Introduction to the Objective Caml Programming Language

Jason Hickey

May 27, 2006

Contents

Chapter 1

Introduction

This document is an introduction to ML programming, specifically for the Objective Caml (*OCaml*) programming language from INRIA [3, 5]. OCaml is a dialect of the ML (*Meta-Language*) family of languages, which derive from the Classic ML language designed by Robin Milner in 1975 for the LCF (*Logic of Computable Functions*) theorem prover [2].

OCaml shares many features with other dialects of ML, and it provides several new features of its own. Throughout this document, we use the term ML to stand for any of the dialects of ML, and OCaml when a feature is specific to OCaml.

- ML is a **functional** language, meaning that functions are treated as first-class values. Functions may be nested, functions may be passed as arguments to other functions, and functions can be stored in data structures. Functions are treated like their mathematical counterparts as much as possible. Assignment statements that permanently change the value of certain expressions are permitted, but used much less frequently than in languages like C or Java.

- ML is **strongly typed**, meaning that the type of every variable and every expression in a program is determined at compile-time. Programs that pass the type checker are *safe*: they will never "go wrong" because of an illegal instruction or memory fault.

- Related to strong typing, ML uses **type inference** to infer types for the expressions in a program. Even though the language is strongly typed, it is rare that the programmer has to

annotate a program with type constraints.

- The ML type system is **polymorphic**, meaning that it is possible to write programs that work for values of any type. For example, it is straightforward to define data structures like lists, stacks, and trees that can contain elements of any type. In a language like C or Java, the programmer would either have to write different implementations for each type (say, lists of integers *vs.* lists of floating-point values), or else use explicit coercions to bypass the type system.

- ML implements a **pattern matching** mechanism that unifies case analysis and data destructors.

- ML includes an expressive **module system** that allows data structures to be specified and defined *abstractly*. The module system includes *functors*, which are are functions over modules that can be used to produce one data structure from another.

- OCaml is also the only widely-available ML implementation to include an **object system**. The module system and object system complement one another: the module system provides data abstraction, and the object system provides inheritance and re-use.

- OCaml includes a compiler that supports **separate compilation**. This makes the development process easier by reducing the amount of code that must be recompiled when a program is modified. OCaml actually includes two compilers: a *byte-code* compiler that produces code for the portable OCaml byte-code interpreter, and a *native-code* compiler that produces efficient code for many machine architectures.

- One other feature should be mentioned: all the languages in the ML family have a **formal semantics**, which means that programs have a mathematical interpretation, making the programming language easier to understand and explain.

1.1 Functional and imperative languages

```
/*                                  (*
 * A C function to                   * An OCaml function to
 * determine the greatest            * determine the greatest
 * common divisor of two             * common divisor of two
 * positive numbers a and b.         * positive numbers a and b.
 * We assume a>b.                    * We assume a>b.
 */                                  *)
int gcd(int a, int b)               let rec gcd a b =
{                                      let r = a mod b in
   int r;                                 if r = 0 then
                                             b
   while((r = a % b) != 0) {              else
      a = b;                                 gcd b r
      b = r;
   }
   return b;
}
```

Figure 1.1: C is an imperative programming language, while OCaml is functional. The code on the left is a C program to compute the greatest common divisor of two natural numbers. The code on the right is equivalent OCaml code, written functionally.

The ML languages are *semi-functional*, which means that the normal programming style is functional, but the language includes assignment and side-effects.

To compare ML with an imperative language, a comparison of two simple implementations of Euclid's algorithm is shown in Figure 1.1.

In a language like C, the algorithm is normally implemented as a loop, and progress is made by modifying the state. Reasoning about this program requires that we reason about the program state: give an invariant for the loop, and show that the state makes progress on each step toward the goal.

In OCaml, Euclid's algorithm is normally implemented using recursion. The steps are the same,

but there are no side-effects. The `let` keyword specifies a definition, the `rec` keyword specifies that the definition is recursive, and the `gcd a b` defines a function with two arguments a and b.

In ML, programs rarely use assignment or side-effects except for I/O. Functional programs have some nice properties: one is that data structures are *persistent* (by definition), which means that no data structure is ever destroyed.

There are problems with taking too strong a stance in favor of functional programming. One is that every updatable data structure has to be passed as an argument to every function that uses it (this is called *threading* the state). This can make the code obscure if there are too many of these data structures. We take a moderate approach. We use imperative code when necessary, but its use is discouraged.

1.2 Organization

This document is organized as a *user guide* to programming in OCaml. It is not a reference manual: there is already an online reference manual. I assume that the reader already has some experience using an imperative programming language like C; I'll point out the differences between ML and C in the cases that seem appropriate.

1.3 Additional Sources of Information

This document was originally used for a course in compiler construction at Caltech. The course material, including exercises, is available at `http://www.cs.caltech.edu/courses/cs134/cs134b`.

The OCaml reference manual [3] is available on the OCaml home page `http://www.ocaml.org/`.

The author can be reached at `jyh@cs.caltech.edu`.

Chapter 2

Simple Expressions

Many functional programming implementations include a runtime environment that defines a standard library and a garbage collector. They also often include a toploop evaluator that can be used to evaluate programs interactively.

OCaml provides a compiler, a runtime, and a toploop. By default, the toploop is called `ocaml`. The toploop prints a prompt (`#`), reads an input expression, evaluates it, and prints the result . Expressions in the toploop are terminated by a double-semicolon '`;;`'.

```
% ocaml
        Objective Caml version 3.08.0
# 1 + 4;;
- : int = 5
#
```

On startup, the `ocaml` toploop prints its version number, then prompts for input with the `#` character. Given an expression ($1 + 4$ in this case), the toploop evaluates the expression, prints the type of the result (`int`) and the value (5). To exit the toploop, you may type the end-of-file character (usually Control-D in Unix, and Control-Z in Microsoft Windows).

2.1 Comment convention

In OCaml, comments are enclosed in matching `(*` and `*)` pairs. Comments may be nested, and

the comment is treated as white space.

2.2 Basic expressions

OCaml is a *strongly typed* language. In OCaml every valid expression must have a type, and expressions of one type may not be used as expressions in another type. Apart from polymorphism, which we discuss in Chapter 5.1, there are no implicit coercions. Normally, you do not have to specify the types of expressions. OCaml uses *type inference* [1] to figure out the types for you.

The primitive types are `unit`, `int`, `char`, `float`, `bool`, and `string`.

2.2.1 `unit`: the singleton type

The simplest type in OCaml is the `unit` type, which contains one element: `()`. This type seems to be a rather silly. However, in a functional language every function must return a value. The `()` value is commonly used as the value of a procedure that computes by side-effect. The `unit` type corresponds to the `void` type in C.

2.2.2 `int`: the integers

The `int` type is the type of signed integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The precision is finite. Integer values are represented by a machine word, minus one bit that is reserved for use by the garbage collector, so on a 32-bit machine architecture, the precision is 31 bits (one bit is reserved for use by the runtime), and on a 64-bit architecture, the precision is 63 bits.

Integers are usually specified in decimal, but there are several alternate forms. In the following table the symbol d denotes a decimal digit (‘`0`’..‘`9`’); o denotes an octal digit (‘`0`’..‘`7`’); b denotes a binary digit (‘`0`’ or ‘`1`’); and h denotes a hexadecimal digit (‘`0`’..‘`9`’, or ‘`a`’..‘`f`’, or ‘`A`’..‘`F`’).

ddd...	an `int` specified in decimal.
`0`*oooo*...	an `int` specified in octal.
`0b`*bbb*...	an `int` specified in binary.
`0x`*hhh*...	an `int` specified in hexadecimal.

There are the usual operations on `int`s, including arithmetic and bitwise operations.

`-`*i* or `~-`*i*	negation.
i `+` *j*	addition.
i `-` *j*	subtraction.
i `*` *j*	multiplication.
i `/` *j*	division.
i `mod` *j*	remainder.
`lnot` *i*	bitwise-inverse.
i `lsl` *j*	logical shift left $i \cdot 2^j$.
i `lsr` *j*	logical shift right $i \div 2^j$ (*i* is treated as an unsigned twos-complement number).
i `asl` *j*	arithmetic shift left $i \cdot 2^j$.
i `asr` *j*	arithmetic shift right $\lfloor i \div 2^j \rfloor$ (the sign of *i* is preserved).
i `land` *j*	bitwise-and.
i `lor` *j*	bitwise-or.
i `lxor` *j*	bitwise exclusive-or.

The precedences of the integer operators are as follows, listed in increasing order.

Operators	Associativity
`+`, `-`	left
`*`, `/`, `mod`, `land`, `lor`, `lxor`	left
`lsl`, `lsr`, `asr`	right
`lnot`	left
`~-`, `-`	right

Here are some example expressions.

```
# 0b1100;;
- : int = 12
# 3 + 4 * 5;;
- : int = 23
# 0x100 lsl (2 + 6);;
- : int = 65536
```

2.2.3 `float`: the floating-point numbers

The type `float` specifies dynamically scaled "floating point" numbers. The syntax of a floating point number includes either a decimal point, or an exponent (base 10) denoted by an '`E`' or '`e`'. A digit is required before the decimal point. Here are a few examples:

0.2, 2e7, 3.1415926, 31.415926E-1

The integer arithmetic operators (`+`, `-`, `*`, `/`, ...) *do not work* with floating point values. The corresponding operators include a '.' as follows:

`-.`x or `~-.`x	floating-point negation
x `+.` y	floating-point addition.
x `-.` y	floating-point subtraction.
x `*.` y	float-point multiplication.
x `/.` y	floating-point division.
`int_of_float` x	`float` to `int` conversion.
`float_of_int` i	`int` to `float` conversion.

Here are some example floating-point expressions.

```
# 31.415926e-1;;
- : float = 3.1415926
# float_of_int 1;;
- : float = 1.
# int_of_float 1.2;;
- : int = 1
# 3.1415926 *. 17.2;;
- : float = 54.03539272
# 1 + 2.0;;
Characters 4-7:
  1 + 2.0;;
      ^^^
This expression has type float but is here used with type int
```

The final expression fails to typecheck because the operator `+`, which works only with `int` expressions, is used with a floating-point expression `2.0`.

2.2.4 `char`: the characters

The character type `char` specifies characters from the ASCII character set. The syntax for a character constants uses the single quote symbol *'c'*.

```
'a', 'Z', ' ', 'W'
```

In addition, there are several kinds of escape sequences with an alternate syntax. Each escape sequence begins with the backslash character '\'.

`'\'`	The backslash character itself.
`'\''`	The single-quote character.
`'\t'`	The tab character.
`'\r'`	The carriage-return character.
`'\n'`	The newline character.
`'\b'`	The backspace character.
'\ddd'	A decimal escape sequence.
'\xhh'	A hexadecimal escape sequence.

A decimal escape sequence must have exactly three decimal characters *d*, and specifies the ASCII character with the specified decimal code. A hexadecimal escape sequence must have exactly two hexadecimal characters *h*.

```
'a', 'Z', '\120', '\t', '\n', '\x7e'
```

There are functions for converting between characters and integers. The function `Char.code` returns the integer corresponding to a character, and `Char.chr` returns the character with the given ASCII code. The `Char.lowercase` and `Char.uppercase` functions give the equivalent lower- or upper-case characters.

```
# '\120';;
- : char = 'x'
# Char.code 'x';;
- : int = 120
# '\x7e';;
- : char = '~'
# Char.uppercase 'z';;
- : char = 'Z'
# Char.uppercase '[';;
- : char = '['
# Char.chr 32;;
- : char = ' '
```

2.2.5 `string`: character strings

In OCaml, character strings belong to a primitive type `string`. Unlike strings in C, character strings are not arrays of characters, and they do not use the null-character `'\000'` for termination.

The syntax for strings uses the double-quote symbol '`"`' as a delimiter. Characters in the string may use the escape sequences defined for characters.

```
"Hello"
"The character '\000' is not a terminator"
"\072\105"
```

The `^` operator performs string concatenation.

```
# "Hello " ^ " world\n";;
- : string = "Hello world\n"
# "The character '\000' is not a terminator";;
- : string = "The character '\000' is not a terminator"
# "\072\105";;
- : string = "Hi"
```

Strings also allow random access. The expression `s.[i]` returns the i'th character from string s; and the expression `s.[i]<- c` replaces the i'th in string s by character c, returning a `unit` value. The `String` module (see Section **??**) also defines many functions to manipulate strings, including the `String.length` function, which returns the length of a string; and the `String.sub` function, which returns a substring.

```
# "Hello".[1];;
- : char = 'e'
# "Hello".[0] <- 'h';;
- : unit = ()
# String.length "Abcd\000";;
- : int = 5
# String.sub "Ab\000cd" 1 3;;
- : string = "b\000c"
```

2.2.6 `bool`: the Boolean values

The `bool` type is used to represent the Boolean values `true` and `false`. Logical negation of Boolean values is performed by the `not` function.

There are several relations that can be used to compare values, returning `true` if the comparison holds and `false` otherwise.

x `=` y	x is equal to y.
x `==` y	x is "identical" to y.
x `<>` y	x is not equal to y.
x `<` y	x is less than y.
x `<=` y	x is no more than y.
x `>=` y	x is no less than y.
x `>` y	x is greater than y.

These relations operate on two values x and y having equal but arbitrary type. For the primitive types in this chapter, the comparison is what you would expect. For values of other types, the value is implementation-dependent, and in some cases may raise a runtime error. For example, functions (discussed in the next chapter) cannot be compared.

The `==` comparison deserves special mention, since we use the word "identical" in an informal sense. The exact semantics is this: if the expression "x `==` y" evaluates to `true`, then so does the

expression "x `=` y". However it is still possible for "x `=` y" to be `true` even if "x `==` y" is not. In the OCaml implementation from INRIA, the expression "x `==` y" evaluates to `true` only if the two values x and y are exactly the same value. The comparison `==` is a constant-time operation that runs in a bounded number of machine instructions; the comparison `=` is not.

```
# 2 < 4;;
- : bool = true
# "A good job" > "All the tea in China";;
- : bool = false
# 2 + 6 = 8;;
- : bool = true
# 1.0 = 1.0;;
- : bool = true
# 1.0 == 1.0;;
- : bool = false
# 2 == 1 + 1;;
- : bool = true
```

Strings are compared lexicographically (in alphabetical-order), so the second example is `false` because the character '`l`' is greater than the space-character '` `' in the ASCII character set. The comparison "`1.0 == 1.0`" in this case returns `false` (because the 2 floating-point numbers were typed separately), but it performs normal comparison on `int` values.

There are two logical operators: `&&` is conjunction (and), and `||` is disjunction (or). Both operators are the "short-circuit" versions: the second clause is not evaluated if the result can be determined from the first clause.

```
# 1 < 2 || (1 / 0) > 0;;
- : bool = true
# 1 < 2 && (1 / 0) > 0;;
Exception: Division_by_zero.
# 1 > 2 && (1 / 0) > 0;;
- : bool = false
```

Conditionals are expressed with the syntax `if` b `then` e_1 `else` e_2.

```
# if 1 < 2 then
     3 + 7
  else
     4;;
- : int = 10
```

2.3 Operator precedences

The precedences of the operators in this section are as follows, listed in increasing order.

Operators	Associativity
`||`	left
`&&`	left
`=`, `==`, `!=`, `<>`, `<`, `<=`, `>`, `>=`	left
`+`, `-`, `+.`, `-.`	left
`*`, `/`, `*.`, `/.`, `mod`, `land`, `lor`, `lxor`	left
`lsl`, `lsr`, `asr`	right
`lnot`	left
`~-`, `-`, `~-.`, `-.`	right

2.4 The OCaml type system

The ML languages are statically and strictly typed. In addition, every expression has a exactly one type. In contrast, C is a weakly-typed language: values of one type can usually be coerced to a value of any other type, whether the coercion makes sense or not. Lisp is not a statically typed language: the compiler (or interpreter) will accept any program that is syntactically correct; the types are checked at run time. The type system is not necessarily related to safety: both Lisp and ML are *safe* languages, while C is not.

What is "safety?" There is a formal definition based on the operational semantics of the programming language, but an approximate definition is that the execution of a valid program will never fail because of an invalid machine operation. All memory accesses will be valid. ML guarantees safety by proving that every program that passes the type checker can never produce a machine fault, and Lisp guarantees it by checking for validity at run time. One surprising (some would say annoying) consequence is that ML has no `nil` or `NULL` values; these would potentially cause machine errors if used where a value is expected.

