

The Lustre V6 Reference Manual (draft)

Initial version: May 5, 2010

Last update: May 5, 2010

Contents

1	An Overview of the Lustre Language	6
1.1	Introduction	6
1.1.1	Synchronous Model	6
1.1.2	Dataflow Model	7
1.1.3	Synchronous Dataflow Model	7
1.1.4	Building a Description	8
1.2	Basic Features	8
1.2.1	Simple control devices	9
1.2.2	Numerical examples	11
1.2.3	Multiple Equation	13
1.2.4	Clocks	14
2	Lustre Core	17
2.1	Notations	17
2.2	Lexical aspects	17
2.3	Pragmas	17
2.4	Identifiers	17
2.5	Types	18
2.6	Constants and Variables	18
2.7	Functions and Nodes	18
2.8	Equations	19
2.9	Assertions	20
2.10	Expressions	20
2.11	Combinational operators	21
2.12	Temporal operators	21
2.13	Clocks	21
2.14	Abstract types	22
2.15	Programs	22
3	Lustre V6	23
3.1	User-defined data types	23
3.2	Array iterators	24
3.2.1	From scalars to arrays: fill	25

3.2.2	From arrays to scalars: <code>red</code>	26
3.2.3	From arrays to arrays: <code>fillred</code>	27
3.2.4	From arrays to arrays, without an accumulator: <code>map</code>	28
3.2.5	From Boolean arrays to Boolean scalar: <code>boolred</code>	28
3.2.6	Lustre iterators versus usual functional languages ones.	29
3.3	Parametric nodes	30
3.4	Packages and models	31
3.4.1	Package body	33
3.5	Predefined entities	34
3.6	The Merge operator	35
3.7	A complete example	37
A	Appendix	39
A.1	The syntax rules summary	39
A.2	The syntax rules (automatically generated)	42
A.3	Lustre History	47
A.4	The Lustre V4 features not supported in Lustre V6	47

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 ough 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.

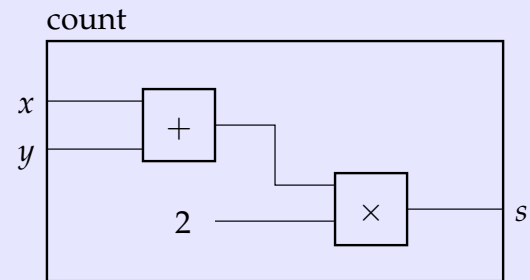
A discrete time scale is introduced. The time granularity is considered to be adapted a priori to the time constraints imposed by the dynamics of the environment on which the system is to react. It is verified a posteriori. Each instant on the time scale corresponds to a computation cycle, i.e., in the 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:

$$\text{for any instant } t, s_t = 2 * (x_t + y_t)$$

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

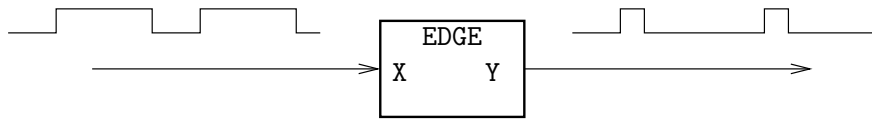


Figure 1.1: A Node

type of the flow. A program has a cyclic behavior. At the n th execution cycle of the program, all the involved flows take their n th 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 $X = (x_1, x_2, \dots, x_n, \dots)$. We want to define another Boolean flow $Y = (y_1, y_2, \dots, y_n, \dots)$ 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 $A = (a_1, a_2, \dots, a_n, \dots)$ and $B = (b_1, b_2, \dots, b_n, \dots)$ are two Boolean flows, then “ A **and** B ” is the Boolean flow $(a_1 \wedge b_1, a_2 \wedge b_2, \dots, a_n \wedge b_n, \dots)$. 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 $A = (a_1, a_2, \dots, a_n, \dots)$ is a flow, `pre(A)` is the flow $(nil, a_1, a_2, \dots, a_{n-1}, \dots)$. Its first value is the undefined value *nil*, and for any $n > 1$, its n th value is the $(n - 1)$ th value of A .

As a consequence, if $X = (x_1, x_2, \dots, x_n, \dots)$, the expression “ X **and not** `pre(X)`” represents the flow $(nil, x_2 \wedge \neg x_1, \dots, x_n \wedge \neg x_{n-1}, \dots)$. Now, since its value at the first cycle is *nil* the program would be rejected¹ by the compiler: it indicates that the output lacks an initialization. A correct equation could be:

¹Or, 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 $A = (a_1, a_2, \dots, a_n, \dots)$ and $B = (b_1, b_2, \dots, b_n, \dots)$ are two flows of the same type, then “A -> B” is the flow $(a_1, b_2, \dots, b_n, \dots)$, 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

$$F_n = f(x_n) \quad \text{and} \quad x_{n+1} = x_n + STEP_{n+1}$$

It computes a real-valued flow Y , such that

$$Y_{n+1} = Y_n + (F_n + F_{n+1}) * STEP_{n+1} / 2$$

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:

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 n th value of \sin needs the n th value of \cos , and conversely. We have to cut the dependence loop, introducing a “pre” operator:

Example 8 The sincos node

```
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
```

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 , ω 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 9 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 10 The TIME_STABLE1 node

```

node TIME1_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 11 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

```

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 $\langle \textit{like_this} \rangle$ and terminals “**like that**”.

2.2 Lexical aspects

TO DO !!!

