11

Tools for lexical analysis and parsing

The development of lexical analysis and parsing tools has been an important area of research in computer science. This work has produced the lexer and parser generators `lex` and `yacc` whose worthy scions `camllex` and `camlyacc` are presented in this chapter. These two tools are the de-facto standard for implementing lexers and parsers, but there are other tools, like streams or the *regular expression* library `str`, that may be adequate for applications which do not need a powerful analysis.

The need for such tools is especially acute in the area of state-of-the-art programming languages, but other applications can profit from such tools: for example, database systems offering the possibility of issuing queries, or spreadsheets defining the contents of cells as the result of the evaluation of a formula. More modestly, it is common to use plain text files to store data; for example system configuration files or spreadsheet data. Even in such limited cases, processing the data involves some amount of lexical analysis and parsing.

In all of these examples the problem that lexical analysis and parsing must solve is that of transforming a linear character stream into a data item with a richer structure: a string of words, a record structure, the abstract syntax tree for a program, etc.

All languages have a set of vocabulary items (lexicon) and a grammar describing how such items may be combined to form larger items (syntax). For a computer or program to be able to correctly process a language, it must obey precise lexical and syntactic rules. A computer does not have the detailed semantic understanding required to resolve ambiguities in natural language. To work around the limitation, computer languages typically obey clearly stated rules without exceptions. The lexical and syntactic structure of such languages has received formal definitions that we briefly introduce in this chapter before introducing their uses.

Chapter Structure

This chapter introduces the tools of the Objective Caml distribution for lexical analysis and parsing. The latter normally supposes that the former has already taken place. In the first section, we introduce a simple tool for lexical analysis provided by module `Genlex`. Next we give details about the definition of sets of lexical units by introducing the formalism of *regular expressions*. We illustrate their behavior within module `Str` and the `ocamllex` tool. In section two we define grammars and give details about sentence production rules for a language to introduce two types of parsing: bottom-up and top-down. They are further illustrated by using *`Stream`* and the `ocamlyacc` tool. These examples use context-free grammars. We then show how to carry out contextual analysis with *`Streams`*. In the third section we go back to the example of a BASIC interpreter from page 159, using `ocamllex` and `ocamlyacc` to implement the lexical analysis and parsing functions.

Lexicon

Lexical analysis is the first step in character string processing: it segments character strings into a sequence of words also known as *lexical units* or *lexemes*.

Module `Genlex`

This module provides a simple primitive allowing the analysis of a string of characters using several categories of predefined lexical units. These categories are distinguished by type:

```
# type token =
    Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char ;;
```

Hence, we will be able to recognize within a character string an integer (constructor `Int`) and to recover its value (constructor argument of type *`int`*). Recognizable strings and characters respect the usual conventions: a string is delimited by two (`"`) characters and character literals by two (`'`) characters. A float is represented by using either floating-point notation (for example `0.01`) or exponent-mantissa notation (for example `1E-2`).

Constructor `Ident` designates the category of *identifiers*. These are the names of variables or functions in programming languages, for example. They comprise all strings

of letters and digits including underscore (_) or apostrophe (’). Such a string should not start with a digit. We also consider as identifiers (for this module at least) strings containing operator symbols, such as `+`, `*`, `>` or `=`. Finally, constructor `Kwd` defines the category of keywords containing distinguished identifiers or special characters (specified by the programmer when invoking the lexer).

The only variant of the token type controlled by parameters is that of keywords. The following primitive allows us to create a lexical analyser (lexer) taking as keywords the list passed as first argument to it.

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

The result of applying `make_lexer` to a list of keywords is a function taking as input a stream of characters and returning a stream of lexical units (of type *token*.)

Thus we can easily obtain a lexer for our BASIC interpreter. We declare the set of keywords:

```
# let keywords =
    [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";
      "-"; "!"; "+"; "-"; "*"; "/"; "%";
      "="; "<"; ">"; "<="; ">="; "<>";
      "&"; "|" ] ;;
```

With this definition in place, we define the lexer:

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

Function `line_lexer` takes as input a string of characters and returns the corresponding stream of lexemes.

Use of Streams

We can carry out the lexical analysis “by hand” by directly manipulating streams.

The following example is a lexer for arithmetical expressions. Function `lexer` takes a character stream and returns a stream of lexical units of type *lexeme Stream.t*[1].

1. Type *lexeme* is defined on page 163

Spaces, tabs and newline characters are removed. To simplify, we do not consider variables or negative integers.

```
# let rec spaces s =
    match s with parser
      [<'' ' ; rest >] → spaces rest
    | [<''\t' ; rest >] → spaces rest
    | [<''\n' ; rest >] → spaces rest
    | [<>] → ();;
val spaces : char Stream.t -> unit = <fun>
# let rec lexer s =
    spaces s;
    match s with parser
      [< ''(' >] → [< 'Lsymbol "(" ; lexer s >]
    | [< '')' >] → [< 'Lsymbol ")" ; lexer s >]
    | [< ''+' >] → [< 'Lsymbol "+" ; lexer s >]
    | [< ''-' >] → [< 'Lsymbol "-" ; lexer s >]
    | [< ''*' >] → [< 'Lsymbol "*" ; lexer s >]
    | [< ''/' >] → [< 'Lsymbol "/" ; lexer s >]
    | [< ''0'..'9' as c;
         i,v = lexint (Char.code c - Char.code('0')) >]
       → [<'Lint i ; lexer v>]
  and lexint r s =
    match s with parser
      [< ''0'..'9' as c >]
      → let u = (Char.code c) - (Char.code '0') in  lexint (10*r + u) s
    | [<>] → r,s
  ;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t = <fun>
```

Function `lexint` carries out the lexical analysis for the portion of a stream describing an integer constant. It is called by function `lexer` when `lexer` finds a digit on the input stream. Function `lexint` then consumes all consecutive digits to obtain the corresponding integer value.