As you learn OCaml, you will initially spend a lot of time getting the OCaml type checker to accept your programs. Be patient, you will eventually find that the type checker is one of your best friends. It will help you figure out where errors may be lurking in your programs. If you make a change, the type checker will help track down the parts of your program that are affected.

In the meantime, here are some rules about type checking.

1. Every expression has exactly one type.

2. When an expression is evaluated, one of four things may happen:

   (a) it may evaluate to a *value* of the same type as the expression,

   (b) it may raise an exception (we'll discuss exceptions in Chapter 8),

   (c) it may not terminate,

   (d) it may exit.

One of the important points here is that there are no "pure commands." Even assignments produce a value (although the value has the trivial `unit` type).

To begin to see how this works, let's look at the conditional expression.

```
<kenai 229>cat -b x.ml
    1 if 1 < 2 then
    2    1
    3 else
    4    1.3
<kenai 230>ocamlc -c x.ml
File "x.ml", line 4, characters 3-6:
This expression has type float but is here used with type int
```

This error message seems rather cryptic: it says that there is a type error on line 4, characters 3-6 (the expression `1.3`). The conditional expression evaluates the test. If the test is `true`, it evaluates the first branch. Otherwise, it evaluates the second branch. In general, the compiler doesn't try to figure out the value of the test during type checking. Instead, it requires that both branches of the conditional have the same type (so that the value will have the same type no matter how the test turns out). Since the expressions `1` and `1.3` have different types, the type checker generates an error.

One other point to mention—the `else` branch is not required in a conditional. If it is omitted, the conditional is treated as if the `else` case returns the `()` value. The following code has a type error.

```
% cat -b y.ml
    1 if 1 < 2 then
    2    1
% ocamlc -c y.ml
File "y.ml", line 2, characters 3-4:
This expression has type int but is here used with type unit
```

In this case, the expression `1` is flagged as a type error, because it does not have the same type as the omitted `else` branch.

2.5 Compiling your code

You aren't required to use the toploop for all your programs. In fact, as your programs become larger, you will begin to use the toploop less, and rely more on the OCaml compilers. Here is a brief introduction to using the compiler; more information is given in the Chapter 10.

If you wish to compile your code, you should place it in a file with the `.ml` suffix. In INRIA OCaml, there are two compilers: `ocamlc` compiles to byte-code, and `ocamlopt` compiles to native machine code. The native code is several times faster, but compile time is longer. The usage is similar to `cc`. The double-semicolon terminators are not necessary in `.ml` source files; you may omit them if the source text is unambiguous.

- To compile a single file, use `ocamlc -g -c` *`file`*`.ml`. This will produce a file *`file`*`.cmo`. The `ocamlopt` programs produces a file *`file`*`.cmx`. The `-g` option is valid only for `ocamlc`; it causes debugging information to be included in the output file.

- To link together several files into a single executable, use `ocamlc` to link the `.cmo` files. Normally, you would also specify the `-o` *`program_file`* option to specify the output file (the default is `a.out`). For example, if you have two program files `x.cmo` and `y.cmo`, the command would be:

```
% ocamlc -g -o program x.cmo y.cmo
% ./program
...
```

There is also a debugger `ocamldebug` that you can use to debug your programs. The usage is a lot like `gdb`, with one major exception: execution can go backwards. The `back` command will go

back one instruction.

Chapter 3

Variables and Functions

So far, we have considered only simple expressions not involving variables. In ML, variables are *names* for values. Variable bindings are introduced with the `let` keyword. The syntax of a simple top-level definition is as follows.

`let` *name* `=` *expr*

For example, the following code defines two variables x and y and adds them together to get a value for z.

```
# let x = 1;;
val x : int = 1
# let y = 2;;
val y : int = 2
# let z = x + y;;
val z : int = 3
```

Definitions using `let` can also be nested using the `in` form.

`let` *name* `=` *expr1* `in` *expr2*

The expression *expr2* is called the *body* of the `let`. The variable *name* is defined as the value of *expr1* within the body. The variable named *name* is defined only in the body *expr2* and not *expr1*.

`Let`s with a body are expressions; the value of a `let` expression is the value of the body.

```
# let x = 1 in
  let y = 2 in
     x + y;;
- : int = 3
# let z =
     let x = 1 in
     let y = 2 in
        x + y;;
val z : int = 3
```

Binding is static (lexical scoping), meaning that the value associated with a variable is determined by the nearest enclosing definition in the program text. For example, when a variable is defined in a `let` expression, the defined value is used within the body of the let (or the rest of the file for toplevel `let` definitions). If the variable was defined previously, the previous value is shadowed, meaning that it becomes inaccessible while the new definition is in effect.

For example, consider the following program, where the variable `x` is initially defined to be 7. Within the definition for `y`, the variable `x` is redefined to be 2. The value of `x` in the final expression `x + y` is still 7, and the final result is 10.

```
# let x = 7 in
  let y =
     let x = 2 in
        x + 1
  in
     x + y;;
- : int = 10
```

Similarly, the value of `z` in the following program is 8, because of the definitions that double the value of `x`.

```
# let x = 1;;
val x : int = 1
# let z =
     let x = x + x in
     let x = x + x in
        x + x;;
val z : int = 8
# x;;
- : int = 1
```

3.1 Functions

Functions are defined with the `fun` keyword.

$$\texttt{fun}\ v_1\ v_2\ \cdots\ v_n\ \texttt{->}\ expr$$

The `fun` is followed by a sequence of variables that define the formal parameters of the function, the `->` separator, and then the body of the function *expr*. By default, functions are anonymous, which is to say that they are not named. In ML, functions are values like any other. Functions may be constructed, passed as arguments, and applied to arguments, and, like any other value, they may be named by using a `let`.

```
# let increment = fun i -> i + 1;;
val increment : int -> int = <fun>
```

Note the type `int -> int` for the function. The arrow `->` stands for a *function type*. The type before the arrow is the type of the function's argument, and the type after the arrow is the type of the result. The `increment` function takes an argument of type `int`, and returns a result of type `int`.

The syntax for function application (function call) is concatenation: the function is followed by its arguments. The precedence of function application is higher than most operators. Parentheses are needed for arguments that are not simple expressions.

```
# increment 2;;
- : int = 3
# increment 2 * 3;;
- : int = 9
# increment (2 * 3);;
- : int = 7
```

Functions may also be defined with multiple arguments. For example, a function to compute the sum of two integers might be defined as follows.

```
# let sum = fun i j -> i + j;;
val sum : int -> int -> int = <fun>
# sum 3 4;;
- : int = 7
```

Note the type for `sum`: `int -> int -> int`. The arrow associates to the right, so this type is the same as `int -> (int-> int)`. That is, `sum` is a function that takes a single integer argument, and returns a function that takes another integer argument and returns an integer. Strictly speaking, all functions in ML take a single argument; multiple-argument functions are implemented as *nested* functions (this is called "Currying," after Haskell Curry, a famous logician who had a significant impact on the design and interpretation of programming languages). The definition of `sum` above is

equivalent to the following explicitly-curried definition.

```
# let sum = (fun i -> (fun j -> i + j));;
val sum : int -> int -> int = <fun>
# sum 4 5;;
- : int = 9
```

The application of a multi-argument function to only one argument is called a "partial application."

```
# let incr = sum 1;;
val incr : int -> int = <fun>
# incr 5;;
- : int = 6
```

Since named functions are so common, OCaml provides an alternate syntax for functions using a `let` definition. The formal parameters of the function are listed after to the function name, before the equality symbol.

$$\texttt{let}\ \mathit{name}\ v_1\ v_2\ \cdots\ v_n = \mathit{expr}$$

For example, the following definition of the `sum` function is equivalent to the ones above.

```
# let sum i j =
     i + j;;
val sum : int -> int -> int = <fun>
```

3.1.1 Scoping and nested functions

Functions may be arbitrarily nested. They may also be passed as arguments. The rule for scoping uses static binding: the value of a variable is determined by the code in which a function is defined—not by the code in which a function is evaluated. For example, another way to define `sum` is as follows.

```
# let sum i =
     let sum2 j =
        i + j
     in
        sum2;;
val sum : int -> int -> int = <fun>
# sum 3 4;;
- : int = 7
```

To illustrate the scoping rules, let's consider the following definition.

```
# let i = 5;;
val i : int = 5
# let addi j =
     i + j;;
val addi : int -> int = <fun>
# let i = 7;;
val i : int = 7
# addi 3;;
- : val = 8
```

In the `addi` function, the value of `i` is defined by the previous definition of `i` as 5. The second definition of `i` has no effect on the definition for `addi`, and the application of `addi` to the argument 3 results in $3 + 5 = 8$.

3.1.2 Recursive functions

Suppose we want to define a recursive function: that is, a function that is used in its own function body. In functional languages, recursion is used to express repetition or looping. For example, the "power" function that computes x^i might be defined as follows.

```
# let rec power i x =
     if i = 0 then
        1.0
     else
        x *. (power (i - 1) x);;
val power : int -> float -> float = <fun>
# power 5 2.0;;
- : float = 32
```

Note the use of the `rec` modifier after the `let` keyword. Normally, the function is not defined in its own body. The following definition is rejected.

```
# let power_broken i x =
     if i = 0 then
        1.0
     else
        x *. (power_broken (i - 1) x);;

Characters 70-82:
        x *. (power_broken (i - 1) x);;
              ^^^^^^^^^^^^
Unbound value power_broken
```

Mutually recursive definitions (functions that call one another) can be defined using the `and`

keyword to connect several `let` definitions.

```
# let rec f i j =
     if i = 0 then
        j
     else
        g (j - 1)
  and g j =
     if j mod 3 = 0 then
        j
     else
        f (j - 1) j;;
val f : int -> int -> int = <fun>
val g : int -> int = <fun>
# g 5;;
- : int = 3
```

3.1.3 Higher order functions

Let's consider a definition where a function is passed as an argument, and another function is returned as a result. Given an arbitrary function f on the real numbers, a numerical derivative is defined approximately as follows.

```
# let dx = 1e-10;;
val dx : float = 1e-10
# let deriv f =
     (fun x -> (f (x +. dx) -. f x) /. dx);;
val deriv : (float -> float) -> float -> float = <fun>
```

Remember, the arrow associates to the right, so another way to write the type is `(float -> float) -> (float -> float)`. That is, the derivative is a function that takes a function as an argument, and returns a function.

Let's apply the `deriv` function to the `power` function defined above, partially applied to the argument 3.

```
# let f = power 3;;
val f : float -> float = <fun>
# f 10.0;;
- : float = 1000
# let f' = deriv f;;
val f' : float -> float = <fun>
# f' 10.0;;
- : float = 300.000237985
# f' 5.0;;
- : float = 75.0000594962
# f' 1.0;;
- : float = 3.00000024822
```

As we would expect, the derivative of x^3 is approximately $3x^2$. To get the second derivative, we apply the `deriv` function to `f'`.

```
# let f'' = deriv f';;
val f'' : float -> float = <fun>
# f'' 0.0;;
- : float = 6e-10
# f'' 1.0;;
- : float = 0
# f'' 10.0;;
- : float = 0
```

The second derivative, which we would expect to be $6x$, is way off! Ok, there are some numerical errors here. Don't expect functional programming to solve all your problems.

```
# let g x = 3.0 *. x *. x;;
val g : float -> float = <fun>
# let g' = deriv g;;
val g' : float -> float = <fun>
# g' 1.0;;
- : float = 6.00000049644
# g' 10.0;;
- : float = 59.9999339101
```

3.2 Variable names

As you may have noticed in the previous section, the single quote symbol (') is a valid character in a variable name. In general, a variable name may contain letters (lower and upper case), digits, and the ' and _ characters. but it must begin with a lowercase letter or the underscore character, and it may not be the _ all by itself.

In OCaml, sequences of characters from the infix operators, like `+, -, *, /, ...` are also valid names. The normal prefix version is obtained by enclosing them in parentheses. For example, the following code is a proper entry for the Obfuscated ML contest. Don't use this style in your code.

```
# let (+) = ( * )
  and (-) = (+)
  and ( * ) = (/)
  and (/) = (-);;
val + : int -> int -> int = <fun>
val - : int -> int -> int = <fun>
val * : int -> int -> int = <fun>
val / : int -> int -> int = <fun>
# 5 + 4 / 1;;
- : int = 15
```

Note that the `*` operator requires space within the parenthesis. This is because of comment conventions—comments start with `(*` and end with `*)`.

The redefinition of infix operators may make sense in some contexts. For example, a program module that defines arithmetic over complex numbers may wish to redefine the arithmetic operators. It is also sensible to add new infix operators. For example, we may wish to have an infix operator for the `power` construction.

```
# let ( ** ) x i = power i x;;
val ** : float -> int -> float = <fun>
# 10.0 ** 5;;
- : float = 100000
```

The precedence and associativity of new infix operators is determined by its first character in the operator name. For example an operator named `+/-` would have the same precedence and associativity as the `+` operator.

Chapter 4

Basic Pattern Matching

One of ML's more powerful features is the use of *pattern matching* to define expressions by case analysis. Pattern matching is indicated by a `match` expression, which has the following syntax.

```
match expression with
    pattern_1 -> expression_1
  | pattern_2 -> expression_2
    ⋮
  | pattern_n -> expression_n
```

When a `match` expression is evaluated, it evaluates the expression *expression*, and compares the value with the patterns. If $pattern_i$ is the first pattern to match, then $expression_i$ is evaluated and returned as the result of the `match`.

A simple *pattern* is an expression made of constants and variables. A constant pattern c matches values that are equal to it, and a variable pattern x matches any expression. A variable pattern x is a binding occurrence; when the match is performed, the variable x is bound the the value being matched.

For example, Fibonacci numbers can be defined succinctly using pattern matching. Fibonacci numbers are defined inductively: `fib` $0 = 0$, `fib` $1 = 1$, and for all other natural numbers i, `fib` $i =$ `fib`$(i-1) +$ `fib`$(i-2)$.

```
# let rec fib i =
    match i with
       0 -> 0
     | 1 -> 1
     | j -> fib (j - 2) + fib (j - 1);;
val fib : int -> int = <fun>
# fib 1;;
- : int = 1
# fib 2;;
- : int = 1
# fib 3;;
- : int = 2
# fib 6;;
- : int = 8
```

In this code, the argument i is compared against the constants 0 and 1. If either of these cases match, the return value is equal to i. The final pattern is the variable j, which matches any argument. When this pattern is reached, j takes on the value of the argument, and the body `fib (j - 2) + fib (j - 1)` computes the returned value.

