OCaml for Scientists

By Jon Harrop
Contents

1 Introduction
   1.1 Good programming style .............................................. 1
   1.2 A Brief History of OCaml ............................................. 2
   1.3 Benefits of OCaml .................................................... 2
   1.4 Running OCaml programs ............................................ 3
      1.4.1 Top-level ........................................................ 3
      1.4.2 Byte-code compilation ......................................... 4
      1.4.3 Native-code compilation ....................................... 5
   1.5 OCaml syntax .......................................................... 5
      1.5.1 Language overview ............................................. 5
         1.5.1.1 Types ...................................................... 5
         1.5.1.2 Variables and functions .................................. 7
         1.5.1.3 Tuples, records and variants .............................. 9
         1.5.1.4 Lists and arrays ......................................... 12
         1.5.1.5 If .......................................................... 13
         1.5.1.6 Program composition ...................................... 14
         1.5.1.7 More about functions ...................................... 14
         1.5.1.8 Modules .................................................. 16
      1.5.2 Pattern matching ................................................ 16
         1.5.2.1 Guarded patterns .......................................... 18
         1.5.2.2 Erroneous patterns ....................................... 19
         1.5.2.3 Good style ............................................... 20
         1.5.2.4 Nested patterns ......................................... 23
         1.5.2.5 Parallel pattern matching ............................... 23
      1.5.3 Exceptions ...................................................... 24
CONTENTS

1.5.4 Polymorphism .................................................. 26
1.5.5 Currying .......................................................... 28
1.6 Functional vs Imperative programming ......................... 28
1.7 Recursion .......................................................... 30
1.8 Applicability ....................................................... 32

2 Program Structure .................................................. 33
2.1 Nesting ............................................................. 33
2.2 Factoring .......................................................... 34
2.3 Modules ............................................................ 37
   2.3.1 Signatures ................................................... 39
   2.3.2 Structures .................................................... 39
   2.3.3 Anonymous signatures ................................. 41
   2.3.4 Use of the IntRange module .......................... 42
   2.3.5 Another example ........................................... 43
2.4 Objects ........................................................... 46
   2.4.1 Classes ....................................................... 46
   2.4.2 Objects ....................................................... 47
      2.4.2.1 Immediate objects ................................... 47
      2.4.2.2 Classed objects .............................. 48
      2.4.2.3 Inheritance ........................................... 49
2.5 OCaml browser ................................................... 50
2.6 Compilation ...................................................... 51
   2.6.1 Linking with libraries .................................. 55
2.7 Custom top-levels ................................................ 57

3 Data Structures .................................................... 59
3.1 Algorithmic Complexity ....................................... 59
   3.1.1 Primitive operations .................................... 60
   3.1.2 Complexity ................................................ 61
      3.1.2.1 Asymptotic complexity .......................... 61
3.2 Arrays ............................................................ 64
3.3 Lists ............................................................... 68
3.4 Sets ............................................................... 73
CONTENTS

3.5 Hash tables .......................................................... 78
3.6 Maps ................................................................. 80
3.7 Summary ............................................................ 83
3.8 Heterogeneous containers ........................................ 84
3.9 Trees ................................................................. 85
   3.9.1 Balanced trees ................................................ 90
   3.9.2 Unbalanced trees ............................................. 91

4 Numerical Analysis .................................................... 101
4.1 Number representation ............................................. 101
   4.1.1 Integers ........................................................ 101
   4.1.2 Floating-point numbers ..................................... 102
4.2 Quirks ............................................................... 105
4.3 Algebra .............................................................. 105
4.4 Interpolation ....................................................... 107
4.5 Quadratic solutions ............................................... 108
4.6 Mean and variance ................................................ 109
4.7 Other forms of arithmetic ....................................... 111
   4.7.1 Rational arithmetic ......................................... 111
   4.7.2 High-precision floating point ................................ 112
   4.7.3 Adaptive precision .......................................... 113

5 Input and Output ...................................................... 115
5.1 Printing to screen ................................................ 115
5.2 Reading from and writing to disc ............................... 116
5.3 Marshalling ........................................................ 117
5.4 Lexing and Parsing ................................................ 118
   5.4.1 Lexing .......................................................... 119
   5.4.2 Parsing ........................................................ 123

