19 Handling JSON Data

Data serialization, i.e., converting data to and from a sequence of bytes that's suitable for writing to disk or sending across the network, is an important and common programming task. You often have to match someone else's data format (such as XML), sometimes you need a highly efficient format, and other times you want something that is easy for humans to edit. To this end, OCaml libraries provide several techniques for data serialization depending on what your problem is.

We'll start by using the popular and simple JSON data format and then look at other serialization formats later in the book. This chapter introduces you to a couple of new techniques that glue together the basic ideas from Part I of the book by using:

- *Polymorphic variants* to write more extensible libraries and protocols (but still retain the ability to extend them if needed)
- *Functional combinators* to compose common operations over data structures in a type-safe way
- External tools to generate boilerplate OCaml modules and signatures from external specification files

19.1 JSON Basics

JSON is a lightweight data-interchange format often used in web services and browsers. It's described in RFC4627[1] and is easier to parse and generate than alternatives such as XML. You'll run into JSON very often when working with modern web APIs, so we'll cover several different ways to manipulate it in this chapter.

JSON consists of two basic structures: an unordered collection of key/value pairs, and an ordered list of values. Values can be strings, Booleans, floats, integers, or null. Let's see what a JSON record for an example book description looks like:

```
{
  "title": "Real World OCaml",
  "tags" : [ "functional programming", "ocaml", "algorithms" ],
  "pages": 450,
  "authors": [
    { "name": "Jason Hickey", "affiliation": "Google" },
    { "name": "Anil Madhavapeddy", "affiliation": "Cambridge"},
```

[1] http://www.ietf.org/rfc/rfc4627.txt

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
      { "name": "Yaron Minsky", "affiliation": "Jane Street"}
  ],
  "is_online": true
}
```

The outermost JSON value is usually a record (delimited by the curly braces) and contains an unordered set of key/value pairs. The keys must be strings, but values can be any JSON type. In the preceding example, `tags` is a string list, while the `authors` field contains a list of records. Unlike OCaml lists, JSON lists can contain multiple different JSON types within a single list.

This free-form nature of JSON types is both a blessing and a curse. It's very easy to generate JSON values, but code that parses them also has to handle subtle variations in how the values are represented. For example, what if the preceding `pages` value is actually represented as a string value of "`450`" instead of an integer?

Our first task is to parse the JSON into a more structured OCaml type so that we can use static typing more effectively. When manipulating JSON in Python or Ruby, you might write unit tests to check that you have handled unusual inputs. The OCaml model prefers compile-time static checking as well as unit tests. For example, using pattern matching can warn you if you've not checked that a value can be `Null` as well as contain an actual value.

Installing the Yojson Library

There are several JSON libraries available for OCaml. For this chapter, we've picked the popular Yojson library, which you can install by running `opam install yojson`. Once installed, you can open it in `utop` as follows:

```
# open Core;;
# #require "yojson";;
# open Yojson;;
```

19.2 Parsing JSON with Yojson

The JSON specification has very few data types, and the `Yojson.Basic.t` type that follows is sufficient to express any valid JSON structure:

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

Some interesting properties should leap out at you after reading this definition:

- The `json` type is *recursive*, which is to say that some of the tags refer back to the

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

overall `json` type. In particular, `Assoc` and `List` types can contain references to further JSON values of different types. This is unlike the OCaml lists, whose contents must be of a uniform type.

- The definition specifically includes a `Null` variant for empty fields. OCaml doesn't allow null values by default, so this must be encoded explicitly.
- The type definition uses polymorphic variants and not normal variants. This will become significant later, when we extend it with custom extensions to the JSON format.

Let's parse the earlier JSON example into this type now. The first stop is the `Yojson.Basic` documentation, where we find these helpful functions:

```
val from_string : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int ->
   string -> json
(* Read a JSON value from a string.
   [buf]   : use this buffer at will during parsing instead of
             creating a new one.
   [fname] : data file name to be used in error messages. It does not
             have to be a real file.
   [lnum]  : number of the first line of input. Default is 1. *)

val from_file : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int ->
   string -> json
(* Read a JSON value from a file. See [from_string] for the meaning
    of the optional
   arguments. *)

val from_channel : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int ->
  in_channel -> json
  (** Read a JSON value from a channel.
      See [from_string] for the meaning of the optional arguments. *)
```

When first reading these interfaces, you can generally ignore the optional arguments (which have the question marks in the type signature), since they should have sensible defaults. In the preceding signature, the optional arguments offer finer control over the memory buffer allocation and error messages from parsing incorrect JSON.

The type signature for these functions with the optional elements removed makes their purpose much clearer. The three ways of parsing JSON are either directly from a string, from a file on a filesystem, or via a buffered input channel:

```
val from_string  : string     -> json
val from_file    : string     -> json
val from_channel : in_channel -> json
```

The next example shows both the `string` and `file` functions in action, assuming the JSON record is stored in a file called *book.json*:

```
open Core

let () =
  (* Read JSON file into an OCaml string *)
  let buf = In_channel.read_all "book.json" in
  (* Use the string JSON constructor *)
  let json1 = Yojson.Basic.from_string buf in
```

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
  (* Use the file JSON constructor *)
  let json2 = Yojson.Basic.from_file "book.json" in
  (* Test that the two values are the same *)
  print_endline (if Yojson.Basic.equal json1 json2 then "OK" else
    "FAIL")
```

You can build this by running `dune`:

```
$ dune exec -- ./read_json.exe
OK
```

The `from_file` function accepts an input filename and takes care of opening and closing it for you. It's far more common to use `from_string` to construct JSON values though, since these strings come in via a network connection (we'll see more of this in Chapter 17 (Concurrent Programming with Async)) or a database. Finally, the example checks that the two input mechanisms actually resulted in the same OCaml data structure.

19.3 Selecting Values from JSON Structures

Now that we've figured out how to parse the example JSON into an OCaml value, let's manipulate it from OCaml code and extract specific fields:

```
open Core

let () =
  (* Read the JSON file *)
  let json = Yojson.Basic.from_file "book.json" in

  (* Locally open the JSON manipulation functions *)
  let open Yojson.Basic.Util in
  let title = json |> member "title" |> to_string in
  let tags = json |> member "tags" |> to_list |> filter_string in
  let pages = json |> member "pages" |> to_int in
  let is_online = json |> member "is_online" |> to_bool_option in
  let is_translated = json |> member "is_translated" |>
    to_bool_option in
  let authors = json |> member "authors" |> to_list in
  let names = List.map authors ~f:(fun json -> member "name" json |>
    to_string) in

  (* Print the results of the parsing *)
  printf "Title: %s (%d)\n" title pages;
  printf "Authors: %s\n" (String.concat ~sep:", " names);
  printf "Tags: %s\n" (String.concat ~sep:", " tags);
  let string_of_bool_option =
    function
    | None -> "<unknown>"
    | Some true -> "yes"
    | Some false -> "no" in
  printf "Online: %s\n" (string_of_bool_option is_online);
  printf "Translated: %s\n" (string_of_bool_option is_translated)
```

Now build and run this in the same way as the previous example:

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
(executable
  (name      parse_book)
  (libraries core yojson))
```

```
$ dune build parse_book.exe
$ ./_build/default/parse_book.exe
Title: Real World OCaml (450)
Authors: Jason Hickey, Anil Madhavapeddy, Yaron Minsky
Tags: functional programming, ocaml, algorithms
Online: yes
Translated: <unknown>
```

This code introduces the `Yojson.Basic.Util` module, which contains *combinator* functions that let you easily map a JSON object into a more strongly typed OCaml value.

Functional Combinators

Combinators are a design pattern that crops up quite often in functional programming. John Hughes defines them as "a function which builds program fragments from program fragments." In a functional language, this generally means higher-order functions that combine other functions to apply useful transformations over values.

You've already run across several of these in the `List` module:

```
val map  : 'a list -> f:('a -> 'b)   -> 'b list
val fold : 'a list -> init:'accum -> f:('accum -> 'a -> 'accum) ->
    'accum
```

`map` and `fold` are extremely common combinators that transform an input list by applying a function to each value of the list. The `map` combinator is simplest, with the resulting list being output directly. `fold` applies each value in the input list to a function that accumulates a single result, and returns that instead:

```
val iter : 'a list -> f:('a -> unit) -> unit
```

`iter` is a more specialized combinator that is only useful when writing imperative code. The input function is applied to every value, but no result is supplied. The function must instead apply some side effect such as changing a mutable record field or printing to the standard output.

`Yojson` provides several combinators in the `Yojson.Basic.Util` module to manipulate values:

- `val member : string -> json -> json` selects a named field from a JSON record.
- `val to_string : json -> string` converts a JSON value into an OCaml `string`. It raises an exception if this is impossible.
- `val to_int : json -> int` converts a JSON value into an `int`. It raises an exception if this is impossible.
- `val filter_string : json list -> string list` filters valid strings from a list of JSON fields, and return them as an OCaml `string list`.

We'll go through each of these uses one by one now. The following examples also use

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

the |> pipe-forward operator that we explained in Chapter 3 (Variables and Functions). This lets us chain together multiple JSON selection functions and feed the output from one into the next one, without having to create separate `let` bindings for each one.

Let's start with selecting a single `title` field from the record:

```
# open Yojson.Basic.Util;;
# let title = json |> member "title" |> to_string;;
val title : string = "Real World OCaml"
```

The `member` function accepts a JSON object and named key and returns the JSON field associated with that key, or `Null`. Since we know that the `title` value is always a string in our example schema, we want to convert it to an OCaml string. The `to_string` function performs this conversion and raises an exception if there is an unexpected JSON type. The |> operator provides a convenient way to chain these operations together:

```
# let tags = json |> member "tags" |> to_list |> filter_string;;
val tags : string list = ["functional programming"; "ocaml";
    "algorithms"]
# let pages = json |> member "pages" |> to_int;;
val pages : int = 450
```

The `tags` field is similar to `title`, but the field is a list of strings instead of a single one. Converting this to an OCaml `string list` is a two-stage process. First, we convert the JSON `List` to an OCaml list of JSON values and then filter out the `String` values as an OCaml `string list`. Remember that OCaml lists must contain values of the same type, so any JSON values that cannot be converted to a `string` will be skipped from the output of `filter_string`:

```
# let is_online = json |> member "is_online" |> to_bool_option;;
val is_online : bool option = Some true
# let is_translated = json |> member "is_translated" |>
    to_bool_option;;
val is_translated : bool option = None
```

The `is_online` and `is_translated` fields are optional in our JSON schema, so no error should be raised if they are not present. The OCaml type is a `bool option` to reflect this and can be extracted via `to_bool_option`. In our example JSON, only `is_online` is present and `is_translated` will be `None`:

```
# let authors = json |> member "authors" |> to_list;;
val authors : Yojson.Basic.t list =
  [`Assoc
     [("name", `String "Jason Hickey"); ("affiliation", `String
      "Google")];
   `Assoc
     [("name", `String "Anil Madhavapeddy");
      ("affiliation", `String "Cambridge")];
   `Assoc
     [("name", `String "Yaron Minsky");
      ("affiliation", `String "Jane Street")]]
```

The final use of JSON combinators is to extract all the `name` fields from the list of authors. We first construct the `author list`, and then `map` it into a `string list`. Notice

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

that the example explicitly binds `authors` to a variable name. It can also be written more succinctly using the pipe-forward operator:

```
# let names =
    json |> member "authors" |> to_list
  |> List.map ~f:(fun json -> member "name" json |> to_string);;
val names : string list =
  ["Jason Hickey"; "Anil Madhavapeddy"; "Yaron Minsky"]
```

This style of programming, which omits variable names and chains functions together, is known as *point-free programming*. It's a succinct style but shouldn't be overused due to the increased difficulty of debugging intermediate values. If an explicit name is assigned to each stage of the transformations, debuggers in particular have an easier time making the program flow simpler to represent to the programmer.

This technique of using statically typed parsing functions is very powerful in combination with the OCaml type system. Many errors that don't make sense at runtime (for example, mixing up lists and objects) will be caught statically via a type error.

19.4 Constructing JSON Values

Building and printing JSON values is pretty straightforward given the `Yojson.Basic.t` type. You can just construct values of type `t` and call the `to_string` function on them. Let's remind ourselves of the `Yojson.Basic.t` type again:

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

We can directly build a JSON value against this type and use the pretty-printing functions in the `Yojson.Basic` module to display JSON output:

```
# let person = `Assoc [ ("name", `String "Anil") ];;
val person : [> `Assoc of (string * [> `String of string ]) list ] =
  `Assoc [("name", `String "Anil")]
```

In the preceding example, we've constructed a simple JSON object that represents a single person. We haven't actually defined the type of `person` explicitly, as we're relying on the magic of polymorphic variants to make this all work.

The OCaml type system infers a type for `person` based on how you construct its value. In this case, only the `Assoc` and `String` variants are used to define the record, and so the inferred type only contains these fields without knowledge of the other possible allowed variants in JSON records that you haven't used yet (e.g. `Int` or `Null`):

```
# Yojson.Basic.pretty_to_string;;
- : ?std:bool -> Yojson.Basic.t -> string = <fun>
```

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

The `pretty_to_string` function has a more explicit signature that requires an argument of type `Yojson.Basic.t`. When `person` is applied to `pretty_to_string`, the inferred type of `person` is statically checked against the structure of the `json` type to ensure that they're compatible:

```
# Yojson.Basic.pretty_to_string person;;
- : string = "{ \"name\": \"Anil\" }"
# Yojson.Basic.pretty_to_channel stdout person;;
{ "name": "Anil" }
- : unit = ()
```

In this case, there are no problems. Our `person` value has an inferred type that is a valid subtype of `json`, and so the conversion to a string just works without us ever having to explicitly specify a type for `person`. Type inference lets you write more succinct code without sacrificing runtime reliability, as all the uses of polymorphic variants are still checked at compile time.

Polymorphic Variants and Easier Type Checking

One difficulty you will encounter is that type errors involving polymorphic variants can be quite verbose. For example, suppose you build an `Assoc` and mistakenly include a single value instead of a list of keys:

```
# let person = `Assoc ("name", `String "Anil");;
val person : [> `Assoc of string * [> `String of string ] ] =
  `Assoc ("name", `String "Anil")
# Yojson.Basic.pretty_to_string person;;
Line 1, characters 31-37:
Error: This expression has type
         [> `Assoc of string * [> `String of string ] ]
       but an expression was expected of type Yojson.Basic.t
       Types for tag `Assoc are incompatible
```

The type error is more verbose than it needs to be, which can be inconvenient to wade through for larger values. You can help the compiler to narrow down this error to a shorter form by adding explicit type annotations as a hint about your intentions:

```
# let (person : Yojson.Basic.t) =
  `Assoc ("name", `String "Anil");;
Line 2, characters 10-34:
Error: This expression has type 'a * 'b
       but an expression was expected of type (string *
    Yojson.Basic.t) list
```

We've annotated `person` as being of type `Yojson.Basic.t`, and as a result, the compiler spots that the argument to the `Assoc` variant has the incorrect type. This illustrates the strengths and weaknesses of polymorphic variants: they're lightweight and flexible, but the error messages can be quite confusing. However, a bit of careful manual type annotation makes tracking down such issues much easier.

We'll discuss more techniques like this that help you interpret type errors more easily in Chapter 26 (The Compiler Frontend: Parsing and Type Checking).

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

19.5 Using Nonstandard JSON Extensions

The standard JSON types are *really* basic, and OCaml types are far more expressive. Yojson supports an extended JSON format for those times when you're not interoperating with external systems and just want a convenient human-readable, local format. The `Yojson.Safe.json` type is a superset of the `Basic` polymorphic variant and looks like this:

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Floatlit of string
  | `Int of int
  | `Intlit of string
  | `List of json list
  | `Null
  | `String of string
  | `Stringlit of string
  | `Tuple of json list
  | `Variant of string * json option
]
```

The `Safe.json` type includes all of the variants from `Basic.json` and extends it with a few more useful ones. A standard JSON type such as a `String` will type-check against both the `Basic` module and also the nonstandard `Safe` module. If you use the extended values with the `Basic` module, however, the compiler will reject your code until you make it compliant with the portable subset of JSON.