Regular Expressions

[2]

Let's abstract a bit and consider the problem of lexical units from a more theoretical point of view.

2. Note of translators: From an academic standpoint, the proper term would have been *"Rational Expressions"*; we chose the term "regular" to follow the programmers' tradition.

From this point of view, a lexical unit is a *word*. A word is formed by concatening items in an *alphabet*. For our purposes, the alphabet we are considering is a subset of the ASCII characters. Theoretically, a word may contain no characters (the *empty word*[3]) or just a single character. The theoretical study of the assembly of lexical items (lexemes) from members of an alphabet has brought about a simple formalism known as *regular expressions*.

Definition A regular expression defines a set of words. For example, a regular expression could specify the set of words that are valid identifiers. Regular expressions are specified by a few set-theoretic operations. Let M and N be two sets of words. Then we can specify:

1. the union of M and N, denoted by M `|` N.
2. the complement of M, denoted by `^`M. This is the set of all words not in M.
3. the concatenation of M and N. This is the set of all the words formed by placing a word from M before a word from N. We denote this set simply by MN.
4. the set of words formed by a finite sequence of words in M, denoted M`+`.
5. for syntactic convenience, we write M`?` to denote the set of words in M, with addition of the empty word.

Individual characters denote the singleton set of words containing just that character. Expression `a | b | c` thus describes the set containing three words: `a`, `b` ant `c`. We will use the more compact syntax `[abc]` to define such a set. As our alphabet is ordered (by the ASCII code order) we can also define intervals. For example, the set of digits can be written: `[0-9]`. We can use parentheses to group expressions.

If we want to use one of the operator characters as a character in a regular expression, it should be preceded by the escape character `\`. For example, `(\*)*` denotes the set of sequences of stars.

Example Let's consider the alphabet comprising digits (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`) the plus (`+`), minus (`-`) and dot (`.`) signs and letter `E`. We can define the set *num* of words denoting numbers. Let's call *integers* the set defined with `[0-9]+`. We define the set *unum* of unsigned numbers as:

$$integers\texttt{?(.}integers\texttt{)?(E(\+|-)?}integers\texttt{)?}$$

The set of signed numbers is thus defined as:

$$unum \texttt{ | -}unum \text{ or with } -?unum$$

Recognition While regular expressions are a useful formalism in their own right, we usually wish to implement a program that determines whether a string of characters (or

3. By convention, the empty word is denoted by the greek character epsilon: ϵ

one of its substrings) is a member of the set of words described by a regular expression. For that we need to translate the formal definition of the set into a recognition and expression processing program. In the case of regular expressions such a translation can be automated. Such translation techniques are carried out by module `Genlex` in library `Str` (described in the next section) and by the `ocamllex` tools that we introduce in the following two sections.

The `Str` *Library*

This module contains an abstract data type *`regexp`* which represents regular expressions as well as a function `regexp` which takes a string describing a regular expression, more or less following the syntax described above, and returns its abstract representation.

This module contains, as well, a number of functions which exploit regular expressions and manipulate strings. The syntax of regular expressions for library `Str` is given in figure 11.1.

`.`	any character except `\n`
`*`	zero or more occurences of the preceding expression
`+`	one or more occurences of the preceding expression
`?`	zero or one occurences of the preceding expression
`[..]`	set of characters (example `[abc]`)
	intervals, denoted by `-` (example `[0-9]`)
	set complements, denoted by `^` (example `[^A-Z]`)
`^`	start of line (not to be mistaken with the use of `^` as a set complement)
`$`	end of line
`|`	alternative
`(..)`	grouping of a complex expression (we can later refer to such an expression by an integer index – see below)
i	an integer constant, referring to the string matched by the i-th complex expression
`\`	escape character (used when matching a reserved character in regular expressions)

Figure 11.1: Regular expressions.

Example We want to write a function translating dates in anglo-saxon format into French dates within a data file. We suppose that the file is organised into lines of data fields and the components of an anglo-saxon date are separated by dots. Let's define a function which takes as argument a string (*i.e.* a line from the file), isolates the date, decomposes and translates it, then replaces the original with the translation.

```
# let french_date_of d =
    match d with
     [mm; dd; yy] → dd^"/"^mm^"/"^yy
    | _ → failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
    try
     let i=Str.search_forward english_date_format l 0 in
     let d1 = Str.matched_string l in
     let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
       Str.global_replace english_date_format d2 l
    with Not_found → l ;;
val trans_date : string -> string = <fun>

# trans_date ".............06.13.99............" ;;
- : string = ".............13/06/99............"
```

The `ocamllex` *Tool*

The `ocamllex` tool is a lexical analyzer generator built for Objective Caml after the model of the `lex` tool for the C language. It generates a source Objective Caml file from a file describing the lexical elements to be recognized in the form of regular expressions. The programmer can augment each lexical element description with a processing action known as a *semantic action*. The generated code manipulates an abstract type ***`lexbuf`*** defined in module `Lexing`. The programmer can use this module to control processing of lexical buffers.

Usually the lexical description files are given the extension `.mll`. Later, to obtain a Objective Caml source from a `lex_file.mll` you type the command

```
ocamllex lex_file.mll
```

A file `lex_file.ml` is generated containing the code for the corresponding analyzer. This file can then be compiled with other modules of an Objective Caml application. For each set of lexical analysis rules there is a corresponding function taking as input a lexical *buffer* (of type ***`Lexing.lexbuf`***) and returning the value defined by the semantic actions. Consequently, all actions in the same rule must produce values of the same type.

