Lecture notes – Compilation with OCaml

Langages et logique – ELU 610

Compilation is based first and foremost on the recognition of programs in a stream of characters. The purpose of this section is to discover the practical aspects of *lexical analysis* and *parsing*. Lexical analysis consists in recognizing words of our language in sequences of characters. It is generally followed by parsing that groups these words together to recognize sentences. In the domain, we speak of *token* for words and *syntactic units* for sentences. For the purposes of the following phases of compilation, syntactic units are built in the form of a tree: the so called *Abstract Syntax Tree (AST)*. The figure 1 represents the chaining of these two transformations representing what sometimes is called the front end of the compiler. It aims at building a data structure representing the program in an efficient way. In red are displayed examples of results of each phase. It should be noted that, in general, the lexical analyzer, often called a *lexer*, is driven by the parser that requests tokens whenever it needs them. Hence, the next commands in red.

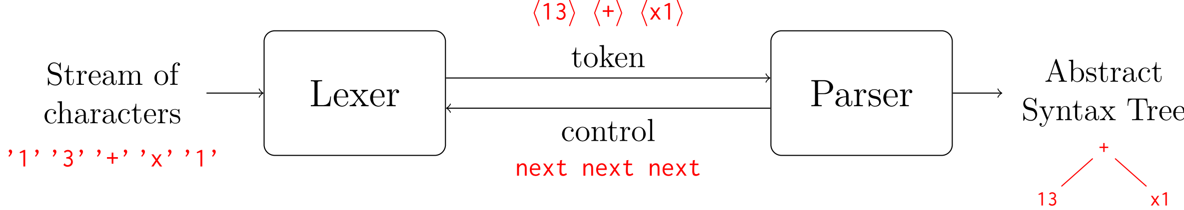

Figure 1: A compiler front end: lexical analysis and parsing.

In general, these analyzers are efficient automata that search for lexical and syntactic patterns using regular expressions. It is difficult and cumbersome to implement an automata. Therefore, *Domain Specific Languages (DSL)* have been proposed to make it easier and smoother.

1 The tools

During this module, we will discover the tools OCamllex whose command is `ocamllex` and Menhir whose command is `menhir`. They are OCaml versions of the standard Unix tools `lex` and `yacc`. The OCaml manual contains a description of `ocamllex` at `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html` and the Menhir page contains its documentation `http://gallium.`

