# 20 Parsing with OCamllex and Menhir

*This chapter includes contributions from Jason Hickey.*

Many programming tasks start with the interpretation of some form of structured textual data. *Parsing* is the process of converting such data into data structures that are easy to program against. For simple formats, it's often enough to parse the data in an ad hoc way, say, by breaking up the data into lines, and then using regular expressions for breaking those lines down into their component pieces.

But this simplistic approach tends to fall down when parsing more complicated data, particularly data with the kind of recursive structure you find in full-blown programming languages or flexible data formats like JSON and XML. Parsing such formats accurately and efficiently while providing useful error messages is a complex task.

Often, you can find an existing parsing library that handles these issues for you. But there are tools to simplify the task when you do need to write a parser, in the form of *parser generators*. A parser generator creates a parser from a specification of the data format that you want to parse, and uses that to generate a parser.

Parser generators have a long history, including tools like `lex` and `yacc` that date back to the early 1970s. OCaml has its own alternatives, including `ocamllex`, which replaces `lex`, and `ocamlyacc` and `menhir`, which replace `yacc`. We'll explore these tools in the course of walking through the implementation of a parser for the JSON serialization format that we discussed in Chapter 19 (Handling JSON Data).

Parsing is a broad and often intricate topic, and our purpose here is not to teach all of the theoretical issues, but to provide a pragmatic introduction of how to build a parser in OCaml.

### Menhir Versus ocamlyacc

Menhir is an alternative parser generator that is generally superior to the venerable `ocamlyacc`, which dates back quite a few years. Menhir is mostly compatible with `ocamlyacc` grammars, and so you can usually just switch to Menhir and expect older code to work (with some minor differences described in the Menhir manual).

The biggest advantage of Menhir is that its error messages are generally more human-comprehensible, and the parsers that it generates are fully reentrant and can be parameterized in OCaml modules more easily. We recommend that any new code you develop should use Menhir instead of `ocamlyacc`.

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

> Menhir isn't distributed directly with OCaml but is available through OPAM by running `opam install menhir`.

## 20.1 Lexing and Parsing

Parsing is traditionally broken down into two parts: *lexical analysis*, which is a kind of simplified parsing phase that converts a stream of characters into a stream of logical tokens; and full-on parsing, which involves converting a stream of tokens into the final representation, which is often in the form of a tree-like data structure called an *abstract syntax tree*, or AST.

It's confusing that the term parsing is applied to both the overall process of converting textual data to structured data, and also more specifically to the second phase of converting a stream of tokens to an AST; so from here on out, we'll use the term parsing to refer only to this second phase.

Let's consider lexing and parsing in the context of the JSON format. Here's a snippet of text that represents a JSON object containing a string labeled `title` and an array containing two objects, each with a name and array of zip codes:

```
{
  "title": "Cities",
  "cities": [
    { "name": "Chicago",  "zips": [60601] },
    { "name": "New York", "zips": [10004] }
  ]
}
```

At a syntactic level, we can think of a JSON file as a series of simple logical units, like curly braces, square brackets, commas, colons, identifiers, numbers, and quoted strings. Thus, we could represent our JSON text as a sequence of tokens of the following type:

```
type token =
  | NULL
  | TRUE
  | FALSE
  | STRING of string
  | INT of int
  | FLOAT of float
  | LEFT_BRACK
  | RIGHT_BRACK
  | LEFT_BRACE
  | RIGHT_BRACE
  | COMMA
  | COLON
  | EOF
```

Note that this representation loses some information about the original text. For example, whitespace is not represented. It's common, and indeed useful, for the token stream to forget some details of the original text that are not required for understanding its meaning.

