEq ::= **package** *Lv6Id Eq_or_Is Lv6Id* **(** *ByNameStaticArgList* **)** **;**
PackBody ::=
OneDecl { *OneDecl* }

OneDecl ::= *ConstDecl* | *TypeDecl* | *ExtNodeDecl* | *NodeDecl*

TypedLv6IdsList ::= *TypedLv6Ids* { `;` *TypedLv6Ids* }

TypedLv6Ids ::= *Lv6Id* { `,` *Lv6Id* } `:` *Type*

TypedValuedLv6Ids ::= *TypedValuedLv6Id* { `;` *TypedValuedLv6Id* }

TypedValuedLv6Id ::= *Lv6Id* (`:` *Type* | `,` *Lv6Id* { `,` *Lv6Id* } `:` *Type* | `:` *Type* `=` *Expression*)

ConstDecl ::= **`const`** *ConstDeclList*

ConstDeclList ::= *OneConstDecl* `;` { *OneConstDecl* `;` }

OneConstDecl ::= *Lv6Id* (`:` *Type* | `,` *Lv6Id* { `,` *Lv6Id* } `:` *Type* | `:` *Type* `=` *Expression* | `=` *Expression*)

TypeDecl ::= **`type`** *TypeDeclList*

TypeDeclList ::= *OneTypeDecl* `;` { *OneTypeDecl* `;` }

OneTypeDecl ::= *Lv6Id* [`=` (*Type* | **`enum`** `{` *Lv6Id* { `,` *Lv6Id* } `}` | [**`struct`**] `{` *TypedValuedLv6Ids* [`;`] `}`)]

Type ::= (**`bool`** | **`int`** | **`real`** | *Lv6IdRef*) { `^` *Expression* }

ExtNodeDecl ::= (**`extern function`** | **`unsafe extern function`** | **`extern node`** | **`unsafe extern node`**) *Lv6Id Params* **`returns`** *Params* [`;`]

NodeDecl ::= *LocalNode*

LocalNode ::= **`node`** *Lv6Id StaticParams Params* **`returns`** *Params* [`;`] *LocalDecls Body* (`.` | [`;`])
| **`function`** *Lv6Id StaticParams Params* **`returns`** *Params* [`;`] *LocalDecls Body* (`.` | [`;`])
| **`node`** *Lv6Id StaticParams NodeProfileOpt* `=` *EffectiveNode* [`;`]
| **`function`** *Lv6Id StaticParams NodeProfileOpt* `=` *EffectiveNode* [`;`]
| **`unsafe node`** *Lv6Id StaticParams Params* **`returns`** *Params* [`;`] *LocalDecls Body* (`.` | [`;`])
| **`unsafe function`** *Lv6Id StaticParams Params* **`returns`** *Params* [`;`] *LocalDecls Body* (`.` | [`;`])
| **`unsafe node`** *Lv6Id StaticParams NodeProfileOpt* `=` *EffectiveNode* [`;`]
| **`unsafe function`** *Lv6Id StaticParams NodeProfileOpt* `=` *EffectiveNode* [`;`]

NodeProfileOpt ::= [*Params* **`returns`** *Params*]

StaticParams ::= [`<<` *StaticParamList* `>>`]

