FREE eBook

LEARNING OCaml

Free unaffiliated eBook created from
Stack Overflow contributors.

#ocaml

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ocaml

It is an unofficial and free OCaml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official OCaml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with OCaml

Remarks

This section provides an overview of what ocaml is, and why a developer might want to use it.

It should also mention any large subjects within ocaml, and link out to the related topics. Since the Documentation for ocaml is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Installing OPAM

OPAM is a package manager for OCaml. It builds and manages compiler versions and OCaml libraries for you easily.

The easiest way to install OPAM on your operating system is to use a package manager for your system. e.g apt-get, yum or homebrew.

Mac OSX Installation Instructions

Update homebrew formulae and install OPAM.

```
brew update
brew install opam
```

Ubuntu Installation Instructions

```
add-apt-repository ppa:avsm/ppa
apt-get update
apt-get install ocaml opam
```

Compiling from source

```
wget http://caml.inria.fr/pub/distrib/ocaml-4.03/ocaml-4.03.0.tar.gz
tar xf ocaml-4.03.0.tar.gz
cd ocaml-4.03.0
./configure -prefix installation_path
make world.opt
make install
```

Initializing OPAM

Once you have OPAM installed, run `opam init` and follow the instructions.

Once done you should be able to run the OCaml interpreter from your shell.

```
$ ocaml
        OCaml version 4.03.0

#
```

Your first program in OCaml

Now that the OCaml distribution is available on your favorite operating system, we can create your first program in OCaml: the Hello World!

We have different ways to launch an OCaml program.

The REPL (toplevel)

You can execute your code *interactively* with the *toplevel*. With the OCaml *toplevel*, you can write and execute OCaml code, as a UNIX shell. Afterwards, the *toplevel* checks the type of your code immediately. So, you can quickly and easily test some parts of code without compilation and execution.

You can launch the *toplevel* with the `ocaml` command. Then, you can write an OCaml *sentence* ended by `;;` which is evaluated immediately. The *toplevel* displays the type and the value of your expression just after:

```
# "Hello Worlds!";;
- : string = "Hello Worlds!"
```

It is also possible to launch the *toplevel* on your file. You can see this explanation about that.

To facilitate your input in the *toplevel*, you can use a tool like `ledit` or `rlwrap` which provides some features (like input history):

```
 $ ledit ocaml

 $ rlwrap ocaml
```

Compilation to the *bytecode*

We have two different compilers, one which compiles to *bytecode* and the other which compiles to native code. The first is the same as the *bytecode* of the Java's virtual machine. So, the *bytecode* is less efficient but more portable.

We have some extensions files used by the OCaml compilers:

extension	definition
`.ml`	The source code (as `.c` in C)
`.mli`	The interface (as `.h` in C)
`.cmo`	Source code compiled by `ocamlc` in *bytecode*
`.cmi`	Interface code compiled by `ocamlc`
`.cmx` and `.o`	Source code compiled by `ocamlopt` in native code
`.cma`	Library (bucket of some `*.cmo`) in *bytecode*
`.cmxa` and `.a`	Library in native code
`.cmxs`	Library in native code (to load dynamicaly)

The *bytecode* compiler is `ocamlc`.

You have different common options:

- `-c`: to compile a source file without the linkage process (to produce an executable). So, the command `ocaml -c foo.ml` produces a `.cmo` file. Unlike C in which the header file does not need to be compiled, it's necessary in OCaml to compile the `.mli` file: `ocaml -c foo.mli`.

You need to compile the interface first. When you compile the source file afterwards, OCaml tries to check that the implementation matches the interface.

The `.mli` file is not a mandatory. If you compile a `.ml` file without a `.mli` file, OCaml will produce a `.cmi` file automatically.

- `-o`: to compile some `.cmo` files to an executable. For example: `ocamlc -o program foo.cmo bar.cmo`. These files need to be arranged by the dependencies for which the first file has no dependence.

- `-I`: to indicate an other directory where the compiler can find the necessary files for the compilation (like the interface or source code). It's the same than the `-I` from a C compiler.

We have many other options. You can see the manual for more information.

So, you can write the `hello.ml` now, and compile this file with `ocamlc -o hello hello.ml` to produce a *bytecode* program:

```
let () = print_endline "Hello World!"
```

The `let () = ...` is the first entry of your program (like the `main` in C). After, we use the function `print_endline` (provided by the standard library) with the argument `"Hello World!"` to print `Hello`

`Worlds` with a newline in the standard output.