6 Visualization .......................................................... 127
6.1 Overview of OpenGL .............................................. 127
   6.1.1 GLUT .......................................................... 128
6.2 Basic rendering ..................................................... 130
6.2.1 Geometric primitives .................................................. 131
6.2.2 Filling ........................................................................ 133
6.2.3 Projection ................................................................... 133
6.2.4 Animation ................................................................... 135
6.3 Transformations ................................................................. 136
6.4 Efficient rendering ............................................................. 137
6.5 Rendering scientific data .................................................... 139

7 Optimization ................................................................. 145
  7.1 Profiling .......................................................................... 145
  7.2 Algorithmic optimization ................................................ 148
  7.3 Lower-level optimizations .............................................. 149
    7.3.1 Benchmarking data structures .................................. 149
    7.3.2 Automated transformations ...................................... 153
      7.3.2.1 Compiler optimizations .................................... 153
      7.3.2.2 Defunctorizing ................................................ 154
    7.3.3 Manual transformations ............................................ 154
      7.3.3.1 Tail-recursion ................................................. 154
      7.3.3.2 Deforesting ................................................... 157
      7.3.3.3 Terminating early ............................................ 158
      7.3.3.4 Specializing data structures .............................. 159
      7.3.3.5 Avoiding polymorphic numerical functions .......... 160
      7.3.3.6 Unboxing data structures .................................. 161

8 Libraries ................................................................. 165
  8.1 Command-line arguments .............................................. 165
  8.2 Timing .......................................................................... 167
  8.3 Big arrays ..................................................................... 169
  8.4 Vector-Matrix ................................................................. 171
  8.5 Fourier transform ........................................................... 172
CONTENTS

9 Simple Examples .................................................. 177
  9.1 Arithmetic .................................................... 177
  9.2 List related .................................................. 181
    9.2.1 count .................................................... 181
    9.2.2 position ................................................ 182
    9.2.3 mapi ..................................................... 182
    9.2.4 chop ..................................................... 184
    9.2.5 dice ..................................................... 185
    9.2.6 replace .................................................. 186
    9.2.7 sub ....................................................... 186
    9.2.8 extract .................................................. 187
    9.2.9 randomize ................................................ 187
    9.2.10 permute ................................................ 188
    9.2.11 Run-length encoding ................................... 189
  9.3 String related ................................................ 189
    9.3.1 string_of_list ......................................... 190
    9.3.2 DNA sequence IO ....................................... 190
    9.3.3 Matrix IO ............................................... 192
  9.4 Array related ................................................ 194
    9.4.1 map2 ..................................................... 194
    9.4.2 Double folds ............................................ 194
    9.4.3 rotate .................................................. 195
    9.4.4 Matrix trace ............................................ 196
  9.5 Higher-order functions ...................................... 196
    9.5.1 Data structures of functions .......................... 196
    9.5.2 Tuple related ........................................... 197
    9.5.3 Generalised products .................................. 198
    9.5.4 Converting between container types ................... 199

10 Complete Examples .............................................. 201
  10.1 Maximum entropy method .................................... 201
    10.1.1 Formulation ............................................ 202
    10.1.2 Implementation ........................................ 203
## CONTENTS

10.1.2.1 Lexer .................................................. 203
10.1.2.2 Parser .................................................. 204
10.1.2.3 Main program .......................................... 205
10.1.2.4 Compilation ........................................... 208
10.1.2.5 Results ................................................ 209
10.1.2.6 Optimisation ........................................... 209

10.2 Global minimization ........................................ 210
   10.2.1 The mutate function ................................... 211
   10.2.2 Efficiency ............................................. 213
   10.2.3 Implementation ......................................... 214
       10.2.3.1 Lexer ............................................. 214
       10.2.3.2 Parser ........................................... 215
       10.2.3.3 Main program ..................................... 216
       10.2.3.4 Compilation ...................................... 221
       10.2.3.5 Results ........................................... 222

10.3 Finding $n^{th}$-nearest neighbours ......................... 222
   10.3.1 Formulation ........................................... 222
   10.3.2 Implementation ........................................ 224
       10.3.2.1 Lexer ............................................. 226
       10.3.2.2 Parser ........................................... 226
       10.3.2.3 Main program ..................................... 228
       10.3.2.4 Compilation ...................................... 233
       10.3.3 Results ............................................. 233

10.4 Eigen problems ............................................ 234
   10.4.1 Implementation ........................................ 235
   10.4.2 Results .............................................. 236