StaticParamList ::=
    StaticParam { **;** *StaticParam* }
StaticParam ::= **type** *Lv6Id*
    | **const** *Lv6Id* **:** *Type*
    | **node** *Lv6Id Params* **returns** *Params*
    | **function** *Lv6Id Params* **returns** *Params*
    | **unsafe node** *Lv6Id Params* **returns** *Params*
    | **unsafe function** *Lv6Id Params* **returns** *Params*
Params ::= **(** [*VarDeclList* [**;**]] **)**
LocalDecls ::= [*LocalDeclList*]
LocalDeclList ::=
    OneLocalDecl { *OneLocalDecl* }
OneLocalDecl ::=
    LocalVars
    | *LocalConsts*
LocalConsts ::= **const** *ConstDeclList*
LocalVars ::= **var** *VarDeclList* **;**
VarDeclList ::=
    VarDecl { **;** *VarDecl* }
VarDecl ::=
    TypedLv6Ids
    | *TypedLv6Ids* **when** *ClockExpr*
    | **(** *TypedLv6IdsList* **)** **when** *ClockExpr*
Body ::= **let** [*EquationList*] **tel**
EquationList ::=
    Equation { *Equation* }
Equation ::= (**assert** | *Left* **=**) *Expression* **;**
Left ::=
    LeftItemList
    | **(** *LeftItemList* **)**
LeftItemList ::=
    LeftItem { **,** *LeftItem* }
LeftItem ::=
    Lv6Id
    | *FieldLeftItem*
    | *TableLeftItem*
FieldLeftItem ::=
    LeftItem **.** *Lv6Id*
TableLeftItem ::=
    LeftItem **[** (*Expression* | *Select*) **]**
Expression ::=
    Constant
    | *Lv6IdRef*
    | **not** *Expression*
    | **-** *Expression*
    | **pre** *Expression*

current *Expression*
int *Expression*
real *Expression*

Expression **when** *ClockExpr*

Expression **fby** *Expression*

Expression **->** *Expression*

Expression **and** *Expression*

Expression **or** *Expression*

Expression **xor** *Expression*

Expression **=>** *Expression*

Expression **=** *Expression*

Expression **<>** *Expression*

Expression **<** *Expression*

Expression **<=** *Expression*

Expression **>** *Expression*

Expression **>=** *Expression*

Expression **div** *Expression*

Expression **mod** *Expression*

Expression **-** *Expression*

Expression **+** *Expression*

Expression **/** *Expression*

Expression ***** *Expression*
if *Expression* **then** *Expression* **else** *Expression*
with *Expression* **then** *Expression* **else** *Expression*
**(** *ExpressionList* **)**
nor **(** *ExpressionList* **)**

CallByPosExpression
[*ExpressionList* **]**

Expression **^** *Expression*

Expression **|** *Expression*

Expression **[** *Expression* **]**

	|	*Expression* `[` *Select* `]`
	|	*Expression* `.` *Lv6Id*
	|	*CallByNameExpression*
	|	`(` *ExpressionList* `)`
	|	**`merge`** *Lv6Id MergeCaseList*
MergeCaseList	::=	[*MergeCase*] { *MergeCase* }
MergeCase	::=	[`(` (*Lv6IdRef* | **`true`** | **`false`**) `->` *Expression* `)`]
ClockExpr	::=	*Lv6IdRef* `(` *Lv6Id* `)`
	|	*Lv6Id*
	|	**`not`** *Lv6Id*
	|	**`not`** `(` *Lv6Id* `)`
PredefOp	::=	**`not`** | **`fby`** | **`pre`** | **`current`** | **`->`** | **`and`** | **`or`** | **`xor`** | **`=>`** | **`=`** | **`<>`** | **`<`** | **`<=`** | **`>`** | **`>=`** | **`div`** | **`mod`** | **`-`** | **`+`** | **`/`** | **`*`** | **`if`**
CallByPosExpression	::=	*EffectiveNode* `(` *ExpressionList* `)`
EffectiveNode	::=	*Lv6IdRef* [**`<<`** *StaticArgList* **`>>`**]
StaticArgList	::=	*StaticArg* { (`,` | `;`) *StaticArg* }
StaticArg	::=	**`type`** *Type*
	|	**`const`** *Expression*
	|	**`node`** *EffectiveNode*
	|	**`function`** *EffectiveNode*
	|	*PredefOp*
	|	*SimpleExp*
	|	*SurelyType*
	|	*SurelyNode*
ByNameStaticArgList	::=	*ByNameStaticArg* { (`,` | `;`) *ByNameStaticArg* }
ByNameStaticArg	::=	**`type`** *Lv6Id* `=` *Type*
	|	**`const`** *Lv6Id* `=` *Expression*
	|	**`node`** *Lv6Id* `=` *EffectiveNode*
	|	**`function`** *Lv6Id* `=` *EffectiveNode*
	|	*Lv6Id* `=` *PredefOp*
	|	*Lv6Id* `=` *SimpleExp*
	|	*Lv6Id* `=` *SurelyType*
	|	*Lv6Id* `=` *SurelyNode*