Yojson supports the following JSON extensions:

The `lit` suffix Denotes that the value is stored as a JSON string. For example, a `Floatlit` will be stored as `"1.234"` instead of `1.234`.

The `Tuple` type Stored as `("abc", 123)` instead of a list.

The `Variant` type Encodes OCaml variants more explicitly, as `<"Foo">` or `<"Bar":123>` for a variant with parameters.

The only purpose of these extensions is to have greater control over how OCaml values are represented in JSON (for instance, storing a floating-point number as a JSON string). The output still obeys the same standard format that can be easily exchanged with other languages.

You can convert a `Safe.json` to a `Basic.json` type by using the `to_basic` function as follows:

```
val to_basic : json -> Yojson.Basic.t
(** Tuples are converted to JSON arrays, Variants are converted to
    JSON strings or arrays of a string (constructor) and a json value
    (argument). Long integers are converted to JSON strings.
    Examples:

    `Tuple [ `Int 1; `Float 2.3 ]   ->    `List [ `Int 1; `Float 2.3 ]
    `Variant ("A", None)            ->    `String "A"
    `Variant ("B", Some x)          ->    `List [ `String "B", x ]
```

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
    `Intlit "12345678901234567890"  ->    `String
    "12345678901234567890"
 *)
```

19.6 Automatically Mapping JSON to OCaml Types

The combinators described previously make it easy to write functions that extract fields from JSON records, but the process is still pretty manual. When you implement larger specifications, it's much easier to generate the mappings from JSON schemas to OCaml values more mechanically than writing conversion functions individually.

We'll cover an alternative JSON processing method that is better for larger-scale JSON handling now, using ATD[2], which provides a *domain specific language*, or DSL, that compiles JSON specifications into OCaml modules, which are then used throughout your application.

You can install the `atdgen` executable by calling `opam install atdgen`.

```
$ opam install atdgen
$ atdgen -version
2.2.1
```

You may need to run `eval $(opam env)` in your shell if you don't find `atdgen` in your path.

19.6.1 ATD Basics

The idea behind ATD is to specify the format of the JSON in a separate file and then run a compiler (`atdgen`) that outputs OCaml code to construct and parse JSON values. This means that you don't need to write any OCaml parsing code at all, as it will all be autogenerated for you.

Let's go straight into looking at an example of how this works, by using a small portion of the GitHub API. GitHub is a popular code hosting and sharing website that provides a JSON-based web API[3]. The following ATD code fragment describes the GitHub authorization API (which is based on a pseudostandard web protocol known as OAuth):

```
type scope = [
    User <json name="user">
  | Public_repo <json name="public_repo">
  | Repo <json name="repo">
  | Repo_status <json name="repo_status">
  | Delete_repo <json name="delete_repo">
  | Gist <json name="gist">
]

type app = {
```

[2] https://github.com/ahrefs/atd
[3] http://developer.github.com

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
  name: string;
  url: string;
}  <ocaml field_prefix="app_">

type authorization_request = {
  scopes: scope list;
  note: string;
} <ocaml field_prefix="auth_req_">

type authorization_response = {
  scopes: scope list;
  token: string;
  app: app;
  url: string;
  id: int;
  ?note: string option;
  ?note_url: string option;
}
```

The ATD specification syntax is deliberately quite similar to OCaml type definitions. Every JSON record is assigned a type name (e.g., `app` in the preceding example). You can also define variants that are similar to OCaml's variant types (e.g., `scope` in the example).

19.6.2 ATD Annotations

ATD does deviate from OCaml syntax due to its support for annotations within the specification. The annotations can customize the code that is generated for a particular target (of which the OCaml backend is of most interest to us).

For example, the preceding GitHub `scope` field is defined as a variant type, with each option starting with an uppercase letter as is conventional for OCaml variants. However, the JSON values that come back from GitHub are actually lowercase and so aren't exactly the same as the option name.