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

If we converted the preceding example into a list of these tokens, it would look something like this:

```
[ LEFT_BRACE; STRING("title"); COLON; STRING("Cities");
  COMMA; STRING("cities"); ... ]
```

This kind of representation is easier to work with than the original text, since it gets rid of some unimportant syntactic details and adds useful structure. But it's still a good deal more low-level than the simple AST we used for representing JSON data in Chapter 19 (Handling JSON Data):

```
type value =
  [ `Assoc of (string * value) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of value list
  | `Null
  | `String of string ]
```

This representation is much richer than our token stream, capturing the fact that JSON values can be nested inside each other and that JSON has a variety of value types, including numbers, strings, arrays, and objects. The parser we'll write will convert a token stream into a value of this AST type, as shown below for our earlier JSON example:

```
`Assoc
  ["title", `String "Cities";
   "cities", `List
     [`Assoc ["name", `String "Chicago"; "zips", `List [`Int 60601]];
      `Assoc ["name", `String "New York"; "zips", `List [`Int
     10004]]]]
```

## 20.2 Defining a Parser

A parser-specification file has suffix `.mly` and contains two sections that are broken up by separator lines consisting of the characters `%%` on a line by themselves. The first section of the file is for declarations, including token and type specifications, precedence directives, and other output directives; and the second section is for specifying the grammar of the language to be parsed.

We'll start by declaring the list of tokens. A token is declared using the syntax `%token` *<type>uid*, where the *<type>* is optional and *uid* is a capitalized identifier. For JSON, we need tokens for numbers, strings, identifiers, and punctuation:

```
%token <int> INT
%token <float> FLOAT
%token <string> ID
%token <string> STRING
%token TRUE
%token FALSE
%token NULL
```

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

```
%token LEFT_BRACE
%token RIGHT_BRACE
%token LEFT_BRACK
%token RIGHT_BRACK
%token COLON
%token COMMA
%token EOF
```

The <*type*> specifications mean that a token carries a value. The `INT` token carries an integer value with it, `FLOAT` has a `float` value, and `STRING` carries a `string` value. The remaining tokens, such as `TRUE`, `FALSE`, or the punctuation, aren't associated with any value, and so we can omit the <*type*> specification.

### 20.2.1 Describing the Grammar

The next thing we need to do is to specify the grammar of a JSON expression. `menhir`, like many parser generators, expresses grammars as *context-free grammars*. (More precisely, `menhir` supports LR(1) grammars, but we will ignore that technical distinction here.) You can think of a context-free grammar as a set of abstract names, called *non-terminal symbols*, along with a collection of rules for transforming a nonterminal symbol into a sequence of tokens and nonterminal symbols. A sequence of tokens is parsable by a grammar if you can apply the grammar's rules to produce a series of transformations, starting at a distinguished *start symbol* that produces the token sequence in question.

We'll start describing the JSON grammar by declaring the start symbol to be the non-terminal symbol `prog`, and by declaring that when parsed, a `prog` value should be converted into an OCaml value of type `Json.value option`. We then end the declaration section of the parser with a `%%`:

```
%start <Json.value option> prog
%%
```

Once that's in place, we can start specifying the productions. In `menhir`, productions are organized into *rules*, where each rule lists all the possible productions for a given nonterminal symbol. Here, for example, is the rule for `prog`:

```
prog:
  | EOF       { None }
  | v = value { Some v }
  ;
```

The syntax for this is reminiscent of an OCaml `match` expression. The pipes separate the individual productions, and the curly braces contain a *semantic action*: OCaml code that generates the OCaml value corresponding to the production in question. Semantic actions are arbitrary OCaml expressions that are evaluated during parsing to produce values that are attached to the non-terminal in the rule.

We have two cases for `prog`: either there's an `EOF`, which means the text is empty, and so there's no JSON value to read, we return the OCaml value `None`; or we have an instance of the `value` nonterminal, which corresponds to a well-formed JSON value,

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

and we wrap the corresponding `Json.value` in a `Some` tag. Note that in the `value` case, we wrote `v = value` to bind the OCaml value that corresponds to the variable `v`, which we can then use within the curly braces for that production.