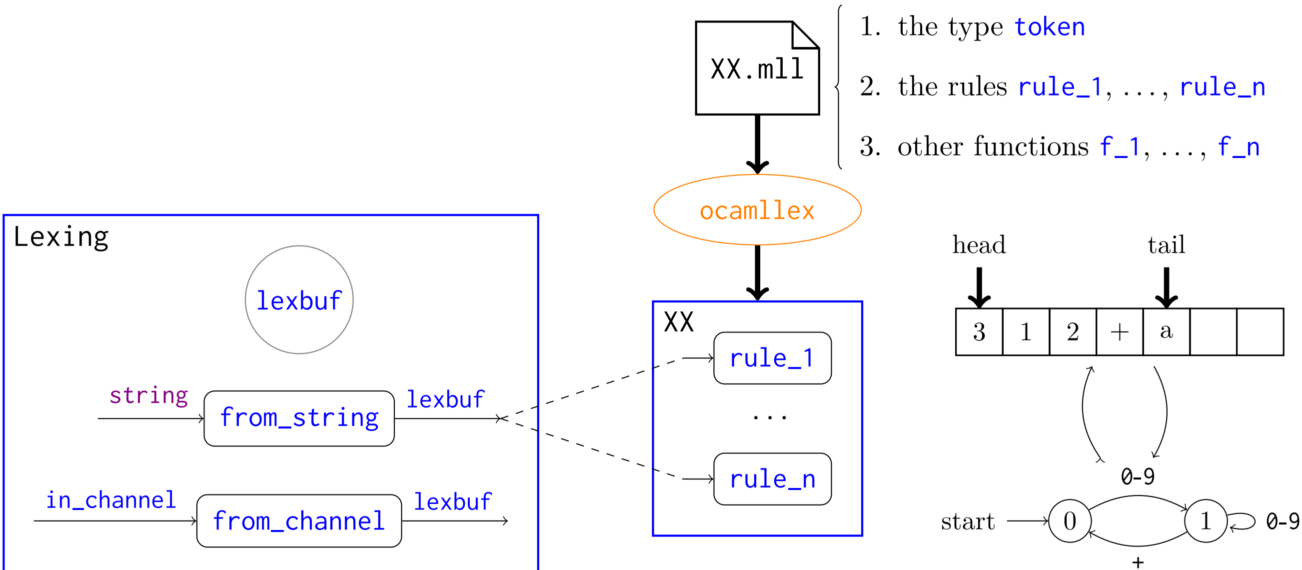

Figure 2: Ocamllex, the big picture.

inria.fr/~fpottier/menhir/menhir.html.fr. The chapter 16 of the *Real World OCaml* book can also help you.

OCamllex allows the construction of lexical analyzers as deterministic automatas. It uses a DSL to specify using rules the actions to be executed when recognizing strings. From these rules, it produces an OCaml module. It is, most of the time, used to transform a buffer of characters into a sequence of tokens.

Menhir allows to build easily deterministic automata for parsing. It also provides a DSL dedicated to the specification of rules describing the actions to execute when recognizing sentences. From these rules, it produces an OCaml module. It is, in general, used to transform a sequence of tokens into an AST.

The `menhir` tool produces an LR(1) stack automaton. To use `menhir`, it will need to be installed by `opam install menhir`.

2 Using Ocamllex

2.1 The syntax of Ocamllex

The files for Ocamllex have the extension `.mll` and follow the following structure:

```
{ (* OCaml code: optional prelude *) }
(* useful regular expressions only for regexp part *)
let ident = regexp
let ident = regexp
(* a group of rules *)
rule ident [ident1 ... identn] = parse
| regexp { (* OCaml code *) }
| regexp { (* OCaml code *) }
(* another  group of rules *)
```

```
and ident [ident1 ... identn] = parse
...
{ (* OCaml code: optional postlude *) }
```

The two sections with Ocaml code at the beginning and end of the file are optional. They contain code defining elements (types, functions, etc.) needed for the actions of the rules. The last section can define functions using the functions corresponding to the rules given in the middle section.

The series of statements `let` preceding the definition of rules allows you to name regular expressions. These regular expression can then be reused in the definition of the rules using their names. The regular expressions of Ocamllex follow the syntax presented Figure 3. Here are some examples:

- `[' ' '\014' '\t' '\012']+` ⇒ at least one space
- `(['\n' '\r'] | "\r\n")` ⇒ newline
- `[^ '\n' '\r']` ⇒ any character except newline
- `"//"[^ '\n' '\r']*` ⇒ C like line comment
- Suppose
  ```
  let digit = ['0'-'9']
  let letter = ['a'-'z''A'-'Z']
  let id_char = (letter | digit | '_')
  ```
- `letter id_char* as id` ⇒ identifiers, `id` contains the result
- `digit+ as nb` ⇒ integers
- `digit* '.' digit* (['e' 'E'] ['+' '-']? digit+)? as nb` ⇒ floating point numbers

Each rule, defined by the keyword `rule`, produces a function with the same name (it must be a valid OCaml identifier). If the rule has arguments, the function obtained will have these arguments in addition to one argument (the last) which is a buffer of type `Lexing.lexbuf`. The behavior of this function is to search for a regular expression from its definition list that represents a buffer prefix. The regular expression corresponding to the longest possible prefix is selected [1] and the associated action is executed. The module `Lexing` contains some lexical buffers manipulation functions that the developer can use to define his treatments. This module contains, among other things [2]:

- the type `lexbuf`;
- two constructors for this type: `from_channel` and `from_string` which respectively creates a buffer from an input-output channel or a string;
- the functions:
  - `lexeme`: returns the string recognized by the regular expression. In general, it is easier to use the syntactic construct `as` allows more easily to extract a sub-part of the recognized chain.