$\langle \textit{Ident} \rangle \langle \textit{string} \rangle \langle \textit{Value} \rangle \langle \textit{comment} \rangle$

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:

$\langle \mathcal{P} \rangle ::= (\text{“\%”} \langle \textit{string} \rangle \text{“\%”})^*$

Example 12 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 $\langle \textit{Ident} \rangle$, and $\langle \textit{Identifier} \rangle$:

$\langle \textit{Identifier} \rangle ::= \langle \textit{Ident} \rangle \mid \langle \textit{Ident} \rangle \text{“::”} \langle \textit{Ident} \rangle$

2.5 Types

```

⟨Type_Decl⟩ ::= "type" ⟨Ident⟩+ ⟨P⟩ ";"
              | "type" ⟨Ident⟩ "=" ⟨Type⟩ ⟨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 13 Type Declarations

```

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

```

2.6 Constants and Variables

```

⟨Const_Decl⟩ ::= "const" ( ⟨One_Const_Decl⟩ )+
⟨One_Const_Decl⟩ ::= ⟨Ident_List⟩ ":" ⟨Type⟩ ⟨P⟩ ";"
                   | ⟨Ident⟩ "=" ⟨Expression⟩ ⟨P⟩ ";"
                   | ⟨Ident⟩ ":" ⟨Type⟩ "=" ⟨Expression⟩ ⟨P⟩ ";"
⟨Ident_List⟩ ::= ⟨Ident⟩ | ⟨Ident⟩ "," ⟨Ident_List⟩

```

Example 14 Constant Declarations

2.7 Functions and Nodes

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

$\langle \text{Function_Decl} \rangle$	$::=$	$\langle \text{Function_Header} \rangle [\langle \text{FN_Body} \rangle]$
$\langle \text{Function_Header} \rangle$	$::=$	<code>"function" "(" $\langle \text{FN_Params} \rangle$ ")"</code> <code>"returns" "(" $\langle \text{FN_Params} \rangle$ ")" $\langle \mathcal{P} \rangle$ ";"</code>
$\langle \text{Node_Decl} \rangle$	$::=$	$\langle \text{Node_Header} \rangle [\langle \text{FN_Body} \rangle]$
$\langle \text{Node_Header} \rangle$	$::=$	<code>"node" "(" $\langle \text{FN_Params} \rangle$ ")"</code> <code>"returns" "(" $\langle \text{FN_Params} \rangle$ ")" $\langle \mathcal{P} \rangle$ ";"</code>
$\langle \text{FN_Params} \rangle$	$::=$	$\langle \text{Var_Decl_List} \rangle$
$\langle \text{Var_Decl_List} \rangle$	$::=$	$\langle \text{Var_Decl} \rangle$ $\langle \text{Var_Decl} \rangle$ ";" $\langle \text{Var_Decl_List} \rangle$
$\langle \text{Var_Decl} \rangle$	$::=$	$\langle \text{Ident_List} \rangle$ ":" $\langle \text{Type} \rangle$ [$\langle \text{Declared_Clock} \rangle$] $\langle \mathcal{P} \rangle$
$\langle \text{Declared_Clock} \rangle$	$::=$	<code>"when" $\langle \text{Clock} \rangle$</code>
$\langle \text{Clock} \rangle$	$::=$	$\langle \text{Identifier} \rangle$
$\langle \text{FN_Body} \rangle$	$::=$	[$\langle \text{Local_Var_Decl} \rangle$] <code>"let" $\langle \text{Equation_List} \rangle$ "tel" [";"]</code>
$\langle \text{Local_Var_Decl} \rangle$	$::=$	<code>"var" $\langle \text{Var_Decl_List} \rangle$ [";"]</code>

Example 15 Node

```
node foo(A:int, B:bool, C: real) returns (X:int, Y: real)
let
  ...
tel
```

XXX definir extern

2.8 Equations

$\langle \text{Equation_List} \rangle$	$::=$	$\langle \text{Eq_or_Ast} \rangle$ $\langle \text{Eq_or_Ast} \rangle$ $\langle \text{Equation_List} \rangle$
$\langle \text{Eq_or_Ast} \rangle$	$::=$	$\langle \text{Equation} \rangle$ $\langle \text{Assertion} \rangle$
$\langle \text{Equation} \rangle$	$::=$	$\langle \text{Left_Part} \rangle$ "=" $\langle \text{Right_Part} \rangle$ $\langle \mathcal{P} \rangle$ ";"
$\langle \text{Left_Part} \rangle$	$::=$	<code>"(" $\langle \text{Left_List} \rangle$ ")"</code> $\langle \text{Left_List} \rangle$
$\langle \text{Left_List} \rangle$	$::=$	$\langle \text{Left} \rangle$ ("," $\langle \text{Left} \rangle$)*
$\langle \text{Left} \rangle$	$::=$	$\langle \text{Identifier} \rangle$ $\langle \text{Left} \rangle$ $\langle \text{Selector} \rangle$
$\langle \text{Selector} \rangle$	$::=$	<code>"." $\langle \text{Ident} \rangle$ "[" $\langle \text{Expression} \rangle$ [$\langle \text{SelTrancheEnd} \rangle$] "]"</code>
$\langle \text{SelTrancheEnd} \rangle$	$::=$	<code>".." $\langle \text{Expression} \rangle$</code>
$\langle \text{Assertion} \rangle$	$::=$	<code>"assert" $\langle \text{Expression} \rangle$ $\langle \mathcal{P} \rangle$ ";"</code>

Example 16 Equations

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

2.9 Assertions

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 $X_i \in \tau$ with $i \in \mathbb{N}$.

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

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 they are *combinational* in the sense that they are operating pointwise on streams.

Example 17 Expressions

$X + Y$ denotes the stream $(X_i + Y_i)_i$ with $i \in \mathbb{N}$.

$Z = X + Y$ defines the stream Z from the streams X and Y

$\langle \text{Expression} \rangle$	$::=$	$\langle \text{Identifier} \rangle$ $\langle \text{Value} \rangle$ $\text{"("} \langle \text{Expression_List} \rangle \text{"}$ $\langle \text{Record_Exp} \rangle$ $\langle \text{Array_Exp} \rangle$ $\langle \text{Unary} \rangle \langle \text{Expression} \rangle$ $\langle \text{Expression} \rangle \langle \text{Binary} \rangle \langle \text{Expression} \rangle$ $\langle \text{Nary} \rangle \langle \text{Expression} \rangle$ $\text{"if"} \langle \text{Expression} \rangle \text{"then"} \langle \text{Expression} \rangle \text{"else"} \langle \text{Expression} \rangle$ $\langle \text{Call} \rangle$ $\langle \text{Expression} \rangle \langle \text{Selector} \rangle$
$\langle \text{Expression_List} \rangle$	$::=$	$\langle \text{Expression} \rangle \mid \langle \text{Expression} \rangle \text{","} \langle \text{Expression_List} \rangle$
$\langle \text{Record_Exp} \rangle$	$::=$	$\langle \text{Ident} \rangle \text{"\{"} \langle \text{Field_Exp_List} \rangle \text{"\}"}$
$\langle \text{Field_Exp_List} \rangle$	$::=$	$\langle \text{Field_Exp} \rangle \mid \langle \text{Field_Exp} \rangle \text{";" } \langle \text{Field_Exp_List} \rangle$
$\langle \text{Field_Exp} \rangle$	$::=$	$\langle \text{Ident} \rangle \text{"="} \langle \text{Expression} \rangle$
$\langle \text{Array_Exp} \rangle$	$::=$	$\text{"["} \langle \text{Expression_List} \rangle \text{"]"} \mid \langle \text{Expression} \rangle \text{"^"} \langle \text{Expression} \rangle$
$\langle \text{Call} \rangle$	$::=$	$\langle \text{User_Op} \rangle \langle \mathcal{P} \rangle \text{"("} \langle \text{Expression_List} \rangle \text{"}"}$
$\langle \text{User_Op} \rangle$	$::=$	$\langle \text{Identifier} \rangle$ $\mid \langle \text{Iterator} \rangle \ll \langle \text{User_Op} \rangle \text{";" } \langle \text{Expression} \rangle \gg$
$\langle \text{Iterator} \rangle$	$::=$	$\text{"map"} \mid \text{"red"} \mid \text{"fill"} \mid \text{"fillred"} \mid \text{"boolred"}$

Example 18 Expressions

2.11 Combinational operators

An operator is a predefined Lustre node. Most of them are infix.



say which one