The general format for an `ocamllex` file is

```
{
```

```
  header
}

let ident = regexp
  ...
rule ruleset1 = parse
       regexp { action }
     | ...
     | regexp { action }
and ruleset2 = parse
     ...
and ...
{

  trailer-and-end
}
```

Both section "header" and "trailer-and-end" are optional. They contain Objective Caml code defining types, functions, etc. needed for processing. The code in the last section can use the lexical analysis functions that will be generated by the middle section. The declaration list preceding the rule definition allows the user to give names to some regular expressions. They can later be invoked by name in the definition of rules.

Example Let's revisit our BASIC example. We will want to refine the type of lexical units returned. We will once again define function `lexer` (as we did on page 163) with the same type of output (*`lexeme`*), but taking as input a *buffer* of type *`Lexing.lexbuf`*.

```
{
 let string_chars s =
  String.sub s 1 ((String.length s)-2) ;;
}

let op_ar = ['-' '+' '*' '%' '/']
let op_bool = ['!' '&' '|']
let rel = ['=' '<' '>']

rule lexer = parse
   [' ']   { lexer lexbuf }

 | op_ar   { Lsymbol (Lexing.lexeme lexbuf) }
 | op_bool { Lsymbol (Lexing.lexeme lexbuf) }

 | "<="    { Lsymbol (Lexing.lexeme lexbuf) }
 | ">="    { Lsymbol (Lexing.lexeme lexbuf) }
 | "<>"    { Lsymbol (Lexing.lexeme lexbuf) }
 | rel     { Lsymbol (Lexing.lexeme lexbuf) }
```

```
| "REM"   { Lsymbol (Lexing.lexeme lexbuf) }
| "LET"   { Lsymbol (Lexing.lexeme lexbuf) }
| "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
| "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
| "IF"    { Lsymbol (Lexing.lexeme lexbuf) }
| "THEN"  { Lsymbol (Lexing.lexeme lexbuf) }

| '-'? ['0'-'9']+   { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+        { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'  { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

The translation of this file by `ocamllex` returns function `lexer` of type *`Lexing.lexbuf -> lexeme`*. We will see later how to use such a function in conjunction with syntactic analysis (see page 305).

Syntax

Thanks to lexical analysis, we can split up input streams into more structured units: lexical units. We still need to know how to assemble these units so that they amount to syntactically correct sentences in a given language. The syntactic assembly rules are defined by *grammar rules*. This formalism was originally developed in the field of linguistics, and has proven immensely useful to language-theoretical mathematicians and computer scientists in that field. We have already seen on page 160 an instance of a grammar for the Basic language. We will resume this example to introduce the basic concepts for grammars.

Grammar

Formally, a grammar is made up of four elements:

1. a set of symbols called *terminals*. Such symbols represent the lexical units of the language. In Basic, the lexical units (terminals) are: the operator- and arithmetical and logical relation-symbols (`+`, `&`, `<`, `<=`, ..), the keywords of the language (`GOTO`, `PRINT`, `IF`, `THEN`, ..), integers (*integer* units) and variables (*variable* units).
2. A set of symbols called *non-terminals*. Such symbols stand for syntactic terms of the language. For example, a Basic program is composed of lines (and thus we have the term Line), a line may contain and Expression, etc.
3. A set of so-called *production* rules. These describe how terminal and non-terminals symbols may be combined to produce a syntactic term. A Basic line is made up of a number followed by an instruction. This is expressed in the following rule:

   Line ::= *integer* Instruction

   For any given term, there may be several alternative ways to form that term. We separate the alternatives with the symbol — as in

INSTRUCTION	::=	`LET` *variable* `=` EXPRESSION
	—	`GOTO` *integer*
	—	`PRINT` EXPRESSION

etc.

4. Finally, we designate a particular non-terminal as the *start symbol*. The start symbol identifies a complete translation unit (program) in our language, and the corresponding production rule is used as the starting point for parsing.

Production and Recognition

Production rules allow *recognition* that a sequence of lexemes belongs to a particular language.

Let's consider, for instance, a simple language for arithmetic expressions:

EXP	::=	*integer*	(R1)
	—	EXP `+` EXP	(R2)
	—	EXP `*` EXP	(R3)
	—	`(` EXP `)`	(R4)

where (R1) (R2) (R3) and (R4) are the names given to our rules. After lexical analysis, the expression `1*(2+3)` becomes the sequence of lexemes:

integer `*` `(` *integer* `+` *integer* `)`

To analyze this sentence and recognize that it really belongs to the language of arithmetic expressions, we are going to use the rules from right to left: if a subexpression matches the right-side member of a rule, we replace it with the corresponding left-side member and we re-run the process until reducing the expression to the non-terminal *start* (here EXP). Here are the stages of such an analysis[4]:

integer `*` `(` *integer* `+` *integer* `)`	$\xleftarrow{(R1)}$	EXP `*` `(` *integer* `+` *integer* `)`
	$\xleftarrow{(R1)}$	EXP `*` `(` EXP `+` *integer* `)`
	$\xleftarrow{(R1)}$	EXP `*` `(` EXP `+` EXP `)`
	$\xleftarrow{(R2)}$	EXP `*` `(` EXP `)`
	$\xleftarrow{(R4)}$	EXP `*` EXP
	$\xleftarrow{(R3)}$	EXP

Starting from the last line containing only EXP and following the arrows upwards we read how our expression could be produced from the start rule EXP: therefore it is a well-formed sentence of the language defined by the grammar.

4. We underline the portion of input processed at each stage and we point out the rule used.

The translation of grammars into programs capable of recognizing that a sequence of lexemes belongs to the language defined by a grammar is a much more complex problem than that of using regular expressions. Indeed, a mathematical result tells us that all sets (of words) defined by means of a regular expression formalism can also be defined by another formalism: *deterministic finite automata*. And these latter are easy to explain as programs taking as input a sequence of characters. We do not have a similar result for the case of generic grammars. However, we have weaker results establishing the equivalence between certain classes of grammars and somewhat richer automata: *pushdown automata*. We do not want to enter into the details of such results, nor give an exact definition of what an automaton is. Still, we need to identify a class of grammars that may be used with parser-generating tools or parsed directly.

Top-down Parsing

The analysis of the expresion `1*(2+3)` introduced in the previous paragraph is not unique: it could also have started by reducing *integers* from right to left, which would have permitted rule (R2) to reduce `2+3` from the beginning instead. These two ways to proceed constitute two types of analysis: top-down parsing (right-to-left) and bottom-up parsing (left-to-right). The latter is easily realizable with lexeme streams using module `Stream`. Bottom-up parsing is that carried-out by the `ocamlyacc` tool. It uses an explicit stack mechanism like the one already described for the parsing of Basic programs. The choice of parsing type is significant, as top-down analysis may or may not be possible given the form of the grammar used to specify the language.

A Simple Case

The canonical example for top-down parsing is the prefix notation of arithmetic expressions defined by:

$$\begin{array}{lcl} \textsc{Expr} & ::= & \textit{integer} \\ & | & \texttt{+}\ \textsc{Expr}\ \textsc{Expr} \\ & | & \texttt{*}\ \textsc{Expr}\ \textsc{Expr} \end{array}$$

In this case, knowing the first lexeme is enough to decide which production rule can be used. This immediate predictability obviates managing the parse stack explicitly by instead using the stack of recursive calls in the parser. Therefore, it is very easy to write a program implementing top-down analysis using the features in modules `Genlex` and `Stream`. Function `infix_of` is an example; it takes a prefix expression and returns its equivalent infix expression.