[1] If there are two regular expressions recognizing strings of the same size, the first in the definition order is used.
[2] For more details: `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html`.

Expression	Meaning
' *char* '	the character *char*
_	any character
eof	end of input
" *string* "	the string *string*
[*ens*]	any character from *ens*, may contain 'c1'-'c2' (all characters between c1 and c2)
[^ *ens*]	any character not in *ens*
regexp *	0 or many time the string matching *regexp*
regexp +	1 or many time the string matching *regexp*
regexp ?	nothing or the string matching *regexp*
regexp1 | *regexp2*	all string matching either *regexp1* or *regexp2*
regexp1 *regexp2*	concatenations of two strings one matching *regexp1* and the other *regexp2*
(*regexp*)	the strings matching *regexp*
ident	the previously defined regular expression *ident*
regexp as *ident*	the result of the matching is bound to the OCaml variable *ident*

Figure 3: Ocamllex regular expression syntax

- `lexeme_start`: returns the position index of the beginning of the recognized chain.
- `lexeme_end`: which returns the position index of end of the recognized chain.

Each regular expression is compiled by the tool in an automaton. All the automata are merged into a single automaton. This automaton is then determinized and minimized. Its code is inserted between the two portions of OCaml code at the beginning and the end of the file `.mll` to form a `.ml` file (which implements the lexical analysis).

2.2 A first example

To better illustrate Ocamllex syntax, let's look at the file `exprLexer_standalone.mll` that implements a lexer for the expression language EXPR of in exercise 6 from the *Discovering OCaml* document.

```
{
 type token =
  | EOF | PLUS | MINUS | TIMES | DIV | MOD | LPAR | RPAR
  | INT of int | IDENT of string
 let mk_int nb =
  try INT (int_of_string nb)
  with Failure _ -> failwith (Printf.sprintf "Illegal integer '%s': " nb)
}
let newline = (['\n' '\r'] | "\r\n")
```

```
10 let blank = [' ' '\014' '\t' '\012']
11 let digit = ['0'-'9']
12 let letter = ['a'-'z' 'A'-'Z']
13 rule token = parse
14   (* newlines *)
15   | newline + { token lexbuf }
16   (* blanks *)
17   | blank + { token lexbuf }
18   (* end of file *)
19   | eof     { EOF }
20   (* integers *)
21   | digit+ as nb        { mk_int nb }
22   (* commands *)
23   | "+"     { PLUS }
24   | "-"     { MINUS }
25   | "/"     { DIV }
26   | "*"     { TIMES }
27   | "%"     { MOD }
28   | "("     { LPAR }
29   | ")"     { RPAR }
30   (* identifiers *)
31   | letter (letter | digit | '_')* as id { IDENT id }
32   (* illegal characters *)
33   | _ as c { failwith (Printf.sprintf "Illegal character '%c': " c) }
```

Lines 1 to 8 contain regular OCaml code that defines the `token` type and a function `mk_int` that builds an integer from a string. The program fails if the string is not a correct integer. Can it happen? Lines 9 to 12 define regular expressions to be reused in the rules. The remaining lines define the only rule of the file, named `token`, that recognizes an EXPR token. As long as there is newlines characters or blank characters, the lexer continues its reading of the input buffer. If we reach the end of the buffer (special character `eof`), we return the token representing it, `EOF`. When we recognize a token, we return it and if an unknown character (one for which no rule matches) is encountered, we fail with an error.

This file[3] can be compiled by `ocamllex`. A file `exprLexer_standalone.ml` is then generated. In this file the type `token` is defined, so do the functions `mk_int` of type `string -> token` and `token` of type `Lexing.lexbuf -> token` implementing the defined automaton.