Note that variables occurring in a pattern are always binding occurrences. For example, the following code produces a result you might not expect. The first case matches all expressions, returning the value matched. The toploop issues a warning for the second and third cases.

```
# let zero = 0;;
# let one = 1;;
# let rec fib i =
     match i with
        zero -> zero
      | one -> one
      | j -> fib (j - 2) + fib (j - 1);;
Characters 57-60:
Warning: this match case is unused.
Characters 74-75:
Warning: this match case is unused.
      | one -> one
        ^^^
      | j -> fib (j - 2) + fib (j - 1);;
        ^
val fib : int -> int = <fun>
# fib 1;;
- : int = 1
# fib 10;;
- : int = 10
# fib 2002;;
- : int = 2002
```

4.1 Functions with matching

It is quite common for the body of an ML function to be a `match` expression. To simplify the syntax somewhat, OCaml defines the `function` keyword (instead of `fun`) to represent a function that is defined by pattern matching. A `function` definition is like a `fun`, where a single argument is used in a pattern match. The `fib` definition using `function` is as follows.

```
# let rec fib = function
     0 -> 0
   | 1 -> 1
   | i -> fib (i - 1) + fib (i - 2);;
val fib : int -> int = <fun>
# fib 1;;
- : int = 1
# fib 6;;
- : int = 8
```

4.2 Values of other types

Patterns can also be used with values having the other basic types, like characters, strings, and Boolean values. In addition, multiple patterns can be used for a single body. For example, one way to check for capital letters is with the following function definition.

```
# let is_uppercase = function
   'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H'
 | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P'
 | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X'
 | 'Y' | 'Z' ->
    true
 | c ->
    false;;
val is_uppercase : char -> bool = <fun>
# is_uppercase 'M';;
- : bool = true
# is_uppercase 'm';;
- : bool = false
```

It is rather tedious to specify all the letters one at a time. OCaml also allows pattern ranges $c_1..c_2$, where c_1 and c_2 are character constants.

```
# let is_uppercase = function
     'A' .. 'Z' -> true
   | c -> false;;
val is_uppercase : char -> bool = <fun>
# is_uppercase 'M';;
- : bool = true
# is_uppercase 'm';;
- : bool = false
```

Note that the pattern variable c in these functions acts as a "wildcard" pattern to handle all non-uppercase characters. The variable itself is not used in the body `false`. This is another commonly occurring structure, and OCaml provides a special pattern for cases like these. The _ pattern (a single underscore character) is a wildcard pattern that matches anything. It is not a variable (so it can't be used in an expression). The `is_uppercase` function would normally be written this way.

```
# let is_uppercase = function
     'A' .. 'Z' -> true
   | _ -> false;;
val is_uppercase : char -> bool = <fun>
# is_uppercase 'M';;
- : bool = true
# is_uppercase 'm';;
- : bool = false
```

The values being matched are not restricted to the basic scalar types like integers and characters. String matching is also supported, using the usual syntax.

```
# let names = function
     "first" -> "George"
   | "last" -> "Washington"
   | _ -> ""
val names : string -> string = <fun>
# names "first";;
- : string = "George"
# names "Last";;
- : string = ""
\end[iverbatim]

Matching against floating-point values, while supported, is rarely
used because of numerical issues.  The following example illustrates
the issue.

@begin[iverbatim]
# match 4.3 -. 1.2 with
     3.1 -> true
   | _ -> false;;
- : bool = false
```

4.3 Incomplete matches

You might wonder about what happens if the match expression does not include patterns for all the possible cases. For example, what happens if we leave off the default case in the `is_uppercase` function?

```
# let is_uppercase = function
     'A' .. 'Z' -> true;;
Characters 19-49:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
'a'
val is_uppercase : char -> bool = <fun>
```

The OCaml compiler and toploop are verbose about inexhaustive patterns. They warn when the pattern match is inexhaustive, and even suggest a case that is not matched. An inexhaustive set of patterns is usually an error—what would happen if we applied the `is_uppercase` function to a non-uppercase character?

```
# is_uppercase 'M';;
- : bool = true
# is_uppercase 'm';;
Uncaught exception: Match_failure("", 19, 49)
```

Again, OCaml is fairly strict. In the case where the pattern does not match, it raises an *exception* (we'll see more about exceptions in Chapter 8). In this case, the exception means that an error occurred during evaluation (a pattern matching failure).

A word to the wise: *heed the compiler warnings!* The compiler generates warnings for possible program errors. As you build and modify a program, these warnings will help you find places in the program text that need work. In some cases, you may be tempted to ignore the compiler. For example, in the following function, we know that a complete match is not needed if the `is_odd` function is always applied to nonnegative numbers.

```
# let is_odd i =
     match i mod 2 with
        0 -> false
      | 1 -> true;;
Characters 18-69:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
val is_odd : int -> bool = <fun>
# is_odd 3;;
- : bool = true
# is_odd 12;;
- : bool = false
```

However, *do not* ignore the warning! If you do, you will find that you begin to ignore *all* the compiler warnings—both real and bogus. Eventually, you will overlook real problems, and your

program will become hard to maintain. For now, you should add the default case that raises an exception manually. The `Invalid_argument` exception is designed for this purpose. It takes a string argument that is usually used to identify the name of the place where the failure occurred. You can generate an exception with the *raise* construction.

```
# let is_odd i =
     match i mod 2 with
        0 -> false
      | 1 -> true
      | _ -> raise (Invalid_argument "is_odd");;
val is_odd : int -> bool = <fun>
# is_odd 3;;
- : bool = true
# is_odd (-1);;
Uncaught exception: Invalid_argument("is_odd")
```

4.4 Patterns are everywhere

It may not be obvious at this point, but patterns are used in *all* the binding mechanisms, including the `let` and `fun` constructions. The general forms are as follows.

```
let pattern = expression
let name pattern ... pattern = expression
fun pattern -> expression
```

These forms aren't much use with constants because the pattern match will always be inexhaustive (except for the `()` pattern). However, they will be handy when we introduce tuples and records in the next chapter.

```
# let is_one = fun 1 -> true;;
Characters 13-26:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val is_one : int -> bool = <fun>
# let is_one 1 = true;;
Characters 11-19:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val is_one : int -> bool = <fun>
# is_one 1;;
- : bool = true
# is_one 2;;
Uncaught exception: Match_failure("", 11, 19)
# let is_unit () = true;;
val is_unit : unit -> bool = <fun>
# is_unit ();;
- : bool = true
```

Chapter 5

Tuples, Lists, and Polymorphism

In the chapters leading up to this one, we have seen simple expressions involving numbers, characters, strings, functions and variables. This language is already Turing complete—we can code arbitrary data types using numbers, functions, and string. Of course, in practice, this would not only be inefficient, it would also make it very hard to understand our programs. For efficient and readable data structure implementations we need to be able to structure and compose data.

OCaml provides a rich set of types for defining data structures, including tuples, lists, disjoint unions (also called tagged unions, or variant records), records, and arrays. In this chapter, we'll look at the simplest part of these—tuples and lists. We'll discuss unions in Chapter 6, and we'll leave the remaining types for Chapter 7, when we introduce side-effects.

5.1 Polymorphism

As we explore the type system, polymorphism will be one of the first concepts that we encounter. The ML languages provide *parametric polymorphism*. That is, types and expressions may be parameterized by type variables. For example, the identity function (the function that returns its argument) can be expressed in ML with a single function.

```
# let identity x = x;;
val identity : 'a -> 'a = <fun>
# identity 1;;
- : int = 1
# identity "Hello";;
- : string = "Hello"
```

Type variables are lowercase identifiers preceded by a single quote (`'`). A type variable represents an arbitrary type. The typing `identity : 'a -> 'a` says that the `identity` function takes an argument of some arbitrary type `'a` and returns a value of the same type `'a`. If the `identity` function is applied to a value with type `int`, then it returns a value of type `int`; if it is applied to a `string`, then it returns a `string`. The `identity` function can even be applied to function arguments.

```
# let succ i = i + 1;;
val succ : int -> int = <fun>
# identity succ;;
- : int -> int = <fun>
# (identity succ) 2;;
- : int = 3
```

In this case, the `(identity succ)` expression returns the `succ` function itself, which can be applied to `2` to return `3`.

5.1.1 Value restriction

What happens if we apply the `identity` to a *polymorphic* function type?

```
# let identity' = identity identity;;
val identity' : '_a -> '_a = <fun>
# identity' 1;;
- : int = 1
# identity';;
- : int -> int = <fun>
# identity' "Hello";;
Characters 10-17:
This expression has type string
but is here used with type int
```

This doesn't quite work as we expect. Note the type assignment `identity' : '_a -> '_a`. The type variables `'_a` are now preceded by an underscore. These type variables specify that the `identity'` function takes an argument of *some* (as yet unknown) type, and returns a value of the

same type. The `identity'` function is not truly polymorphic, because it can be used with values of only one type. When we apply the `identity'` function to a number, the type of the `identity'` function becomes `int -> int`, and it is no longer possible to apply it to a string.

This behavior is due to the *value restriction*: for an expression to be truly polymorphic, it must be a value. Values are immutable expressions that are not applications. For example, numbers and characters are values. Functions are also values. Function applications, like `identity identity` are *not* values, because they can be simplified (the `identity identity` expression evaluates to `identity`).

Why does OCaml have this restriction? It probably seems silly, but the value restriction is a simple way to maintain correct typing in the presence of side-effects. For example, suppose we had two functions `set : 'a -> unit` and `get : unit -> 'a` that share a storage location. The intent is that the function `get` should return the last value that was saved with `set`. That is, if we call `set 10`, then `get ()` should return the `10` (of type `int`). However, the type `get : unit ->'a` is clearly too permissive. It states that `get` returns a value of arbitrary type, no matter what value was saved with `set`.

The solution here is to use the restricted types `set : '_a ->unit` and `get : unit -> '_a`. In this case, the `set` and `get` functions can be used only with values of a single type. Now, if we call `set 10`, the type variable `'_a` becomes `int`, and the type of the `get` function becomes `unit-> int`.

The general principle of the value restriction is that mutable values are not polymorphic. In addition, applications are not polymorphic because the function might create a mutable value, or perform an assignment. This is the case even for simple applications like `identity identity` where it is obvious that no assignments are being performed.

However, it is usually easy to get around the value restriction by using a technique called *eta-expansion*. Suppose we have an expression `e` of function type. The expression `(fun x ->e x)` is nearly equivalent—in fact, it is equivalent if `e` does not contain side-effects. The expression `(fun x -> e x)` is a function, so it is a value, and it may be polymorphic. Consider this redefinition of the `identity'` function.

```
# let identity' = (fun x -> (identity identity) x);;
val identity' : 'a -> 'a = <fun>
# identity' 1;;
- : int = 1
# identity' "Hello";;
- : string = "Hello"
```

The new version of `identity'` computes the same value as the previous definition of `identity'`, but now it is properly polymorphic.

5.1.2 Other kinds of polymorphism

Polymorphism can be a powerful tool. In ML, a single identity function can be defined that works on all types. In a non-polymorphic language like C, a separate identity function would have to be defined for each type.

```
int int_identity(int i)
{
   return i;
}

struct complex { float real; float imag; };

struct complex complex_identity(struct complex x)
{
   return x;
}
```

Overloading

Another kind of polymorphism present in some languages is *overloading* (also called *ad-hoc* polymorphism). Overloading allows functions definitions to have the same name if they have different parameter types. When an application is encountered, the compiler selects the appropriate function by comparing the available functions against the type of the arguments. For example, in Java we could define a class that includes several definitions of addition for different types (note that the `+` operator is already overloaded).

```
class Adder {
    static int Add(int i, int j) {
       return i + j;
    }
    static float Add(float x, float y) {
       return x + y;
    }
    static String Add(String s1, String s2) {
       return s1.concat(s2);
    }
}
```

The expression `Adder.Add(5, 7)` would evaluate to `12`, while the expression `Adder.Add("Hello ", "world")` would evaluate to the string `"Hello world"`.

OCaml does *not* provide overloading. There are probably two main reasons. One has to do with a technical difficulty. It is hard to provide both type inference and overloading at the same time. For example, suppose the `+` function were overloaded to work both on integers and floating-point values. What would be the type of the following `add` function? Would it be `int ->int -> int`, or `float -> float -> float`?

```
let add x y =
   x + y;;
```

The best solution would probably to have the compiler produce *two* instances of the `add` function, one for integers and another for floating point values. This complicates the compiler, and with a sufficiently rich type system, type inference would become uncomputable. *That* would be a problem.

The second reason for not providing overloading is that programs can become more difficult to understand. It may not be obvious by looking at the program text which one of a function's definitions is being called, and there is no way for a compiler to check if all the function's definitions do "similar" things[1].

Subtype polymorphism and dynamic method dispatch

Subtype polymorphism and dynamic method dispatch are concepts used extensively in object-oriented programs. Both kinds of polymorphism are fully supported in OCaml. We discuss the

[1] The second reason is weaker. Properly used, overloading reduces namespace clutter by grouping similar functions under the same name. True, overloading is grounds for obfuscation, but OCaml is already ripe for obfuscation by allowing arithmetic functions like `+` to be redefined.

object system in Chapter **??**.

5.2 Tuples

Tuples are the simplest aggregate type. They correspond to the ordered tuples you have seen in mathematics, or set theory. A tuple is a collection of values of arbitrary types. The syntax for a tuple is a sequence of expressions separated by commas. For example, the following tuple is a pair containing a number and a string.

```
# let p = 1, "Hello";;
val p : int * string = 1, "Hello"
```

The syntax for the type of a tuple is a `*`-separated list of the types of the components. In this case, the type of the pair is `int * string`.

Tuples can be *deconstructed* by pattern matching with any of the pattern matching constructs like `let`, `match`, `fun`, or `function`. For example, to recover the parts of the pair in the variables `x` and `y`, we might use a `let` form.

```
# let x, y = p;;
val x : int = 1
val y : string = "Hello"
```

The built-in functions `fst` and `snd` return the components of a pair, defined as follows.

```
# let fst (x, _) = x;;
val fst : 'a * 'b -> 'a = <fun>
# let snd (_, y) = y;;
val snd : 'a * 'b -> 'b = <fun>
# fst p;;
- : int = 1
# snd p;;
- : string = "Hello"
```

Tuple patterns in a function argument must be enclosed in parentheses. Note that the `fst` and `snd` functions are polymorphic. They can be applied to a pair of any type `'a * 'b`; `fst` returns a value of type `'a`, and `snd` returns a value of type `'b`. There are no similar built-in functions for tuples with more than two elements, but they can be defined.

```
# let t = 1, "Hello", 2.7;;
val t : int * string * float = 1, "Hello", 2.7
# let fst3 (x, _, _) = x;;
val fst3 : 'a * 'b * 'c -> 'a = <fun>
# fst3 t;;
- : int = 1
```

Note also that the pattern assignment is simultaneous. The following expression swaps the values of `x` and `y`.

```
# let x = 1;;
val x : int = 1
# let y = "Hello";;
val y : string = "Hello"
# let x, y = y, x;;
val x : string = "Hello"
val y : int = 1
```

Since the components of a tuple are unnamed, tuples are most appropriate if they have a small number of well-defined components. For example, tuples would be an appropriate way of defining Cartesian coordinates.

```
# let make_coord x y = x, y;;
val make_coord : 'a -> 'b -> 'a * 'b = <fun>
# let x_of_coord = fst;;
val x_of_coord : 'a * 'b -> 'a = <fun>
# let y_of_coord = snd;;
val y_of_coord : 'a * 'b -> 'b = <fun>
```

However, it would be awkward to use tuples for defining database entries, like the following. For that purpose, records would be more appropriate. Records are defined in Chapter 7.

```
# (* Name, Height, Phone, Salary *)
  let jason = ("Jason", 6.25, "626-395-6568", 50.0);;
val jason : string * float * string * float =
# let name_of_entry (name, _, _, _) = name;;
val name_of_entry : 'a * 'b * 'c * 'd -> 'a = <fun>
  "Jason", 6.25, "626-395-6568", 50
# name_of_entry jason;;
- : string = "Jason"
```

5.3 Lists

Lists are also used extensively in OCaml programs. A list is a sequence of values of the same type. There are two constructors: the `[]` expression is the empty list, and the e_1::e_2 expression,

called a *cons* operation, creates a *cons cell*—a new list where the first element is e_1 and the rest of the list is e_2. The shorthand notation $[e_1;e_2;\cdots;e_n]$ is identical to $e_1::e_2::\cdots::e_n::[]$.

```
# let l = "Hello" :: "World" :: [];;
val l : string list = ["Hello"; "World"]
```

The syntax for the type of a list with elements of type `t` is `t list`. The `list` type is an example of a *parameterized* type. An `int list` is a list containing integers, a `string list` is a list containing strings, and an `'a list` is a list containing elements of some type `'a` (but all the elements have to have the same type).

Lists can be deconstructed using pattern matching. For example, here is a function that adds up all the numbers in an `int list`.

```
# let rec sum = function
     [] -> 0
   | i :: l -> i + sum l;;
val sum : int list -> int = <fun>
# sum [1; 2; 3; 4];;
- : int = 10
```

Functions on list can also be polymorphic. The function to check if a value `x` is in a list `l` might be defined as follows.

```
# let rec mem x l =
     match l with
        [] -> false
      | y :: l -> x = y || mem x l;;
val mem : 'a -> 'a list -> bool = <fun>
# mem 5 [1; 7; 3];;
- : bool = false
# mem "do" ["I'm"; "afraid"; "I"; "can't";
            "do"; "that"; "Dave"];;
- : bool = true
```

The function `mem` shown above takes an argument `x` of any type `'a`, and checks if the element is in the list `l`, which must have type `'a list`.

Similarly, the standard map function, `List.map`, might be defined as follows.

```
# let rec map f = function
   [] -> []
 | x :: l -> f x :: map f l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map succ [1; 2; 3; 4];;
- : int list = [2; 3; 4; 5]
```

The function `map` shown above takes a function `f` of type `'a -> 'b` (this argument function takes a value of type `'a` and returns a value of type `'b`), and a list containing elements of type `'a`, and it

returns a list containing elements of type `'b`—a `'b list`.

