

# A Compiler for mini-Lustre

The purpose of this exercise is to write a code generator for mini-Lustre, a small language similar to Lustre (the syntax and primitives of mini-Lustre are given in the appendix). The compilation scheme is decomposed into a series of elementary transformations until the generation of sequential code.

## 1 Code Generation

### 1.1 Normalisation

Normalisation is the transformation of a mini-Lustre program into a new mini-Lustre program in which expressions  $\langle expr \rangle$  on the right of equations have the following form :

$$\begin{aligned}\langle atom \rangle &:= \langle ident \rangle \mid \langle const \rangle \\ \langle bexpr \rangle &:= \langle atom \rangle \mid ( \langle bexpr \rangle ) \mid \langle bexpr \rangle \langle op \rangle \langle bexpr \rangle \mid \langle unop \rangle \langle bexpr \rangle \mid \\ &\quad \text{if } \langle bexpr \rangle \text{ then } \langle bexpr \rangle \text{ else } \langle bexpr \rangle \mid ( \langle bexpr \rangle , \langle bexpr \rangle^+ ) \\ \langle expr \rangle &:= \langle bexpr \rangle \mid \langle ident \rangle ( \langle bexpr \rangle^* , ) \mid \\ &\quad \langle atom \rangle \text{ fby } \langle atom \rangle \mid ( \langle atom \rangle , \langle atom \rangle^+ ) \text{ fby } ( \langle atom \rangle , \langle atom \rangle^+ )\end{aligned}$$

This program transformation may introduce new equations. For example, the normalisation of the following program :

```
node f(x:int) returns (o:int);
let
  o = 1 fby 2 fby x;
tel
```

```
node g() returns (o:int);
let
  o = f(f(1));
tel
```

results in :

```
node f(x:int) returns (o:int);
var aux1, aux2 : int;
let
  aux1 = 2 fby x;
  aux2 = 1 fby aux1;
  o = aux2;
tel
```

```

node g() returns (o:int);
var aux: int;
let
  aux = f(1);
  o = f(aux);
tel

```

## 1.2 Static Scheduling

The second transformation is static scheduling. This transformation orders equations from a node so that they can be executed sequentially. The order between equations should respect the causality relation between variables. Precisely, an equation  $x = e$  must be scheduled after all the variables read in  $e$  have been scheduled. An equation  $x = v \text{ fby } e$  must be scheduled before all equations that read  $x$ .

## 1.3 Generating Sequential OCaml Code

**Important.** In the following, we suppose that both normalisation and scheduling have been done.

**Remark.** To help you understand what is required in this exercise, we provide a complete version of the `minilustre` compiler. It is available with the other files on the website.

### 1.3.1 General Principals

A particular point to take care of in the compilation of mini-Lustre is the translation of equations containing the `fby` operator.

Consider the example of an equation  $x = e1 \text{ fby } e2$ . At the initial instant  $t_0$ , the data-flow  $x$  equals  $e1$  then, at the later instant  $t_i$ ,  $x$  has the value  $e2$  had at instant  $t_{i-1}$ . The compilation of such an expression needs a way to “remember” the value  $e2$  had at instant  $t_{i-1}$  in order to compute the value of  $x$  at instant  $t_i$ . This is achieved by allocating a memory cell to store the current value of  $e2$ .

Memory cells for `fby` expressions must be allocated for every call to a node. Indeed, consider :

```

node f(x:int) returns (o:int);
let
  o = 2 fby (o+x);
tel

node g() returns (o1,o2:int);
let
  o1 = f(5);
  o2 = f(10);
tel

```

The evaluation of equations  $o1 = f(5)$  and  $o2 = f(10)$  in the node `g` requires the allocation of two different memory cells for the equation  $o = 2 \text{ fby } (o+x)$  in node `f` (since the data-flow  $o$  returned by node `f` is different for every value of argument  $x$ ).

These memory cells being different for different calls to node `f`, they become supplementary arguments to every call to `f`. These node memories are called *internal memories* in the following. All the memories associated to `fby` expressions from the same node will be gathered in a single data-structure to minimize the number of extra arguments of a node.