Now let's consider a more complex example, the rule for the `value` symbol:

```
value:
  | LEFT_BRACE; obj = object_fields; RIGHT_BRACE
    { `Assoc obj }
  | LEFT_BRACK; vl = array_values; RIGHT_BRACK
    { `List vl }
  | s = STRING
    { `String s }
  | i = INT
    { `Int i }
  | x = FLOAT
    { `Float x }
  | TRUE
    { `Bool true }
  | FALSE
    { `Bool false }
  | NULL
    { `Null }
  ;
```

According to these rules, a JSON `value` is either:

- An object bracketed by curly braces
- An array bracketed by square braces
- A string, integer, float, bool, or null value

In each of the productions, the OCaml code in curly braces shows what to transform the object in question to. Note that we still have two nonterminals whose definitions we depend on here but have not yet defined: `object_fields` and `array_values`. We'll look at how these are parsed next.

### 20.2.2 Parsing Sequences

The rule for `object_fields` follows, and is really just a thin wrapper that reverses the list returned by the following rule for `rev_object_fields`. Note that the first production in `rev_object_fields` has an empty left-hand side, because what we're matching on in this case is an empty sequence of tokens. The comment `(* empty *)` is used to make this clear:

```
object_fields: obj = rev_object_fields { List.rev obj };

rev_object_fields:
  | (* empty *) { [] }
  | obj = rev_object_fields; COMMA; k = ID; COLON; v = value
    { (k, v) :: obj }
  ;
```

The rules are structured as they are because `menhir` generates left-recursive parsers, which means that the constructed pushdown automaton uses less stack space with

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

left-recursive definitions. The following right-recursive rule accepts the same input, but during parsing, it requires linear stack space to read object field definitions:

```
(* Inefficient right-recursive rule *)
object_fields:
  | (* empty *) { [] }
  | k = STRING; COLON; v = value; COMMA; obj = object_fields
    { (k, v) :: obj }
```

Alternatively, we could keep the left-recursive definition and simply construct the returned value in left-to-right order. This is even less efficient, since the complexity of building the list incrementally in this way is quadratic in the length of the list:

```
(* Quadratic left-recursive rule *)
object_fields:
  | (* empty *) { [] }
  | obj = object_fields; COMMA; k = STRING; COLON; v = value
    { obj @ [k, v] }
  ;
```

Assembling lists like this is a pretty common requirement in most realistic grammars, and the preceding rules (while useful for illustrating how parsing works) are rather verbose. Menhir features an extended standard library of built-in rules to simplify this handling. These rules are detailed in the Menhir manual and include optional values, pairs of values with optional separators, and lists of elements (also with optional separators).

A version of the JSON grammar using these more succinct Menhir rules follows. Notice the use of `separated_list` to parse both JSON objects and lists with one rule:

```
%token <int> INT
%token <float> FLOAT
%token <string> STRING
%token TRUE
%token FALSE
%token NULL
%token LEFT_BRACE
%token RIGHT_BRACE
%token LEFT_BRACK
%token RIGHT_BRACK
%token COLON
%token COMMA
%token EOF
%start <Json.value option> prog
%%

prog:
  | v = value { Some v }
  | EOF       { None   } ;

value:
  | LEFT_BRACE; obj = obj_fields; RIGHT_BRACE { `Assoc obj  }
  | LEFT_BRACK; vl = list_fields; RIGHT_BRACK { `List vl    }
  | s = STRING                                { `String s   }
  | i = INT                                   { `Int i      }
  | x = FLOAT                                 { `Float x    }
```

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

```
  | TRUE                                        { `Bool true  }
  | FALSE                                       { `Bool false }
  | NULL                                        { `Null       } ;

obj_fields:
    obj = separated_list(COMMA, obj_field)      { obj } ;

obj_field:
    k = STRING; COLON; v = value                { (k, v) } ;

list_fields:
    vl = separated_list(COMMA, value)           { vl } ;
```