Lists are commonly used to represent sets of values or key-value relationships. The `List` library contains many list functions. For example, the `List.assoc` function returns the value associated with a key in a list of key-value pairs. This function might be defined as follows:

```
# let rec assoc key = function
     (key2, value) :: l ->
        if key2 = key then
           value
        else
           assoc x l
   | [] ->
        raise Not_found;;
```

Here we see a combination of list and tuple pattern matching. The pattern `(key2, value) :: l` should be read from the outside-in. The outermost operator is `::`, so this pattern matches a nonempty list, where the first element should be a pair `(key2,value)` and the rest of the list is `l`. If this pattern matches, and if the `key2` is equal to the argument `key`, then the `value` is returned as a result. Otherwise, the search continues. If the search bottoms out with the empty list, the default action is to raise an exception. According to convention in the `List` library, the `Not_found` exception is normally used by functions that search through a list and terminate unsuccessfully.

Association lists can be used to represent a variety of data structures, with the restriction that all values must have the same type. Here is a simple example.

```
# let entry =
     [("name", "Jason");
      ("height", "6' 3''");
      ("phone", "626-395-6568");
      ("salary", "$50")];;
val entry : (string * string) list =
  ["name", "Jason"; "height", "6' 3''";
   "phone", "626-345-9692"; "salary", "$50"]
# List.assoc "phone" entry;;
- : string = "626-395-6568"
```

Note that commas separate the elements of the pairs in the list, and semicolon separates the items of the list.

Chapter 6

Unions

Disjoint unions, also called tagged unions or variant records, are an important part of the OCaml type system. A disjoint union, or union for short, represents the union of several different types, where each of the parts is given an unique, explicit name.

OCaml allows the definition of *exact* and *open* union types. The following syntax is used for an exact union type; we discuss open types later in this chapter 6.6.

$$
\begin{aligned}
&\texttt{type } \textit{typename} = \\
&\quad \textit{Name}_1 \texttt{ of } \textit{type}_1 \\
&\mid \textit{Name}_2 \texttt{ of } \textit{type}_2 \\
&\quad \vdots \\
&\mid \textit{Name}_n \texttt{ of } \textit{type}_n
\end{aligned}
$$

The union type is defined by a set of cases separated by the vertical bar (|) character. Each case i has an explicit name $Name_i$, called a *constructor*; and it has an optional value of type $type_i$. The constructor name must be capitalized. The definition `of` $type_n$ is optional; if omitted there is no explicit value associated with the constructor.

Let's look at a simple example using unions, where we wish to define a numeric type that is either a value of type `int` or `float` or a canonical value `Zero`. We might define this type as follows.

```
# type number =
     Zero
   | Integer of int
   | Real of float;;
type number = Zero | Integer of int | Real of float
```

Values in a disjoint union are formed by applying a constructor to an expression of the appropriate type.

```
# let zero = Zero;;
val zero : number = Zero
# let i = Integer 1;;
val i : number = Integer 1
# let x = Real 3.2;;
val x : number = Real 3.2
```

Patterns also use the constructor name. For example, we can define a function that returns a floating-point representation of a number as follows. In this program, each pattern specifies a constructor name as well as a variable for the constructors that have values.

```
# let float_of_number = function
     Zero -> 0.0
   | Integer i -> float_of_int i
   | Real x -> x
```

Patterns can be arbitrarily nested. The following function represents one way that we might perform addition of values in the `number` type.

```
# let add n1 n2 =
     match n1, n2 with
        Zero, n
      | n, Zero ->
          n
      | Integer i1, Integer i2 ->
          Integer (i1 + i2)
      | Integer i, Real x
      | Real x, Integer i ->
          Real (x +. float_of_int i)
      | Real x1, Real x2 ->
          Real (x1 +. x2);;
val add : number -> number -> number = <fun>
# add x i;;
- : number = Real 4.2
```

There are a few things to note in this pattern matching. First, we are matching against the pair `(n1, n2)` of the numbers `n1` and `n2` being added. The patterns are then pair patterns. The first clause specifies that if the first number is `Zero` and the second is `n`, or if the second number is `Zero`

and the first is `n`, then the sum is `n`.

```
        Zero, n
      | n, Zero ->
          n
```

The second thing to note is that we are able to collapse some of the cases using similar patterns. For example, the code for adding `Integer` and `Real` values is the same, whether the first number is an `Integer` or `Real`. In both cases, the variable `i` is bound to the `Integer` value, and `x` to the `Real` value.

OCaml allows two patterns p_1 and p_2 to be combined into a choice pattern $p_1 \mid p_2$ under two conditions: both patterns must define the same variables; and, the value being matched by multiple occurrences of a variable must have the same types. Otherwise, the placement of variables in p_1 and p_2 is unrestricted.

In the remainder of this chapter we will describe the the disjoint union type more completely, using a running example for building balanced binary trees, a frequently-used data structure in functional programs.

6.1 Binary trees

Binary trees are frequently used for representing collections of data. A binary tree is a collection of nodes (also called vertices), where each node has either zero or two nodes called *children*. If node n_2 is a child of n_1, then n_1 is called the *parent* of n_2. One node, called the *root*, has no parents; all other nodes have exactly one parent.

One way to represent this data structure is by defining a disjoint union for the type of a node and its children. Since each node has either zero or two children, we need two cases. The following definition defines the type for a labeled tree: the `'a` type variable represents the type of labels; the `Node` constructor represents a node with two children; and the `Leaf` constructor represents a node with no children. Note that the type `'a tree` is defined with a type parameter `'a` for the type of labels. Note that this type definition is recursive. The type `'a tree` is mentioned in its own definition.

```
# type 'a tree =
     Node of 'a * 'a tree * 'a tree
   | Leaf;;
type 'a tree = | Node of 'a * 'a tree * 'a tree | Leaf
```

The use of tuple types in a constructor definition (for example, `Node of 'a * 'a tree * 'a tree`) is quite common, and has an efficient implementation. When applying a constructor, parentheses are required around the elements of the tuple. In addition, even though constructors with arguments are similar to functions, they are not functions, and may not be used as values.

```
# Leaf;;
- : 'a btree = Leaf
# Node (1, Leaf, Leaf);;
- : int btree = Node (1, Leaf, Leaf)
# Node;;
The constructor Node expects 3 argument(s),
but is here applied to 0 argument(s)
```

Since the type definition for `'a tree` is recursive, many of the functions defined on the tree will also be recursive. For example, the following function defines one way to count the number of non-leaf nodes in the tree.

```
# let rec cardinality = function
     Leaf -> 0
   | Node (_, left, right) ->
        cardinality left + cardinality right + 1;;
val cardinality : 'a btree -> int = <fun>
# cardinality (Node (1, Node (2, Leaf, Leaf), Leaf));;
- : int = 2
```

6.2 Unbalanced binary trees

Now that we have defined the type of binary trees, lets build a simple data structure for representing sets of values of type `'a`.

The empty set is just a `Leaf`. To add an element to a set `s`, we create a new `Node` with a `Leaf` as a left-child, and `s` as the right child.

```
# let empty = Leaf;;
val empty : 'a btree = Leaf
# let insert x s = Node (x, Leaf, s);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
# let rec set_of_list = function
     [] -> empty
   | x :: l -> insert x (set_of_list l);;
val set_of_list : 'a list -> 'a btree = <fun>
# let s = set_of_list [3; 5; 7; 11; 13];;
val s : int btree =
  Node
   (3, Leaf,
    Node (5, Leaf,
      Node (7, Leaf,
        Node (11, Leaf, Node (13, Leaf, Leaf)))))
```

The membership function is defined recursively: an element x is a member of a tree iff the tree is a `Node` and x is the label, or x is in the left or right subtrees.

```
# let rec mem x = function
     Leaf -> false
   | Node (y, left, right) ->
       x = y || mem x left || mem x right;;
val mem : 'a -> 'a btree -> bool = <fun>
# mem 11 s;;
- : bool = true
# mem 12 s;;
- : bool = false
```

6.3 Unbalanced, ordered, binary trees

One problem with the unbalanced tree defined here is that the complexity of the membership operation is $O(n)$, where n is cardinality of the set.

We can can begin to address the performance by ordering the nodes in the tree. The invariant we would like to maintain is the following: for any interior node `Node (x, left, right)`, all the labels in the left child are smaller than `x`, and all the labels in the right child are larger than `x`. To maintain this invariant, we must modify the insertion function.

```
# let rec insert x = function
     Leaf -> Node (x, Leaf, Leaf)
   | Node (y, left, right) ->
      if x < y then
         Node (y, insert x left, right)
      else if x > y then
         Node (y, left, insert x right)
      else
         Node (y, left, right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
# let rec set_of_list = function
     [] -> empty
   | x :: l -> insert x (set_of_list l);;
val set_of_list : 'a list -> 'a btree = <fun>
# let s = set_of_list [7; 5; 9; 11; 3];;
val s : int btree =
  Node
   (3, Leaf,
    Node (11,
      Node (9,
        Node (5, Leaf, Node (7, Leaf, Leaf)), Leaf), Leaf))
```

Note that this insertion function still does not build balanced trees. For example, if elements are inserted in increasing order, the tree will be completely unbalanced, with all the elements inserted along the right branch.

For the membership function, we can take advantage of the set ordering to speed up the search.

```
# let rec mem x = function
     Leaf -> false
   | Node (y, left, right) ->
        x = y || (x < y && mem x left) || (x > y && mem y right);;
val mem : 'a -> 'a btree -> bool = <fun>
# mem 5 s;;
- : bool = true
# mem 9 s;;
- : bool = true
# mem 12 s;;
- : bool = false
```

The complexity of this membership function is $O(l)$ where l is the maximal depth of the tree. Since the `insert` function does not guarantee balancing, the complexity is still $O(n)$, worst case.

6.4 Revisiting pattern matching

The `insert` function as expressed above is slightly inefficient. The final `else` clause (containing the expression `Node (y,left, right)`) returns a value that is equal to the one matched, but the application of the `Node` constructor creates a new value. The code would be more concise, and likely more efficient, if the matched value were used as the result.

OCaml provides a pattern form for binding the matched value using the syntax *pattern* `as` *variable*. In a clause p `as` v `->`e, the variable v is a binding occurrence. When a value is successfully matched with the pattern p, the variable v is bound to the value during evaluation of the body e. The simplified `insert` function is as follows.

```
# let rec insert x = function
     Leaf -> Node (x, Leaf, Leaf)
   | Node (y, left, right) as node ->
      if x < y then
         Node (y, insert x left, right)
      else if x > y then
         Node (y, left, insert x right)
      else
         node;;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

Patterns with `as` bindings may occur anywhere in a pattern. For example, the pattern `Node (y, left, right)` is equivalent to the pattern `Node (_ as y, (_ as left), (_ as right))`, though the former is preferred of course. The parentheses are required because the `as` keyword has very low precedence, lower than comma (`,`) and even the vertical bar (`|`).

Another extension to pattern matching is conditional matching with `when` clauses. The syntax of a conditional match has the form *pattern* `when` *expression*. The *expression* is a predicate to be evaluated if the *pattern* matches. The variables bound in the pattern may be used in the *expression*. The match is successful if, and only if, the *expression* evaluates to true.

A version of the `insert` function using `when` clauses is listed below. When the pattern match is performed, if the value is a `Node`, the second clause `Node (y, left, right) when x < y` is considered. If x is less than y, then x is inserted into the left branch. Otherwise, then evaluation falls through the the third clause `Node (y, left, right) when x > y`. If x is greater than y, then

x is inserted into the right branch. Otherwise, evaluation falls through to the final clause, which returns the original node.

```
# let rec insert x = function
     Leaf ->
         Node (x, Leaf, Leaf)
   | Node (y, left, right) when x < y ->
         Node (y, insert x left, right)
   | Node (y, left, right) when x > y ->
         Node (y, left, insert x right)
   | node ->
         node;;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

The performance of this version of the `insert` function is nearly identical to the previous definition using `if` to perform the comparison between x and y. Whether to use `when` conditions is usually a matter of style and preference.

6.5 Balanced red-black trees

In order to address the performance problem, we turn to an implementation of balanced binary trees. We'll use a functional implementation of red-black trees due to Chris Okasaki [4]. Red-black trees add a label, either `Red` or `Black`, to each non-leaf node. We will establish several new invariants.

1. Every leaf is colored black.
2. All children of every red node are black.
3. Every path from the root to a leaf has the same number of black nodes as every other path.
4. The root is always black.

These invariants guarantee the balancing. Since all the children of a red node are black, and each path from the root to a leaf has the same number of black nodes, the longest path is at most twice as long as the shortest path.

The type definitions are similar to the unbalanced binary tree; we just need to add a red/black label.

```
type color =
   Red
 | Black

type 'a rbtree =
   Node of color * 'a * 'a rbtree * 'a rbtree
 | Leaf
```

The membership function also has to be redefined for the new type.

```
let rec mem x = function
   Leaf -> false
 | Node (_, y, left, right) ->
      x = y || (x < y && mem x left) || (x > y && mem x right)
```

The difficult part of the data structure is maintaining the invariants when a value is added to the tree with the `insert` function. This can be done in two parts. First find the location where the node is to be inserted. If possible, add the new node with a `Red` label because this would preserve invariant 3. This may, however, violate invariant 2 because the new `Red` node may have a `Red` parent.

In order to preserve the invariant, we implement the `balance` function, which considers all the cases where a `Red` node has a `Red` child and rearranges the tree.

```
# let balance = function
     Black, z, Node (Red, y, Node (Red, x, a, b), c), d
   | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
   | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
   | Black, x, a, Node (Red, y, b, Node (Red, z, c, d)) ->
        Node (Red, y, Node (Black, x, a, b), Node (Black, z, c, d))
   | a, b, c, d ->
        Node (a, b, c, d)

  let insert x s =
     let rec ins = function
        Leaf -> Node (Red, x, Leaf, Leaf)
      | Node (color, y, a, b) as s ->
           if x < y then balance (color, y, ins a, b)
           else if x > y then balance (color, y, a, ins b)
           else s
     in
        match ins s with  (* guaranteed to be non-empty *)
           Node (_, y, a, b) -> Node (Black, y, a, b)
         | Leaf -> raise (Invalid_argument "insert");;
val balance : color * 'a * 'a rbtree * 'a rbtree -> 'a rbtree = <fun>
val insert : 'a -> 'a rbtree -> 'a rbtree = <fun>
```

Note the use of nested patterns in the `balance` function. The `balance` function takes a 4-tuple,

with a `color`, two `btree`s, and an element, and it splits the analysis into five cases: four of the cases are for the situation where invariant 2 needs to be re-established because `Red` nodes are nested, and the final case is the case where the tree does not need rebalancing.

Since the longest path from the root is at most twice as long as the shortest path, the depth of the tree is $O(log\ n)$. The `balance` function takes $O(1)$ (constant) time. This means that the `insert` and `mem` functions each take time $O(log\ n)$.

```
# let empty = Leaf;;
val empty : 'a rbtree = Leaf
# let rec set_of_list = function
       [] -> empty
     | x :: l -> insert x (set_of_list l);;
val set_of_list : 'a list -> 'a rbtree = <fun>
# let s = set_of_list [3; 9; 5; 7; 11];;
val s : int rbtree =
  Node (Black, 7, Node (Black, 5, Node (Red, 3, Leaf, Leaf), Leaf),
   Node (Black, 11, Node (Red, 9, Leaf, Leaf), Leaf))
# mem 5 s;;
- : bool = true
# mem 6 s;;
- : bool = false
```

6.6 Open union types

OCaml defines a second kind of union type where the type is open—that is, other definitions may add more cases to the type definition. The syntax is similar to the exact definition discussed previously, but the type but the constructor names are prefixed with a backquote (`) symbol, and the type definition is enclosed in `[> ...]` brackets.[1]

For example, let build an extensible version of the numbers from the first example in this chapter. Initially, we might define an `add` function for `` `Integer `` values.

```
# let string_of_number1 n =
     match n with
        `Integer i -> string_of_int i
      | _ -> raise (Invalid_argument "unknown number");;
val string_of_number1 : [> `Integer of int ] -> string = <fun>
# string_of_number1 (`Integer 17);;
- : string = "17"
```

[1] As of OCaml 3.08.0, the language does not allow open union types in type definitions.

The type `[> ‘Integer of int ]` specifies that the function takes an argument having an open union type, where one of the constructors is `‘Integer` (with a value of type `int`).

