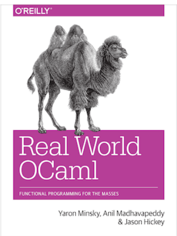

Buy in print and eBook.

Table of Contents

Chapter 16. Parsing with OCamllex and Menhir

Many programming tasks start with the interpretion 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.0 comments

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.0 comments

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.0 comments

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 15, *Handling JSON Data*.0 comments

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.0 comments

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).0 comments

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**.0 comments

Menhir isn't distributed directly with OCaml but is available through OPAM by running `opam install menhir`.0 comments

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.0 comments

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.0 comments

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:0 comments

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

```
  ]
}
```
JSON ∗ parsing/example.json ∗ all code

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:0 comments

```
type token =
  | NULL
  | TRUE
  | FALSE
  | STRING of string
  | INT of int
  | FLOAT of float
  | ID of string
  | LEFT_BRACK
  | RIGHT_BRACK
  | LEFT_BRACE
  | RIGHT_BRACE
  | COMMA
  | COLON
  | EOF
```
OCaml ∗ parsing/manual_token_type.ml ∗ all code

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.0 comments

If we converted the preceding example into a list of these tokens, it would look something like this:0 comments

```
[ LEFT_BRACE; ID("title"); COLON; STRING("Cities"); COMMA; ID("cities"); ...
```
OCaml ∗ parsing/tokens.ml ∗ all code

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 15, *Handling JSON Data*:0 comments

```
type value = [
  | `Assoc of (string * value) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of value list
  | `Null
  | `String of string
]
```
OCaml ∗ parsing/json.ml ∗ all code

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:0 comments

```
`Assoc
  ["title", `String "Cities";
   "cities", `List
     [`Assoc ["name", `String "Chicago"; "zips", `List [`Int 60601]];
      `Assoc ["name", `String "New York"; "zips", `List [`Int 10004]]]]
```
OCaml ∗ parsing/parsed_example.ml ∗ all code

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.0 comments

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:0 comments

```
%token <int> INT
%token <float> FLOAT
%token <string> ID
%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
```

OCaml ∗ parsing/parser.mly ∗ all code

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.0 comments

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.0 comments

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 `%%`:0 comments

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

OCaml ∗ parsing/parser.mly , continued (part 1) ∗ all code

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 symbols. Here, for example, is the rule for `prog`:0 comments

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

OCaml ∗ parsing/parser.mly , continued (part 2) ∗ all code

The syntax for this is reminiscent of an OCaml `match` statement. 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.0 comments

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, 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.0 comments

Now let's consider a more complex example, the rule for the `value` symbol:0 comments

```
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 }
  ;
```

OCaml ∗ parsing/parser.mly , continued (part 3) ∗ all code

According to these rules, a JSON `value` is either:0 comments

- An object bracketed by curly braces0 comments
- An array bracketed by square braces0 comments
- A string, integer, float, bool, or null value0 comments

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.0 comments

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 lefthand 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:0 comments

```
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 }
  ;
```

OCaml ∗ parsing/parser.mly , continued (part 4) ∗ all code

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 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:0 comments

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

OCaml ∗ parsing/right_rec_rule.mly , continued (part 4) ∗ all code

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:0 comments

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

OCaml ∗ parsing/quadratic_rule.mly , continued (part 4) ∗ all code

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).0 comments

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:0 comments

```
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    }
  | 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 } ;
```

OCaml ∗ parsing/short_parser.mly , continued (part 1) ∗ all code

We can invoke **menhir** by using **corebuild** with the `-use-menhir` flag. This tells the build system to switch to using **menhir** instead of **ocamlyacc** to handle files with the `.mly` suffix:0 comments

```
$ corebuild -use-menhir short_parser.mli
```

Terminal ∗ parsing/build_short_parser.out ∗ all code

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.0 comments

OCaml Prelude

Let's walk through the definition of a lexer section by section. The first section is on optional chunk of OCaml code that is bounded by a pair of curly braces:0 comments

```
{
open Lexing
open Parser

exception SyntaxError of string

let next_line lexbuf =
  let pos = lexbuf.lex_curr_p in
  lexbuf.lex_curr_p <-
    { pos with pos_bol = lexbuf.lex_curr_pos;
               pos_lnum = pos.pos_lnum + 1
    }
}
```

OCaml ∗ parsing/lexer.mll ∗ all code

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`.0 comments

We also define a utility function `next_line` for tracking the location of tokens across line breaks. The `Lexing` module defines a `lexbuf` structure that holds the state of the lexer, including the current location within the source file. The `next_line` function simply accesses the `lex_curr_p` field that holds the current location and updates its line number.0 comments

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:0 comments

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

OCaml ∗ parsing/lexer.mll , continued (part 1) ∗ all code

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.0 comments

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:0 comments

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

OCaml * parsing/lexer.mll , continued (part 2) * all code

Finally, we define whitespace, newlines, and identifiers:0 comments

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

OCaml * parsing/lexer.mll , continued (part 3) * all code

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).0 comments

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:0 comments

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

OCaml * parsing/lexer.mll , continued (part 4) * all code

The rules are structured very similarly to pattern matches, except that the variants are replaced by regular expressions on the lefthand side. The righthand-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.0 comments

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:0 comments

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

OCaml * parsing/lexer_int_fragment.mll * all code

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.0 comments

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 }`.0 comments

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:0 comments

- 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"`.0 comments
- 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`.0 comments

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:0 comments

```
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")) }
```

OCaml ∗ parsing/lexer.mll , continued (part 5) ∗ all code

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`.0 comments

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.0 comments

That covers the lexer. Next, we need to combine the lexer with the parser to bring it all together.0 comments

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:0 comments

- Camomile supports the full spectrum of Unicode character types, conversion from around 200 encodings, and collation and locale-sensitive case mappings.0 comments
- Ulex is a lexer generator for Unicode that can serve as a Unicode-aware replacement for **ocamllex**.0 comments
- Uutf is a nonblocking streaming Unicode codec for OCaml, available as a standalone library. It is accompanied by the Uunf text normalization and Uucd Unicode character database libraries. There is also a robust parser for JSON available that illustrates the use of Uutf in your own libraries.0 comments

All of these libraries are available via OPAM under their respective names.0 comments

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`:0 comments

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

OCaml ∗ parsing/prog.mli ∗ all code

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:0 comments

```
open Core.Std
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)
```

OCaml ∗ parsing-test/test.ml ∗ all code

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.0 comments

The standard lexing library `Lexing` provides a function `from_channel` to read 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`:0 comments

```
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
```

OCaml ∗ parsing-test/test.ml , continued (part 1) ∗ all code

Here's a test input file we can use to test the code we just wrote:0 comments

```
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" }
}
```

JSON ∗ parsing-test/test1.json ∗ all code

Now build and run the example using this file, and you can see the full parser in action:0 comments

```
$ ocamlbuild -use-menhir -tag thread -use-ocamlfind -quiet -pkg core test.native
$ ./test.native 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" } }
```

Terminal ∗ parsing-test/build_test.out ∗ all code

With our simple error handling scheme, errors are fatal and cause the program to terminate with a nonzero exit code:0 comments

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

Terminal ∗ parsing-test/run_broken_test.out ∗ all code

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 15, *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.0 comments