We can hook in `menhir` by adding a `(menhir)` stanza to our dune file, which tells the build system to switch to using `menhir` instead of `ocamlyacc` to handle files with the `.mly` suffix:

```
(menhir
 (modules parser))

(ocamllex lexer)

(library
 (name json_parser)
 (modules parser lexer json)
 (libraries core))
```

## 20.3 Defining a Lexer

Now we can define a lexer, using `ocamllex`, to convert our input text into a stream of tokens. The specification of the lexer is placed in a file with an `.mll` suffix.

### 20.3.1 OCaml Prelude

Let's walk through the definition of a lexer section by section. The first section is an optional chunk of OCaml code that is bounded by a pair of curly braces:

```
{
open Lexing
open Parser

exception SyntaxError of string
}
```

This code is there to define utility functions used by later snippets of OCaml code and to set up the environment by opening useful modules and define an exception, `SyntaxError`. Any OCaml functions you define here will be subsequently available in the remainder of the lexer definition.

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

### 20.3.2 Regular Expressions

The next section of the lexing file is a collection of named regular expressions. These look syntactically like ordinary OCaml `let` bindings, but really this is a specialized syntax for declaring regular expressions. Here's an example:

```
let int = '-'? ['0'-'9'] ['0'-'9']*
```

The syntax here is something of a hybrid between OCaml syntax and traditional regular expression syntax. The `int` regular expression specifies an optional leading `-`, followed by a digit from `0` to `9`, followed by some number of digits from `0` to `9`. The question mark is used to indicate an optional component of a regular expression; the square brackets are used to specify ranges; and the `*` operator is used to indicate a (possibly empty) repetition.

Floating-point numbers are specified similarly, but we deal with decimal points and exponents. We make the expression easier to read by building up a sequence of named regular expressions, rather than creating one big and impenetrable expression:

```
let digit = ['0'-'9']
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let float = digit* frac? exp?
```

Finally, we define whitespace, newlines, and identifiers:

```
let white = [' ' '\t']+
let newline = '\r' | '\n' | "\r\n"
let id = ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

The `newline` introduces the `|` operator, which lets one of several alternative regular expressions match (in this case, the various carriage-return combinations of CR, LF, or CRLF).

### 20.3.3 Lexing Rules

The lexing rules are essentially functions that consume the data, producing OCaml expressions that evaluate to tokens. These OCaml expressions can be quite complicated, using side effects and invoking other rules as part of the body of the rule. Let's look at the `read` rule for parsing a JSON expression:

```
rule read =
  parse
  | white    { read lexbuf }
  | newline  { next_line lexbuf; read lexbuf }
  | int      { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | float    { FLOAT (float_of_string (Lexing.lexeme lexbuf)) }
  | "true"   { TRUE }
  | "false"  { FALSE }
  | "null"   { NULL }
  | '"'      { read_string (Buffer.create 17) lexbuf }
  | '{'      { LEFT_BRACE }
  | '}'      { RIGHT_BRACE }
  | '['      { LEFT_BRACK }
```

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

```
  | ']'      { RIGHT_BRACK }
  | ':'      { COLON }
  | ','      { COMMA }
  | _ { raise (SyntaxError ("Unexpected char: " ^ Lexing.lexeme
    lexbuf)) }
  | eof      { EOF }
```

The rules are structured very similarly to pattern matches, except that the variants are replaced by regular expressions on the left-hand side. The right-hand side clause is the parsed OCaml return value of that rule. The OCaml code for the rules has a parameter called `lexbuf` that defines the input, including the position in the input file, as well as the text that was matched by the regular expression.

The first `white { read lexbuf }` calls the lexer recursively. That is, it skips the input whitespace and returns the following token. The action `newline { next_line lexbuf; read lexbuf }` is similar, but we use it to advance the line number for the lexer using the utility function that we defined at the top of the file. Let's skip to the third action:

```
| int { INT (int_of_string (Lexing.lexeme lexbuf)) }
```

This action specifies that when the input matches the `int` regular expression, then the lexer should return the expression `INT (int_of_string (Lexing.lexeme lexbuf))`. The expression `Lexing.lexeme lexbuf` returns the complete string matched by the regular expression. In this case, the string represents a number, so we use the `int_of_string` function to convert it to a number.

There are actions for each different kind of token. The string expressions like `"true" { TRUE }` are used for keywords, and the special characters have actions, too, like `'{' { LEFT_BRACE }`.

Some of these patterns overlap. For example, the regular expression `"true"` is also matched by the `id` pattern. `ocamllex` used the following disambiguation when a prefix of the input is matched by more than one pattern:

- The longest match always wins. For example, the first input `trueX: 167` matches the regular expression `"true"` for four characters, and it matches `id` for five characters. The longer match wins, and the return value is `ID "trueX"`.
- If all matches have the same length, then the first action wins. If the input were `true: 167`, then both `"true"` and `id` match the first four characters; `"true"` is first, so the return value is `TRUE`.

### Unused Lexing Values

In our parser, we have not used all the token regexps that we defined in the lexer. For instance, `id` is unused since we do not parse unquoted strings for object identifiers (something that is allowed by JavaScript, but not the subset of it that is JSON). If we included a token pattern match for this in the lexer, then we would have to adjust the parser accordingly to add a `%token <string> ID`. This would in turn trigger an "unused" warning since the parser never constructs a value with type ID:

```
File "parser.mly", line 4, characters 16-18:
```

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

```
Warning: the token ID is unused.
```

It's completely fine to define unused regexps as we've done, and to hook them into parsers as required. For example, we might use `ID` if we add an extension to our parser for supporting unquoted string identifiers as a non-standard JSON extension.

### 20.3.4 Recursive Rules

Unlike many other lexer generators, `ocamllex` allows the definition of multiple lexers in the same file, and the definitions can be recursive. In this case, we use recursion to match string literals using the following rule definition:

```
and read_string buf =
  parse
  | '"'       { STRING (Buffer.contents buf) }
  | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _ { raise (SyntaxError ("Illegal string character: " ^
    Lexing.lexeme lexbuf)) }
  | eof { raise (SyntaxError ("String is not terminated")) }
```

This rule takes a `buf : Buffer.t` as an argument. If we reach the terminating double quote ", then we return the contents of the buffer as a `STRING`.

The other cases are for handling the string contents. The action `[^ '"' '\\']+ { ... }` matches normal input that does not contain a double quote or backslash. The actions beginning with a backslash \ define what to do for escape sequences. In each of these cases, the final step includes a recursive call to the lexer.

That covers the lexer. Next, we need to combine the lexer with the parser to bring it all together.

#### Handling Unicode

We've glossed over an important detail here: parsing Unicode characters to handle the full spectrum of the world's writing systems. OCaml has several third-party solutions to handling Unicode, with varying degrees of flexibility and complexity:

- Uutfa is a nonblocking streaming Unicode codec for OCaml, available as a standalone library. It is accompanied by the Uunfb text normalization and Uucdc Unicode character database libraries. There is also a robust parser for JSONd available that illustrates the use of Uutf in your own libraries.

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

- Camomilee supports the full spectrum of Unicode character types, conversion from around 200 encodings, and collation and locale-sensitive case mappings.
- sedlexf is a lexer generator for Unicode that can serve as a Unicode-aware replacement for `ocamllex`.

All of these libraries are available via opam under their respective names.

a http://erratique.ch/software/uutf
b http://erratique.ch/software/uunf
c http://erratique.ch/software/uucd
d http://erratique.ch/software/jsonm
e https://github.com/yoriyuki/Camomile
f https://github.com/ocaml-community/sedlex

## 20.4 Bringing It All Together

For the final part, we need to compose the lexer and parser. As we saw in the type definition in `parser.mli`, the parsing function expects a lexer of type `Lexing.lexbuf -> token`, and a `lexbuf`:

```
val prog : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Json.value
    option
```

Before we start with the lexing, let's first define some functions to handle parsing errors. There are currently two errors: `Parser.Error` and `Lexer.SyntaxError`. A simple solution when encountering an error is to print the error and give up:

```
open Core
open Lexer
open Lexing

let print_position outx lexbuf =
  let pos = lexbuf.lex_curr_p in
  fprintf outx "%s:%d:%d" pos.pos_fname
    pos.pos_lnum (pos.pos_cnum - pos.pos_bol + 1)

let parse_with_error lexbuf =
  try Parser.prog Lexer.read lexbuf with
  | SyntaxError msg ->
    fprintf stderr "%a: %s\n" print_position lexbuf msg;
    None
  | Parser.Error ->
    fprintf stderr "%a: syntax error\n" print_position lexbuf;
    exit (-1)
```

The "give up on the first error" approach is easy to implement but isn't very friendly. In general, error handling can be pretty intricate, and we won't discuss it here. However, the Menhir parser defines additional mechanisms you can use to try and recover from errors. These are described in detail in its reference manual[1].

The standard lexing library `Lexing` provides a function `from_channel` to read

[1] http://gallium.inria.fr/~fpottier/menhir/

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

the input from a channel. The following function describes the structure, where the `Lexing.from_channel` function is used to construct a `lexbuf`, which is passed with the lexing function `Lexer.read` to the `Parser.prog` function. `Parsing.prog` returns `None` when it reaches end of file. We define a function `Json.output_value`, not shown here, to print a `Json.value`:

```
let rec parse_and_print lexbuf =
  match parse_with_error lexbuf with
  | Some value ->
    printf "%a\n" Json.output_value value;
    parse_and_print lexbuf
  | None -> ()

let loop filename () =
  let inx = In_channel.create filename in
  let lexbuf = Lexing.from_channel inx in
  lexbuf.lex_curr_p <- { lexbuf.lex_curr_p with pos_fname = filename
    };
  parse_and_print lexbuf;
  In_channel.close inx

let () =
  Command.basic_spec ~summary:"Parse and display JSON"
    Command.Spec.(empty +> anon ("filename" %: string))
    loop
  |> Command.run
```

Here's a test input file we can use to test the code we just wrote:

```
true
false
null
[1, 2, 3., 4.0, .5, 5.5e5, 6.3]
"Hello World"
{ "field1": "Hello",
  "field2": 17e13,
  "field3": [1, 2, 3],
  "field4": { "fieldA": 1, "fieldB": "Hello" }
}
```

Now build and run the example using this file, and you can see the full parser in action:

```
$ dune exec ./test.exe test1.json
true
false
null
[1, 2, 3.000000, 4.000000, 0.500000, 550000.000000, 6.300000]
"Hello World"
{ "field1": "Hello",
  "field2": 170000000000000.000000,
  "field3": [1, 2, 3],
  "field4": { "fieldA": 1,
  "fieldB": "Hello" } }
```

With our simple error handling scheme, errors are fatal and cause the program to terminate with a nonzero exit code:

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press

```
$ cat test2.json
{ "name": "Chicago",
  "zips": [12345,
}
{ "name": "New York",
  "zips": [10004]
}
$ dune exec ./test.exe test2.json
test2.json:3:2: syntax error
[255]
```

That wraps up our parsing tutorial. As an aside, notice that the JSON polymorphic variant type that we defined in this chapter is actually structurally compatible with the Yojson representation explained in Chapter 19 (Handling JSON Data). That means that you can take this parser and use it with the helper functions in Yojson to build more sophisticated applications.

https://doi.org/10.1017/9781009129220.023 Published online by Cambridge University Press