Institut Polytechnique de Paris
Master 2 - Master Parisien de Recherche en Informatique

January 4, 2026

SYNC- Synchronous Programming of Reactive Systems

An intepreter for a subset of the Lustre language

TABET GONZALEZ Salwa

1 A subset of the Lustre programming language

1.1 Grammar

We consider a deliberately restricted yet expressive subset of the Lustre synchronous dataflow language. The goal of this grammar is to capture the essential constructs required for defining reactive stream computations and node-based modular programs, while remaining suitable to a direct operational interpretation.

A program consists of a finite list of node definitions. Each node introduces a named, reusable stream transformer with explicit inputs and outputs.

A node is defined by:

- a name,
- a list of typed inbput variables,
- a list of typed output variables,
- a body composed of local equations defining output streams and intermediate streams.

Nodes are first-order: they may call other nodes, but recursion is not allowed directly at the syntactic level.

The grammar supports a small set of base types: integers and booleans. All streams are implicitly infinite and synchronous: each variable denotes a value at every logical instant.

Expressions define stream values. The grammar includes:

- constants and variables, with their respective identifiers,
- integer literals
- boolean literals
- variable references
- unary and binary operators
- arithmetic operators on integers (e.g. addition, substraction, multiplication, modulo)

— Boolean operators (e.g. negation, conjunction, disjunction)

— comparison operators producing boolean streams

Then, we defined temporal operators, to express some of the language richness. These are:

— `pre` e, the previous value operator, which accesses the value of expression e at the previous instant,

— e_1 `fby` e_2, "followed by", which defines a stream in initialized with e_1 at the first instant, then continuing with e_2

These operators are the core of Lustre's stateful behavior.

Finally, we included some clocking and control operators:

— `when` e `on` c, which samples stream e when clock c is `true`,

— `merge` c e_1 e_2, which selects between two streams depending on the boolean clock c

Here, clocks are boolean streams. This means that no derived nor polyphonic clocks are included in the subset.

To be able to interpret an early example of the course, we had to implement node calls. Node calls are expressions and may appear wherever an expression is expected. They are of the form `f`$(e_1, ..., e_n)$.

The body of a node is a set of equations of the form:

$$x = e$$

where x is a variable and e an expression. Each variable is defined exactly once. Also, local variables may be introduced implicitly by equations. Finally, all variables are statically scoped within the node.

1.2 Restrictions and design choices

Several restrictions are imposed intentionally. For instance, we decided not to implement the following:

— higher-order nodes or dynamic node creation

— explicit recursion

— polymorphic types

— arrays or records

— side effects outside stream definitions

These choices keep the grammar close to the mathematical core of Lustre, while simplifying both parsing and semantic interpretation. This grammar still captures the essence of Lustre as a synchronous, declarative, stream-based language. Despite its simplicity, the subset is expressive enough to model non-trivial reactive systems and to study precise operational and denotational semantics.

2 Interpreter's front-end

2.1 Lexer

The lexer constitutes the first stage of the interpreter's front-end. Its role is to transform a raw input program, given as a character stream, into a sequence of tokens that can be consumed by the parser. This phase abstracts away low-level textual details such as whitespace and delimiters, and exposes a structured view of the program's syntactic elements.

The lexer recognizes the following categories of tokens:

— Identifiers, used for variable names and node names

— Keywords, corresponding to reserved Lustre constructs (such as `node`, `returns`, `let`, `tel`)

— Literals, including integer constants and boolean constants (`true`, `false`)

— Operators and punctuation, such as arithmetic operators, boolean operators, comparison operators, parentheses, commas, and equation symbols

Whitespace and line breaks are ignored, except when they separate tokens.

The tokenization process is deterministic and total: any well-formed input program in the considered Lustre subset produces a corresponding token stream, while ill-formed character sequences result in a lexical error. These errors are reported explicitly and prevent further stages of the interpreter from running.

By separating lexical analysis from parsing, the implementation follows a classical compiler architecture. This separation simplifies the parser, which can operate over a clean symbolic representation of the program, and allows the grammar to be expressed independently of low-level textual concerns.

2.2 Abstract Syntax Tree

Once the input program has been tokenized by the lexer, the parser constructs an Abstract Syntax Tree (AST) representing the structural organization of the program. The AST provides a syntax-directed, hierarchical representation of Lustre programs, independent of concrete textual details such as parentheses, keywords placement, or formatting.

The AST mirrors the grammar of the considered Lustre subset and serves as the central intermediate representation used by all subsequent phases of the interpreter.