You can test it using `utop`:

```
# let add i = i + 10 ;;
val add : int -> int = <fun>
```

[3]It can be downloaded from moodle.

```
utop # #use "exprLexer_standalone.ml" ;;
type token =
    EOF
  | PLUS
  | MINUS
  | TIMES
  | DIV
  | MOD
  | LPAR
  | RPAR
  | INT of int
  | IDENT of string
val mk_int : string -> token = <fun>
val token : Lexing.lexbuf -> token = <fun>
utop # let buffer = Lexing.from_string "13 + x1" ;;
val buffer : Lexing.lexbuf =
  {Lexing.refill_buff = <fun>; lex_buffer = Bytes.of_string "13 + x1";
  lex_buffer_len = 7; lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0;
  lex_last_pos = 0; lex_last_action = 0; lex_eof_reached = true;
  lex_mem = [||];
  lex_start_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
  lex_curr_p =
  {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
utop # token buffer ;;
- : token = INT 13
utop # token buffer ;;
- : token = PLUS
utop # token buffer ;;
- : token = IDENT "x1"
utop # token buffer ;;
- : token = EOF
```

2.3 Precisions

An Ocamllex rule can be considered as a function because `ocamllex` generates a function with the same name. This function is recursive as we have seen in the previous example (`token` calls `token` when eliminating spaces and newlines).

You can add parameters to a rule. We will illustrate it with a small example that probably would not require the use of Ocamllex. The objective is to write an automaton to count the occurrences of the character `'a'` in a string. For this, we define a rule which has as parameter (`value`) containing the number of `'a'` already met. When the automaton encounters a `'a'`, it calls itself recursively with a value incremented by 1 (line 3). When the string ends, the automaton returns the number of `'a'` (line 4). The desired function is then defined in the postlude reusing the rule (line 6).

```
1 rule count value = parse
```

```
2  | [^'a']* { count value lexbuf }
3  | 'a'   { count (value + 1) lexbuf }
4  | eof   { value }
5 {
6  let count_a s = let buffer = Lexing.from_string s in count 0 buffer
7 }
```

Once compiled by `ocamllex` and loaded in the interpreter, the function can be tested.

```
utop # #use "count_a.ml" ;;
val count : int -> Lexing.lexbuf -> int = <fun>
val count_a : string -> int = <fun>
utop # count_a "eratatata" ;;
- : int = 4
```

⚠ ocamlbuild

Now and for the rest of the UV, you should use `ocamlbuild` to compile and directly produce an executable file. You need to add the following line to bind the definition of the main function (here `compile`)[4].

```
let _ = Arg.parse [] compile ""
```

3 Parsing with Menhir

The Menhir files use the extension `.mly` and have the following form:

```
%{
  (* OCaml code *)
%}
  (* Declarations of symbols *)
%%
  (* Rules *)
%%
  (* OCaml code *)
```

The main structure is similar to Ocamllex with a prelude and a postlude, some declarations and a set of rules.

3.1 The declarations

The following declarations are possible:

- `%token` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the n symbols as lexical tokens. They are added as constructors to the type `token`. When a type is given the n constructors take it as argument. By consequences the lexer does not need to define the token type anymore, instead it uses the parser's one.

[4] For more details, do not hesitate to consult the manual section on the module `Arg`.

- `%start` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the n symbols as entry points. A parsing function of the same name is defined for each symbol. The symbol must be a non terminal left part of a rule. The return type of this function can be given simultaneously or using the following declaration.
- `%type` (< *type* >)? $symbol_1$ ... $symbol_n$: defines the return types of the actions corresponding to the n symbols. Each symbol must be a non terminal left part of a rule. These type declarations are only mandatories for entry points.
- the priority and associativity of symbols:
  - `%left` $symbol_1$ ... $symbol_n$
  - `%right` $symbol_1$ ... $symbol_n$
  - `%nonassoc` $symbol_1$ ... $symbol_n$

  The name specifies the associativity and the order of appearance in the file specifies the priority. The first defined has the weakest priority. When on the same line they share associativity and priority. Associativity and priority are used when there is a conflict. For example, the expressions $1 + 2 * 3$ can be recognized (depending on the defined rules) as $(1 + 2) * 3$ or $1 + (2 * 3)$. First, the parser use priority. Here for example, if $*$ has a higher priority than $+$, the second form is adopted. In case of similar priority, it uses associativity. The first term corresponds to left and the second to right. If the symbol is declared non associative, a parsing error is raised. Often binary operators are left associative and unary ones are right associative.