After the compilation, you have the `.cmo` file and the `.cmi` file automatically produced by the compiler and your program `hello`. You can open your program, and in the top of this file, you can see:

```
#!/usr/local/bin/ocamlrun
```

That means your program need the `ocamlrun` program (provided by the distribution) to execute the *bytecode* (like the JVM).

Compilation to the native code

We have an another compiler that produces native code. The compiler is: `ocamlopt`. However, the resultant executable can't work on most other architectures.

`ocamlopt` uses the same options as `ocamlc` so you can execute `ocamlopt -o hello hello.ml`. After, you can see a `.cmx` and a `.o` file.

Finally, from your *bytecode*/native code program, you can execute:

```
$ ./hello
Hello World!
$
```

Installation on Windows (native)

Premise

These instruction shows a procedure to install native OCaml binaries in Windows. If your operative system is `Windows 10 (Insider Preview) build 14316` or later you can also install OCaml through Bash on Ubuntu on Windows. In this case, follow the instruction to install OCaml on Ubuntu.

Install OCaml and Opam

Download OCaml official distribution. It contains both OCaml compilers and Opam packet manager. Suppose you have installed the software in `C:/OCaml`. To be sure you've correcly installed OCaml open `cmd.exe` and type `ocaml`.

If you see the message `'ocaml' is not recognized as an internal or external command, operable program or batch file` you need to add `C:/OCaml/bin` to your Path (Environment Variable).

Add OCaml binaries to path

in `Control Panel > System and Security > System > Advanced system settings (on the left) >`

`Environment Variables` and then select `Path` in `System Variable` tab, then `Edit`.

Add `C:/OCaml/bin;` to the list.

Install Cygwin

Without Cygwin you can't use Opam. In fact, if you try to open Opam typing `opam` in `cmd.exe` it shows a message: `Fatal error: exception Unix.Unix_error(20, "create_process", "cygcheck").`

Download Cygwin and start the installer. Be sure to check the following packages:

- automake
- diffutils
- libreadline
- make
- m4
- mingw64-x86_64-gcc-core
- mingw64-x86_64-gmp
- mingw64-x86_64-openssl
- mingw64-x86_64-pkg-config
- mingw64-x86_64-sqlite3
- patch
- rlwrap
- unzip
- wget

Suppose you have installed the software in `C:/cygwin` (`C:/cygwin64` for 64bit version). Open `cmd` and type `wget` (or one of the executable present in `C:/cygwin/bin`) to check if you can use the Cygwin executables. If the executable won't open, add `C:/cygwin/bin` to your Path (Environment Variable).

Configure Opam

Open `cmd.exe` and type `opam init` to configure Opam.

Then install `ocamlfind` (part of the OCaml compiler) with

```
opam install ocamlfind
opam config env
```

Check if `ocamlfind` is installed typing it in `cmd.exe`.

The command `opam config env` is used to add `opam`'s executables directory to the enviroment path. If after logout you cannot reach `ocamlfind` anymore, you can manually add it adding to path the following line: `C:/Users/<your user>/Documents/.opam/system/bin/`.

Installing packages

Packages are installed through Opam with the command `opam install xyz` where `xyz` is the name of the package.

Install UTop

Try running the command `opam install utop`. If you have no errors, then typing `utop` will open the executable.

If you see the message

```
[ERROR] The compilation of zed failed at "ocaml setup.ml -build".
```

you have to manually install the single packages. Try again typing:

```
opam install zed
opam install lambda-term
opam install utop
```

Both `lambda-term` and `utop` might not install. See Troubleshoot section.

Installing Core

You can install `core` package with `opam install core`. On Windows 64bit version (and 64bit Cygwin) you will see the following error:

```
[ERROR] core is not available because your system doesn't comply with os != "win32" & ocaml-
version = "4.02.3".
```

Troubleshoot: cannot create regular file

If package with name `xyz.10.1` fails to install (where xyz is the name of the package, and 10.1 its version) with the following message:

```
install: cannot create regular file '/cygdrive/c/Users/<your
user>/Documents/.opam/system/bin/<something>': File exists
```

You have to go in this directory:

```
C:\Users\<your user>\Documents\.opam\repo\default\packages\<xyz>\<xyz.10.1>\files
```

and delete the file `xyz.10.1.install`.

