# MORE OCAML *Algorithms, Methods & Diversions*

In *More OCaml* John Whitington takes a meandering tour of functional programming with OCaml, introducing various language features and describing some classic algorithms. The book ends with a large worked example dealing with the production of PDF files. There are questions for each chapter together with worked answers and hints.

*More OCaml* will appeal both to existing OCaml programmers who wish to brush up their skills, and to experienced programmers eager to explore functional languages such as OCaml. It is hoped that each reader will find something new, or see an old thing in a new light. For the more casual reader, or those who are used to a different functional language, a summary of basic OCaml is provided at the front of the book.

JOHN WHITINGTON founded a software company which uses OCaml extensively. He teaches functional programming to students of Computer Science at the University of Cambridge. His other books include *"PDF Explained"* (O'Reilly, 2012) and *"OCaml from the Very Beginning"* (Coherent, 2013).

# More OCaml

*Algorithms, Methods & Diversions*

John Whitington

COHERENT PRESS

COHERENT PRESS
Cambridge

Published in the United Kingdom by Coherent Press, Cambridge



First published August 2014
Reprinted with corrections July 2015
Reprinted 2016
Updated for OCaml language changes October 2017

*A catalogue record for this book is available from the British Library*

ISBN 978-0-9576711-1-9 Paperback

*by the same author*

PDF Explained (O'Reilly, 2012)
OCaml from the Very Beginning (Coherent, 2013)
A Machine Made this Book: Ten Sketches of Computer Science (Coherent, 2016)

# Contents

# Preface

When I wrote "OCaml from the Very Beginning", the intention was to have a book with no prerequisites – a bright individual, new to programming, could follow it. Because of this, and for length concerns, plenty of interesting material had to be omitted. This text, not being constrained in the same way, contains a variety of topics which require some existing experience with a functional language. Those who have read the previous text should have no problem with this one. Equally, it should be comprehensible to a functional programmer familiar with another language such as Standard ML or Haskell. The reader may need to make occasional reference to the OCaml manual.

There are, typically, two different activities when writing programs larger than a few dozen lines: firstly, dealing with the challenges of complexity inherent in the problem, by finding appropriate abstraction mechanisms and, secondly, finding and using the wide range of third-party libraries available for a given language. Most projects involve a combination of the two. In this text we concentrate wholly on the former, using nothing other than the OCaml Standard Library. Keeping up with the myriad third-party OCaml libraries is a task better suited to other media.

The book consists of sixteen short chapters falling broadly into three categories. Some introduce pieces of OCaml syntax with worked examples. Some survey practical topics such as input/output. Some cover little diversions or puzzles. The main matter of the book ends with a lengthy worked example: a program to build PDF files containing computer-generated drawings and text. There are full answers and hints for all questions in the book, and additional material in the online resources.

## Acknowledgments

The quotation in Chapter 6 is taken from ISO-32000 © International Organization for Standardization. The tables of codes in the same chapter are taken from ITU-T T.30 © International Telecommunication Union. The presentation of the balancing operation for Red-Black trees in Chapter 11 and its functional implementation is due to Chris Okasaki, as described in the invaluable "Purely Functional Data Structures" (Cambridge University Press, ISBN 978-0521663502, 1998). Chapter 12 was inspired by a University of Cambridge Computer Science Tripos exam question set by Lawrence C. Paulson in 1999. Question 3 of that chapter is due to Peter D. Schumer in "Mathematical Journeys" (John Wiley & Sons, ISBN 0-471-22066-3, 2004).

I am grateful to the many colleagues and friends with whom I have been able to discuss OCaml style and substance, including Mark Shinwell, Leo White, Daniel Bünzli, Anil Madhavapeddy, Stephen Dolan and many others whom I have forgotten. Helpful comments on an earlier draft were provided by Stefan Schmiedl, Manuel Cornes, Jonas Bülow, Emmanuel Delaborde, Mario Alvarez Picallo, Giannis Tsaraias, Emmanuel Oga, and André Bjärby.

# Summary of Basic OCaml

*This chapter contains a summary of each OCaml construct used in the book, together with some examples. Pieces of OCaml syntax not contained in this chapter will be introduced as and when they are needed throughout the rest of the book. Existing OCaml programmers may skip this chapter.*

## Simple Data Types

Integers `min_int` ... `-3 -2 -1 0 1 2 3` ... `max_int` of type **int**. Booleans `true` and `false` of type **bool**. Characters of type **char** like `'X'` and `'!'`.

Mathematical operators `+ - * / mod` which take two integers and give another.

```
    6 * 2
⟹  12
```

Operators `= < <= > >= <>` which compare two values and evaluate to either `true` or `false`.

```
    1 + 2 + 3 = 1 * 2 * 3
⟹  true
```

The conditional **`if`** *expression1* **`then`** *expression2* **`else`** *expression3*, where *expresssion1* has type **bool** and *expression2* and *expression3* have the same type as one another.

```
    if 4 * 3 > 2 * 2 then 1 else 0
⟹  1
```

The boolean operators && (logical AND) and || (logical OR) which allow us to build compound boolean expressions.

```
    1 = 2 || 2 = 2
⟹  true
```

Tuples to combine a fixed number of elements `(a, b)`, `(a, b, c)` etc. with types $\alpha \times \beta$, $\alpha \times \beta \times \gamma$ etc. For example, `(1, '1')` is a tuple of type **int** $\times$ **char**. On the screen, OCaml writes `'a` for $\alpha$ etc.