The remaining describes this compilation principle into OCaml more precisely.

**Important.** To simplify the compilation process, we suppose that the left argument of `fby` is a constant expression (constants or tuples of constants).

### 1.3.2 Compilation of a node

The compilation of a node from mini-Lustre into OCaml produces the three following definitions :

- un record type to represent the internal memory of the node ;
- a function to allocate and initialize this memory ;
- and a transition function to compute a reaction (this function is parameterized by the memory of the node and its current inputs).

Let us illustrate the compilation process on the following example :

```
node minmax (x:int) returns (min,max:int);
var pmin,pmax:int; first:bool;
    aux1, aux2: int;
let
  first = true fby false;
  (aux1, aux2) = (0,0) fby (min, max);
  (pmin, pmax) = if first then (x, x) else (aux1, aux2);
  (min, max) = if x < pmin then (x, pmax)
               else if x > pmax then (pmin, x)
               else (pmin, pmax);
tel
```

The record type produced for this node is :

```
type minmax_mem =
{ mutable next_first: bool;
  mutable next_aux1: int;
  mutable next_aux2: int; }
```

It contains three *mutables* fields `next_first`, `next_aux1` and `next_aux2`. The first corresponds to the memory used to compile the equation `first = true fby false`. The following two fields are used to store the state of equation `(aux1,aux2) = (0,0) fby (min,max)`. These fields will contain at every instant the value of the data-flow for the next step. The record structure is allocated by the initialization function for node (`minmax_init` dans la suite), which initializes every field with the left hand side of an expression `fby`. The record is updated by the transition function `minmax_step`.

The initialization function for node `minmax` is defined in the following way :

```

let minmax_init () =
  { next_first = true;
    next_aux1 = 0;
    next_aux2 = 0; }

```

Finally, the transition function is given below :

```

let minmax_step mem x =
  let first = mem.next_first in
  let (aux1, aux2) = (mem.next_aux1, mem.next_aux2) in
  let (pmin, pmax) = if first then (x, x) else (aux1, aux2) in
  let (min, max) = if x < pmin then (x, pmax)
                    else if x > pmax then (pmin, x)
                    else (pmin, pmax) in
  mem.next_first <- false;
  mem.next_aux1 <- min;
  mem.next_aux2 <- max;
  (min, max)

```

This function takes the internal memory of the node and its current inputs. The body of the function can be decomposed into three parts :

- the computation of the current value of every data-flow : `first`, `aux1`, `aux2`, `pmin`, `pmax`, `min` and `max`
- the update of memories for the right hand side of `fby` expressions : `mem.next_first`, `mem.next_aux1` and `mem.next_aux2`
- the return of output data-flows computed by the node : `(min,max)`

**Calling a node.** Consider the code generated for the call of a node on the exemple below :

```

node minmax2(x,y: int) returns (min,max: int);
var min_x, max_x, min_y, max_y: int;
let
  (min_x, max_x) = minmax(x);
  (min_y, max_y) = minmax(y);
  min = if min_x < min_y then min_x else min_y;
  max = if max_x > max_y then max_x else max_y;
tel

```

The record type that is produced for this node is :

```

type minmax2_mem =
  { mem1: minmax_mem;
    mem2: minmax_mem; }

```

It contains two fields `mem1` and `mem2` of type `minmax_mem` which correspond to the memory state of the two calls to `minmax`. These fields are given as arguments to the function `minmax_step`. The initialization function of the node is defined in the following way :

```

let minmax2_init () =
  { mem1 = minmax_init();
    mem2 = minmax_init(); }

```

Every call to this function creates a value of type `minmax2_mem` which corresponds to the memory necessary to execute the `minmax2` node. This memory contains the memory fields `mem1` and `mem2` that are necessary to execute the two calls to `minmax`. These two fields are initialized by calling the function `minmax_init`.

The transition function `minmax2_step` is defined by :

```
let minmax2_step mem (x, y) =
  let (min_x, max_x) = minmax_step mem.mem1 x in
  let (min_y, max_y) = minmax_step mem.mem2 y in
  let min = if min_x < min_y then min_x else min_y in
  let max = if max_x > max_y then max_x else max_y in
  (min, max)
```

`mem.mem1` and `mem.mem2` are the two local memories.