```

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

2.12 Temporal operators

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

Example 19 Temporal operators

The equation

$$X = 0 \rightarrow \text{pre } X + 1;$$

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

```

<Unary> ::= "pre" | "current"
<Binary> ::= "->" | "when"
  
```

Example 20 Operators

2.13 Clocks

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

Example 21

TO DO !!!

2.14 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 (see the Lustre V6 Code Generator Manual).

cf sec 2.2 man-ref.v5 (p17)

Example 22

TO DO !!!

2.15 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. TO DO !!!

Example 23 Enumerations

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

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.

TO DO !!!

Example 24 Records

```

type complex = { re : real ; im : real };
const j = { re = -sqrt(3)/2; im = sqrt(3)/2 };

```

Arrays. **TO DO !!!**

Example 25 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⟩+ ⟨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⟩ "}"

```

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:

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 discussion 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:

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

Node expressions can also be used to define a node:

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.

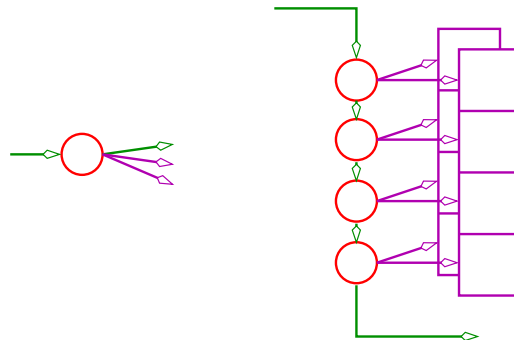


Figure 3.1: A node `N` (1 input, 1+2 outputs), and the node `fill«N; 4»`

Definition 1: fill

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

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

`fill«N; n»` denotes a node of type:

$$\tau \rightarrow \tau \times \theta_1^{\wedge n} \times \dots \times \theta_\ell^{\wedge n}$$

such that

$$(a_{out}, Y_1, \dots, Y_\ell) = \text{fill}\langle N; n \rangle(a_{in})$$

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

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

Example 26 fill

```
fill<<incr; 4>>(0)  $\rightsquigarrow$  (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 iterator 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.

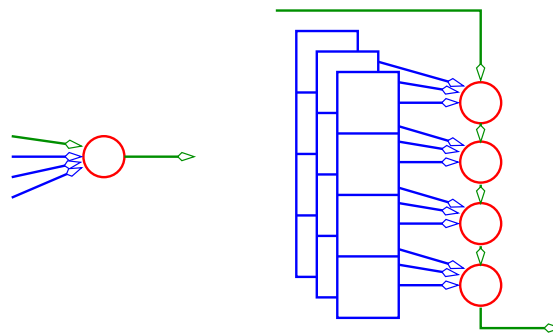


Figure 3.2: A node N (1+3 inputs, 1 output), and the node $\text{red}\langle N; 4 \rangle$

Definition 2: red

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

$$\tau \times \tau_1 \times \dots \times \tau_k \rightarrow \tau,$$

$\text{red}\langle N; n \rangle$ denotes a node of type:

$$\tau \times \tau_1^n \times \dots \times \tau_k^n \rightarrow \tau$$

such that

$$a_{out} = \text{red}\langle N; n \rangle(a_{in}, X_1, \dots, X_k)$$

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

$$\forall i = 0 \dots n - 1, a_{i+1} = N(a_i, X_1[i], \dots, X_k[i])$$

Example 27 red

```
red<<+; 3>>(0, [1,2,3])  $\rightsquigarrow$  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.

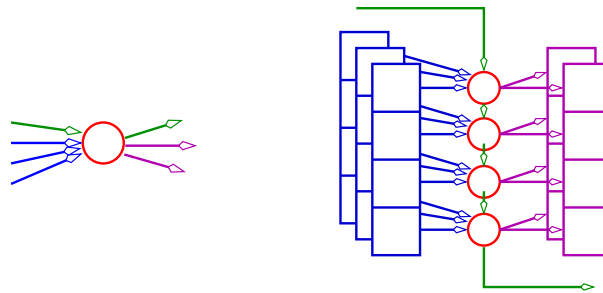


Figure 3.3: A node N (1+3 inputs, 1+2 outputs), and the node $+fillred\langle N; 4 \rangle$

Definition 3: fillred

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

$$\tau \times \tau_1 \times \dots \times \tau_k \rightarrow \tau \times \theta_1 \times \dots \times \theta_\ell,$$

where k and $\ell \geq 0$; $fillred\langle\langle N; n \rangle\rangle$ denotes a node of type:

$$\tau \times \tau_1^{\wedge n} \times \dots \times \tau_k^{\wedge n} \rightarrow \tau \times \theta_1^{\wedge n} \times \dots \times \theta_\ell^{\wedge n}$$

such that

$$(a_{out}, Y_1, \dots, Y_\ell) = fillred\langle\langle N; n \rangle\rangle(a_{in}, X_1, \dots, X_k)$$

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

$$\forall i = 0 \dots n - 1, (a_{i+1}, Y_1[i], \dots, Y_\ell[i]) = N(a_i, X_1[i], \dots, X_k[i])$$

Example 28 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:

$$(over, Z) = fillred\langle fulladd, 8 \rangle(false, X, Y)$$

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.

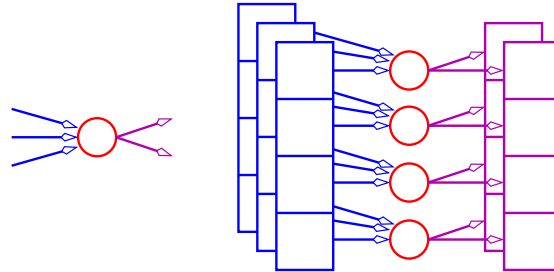


Figure 3.4: A node N (3 inputs, 2 outputs), and the node $\text{map}\langle N; 4 \rangle$

Definition 4: map

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

$$\tau_1 \times \dots \times \tau_k \rightarrow \theta_1 \times \dots \times \theta_\ell,$$

$\text{map}\langle N; n \rangle$ denotes a node of type:

$$\tau_1^{\wedge n} \times \dots \times \tau_k^{\wedge n} \rightarrow \theta_1^{\wedge n} \times \dots \times \theta_\ell^{\wedge n}$$

such that

$$(Y_1, \dots, Y_\ell) = \text{map}\langle N; n \rangle(X_1, \dots, X_k)$$

if and only if

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

Example 29 map

$$\text{map} \langle \langle +; 3 \rangle \rangle ([1, 0, 2], [3, 6, -1]) \rightsquigarrow [4, 6, 1]$$

3.2.5 From Boolean arrays to Boolean scalar: boolred

Definition 5: boolred

This iterator has 3 integer static input arguments:

$$\text{boolred}\langle \langle i; j; k \rangle \rangle$$

such that $0 \leq i \leq j \leq k$ and $k > 0$.

It denotes a combinational node whose profile is $\text{bool}^k \rightarrow \text{bool}$, 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 30 boolred

```
#(a1, ..., an) ⇔ boolred(1,1,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 libraries. 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 is even 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 31 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 tab_bool4 = mk_tab<<bool, true, 4>>;

```

Example 32 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 toto_3 = toto_n<<Lustre::iplus, 3>>;

```

nodes can even be defined recursively using the “with” construct

Example 33 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 main = consensus<<8>>;

```

3.4 Packages and models

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

XXX nb: a basic Lustre program is not a valid Lustre V6 program according to those syntax rules. However, basic lustre programs are still accepted by the lustre V6 compiler, which consider that a program without package annotations :

- uses no other package
- provides all the package parameters it defines

```
⟨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⟩ ⟨P⟩
                    [ "uses" ⟨Ident_List⟩ ]
                    "provides" ⟨Package_Params⟩
⟨Package_Params⟩ ::= (⟨Package_Param⟩)+
⟨Package_Param⟩ ::= "const" ⟨Ident⟩ ":" ⟨Type_Identifier⟩ ⟨P⟩ ";"
                  | "type" ⟨Type_Identifier_List⟩ ⟨P⟩ ";"
                  | ⟨Function_Header⟩
                  | ⟨Node_header⟩
⟨Type_Identifier⟩ ::= ⟨Identifier⟩
⟨Type_Identifier_List⟩ ::= ⟨Ident⟩ ";" | ⟨Ident⟩ "," ⟨Type_Identifier_List⟩
```

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

Example 34 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

```

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⟩ ⟨P⟩
                  [ "uses" ⟨Ident_List⟩ ]
                  "needs" ⟨Package_Params⟩
                  "provides" ⟨Package_Params⟩

```

Example 35 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⟩ ")" ⟨P⟩ ";"
⟨Model_Actual_List⟩ ::= ⟨Model_Actual⟩ | ⟨Model_Actual⟩ "," ⟨Model_Actual_List⟩
⟨Model_Actual⟩ ::= ⟨Identifier⟩ ⟨P⟩ | ⟨Expression⟩ ⟨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 36 Model instance

Here is how to obtain packages by instanciating the model given in Example 35:

```

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 37 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 38 The Merge operator

```

type trival = enum { Pile, Face, Tranche };
node merge_node(clk: trival;
  i1 : int when Pile(clk); i2 : int when Face(clk);
  i3 : int when Tranche(clk))
returns (y: int);
let
  y = merge clk
    (Pile: i1)
    (Face: i2)
    (Tranche: i3);
tel

```

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

3.7 A complete example

TO DO !!!add a complete Lustre V6 commented exemple...