Troubleshoot: cannot load shared library

If you try to open some Opam's package (eg: `utop`) and you see this error:

```
Fatal error: cannot load shared library dlllwt-unix_stubs
Reason: The specified module could not be found.
```

Run `opam config env` again and try to reopen the executable.

Read Getting started with OCaml online: https://riptutorial.com/ocaml/topic/1826/getting-started-with-ocaml

Chapter 2: Common Pitfalls

Examples

Using the wrong operator

In OCaml, there are different arithmetic operators for floats and integers. Additionally, these operators can only be used on 2 floats or 2 integers. Here are invalid expressions in OCaml

```
1.0 + 2.0
1 + 2.0
1 +. 2
1 +. 2.0
```

The correct expression for each of these respectively are

```
1. +. 2.
float_of_int 1 +. 2.
1 + 2
float_of_int 1 +. 2.
```

There is no automatic casting of integers to floats or vice-versa in OCaml. Everything is explicit. Here is a list of the integer and float operators

Operation	Integer Operator	Float Operator
Addition	`a + b`	`c +. d`
Subtraction	`a - b`	`c -. d`
Multiplication	`a * b`	`c *. c`
Division	`a / b`	`c /. d`
Modulus	`a mod b`	`modfloat c d`
Exponentiation	N/a	`c ** d`

Where `a` and `b` are integers and `c` and `d` are floats.

Forgetting parentheses around function arguments

A common mistake is to forget surrounding compound function arguments with parentheses, leading to type errors.

```
# string_of_int 1+1;;
```

```
Error: This expression has type string but an expression was expected of type int
```

This is because of the precedence. In fact, the above evaluates to

```
# (string_of_int 1) + 1;;
```

which is wrong. A correct syntax would be

```
# string_of_int (1+1);;

- : string = "2"
```

Read Common Pitfalls online: https://riptutorial.com/ocaml/topic/8146/common-pitfalls

Chapter 3: Functions

Examples

Defining a Function with a let Binding

Values can be given names using `let`:

```
# let a = 1;;
val a : int = 1
```

You can use similar syntax to define a function. Just provide additional parameters for the arguments.

```
# let add arg1 arg2 = arg1 + arg2;;
val add : int -> int -> int = <fun>
```

We can call it like this:

```
# add 1 2;;
- : int = 3
```

We can pass values in directly like that, or we can pass values bound to names:

```
# add a 2;;
- : int = 3
```

The line that the interpreter gives us after we define something is the value of the object with its type signature. When we gave it a simple value bound to `a`, it came back with:

```
val a : int = 1
```

Which means `a` is an `int`, and its value is `1`.

The type signature of our function is a little more complicated:

```
val add : int -> int -> int = <fun>
```

The type signature of `add` looks like a bunch of ints and arrows. This is because a function that takes two arguments is actually a function which just takes one argument, but returns another function that takes the next argument. You could instead read it like this:

```
val add : int -> (int -> int) = <fun>
```

This is useful when we want to create different sorts of functions on the fly. For example, a function that adds 5 to everything:

```
# let add_five = add 5;;
val add_five : int -> int = <fun>
# add_five 5;;
- : int = 10
# add_five 10;;
- : int = 15
```

Using the function keyword

The `function` keyword automatically has pattern matching when you define the body of your function. Observe it below:

```
# let foo = function
0 -> "zero"
| 1 -> "one"
| 2 -> "couple"
| 3 -> "few"
| _ -> "many";;
val foo : int -> bytes = <fun>

# foo 0;;
- : bytes = "zero"

# foo 3;;
- : bytes = "few"

# foo 10;;
- : bytes = "many"

# let bar = function
"a" | "i" | "e" | "o" | "u" -> "vowel"
| _ -> "consonant";;
val bar : bytes -> bytes = <fun>

# bar "a";;
- : bytes = "vowel"

# bar "k";;
- : bytes = "consonant"
```

Anonymous functions

Since functions are ordinary values, there is a convenient syntax for creating functions without names:

```
List.map (fun x -> x * x) [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

This is handy, as we would otherwise have to name the function first (see let) to be able to use it:

```
let square x = x * x
(* val square : int -> int = <fun> *)

List.map square [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

Recursive and Mutually Recursive Functions

You can define a function to be recursive with the `rec` keyword, so it can call itself.

```
# let rec fact n = match n with
    | 0 -> 1
    | n -> n * fact (n - 1);;

val fact : int -> int = <fun>

# fact 0;;
- : int = 1
# fact 4;;
- : int = 24
```

You can also define mutually recursive functions with the `and` keyword, so they can call each other.

```
# let rec first x = match x with
    | 1 -> 1
    | x -> second (x mod 10)

  and second x = first (x + 1);;

val first : int -> int = <fun>
val second : int -> int = <fun>

# first 20;;
- : int = 1
# first 12345;;
- : int = 1
```

Notice that the second function does not have the `req` keyword.