**Remark.** The compilation of function calls to primitives does not need extra memories.

### 1.3.3 The Main Program

The execution of a program from mini-Lustre never ends. It is an infinite sequence of calls to the transition function of the main node. If the node `main` is the main node of the program (its type signature must be `unit→unit`), the corresponding driver code resembles :

```
let _ =
  let mem = main_init () in
  Graphics.open_graph "";
  Graphics.auto_synchronize false;
  while true do
    Graphics.clear_graph ();
    main_step mem ();
    Graphics.synchronize();
    wait()
  done
```

## 2 The compiler

You can download the compiler of mini-Lustre with “holes” at [http://www.di.ens.fr/~pouzet/cours/parallele\\_et\\_reactif/cours4/mini-lustre.tgz](http://www.di.ens.fr/~pouzet/cours/parallele_et_reactif/cours4/mini-lustre.tgz). This archive contains two directories. The directory `examples` contains programs written in mini-Lustre and the directory `compiler` contains the following files :

- `Makefile` : to compile the compiler by entering `make` ;
- `asttypes.mli` : the definition of types used in the abstract syntax tree ;
- `ast.mli` : the definition of types for the abstract syntax tree for mini-Lustre obtained after the syntactic analysis ;
- `typed_ast.mli` : the definition of types to represent the abstract syntax tree annotated with type annotations ;
- `imp_ast.mli` : the definition of types to represent abstract syntax tree from the target imperative language ;

- `lexer.mll` : lexical analysis ;
- `parser.mly` : syntax analysis ;
- `typing.ml` : functions to typecheck a program ;
- `normalization.ml` : functions that implement the normalization step ;
- `scheduling.ml` : functions to schedule typed syntax trees ;
- `imp.ml` : functions to translate the typed abstract syntax into the abstract syntax of the target imperative language ;
- `ocaml_printer.ml` : printer for the imperative target language into OCaml code ;
- `typed_ast_printer.ml` : functions to print a typed abstract syntax tree ;
- `typed_ast_utils.ml` : functions to browse the typed abstract syntax tree ;
- `checks.ml` : a verification procedure to check that transformations are correct ;
- `minilustre.ml` : main entry of the compiler.

The compiler takes a file mini-Lustre (extension `.mls`) and generates OCaml code. The option `-main` defines the name of the main node and causes the driver loop to be generated. The option `-verbose` causes the source code to be printed after every transformation.

### Question 1

Download the compiler then compile and execute the corresponding code for program `simple.mls`.

### Question 2

Carefully read the files `typed_ast.mli` and `imp_ast.mli` that represent abstract syntax trees that you have to manipulate.

### Question 3

Complete places with holes in the files `normalization.ml`, `scheduling.ml` and `imp.ml`. These places are indicated with a `(* TODO *)`.

## A Syntax of the language

We use the following notation in the grammar :

$\langle rule \rangle^*$	repetition of the rule $\langle rule \rangle$ an arbitrary number of times (possibly zero)
$\langle rule \rangle_t^*$	repetition of the rule $\langle rule \rangle$ an arbitrary number of time (possibly zero), every occurrence being separated by the terminal $t$
$\langle rule \rangle^+$	repetition of the rule $\langle rule \rangle$ at least once
$\langle rule \rangle_t^+$	repetition of the rule $\langle rule \rangle$ at least once, occurrences being separated by the terminal $t$
$\langle rule \rangle?$	optional use of the rule $\langle rule \rangle$ (i.e. zero or once)
$( \langle rule \rangle )$	parenthesis ; be careful not to mix them with terminals $($ and $)$

Spaces, tagulation and new lines are blanks. Comments start with `/*` and ends at `*/`, and they must not be nested. Identifiers follow the regular expression syntax  $\langle ident \rangle$  defined below :

$$\begin{aligned}
 \langle digit \rangle &::= 0-9 \\
 \langle alpha \rangle &::= a-z \mid A-Z \\
 \langle ident \rangle &::= \langle alpha \rangle (\langle alpha \rangle \mid - \mid \langle digit \rangle)^*
 \end{aligned}$$