Later, we might want to define a function that includes a constructor `‘Real` for floating-point values. We can extend the definition as follows.

```
# let string_of_number2 n =
     match n with
        ‘Real x -> string_of_float x
      | _ -> string_of_number1 n;;
val string_of_number2 : [> ‘Integer of int | ‘Real of float ] -> string =
  <fun>
```

If passed a floating-point number with the `‘Real` constructor, the string is created with `string_of_float` function. Otherwise, the original function `string_of_number1` is used.

The type `[> ‘Integer of int | ‘Real of float ]` specifies that the function takes an argument in an open union type, and handles the constructors `‘Integer` with a value of type `int`, and `‘Real` with a value of type `float`. Unlike the exact union, the constructors may still be used with expressions of other types. However, application to a value of the wrong type remains disallowed.

```
# let n = ‘Real 1;;
val n : [> ‘Real of int ] = ‘Real 1
# string_of_number2 n;;
Characters 18-19:
  string_of_number2 n;;
                    ^
This expression has type [> ‘Real of int ] but is here used with type
  [> ‘Integer of int | ‘Real of float ]
Types for tag ‘Real are incompatible
```

6.7 Some common built-in unions

A few of the types we have already seen are unions. The built-in Boolean type `bool` is defined as a union. Normally, the constructor names in a union must be capitalized. OCaml defines an exception in this case by treating `true` and `false` as capitalized identifiers.

```
# type bool =
     true
   | false
type bool = | true | false
```

The list type is similar, having the following effective definition. However, the `’a list` type is

primitive in this case because `[]` is not considered a legal constructor name.

```
type 'a list =
   []
 | :: of 'a * 'a list;;
```

Although it is periodically suggested on the OCaml mailing list, OCaml does not have a NIL value that can be assigned to a variable of any type. Instead, the built-in `'a option` type is used.

```
# type 'a option =
     None
   | Some of 'a;;
type 'a option = | None | Some of 'a
```

The `None` case is intended to represent a NIL value, while the `Some` case handles non-NIL values.

Chapter 7

Reference cells, Side-Effects, and Loops

As we have seen, functional programming has one central feature—functions are first-class. Functions may be passed as arguments, returned as the result of function calls, and stored in data structures just like any other value. Indeed, the presence of first-class functions is the only requirement for a programming language to be considered functional. By this definition, many programming languages are functional, including not only the usual examples like OCaml, Lisp, and Haskell, but also languages like Javascript (where functions are associated with fields on a web page), or even C (where functions are represented with pointers).

Another property of a programming language is purity. A *pure* programming language is one without assignment, where variables cannot be modified by side-effect. Haskell is an example of a pure functional programming language; OCaml and most Lisp dialects are impure, meaning that they allow side-effects in some form. The motivation for pure functional programming stems in part from their simplicity and mathematical foundations. Mathematically speaking, a function is a single-valued map, meaning that if f is a function and $f(x)$ is defined, then there is only one value for $f(x)$. Consider the following "counter function," written in C.

```
int count = 0;
int counter() {
    count = count + 1;
    return count;
}
```

Clearly, this is not a function in the mathematical sense, since the value returned by the `counter` function is different each time it is called; in fact the expression `counter() == counter()` is always false.

Reasoning about languages with assignment and side-effects is more difficult than for the pure languages because of the need to specify the program "state," which defines the values for the variables in the program. To be fair, pure languages have issues of their own. It isn't always easy to write a pure program that is as efficient as an impure one. Furthermore, the world is impure in some sense. When I run a program that displays the message "Hello world" on my screen, the display is ultimately modified by side-effect to show the message.

For these reasons, and perhaps others, OCaml is an impure language that allows side-effects. However, it should be noted that the *predominant* style used by OCaml programmers is pure; assignment and side-effects are used infrequently, if at all.

7.1 Reference cells

The simplest mutable value in OCaml is the *reference cell*, which can be viewed as a "box" where the contents can be replaced by assignment. Reference cells are created with the `ref` function, which takes an initial value for the cell; they are mutated with the `:=` operator, which assigns a new value to the cell; and they are dereferenced with the `!` operator.

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# i := 2;;
- : unit = ()
# !i;;
- : int = 2
```

The built-in type `'a ref` is the type of a reference cell. Don't get confused with the `!` operator in C. The following code illustrates a potential pitfall.

```
# let flag = ref true;;
val flag : bool ref = {contents=true}
# if !flag then 1 else 2;;
- : int = 1
```

If you have programmed in C, you may be tempted to read `if !flagthen ...` as testing if the `flag` is false. This is *not* the case; the `!` operator is more like the `*` operator in C.

Another key difference between reference cells and assignment in languages like C is that it is the *cell* that is modified by assignment, not the variable (variables are always immutable in OCaml). For example, in the following code, the two variables i and j refer to the same reference cell, so an assignment to the cell affects the value of both variables.

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# let j = i;;
val j : int ref = {contents = 1}
# i := 2;;
- : unit = ()
# !j;;
- : int = 2
```

7.1.1 Value restriction

As we mentioned in Section 5.1.1, mutability and side-effects interact with type inference. For example, consider a "one-shot" function that saves a value on its first call, and returns that value on all future calls. This function is not properly polymorphic because it contains a mutable field. The following example illustrates the issue.

```
# let x = ref None;;
val x : '_a option ref = {contents=None}
# let one_shot y =
     match !x with
        None ->
           x := Some y;
           y
      | Some z ->
           z;;
val one_shot : '_a -> '_a = <fun>
# one_shot 1;;
- : int = 1
# one_shot;;
val one_shot : int -> int = <fun>
# one_shot 2;;
- : int = 1
# one_shot "Hello";;
Characters 9-16:
This expression has type string but is here used with type int
```

The value restriction requires that polymorphism be restricted to immutable values, including functions, constants, and constructors with fields that are values. A function application is *not* a value, and a reference cell is not a value. By this definition, the `x` variable and the `one_shot` function cannot be polymorphic, as the type constants `'_a` indicate.

7.1.2 Imperative programming and loops

In an imperative programming language, iteration and looping are used much more frequently than recursion. The examples in Figure 7.3 show an example of a C function to compute the factorial of a number, and a corresponding OCaml program written in the same style.

A `for` loop defines iteration over an integer range. In the factorial example, the loop index is `k`, the initial value is `2`, the final value is `i`. The loop body is evaluated for each integer value of `k` between `2` and `i` *inclusive*. If `i` is less than `2`, the loop body is not evaluated at all.

OCaml also includes a for-loop that iterates downward, specified by using the keyword `downto` instead of `to`, as well as a general `while`-loop. These variations are shown in Figure **??**.

For the factorial function, there isn't really any reason to use iteration over recursion, and there are several reasons not to. For reference, two pure functional versions of the factorial function are

```
int fact(int i) {
    int j = 1, k;
    for(k = 2; k <= i; k++)
       j *= k;
    return j;
}
```

```
let fact i =
   let j = ref 1 in
      for k := 2 to i do
         j := !j * k
      done;
      !j
```

Figure 7.1: Two examples of a factorial function written in an imperative style.

```
let fact i =
   let j = ref 1 in
      for k := i downto 2 do
         j := !j * k
      done;
      !j
```

```
let fact i =
   let j = ref 1 in
   let k = ref 2 in
      while !k <= i do
         j := !j * !k
      done;
      !j
```

Figure 7.2: Two variations on the factorial using a downward-iterating for loop, and a while loop.

shown in Figure **??**. One reason to prefer the pure functional version is that it is simpler and more clearly expresses the computation being performed. While it can be argued what the properties "simple" and "clear" are never simple and clear in the context of programming language, most OCaml programmers would find the pure functional versions easier to read.

—JYH: need to add a **difficult** marker —

Another reason is that the pure functional version is likely to be more efficient because there is no penalty for the overhead of assigning to and dereferencing reference cells. In addition, the compiler is particularly effective in generating code for tail-recursive functions. A *tail-recursive* function is a function where the result is a constant or a call to another function. The second version of the factorial function in Figure **??** is tail-recursive because it returns either the constant `1` or the value from the recursive call `loop (i -1) (i * j)`. In the latter case, the compiler notices that the storage for the current argument list is no longer needed, so it may be reallocated before the recursive call. This small optimization means that the tail-recursive version runs in constant space,

```
let rec fact i =
   if i <= 1 then
      1
   else
      i * fact (i - 1)
```

```
let fact i =
   let rec loop i j =
      if i <= 1 then
         j
      else
         loop (i - 1) (i * j)
   in
      loop i 1
```

Figure 7.3: Pure functional versions for computing the factorial. The version on the left is the simple translation. The version on the right is a somewhat more efficient tail-recursive implementation.

which often results in a large performance improvement.

7.2 Examples of using reference cells

7.2.1 Queues

A *queue* is a data structure that supports an *enqueue* operation that adds a new value to the queue, and a *dequeue* operation that removes an element from the queue. The elements are dequeued in FIFO (first-in-first-out) order. Queues are often implemented as imperatide data structures, where the operations are performed by side-effect. The following signature gives the types of the functions to be implemented.

```
type 'a queue

val create  : unit -> 'a queue
val enqueue : 'a queue -> 'a -> unit
val dequeue : 'a queue -> 'a
```

For efficiency, we would like the queue operations to take constant time. One simple implementation is to represent the queue as two lists, an *enqueue* list and a *dequeue* list. When a value is

enqueued, it is added to the enqueue list. When an element is dequeued, it is taken from the dequeue list. If the dequeue list is empty, the queue is *shifted* by setting the dequeue list to the reversal of the enqueue list.

```
(* 'a queue = (enqueue_list, dequeue_list) *)
type 'a queue = 'a list ref * 'a list ref

(* Create a new empty queue *)
let create () =
   (ref [], ref [])

(* Add to the element to the enqueue list *)
let enqueue (eq, _) x =
   eq := x :: !eq

(* Remove an element from the dequeue list *)
let rec dequeue ((eq, dq) as queue) =
    match !dq with
       x :: rest ->
          dq := rest;
          x
     | [] ->  (* Shift the queue *)
          if !eq = [] then
             raise Not_found;
          dq := List.rev !eq;
          eq := [];
          dequeue queue
```

Note that the `dequeue` function is defined recursively. When the `dq` list is empty, the function raises an error if the `eq` list is also empty; otherwise, the lists are shifted and the operation is retried. The explicit check for an empty queue prevents infinite recursion.

7.2.2 Cyclic data structures

One issue with the previous implementation is that the queue must be shifted whenever the dequeue list becomes empty, which means that the time to perform a dequeue operation can be unpredictable. In situations where timing is an issue, another common implementation of queues uses a *circular linked list*, where each element in the list points to the previous element that was inserted, and the newest element points to the oldest. If we have a pointer to the newest element, then we can implement the queue operations in constant time as follows.

- To enqueue an element to the queue, add it beween the newest and the oldest.
- The oldest element is the next one after the first. To dequeue it, remove it from the queue.

This implementation seems straightforward enough, we simply need to construct a circular linked list. But this is a problem. In a pure functional language, cyclic data structures of this form are not implementable. When a data value is constructed, it can only be constructed from values that already exist, not itself.

Once again, reference cells provide a simple way to get around the problem by allowing links in the list to be set after the elements have already been created.

To begin, we first need to choose a representation for the queue. First, the elements in the circular list are of type `elem`, which is a pair `(x, next)`, where `x` is the value of the element, and `next` is the pointer to the next element of the queue. The queue itself can be empty, so we define the type as a reference to an `elem option`.

```
(* An element is a pair (value, previous-element) *)
type 'a elem = 'a * 'a pointer
and 'a pointer = Pointer of 'a elem ref

type 'a queue = 'a elem option ref

let create () =
   None
```

You might wonder why not give the `'a elem` type a more straightforward definition as `'a* 'a elem ref`. The problem with this type definition is that it is *cyclic* (since the type `'a elem` appears in its own definition). By default, OCaml rejects cyclic definitions because they can be confusing.

```
% ocaml
     Objective Caml version 3.08.3

# type 'a elem = 'a * 'a elem ref;;
The type abbreviation elem is cyclic
```

The solution is to introduce a union type (`pointer` in this case). This introduces the `Pointer` constructor, which makes the definition acceptable because the recursive occurence of `elem` in `Pointer of 'a elem ref` is now within a constructor.

Next, let's consider the function to add an element to the queue. The invariant of the queue data structure is that the each element in the circular list points to the next newer element, and

the newest points to the oldest. The one exception is when the queue is empty, since there are no elements. In this case, when adding the element, we need to create it so that it refers to itself, since it is simultaneously the oldest and newest element. This is done with a recursive value definition, `let rec elem = (x, Pointer (ref elem))`, where the element `elem` is defined to point to itself.

```
let enqueue queue x =
   match !queue with
     None ->
         (* The element should point to itself *)
         let rec elem = (x, Pointer (ref elem)) in
             queue := Some elem
   | Some (_, Pointer prev_next) ->
         (* Insert after the previous element *)
         let oldest = !prev_next in
         let elem = (x, Pointer (ref oldest)) in
            prev_next := elem;
            queue := Some elem
```

For the second case, where the queue is non-empty, we create a new element `elem` that points to the oldest element, modify the previous element so that it points to the new element (by setting the `prev_next` reference), and set the queue to point to the new element.

To finish off the implementation, we need to add a function to dequeue an element from the queue. According to the queue invariant, the oldest element is the element after the newest. To dequeue it, we simply unlink it from the queue, with one exception. If the queue contains only one element, then that element will point to itself. We can test for this using the operator `==` for pointer equality; and if so, set the queue to `None` to indicate that it is empty.

```
let dequeue queue =
   match !queue with
      None ->
         (* The queue is empty *)
         raise Not_found
    | Some (_, Pointer oldest_ref) ->
         let oldest = !oldest_ref in
         let (x, Pointer next_ref) = oldest in
         let next = !next_ref in
            (* Test whether the element points to itself *)
            if next == oldest then
               (* It does, so the queue becomes empty *)
               queue := None
            else
               (* It doesn't, unlink it *)
               oldest_ref := next;
            x
```

There are a few things to learn from this example. For one, it is much more complicated than the first implementation using two lists. The type definitions and the data structure itself are cyclic, and so the implementation is less natural. For another, we had to make use of two new operations, the comparison `==` for pointer equality, and a `let rec` for a recursive value definition. In the end, the data structure is more difficult to understand than the two-list version, and is less likely to be encountered in practice.

7.2.3 Functional queues with reference cells

The previous two examples of queues are imperative, meaning that the `enqueue` and `dequeue` functions modify the queue in-place. One might also wonder if there are efficient functional implementations—that is, rather than modifying the queue in place, the `enqueue` and `dequeue` operations produce new queues without effecting the old one. There are many advantages to functional data structures. Among the most important is that functional data structures are *persistent*—their operations produce new data without destroying old.

It is easy enough to construct a functional version for queues. Since the operations now return new queues, the signature changes to the following.

```
type 'a queue

val empty   : 'a queue
val enqueue : 'a queue -> 'a -> 'a queue
val dequeue : 'a queue -> 'a * 'a queue
```

Note that there is no longer a need for a `create` function to create a new queue, we can simply use a canonical empty queue.

For the implementation, let's return to the simpler implementation using two lists. The first step is to eliminate all reference cells. The following code provides this translation. Note that the `enqueue` operation returns a new queue, and the `dequeue` operation returns a pair of an element and a new queue.

```
(* The queue is (enqueue_list, dequeue_list) *)
type 'a queue = 'a list * 'a list

(* Construct an empty queue *)
let create () =
   ([], [])

(* Add the new element to the enqueue_list *)
let enqueue (eq, dq) x =
   (x :: eq, dq)

(* Take an element from the dequeue list *)
let rec dequeue = function
   (eq, x :: dq) ->
      x, (eq, dq)
 | ([], []) ->
      raise Not_found
 | (eq, []) ->
      (* Shift the queue, and dequeue again *)
      dequeue ([], List.rev eq)
```

This seems simple enough, and indeed the code is simpler and smaller than the original imperative version. Unfortunately, the `dequeue` function no longer takes constant time! Imagine a scenario where a large number of elements are added to a queue without any intervening dequeue operations. The result will be a queue that is maximally imbalanced, with all the elements in the enqueue list. If we wish to use the queue multiple times, each time we use the dequeue function, the queue will have to be shifted by reversing the enqueue list, taking time linear in the number of elements.

The solution around this uses reference cells to "remember" the results of the shift operation.