Read Functions online: https://riptutorial.com/ocaml/topic/2793/functions

Chapter 4: Higher Order Functions

Syntax

- `val (|>) : 'a -> ('a -> 'b) -> 'b`
- `val (@@) : ('a -> 'b) -> 'a -> 'b`

Examples

Generic algorithms

Higher-order functions can be used to implement generic algorithms, giving up the responsibility of providing final details to the user. For instance `List.sort` expects a comparison function, which allows to implement various ways of sorting. Here we implement case-insensitive sorting of strings:

```
let string_case_insensitive_sort lst =
  let case_insensitive_compare a b =
    String.compare (String.lowercase a) (String.lowercase b)
  in
  List.sort case_insensitive_compare lst
```

There is a rich list of higher-order functions in the standard library, especially in the *List* module, see `List.fold_left` and `List.sort` for instance. More advanced examples can be found in third-party libraries. A good example is the simulated annealing implemented in **ocaml-gsl**. Simulated annealing is a generic optimisation procedure which is parametrised by a function used to explore the set of states of the problem and an error function (called here energy function).

Users familiar with C++ can compare this to the *Strategy* pattern.

Dispose system resources even when an exception is raised

Higher-order functions can be used to ensure that system resources are disposed, even when a treatment raises an exception. The pattern used by `with_output_file` allows a clean separation of concerns: the higher-order `with_output_file` functions takes care of managing the system resources bound to file manipulation while the treatment `f` only consumes the output channel.

```
let with_output_file path f =
  let c = open_out path in
  try
    let answer = f c in
    (close_out c; answer)
  with exn -> (close_out c; raise exn)
```

Let us use this higher-order function to implement a function writing a string to a file:

```
let save_string path s =
```

```
  (with_output_file path) (fun c -> output_string c s)
```

Using more advanced functions than `fun c -> output_string c s` it is possible to save more complex values. See for instance the *Marshal* module in the standard library or the *Yojson* library by Martin Jambon.

Composition operators

Two useful higher-order functions are the binary *application* (`@@`) and *reverse-application* or "pipe" (`|>`) operators. Although since 4.01 they're available as primitives, it might still be instructive to define them here:

```
let (|>) x f = f x
let (@@) f x = f x
```

Consider the problem of incrementing the square of 3. One way of expressing that computation is this:

```
(* 1 -- Using parentheses *)
succ (square 3)
(* - : int = 10 *)

(* where `square` is defined as: *)
let square x = x * x
```

Note that we couldn't simply do `succ square 3` because (due to left-associativity) that would reduce to the meaningless `(succ square) 3`. Using application (`@@`) we can express that without the parentheses:

```
(* 2 -- Using the application operator *)
succ @@ square 3
(* - : int = 10 *)
```

Notice how the last operation to be performed (namely `succ`) occurs first in the expression? The *reverse-application* operator (`|>`) allows us to, well, reverse this:

```
(* 3 -- Using the reverse-application operator *)
3 |> square |> succ
(* - : int = 10 *)
```

The number 3 is now "piped" through `square` and then `succ`, as opposed to being applied to `square` to yield a result that `succ` is applied to.

Read Higher Order Functions online: https://riptutorial.com/ocaml/topic/2729/higher-order-functions

Chapter 5: List Processing

Examples

List.Map

`List.map` has the signature `('a -> 'b) -> 'a list -> 'b list` which in English is a function that takes a function (we'll call this the mapping function) from one type (namely `'a`) to another type (namely `'b`) and a list of the first type. The function returns a list of the second type where every element is the result of calling the mapping function on an element of the first list.

```
List.map string_of_int [ 1; 2; 3; 4 ]
#- [ "1"; "2"; "3"; "4" ] : string list
```

The types `'a` and `'b` don't have to be different. For example, we can map numbers to their squares just as easily.

```
let square x = x * x in
List.map square [ 1; 2; 3; 4 ]
#- [ 1; 4; 9; 16 ] : int list
```

Aggregate data in a list

The `List.fold_left` and `List.fold_right` functions are higher-order functions that implement the outer logic of list aggregation. Aggregating a list, sometimes also referred to as reducing a list, means computing a value derived from the sequential inspection of all items in that list.