The annotation `<json name="user">` signals that the JSON value of the field is `user`, but that the variable name of the parsed variant in OCaml should be `User`. These annotations are often useful to map JSON values to reserved keywords in OCaml (e.g., `type`).

19.6.3 Compiling ATD Specifications to OCaml

The ATD specification we defined can be compiled to OCaml code using the `atdgen` command-line tool. Let's run the compiler twice to generate some OCaml type definitions and a JSON serializing module that converts between input data and those type definitions.

The `atdgen` command will generate some new files in your current directory. `github_t.ml` and `github_t.mli` will contain an OCaml module with types defined that correspond to the ATD file:

```
$ atdgen -t github.atd
```

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
$ atdgen -j github.atd
$ ocamlfind ocamlc -package atd -i github_t.mli
type scope =
    [ `Delete_repo | `Gist | `Public_repo | `Repo | `Repo_status |
    `User ]
type app = { app_name : string; app_url : string; }
type authorization_response = {
  scopes : scope list;
  token : string;
  app : app;
  url : string;
  id : int;
  note : string option;
  note_url : string option;
}
type authorization_request = {
  auth_req_scopes : scope list;
  auth_req_note : string;
}
```

There is an obvious correspondence to the ATD definition. Note that field names in OCaml records in the same module cannot shadow one another, and so we instruct ATDgen to prefix every field with a name that distinguishes it from other records in the same module. For example, `<ocaml field_prefix="auth_req_">` in the ATD spec prefixes every field name in the generated `authorization_request` record with `auth_req`.

The `Github_t` module only contains the type definitions, while `Github_j` provides serialization functions to and from JSON. You can read the `github_j.mli` to see the full interface, but the important functions for most uses are the conversion functions to and from a string. For our preceding example, this looks like:

```
val string_of_authorization_request :
  ?len:int -> authorization_request -> string
  (** Serialize a value of type {!authorization_request}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val string_of_authorization_response :
  ?len:int -> authorization_response -> string
  (** Serialize a value of type {!authorization_response}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)
```

This is pretty convenient! We've now written a single ATD file, and all the OCaml boilerplate to convert between JSON and a strongly typed record has been generated for us. You can control various aspects of the serializer by passing flags to `atdgen`. The important ones for JSON are:

`-j-std` Converts tuples and variants into standard JSON and refuses to print NaN and

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

infinities. You should specify this if you intend to interoperate with services that aren't using ATD.

-j-custom-fields FUNCTION Calls a custom function for every unknown field encountered, instead of raising a parsing exception.

-j-defaults Always explicitly outputs a JSON value if possible. This requires the default value for that field to be defined in the ATD specification.

The full ATD specification[4] is quite sophisticated and documented online. The ATD compiler can also target formats other than JSON and outputs code for other languages (such as Java) if you need more interoperability.

There are also several similar projects that automate the code generation process. Piqi[5] supports conversions between XML, JSON, and the Google protobuf format; and Thrift[6] supports many other programming languages and includes OCaml bindings.

19.6.4 Example: Querying GitHub Organization Information

Let's finish up with an example of some live JSON parsing from GitHub and build a tool to query organization information via their API. Start by looking at the online API documentation[7] for GitHub to see what the JSON schema for retrieving the organization information looks like.

Now create an ATD file that covers the fields we need. Any extra fields present in the response will be ignored by the ATD parser, so we don't need a completely exhaustive specification of every field that GitHub might send back:

```
type org = {
  login: string;
  id: int;
  url: string;
  ?name: string option;
  ?blog: string option;
  ?email: string option;
  public_repos: int
}
```

Let's build the OCaml type declaration first by calling `atdgen -t` on the specification file:

```
$ dune build github_org_t.mli
$ cat _build/default/github_org_t.mli
(* Auto-generated from "github_org.atd" *)
              [@@@ocaml.warning "-27-32-35-39"]

type org = {
  login: string;
  id: int;
  url: string;
```

[4] https://atd.readthedocs.io/en/latest/
[5] http://piqi.org
[6] http://thrift.apache.org
[7] http://developer.github.com/v3/orgs/

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