After all, the shift doesn't change the elements in the queue, it just changes their representation. Externally, we can preserve the functional appearance of the queue data structure; the implementation will still be a queue, and it will still be persistent. The modification that is needed is to add a reference cell that can be used to shift the queue in-place.

```
(* The queue is (enqueue_list, dequeue_list) *)
type 'a queue = ('a list * 'a list) ref

(* The empty queue is a value *)
let create () =
   ref ([], [])

(* Add the new element to the enqueue_list *)
let enqueue queue x =
   let (eq, dq) = !queue in
      ref (x :: eq, dq)

(* Take an element from the dequeue list *)
let rec dequeue queue =
   match !queue with
      (eq, x :: dq) ->
         x, ref (eq, dq)
    | ([], []) ->
         raise Not_found
    | (eq, []) ->
         (* Shift the queue in-place *)
 queue := ([], List.rev eq);
         dequeue queue
```

In this revised version, reference cells are used purely as an optimization. To preserve the behavior of the original functional version, when a new queue is created, it is created with a new reference cell. This prevents operations on one queue from affecting any others; the data remains persistent.

7.2.4 Summary

7.2.5 Exercises

JYH: these are just thoughts for now.

1. In the implementation of queues as circular lists, we used a recursive value definition.

```
let rec elem = (x, Pointer (ref elem)) in ...
```

   Many languages do not have this feature. What would you need to do if values could not be defined recursively? What would be the impact on performance?

2. While the comparison `==` is frequently understood as physical (pointer) equality, the OCaml documentation gives a weaker definition, "For any two values `x` and `y`, if `x == y` then `x = y`." According to this definition, it would be acceptable if the `==` comparison always returns `false`. What would happen to the implementation of queues using circular linked lists if so? How could it be fixed?

3. The functional versions of the queue have a `create` function that returns a fresh, empty queue. Since the data structure is functional, it would be reasonable to replace the `create` function with a value `code` empty that represents the empty queue. For example, in the purely function version, we could define the empty queue as the following, and remove the `create` function.

```
let empty = ([], [])
```

   Why won't this work in the version of the queue that uses reference cells?

4. Is it possible to implement a persistent queue using circular linked lists, and all operations are $O(1)$ (constant time)? If so, provide an implementation. If not, explain why not.

Chapter 8

Exceptions

Exceptions are used in OCaml as a control mechanism, either to signal errors, or control the flow of execution in some other way. In their simplest form, exceptions are used to signal that the current computation cannot proceed because of a run-time error. For example, if we try to evaluate the quotient `1 / 0` in the toploop, the runtime signals a `Division_by_zero` error, the computation is aborted, and the toploop prints an error message.

```
# 1 / 0;;
Exception: Division_by_zero.
```

Exceptions can also be defined and used explicitly by the programmer. For example, suppose we define a function `head` that returns the first element of a list. If the list is empty, we would like to signal an error.

```
# exception Fail of string;;
exception Fail of string
# let head = function
     h :: _ -> h
   | [] -> raise (Fail "head: the list is empty");;
val head : 'a list -> 'a = <fun>
# head [3; 5; 7];;
- : int = 3
# head [];;
Exception: Fail "head: the list is empty".
```

The first line of this program defines a new exception, declaring `Fail` as a new exception with a string argument. The `head` function computes by pattern matching—the result is `h` if the list has

first element `h`; otherwise, there is no first element, and the `head` function raises a `Fail` exception. The expression `(Fail "head: the list is empty")` is a value of type `exn`; the `raise` function is responsible for aborting the current computation.

```
# Fail "message";;
- : exn = Fail "message"
# raise;;
- : exn -> 'a = <fun>
# raise (Fail "message");;
Exception: Fail "message".
```

The type `exn -> 'a` for the `raise` function may seem surprising at first—it appears to say that the raise function can produce a value having *any* type. In fact, what it really means is that the `raise` function never returns, so the type of the result doesn't matter. When a `raise` expression occurs in a larger computation, the entire computation is aborted.

```
# 1 + raise (Fail "abort") * 21;;
Exception: Fail "abort".
```

When an exception is raised, the current computation is aborted, and control is passed directly to the currently active exception handler, which in this case is the toploop itself. It is also possible to define explicit exception handlers. For example, suppose we wish to define a function `head_default`, similar to `head`, but returning a default value if the list is empty. One way would be to write a new function from scratch, but we can also choose to handle the exception from `head`.

```
# let head_default l default =
    try head l with
      Fail _ -> default;;
val head_default : 'a list -> 'a -> 'a = <fun>
# head_default [3; 5; 7] 0;;
- : int = 3
# head_default [] 0;;
- : int = 0
```

The `try` *e* `with` *cases* expression is very much like a `match` expression, but it matches exceptions that are raised during evaluation of the expression *e*. If *e* evaluates to a value without raising an exception, the value is returned as the result of the `try` expression. Otherwise, the raised exception is matched against the patterns in *cases*, and the first matching case is selected. In the example, if evaluation of `head l` raises the `Fail` exception, the value `default` is returned.

8.1 Nested exception handlers

Exceptions are handled dynamically, and at run-time there may be many active exception handlers. To illustrate this, let's consider an alternate form of a list-map function, defined using a function `split` that splits a non-empty list into its head and tail.

```
# exception Empty;;
exception Empty
# let split = function
     h :: t -> h, t
   | [] -> raise Empty;;
val split : 'a list -> 'a * 'a list = <fun>
# let rec map f l =
     try
        let h, t = split l in
           f h :: map f t
     with
        Empty -> [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun i -> i + 1) [3; 5; 7];;
- : int list = [4; 6; 8]
```

The call to `map` on the three-element list `[3; 5; 7]` results in four recursive calls corresponding to `map f [3; 5;7]`, `map f [5; 7]`, `map f [7]`, and `map f []`, before the function `split` is called on the empty list. Each of the calls defines a new exception handler.

It is appropriate to think of these handlers forming an exception stack corresponding to the call stack (this is, in fact, they way it is implemented in the OCaml implementation from INRIA). When a `try` expression is evaluated, a new exception handler is pushed onto the the stack; the handler is removed when evaluation completes. When an exception is raised, the entries of the stack are examined in stack order. If the topmost handler contains a pattern that matches the raised exception, it receives control. Otherwise, the handler is popped from the stack, and the next handler is examined.

In our example, when the `split` function raises the `Empty` exception, the top four elements of the exception stack contain handlers corresponding to each of the recursive calls of the `map` function. When the `Empty` exception is raised, control is passed to the innermost call `map f []`, which returns the empty list as a result.

map f []
map f [7]
map f [5; 7]
map f [3; 5; 7]

This example also contains a something of a surprise. Suppose the function `f` raises the `Empty` exception? The program gives no special status to `f`, and control is passed to the uppermost handler on the exception stack. As a result, the list is truncated at the point where the exception occurs.

```
# map (fun i ->
          if i = 0 then
             raise Empty
          else
             i + 1) [3; 5; 0; 7; 0; 9];;
- : int list = [4; 6]
```

8.2 Examples of uses of exceptions

Like many other powerful language constructs, exceptions can used to simplify programs and improve their clarity. They can also be abused in many ways. In this section we cover some standard uses of exceptions, and some of the abuses.

8.2.1 Pattern matching failure

The OCaml standard library uses exceptions for many purposes. We have already seen how exceptions are used to handle some run-time errors like incomplete pattern matches. When a pattern matching is incompletely specified, the OCaml compiler issues a warning (and a suggestion for the missing pattern). At runtime, if the matching fails because it is incomplete, the `Match_failure` exception is raised. The three values are the name of the file, the line number, and the character offset within the line where the match failed. It is often considered bad practice to catch the `Match_failure` exception because the failure usually indicates a programming error (in fact, proper programming practice would dictate that all pattern matches be complete).

```
# let f x =
     match x with
        Some y -> y;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
val f : 'a option -> 'a = <fun>
# f None;;
Exception: Match_failure ("", 2, 3).
```

8.2.2 Assertions

Another common use of exceptions is for checking runtime invariants. The `assert` operator evaluates a Boolean expression, raising an `Assert_failure` exception if the value is `false`. For example, in the following version of the factorial function, an assertion is used to generate a runtime error if the function is not called with a negative argument. The three arguments represent the file, line, and character offset of the failed assertion. As with `Match_failure`, it is considered bad programming practice to catch the `Assert_failure` exception.

```
# let rec fact i =
     assert (i >= 0);
     if i = 0 then
        1
     else
        i * fact (i - 1);;
val fact : int -> int = <fun>
# fact 10;;
- : int = 3628800
# fact (-10);;
Exception: Assert_failure ("", 9, 3).
```

8.2.3 Invalid_argument and Failure

The `Invalid_argument` exception is similar to an assertion failure; it indicates that some kind of runtime error occurred. One of the more common causes is array and string subscripts that are out-of-bounds. The `Invalid_argument` exception includes a string describing the error.

```
# let a = [|5; 6; 7|];;
val a : int array = [|5; 6; 7|]
# a.(2);;
- : int = 7
# a.(3);;
Exception: Invalid_argument "index out of bounds".
```

The `Failure` exception is similar, but it is usually used to signal errors that are considered less severe. The `Failure` exception also includes a string describing the error. The standard convention is that the string describing the failure should be the name of the function that failed.

```
# int_of_string "0xa0";;
- : int = 160
# int_of_string "0xag";;
Exception: Failure "int_of_string".
```

The `Invalid_argument` and `Failure` exceptions are quite similar—they each indicate a run-time error, using a string to describe it, so what is the difference?

The difference is primarily a matter of style. The `Invalid_argument` exception is usually used to indicate *programming* errors, or errors that should never happen if the program is correct, similar to assertion failures. The `Failure` exception is used to indicate errors that are more benign, where it is possible to recover, and where the cause is often due to external events (for example, when a string `0xag` is read in a place where a number is expected).

For illustration, suppose we are given a pair of lists, `names` and `grades`, that describe the students taking a class. We are told that every student in the class must have a grade, but not every student is taking the class. We might define the function to return a student's grade by recursively search through the two lists until the entry for the student is found.

```
let rec find_grade student (names, grades) =
   match (names, grades) with
      (name :: names'), (grade :: grades') ->
         if name = student then
            grade
         else
            find_grade student (names', grades')
    | [], [] ->
         raise (Failure "student not enrolled in the class")
    | [], (_ ::_)
    | (_ :: _), [] ->
         raise (Invalid_argument "corrupted database")
```

The first match clause handles the case where the two lists are nonempty, returning the student's

grade if the name matches, and continuing with the rest of the lists otherwise. In the second clause, when both lists are empty, the search fails. Since this kind of failure is expected to happen occasionally, the proper exception is `Failure`. In the final clause, it is found that the `names` and `grades` lists have different lengths. The proper exception in this case is `Invalid_argument` because i) the error violates a key programming invariant (that every student has a grade), and ii) there is no obvious way to recover. As a matter of style, it is usually considered bad practice to catch `Invalid_argument` exceptions (in fact, some early OCaml implementations did not even allow it). In contrast, `Failure` exceptions are routinely caught so that the error can be corrected.

8.2.4 The Not_found exception

The `Not_found` exception is used by search functions to indicate that a search failed. There are many such functions in OCaml. One example is the `List.assoc` function, which searches for a key-value pair in a list. For instance, instead of representing the grades in the previous example as two lists, we might represent the grades as a list of pairs (this will also enforce the requirement that every student have a grade).

```
# let grades = [("John", "B-"); ("Jane", "A"); ("Joan", "C")];;
val grades : (string * string) list = ...
# List.assoc "Jane" grades;;
- : string = "A"
# List.assoc "June" grades;;
Exception: Not_found.
```

Stylistically, the `Not_found` exception is often routine, and expected to happen during normal program operation.

8.2.5 Memory exhaustion exceptions

The two exceptions `Out_of_memory` and `Stack_overflow` indicate that memory resources have been exhausted. The `Out_of_memory` exception is raised by the garbage collector when there is insufficient memory to continue running the program. The `Stack_overflow` exception is similar, but it is restricted just to stack space. The most common cause of a `Stack_overflow` error is deep recursion (for example, using the `List.map` function on a list with more than a few thousand

elements), or an infinite loop in the program.

Both errors are severe, and the exceptions should not be caught casually. For the `Out_of_memory` exception it is often useless to catch the exception without freeing some resources, since the exception handler will usually not be able to execute if all memory has been exhausted.

Catching the `Stack_overflow` exception is not advised for a different reason. While the `Stack_overflow` exception can be caught reliably by the byte-code interpreter, it is not supported by the native-code compiler on all architectures. In many cases, a stack overflow will result in a system error (a "segmentation fault"), instead of a runtime exception. For portability, it is often better to avoid catching the exception.

8.3 Other uses of exceptions

Exceptions are also frequently used to modify the control flow of a program, without necessarily being associated with any kind of error condition.

8.3.1 Decreasing memory usage

As a simple example, suppose we wish to write a function to remove the first occurrence of a particular element "x" in a list "l". The straightforward implementation is defined as a recursive function.

```
let rec remove x = function
   y :: l when x = y ->
      l
 | y :: l (* x <> y *) ->
      y :: remove x l
 | [] ->
      []
```

The `remove` function searches through the list for the first occurrence of an element `y` that is equal to `x`, reconstructing the list after the removal.

One problem with this function is that the entire list is copied needlessly when the element is not found, potentially increasing the space needed to run the program. Exceptions provide a convenient way around this problem. By raising an exception in the case where the element is not found, we

can avoid reconstructing the entire list. In the following function, when the `Unchanged` exception is raised, the `remove` function returns the original list `l`.

```
exception Unchanged

let rec remove_inner x = function
   y :: l when x = y ->
      l
 | y :: l (* x <> y *) ->
      y :: remove_inner x l
 | [] ->
      raise Unchanged

let remove x l =
   try remove_inner x l with
      Unchanged ->
         l
```

8.3.2 Break statements

While OCaml provides both "`for`" and "`while`" loops, there is no "`break`" statement as found in languages like C and Java. Instead, exceptions can be used to abort a loop execution. To illustrate this, suppose we want to define a function `cat` that prints out all the lines from the standard input channel. We discuss input/output in more detail in Section 9, but for this problem we can just use the standard functions `input_char` to read a character from the input channel, and `output_char` to write it to the output channel. The `input_char` function raises the exception `End_of_file` when the end of the input has been reached.

```
let cat in_channel out_channel =
   try
      while true do
         output_char out_channel (input_char in_channel)
      done
   with
      End_of_file ->
         ()
```

The `cat` function defined an infinite loop (`while true do... done`) to copy the input data to the output channel. When the end of the input has been reached, the `input_char` function raises the `End_of_file` exception, breaking out of the loop, returning the `()` value as the result of the function.

8.3.3 Unwind-protect (finally)

In some cases where state is used, it is useful to define a "finally" clause (similar to an "unwind-protect" as seen in Lisp languages). The purpose of a "finally" clause is to execute some code (usually to clean up) after an expression is evaluated. In addition, the finally clause should be executed even if an exception is raised. A generic `finally` function can be defined using a wildcard exception handler. In the following function, the `result` type is used to represent the result of executing the function "f" on argument "x," returning a `Success` value if the evaluation was successful, and `Failure` otherwise. Once the result is computed, the `cleanup` function is called, and i) the result is returned on `Success`, or ii) the exception is re-raised on `Failure`.

```
type 'a result =
   Success of 'a
 | Failure of exn

let finally f x cleanup =
   let result =
      try Success (f x) with
         exn ->
            Failure exn
   in
      cleanup ();
      match result with
         Success y -> y
       | Failure exn -> raise exn
```

For example, suppose we wish to process in input file. The file should be opened, processed, and it should be closed afterward (whether or not the processing was successful). We can implement this as follows.

```
let process in_channel =
   ...

let process_file file_name =
   let in_channel = open_in file_name in
      finally process in_channel (fun () -> close_in in_channel)
```

In this example the `finally` function is used to ensure that the `in_channel` is closed after the input file is processed, whether or not the `process` function was successful.

8.3.4 The `exn` type

We close with a somewhat unorthodox use of exceptions completely unrelated to control flow. Exceptions (values of the `exn` type) are first-class values; they can be passed as arguments, stored in data structures, etc. The values in the `exn` type are specified with `exception` definitions. One unique property of the `exn` type is that it is *open* so that new exceptions can be declared when desired. This mechanism can be used to provide a kind of dynamic typing, much like the open unions discussed in Section **??**.