The documentation of the List module states that

- `List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
- `List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. (This latter function is not tail-recursive.)

In plain English computing `List.fold_left f a [b1; ...; bn]` amounts to running through the list `[b1; ...; bn]` keeping track of an *accumulator* initially set to `a`: each time we see an item in the list, we use `f` to update the value of the accumulator, and when we are done, the accumulator is the final value of our computation. The `List.fold_right` function is similar.

Here are a few practical examples:

Compute the total sum of a list of numbers

```
List.fold_left ( + ) 0 lst
```

Compute the average of a list of floats

```
let average lst =
  let (sum, n) =
    List.fold_left (fun (sum, n) x -> (sum +. x, n + 1)) (0.0, 0) lst
  in
  sum /. (float_of_int n)
```

Re-implement basic list processing

The functions `List.fold_left` and `List.fold_right` are so general that they can be used to implement almost every other functions from the list module:

```
let list_length lst = (* Alternative implementation to List.length *)
  List.fold_left ( + ) 0 lst

let list_filter predicate lst = (* Alternative implementation to List.filter *)
  List.fold_right (fun a b -> if predicate a then a :: b else b) lst []
```

It is even possible to reimplement the `List.iter` function, remember that `()` is the global state of the program to interpret this code as a further example of *list aggregation*:

```
let list_iter f lst = (* Alternation implementation to List.iter *)
  List.fold_left (fun () b -> f b) () lst
```

These examples are meant to be learning material, these implementations have no virtue over the corresponding functions from the standard library.

Read List Processing online: https://riptutorial.com/ocaml/topic/2730/list-processing

Chapter 6: Mutable record fields

Introduction

Like most OCaml values, records are immutable by default. However, since OCaml also handles imperative programming, it provides a way to make individual fields *mutable*. Mutable fields can be modified in-place by assignment, rather than having to resort to usual functional techniques, such as functional update.

While introducing side-effects, mutable fields can result in an improved performance when used correctly.

Examples

Declaring a record with mutable fields

In the following, `weight` is declared as a mutable field.

```
type person = {
  name: string;
  mutable weight: int
};;
```

Remark: As far as design is concerned here, one would consider the fact that a `person`'s name isn't likely to change, but their weight is.

Initializing a record with mutable fields

Initializing a record with mutable fields isn't different from a regular record initialization.

```
let john = { name = "John"; weight = 115 };;
```

Setting the value to a mutable field

To assign a new value to a mutable record field, use the `<-` operator.

```
john.weight <- 120;;
```

Note: The previous expression has a `unit` type.

Read Mutable record fields online: https://riptutorial.com/ocaml/topic/9367/mutable-record-fields

Chapter 7: Ocamlbuild

Examples

Project depending on external libraries

If your project depends on the external libraries, you should first install them with opam. Assuming your dependencies are `foo` and `bar` and the main entry point of your project is `foobar.ml` you can then build a bytecode executable with

```
ocamlbuild -use-ocamlfind -pkgs 'foo,bar' foobar.byte
```

Warning: the names `foo` and `bar` must be the names of the ocamlfind packages, they may differ from the names of the opam packages.

Instead of specifying the packages on the command line, you can create a config file named `_tags` with the following content

```
true: package(foo), package(bar)
```

Basic example with no external dependency

If your project has no external dependency and has `foo.ml` as its main entry point, you can compile a bytecode version with

```
ocamlbuild foo.byte
```

To get a native executable, run

```
ocamlbuild foo.native
```

Read Ocamlbuild online: https://riptutorial.com/ocaml/topic/2183/ocamlbuild

Chapter 8: Pattern Matching

Examples

Factorial Function using Pattern Matching

```
let rec factorial n = match n with
| 0 | 1 -> 1
| n -> n * (factorial (n - 1))
```

This function matches on both the values 0 and 1 and maps them to the base case of our recursive definition. Then all other numbers map to the recursive call of this function.