10.5 Discrete wavelet transform ................................ 237

Bibliography .................................................. 239
## CONTENTS

### A Advanced Topics
- A.1 Data sharing ........................................... 241
- A.2 Labelled and optional arguments ...................... 242
- A.3 Defining binary infix operators ......................... 244
- A.4 Installing top-level pretty printers .................... 245
- A.5 Monomorphism ........................................ 245
- A.6 Functors ............................................. 246
- A.7 Memoization ............................................. 247
- A.8 Polymorphic variants .................................... 249
- A.9 Phantom types .......................................... 250
- A.10 Exponential type growth .............................. 251

### B Troubleshooting
- B.1 Dangerous if ........................................... 253
- B.2 Scoping subtleties ....................................... 255
- B.3 Evaluation order ....................................... 256
- B.4 Constructor arguments ................................ 257
- B.5 Recycled types ......................................... 257
- B.6 Mutable array contents ................................ 258
- B.7 Polymorphic problems .................................. 259
- B.8 Local and non-local variable definitions .............. 259
Preface

This book aims to encourage the scientific community to adopt stricter approaches to computer programming, emphasising correctness over performance, beginning with the selection of the Objective CAML language due to its inherent safety. Although scientists are the principal target audience, anyone interested in learning more about modern programming techniques is likely to benefit from reading this book.

Due to the widespread adoption of computers for everything from the logging and analysis of experimentally observed data to the computationally-intensive simulation of physical systems, the computer is now a vitally important tool for scientists. However, poor approaches to programming are endemic in current scientific culture. Specifically, more worth is placed on scientific results than on the creation of generic programs which could have been used to generate many more such results, leading to the constant redevelopment of disposable programs. If this can be cured, science will benefit from professional quality (correct, reusable and future-proof) programs and data formats which will greatly accelerate the rate of scientific discovery.

This book may be divided into three main parts. Chapters 1–5 introduce the reader to the syntax of Objective CAML and the creation and execution of working programs based upon the most important features found in the language. Chapters 6–8 deal with extended functionality available via libraries and optimisation. Finally, chapters 9 and 10 present a variety of examples. In particular, chapter 10 describes the creation of complete programs capable of solving some of the most important types of problem found in computational science.
Notation

The sets of integers, real and complex numbers are denoted as \( \mathbb{Z} \), \( \mathbb{R} \) and \( \mathbb{C} \) respectively.

The unit imaginary number is denoted as \( i = \sqrt{-1} \) according to the convention of the physical sciences (engineers use \( j \)). Real and imaginary parts of a complex number \( z = x + iy \), \( x, y \in \mathbb{R} \) are denoted as \( \text{Re}[z] = x \) and \( \text{Im}[z] = y \) respectively. The complex conjugate is denoted \( z^* = x - iy \).

In complex polar notation \( z \) is written \( z = r e^{i \theta} \), \( r, \theta \in \mathbb{R} \) where \( r = |z| \) is termed the modulus of \( z \) and \( \theta = \text{arg}[z] \) is termed the argument of \( z \).

Vectors are written in bold typeface (e.g. \( \mathbf{r} \)) and default to \( \mathbf{r} \in \mathbb{R}^3 \).

Directed integer rounding functions are referred to as floor and ceiling and are denoted by and formulated as:

\[
[x] = \max \{n \in \mathbb{Z} ; n \leq x\} \quad \text{and} \quad [x] = \min \{n \in \mathbb{Z} ; n \geq x\}
\]

respectively.

Ranges are written using round or square braces to indicate exclusive or inclusive range ends respectively, e.g. for an integer range \( [1 \ldots n] \equiv \{1 \ldots n-1\} \).

Derivatives of functions with respect to the first argument can be written in shorthand notation, e.g. \( \Phi''(x, y) = \frac{\partial^2}{\partial x^2} \Phi(x, y) \) and \( \Phi'(1, y) = \frac{\partial \Phi}{\partial x} |_{x=1} \).

Inner products of complex-valued functions are written using Dirac notation, i.e.

\[
\langle f|g \rangle = \int_{-\infty}^{\infty} f^*(x)g(x) \, dx
\]

Readers should be aware that the standard mathematical notation \( \langle g, f \rangle \equiv \langle f|g \rangle \) is often used in other literature.