For example, suppose we want to define a list of values, where the type of the values can be extended. Initially, we might want lists containing strings and integers, and suppose we wish to define a `succ` function that increments every integer in the list.

```
# exception String of string;;
# exception Int of int;;
# let succ l =
     List.map (fun x ->
        match x with
           Int i -> Int (i + 1)
         | _ -> x) l;;
val succ : exn list -> exn list = <fun>
# let l = succ [String "hello"; Int 1; Int 7];;
val l : exn list = [String "hello"; Int 2; Int 8]
```

Later, we might also decide to add floating-point numbers to the list, with their own successor function.

```
# exception Float of float;;
exception Float of float
# let succ_float l =
     List.map (fun x ->
        match x with
           Float y -> Float (y +. 1.0)
         | _ -> x) l;;
val succ_float : exn list -> exn list = <fun>
# succ_float (Float 2.3 :: l);;
- : exn list = [Float 3.3; String "hello"; Int 2; Int 8]
```

The main purpose of this example is to illustrate properties of exception values. In cases where extendable unions are needed, the use of open union types is more appropriate. Needless to say, it can be quite confusing to encounter data structures constructed from exceptions!

Chapter 9

Input and Output

The I/O library in OCaml is fairly expressive, including a `Unix` library that implements most of the portable Unix system calls. In this chapter, we'll cover many of the standard built-in I/O functions.

The I/O library uses two data types: the `in_channel` is the type of I/O channels from which characters can be read, and the `out_channel` is an I/O channel to which characters can be written. I/O channels may represent files, communication channels, or some other device; the exact operation depends on the context.

At program startup, there are three channels open, corresponding to the standard file descriptors in Unix.

```
val stdin : in_channel
val stdout : out_channel
val stderr : out_channel
```

9.1 File opening and closing

There are two functions to open an output file: the `open_out` function opens a file for writing text data, and the `open_out_bin` opens a file for writing binary data. These two functions are identical on a Unix system. On a Macintosh or Windows system, the `open_out` function performs

line termination translation (why do all these systems use different line terminators?), while the `open_out_bin` function writes the data exactly as written. These functions raise the `Sys_error` exception if the file can't be opened; otherwise they return an `out_channel`.

A file can be opened for reading with the functions `open_in` and `open_in_bin`.

```
val open_out : string -> out_channel
val open_out_bin : string -> out_channel
val open_in : string -> in_channel
val open_in_bin : string -> in_channel
```

The `open_out_gen` and `open_in_gen` functions can be used to perform more sophisticated file opening. The function requires an argument of type `open_flag` that describes exactly how to open the file.

```
type open_flag =
    Open_rdonly | Open_wronly | Open_append
  | Open_creat | Open_trunc | Open_excl
  | Open_binary | Open_text | Open_nonblock
```

These opening modes have the following interpretation.

Open_rdonly open for reading

Open_wronly open for writing

Open_append open for appending

Open_creat create the file if it does not exist

Open_trunc empty the file if it already exists

Open_excl fail if the file already exists

Open_binary open in binary mode (no conversion)

Open_text open in text mode (may perform conversions)

Open_nonblock open in non-blocking mode

The `open_in_gen` and `open_out_gen` functions have types

```
val open_in_gen : open_flag list -> int -> string -> in_channel.
val open_out_gen : open_flag list -> int -> string -> out_channel.
```

The `open_flag list` describe how to open the file, the `int` argument describes the Unix mode to apply to the file if the file is created, and the `string` argument is the name of the file.

The closing operations `close_out` and `close_in` close the channels. If you forget to close a file, the garbage collector will eventually close it for you. However, it is good practice to close the channel manually when you are done with it.

```
val close_out : out_channel -> unit
val close_in : in_channel -> unit
```

9.2 Writing and reading values on a channel

There are several functions for writing values to an `out_channel`. The `output_char` writes a single character to the channel, and the `output_string` writes all the characters in a string to the channel. The `output` function can be used to write part of a string to the channel; the `int` arguments are the offset into the string, and the length of the substring.

```
val output_char : out_channel -> char -> unit
val output_string : out_channel -> string -> unit
val output : out_channel -> string -> int -> int -> unit
```

The input functions are slightly different. The `input_char` function reads a single character, and the `input_line` function reads an entire line, discarding the line terminator. The `input` functions raise the exception `End_of_file` if the end of the file is reached before the entire value could be read.

```
val input_char : in_channel -> char
val input_line : in_channel -> string
val input : in_channel -> string -> int -> int -> int
```

There are also several functions for passing arbitrary OCaml values on a channel opened in binary mode. The format of these values is implementation specific, but it is portable across all standard implementations of OCaml. The `output_byte` and `input_byte` functions write/read a single byte value. The `output_binary_int` and `input_binary_int` functions write/read a single integer value.

The `output_value` and `input_value` functions write/read arbitrary OCaml values. These functions are unsafe! Note that the `input_value` function returns a value of arbitrary type `'a`. OCaml makes no effort to check the type of the value read with `input_value` against the type of the value that was written with `output_value`. If these differ, the compiler will not know, and most likely your program will generate a segmentation fault.

```
val output_byte : out_channel -> int -> unit
val output_binary_int : out_channel -> int -> unit
val output_value : out_channel -> 'a -> unit
val input_byte : in_channel -> int
val input_binary_int : in_channel -> int
val input_value : in_channel -> 'a
```

9.3 Channel manipulation

If the channel is a normal file, there are several functions that can modify the position in the file. The `seek_out` and `seek_in` function change the file position. The `pos_out` and `pos_in` function return the current position in the file. The `out_channel_length` and `in_channel_length` return the total number of characters in the file.

```
val seek_out : out_channel -> int -> unit
val pos_out : out_channel -> int
val out_channel_length : out_channel -> int
val seek_in : in_channel -> int -> unit
val pos_in : in_channel -> int
val in_channel_length : in_channel -> int
```

If a file may contain both text and binary values, or if the mode of the the file is not know when it is opened, the `set_binary_mode_out` and `set_binary_mode_in` functions can be used to change the file mode.

```
val set_binary_mode_out : out_channel -> bool -> unit
val set_binary_mode_in : in_channel -> bool -> unit
```

The channels perform *buffered* I/O. By default, the characters on an `out_channel` are not all written until the file is closed. To force the writing on the buffer, use the `flush` function.

```
val flush : out_channel -> unit
```

9.4 Printf

The regular functions for I/O can be somewhat awkward. OCaml also implements a `printf` function similar to the `printf` in Unix/C. These functions are defined in the library module `Printf`. The general form is given by `fprintf`.

```
val fprintf: out_channel -> ('a, out_channel, unit) format -> 'a
```

Don't be worried if you don't understand this type definition. The `format` type is a built-in type intended to match a format string. The normal usage uses a format string. For example, the following statement will print a line containing an integer i and a string s.

```
fprintf stdout "Number = %d, String = %s\n" i s
```

The strange typing of this function is because OCaml checks the type of the format string and the arguments. For example, OCaml analyzes the format string to tell that the following `fprintf` function should take a `float`, `int`, and `string` argument.

```
# let f = fprintf stdout "Float = %g, Int = %d, String = %s\n";;
Float = val f : float -> int -> string -> unit = <fun>
```

The format specification corresponds roughly to the C specification. Each format argument takes a width and length specifier that corresponds to the C specification.

d or i convert an integer argument to signed decimal

u convert an integer argument to unsigned decimal

x convert an integer argument to unsigned hexadecimal, using lowercase letters.

X convert an integer argument to unsigned hexadecimal, using uppercase letters

s insert a string argument

c insert a character argument

f convert a floating-point argument to decimal notation, in the style *dddd.ddd*

e or E convert a floating-point argument to decimal notation, in the style *d.ddd e+-dd* (mantissa and exponent)

g or G convert a floating-point argument to decimal notation, in style f or e, E (whichever is more compact)

b convert a Boolean argument to the string `true` or `false`

a user-defined printer. It takes two arguments; it applies the first one to the current output channel and to the second argument. The first argument must therefore have type `out_channel -> 'b -> unit` and the second one has type `'b`. The output produced by the function is therefore inserted into the output of `fprintf` at the current point.

`t` same as %a, but takes only one argument (with type `out_channel -> unit`) and applies it to the current `out_channel`.

`%` takes no argument and output one `%` character.

The `Printf` module also provides several additional functions for printing on the standard channels. The `printf` function prints in the channel `stdout`, and `eprintf` prints on `stderr`.

```
let printf = fprintf stdout
let eprintf = fprintf stderr
```

The `sprintf` function has the same format specification as `printf`, but it prints the output to a string and returns the result.

```
val sprintf: ('a, unit, string) format -> 'a
```

9.5 String buffers

The `Buffer` library module provides string buffers. The string buffers can be significantly more efficient that using the native string operations. String buffers have type `Buffer.t`. The type is *abstract*, meaning that the implementation of the buffer is not specified. Buffers can be created with the `Buffer.create` function.

```
type t (* Abstract type of string buffers *)
val create : unit -> t
```

There are several functions to examine the state of the buffer. The `contents` function returns the current contents of the buffer as a string. The `length` function returns the total number of characters stored in the buffer. The `clear` and `reset` function remove the buffer contents; the difference is that `reset` also deallocates the internal storage used to save the current contents.

```
val contents : t -> string
val length : t -> int
val clear : t -> unit
val reset : t -> unit
```

There are also several functions to add values to the buffer. The `add_char` function appends a character to the buffer contents. The `add_string` function appends a string to the contents; there is also an `add_substring` function to append part of a string. The `add_buffer` function appends the contents of another buffer, and the `add_channel` reads input off a channel and appends it to the buffer.

```
val add_char : t -> char -> unit
val add_string : t -> string -> unit
val add_substring : t -> string -> int -> int -> unit
val add_buffer : t -> t -> unit
val add_channel : t -> in_channel -> int -> unit
```

The `output_buffer` function can be used to write the contents of the buffer to an `out_channel`.

```
val output_buffer : out_channel -> t -> unit
```

The `Printf` module also provides formatted output to a string buffer. The `bprintf` function takes a `printf`-style format string, and formats output to a buffer.

```
val bprintf: Buffer.t -> ('a, Buffer.t, unit) format -> 'a
```

Chapter 10

Files, Compilation Units, and Programs

Until now, we have been writing programs using the OCaml toploop. As programs get larger, it is natural to want to save them in files so that they can be re-used and shared with others. There are other advantages to doing so, including the ability to partition a program into multiple files that can be written and compiled separately, making it easier to construct and maintain the program. Perhaps the most important reason to use files is that they serve as *abstraction boundaries* that divide a program into conceptual parts. We will see more about abstraction during the next few chapters as we cover the OCaml module system, but for now let's begin with an example of a complete program implemented in a single file.

10.1 Single-file programs

For this example, let's build a simple program that removes duplicate lines in an input file. That is, the program should read its input a line at a time, printing the line only if it hasn't seen it before.

One of the simplest implementations is to use a list to keep track of which lines have been read.

File: `unique.ml`

```
let rec unique already_read =
  output_string stdout "> ";
  flush stdout;
  let line = input_line stdin in
    if not (List.mem line already_read) then begin
      output_string stdout line;
      output_char stdout '\n';
      unique (line :: already_read)
    end else
      unique already_read;;

(* "Main program" *)
try unique []with
  End_of_file ->
    ();;
```

Example run

```
% ocamlc -o unique unique.ml
% ./unique
> Great Expectations
Great Expectations
> Vanity Fair
Vanity Fair
> The First Circle
The First Circle
> Vanity Fair
> Paradise Lost
Paradise Lost
```

The program can be implemented as a single recursive function that 1) reads a line of input, 2) compares it with lines that have been previously read, and 3) outputs the line if it has not been read. The entire program is implemented in the single file `unique.ml`, shown in Figure 10.1 with an example run.

In this case, we can compile the entire program in a single step with the command `ocamlc -ounique unique.ml`, where `ocamlc` is the OCaml compiler, `unique.ml` is the program file, and the `-o` option is used to specify the program executable `unique`.

10.1.1 Where is the main function?

Unlike C programs, OCaml program do not have a "`main`" function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated. In general, implementation files can contain arbitrary expressions, not just function definitions. For this example, the "main

program" is the **try** expression in the `unique.ml` file, which gets evaluated when the `unique.cmo` file is evaluated.

10.1.2 OCaml compilers

The INRIA OCaml implementation, most likely the one you are using, provides two compilers—the `ocamlc` byte-code compiler, and the `ocamlopt` native-code compiler. Programs compiled with `ocamlc` are *interpreted*, while programs compiled with `ocamlopt` are compiled to native machine code to be run on a specific operating system and machine architecture. While the two compilers produce programs that behave identically functionally, there are a few differences.

1. Compile time is shorter with the `ocamlc` compiler. Compiled byte-code is portable to any operating system and architecture supported by OCaml, without the need to recompile. Some tasks, like debugging, work only with byte-code executables.

2. Compile time is longer with the `ocamlopt` compiler, but program execution is usually faster. Program executables are not portable, and not every operating system and machine architecture is supported.

We generally won't be concerned with the compiler being used, since the two compilers produce programs that behave identically (arapart from performance). During rapid development, it may be useful to use the byte-code compiler because compilation times are shorter. If performance becomes an issue, it is usually a straightforward process to begin using the native-code compiler.

10.2 Multiple files and abstraction

OCaml uses files as a basic unit for providing data hiding and encapsulation, two important properties that can be used to strengthen the guarantees provided by the implementation. We will see more about data hiding and encapsulation in Chapter 11, but for now the important part is that each file can be assigned a *interface* that declares types for all the accessible parts of the implementation, and everything *not* declared is inaccessible outside the file.

In general, a program will have many files and interfaces. An implementation file is defined in a file with a `.ml` suffix, called a *compilation unit*. An interface for a file *filename.ml* is defined in a file named *filename.mli*. There are four major steps to planning and building a program.

1. Decide how to *factor* the program into separate parts. Each part will be implemented in a separate compilation unit.

2. Implement each of compilation units as a file with a `.ml` suffix, and optionally define an interface for the compilation unit in a file with a `.mli` suffix.

3. Compile each file and interface with the OCaml compiler.

4. Link the compiled files to produce an executable program.

One nice consequence of implementing the parts of a program in separate files is that each file can be compiled separately. When a project is modified, only the files that are affected must be recompiled; there is there is usually no need to recompile the entire project.

Getting back to the example `unique.ml`, the implementation is already too concrete. We chose to use a list to represent the set of lines that have been read, but one problem with using lists is that checking for membership (with `List.mem`) takes time linear in the length of the list, which means that the time to process a file is quadratic in the number of lines in the file! There are clearly better data structures than lists for the set of lines that have been read.

As a first step, let's partition the program into two files. The first file `set.ml` is to provide a generic implementation of sets, and the file `unique.ml` provides the `unique` function as before. For now, we'll keep the list representation in hopes of improving it later—for now we just want to factor the project.

The new project is shown in Figure 10.2.1. We have split the set operations into a file called `set.ml`, and instead of using the `List.mem` function we now use the `Set.mem` function. This naming convention is standard throughout OCaml—the way to refer to a definition f in a file named *filename* is by capitalizing the filename and using the infix `.` operator to project the value. The `Set.mem` expression refers to the `mem` function in the `set.ml` file. In fact, the `List.mem` function is the same way. The OCaml standard library contains a file `list.ml` that defines a function `mem`.

Compilation is now several steps. In the first step, the `set.ml` and `unique.ml` files are compiled with the `-c` option, which specifies that the compiler should produce an intermediate file

File: set.ml

```
let empty = []
let add x l = x :: l
let mem x l = List.mem x l
```

File: unique.ml

```
let rec unique already_read =
    output_string stdout "> ";
    flush stdout;
    let line = input_line stdin in
      if not (Set.mem line already_read) then begin
        output_string stdout line;
        output_char stdout '\n';
        uniq (line :: already_read)
      end else
        unique already_read;;

(* Main program *)
try unique []with
    End_of_file ->
      ();;
```

Example run

```
% ocamlc -c set.ml
% ocamlc -c unique.ml
% ocamlc -o unique set.cmo unique.cmo
% ./unique
> Adam Bede
Adam Bede
> A Passage to India
A Passage to India
> Adam Bede
> Moby Dick
Moby Dick
```

with a `.cmo` suffix. These files are then linked to produce an executable with the command `ocamlc -o unique set.cmo unique.cmo`.