At the top level, a program is represented as a finite list of node declarations. Each node contains:

— a node identifier,

— a list of typed input variables,

— a list of typed output variables,

— a body consisting of a set of equations.

Types in the AST are limited to the base types of the language (`int` and `bool`). Each variable declaration associates an identifier with one of these types.

Expressions form the core of the AST. They are represented inductively and include:

— constants, such as integer and boolean literals,

— variable references, identifying stream variables,

— unary and binary operator applications, covering arithmetic, boolean, and comparison operators,

— temporal constructs, namely pre and fby, which introduce state into stream definitions,

— clocked expressions, including when and merge, used to express conditional stream activation,

— node calls, which apply a node to a list of argument expressions.

Each expression node explicitly records its operator and its sub-expressions, making evaluation order and dependencies explicit in the syntax tree.

The body of a node is represented as a list of equations, each binding a variable to an expression. The AST enforces the single-assignment discipline of Lustre: each variable appears exactly once on the left-hand side of an equation. All variables are statically scoped within their enclosing node.

This abstract representation is intentionally minimal. It omits all concrete syntactic sugar and focuses exclusively on the semantic structure of programs. By doing so, it provides a stable and precise foundation for defining the operational semantics of the language, implementing expression evaluation, and interpreting nodes over logical instants.

2.3 Parser

The parser constitutes the final component of the interpreter's front-end. Its role is to transform the linear sequence of tokens produced by the lexer into a well-formed Abstract Syntax Tree, in accordance with the grammar of the Lustre subset described in 1.1.

The parser is syntax-directed: it follows the structure of the grammar to recognize programs, node declarations, equations, and expressions, and constructs the corresponding AST nodes as it proceeds. At this stage, the hierarchical organization of the program is made explicit, and syntactic precedence and associativity rules are resolved.

Parsing proceeds in several stages. First, the parser recognizes a program as a sequence of node definitions. Each node is parsed by identifying its name, its lists of typed input and output variables, and its body. The body itself is parsed as a collection of equations, each associating a variable identifier with an expression.

Expressions are parsed recursively, respecting operator precedence and syntactic structure. The parser distinguishes between:

— atomic expressions (constants and variable references),

— unary and binary operator applications,

— temporal constructs (pre, fby),

— clocking constructs (when, merge),

— node calls with their argument lists.

Special care is taken to correctly parse temporal and clocked operators, whose syntax introduces non-trivial nesting and precedence constraints.

The parser performs syntactic validation of the input program. Ill-formed programs, such as malformed expressions, missing delimiters, or unexpected tokens, are rejected at this stage and reported as parsing errors. However, the parser does not enforce semantic properties. These aspects are handled later by the semantic interpretation.

By separating parsing from both lexical analysis and semantic evaluation, the implementation adheres to a classical front-end architecture. The parser produces a clean, structured AST that faithfully reflects the grammatical structure of the source program and serves as the input for the semantic and interpretative phases of the interpreter.

Now that the front-end plumbing part is done, we have to move from syntax to semantics. We will define small-step stream semantics, with a state that stores previous values (`pre`, `fby`) and current clocks, as well as a single-tick evaluator for expressions and a node interpreter that executes one logical instant.

3 Interpreter

Having defined the syntax of our Lustre subset and constructed a front-end capable of producing well-formed abstract syntax trees, we now turn to the semantic interpretation of programs. The goal of this section is to make precise how Lustre programs compute. That is, how streams evolve over logical time and how expressions are evaluated at each instant.

3.1 Defining semantics

Lustre is a synchronous language: computation proceeds in discrete, global logical instants. At each instant, all streams conceptually produce a value simultaneously. Time is therefore modeled as a sequence of "ticks", indexed implicitly by natural numbers. There is no notion of partial execution or interleaving: either a value exists at a given instant, or it is absent due to clocking constructs.

Each evaluation step corresponds to exactly one logical instant.

Streams and state Every variable in a Lustre program denotes a stream (i.e. an infinite sequence of values indexed by logical time). However, evaluation at a given instant may depend on values computed at previous instants, as we immediately think of the construct `pre`.

To account for this, we introduce an explicit state in the semantics. Intuitively, the state stores:

— previous values of expressions needed to evaluate `pre`,

— initialization information for `fby`,

— intermediate values required for node calls across instants.

The state is updated deterministically at each tick, reflecting the evolution of streams over time.

Environments At each logical instant, evaluation occurs relative to an environment mapping variables to their current values. Environments represent the instant values of all input, output, and local streams at a given tick.

Inputs are provided externally by the execution context. Outputs and local variables are computed internally by evaluating the equations of the node. Environments are immutable during the evaluation of a single tick.