Fourier transforms are denoted and formulated according to convention for the physical sciences, with the forward transform:

\[
\hat{f} = (2\pi)^{-\frac{1}{2}} \langle e^{i\omega t}|f \rangle
\]

and the reverse transform:

\[
f = (2\pi)^{-\frac{1}{2}} \langle e^{-i\omega t}|\hat{f} \rangle
\]

Readers should be aware that alternate formulations exist in related fields.
The set operations union, intersection and difference are denoted by the symbols $\cup$, $\cap$ and $\setminus$ respectively. The cartesian product of two sets $A$ and $B$ is written $A \times B$. For example, \( \{1, 2\} \times \{a, b, c\} \) is the set of pairs \( \{\{1, a\}, \{1, b\}, \{1, c\}, \{2, a\}, \{2, b\}, \{2, c\}\} \).

Function $L^p$ norms, denoted by $\| f \|_{L^p}$, are defined as:

$$
\| f \|_{L^p} = \left( \int_{-\infty}^{\infty} |f(x)|^p \, dx \right)^{\frac{1}{p}}
$$

and we default to the $L^2$ norm, e.g. the Plancherel equality may be written $\| \hat{f} \|_2^2 = \| f \|_2^2$ $\forall f \in L^2(\mathbb{R})$. In particular, $L^2(\mathbb{R})$ is the Hilbert space of functions $f : \mathbb{R} \rightarrow \mathbb{C}$ with $\| f \|_2^2 \in \mathbb{R}$.

A function $f$ which maps values from a set $A$ onto a set $B$ is written $f : A \rightarrow B$. Typically, this is used to indicate the argument and return types of a function, e.g. $\tilde{f} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{C}$ (parsed as $\tilde{f} : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{C}$) is a function which maps two real numbers (expressed as an element in the cartesian product of the set of real numbers with itself) onto a complex number.

The variance $\sigma_f^2$ of a function $f$ satisfying $\| f \|^2 = 1$ is defined as:

$$
\sigma_f^2 = \int_{-\infty}^{\infty} t^2 f(t) \, dt - \left( \int_{-\infty}^{\infty} t f(t) \, dt \right)^2
$$

where $\sigma_f$ is known as the standard deviation.

The $\Gamma$-function $\Gamma[z] : \mathbb{C} \rightarrow \mathbb{C}$ is defined to be:

$$
\Gamma[z] = \int_{0}^{\infty} t^{z-1} e^{-t} \, dt
$$
Glossary

Glossary of terms

λ-function an anonymous function.

Abstract type a type with a visible name but hidden implementation. Abstract types are created by declaring only the name of the type in a module signature, and not the complete implementation of the type as given in the module structure.

Accumulator a variable used to build the result of a computation. The concept of an accumulator underpins the fold algorithm (introduced on page 36). For example, in the case of a function which sums the elements of a list, the accumulator is the variable which holds the cumulative sum while the algorithm is running.

Algorithm a mathematical recipe for solving a problem. For example, Euler's method is a well-known algorithm for finding the largest common divisor of two numbers.

Array a flat container which provides random access to its elements in $O(1)$ time-complexity.

See section 3.2.

Associative container A container which represents a mapping from keys to values.

Asymptotic complexity an approximation to the complexity of an algorithm, derived in the limit of infinite input complexity and, typically, as a lower or upper bound. For example, an algorithm with a complexity $f(n) = 3 + 3n + n^2$ has an asymptotic complexity $O(n^2)$.

See section 3.1.

Balanced tree a tree data structure in which the maximum variation in depth can be shown to tend to a finite value in the limit of an infinite number of leaves in the tree. Often this restriction is tightened to require that the variation in depth is no more than a single level. See section 3.9.1 for a brief discussion.

Binary tree a tree data structure in which all non-leaf nodes contain exactly two binary trees.

Byte code a representation of a program which is intermediate between the source code and machine code. For example, the ocamlc compiler transforms OCaml source code into a platform-independent byte-code. Section 1.4.2 describes how to compile OCaml programs into byte code.
Cache an intermediate store used to accelerate the fetching of a subset of data.

Cache hit the quick process of retrieving data which is already in the cache.

Cache miss the slow process of fetching data to fill the cache when a request is made for data not already in the cache.

Cache coherent accessing data (typically in memory) sequentially, or more sequentially than random access, in order to minimize cache misses.

Cartesian cross product a set-theoretic form of outer product. For example, the cartesian cross product of the set $A = \{a, b\}$ with the set $B = \{c, d, e\}$ is the set of pairs $A \times B = \{(a, c), (a, d), (a, e), (b, c), (b, d), (b, e)\}$.

