Lexical Analysis with ocamllex

Letterio Galletta

Agenda

- Examples of hand-coded scanners
- Tutorial ocamllex
- Some examples of scanners

Lexical analysis (scanning)

Translate a stream of characters to a stream of **tokens**

result _ =_ i_*_bar (0, _42, _q);

ID	EQ	ID	TIMES	ID	LPAREN	INT	COMMA
INT	COMMA	ID	RPAREN	SEMICOL			

Lexical analysis (scanning)

Translate a stream of characters to a stream of **tokens**

result _ =_ i_*_bar (0, _42, _q);

Token	**Lexeme**	**Pattern**
EQ	=	An equal sign
TIMES	*	A star symbol
ID	result, i, bar, q	A letter followed by a letter or digit
INT	0, 42	One or more digit

Lexical analysis (scanning)

Goals:

- Simplify the job of the parser
- Discard as many irrelevant details as possible (e.g., whitespace, comments)

Describing tokens:

- Tokens are usually specified through regular expressions

Implementing scanners

Two approaches:

- Use scanner generators, e.g., flex, ocamllex, etc.

  They take as input a description of tokens and code to run when a token is recognised

- Hand-coded scanners

  You need to implement all the details: input management, lexeme extraction, etc.

Hand-coded scanners

Implement all the code for recognizing the tokens by hand

Typically, it consists of

- A mutable data structure to store the state of the scanner, e.g., the input buffer, line numbers, errors, etc.
- A function for each class of complex token, i.e., string literals, integers, comments, identifiers
- The main procedure is a big switch on the current characters (or some lookahead characters)

Hand-coded scanners: examples

Toy compilers:

- Lox language from the book Crafting Interpreters

  See Scanner.java (homework read Chapter 4)

- Subset of Pascal language from the book Writing Compilers and Interpreters

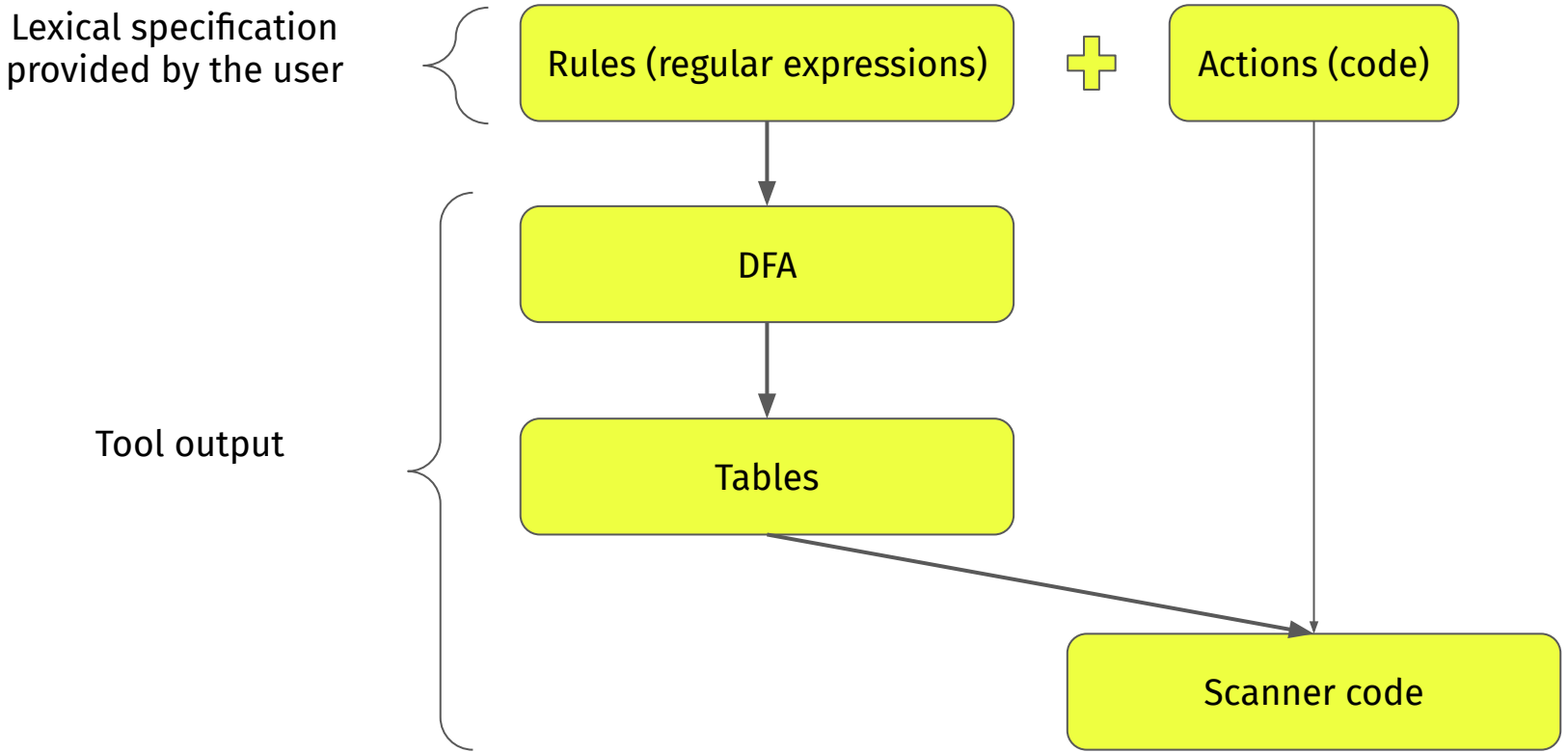
  See code of Chapter 3 package Frontend