Evaluation of boolean expressions

We define the type of boolean expressions whose atoms are identified by strings as

```
type expr =
| Atom of string
| Not of expr
| And of expr * expr
| Or of expr * expr
```

and can evaluate these expressions using an `oracle : string -> bool` giving the values of the *atoms* we find as follows:

```
let rec eval oracle = function
| Atom(name) -> oracle name
| Not(expr) -> not(eval oracle expr)
| And(expr1, expr2) -> (eval oracle expr1) && (eval oracle expr2)
| Or(expr1, expr2)  -> (eval oracle expr1) || (eval oracle expr2)
```

See how the function is clear and easy to read. Thanks to correct use of pattern matching, a programmer reading this function needs little time to ensure it is correctly implemented.

Negation normal form : deep pattern matching

Pattern matching allows to deconstruct complex values and it is by no way limited to the “outer most” level of the representation of a value. To illustrate this, we implement the function transforming a boolean expression into a boolean expression where all negations are only on atoms, the so called *negation normal form* and a predicate recognising expressions in this form:

We define the type of boolean expressions whose atoms are identified by strings as

```
type expr =
| Atom of string
| Not of expr
| And of expr * expr
```

```
| Or of expr * expr
```

Let us first define a predicate recognising expressions in *negation normal form*:

```
let rec is_nnf = function
| (Atom(_) | Not(Atom(_))) -> true
| Not(_) -> false
| (And(expr1, expr2) | Or(expr1, expr2)) -> is_nnf expr1 && is_nnf expr2
```

As you see, it is possible to match against nested patterns like `Not(Atom(_))`. Now we implement a function mapping a boolean expression to an equivalent boolean expression in negation normal form:

```
let rec nnf = function
| (Atom(_) | Not(Atom(_))) as expr -> expr
| Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
| Not(Or(expr1, expr2)) -> And(nnf(Not(expr1)),nnf(Not(expr2)))
| And(expr1, expr2) -> And(nnf expr1, nnf expr2)
| Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
| Not(Not(expr)) -> nnf expr
```

This second function makes even more uses of nested patterns. We finally can test our code in the toplevel on the negation of an implication:

```
# let impl a b =
Or(Not(a), b);;
  val impl : expr -> expr -> expr = <fun>
# let expr = Not(impl (Atom "A") (Atom "B"));;
val expr : expr = Not (Or (Not (Atom "A"), Atom "B"))
# nnf expr;;
- : expr = And (Atom "A", Not (Atom "B"))
# is_nnf (nnf expr);;
- : bool = true
```

The OCaml type system is able to verify the exhaustivity of a pattern matching. For instance, if we omit the `Not(Or(expr1, expr2))` case in the `nnf` function, the compiler issues a warning:

```
# let rec non_exhaustive_nnf = function
| (Atom(_) | Not(Atom(_))) as expr -> expr
| Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
| And(expr1, expr2) -> And(nnf expr1, nnf expr2)
| Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
| Not(Not(expr)) -> nnf expr;;
          Characters 14-254:
  ..............function
  | (Atom(_) | Not(Atom(_))) as expr -> expr
  | Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
  | And(expr1, expr2) -> And(nnf expr1, nnf expr2)
  | Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
  | Not(Not(expr)) -> nnf expr..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Not (Or (_, _))
val non_exhaustive_nnf : expr -> expr = <fun>
```

(This warning can be treated as an error with the `-w @8` option when invoking the compiler or the interpreter.)

This feature provides an increased level of safety and correctness in programs that are accepted by the compiler. It has however other uses and can for instance be used in explorative programming. It is is very fun to write a conversion to a normal form, starting with crude versions of the function that handle the easy cases and using examples of non-matched cases provided by the compiler to refine the treatment.

Matching record fields

Pattern matching can be used to deconstruct records. We illustrate this with a record type representing locations in a text file, *e.g.* the source code of a program.

```
type location = {
  filename : string;
  line: int;
  column: int;
  offset: int;
}
```

A value `x` of type location can be deconstructed like this:

```
let { filename; line; column; offset; } = x
```

A similar syntax can be used to define functions, for instance a function to print locations:

```
let print_location { filename; line; column; offset; } =
  Printf.printf "%s: %d: %d" filename line column
```

or alternatively

```
let print_location = function { filename; line; column; offset; } ->
  Printf.printf "%s: %d: %d" filename line column
```

Patterns matching records do not need to mention all fields of a record. Since the function does not use the `offset` field, we can leave it out:

```
let print_location { filename; line; column; } =
  Printf.printf "%s: %d: %d" filename line column
```