Class expression definition of values and methods implemented in any object created from this class.

Class type declaration of values and methods which any object adhering to this class type must provide.

Compile-time while a program is being compiled.

Compiler a program capable of transforming other programs. For example, the compiler ocamlopt transforms OCaml source code into executable machine code.

Complexity a quantitative indication of the growth of the computational requirements (such as time or memory) of an algorithm with respect to its input. Algorithmic complexity is described in section 3.1.

Cons the $::$ operator. When used in a pattern, $h :: t$ is said to decapitate a list, binding $h$ to the first element of a list (the head) and $t$ to a list containing the remaining elements (the tail). When used in an expression, $h :: t$ prepends the element $h$ onto the list $t$. See sections 3.3 and 9.2 for example uses of the cons operator.

Container a data structure used to store values. The values held in a data structure are known as the elements of the data structure. Arrays, lists and sets are examples of data structures.

Curried function any function which returns a function as its result. See section 1.5.5.

Data structure a scheme for organizing related pieces of information.

Decapitate splitting a list into its first element (the head) and a list containing the remaining elements (the tail).

Exception a programming construct which allows the flow of execution to be altered by the raising of an exception. Execution then continues at the most recently defined exception handler capable of dealing with the exception. See section 1.5.3.

Flat container a non-hierarchical data structure representing a collection of values (elements). For example, arrays and lists.
Fixed point an int and a (possibly implicit) scaling. Used to represent real-valued numbers \( x \in \mathbb{R} \) approximately, with a constant absolute error.

Float a type which, in OCaml, represents a double-precision IEEE floating-point number.

Floating point a number representation commonly used to approximate real-valued numbers \( x \in \mathbb{R} \). See section 4.1.2.

Folds a higher-order function which applies its function argument to an accumulator and each element of a container. Introduced on page 36.

Function a mapping from input values to output values which may be described implicitly as an algorithm or explicitly, e.g. by a pattern match.

Functional programming a style of programming in which a computation is performed by composing the results of expressions without side-effects.

Functional language any programming language which allows functions to be passed as arguments to other functions, returned as the result of functions and stored as values in data structures.

Garbage collection the process of identifying data which are no longer accessible to a running program, destroying them and reclaiming the resources they required.

Generic programming the use of polymorphic functions and types.

Graph a data structure composed of vertices, and edges which link pairs of vertices.

Hash a constant-sized datum computed from an arbitrarily complicated value.

Hash table a data structure providing fast random access to its elements via their associated hash values.

Head the element at the front of a list.

Higher-order function any function which accepts another function as an argument. For example, \( f \) is a higher-order function in the definition \( f(g, x) = g(g(x)) \) because \( g \) must be a function (as \( g \) is applied to \( x \) and then to \( g(x) \)).

Heterogeneous container a data structure capable of storing several elements of different types. See section 3.8.

Homogeneous container a data structure (container) capable of storing several elements of the same type. For example, an array of integers is a homogeneous container because an array is a data structure containing elements of a single type, in this case integers.

Imperative programming a style of programming in which the result of a computation is generated by statements which act by way of side-effects, as opposed to functional programming.
Iteration a homonym with different meanings in different contexts and disciplines. In the context of numerical algorithms, an "iterative algorithm" means an algorithm designed to produce an approximate result by progressively converging on the solution. More generally, the word iterative is often used to describe repetitive algorithms, where a single repeat is known as an iteration.

Impure functional language a language, such as OCaml, which provides both functional and imperative programming constructs.

Int a type which exactly represents a contiguous subset of the integers \( \mathbb{Z} \). See section 4.1.1.

IO input and output operations, such as printing to the screen or reading from disc.

Leaf in the context of tree data structures, a leaf node is a node containing no remaining trees.

Lex converting a character stream into a token stream. For example, recognising the keywords in a language before parsing them.

Linked List see list.

List a flat container providing prepend and decapitation operators in \( O(1) \) time-complexity. In OCaml, these are performed by the :: operator, known as the cons operator. A list is traversed by repeated decapitation. See section 3.3.

Maps either a container or a higher-order function:

- A data structure implementing a container which allows key-values pairs to be inserted and keys to be subsequently mapped onto their corresponding values. See sections 3.5 and 3.6.
- A higher-order function \( \text{map } f \{ l_0, \ldots, l_{n-1} \} \rightarrow \{ f(l_0), \ldots, f(l_{n-1}) \} \) which acts upon a container of elements to create a new container, the elements of which are the result of applying the given function \( f \) to each element \( l_i \) in the given container. Sometimes known as inner map.