Hand-coded scanners: examples

Real world compilers:

- Golang, see scanner.go
- Rust, see rustc_lexer/src/lib.rs
- Python (CPython implementation), see Parser/tokenizer.h Parser/tokenizer.c
- Clang, see Lex/Lexer.h and Lex/Lexer.c
- Many others, e.g., Solidity, V8, Scala, ...

Implementing Scanner Automatically

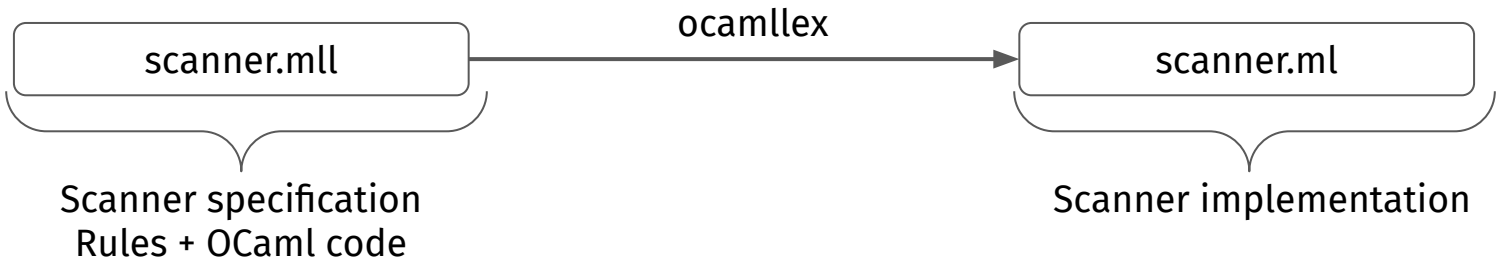


Generated scanners: examples

Real world compilers:

- OCaml, see lex/lexer.mll
- Haskell, see GHC/Parser/Lexer.x
- FSharp, see fsharp/lex.fsl

Constructing Scanners with ocamllex



- Typically, already installed with the ocaml platform
- The generated file is compiled and linked with the application
- The scanner is available through a specific function
- The scanner analyzes the input for occurrences of regular expressions, whenever it finds one, it executes the corresponding actions

A first example

The idea:

a scanner that whenever it finds the string “current_directory” will replace it with the path of the current directory

Two rules:

1. Handling the occurrences of “current_directory”
2. Every other characters

A first example

```
{ }

rule translate = parse
| "current_directory"        { print_string (Sys.getcwd ()); translate lexbuf }
| _ as c                     { print_char c; translate lexbuf }
| eof                        { exit 0 }

{
let _ =
  let lexbuf = Lexing.from_channel stdin in
  translate lexbuf
}
```

See cwd.mll

A first example

The specification defines a scanner named translate, there are three parts in the specification:

1. An empty header section
2. The definition of the scanner translate
3. A trailer section for code: create a buffer for the lexer from a channel and invoke the lexer translate

Note: other OCaml code see the scanner as a function taking a buffer and returning a result (a unit value in this case, why?)