Expression semantics The semantics of expressions is defined per instant, but parameterized by the current state and environment. An expression may:

— produce a value immediately (constants, variables, operators),

— consult the previous state (`pre`),

— conditionally produce a value depending on clocks (`when`, `merge`),

— invoke another node (node calls) for multi-node programs.

Importantly, expression evaluation is pure with respect to the current instant: it does not mutate the state directly. Instead, any state evolution is captured separately and made explicit when moving to the next tick.

This separation allows us to define expression semantics independently of node execution order and supports a clean, compositional interpretation.

Node semantics Nodes are interpreted as stream transformers: given input streams, they produce output streams over time. Semantically, executing a node for one instant consists of:

— receiving the current input values,

— evaluating all equations of the node in a well-defined order,

— producing output values and an updated state for the next instant.

Node calls are handled by maintaining separate states for each invoked node instance, ensuring that stateful behavior remains local and well-scoped.

Small-step operational perspective Overall, we adopt a small-step operational semantics. Each semantic step corresponds to:

— evaluating expressions at the current tick,

— updating the state,

— advancing time by one instant.

This approach is well-suited to an interpreter implementation and aligns with Lustre's execution model.

The next sections make this semantics concrete. For instance, in 3.2 we define how expressions are evaluated for a single instant, and in 3.3 we lift this evaluation to nodes, describing how complete (subset of) Lustre programs execute over time.

3.2 Evaluating expressions

This section defines the semantics of expression evaluation at a single logical instant. Expressions form the computational core of Lustre programs: they specify how stream values are produced from inputs, previous values, and intermediate computations. In our implementation, this semantics is captured by the expression evaluator defined in eval_expr.v.

Instantaneous evaluation Expression evaluation is defined per logical instant. At a given tick, an expression is evaluated relative to:

— the current environment, mapping variables to their instantaneous values,

— the current state, which stores information about previous instants (needed for temporal operators and node calls).

In our implementation, this corresponds to the central evaluation function `eval_expr`, which takes as arguments:

— the current program (for resolving node calls),

— the current state,

— the current environment,

— the expression to be evaluated.

The result of evaluation is either a value paired with an updated state, or a failure indicating that the expression cannot produce a value at this instant.

Values and absence The evaluator distinguishes between present values, represented explicitly as values, and absence, represented by failure of evaluation. This distinction is essential to correctly model clocked expressions. In particular, the absence of a value is not silently replaced by a default: it is propagated explicitly through evaluation. This behavior is reflected in the return type of `eval_expr`, which uses an option structure to model possible absence.

Pure expressions Basic expressions are evaluated in a purely functional manner: integer and boolean constants are evaluated directly, while variable references are resolved by looking up the variable in the environment. Besides, unary and binary operators are evaluated by recursively evaluating their sub-expressions and applying the corresponding operator.

These cases correspond directly to the non-temporal constructors handled in `eval_expr`. They do not consult the state and do not introduce any temporal dependency. As a result, their evaluation is instantaneous.

Temporal operators Temporal operators introduce dependencies on previous instants and therefore consult the state.

Let's look at the `pre` e case : the expression evaluates to the value of e at the previous instant. In the implementation, this is handled by looking up the stored previous value of e in the state. At the first instant, no such value exists, and evaluation fails. This behavior is encoded explicitly in the corresponding `eval_expr` case for pre.

Then, we have the $e1$ `fby` $e2$ case : the expression produces the value of $e1$ at the first instant, and behaves like $e2$ at subsequent instants. The evaluator distinguishes these cases by consulting the state and recording initialization information. This ensures that `fby` introduces stateful behavior.

`eval_expr` does not update the state itself: it only reads from it. State updates are performed later, when moving to the next logical instant at the node level.

Clocked expressions This was probably the most difficult part of the interpreter, up to the implementation. The proofs were also very difficult to do, but we will discuss them in 4. Clocking constructs control whether an expression is active at a given instant.

— e `when` c : The expression is evaluated only if the clock expression c evaluates to `true`. Otherwise, evaluation yields absence. In our implementation, this corresponds to first evaluating c using `eval_expr`, and conditionally evaluating e depending on the result.

— `merge` c e_1 e_2 : The merge construct selects which expression to evaluate based on the boolean value of c. If c is true, e_1 is evaluated. Otherwise, e_2 is evaluated. This behavior is directly reflected in the corresponding branch of `eval_expr`.

These constructs allow conditional stream activation without breaking the global synchrony assumption.

Node calls Node calls are expressions and are evaluated as part of expression evaluation. Evaluating a node call involves evaluating all argument expressions at the current instant, invoking the semantics of the called node and returning the resulting output value and updated state.