Appendix A

Appendix

A.1 The syntax rules summary

$\langle \text{Program} \rangle ::= (\langle \text{Package} \rangle \mid \langle \text{Model} \rangle \mid \langle \text{Model_Instance} \rangle)^*$

$\langle \text{Package} \rangle ::= \langle \text{Package_Header} \rangle [\langle \text{Package_Body} \rangle] \text{“end”}$
 $\langle \text{Package_Header} \rangle ::= \text{“package”} \langle \text{Ident} \rangle \langle \mathcal{P} \rangle$
 $\quad [\text{“uses”} \langle \text{Ident_List} \rangle]$
 $\quad \text{“provides”} \langle \text{Package_Params} \rangle$
 $\langle \text{Package_Params} \rangle ::= (\langle \text{Package_Param} \rangle)^+$
 $\langle \text{Package_Param} \rangle ::= \text{“const”} \langle \text{Ident} \rangle \text{“:”} \langle \text{Type_Identifier} \rangle \langle \mathcal{P} \rangle \text{“;”}$
 $\quad \mid \text{“type”} \langle \text{Type_Ident_List} \rangle \langle \mathcal{P} \rangle \text{“;”}$
 $\quad \mid \langle \text{Function_Header} \rangle$
 $\quad \mid \langle \text{Node_header} \rangle$
 $\langle \text{Type_Identifier} \rangle ::= \langle \text{Identifier} \rangle$
 $\langle \text{Type_Ident_List} \rangle ::= \langle \text{Ident} \rangle \text{“;”} \mid \langle \text{Ident} \rangle \text{“,”} \langle \text{Type_Ident_List} \rangle$

$\langle \text{Model} \rangle ::= \langle \text{Model_Header} \rangle [\langle \text{Body} \rangle] \text{“end”}$
 $\langle \text{Model_Header} \rangle ::= \text{“model”} \langle \text{Ident} \rangle \langle \mathcal{P} \rangle$
 $\quad [\text{“uses”} \langle \text{Ident_List} \rangle]$
 $\quad \text{“needs”} \langle \text{Package_Params} \rangle$
 $\quad \text{“provides”} \langle \text{Package_Params} \rangle$

$\langle \text{Model_Instance} \rangle ::= \text{“package”} \langle \text{Ident} \rangle$
 $\quad [\text{“uses”} \langle \text{Ident_List} \rangle]$
 $\quad \text{“is”} \langle \text{Ident} \rangle \text{“(”} \langle \text{Model_Actual_List} \rangle \text{“)”} \langle \mathcal{P} \rangle \text{“;”}$
 $\langle \text{Model_Actual_List} \rangle ::= \langle \text{Model_Actual} \rangle \mid \langle \text{Model_Actual} \rangle \text{“,”} \langle \text{Model_Actual_List} \rangle$
 $\langle \text{Model_Actual} \rangle ::= \langle \text{Identifier} \rangle \langle \mathcal{P} \rangle \mid \langle \text{Expression} \rangle \langle \mathcal{P} \rangle$

$\langle \text{Identifier} \rangle ::= \langle \text{Ident} \rangle \mid \langle \text{Ident} \rangle ":: $\langle \text{Ident} \rangle$$

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

$\langle \text{Package_Body} \rangle ::= [" \text{body} "] \langle \text{Entity_Decl} \rangle ^ +$
 $\langle \text{Entity_Decl} \rangle ::= \langle \text{Const_Decl} \rangle$
 $\quad \mid \langle \text{Type_Decl} \rangle$
 $\quad \mid \langle \text{Model_Instance} \rangle$
 $\quad \mid \langle \text{Function_Decl} \rangle$
 $\quad \mid \langle \text{Node_Decl} \rangle$

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

$\langle \text{Type_Decl} \rangle ::= " \text{type} " \langle \text{Ident} \rangle ^ + \langle \mathcal{P} \rangle " ; "$
 $\quad \mid " \text{type} " \langle \text{Ident} \rangle " = " \langle \text{Type} \rangle \langle \mathcal{P} \rangle " ; "$
 $\langle \text{Type} \rangle ::= \langle \text{Ident} \rangle \mid \langle \text{Record_Type} \rangle \mid \langle \text{Array_Type} \rangle \mid \langle \text{Enum_Type} \rangle$
 $\langle \text{Record_Type} \rangle ::= " \{ " \langle \text{Field_List} \rangle " \}$
 $\langle \text{Field_List} \rangle ::= \langle \text{Field} \rangle \mid \langle \text{Field} \rangle " ; " \langle \text{Field_List} \rangle$
 $\langle \text{Field} \rangle ::= \langle \text{Ident} \rangle " : " \langle \text{Type} \rangle$
 $\langle \text{Array_Type} \rangle ::= \langle \text{Type} \rangle " ^ " \langle \text{Expression} \rangle$
 $\langle \text{Enum_Type} \rangle ::= " \text{enum} " " \{ " \langle \text{Ident_List} \rangle " \}$

$\langle \text{Function_Decl} \rangle ::= \langle \text{Function_Header} \rangle [\langle \text{FN_Body} \rangle]$
 $\langle \text{Function_Header} \rangle ::= " \text{function} " " (" \langle \text{FN_Params} \rangle ") "$
 $\quad " \text{returns} " " (" \langle \text{FN_Params} \rangle ") " \langle \mathcal{P} \rangle " ; "$
 $\langle \text{Node_Decl} \rangle ::= \langle \text{Node_Header} \rangle [\langle \text{FN_Body} \rangle]$
 $\langle \text{Node_Header} \rangle ::= " \text{node} " " (" \langle \text{FN_Params} \rangle ") "$
 $\quad " \text{returns} " " (" \langle \text{FN_Params} \rangle ") " \langle \mathcal{P} \rangle " ; "$
 $\langle \text{FN_Params} \rangle ::= \langle \text{Var_Decl_List} \rangle$
 $\langle \text{Var_Decl_List} \rangle ::= \langle \text{Var_Decl} \rangle \mid \langle \text{Var_Decl} \rangle " ; " \langle \text{Var_Decl_List} \rangle$
 $\langle \text{Var_Decl} \rangle ::= \langle \text{Ident_List} \rangle " : " \langle \text{Type} \rangle [\langle \text{Declared_Clock} \rangle] \langle \mathcal{P} \rangle$
 $\langle \text{Declared_Clock} \rangle ::= " \text{when} " \langle \text{Clock} \rangle$
 $\langle \text{Clock} \rangle ::= \langle \text{Identifier} \rangle$
 $\langle \text{FN_Body} \rangle ::= [\langle \text{Local_Var_Decl} \rangle] " \text{let} " \langle \text{Equation_List} \rangle " \text{tel} " [" ; "]$
 $\langle \text{Local_Var_Decl} \rangle ::= " \text{var} " \langle \text{Var_Decl_List} \rangle [" ; "]$