When the record is defined in a module, it is enough to qualify the first field occurring in the pattern:

```
module Location =
struct
  type t = {
      filename : string;
      line: int;
      column: int;
```

```
      offset: int;
    }
end

let print_location { Location.filename; line; column; } =
  Printf.printf "%s: %d: %d" filename line column
```

Recursive list processing with pattern matching

Here we demonstrate how to process lists recursively using OCaml's pattern matching syntax.

```
let rec map f lst =
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

In this case, the pattern `[]` matches the empty list, while `hd::tl` matches any list that has *at least one* element, and will assign the first element of the list to `hd` and the rest of the list (could be empty) to `tl`.

Note that `hd::tl` is a very general pattern and will match any list that isn't empty. We can also write patterns that match on lists with a specific number of elements:

```
(* Return the last element of a list. Fails if the list is empty. *)
let rec last lst =
  match lst with
  | [] -> failwith "Empty list"
  | [x] -> x (* Equivalent to x::[], [x] matches a list with only one element *)
  | hd::tl -> last tl

(* The second to last element of a list. *)
let rec second_to_last lst =
  match lst with
  | [] -> failwith "Empty list"
  | x::[] -> failwith "Singleton list"
  | fst::snd::[] -> snd
  | hd::tl -> second_to_last tl
```

Additionally, OCaml supports pattern matching on the elements of lists themselves. We can be more specific about the structure of elements inside a list, and OCaml will infer the correct function type:

```
(* Assuming a list of tuples, return a list with first element of each tuple. *)
let rec first_elements lst =
  match lst with
  | [] -> []
  | (a, b)::tl -> a::(first_elements tl)
(* val first_elements : ('a * 'b) list -> 'a list = <fun> *)
```

By combining these patterns together, we can process any arbitrarily complex list.

Defining a function using pattern matching

The keyword `function` can be used to initiate pattern-matching on the the last argument of a function. For example, we can write a function called `sum`, which computes the sum of a list of integers, this way

```
let rec sum = function
  | [] -> 0
  | h::t -> h + sum t
;;

val sum : int list -> int = <fun>
```

Read Pattern Matching online: https://riptutorial.com/ocaml/topic/2656/pattern-matching

Chapter 9: Pipes, Files, and Streams

Examples

Read from Standard Input and Print to Standard Output

We prepare a file called `reverser.ml` with the following contents:

```
let acc = ref [] in
    try
        while true do
            acc := read_line () :: !acc;
        done
    with
        End_of_file -> print_string (String.concat "\n" !acc)
```

We then compile our program using the following command:

```
$ ocamlc -o reverser.byte reverser.ml
```

We test it out by piping data to our new executable:

```
$ cat data.txt
one
two
three
$ ./reverser.byte < data.txt
three
two
one
```

The `reserver.ml` program is written in an imperative style. While imperative style is fine, it is interesting to compare this to the functional translation:

```
let maybe_read_line () =
  try Some(read_line())
  with End_of_file -> None

let rec loop acc =
  match maybe_read_line () with
  | Some(line) -> loop (line :: acc)
  | None -> List.iter print_endline acc

let () = loop []
```

Thanks to introducing the function `maybe_read_line` the control flow is much simpler in this second version than in the first.

Read Pipes, Files, and Streams online: https://riptutorial.com/ocaml/topic/3252/pipes--files--and-

streams

Chapter 10: Tail recursion

Introduction

Functional languages such as OCaml rely heavily on recursive functions. However, such functions can lead to memory over consumption or, when handling large datasets, to stack overflows.

Tail recursion is an important source of optimization in such cases. It allows a program to drop the caller context **when the recursive call is the last of the function**.

Examples

Sum function

Below is a non-tail-recursive function to compute the sum of a list of integers.

```
let rec sum = function
  | [] -> 0
  | h::t -> h + (sum t)
```

The last operation the function performs is the addition. Thus, the function isn't tail-recursive.

Below is a tail-recursive version of the same function.