Module a construct which encapsulates definitions in a structure and, optionally, allow the externally-visible portion of the definitions to be restricted to a subset of the definitions by way of a signature. See section 2.3.

Module signature a module interface, declaring the types, exceptions, variables and functions which are to be accessible to code outside a module using the signature. See section 2.3.1.

Module structure the body of a module, containing definitions of the constituent types, exceptions, variables and functions which make up the module. See section 2.3.2.

Monomorphic a single, possibly not-yet-known, type. See section A.5.

Mutable can be altered.
Native code  the result of compiling a program into the machine language (machine code) un­
derstood natively by the CPU. Section 1.4.3 describes how to compile OCaml programs to native code.

Object-oriented programming  the creation of objects, which encapsulate functions and data, at run-time. In particular, the use of inheritance to specify relationships between types of object.

Parse  the act of understanding something formally. Parsing often refers to the recognition of grammatical constructs. See section 5.4.2.

Partial specialisation  the specialisation of a program or function to part of its input data. For example, given a function to compute $x^n$ for any given floating-point number $x$ and integer $n$, generating a function to compute $x^3$ for any floating-point number $x$ is partial specialising the original to $n = 3$.

Pattern matching  a construct in a programming language which allows patterns to be found and extracted from data structures.

Persistence  the ability to reuse old data structures without having to worry about undoing state changes and unwanted interactions. An advantage of functional programming.

Platform  a CPU architecture (e.g. ARM, MIPS, AMD) and operating system (e.g. IRIX, Linux, Mac OS X).

Polymorphic  one of any type. In particular, polymorphic functions are generic over the types of at least one of their arguments. Variant types can be generic over polymorphic type-arguments. See section 1.5.4.

Primitive operation  a low-level function or operation, used to formulate the time-complexity of an algorithm. See section 3.1.1.

Record  a tuple with named fields. For example, a record of type: 

```plaintext
{x: float; y: float}
```

can have a value `{ x=1.; y=2. }.

Regular Expression  a form of pattern matching.

Regexp  common abbreviation of regular expression.

Root  in the context data structures, the root is the origin of the data structure, from which all other portions may be accessed.

Run-time  while a program is being executed.

Side-effect  any result of an expression apart from the value which the expression returns, e.g. altering a mutable variable or performing IO.

Signature  see Module signature.

Source code  the initial, manually-entered form of a program. For example, the source code to the FFTW library (for computing Fast Fourier Transforms) is written in OCaml. This OCaml code can be compiled and run to generate C code which can, in turn, be compiled into a library and linked into a final program.
Static typing completely type checking at compile-time such that no type checking is required at run-time.

Structure see Module structure.

Tail the remainder of a list without its front element.

Time-complexity complexity of the time taken to execute an algorithm, specified as the number of times a set of primitive operations are performed.

Top-level an interactive OCaml interpreter started by running the `ocaml` program. See section 1.4.1.

Tree a recursive data structure represented by nodes which may contain further trees. The root node is the node from which all others may be reached. Leaf nodes are those which contain no further trees. Trees are traversed by examining the child nodes of the current node recursively.

Tuple a type representing elements in the set of the cartesian cross product of the sets of types in the tuple. For example, the 2-tuple of floating-point numbers \((x, y)\) of the type `float * float` is typically used to represent the set \(\mathbb{R} \times \mathbb{R}\).

Type the set of possible values of a variable, function argument or result, or the mapping between argument and result types of a function.

Variant type explicitly listed sets of possible values. See section 1.5.1.3.

Glossary of acronyms

AST Abstract-syntext tree

BNF Backus-Naur form

CAML Categorical abstract machine language

FFT Fast Fourier transform

FFTW Fastest Fourier transform in the west

GOE Gaussian orthogonal ensemble

INRIA Institut National de Recherche en Informatique et en Automatique

IO Input and output

LCF Logic of computable functions

MEM Maximum entropy method

ML Meta-language

OCaml Objective CAML
CONTENTS

OO  Object-oriented
OOP  Object-oriented programming
OpenGL  Open graphics library
SGI  Silicon Graphics Incorporated
VM  Virtual machine
XML  Extensible markup language