In our implementation, this behavior is delegated from `eval_expr` to the node evaluator (defined later in 3.3), ensuring a clean separation between expression-level and node-level semantics. Each node call is associated with its own state, guaranteeing that stateful behavior is properly encapsulated.

3.3 Evaluating nodes

We now explain how nodes are evaluated, building on the expression semantics defined in 3.2. While expressions describe how individual values are computed at a given instant, nodes describe how collections of equations are evaluated together to produce outputs and update the state.

In our implementation, node evaluation is defined in eval_node.v, and it proceeds in a simple two-phase way. This strategy corresponds to the classical synchronous execution model underlying Lustre, in which each logical instant consists of an atomic reaction computing current values, followed by a state update preparing the next instant. This separation must be made explicit in an operational interpreter to correctly handle temporal operators and node calls. Thus, evaluating a node for one logical instant means:

- taking the current input values,
- computing all local and output variables for this instant,
- producing the outputs,
- and preparing the state for the next instant.

This behavior is implemented by the main node evaluator `eval_node`, which is responsible for executing one tick of a node.

There are two phases (passes) in this file.

3.3.a Computing values for this instant

The first phase consists in computing the values of all equations of the node for the current instant. In more concrete terms:

- the evaluator starts from an environment containing the input variables,
- it then evaluates the node's equations one by one,
- each equation evaluates its right-hand side expression using `eval_expr`,
- the resulting value is added to the environment.

This phase is implemented by the function `eval_equations`, which takes as input:

- the current program,

— the current state,

— the current environment,

— the list of equations of the node.

As output, we obtain a complete environment containing inputs, locals, and outputs for this instant, but the state has not been updated yet. In other words, given the current inputs and the previous state, it says which are all the values "right now".

3.3.b Updating the state for the next instant

Once all values for the current instant are known, the evaluator moves to updating the state. This second phase is needed for temporal constructs such as `pre`, `fby`, and node calls. These constructs require remembering values so they can be reused at the next instant.

In the implementation, this phase extracts the relevant values from the environment, and stores them in the state in the appropriate places. This logic is handled inside `eval_node`, after `eval_equations` has finished successfully. In other words, this phase states what we need to remember so that the next tick can be evaluated correctly.

After both phases, the evaluator returns the output values for the current instant, along with the updated state. These results correspond exactly to one logical tick of the node.

Repeated execution over time

Evaluating a node once only gives the result for a single instant. To execute a node over several instants, the interpreter repeatedly applies node evaluation, threading the state from one instant to the next.

This repeated execution is implemented by `run_node`, which executes a node for a fixed number of ticks, and `run_top`, which selects an entry node and runs it as a complete program.

Each iteration corresponds to one logical instant:

1. inputs are provided,
2. `eval_node` is called,

3. outputs are produced,
4. the state is updated.

Without this two-phased evaluation, executing the program that we saw in class about `tf_count` resulted in a single "count" result instead of two interleaved counts that got active alternatingly.

4 Proving properties about the interpreter

With the interpreter now fully defined, we can turn to reasoning about its behavior. The operational semantics introduced in the previous sections is precise enough to support formal arguments about correctness and well-definedness. In particular, the explicit handling of environments, state, and logical instants makes it possible to state and prove properties about determinism, progress of evaluation, and the evolution of state over time.

In this section, we use these definitions to establish several (but far from exhaustive) fundamental properties of the interpreter, illustrating how the semantic structure of the Lustre subset supports reasoning about reactive programs.

4.1 Environment lemmas

The proofs in EnvLemmas.v establish fundamental invariants about runtime environments. They show, for example, that looking up a variable after updating an environment yields the newly bound value, and that updating one variable does not affect the values of other variables.

These properties are relied upon implicitly throughout the interpreter, in particular during expression evaluation and equation execution, where environments are incrementally extended with new computed values. By isolating these results in a dedicated file, the rest of the development can assume standard environment behavior without repeatedly reproving these facts. However, we ultimately didn't go very far in proving our interpreter, so some of these lemmas were left untouched later.

4.2 Evaluating expressions lemmas

The file EvalExprLemmas.v contains auxiliary lemmas about expression evaluation. These lemmas establish structural and invariance properties of the expression evaluator, such as how evaluation interacts with the environment, memory, and call state.

In particular, they are used to show that evaluation preserves certain structural invariants, for example that evaluation does not consume more memory or call-state than anticipated, and that skipped subexpressions behave consistently with evaluated ones in terms of resource consumption.

These results are essential for reasoning about the correctness and robustness of the interpreter. However, we didn't have enough time to go further in our reasoning.