```
let sum l =
  let rec aux acc = function
    | [] -> acc
    | h::t -> aux (acc+h) t
  in
  aux 0 l
```

Here, the `aux` function is tail-recursive: the last operation it performs is calling itself. As a consequence, the latter version of `sum` can be used with lists of any length.

Read Tail recursion online: https://riptutorial.com/ocaml/topic/9650/tail-recursion

Chapter 11: Write your first OCaml Script

Examples

Hello World

This example assumes you've installed OCaml.

Compiling OCaml Code

Create a new file named `hello.ml`, with the following contents:

```
print_string "Hello world!\n"
```

`ocamlc` is the OCaml compiler. To compile and run this script, run

```
$ ocamlc -o hello hello.ml
```

and then execute the resulting binary

```
$ ./hello
Hello world!
```

Executing OCaml Code

You can also run this script without compiling it into a binary. You can do so by using `ocaml`, the ocaml toplevel system that permits interactive use of OCaml. In your shell, simply run

```
$ ocaml hello.ml
Hello world!
```

In the REPL

Open a new shell, and type `ocaml` to open the toplevel system. Once in the session, you can type the same program:

```
        OCaml version 4.02.1

# print_string "hello world!\n";;
```

press enter to evaluate the expression, and trigger the print.

```
hello world!
- : unit = ()
```

Success! We see it printed `hello world!`, but what is the `- : unit = ()` about? OCaml has no statements, everything is an expression that evaluates to some typed value. In this case, `print_string` is a function that takes in a `string`as input, and returns a `unit`. Think of `unit` as a type that can only take one value, `()` (also referred to as unit), and represents a finished computation that returns no meaningful value.

In this case, `print_string` also has the side-effect of putting characters it received as input onto the screen, which is why we see the first line.

To exit the REPL, press `ctrl+D`.

As a Unix script

We have two ways to create an OCaml script. The first use the system toplevel (provided by your package manager like `apt-get`) and the second use the toplevel provided by OPAM.

Use the system toplevel

Open your favorite editor, and write:

```
#!/usr/bin/ocaml

print_string "hello worlds!\n";;
```

After, you can use `chmod +x your_file.ml` and you can execute your script with `./your_file.ml`.

Use the toplevel provided by OPAM

```
#!/usr/bin/env ocaml

print_string "hello worlds!\n";;
```

The big difference is about the version of your toplevel. Indeed, if you configured your OPAM with a specific switch (like `opam switch 4.03.0`), the script will use OCaml 4.03.0. In the first way, in Debian Sid for example, the script will use OCaml 4.02.3.

You can replace the *shebang* by `#!/usr/bin/env utop` to use `utop` instead the vanilla toplevel.

utop

`utop` is another ocaml toplevel outside the distribution - that means, you need to download and install `utop` (the easy way is to use OPAM: `opam install utop`). `utop` has many features like the

historic, the completion and the interactive line editing.

So, if you want an easy way to try some ocaml codes, `utop` is the best.

Why `utop` and not `ocaml`?

`utop` and `ocaml` have no a big difference if you want an ocaml script like above. But the common thing in the OCaml community is to use `utop` instead `ocaml`.

In fact, the `ocaml` REPL is provided by the ocaml distribution. So, this REPL follows the release cycle of the compiler and if you want some extras features, you need to wait the next release of the compiler. `utop`, as we explained, is outside the distribution, so the release cycle is not constraint by the compiler and if you want an extra feature, you will be more likely to try to push this feature inside `utop` than `ocaml` :) !

For this point (and for the historic feature) most people in the ocaml community prefer to use `utop` than `ocaml`.

Read Write your first OCaml Script online: https://riptutorial.com/ocaml/topic/2168/write-your-first-ocaml-script

Credits

S. No	Chapters	Contributors
1	Getting started with OCaml	chucksys, Community, incud, Jason Yeo, Pierre Chambart, Romain Calascibetta, Thomash
2	Common Pitfalls	Eli Sadoff, RichouHunter
3	Functions	Conrad.Dean, fileyfood500, Hunan Rostomyan, Jason Yeo
4	Higher Order Functions	chucksys, Hunan Rostomyan, Michael Le Barbier Grünewald
5	List Processing	Hunan Rostomyan, Kyle, Michael Le Barbier Grünewald
6	Mutable record fields	RichouHunter
7	Ocamlbuild	Thomash
8	Pattern Matching	Conrad.Dean, jayelm, Michael Le Barbier Grünewald, RichouHunter
9	Pipes, Files, and Streams	Conrad.Dean, ivg, Marco Predari, Michael Le Barbier Grünewald
10	Tail recursion	RichouHunter
11	Write your first OCaml Script	Conrad.Dean, Jason Yeo, Kevin Chavez, Romain Calascibetta, Uncle Pa