SurelyNode ::= *Lv6IdRef* **<<** *StaticArgList* **>>**

SurelyType ::= (**bool** | **int** | **real**) { **^** *Expression* }

SimpleExp ::=
- *Constant*
- *Lv6IdRef*
- *SimpleTuple*
- **not** *SimpleExp*
- **-** *SimpleExp*
- *SimpleExp* **and** *SimpleExp*
- *SimpleExp* **or** *SimpleExp*
- *SimpleExp* **xor** *SimpleExp*
- *SimpleExp* **=>** *SimpleExp*
- *SimpleExp* **=** *SimpleExp*
- *SimpleExp* **<>** *SimpleExp*
- *SimpleExp* **<** *SimpleExp*
- *SimpleExp* **<=** *SimpleExp*
- *SimpleExp* **>** *SimpleExp*
- *SimpleExp* **>=** *SimpleExp*
- *SimpleExp* **div** *SimpleExp*
- *SimpleExp* **mod** *SimpleExp*
- *SimpleExp* **-** *SimpleExp*
- *SimpleExp* **+** *SimpleExp*
- *SimpleExp* **/** *SimpleExp*
- *SimpleExp* ***** *SimpleExp*
- **if** *SimpleExp* **then** *SimpleExp* **else** *SimpleExp*

SimpleTuple ::= [**(** *SimpleExpList* **)**]

SimpleExpList ::= *SimpleExp* { **,** *SimpleExp* }

CallByNameExpression ::= [*Lv6IdRef* **{** [[*Lv6IdRef* **with**] *CallByNameParamList* [**;**]] **}**]

CallByNameParamList ::= *CallByNameParam* { (**;** | **,**) *CallByNameParam* }

CallByNameParam ::= *Lv6Id* **=** *Expression*

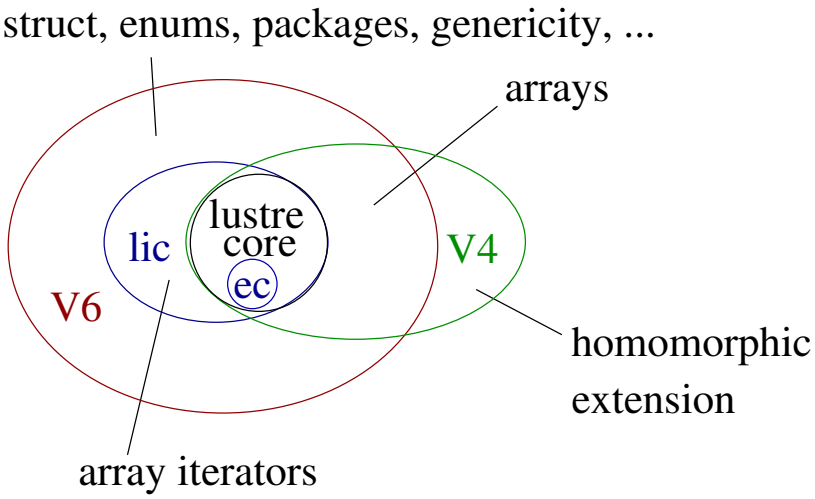


Figure A.1: Lustre potatoes

ExpressionList	::=	[*Expression*] { **,** *Expression* }
Constant	::=	**true** | **false** | *IntConst* | *RealConst*
Select	::=	*Expression* **..** *Expression Step*
Step	::=	[**step** *Expression*]
Pragma	::=	{ **%** *TK_IDENT* **:** *TK_IDENT* **%** }

A.3 Lustre History

Lustre V1, v2, v3, ..., v6

A.4 Some Lustre V4 features not supported in Lustre V6

- recursive arrays slices : use iterators instead

```
[int, real] -> use structures instead
```

```
[int, int] -> use int^2 instead
```

Bibliography