Strings, which are sequences of characters written between double quotes and are of type **string**. For example, `"one"` has type **string**.

## Names and Functions

Assigning a name to the result of evaluating an expression using the **`let`** *name* `=` *expression* construct.

```
let x = 5 > 2          x is new, and is true
```

Building compound expressions using **`let`** *name1* `=` *expression1* **`in`** **`let`** *name2* `=` *expression2* **`in`** ...

```
let x = 4 in let y = 5 in x + y
```

Anonymous (un-named) functions **`fun`** *name* `->` *expression*.

```
    (fun x -> x * 2) 4
⟹  8
```

Making operators into functions as in `( < )` and `( + )`.

```
( + ) 1 2 ⟹ 3
```

Functions, introduced by **`let`** *name argument1 argument2* ... `=` *expression*. These have type $\alpha \to \beta$, $\alpha \to \beta \to \gamma$ etc. for some types $\alpha$, $\beta$, $\gamma$ etc. For example, **`let`** `f a b = a > b` is a function of type $\alpha \to \alpha \to$ **bool**.

Recursive functions, which are introduced in the

same way, but using **let rec** instead of **let**. For example, here is a function g which calculates the smallest power of two greater than or equal to a given positive integer, using the recursive function f:

```
let rec f x y =
  if y < x then f x (2 * y) else y

let g z = f z 1
```

Mutually recursive functions, introduced by writing **let rec** f x = **...** **and** g y = **...** **and** **...**

# Pattern Matching

Matching patterns using **match** *expression1* **with** *pattern1* | ... -> *expression2* | *pattern2* | ... -> *expression3* |... The expressions *expression2*, *expression3* etc. must have the same type as one another, and this is the type of the whole **match** ... **with** expression. The special pattern _ which matches anything.

```
match x with
  0 -> 1
| 1 | 2 -> 3
| _ -> 4
```

Matching two or more things at once, using commas to separate as in **match** a, b **with** 0, 0 -> *expression1* | x, y -> *expression2* | ...

```
match x, y, z with
  0, 0, 0 -> true
| _, _, _ -> false
```

# Lists

Lists, which are ordered collections of zero or more elements of like type. They are written between square brackets, with elements separated by semicolons e.g. [1; 2; 3; 4; 5]. If a list is non-empty, it has a head, which is its first element, and a tail, which is the list composed of the rest of the elements.

The :: "cons" operator, which adds an element to the front of a list. The @ "append" operator, which concatenates two lists together.

```
1 :: [2; 3] ⟹ [1; 2; 3]
[1; 2] @ [3] ⟹ [1; 2; 3]
```

Lists and the :: "cons" symbol may be used for pattern matching to distinguish lists of length zero, one, etc. and with particular contents. For example, we can calculate the length of a list:

```
let rec length l =
  match l with
    [] -> 0
  | _::t -> 1 + length t
```

# Exceptions

Defining exceptions with **exception** *name*. They can carry extra information by adding **of** *type*. Raising exceptions with **raise**. Handling exceptions with **try** ... **with** ...

```
exception Problem of int

let f x y =
  if y = 0
    then raise (Problem x)
    else x / y

let g x y =
  try f x y with Problem p -> p
```

# Partial Application

Partial application of functions by giving fewer than the full number of arguments. Partial application with functions built from operators.

```
let add x y = x + y

List.map (add 3) [1; 2; 3]
⟹ [4; 5; 6]

List.map (( + ) 3) [1; 2; 3]
⟹ [4; 5; 6]
```

# New Data Types

New types with **type** *name* = *constructor1* **of** *type1* | *constructor2* **of** *type2* | ... Pattern matching on them as with the built-in types. Polymorphic types.

```
type colour =
  Red | Blue | Green | Grey of int
```

```
[Red; Blue; Grey 16]
```
*this has type colour* ***list***

```
type 'a tree =
   Lf
 | Br of 'a tree * 'a * 'a tree
```

For example, `Br (Lf, 'X', Br (Lf, 'Y', Lf))` has type **char** tree. A useful built-in data type is the **option** type, defined as **type** `'a option = None | Some of 'a`. A type can be polymorphic in more than one type parameter, for example `('a, 'b) Hashtbl.t`, as in the Standard Library.

# Basic Input / Output

The value `()` and its type **unit**. Input channels of type **in_channel** and output channels of type **out_channel**. Built-in functions such as `open_in`, `close_in`, `open_out`, `close_out`, `input_char`, `output_char` etc. for reading from and writing to them respectively.