A first example

In the translate scanner there are three rules:

1. The literal "current_directory" is the pattern and the expression between braces is the action: it prints the current directory name and call the scanner again (to analyze the rest of the input)
2. Any other character which is not matched by the first rule: the action copies the character to its output and call the scanner again
3. Match the end-of-file: the program exit

Note: the rules should cover any possible input, any text not matched by a scanner generates exception Failure "lexing: empty token"

The generated scanner

The output of ocamllex lex.mll is a file lex.ml that includes:

- The scanning functions
- A set of tables used by the scanning functions the code of the header and trailer

The scanning functions are declared as

```
let entrypoint1 [arg1... argn] lexbuf =
...
and entrypoint2 [arg1... argm] lexbuf =
...
```

A second example

The idea:

a scanner that counts the number of characters and the number of lines in its input and reports on the final counts

Two rules:

1. Handling the occurrences of new lines
2. Every other characters

A second example

```
{
let num_lines = ref 0
let num_chars = ref 0
}
rule count = parse
| '\n'          { incr num_lines; incr num_chars; count lexbuf }
| _             { incr num_chars; count lexbuf }
| eof           { () }
{
let () =
let lexbuf = Lexing.from_channel stdin in
count lexbuf;
Printf.printf "# of lines = %d, # of chars = %d\n" !num_lines !num_chars
}
```

See count.mll

A second example

The specification defines a scanner named count:

1. The header section declares to global mutable variables recording the number of read lines and characters
2. The count defines tree rules:
   a. The first deals with the new line characters: increments both counters and call the scanner again
   b. The second deals with a character which is not a new line: increment the corresponding counter and call again the scanner
   c. The third deals with end-of-file: it ends the scanner
3. The trailer section for code: create a buffer for the lexer, invoke it and prints the reports

Ocamllex specification

```
{ header }              (* header section: verbatim OCaml code; mandatory *)

let ident = regexp   (* definition section: optional *)
let ...

rule entrypoint [arg1... argn] = parse          (* rules section; mandatory  *)
| pattern            { action }                 (* OCaml code *)
| ...
| pattern            { action }
and ...
{ trailer }                    (* trailer section: verbatim OCaml code; optional *)
```

Format of the input file: header and trailer

The header and the trailer sections can contain arbitrary OCaml code: they are copied at the beginning and at the end of the output file, resp.

You can code open directives and some auxiliary functions in the header section, test functions in the trailer section

Format of the input file: definition

The definitions section contains declarations of simple identifier

```
let ident = regexp
let ...
```

The ident must be valid OCaml identifiers (starting with a lowercase letter). For example,

```
let digit = ['0'-'9']
let id = ['a'-'z']['a'-'z' '0'-'9']*
```

Can be used in a complex pattern, e.g. `digit+ "." digit*`

Patterns 1 (in order of decreasing precedence)

Pattern	Meaning
‘c’	The character “c”
_	Any character
eof	The end-of-file
“foo”	A literal string, e.g., the “foo” string
[‘1’ ‘5’ ‘a’-’z’]	Set of characters and ranges: the character ‘1’ or ‘5’ or any lower case letter
[^ ‘0’ - ‘9’]	Negated character set, e.g., any characters except a digit
(pattern)	Grouping

Patterns 2 (in order of decreasing precedence)

Pattern	Meaning
identifier	A pattern defined in the definition section
pattern*	Zero or more pattern
pattern+	One or more pattern
pattern?	Zero or one pattern
pattern1 pattern2	pattern1 followed by pattern2
pattern1 | pattern2	Either pattern1 or pattern2
pattern as id	Bind the matched pattern to variable id inside the action

Example 3: a simple tokenizer

See the file stokens.mll

The scanner recognize 4 types of tokens:

1. An operator symbol, +
2. A simple keyword, if
3. Identifiers
4. Numbers

And prints them in output

Functions to create a lexbuf

These are the main functions to create a lexbuf (see doc for others)

Functions	Meaning
Lexing.from_channel inchan	returns a lexer buffer which reads from an input channel inchan
Lexing.from_string str	returns a lexer buffer which reads from an input channel str

The functions take an optional parameter to disable the position tracking (see later)