3.2 The rules

The rules follow the syntax:

```
nonterminal:
 | symbol ... symbol { (* semantic action *) }
 | ...
 | symbol ... symbol { (* semantic action *) }
```

A semantic action contains OCaml code that builds and returns the semantic value of the non-terminal in the corresponding case. This semantic action can use the semantic value of all terminals and non-terminals that appear in the corresponding production. Two ways to access their value are available: (1) by naming these symbols in production or (2) by position `$1` for the first symbol and up to `$9`. The second possibility is deprecated because it creates a strong coupling between action and production (if a symbol is moved, the action code must be changed). In general, the semantic action consists in constructing the node of the Abstract Syntax Tree associated with the recognized sentence.

To demonstrate how Menhir works, we're going to examine an example in details. Let's look at a parser for Expr.

```
1 %{
2  open ExprAst
3  open BinOp
4 %}
```

```
5 %token EOF PLUS MINUS TIMES DIV MOD LPAR RPAR
6 %token <int> INT
7 %token <string> IDENT
8 %start < ExprAst.expression > expression
9 %left PLUS MINUS
10 %left TIMES DIV MOD
11 %right UMINUS
12 %%
13 expression:
14  | e=expr EOF         { e }
15 expr:
16   | MINUS e=expr %prec UMINUS  { Uminus e }
17   | e1=expr o=bop e2=expr     { Binop(o,e1,e2) }
18   | e=simple_expr            { e }
19 simple_expr:
20   | LPAR e=expr RPAR         { e }
21   | id=IDENT               { Var id }
22   | i=INT                 { Const i }
23 %inline bop:
24   | MINUS    { Bsub }
25   | PLUS    { Badd }
26   | TIMES    { Bmul }
27   | DIV     { Bdiv }
28   | MOD     { Bmod }
29 %%
```

Lines 2 and 3 open modules to make them available inside actions. Lines 5 to 7 define the various tokens. The two last tokens `INT` and `IDENT` can carry information. More precisely, a token of type `INT` will contain the corresponding integer and the token `IDENT` contains the string of the identifier. Line 8 defines the only entry point: the non terminal `expression`, its type corresponds to the AST defined for expressions. Lines 9 to 11 define priorities and associativities. `UMINUS` has a higher priority than `TIMES`, `DIV` and `MOD` that have a higher priority than `PLUS` and `MINUS`. Notice here that the symbol `UMINUS` is not defined using `%token`. Such a symbol will have to be used by a `%prec` in a rule (here on line 16). Lines 13, 15 and 21 define the rules of derivation for three non terminals. If we look in more details line 17, an `expr` can be derived as an `expr` followed by a `bop` and followed by an `expr`. The result of the action is a term `Binop` collecting the content of these three elements using variables.

When compiling this file, Menhir will produce an OCaml module of signature `exprParser.mli` and of implementation `exprParser.ml`. This module defines:

- the type `token`:

  ```
  type token =
    | TIMES
    | RPAR
    | PLUS
    | MOD
    | MINUS
    | LPAR
  ```

```
| INT of (int)
| IDENT of (string)
| EOF
| DIV
```

- an exception for parsing errors `Error`
- a parsing function named `expression` of type `(Lexing.lexbuf -> token) -> Lexing.lexbuf -> exprAst.expression` that takes as argument, the lexer, the input buffer and returns an AST expression.

The lexer corresponds to the code already presented in section 2.2 without the definition of the type `token` which is now defined by the parser.

Remark:

The signature does not include the prelude of the Menhir specification file. So any types that the signature will contain must be fully qualified. (using the dot notation and the name of their module). By consequence, all the types of entry points and the types parameterizing a token must be fully qualified (as in line 8 above).