$\langle \text{Equation_List} \rangle$::=	$\langle \text{Eq_or_Ast} \rangle$ $\langle \text{Eq_or_Ast} \rangle \langle \text{Equation_List} \rangle$
$\langle \text{Eq_or_Ast} \rangle$::=	$\langle \text{Equation} \rangle$ $\langle \text{Assertion} \rangle$
$\langle \text{Equation} \rangle$::=	$\langle \text{Left_Part} \rangle$ “=” $\langle \text{Right_Part} \rangle$ $\langle \mathcal{P} \rangle$ “;”
$\langle \text{Left_Part} \rangle$::=	“(” $\langle \text{Left_List} \rangle$ “)” $\langle \text{Left_List} \rangle$
$\langle \text{Left_List} \rangle$::=	$\langle \text{Left} \rangle$ (“,” $\langle \text{Left} \rangle$)*
$\langle \text{Left} \rangle$::=	$\langle \text{Identifier} \rangle$ $\langle \text{Left} \rangle$ $\langle \text{Selector} \rangle$
$\langle \text{Selector} \rangle$::=	“.” $\langle \text{Ident} \rangle$ “[” $\langle \text{Expression} \rangle$ [$\langle \text{SelTrancheEnd} \rangle$] “]”
$\langle \text{SelTrancheEnd} \rangle$::=	“..” $\langle \text{Expression} \rangle$
$\langle \text{Assertion} \rangle$::=	“ assert ” $\langle \text{Expression} \rangle$ $\langle \mathcal{P} \rangle$ “;”

$\langle \text{Expression} \rangle$::=	$\langle \text{Identifier} \rangle$ $\langle \text{Value} \rangle$ “(” $\langle \text{Expression_List} \rangle$ “)” $\langle \text{Record_Exp} \rangle$ $\langle \text{Array_Exp} \rangle$ $\langle \text{Unary} \rangle$ $\langle \text{Expression} \rangle$ $\langle \text{Expression} \rangle$ $\langle \text{Binary} \rangle$ $\langle \text{Expression} \rangle$ $\langle \text{Nary} \rangle$ $\langle \text{Expression} \rangle$ “if” $\langle \text{Expression} \rangle$ “then” $\langle \text{Expression} \rangle$ “else” $\langle \text{Expression} \rangle$ $\langle \text{Call} \rangle$ $\langle \text{Expression} \rangle$ $\langle \text{Selector} \rangle$
$\langle \text{Expression_List} \rangle$::=	$\langle \text{Expression} \rangle$ $\langle \text{Expression} \rangle$ “,” $\langle \text{Expression_List} \rangle$
$\langle \text{Record_Exp} \rangle$::=	$\langle \text{Ident} \rangle$ “{” $\langle \text{Field_Exp_List} \rangle$ “}”
$\langle \text{Field_Exp_List} \rangle$::=	$\langle \text{Field_Exp} \rangle$ $\langle \text{Field_Exp} \rangle$ “;” $\langle \text{Field_Exp_List} \rangle$
$\langle \text{Field_Exp} \rangle$::=	$\langle \text{Ident} \rangle$ “=” $\langle \text{Expression} \rangle$
$\langle \text{Array_Exp} \rangle$::=	“[” $\langle \text{Expression_List} \rangle$ “]” $\langle \text{Expression} \rangle$ “^” $\langle \text{Expression} \rangle$
$\langle \text{Call} \rangle$::=	$\langle \text{User_Op} \rangle$ $\langle \mathcal{P} \rangle$ “(” $\langle \text{Expression_List} \rangle$ “)”
$\langle \text{User_Op} \rangle$::=	$\langle \text{Identifier} \rangle$ $\langle \text{Iterator} \rangle$ « $\langle \text{User_Op} \rangle$ “;” $\langle \text{Expression} \rangle$ »
$\langle \text{Iterator} \rangle$::=	“ map ” “ red ” “ fill ” “ fillred ” “ boolred ”

$\langle \text{Unary} \rangle$::=	“-” “ not ” “ mirror ”
$\langle \text{Binary} \rangle$::=	“+” “-” “*” “/” “ div ” “ mod ” “>” “<” “>=” “<=” “<>” “=” “ or ” “ and ” “ xor ” “=>”
$\langle \text{Nary} \rangle$::=	“#” “ nor ”

$\langle \text{Unary} \rangle$::=	“ pre ” “ current ”
$\langle \text{Binary} \rangle$::=	“->” “ when ”

A.2 The syntax rules (automatically generated)

```

program ::= IncludeList PackBody
          | IncludeList PackList
PackList ::= OnePack { OnePack }
OnePack ::= ModelDecl
          | PackDecl
          | PackEq
Include ::= include "<string>"
IncludeList ::= [Include IncludeList ]
Provides ::= [provides ProvideList ; ]
ProvideList ::= Provide { ; Provide }
Provide ::= const Ident : Type ConstDefOpt
          | node Ident Params returns Params
          | function Ident Params returns Params
          | type Ident
          | type Ident = Type
          | type Ident = enum { IdentList }
          | type Ident = OptStruct { TypedValuedIdents }
ConstDefOpt ::= [= Expression ]
ModelDecl ::= model Ident Uses needs StaticParamList ; Provides body
          PackBody end
PackDecl ::= package Ident Uses Provides body PackBody end
Uses ::= [uses IdentList ; ]
Eq_or_Is ::= =
          | is
PackEq ::= package Ident Eq_or_Is Ident ( ByNameStaticArgList ) ;
PackBody ::= DeclList
DeclList ::= OneDecl { OneDecl }
OneDecl ::= ConstDecl
          | TypeDecl
          | ExtNodeDecl
          | NodeDecl
IdentRef ::= <ident>
          | <ident>::<ident>
Ident ::= <ident> Pragma
IdentList ::= Ident { , Ident }
TypedIdentsList ::= TypedIdents { ; TypedIdents }
TypedIdents ::= IdentList : Type
TypedValuedIdents ::= TypedValuedIdent { ; TypedValuedIdent }
TypedValuedIdent ::= Ident : Type
          | Ident , IdentList : Type