```
# let lexer s =
    let ll = Genlex.make_lexer ["+";"*"]
    in  ll (Stream.of_string s) ;;
val lexer : string -> Genlex.token Stream.t = <fun>
# let rec stream_parse s =
```

```
    match s with parser
      [<'Genlex.Ident x>] → x
    | [<'Genlex.Int n>] → string_of_int n
    | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] → "("^e1^"+"^e2^")"
    | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] → "("^e1^"*"^e2^")"
    | [<>] → failwith "Parse error"
  ;;
val stream_parse : Genlex.token Stream.t -> string = <fun>
# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>
# infix_of "* +3 11 22";;
- : string = "((3+11)*22)"
```

One has to be careful, because this parser is rather unforgiving. It is advisable to introduce a blank between lexical units in the input string systematically.

```
# infix_of "*+3 11 22";;
- : string = "*+"
```

A Less Simple Case

Parsing using streams is predictive. It imposes two conditions on grammars.

1. There must be no *left-recursive* rules in the grammar. A rule is left-recursive when a right-hand expression starts with a non-terminal which is the left-hand member of the expression, as in Exp ::= Exp + Exp;
2. No two rules may start with the same expression.

The usual grammar for arithmetical expressions on page 296 is not directly suitable for top-down analysis: it does not satisfy any of the above-stated criteria. To be able to use top-down parsing, we must reformulate the grammar so as to suppress left-recursion and non-determinism in the rules. For arithmetic expressions, we may use, for instance:

Expr	::=	Atom NextExpr
NextExpr	::=	+ Atom
	—	- Atom
	—	* Atom
	—	/ Atom
	—	ϵ
Atom	::=	*integer*
	—	(Expr)

Note that the use of the empty word ϵ in the definition of NextExpr is compulsory if we want a single integer to be an expression.

Our grammar allows the implementation of the following parser which is a simple translation of the production rules. This parser produces the abstract syntax tree of arithmetic expressions.