The following identifiers are keywords :

and	bool	const	else	false	fby	float
if	int	let	node	not	or	returns
string	tel	then	true	unit	var	

Integer constants are defined by the following regular expression for  $\langle integer \rangle$  :

$$\langle integer \rangle ::= \langle digit \rangle^+$$

and floating point values by  $\langle float \rangle$  :

$$\begin{aligned} \langle exponent \rangle &::= (e \mid E) (- \mid +)? \langle digit \rangle^+ \\ \langle float \rangle &::= \langle digit \rangle^+ . \langle digit \rangle^* \langle exponent \rangle ? \\ &\quad \mid \langle digit \rangle^* . \langle digit \rangle^+ \langle exponent \rangle ? \\ &\quad \mid \langle digit \rangle^+ \langle exponent \rangle \end{aligned}$$

Character strings are delimited by the character ". They can contain any character, excepted ", \ and new line. These three special characters must be encoded (inside a string) by the sequences "\", "\\" and "\n", respectively.

The grammar for source files is given below. The input entry is the non terminal  $\langle fichier \rangle$ .

$$\begin{aligned} \langle fichier \rangle &::= \langle noeud \rangle^* \text{ EOF} \\ \langle noeud \rangle &::= \text{ node } \langle ident \rangle ( \langle decl \rangle^* ; ) \text{ returns } ( \langle decl \rangle^+ ) ; \langle local \rangle ? \text{ let } \langle eq \rangle^+ \text{ tel} \\ \langle decl \rangle &::= \langle ident \rangle^+ : \langle type \rangle \\ \langle local \rangle &::= \text{ var } \langle decl \rangle^+ ; \\ \langle eq \rangle &::= \langle motif \rangle = \langle expr \rangle ; \\ \langle motif \rangle &::= \langle ident \rangle \mid ( \langle ident \rangle , \langle ident \rangle^+ ) \\ \langle expr \rangle &::= \langle ident \rangle \mid \langle const \rangle \mid \langle expr \rangle \langle op \rangle \langle expr \rangle \mid \langle unop \rangle \langle expr \rangle \mid \\ &\quad \langle ident \rangle ( \langle expr \rangle^* ) \mid \text{ if } \langle expr \rangle \text{ then } \langle expr \rangle \text{ else } \langle expr \rangle \\ &\quad \mid ( \langle expr \rangle , \langle expr \rangle^+ ) \mid \langle expr \rangle \text{ fby } \langle expr \rangle \mid ( \langle expr \rangle ) \\ \langle op \rangle &::= + \mid - \mid * \mid / \mid +. \mid -. \mid *. \mid /. \mid \\ &\quad <= \mid >= \mid > \mid < \mid <> \mid = \mid \text{ and } \mid \text{ or} \\ \langle unop \rangle &::= - \mid -. \mid \text{ not} \\ \langle const \rangle &::= \text{ true } \mid \text{ false } \mid \langle integer \rangle \mid \langle float \rangle \mid \langle string \rangle \mid ( ) \\ \langle type \rangle &::= \text{ bool } \mid \text{ int } \mid \text{ float } \mid \text{ string } \mid \text{ unit} \end{aligned}$$

The language mini-Lustre contains a set of primitives gathered into four groups. The first gathers primitives to convert base types.

```
float_of_int : int → float
int_of_float : float → int
float_of_string : string → float
int_of_string : string → int
bool_of_string : string → bool
```

The second set contains two random number generators and two trigonometric functions :

```
random_int : int → int
random_float : float → float
cos : float → float
sin : float → float
```

The third set contains primitives for graphics :

```
draw_point : (int, int) → unit
draw_line : (int, int, int, int) → unit
draw_circle : (int, int, int) → unit
draw_rect : (int, int, int, int) → unit
fill_rect : (int, int, int, int) → unit
get_mouse : unit → (int,int)
```

Finally, the last set contains two input/output primitives : the function **read**, with signature **unit** → **string**, reads characters on the standard input until a new line, and it returns the input string, and the primitive **print** takes an n-tuple of arguments with arbitrary size and has a returned value of type **unit**.