```
  name: string option;
  blog: string option;
  email: string option;
  public_repos: int
}
```

The OCaml type has an obvious mapping to the ATD spec, but we still need the logic to convert JSON buffers to and from this type. Calling `atdgen -j` will generate this serialization code for us in a new file called `github_org_j.ml`:

```
$ dune build github_org_j.mli
$ cat _build/default/github_org_j.mli
(* Auto-generated from "github_org.atd" *)
[@@@ocaml.warning "-27-32-35-39"]

type org = Github_org_t.org = {
  login: string;
  id: int;
  url: string;
  name: string option;
  blog: string option;
  email: string option;
  public_repos: int
}

val write_org :
  Bi_outbuf.t -> org -> unit
  (** Output a JSON value of type {!org}. *)

val string_of_org :
  ?len:int -> org -> string
  (** Serialize a value of type {!org}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val read_org :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> org
  (** Input JSON data of type {!org}. *)

val org_of_string :
  string -> org
  (** Deserialize JSON data of type {!org}. *)
```

The `Github_org_j` serializer interface contains everything we need to map to and from the OCaml types and JSON. The easiest way to use this interface is by using the `string_of_org` and `org_of_string` functions, but there are also more advanced low-level buffer functions available if you need higher performance (but we won't go into that in this tutorial).

All we need to complete our example is an OCaml program that fetches the JSON and uses these modules to output a one-line summary. Our following example does just that.

The following code calls the cURL command-line utility by using the `Shell` interface

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

to run an external command and capture its output. You'll need to ensure that you have cURL installed on your system before running the example. You might also need to `opam install shell` if you haven't installed it previously:

```
open Core

let print_org file () =
  let url = sprintf "https://api.github.com/orgs/%s" file in
  Shell.run_full "curl" [url]
  |> Github_org_j.org_of_string
  |> fun org ->
  let open Github_org_t in
  let name = Option.value ~default:"???" org.name in
  printf "%s (%d) with %d public repos\n"
    name org.id org.public_repos

let () =
  Command.basic_spec ~summary:"Print Github organization information"
    Command.Spec.(empty +> anon ("organization" %: string))
    print_org
  |> Command.run
```

The following is a short shell script that generates all of the OCaml code and also builds the final executable:

```
(rule
  (targets github_org_j.ml github_org_j.mli)
  (deps    github_org.atd)
  (mode    fallback)
  (action  (run atdgen -j %{deps})))

(rule
  (targets github_org_t.ml github_org_t.mli)
  (deps    github_org.atd)
  (mode    fallback)
  (action  (run atdgen -t %{deps})))

(executable
  (name      github_org_info)
  (libraries core yojson atdgen shell)
  (flags     :standard -w -32)
  (modules   github_org_info github_org_t github_org_j))
```

```
$ dune build github_org_info.exe
```

You can now run the command-line tool with a single argument to specify the name of the organization, and it will dynamically fetch the JSON from the web, parse it, and render the summary to your console:

```
$ dune exec -- ./github_org_info.exe mirage
MirageOS (131943) with 125 public repos
$ dune exec -- ./github_org_info.exe janestreet
??? (3384712) with 145 public repos
```

The JSON returned from the `janestreet` query is missing an organization name, but this is explicitly reflected in the OCaml type, since the ATD spec marked `name`

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press

as an optional field. Our OCaml code explicitly handles this case and doesn't have to worry about null-pointer exceptions. Similarly, the JSON integer for the `id` is mapped into a native OCaml integer via the ATD conversion.

While this tool is obviously quite simple, the ability to specify optional and default fields is very powerful. Take a look at the full ATD specification for the GitHub API in the `ocaml-github`[8] repository online, which has lots of quirks typical in real-world web APIs.

Our example shells out to `curl` on the command line to obtain the JSON, which is rather inefficient. You could integrate an Async-based HTTP fetch directly into your OCaml application, as described in Chapter 17 (Concurrent Programming with Async).

[8] http://github.com/avsm/ocaml-github

https://doi.org/10.1017/9781009129220.022 Published online by Cambridge University Press