```
# let rec rest = parser
      [< 'Lsymbol "+"; e2 = atom >]  → Some (PLUS,e2)
    | [< 'Lsymbol "-"; e2 = atom >]  → Some (MINUS,e2)
    | [< 'Lsymbol "*"; e2 = atom >]  → Some (MULT,e2)
    | [< 'Lsymbol "/"; e2 = atom >]  → Some (DIV,e2)
    | [< >] → None
  and atom = parser
      [< 'Lint i >] → ExpInt i
    | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] → e
  and expr s =
    match s with parser
      [< e1 = atom >] →
        match rest s with
            None → e1
          | Some (op,e2) → ExpBin(e1,op,e2) ;;
val rest : lexeme Stream.t -> (bin_op * expression) option = <fun>
val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>
```

The problem with using top-down parsing is that it forces us to use a grammar which is very restricted in its form. Moreover, when the object language is naturally described with a left-recursive grammar (as in the case of infix expressions) it is not always trivial to find an equivalent grammar (*i.e.* one defining the same language) that satisfies the requirements of top-down parsing. This is the reason why tools such as `yacc` and `ocamlyacc` use a bottom-up parsing mechanism which allows the definition of more natural-looking grammars. We will see, however, that not everything is possible with them, either.

Bottom-up Parsing

On page 165, we introduced intuitively the actions of bottom-up parsing: *shift* and *reduce*. With each of these actions the state of the stack is modified. We can deduce from this sequence of actions the grammar rules, provided the grammar allows it, as in the case of top-down parsing. Here, also, the difficulty lies in the non-determinism of the rules which prevents choosing between shifting and reducing. We are going to illustrate the inner workings of bottom-up parsing and its failures by considering those pervasive arithmetic expressions in postfix and prefix notation.

The Good News The simplified grammar for postfix arithmetic expressions is:

Expr	::=	*integer*	(R1)
	|	Expr Expr `+`	(R2)
	|	Expr Expr `*`	(R3)

This grammar is dual to that of prefix expressions: it is necessary to wait until the end of each analysis to know which rule has been used, but then one knows exactly what to do. In fact, the bottom-up analysis of such expressions resembles quite closely a stack-based evaluation mechanism. Instead of pushing the results of each calculation, we simply push the grammar symbols. The idea is to start with an empty stack, then obtain a stack which contains only the start symbol once the input is used up. The modifications to the stack are the following: when we shift, we push the present non-terminal; if we may reduce, it is because the first elements in the stack match the right-hand member of a rule (in reverse order), in which case we replace these elements by the corresponding left-hand non-terminal.

Figure 11.2 illustrates how bottom-up parsing processes expression: `1 2 + 3 * 4 +`. The input lexical unit is underlined. The end of input is noted with a `$` sign.

Action	Input	Stack
	1`2+3*4+$`	[]
Shift		
	2`+3*4+$`	[1]
Reduce (R1)		
	2`+3*4+$`	[Expr]
Shift		
	+`3*4+$`	[2Expr]
Reduce (R1)		
	+`3*4+$`	[Expr Expr]
Shift, Reduce (R2)		
	3`*4+$`	[Expr]
Shift, Reduce (R1)		
	*`4+$`	[Expr Expr]
Shift, Reduce (R3)		
	4`+$`	[Expr]
Shift, Reduce (R1)		
	+`$`	[Expr Expr]
Shift, Reduce (R2)		
	`$`	[Expr]

Figure 11.2: Bottom-up parsing example.

The Bad News The difficulty of migrating the grammar into the recognition program is determining which type of action to apply. We will illustrate this difficulty with three examples which generate three types of indeterminacy.

The first example is a grammar for expressions using only addition:

$$\begin{array}{lcll} \text{E0} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E0 + E0} & \text{(R2)} \end{array}$$

The indeterminacy in this grammar stems from rule (R2). Let's suppose the following situation:

Action	Input	Stack
⋮		
	$\underline{+}\ldots$	[E0 + E0 ...]
⋮		

In such a case, it is impossible to determine whether we have to shift and push the + or to reduce using (R2) both E0's and the + in the stack. We are in the presence of a shift-reduce conflict. This is because expression *integer* + *integer* + *integer* can be produced in two ways by right-derivation.

First way:
$$\begin{array}{lcl} \text{E0} & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R1)} & \underline{\text{E0}} \text{ + } \textit{integer} \\ & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \text{ + } \textit{integer} \end{array}$$
etc.

Second way:
$$\begin{array}{lcl} \text{E0} & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R2)} & \text{E0 + E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R1)} & \text{E0 + } \underline{\text{E0}} \text{ + } \textit{integer} \end{array}$$
etc.

The expressions obtained by each derivation may look similar from the point of view of expression evaluation:

(*integer* + *integer*) + *integer* and *integer* + (*integer* + *integer*)

but different for building a syntax tree (see figure 6.3 on page 166).

The second instance of a grammar generating a conflict between shifting and reducing has the same type of ambiguity: an implicit parenthesizing. But contrary to the previous case, the choice between shifting and reducing modifies the meaning of the parsed expression. Let's consider the following grammar:

$$\begin{array}{lcll} \text{E1} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E1 + E1} & \text{(R2)} \\ & | & \text{E1 * E1} & \text{(R3)} \end{array}$$

We find in this grammar the above-mentioned conflict both for + and for *. But there is an added conflict between + and *. Here again, an expression may be produced in two ways. There are two right-hand derivations of

$$\textit{integer} + \textit{integer} * \textit{integer}$$

$$\begin{array}{lll} \text{First way:} \quad \text{E1} & \xrightarrow{(R3)} & \text{E1} * \underline{\text{E1}} \\ & \xrightarrow{(R1)} & \underline{\text{E1}} * \textit{integer} \\ & \xrightarrow{(R2)} & \text{E1} + \underline{\text{E1}} * \textit{integer} \\ & \text{etc.} & \end{array}$$

$$\begin{array}{lll} \text{Second way:} \quad \text{E1} & \xrightarrow{(R2)} & \text{E1} + \underline{\text{E1}} \\ & \xrightarrow{(R3)} & \text{E1} + \text{E1} * \underline{\text{E1}} \\ & \xrightarrow{(R1)} & \text{E1} + \underline{\text{E1}} * \textit{integer} \\ & \text{etc.} & \end{array}$$

Here both pairs of parenthesis (implicit) are not equivalent:

$$(\textit{integer} + \textit{integer}) * \textit{integer} \neq \textit{integer} + (\textit{integer} * \textit{integer})$$

This problem has already been cited for Basic expressions (see page 165). It was solved by attributing different precedence to each operator: we reduce (R3) before (R2), which is equivalent to parenthesizing products.

We can also solve the problem of choosing between + and * by modifying the grammar. We introduce two new terminals: T (for terms), and F (for factors), which gives:

$$\begin{array}{lcll} \text{E} & ::= & \text{E + T} & \text{(R1)} \\ & | & \text{T} & \text{(R2)} \\ \text{T} & ::= & \text{T * F} & \text{(R3)} \\ & | & \text{F} & \text{(R4)} \\ \text{F} & ::= & \textit{integer} & \text{(R5)} \end{array}$$

There is now but a single way to reach the production sequence *integer* + *integer* * *integer*: using rule (R1).

The third example concerns conditional instructions in programming languages. A language such as Pascal offers two conditionals : `if .. then` and `if .. then .. else`. Let's imagine the following grammar:

$$\begin{array}{lcll} \textsc{Instr} & ::= & \texttt{if } \textsc{Exp} \texttt{ then } \textsc{Instr} & \text{(R1)} \\ & — & \texttt{if } \textsc{Exp} \texttt{ then } \textsc{Instr} \texttt{ else } \textsc{Instr} & \text{(R2)} \\ & — & \text{etc...} & \end{array}$$

In the following situation:

ACTION	INPUT	STACK
⋮		
	`else`...	[INSTR `then` EXP `if`...]
⋮		

We cannot decide whether the first elements in the stack relate to conditional (R1), in which case it must be reduced, or to the first INSTR in rule (R2), in which case it must be shifted.

Besides shift-reduce conflicts, bottom-up parsing may also generate reduce-reduce conflicts.

We now introduce the `ocamlyacc` tool which uses the bottom-up parsing technique and may find these conflicts.

The `ocamlyacc` *Tool*

The `ocamlyacc` tools is built with the same principles as `ocamllex`: it takes as input a file describing a grammar whose rules have semantic actions attached, and returns two Objective Caml files with extensions `.ml` ant `.mli` containing a parsing function and its interface.

General format The syntax description files for `ocamlyacc` use extension `.mly` by convention and they have the following structure:

```
%{
    header
}%
    declarations
%%
    rules
%%
    trailer-and-end
```

The rule format is:

```
non-terminal   :   symbol...symbol   { semantic action }
               |   ...
               |   symbol...symbol   { semantic action }
               ;
```

A symbol is either a terminal or a non-terminal. Sections "header" and "trailer-and-end" play the same role as in `ocamllex` with the only exception that the header is only

visible by the rules and not by declarations. In particular, this implies that module openings (**open**) are not taken into consideration in the declaration part and the types must therefore be fully qualified.

Semantic actions Semantic actions are pieces of Objective Caml code executed when the parser reduces the rule they are associated with. The body of a semantic action may reference the components of the right-hand term of the rule. These are numbered from left to right starting with 1. The first component is referenced by `$1`, the second by `$2`, etc.

Start Symbols We may declare several start symbols in the grammar, by writing in the declaration section:

```
%start non-terminal .. non-terminal
```

For each of them a parsing function will be generated. We must precisely note, always in the declaration section, the output type of these functions.

```
%type <output-type> non-terminal
```

The `output-type` must be qualified.

Warning Non-terminal symbols become the name of parsing functions. Therefore, they must not start with a capital letter which is reserved for constructors.

Lexical units Grammar rules make reference to lexical units, the terminals or terminal symbols in the rules.

One (or several) lexemes are declared in the following fashion:

```
%token PLUS MINUS MULT DIV MOD
```

Certain lexical units, like identifiers, represent a set of (character) strings. When we find an identifier we may be interested in recovering its character string. We specify in the parser that these lexemes have an associated value by enclosing the type of this value between `<` and `>`:

```
%token <string> IDENT
```

Warning After being processed by `ocamlyacc` all these declarations are transformed into constructors of type *`token`*. Therefore, they must start with a capital letter.

We may use character strings as implicit terminals as in:

```
expr : expr "+" expr   { ... }
     | expr "*" expr   { ... }
     | ...
     ;
```

in which case it is pointless to declare a symbol which represents them: they are directly processed by the parser without passing through the lexer. In the interest of uniformity, we do not advise this procedure.

Precedence, associativity We have seen that many bottom-up parsing conflicts arise from implicit operator association rules or precedence conflicts between operators. To handle these conflicts, we may declare default associativity rules (left-to-right or non-associative) for operators as well as precedence rules. The following declaration states that operators `+` (lexeme `PLUS`) and `*` (lexeme `MULT`) associate to the right by default and `*` has a higher precedence than `+` because `MULT` is declared after `PLUS`.

```
%left PLUS
%left MULT
```

Two operators declared on the same line have the same precedence.

Command options `ocamlyacc` has two options:

- `-b name`: the generated Objective Caml files are `name.ml` and `name.mli`;
- `-v`: create a file with extension `.output` contaning rule numeration, the states in the automaton recognizing the grammar and the sources of conflicts.

Joint usage with `ocamllex` We may compose both tools `ocamllex` and `ocamlyacc` so that the transformation of a character stream into a lexeme stream is the input to the parser. To do this, type *`lexeme`* should be known to both. This type is defined in the files with extensions `.mli` and `.ml` generated by `ocamlyacc` from the declaration of the **`token`**s in the matching file with extension `.mly`. The `.mll` file imports this type; `ocamllex` translates this file into an Objective Caml function of type *`Lexing.lexbuf -> lexeme`*. The example on page 307 illustrates this interaction and describes the different phases of compilation.

Contextual Grammars

Types generated by `ocamlyacc` process languages produced by so-called *context-free* grammars. A parser for such a grammar does not depend on previously processed syntactic values to process the next lexeme. This is not the case of the language L described by the following formula:

$$L ::= wCw \mid w \text{ with } w \in (A|B)^*$$

where A, B and C are terminal symbols. We have written wCw (with $w \in (A|B)^*$) and not simply $(A|B)^*C(A|B)^*$ because we want the *same* word to the left and right of the middle C.

To parse the words in L, we must remember what has already been found before letter C to verify that we find exactly the same thing afterwards. Here is a solution for this problem based on "visiting" a stream. The general idea of the algorithm is to build a stream parsing function which will recognize exactly the subword before the possible occurrence of C.

We use the type:

```
# type token = A | B | C ;;
```

Function `parse_w1` builds the memorizing function for the first w under the guise of a list of atomic stream parsers (*i.e.* for a single *token*):

```
# let rec parse_w1 s =
    match s with parser
      [<'A; l = parse_w1  >] → (parser [<'A >] → "a") :: l
    | [<'B; l = parse_w1  >] → (parser [<'B >] → "b") :: l
    | [< >] → [] ;;
val parse_w1 : token Stream.t -> (token Stream.t -> string) list = <fun>
```

The result of the function returned by `parse_w1` is simply the character string containing the parsed lexical unit.

Function `parse_w2` takes as argument a list built by `parse_w1` to compose each of its elements into a single parsing function:

```
# let rec parse_w2 l =
    match l with
      p :: pl → (parser [< x = p; l = (parse_w2 pl)  >] → x^l)
    | [] → parser [<>] → "" ;;
val parse_w2 : ('a Stream.t -> string) list -> 'a Stream.t -> string = <fun>
```

The result of applying `parse_w2` will be the string representing subword w. By construction, function `parse_w2` will not be able to recognize anything but the subword visited by `parse_w1`.

Using the ability to name intermediate results in streams, we write the recognition function for the words in the language L:

```
# let parse_L = parser  [< l = parse_w1 ; 'C; r = (parse_w2 l) >] → r ;;
val parse_L : token Stream.t -> string = <fun>
```

Here are two small examples. The first results in the string surrounding C, the second fails because the words surrounding C are different:

```
# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"
# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Uncaught exception: Stream.Error("")
```

Basic Revisited

We now want to use `ocamllex` and `ocamlyacc` to replace function `parse` on page 169 for Basic by some functions generated from files specifying the lexicon and syntax of the language.

To do this, we may not re-use as-is the type of lexical units that we have defined. We will be forced to define a more precise type which permits us to distinguish between operators, commands and keywords.

We will also need to isolate the type declarations describing abstract syntax within a file `basic_types.mli`. This will contain the declaration of type `sentences` and all types needed by it.

File `basic_parser.mly`

Header The file header imports the types needed for the abstract syntax as well as two auxiliary functions to convert from character strings to their equivalent in the abstract syntax.

```
%{
open Basic_types ;;

let phrase_of_cmd c =
 match c with
   "RUN" → Run
 | "LIST" → List
 | "END" → End
 | _ → failwith "line : unexpected command"
;;

let bin_op_of_rel r =
 match r with
   "=" → EQUAL
 | "<" → INF
 | "<=" → INFEQ
 | ">" → SUP
 | ">=" → SUPEQ
 | "<>" → DIFF
```

```
 | _ → failwith "line : unexpected relation symbol"
;;

%}
```

Declarations contains three sections: lexeme declarations, their rule associativity and precedence declarations, and the declaration of the start symbol `line` which stands for the parsing of a command or program line.

Lexical units are the following:

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

Their names are self-explanatory and they are described in file `basic_lexer.mll` (see page 310).

Precedence rules between operators once again take the values assigned by functions `priority_uop` and `priority_binop` defined when first giving the grammar for our Basic (see page 160).

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

Symbol `Lop` will be used to process unary minus. It is not a terminal in the grammar, but a "pseudo non-terminal" which allows overloading of operators when two uses of an operator should not receive the same precedence depending on context. This is the case with the minus symbol (`-`). We will reconsider this point once we have specified the rules in the grammar.

Since the start symbol is `line`, the function generated will return the syntax tree for the parsed line.

```
%start line
%type <Basic_types.phrase> line
```

Grammar rules are decomposed into three non-terminals: `line` for a line; `inst` for an instruction in the language; `exp` for expressions. The action associated with each rule simply builds the corresponding abstract syntax tree.

```
%%
line :
   Lint inst Leol              { Line {num=$1; inst=$2} }
 | Lcmd Leol                   { phrase_of_cmd $1 }
 ;

inst :
   Lrem                        { Rem $1 }
 | Lgoto Lint                  { Goto $2 }
 | Lprint exp                  { Print $2 }
 | Linput Lident               { Input $2 }
 | Lif exp Lthen Lint          { If ($2, $4) }
 | Llet Lident Lequal exp       { Let ($2, $4) }
 ;

exp :
   Lint                        { ExpInt $1 }
 | Lident                      { ExpVar $1 }
 | Lstring                     { ExpStr $1 }
 | Lneg exp                    { ExpUnr (NOT, $2) }
 | exp Lplus exp               { ExpBin ($1, PLUS, $3) }
 | exp Lminus exp              { ExpBin ($1, MINUS, $3) }
 | exp Lmult exp               { ExpBin ($1, MULT, $3) }
 | exp Ldiv exp                { ExpBin ($1, DIV, $3) }
 | exp Lmod exp                { ExpBin ($1, MOD, $3) }
 | exp Lequal exp               { ExpBin ($1, EQUAL, $3) }
 | exp Lrel exp                { ExpBin ($1, (bin_op_of_rel $2), $3) }
 | exp Land exp                { ExpBin ($1, AND, $3) }
 | exp Lor exp                 { ExpBin ($1, OR, $3) }
 | Lminus exp %prec Lop       { ExpUnr(OPPOSITE, $2) }
 | Lpar exp Rpar               { $2 }
 ;
%%
```

These rules do not call for particular remarks except:

```
exp :
 ...
 | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
```

It concerns the use of unary `-`. Keyword `%prec` that we find in it declares that this rule should receive the precedence of `Lop` (here the highest precedence).

File `basic_lexer.mll`

Lexical analysis only contains one rule, `lexer`, which corresponds closely to the old function `lexer` (see page 165). The semantic action associated with the recognition of each lexical unit is simply the emission of the related constructor. As the type of lexical units is declared in the syntax rule file, we have to include the file here. We add a simple auxiliary function that strips double quotation marks from character strings.