# Mutable State

References of type $\alpha$ **ref**. Building them using `ref`, accessing their contents using `!` and updating them using the `:=` operator.

```
# let p = ref 0;;
val p : int ref = {contents = 0}
# p := 5;;
- : unit = ()
# !p;;
- : int = 5
```

Arrays of type $\alpha$ **array** written like `[|1; 2; 3|]`. Creating an array with the built-in function `Array.make`, finding its length with `Array.length`, accessing an element with `a.(`*subscript*`)`. Updating with `a.(`*subscript*`) <-` *expression*.

```
let swap a x y =
  let t = a.(x) in
    a.(x) <- a.(y); a.(y) <- t
```

Bracketing expressions together with **begin** and **end** instead of parentheses for readability.

```
if x = y then
  begin
    a := b;
    c := d
  end
else
  e := f
```

Performing an action many times based on a boolean condition with the **while** *boolean expression* **do** *expression* **done** construct.

```
while !x < y do x := !x * 2 done
```

Performing an action a fixed number of times with a varying parameter using the **for** *name* = *start* **to** *end* **do** *expression* **done** construct.

```
for x = 1 to 10 do print_int x done
```

# Floating-point Numbers

Floating-point numbers `min_float` ... `max_float` of type **float**. Floating-point operators `+. *. -. /. **` and built-in functions `sqrt` `log` etc.

`2. ** 0.2` $\Longrightarrow$ `1.1486983549970351`

# The OCaml Standard Library

Using functions from the OCaml Standard Library with the form *Module*.*function*. For example, `List.map`, `String.length`, `Array.copy` etc. The **Buffer** module allows the efficient collation of strings into larger ones.

# Simple Modules

Writing modules in `.ml` files. Building interfaces in `.mli` files with types and the **val** keyword. For example, the `.ml` file with contents **let** `f x = x + 1` might have the interface **val** `f : int -> int`

# Compiling Programs

The `ocamlc` and `ocamlopt` compilers. For example:

`ocamlc -o x x.ml` builds `x` (or `x.exe`) from `x.ml` with the bytecode compiler.

`ocamlopt -o x x.ml` builds `x` (or `x.exe`) from `x.ml` with the native code compiler.

# Our Working Environment

Every piece of code in an example, and every answer to an end-of-chapter question can be downloaded from the book's website at `http://www.ocaml-book.com/` and built on the reader's machine. The programs work on Unix (including Linux), Mac OS X, and Microsoft Windows, using OCaml 4.02 or later.

Since several of our programs require tail-recursive list functions, we provide a wrapper to the Standard Library **List** module which provides them. In addition, three small utility functions (`take`, `drop`, and `from`) are provided in a Util module. These modules are contained in the module More. So, by writing **open** `More`, the Util module is available, and all functions in the **List** module are tail-recursive.

If using OPAM, the OCaml Package Manager, this package may be installed by writing `opam install more-ocaml`. Then (or by installing in another manner – see the online resources for details) our modules are available in the top level:

```
        OCaml

# #use "topfind";;
- : unit = ()
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads

- : unit = ()
# #require "more";;
/Users/john/.opam/4.02.0/lib/ocaml/more.cma: loaded
/Users/john/.opam/4.02.0/lib/more: added to search path
# open More;;
# Util.take;;
- : 'a list -> int -> 'a list = <fun>
```

They are also available when compiling stand-alone programs with the bytecode compiler `ocamlc` or the native code compiler `ocamlopt`:

```
ocamlfind ocamlc -package more program.ml -linkpkg -o program
ocamlfind ocamlopt -package more program.ml -linkpkg -o program
```

Further instructions, including for use on platforms where OPAM is not supported, are given in the online resources.

## Timing with the **Unix** module

Sometimes we will wish to see how much time a piece of code takes. This can be achieved using the function `gettimeofday` from the **Unix** module (contrary to its name, this module also works on Windows). This function returns a floating-point number representing the time since 00:00:00 GMT, Jan. 1, 1970, in seconds:

```
        OCaml

# #use "topfind";;
- : unit = ()
Findlib has been successfully loaded. Additional directives:
  #require "package";;      to load a package
  #list;;                   to list the available packages
  #camlp4o;;                to load camlp4 (standard syntax)
  #camlp4r;;                to load camlp4 (revised syntax)
  #predicates "p,q,...";;   to set these predicates
  Topfind.reset();;         to force that packages will be reloaded
  #thread;;                 to enable threads

- : unit = ()
# #require "unix";;
/Users/john/.opam/4.01.0/lib/ocaml/unix.cma: loaded
# Unix.gettimeofday ();;
- : float = 1407679005.43897
# Unix.gettimeofday ();;
- : float = 1407679011.08860302
```

Now, by evaluating `Unix.gettimeofday ()`, running a piece of code, and evaluating `Unix.gettimeofday ()` once more, we can calculate the elapsed time. To use the **Unix** module when compiling stand-alone programs:

```
ocamlfind ocamlc -package more,unix program.ml -linkpkg -o program
ocamlfind ocamlopt -package more,unix program.ml -linkpkg -o program
```

## Coping with Errors

Finding and fixing errors is an inevitable part of programming. A short guide to the most common errors and how to deal with them has been included on page 195.