```

		<i>Ident</i> : <i>Type</i> = <i>Expression</i>
<i>ConstDecl</i>	::=	const <i>ConstDeclList</i>
<i>ConstDeclList</i>	::=	<i>OneConstDecl</i> ; { <i>OneConstDecl</i> ; }
<i>OneConstDecl</i>	::=	<i>Ident</i> : <i>Type</i>
		<i>Ident</i> , <i>IdentList</i> : <i>Type</i>
		<i>Ident</i> : <i>Type</i> = <i>Expression</i>
		<i>Ident</i> = <i>Expression</i>
<i>TypeDecl</i>	::=	type <i>TypeDeclList</i>
<i>TypeDeclList</i>	::=	<i>OneTypeDecl</i> ; { <i>OneTypeDecl</i> ; }
<i>OneTypeDecl</i>	::=	<i>IdentList</i>
		<i>Ident</i> = <i>Type</i>
		<i>Ident</i> = enum { <i>IdentList</i> }
		<i>Ident</i> = <i>OptStruct</i> { <i>TypedValuedIdents</i> }
<i>OptStruct</i>	::=	[<i>struct</i>]
<i>Type</i>	::=	(bool int real <i>IdentRef</i>) { ~ <i>Expression</i> }
<i>ExtNodeDecl</i>	::=	extern function <i>Ident Params</i> returns <i>Params</i>
		<i>OptSemicol</i>
		extern node <i>Ident Params</i> returns <i>Params OptSemicol</i>
<i>NodeDecl</i>	::=	<i>LocalNode</i>
<i>LocalNode</i>	::=	node <i>Ident StaticParams Params</i> returns <i>Params</i>
		<i>OptSemicol Locals Body OptEndNode</i>
		function <i>Ident StaticParams Params</i> returns <i>Params</i>
		<i>OptSemicol Locals Body OptEndNode</i>
		node <i>Ident StaticParams NodeProfileOpt</i> = <i>EffectiveNode</i>
		<i>OptSemicol</i>
		function <i>Ident StaticParams NodeProfileOpt</i> =
		<i>EffectiveNode OptSemicol</i>
<i>NodeProfileOpt</i>	::=	[<i>Params</i> returns <i>Params</i>]
<i>StaticParams</i>	::=	[« <i>StaticParamList</i> »]
<i>StaticParamList</i>	::=	<i>StaticParam</i> { ; <i>StaticParam</i> }
<i>StaticParam</i>	::=	type <i>Ident</i>
		const <i>Ident</i> : <i>Type</i>
		node <i>Ident Params</i> returns <i>Params</i>
		function <i>Ident Params</i> returns <i>Params</i>
<i>OptEndNode</i>	::=	.
		<i>OptSemicol</i>
<i>Params</i>	::=	()
		(<i>VarDeclList OptSemicol</i>)
<i>Locals</i>	::=	[var <i>VarDeclList</i> ;]
<i>VarDeclList</i>	::=	<i>VarDecl</i> { ; <i>VarDecl</i> }
<i>VarDecl</i>	::=	<i>TypedIdents</i>
		<i>TypedIdents</i> when <i>ClockExpr</i>
		(<i>TypedIdentsList</i>) when <i>ClockExpr</i>

<i>Body</i>	::=	let tel let <i>EquationList</i> tel
<i>EquationList</i>	::=	<i>Equation</i> { <i>Equation</i> }
<i>Equation</i>	::=	assert <i>Expression</i> ; <i>Left</i> = <i>Expression</i> ;
<i>Left</i>	::=	<i>LeftItemList</i> (<i>LeftItemList</i>)
<i>LeftItemList</i>	::=	<i>LeftItem</i> { , <i>LeftItem</i> }
<i>LeftItem</i>	::=	<i>Ident</i> <i>FieldLeftItem</i> <i>TableLeftItem</i>
<i>FieldLeftItem</i>	::=	<i>LeftItem</i> . <i>Ident</i>
<i>TableLeftItem</i>	::=	<i>LeftItem</i> [<i>Expression</i>] <i>LeftItem</i> [<i>Select</i>]
<i>Expression</i>	::=	<i>Constant</i> <i>IdentRef</i> not <i>Expression</i> - <i>Expression</i> pre <i>Expression</i> current <i>Expression</i> int <i>Expression</i> real <i>Expression</i> <i>Expression</i> when <i>ClockExpr</i> <i>Expression</i> fby <i>Expression</i> <i>Expression</i> -> <i>Expression</i> <i>Expression</i> and <i>Expression</i> <i>Expression</i> or <i>Expression</i> <i>Expression</i> xor <i>Expression</i> <i>Expression</i> => <i>Expression</i> <i>Expression</i> = <i>Expression</i> <i>Expression</i> <> <i>Expression</i> <i>Expression</i> < <i>Expression</i> <i>Expression</i> <= <i>Expression</i> <i>Expression</i> > <i>Expression</i> <i>Expression</i> >= <i>Expression</i> <i>Expression</i> div <i>Expression</i> <i>Expression</i> mod <i>Expression</i> <i>Expression</i> - <i>Expression</i> <i>Expression</i> + <i>Expression</i> <i>Expression</i> / <i>Expression</i> <i>Expression</i> * <i>Expression</i> if <i>Expression</i> then <i>Expression</i> else <i>Expression</i> with <i>Expression</i> then <i>Expression</i> else <i>Expression</i>