```
{
 open Basic_parser ;;

 let string_chars s = String.sub s 1 ((String.length s)-2) ;;
}

rule lexer = parse
   [' ' '\t']            { lexer lexbuf }

 | '\n'                  { Leol }

 | '!'                   { Lneg }
 | '&'                   { Land }
 | '|'                   { Lor }
 | '='                   { Lequal }
 | '%'                   { Lmod }
 | '+'                   { Lplus }
 | '-'                   { Lminus }
 | '*'                   { Lmult }
 | '/'                   { Ldiv }

 | ['<' '>']             { Lrel (Lexing.lexeme lexbuf) }
 | "<="                  { Lrel (Lexing.lexeme lexbuf) }
 | ">="                  { Lrel (Lexing.lexeme lexbuf) }

 | "REM" [^ '\n']*       { Lrem (Lexing.lexeme lexbuf) }
 | "LET"                 { Llet }
 | "PRINT"               { Lprint }
 | "INPUT"               { Linput }
 | "IF"                  { Lif }
 | "THEN"                { Lthen }
 | "GOTO"                { Lgoto }

 | "RUN"                { Lcmd (Lexing.lexeme lexbuf) }
 | "LIST"               { Lcmd (Lexing.lexeme lexbuf) }
 | "END"                { Lcmd (Lexing.lexeme lexbuf) }
```

```
| ['0'-'9']+          { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+          { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'    { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

Note that we isolated symbol `=` which is used in both expressions and assignments.

Only two of these regular expressions need further remarks. The first concerns comment lines (`"REM" [^ '\n']*`). This rule recognizes keyword `REM` followed by an arbitrary number of characters other than `'\n'`. The second remark concerns character strings (`'"' [^ '"']* '"'`) considered as sequences of characters different from `"` and contained between two `"`.

Compiling, Linking

The compilation of the lexer and parser must be carried out in a definite order. This is due to the mutual dependency between the declaration of lexemes. To compile our example, we must enter the following sequence of commands:

```
ocamlc -c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc -c basic_parser.mli
ocamlc -c basic_lexer.ml
ocamlc -c basic_parser.ml
```

Which will generate files `basic_lexer.cmo` and `basic_parser.cmo` which may be linked into an application.

We now have at our disposal all the material needed to reimplement the application.

We suppress all types and all functions in paragraphs "lexical analysis" (on page 163) and "parsing" (on page 165) of our Basic application; in function `one_command` (on page 174), we replace expression

```
    match parse (input_line stdin) with
```

with

```
    match line lexer (Lexing.from_string ((input_line stdin)^"\n")) with
```

We need to remark that we must put back at the end of the line the character `'\n'` which function `input_line` had filtered out. This is necessary because the `'\n'` character indicates the end of a command line (`Leol`).

Exercises

Filtering Comments Out

Comments in Objective Caml are hierarchical. We can thus comment away sections of text, including those containing comments. A comment starts with characters `(*` and finishes with `*)`. Here's an example:

```
(* comment spread
      over several
        lines *)

 let succ x = (* successor function *)
   x + 1;;

(* level 1 commented text
  let old_succ y = (* level 2 successor function level 2 *)
    y +1;;
    level 1 *)
  succ 2;;
```

The aim of this exercise is to create a new text without comments. You are free to choose whatever lexical analysis tool you wish.

1. Write a lexer able to recognize Objective Caml comments. These start with a `(*` and end with a `*)`. Your lexer should ensure comments are balanced, that is to say the number of comment openings equals the number of comment closings. We are not interested in other constructions in the language which may contain characters `(*` and `*)`.
2. Write a program which takes a file, reads it, filters comments away and writes a new file with the remaining text.
3. In Objective Caml character strings may contain any character, even the sequences `(*` and `*)`. For example, character string `"what(*ever te*)xt"` should not be considered a comment. Modify your lexer to consider character strings.
4. Use this new lexer to remove comments from an Objective Caml program .

Evaluator

We will use `ocamlyacc` to implement an expression evaluator. The idea is to perform the evaluation of expressions directly in the grammar rules.

We choose a (completely parenthesized) prefix arithmetic expression language with variable arity operators. For example, expression `(ADD e1 e2 .. en)` is equivalent to `e1 + e2 + .. + en`. Plus and times operators are right-associative and subtraction and division are left-associative.

1. Define in file `opn_parser.mly` the parsing and evaluation rules for an expression.

2. Define in file `opn_lexer.mll` the lexical analysis of expressions.
3. Write a simple main program `opn` which reads a line from standard input containing an expression and prints the result of evaluating the expression.

Summary

This chapter has introduced several Objective Caml tools for lexical analysis (lexing) and syntax analysis (parsing). We explored (in order of occurrence):

- module `Str` to filter rational expressions;
- module `Genlex` to easily build simple lexers;
- the `ocamllex` tool, a typed integration of the `lex` tool;
- the `ocamlyacc` tool, a typed integration of the `yacc` tool;
- the use of streams to build top-down parsers, including contextual parsers.

Tools `ocamllex` and `ocamlyacc` were used to define a parser for the language Basic more easily maintained than that introduced in page 159.

To Learn More

The reference book on lexical analysis and parsing is known affectionately as the "dragon book", a reference to the book's cover illustration. Its real name is *Compilers: principles, techniques and tools* ([ASU86]). It covers all aspects of compiler design and implementation. It explains clearly the construction of automata matching a given context-free grammar and the techniques to minimize it. The tools `lex` and `yacc` are described in-depth in several books, a good reference being [LMB92]. The interesting features of `ocamllex` and `ocamlyac` with respect to their original versions are the integration of the Objective Caml language and, above all, the ability to write typed lexers and parsers. With regard to *streams*, the research report by Michel Mauny and Daniel de Rauglaudre [MdR92] gives a good description of the operational semantics of this extension. On the other hand, [CM98] shows how to build such an extension. For a better integration of grammars within the Objective Caml language, or to modify the grammars of the latter, we may also use the `camlp4` tool found at:

Link: http://caml.inria.fr/camlp4/