The longest match principles

If ocamllex finds more than one pattern matching the input it takes the longer one

```
rule token = parse
  | "ding"          { print_endline "Ding" } (* "ding" pattern *)
  | "dong"          { print_endline "Dong" } (* "dong" pattern *)
  | "dingdong"      { print_endline "Ding-Dong" } (* "dingdong" pattern *)
```

When “dingdong” is given as input the last rule applies

Use shortest keyword to match the shortest prefix of the input

The first match principles

If ocamllex finds more than one pattern of the same length matching the input it takes the first one

```
rule token = parse
  | "ding"                { print_endline "Ding" } (* "ding" pattern *)
  | ['a'-'z']+ as word    { print_endline ("Word: " ^ word) } (* "word" pattern *)
```

When “ding” is given as input the first rule applies

Actions

Actions can include arbitrary code that returns a value

Each time we call the scanner function is called it continues processing tokens from where it last stopped

Inside actions:

- the variable `lexbuf` is bound to the current lexer buffer
- the identifiers following the keyword `as` are bound to the matched string

Functions for lexer semantic actions

These functions can be called from the semantic actions (see doc)

Functions	**Meaning**
Lexing.lexeme lexbuf	returns the string matched by the regular expression
Lexing.lexeme_char lexbuf i	returns character number i in the matched string
Lexing.lexeme_start lexbuf	returns the offset in the input stream of the first character of the matched string
Lexing.lexeme_end lexbuf	returns the offset in the input stream of the character following the last character of the matched string

Lexer position 1

A point in a source file is described by a position value (see doc)

```
type position = {
    pos_fname : string;
    pos_lnum : int;
    pos_bol : int;
    pos_cnum : int;
}
```

- pos_fname is the file name
- pos_lnum is the line number
- pos_bol is number of characters between the beginning of the lexbuf and the beginning of the line
- pos_cnum is number of characters between the beginning of the lexbuf and the current position

pos_cnum-pos_bol is the character offset within the line

Lexer position 2

These functions can be called from the semantic actions (see doc)

Functions	Meaning
Lexing.lexeme_start_p lexbuf	returns a position value describing where the lexeme starts. When position tracking is disabled, the function returns dummy_pos
Lexing.lexeme_end_p lexbuf	returns a position value describing where the lexeme ends. When position tracking is disabled, the function returns dummy_pos

The position tracking is managed by the scanner engine

Example 4: a simple tokenizer with position

See the file stokens_pos.mll

The scanner recognize 4 types of tokens and prints them in output together with their position

Start conditions

ocamllex provides a mechanism for conditionally activating rules: just call the corresponding entrypoint function

```
{}
rule token = parse
  | [' ' '\t' '\n']+     { token lexbuf }     (* skip spaces *)
  | "(*"                 { comment lexbuf } (* activate "comment" rule *)
  ...
and comment = parse
| "*)"                   { token lexbuf }   (* go to the "token" rule *)
| _                      { comment lexbuf } (* skip comments *)
```

Example 5: managing nested comments

See the file comments.mll

The main idea is that the scanner function comment take the nesting level as parameter: when we detect the end of a comment and the level is 0, we call back the scanner function token

Example 6: CSV scanner

See csv_simple.mll

The comma separated values format (CSV) has been used for exchanging and converting data in tabular form.

It is specified by the RFC4180

Keyword hashtable 1

To keep the generated automaton small it is convenient to store the keywords in a hashtable

```
{
let keyword_table = Hashtbl.create 72
let _ =   List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
                    [ ("keyword1", KEYWORD1);
                      ("keyword2", KEYWORD2);
                      ...
                    ]
}
```

Keyword hashtable 2

In the action use the table to distinguish between a keyword and an identifier

```
rule token = parse
  | ...
  | ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']* as id
      { try
          Hashtbl.find keyword_table id
        with Not_found -> IDENT id
      }
  | ...
```

Example 7: a toy language

See toy_lang.mll

A simple language with identifiers, numbers, and operators

ocamllex: limitations

- No support for unicode character set

  To overcome this issue is possible to use an alternative scanner generator sedlex

- Limited API

Conclusions

- Examples of handwritten scanners
- Tutorial ocamllex
- Some examples of scanners

References

- OCamllex tutorial
- ocamllex documentation
- Lexing API documentation