		# (<i>ExpressionList</i>)
		nor (<i>ExpressionList</i>)
		<i>CallByPosExpression</i>
		(<i>ExpList2</i>)
		[<i>ExpressionList</i>]
		<i>Expression</i> ~ <i>Expression</i>
		<i>Expression</i> <i>Expression</i>
		<i>Expression</i> [<i>Expression</i>]
		<i>Expression</i> [<i>Select</i>]
		<i>Expression</i> . <i>Ident</i>
		<i>CallByNameExpression</i>
		(<i>Expression</i>)
<i>ClockExpr</i>	::=	<i>Ident</i> (<i>Ident</i>)
		<i>Ident</i>
		not <i>Ident</i>
		not (<i>Ident</i>)
<i>PredefOp</i>	::=	not
		fby
		pre
		current
		->
		when
		and
		or
		xor
		=>
		=
		<>
		<
		<=
		>
		>=
		div
		mod
		-
		+
		/
		*
		if
		when <i>ClockExpr</i>
<i>CallByPosExpression</i>	::=	<i>EffectiveNode</i> (<i>Expression</i>)
		<i>EffectiveNode</i> (<i>ExpList2</i>)
<i>EffectiveNode</i>	::=	<i>IdentRef</i>

		<i>IdentRef</i> « <i>StaticArgList</i> »
<i>StaticArgList</i>	::=	<i>StaticArg</i> { , <i>StaticArg</i> ; <i>StaticArg</i> }
<i>StaticArg</i>	::=	type <i>Type</i>
		const <i>Expression</i>
		node <i>EffectiveNode</i>
		function <i>EffectiveNode</i>
		<i>PredefOp</i>
		<i>SimpleExp</i>
		<i>SurelyType</i>
		<i>SurelyNode</i>
<i>ByNameStaticArgList</i>	::=	<i>ByNameStaticArg</i> { , <i>ByNameStaticArg</i> ; <i>ByNameStaticArg</i> }
<i>ByNameStaticArg</i>	::=	type <i>Ident</i> = <i>Type</i>
		const <i>Ident</i> = <i>Expression</i>
		node <i>Ident</i> = <i>EffectiveNode</i>
		function <i>Ident</i> = <i>EffectiveNode</i>
		<i>Ident</i> = <i>PredefOp</i>
		<i>Ident</i> = <i>SimpleExp</i>
		<i>Ident</i> = <i>SurelyType</i>
		<i>Ident</i> = <i>SurelyNode</i>
<i>SurelyNode</i>	::=	<i>IdentRef</i> « <i>StaticArgList</i> »
<i>SurelyType</i>	::=	(<i>bool</i> <i>int</i> <i>real</i>) { ~ <i>Expression</i> }
<i>SimpleExp</i>	::=	<i>Constant</i>
		<i>IdentRef</i>
		(<i>SimpleExp</i>)
		not <i>SimpleExp</i>
		- <i>SimpleExp</i>
		<i>SimpleExp</i> and <i>SimpleExp</i>
		<i>SimpleExp</i> or <i>SimpleExp</i>
		<i>SimpleExp</i> xor <i>SimpleExp</i>
		<i>SimpleExp</i> => <i>SimpleExp</i>
		<i>SimpleExp</i> = <i>SimpleExp</i>
		<i>SimpleExp</i> <> <i>SimpleExp</i>
		<i>SimpleExp</i> < <i>SimpleExp</i>
		<i>SimpleExp</i> <= <i>SimpleExp</i>
		<i>SimpleExp</i> > <i>SimpleExp</i>
		<i>SimpleExp</i> >= <i>SimpleExp</i>
		<i>SimpleExp</i> div <i>SimpleExp</i>
		<i>SimpleExp</i> mod <i>SimpleExp</i>
		<i>SimpleExp</i> - <i>SimpleExp</i>
		<i>SimpleExp</i> + <i>SimpleExp</i>
		<i>SimpleExp</i> / <i>SimpleExp</i>
		<i>SimpleExp</i> * <i>SimpleExp</i>

<i>CallByNameExpression</i>	::=	if <i>SimpleExp</i> then <i>SimpleExp</i> else <i>SimpleExp</i> <i>IdentRef</i> { <i>CallByNameParamList</i> <i>OptSemicol</i> }
<i>CallByNameParamList</i>	::=	<i>IdentRef</i> { }
<i>sepVariant</i>	::=	<i>CallByNameParam</i> { <i>sepVariant</i> <i>CallByNameParam</i> }
	::=	;
<i>CallByNameParam</i>	::=	,
<i>ExpressionList</i>	::=	<i>Ident</i> = <i>Expression</i>
	::=	<i>Expression</i>
		<i>ExpList2</i>
<i>Constant</i>	::=	true
		false
		<integer>
		<floating>
<i>ExpList2</i>	::=	<i>ExpressionList</i> , <i>Expression</i>
<i>Select</i>	::=	<i>Expression</i> .. <i>Expression</i> <i>Step</i>
<i>Step</i>	::=	[<i>step</i> <i>Expression</i>]
<i>OptSemicol</i>	::=	[;]
<i>Pragma</i>	::=	[% <ident> : <ident> % <i>Pragma</i>]

A.3 Lustre History

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

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

- arrays - include - autre ???

[int, real] -> use structures instead

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

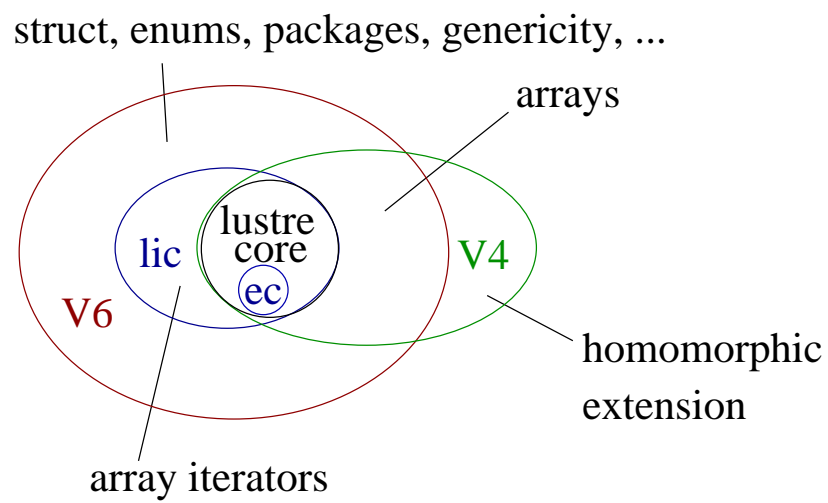


Figure A.1: Lustre potatoes