The order of compilation and linking here is significant. The `unique.ml` file refers to the `set.ml` file by using the `Set.mem` function. Due to this dependency, the `set.ml` file must be compiled before the `unique.ml` file, and the `set.cmo` file must appear before the `unique.cmo` file during linking. Note that cyclic dependencies are *not allowed*. It is not legal to have a file `a.ml` refer to a value `B.x`, and a file `b.ml` that refers to a value `A.y`.

10.2.1 Defining a signature

One of the reasons for factoring the program was to be able to improve the implementation of sets. To begin, we should make the type of sets *abstract*—that is, we should hide the details of how it is implemented so that we can be sure the rest of the program does not uninitentionally depend on the implementation details. To do this, we can define an abstract signature for sets, in a file `set.mli`.

A signature should declare types for each of the values that are publicly accessible in a module, as well as any needed type declarations or definitions. For our purposes, we need to define a polymorphic type of sets `'a set` abstractly. That is, in the signature we will declare a type `'a set` without giving a definition, preventing other parts of the program from knowing, or depending on, the particular representation of sets we have chosen. The signature also needs to declare types for the public values `empty`, `add`, and `mem` values, as a declaration of the form "**val** *name* : *type*". The complete signature is shown in Figure **??**. The implementation remains mostly unchanged, except that a specific, concrete type definition must be given for the type `'a set`.

Now, when we compile the program, we first compile the interface file `set.mli`, then the implementations `set.ml` and `uniq.ml`. But something has changed, the `uniq.ml` file no longer compiles! Following the error message, we find that the error is due to the expression `line :: already_read`, which uses a `List` operation instead of a `Set` operation. Since the `'a set` type is abstract, it is now an error to treat the set as a list, and the compiler complains appropriately.

Changing this expression to `Set.add line already_read` fixes the error. Note that, while the `set.mli` file must be compiled, it does not need to be specified during linking

File: set.mli

```
type 'a set
val empty : 'a set
val add : 'a ->'a set ->'a set
val mem : 'a ->'a set ->bool
```

File: set.ml

```
type 'a set = 'a list
let empty = []
let add x l = x :: l
let mem x l = List.mem x l
```

Example run

```
% ocamlc -c set.mli
% ocamlc -c set.ml
% ocamlc -c uniq.ml
File "uniq.ml", line 8, characters 14-36:
This expression has type 'a list but is
    here used with type string Set.set
```

File: uniq.ml

```
let rec uniq already_read =
    output_string stdout "> ";
    flush stdout;
    let line = input_line stdin in
      if not (Set.mem line already_read) then begin
        output_string stdout line;
        output_char stdout '\n';
        (* uniq (line :: already_read) *)
        uniq (Set.add line already_read)
      end else
        uniq already_read;;

(* Main program *)
try uniq Set.empty with
    End_of_file ->
      ();;
```

Example run

```
% ocamlc -c set.mli
% ocamlc -c set.ml
% ocamlc -c uniq.ml
% ocamlc -o uniq set.cmo uniq.cmo
% ./uniq
> Siddhartha
Siddhartha
> Siddhartha
> Siddharta
Siddharta
```

File: set.mli

```
type 'a set
type 'a choice =
    | Element of 'a
    | Empty
val empty : 'a set
val add : 'a ->'a set ->'a set
val mem : 'a ->'a set ->bool
val choose : 'a set ->'a choice
```

File: set.ml

```
type 'a set = 'a list
type 'a choice =
    | Element of 'a
    | Empty
let empty = []
let add x l = x :: l
let mem x l = List.mem x l
let choose = function
    | x :: _ ->Element x
    | []->Empty
```

`ocamlc -o uniq set.cmo uniq.cmo`.

At this point, the `set.ml` implementation is fully abstract, making it easy to replace the implementation with a better one (for example, the implementation of sets using red-black trees in Chapter **??**).

10.2.2 Transparent type definitions

In some cases, abstract type definitions are too strict. There are times when we want a type definition to be *transparent*—that is, visible outside the file. For example, suppose we wanted to add a `choose` function to the set implementation, where, given a set s, the expression (`choose` s) returns some element of the set if the set is non-empty, and nothing otherwise. One possible way to write this function is to define a union type `choice` that defines the two cases, as shown in Figure 10.2.2.

The type definition for `choice` must be transparent (otherwise there isn't much point in defining the function). For the type to be transparent, the signature simply need to provide the definition. The implementation must contain the *same* definition.

10.3 Some common errors

As you develop programs with several files, you will undoubtably encounter some errors. The following subsections list some of the more common errors.

10.3.1 Interface errors

When a file is compiled, the compiler compares the implementation with the signature in a `.cmi` file compile from the `.mli` file. If a definition does not match the signature, the compiler will print an error and refuse to compile the file.

Type errors

For example, suppose we had reversed the order of arguments in the `Set.add` function so that the set argument is first.

```
let add s x = x :: s
```

When we compile the file, we get an error. The compiler prints the types of the mismatched values, and exits with an error code.

```
% ocamlc -c set.mli
% ocamlc -c set.ml
The implementation set.ml does not match the interface set.cmi:
Values do not match:
  val add : 'a list -> 'a -> 'a list
is not included in
  val add : 'a -> 'a set -> 'a set
```

The first declaration is the type the compiler infered for the definition; the second declaration is from the signature. Note that the definition's type is not abstract (using `'a list` instead of `'a set`). For this example, it is clear that the argument ordering doesn't match, and the definition or the signature must be changed.

Missing definition errors

Another common error occurs when a function declared in the signature is not defined in the implementation. For example, suppose we had defined an `insert` function istead of an `add` function. In this case, the compiler prints the name of the missing function, and exits with an error code.

```
% ocamlc -c set.ml
The implementation set.ml does not match the interface set.cmi:
The field ‘add’ is required but not provided
```

Type definition mismatch errors

Transparent type definitions in the signature can also cause an error if the type definition in the implementation does not match. For example, in the definition of the `choice` type, suppose we had declared the cases in different orders.

File: set.mli	File: set.ml
type 'a set	**type** 'a set = 'a list
type 'a choice =	**type** 'a choice =
| Element of 'a	| Empty
| Empty	| Element of 'a
⋮	⋮

When we compile the `set.ml` file, the compiler will produce an error with the mismatched types.

```
% ocamlc -c set.mli
% ocamlc -c set.ml
The implementation set.ml does not match the interface set.cmi:
Type declarations do not match:
  type 'a choice = Empty | Element of 'a
is not included in
  type 'a choice = Element of 'a | Empty
```

The type definitions are required to be *exactly* the same. Some programmers find this duplication of type definitions to be annoying. While it is difficult to avoid all duplication of type definitions, one common solution is to define the transparent types in a separate `.ml` file without a signature, for example by moving the definition of `'a choice` to a file `set_types.ml`. By default, when an interface file does not exist, all definitions from the implementation are fully visible. As a result, the type in `set_types.ml` needs to be defined just once.

Compile dependency errors

The compiler will also produce errors if the compile state is inconsistent. Each time an interface is compile, all the files that uses that interface must be recompiled. For example, suppose we update the `set.mli` file, and recompile it and the `uniq.ml` file (but we forget to recompile the `set.ml` file). The compiler produces the following error.

```
% ocamlc -c set.mli
% ocamlc -c uniq.ml
% ocamlc -o uniq set.cmo uniq.cmo
Files uniq.cmo and set.cmo make inconsistent
assumptions over interface Set
```

It takes a little work to detect the cause of the error. The compiler says that the files make inconsistent assumptions for interface `Set`. The interface is defined in the file `set.cmi`, and so this error message states that at least one of `set.ml` or `uniq.ml` needs to be recompiled. In general, we don't know which file is out of date, and the best solution is usually to recompile them all.

10.4 Using `open` to expose a namespace

Using the full name *`File_name.name`* to refer to the values in a module can get tedious. The `open` *`File_name`* statement can be used to "open" an interface, allowing the use of unqualified names for types, exceptions, and values. For example, the `unique.ml` module can be somewhat simplified by using the `open` directive for the `Set` module. In the following listing, the underlined variables refer to the value in the Set implementation.

File: uniq.ml

```
open Set
let rec uniq already_read =
    output_string stdout "> ";
    flush stdout;
    let line = input_line stdin in
      if not (mem line already_read) then begin
        output_string stdout line;
        output_char stdout '\n';
        uniq (add line already_read)
      end else
        uniq already_read;;

(* Main program *)
try uniq empty with
    End_of_file ->
      ();;
```

Sometimes multiple `open`ed files will define the same name. In this case, the *last* file with an `open` statement will determine the value of that symbol. Fully qualified names (of the form *`File_name.name`*) may still be used even if the file has been opened. Fully qualified names can be used to access values that may have been hidden by an `open` statement.

10.4.1 A note about `open`

Be careful with the use of `open`. In general, fully qualified names provide more information, specifying not only the name of the value, but the name of the module where the value is defined. For example, the `Set` and `List` modules both define a `mem` function. In the `Uniq` module we just defined, it may not be immediately obvious to a programmer that the `mem` symbol refers to `Set.mem`, not `List.mem`.

In general, you should use `open` statement sparingly. Also, as a matter of style, it is better

not to `open` most of the library modules, like the `Array`, `List`, and `String` modules, all of which define methods (like `create`) with common names. Also, you should *never* `open` the `Unix`, `Obj`, and `Marshal` modules! The functions in these modules are not completely portable, and the fully qualified names identify all the places where portability may be a problem (for instance, the Unix `grep` command can be used to find all the places where `Unix` functions are used).

The behavior of the `open` statement is not like an `#include` statement in C. An implementation file `mod.ml` should not include an `open Mod` statement. One common source of errors is defining a type in a `.mli` interface, then attempting to use `open` to "include" the definition in the `.ml` implementation. This won't work—the implementation must include an identical type definition. True, this might be considered to be an annoying feature of OCaml. But it preserves a simple semantics: the implementation must provide a definition for each declaration in the signature.

10.5 Debugging a program

The `ocamldebug` program can be used to debug a program compiled with `ocamlc`. The `ocamldebug` program is a little like the GNU `gdb` program; it allows breakpoints to be set. When a breakpoint is reached, control is returned to the debugger so that program variables can be examined.

To use `ocamldebug`, the program must be compiled with the `-g` flag.

```
% ocamlc -c -g set.mli
% ocamlc -c -g set.ml
% ocamlc -c -g uniq.ml
% ocamlc -o uniq -g set.cmo uniq.cmo
```

The debugger is invoked using by specifying the program to be debugged on the `ocamldebug` command line.

```
% ocamldebug ./uniq
Objective Caml Debugger version 3.08.3

(ocd) help
List of commands :cd complete pwd directory kill help quit shell run reverse
step backstep goto finish next start previous print display source break
delete set show info frame backtrace bt up down last list load_printer
install_printer remove_printer
```

There are several commands that can be used. The basic commands are `run`, `step`, `next`, `break`, `list`, `print`, and `goto`.

run Start or continue execution of the program.

break @ module linenum Set a breakpoint on line *linenum* in module *module*.

list display the lines around the current execution point.

print expr Print the value of an expression. The expression must be a variable.

goto time Execution of the program is measured in time steps, starting from 0. Each time a breakpoint is reached, the debugger will print the current time. The `goto` command may be used to continue execution to a future time, or to a *previous* timestep.

step Go forward one time step.

next If the current value to be executed is a function, evaluate the function, a return control to the debugger when the function completes. Otherwise, step forward one time step.

For debugging the `uniq` program, we need to know the line numbers. Let's set a breakpoint in the `uniq` function, which starts in line 1 in the `Uniq` module. We'll want to stop at the first line of the function.

```
(ocd) break @ Uniq 1
Loading program... done.
Breakpoint 1 at 21656 : file uniq.ml, line 2, character 4
(ocd) run
Time : 12 - pc : 21656 - module Uniq
Breakpoint : 1
2    <|b|>output_string stdout "> ";
(ocd) n
Time : 14 - pc : 21692 - module Uniq
2    output_string stdout "> "<|a|>;
(ocd) n
> Time : 15 - pc : 21720 - module Uniq
3    flush stdout<|a|>;
(ocd) n
Robinson Crusoe
Time : 29 - pc : 21752 - module Uniq
5       <|b|>if not (Set.mem line already_read) then begin
(ocd) p line
line : string = "Robinson Crusoe"
```

Next, let's set a breakpoint just before calling the `uniq` function recursively.

```
(ocd) list
1 let rec uniq already_read =
2    output_string stdout "> ";
3    flush stdout;
4    let line = input_line stdin in
5       <|b|>if not (Set.mem line already_read) then begin
6          output_string stdout line;
7          output_char stdout '\n';
8          uniq (Set.add line already_read)
9       end
10       else
11          uniq already_read;;
12
13 (* Main program *)
14 try uniq Set.empty with
15    End_of_file ->
16       ();;
Position out of range.
(ocd) break @ 8
Breakpoint 2 at 21872 : file uniq.ml, line 8, character 42
(ocd) run
Time : 38 - pc : 21872 - module Uniq
Breakpoint : 2
8          uniq (Set.add line already_read)<|a|>
```

Next, suppose we don't like adding this line. We can go back to time `15` (the time just before the `input_line` function is called).

```
(ocd) goto 15
> Time : 15 - pc : 21720 - module Uniq
3    flush stdout<|a|>;
(ocd) n
Mrs Dalloway
Time : 29 - pc : 21752 - module Uniq
5       <|b|>if not (Set.mem line already_read) then begin
```

Note that when we go back in time, the program prompts us again for an input line. This is due to way time travel is implemented in `ocamldebug`. Periodically, the debugger takes a checkpoint of the program (using the Unix `fork()` system call). When reverse time travel is requested, the debugger restarts the program from the closest checkpoint before the time requested. In this case, the checkpoint was taken before the call to `input_line`, and the program resumption requires another input value.

We can continue from here, examining the remaining functions and variables. You may wish

to explore the other features of the debugger. Further documentation can be found in the OCaml reference manual.

Chapter 11

The OCaml Module System

As we saw in the previous chapter, programs can be divided into parts that can be implemented in files, and each file can be given an interface that specifies what its public types and values are. Files are not the only way to partition a program, OCaml also provides a *module system* that allows programs to be partitioned even within a single file. There are three key parts in the module system: *signatures*, *structures*, and *functors*, where signatures correspond to interfaces, structures correspond to implementations, and functors are functions over structures. In this chapter, we will discuss the first two; we'll leave discussion of functors in Chapter 12.

There are several reasons for using the module system. Perhaps the simplest reason is that each structure has its own namespace, so name conflicts are less likely when modules are used. Another reason is that abstraction can be specified explicitly by assigning a signature to a structure. To begin, let's return to the `unique` example from the previous chapter, this time using modules instead of separate files.

11.1 Simple modules

Named structures are defined with the `module` and `struct` keywords using the following syntax.

`module` *Name* `=` `struct` *implementation* `end`

The module *Name* must begin with an uppercase letter. The *implementation* can include definition that might occur in a `.ml` file. Let's return to the `unique.ml` example from the previous chapter, using a simple list-based implementation of sets. This time, instead of defining the set data structure in a separate file, let's define it as a module, called `Set`, using an explicit `module/struct` definition. The program is shown in Figure 12.1.

In this new program, the main role of the module `Set` is to collect the set functions into a single block of code that has an explicit name. The values are now named using the module name as a prefix as `Set.empty`, `Set.add`, and `Set.mem`. Otherwise, the program is as before.

One problem with this program is that the implementation of the `Set` module is visible. As usual, we would like to hide the type of set, making it easier to replace the implementation later if we wish to improve its performance. To do this, we can assign an explicit signature that hides the set implementation. A named signature is defined with a `module type` definition.

`module type` *Name* `= sig` *signature* `end`

As before, the name of the signature must begin with an uppercase letter. The signature can contain any of the items that can occur in an interface `.mli` file. For our example, the signature should include an abstract type declaration for the `'a set` type and `val` declarations for each of the values. The `Set` module's signature is constrained by specifying the signature after a colon in the module definition **module** Set : SetSig = **struct** $\cdots$ **end**, as shown in Figure 11.1.

11.2 Module definitions

In general, structures and signatures are just like implementation files and their interfaces. Structures are allowed to contain any of the definitions that might occur in a implementation, including any of the following.

- `type` definitions
- `exception` definitions
- `let` definitions

File: unique.ml

```
module Set = struct
    let empty = []
    let add x l = x :: l
    let mem x l = List.mem x l
end;;

let rec unique already_read =
    output_string stdout "> ";
    flush stdout;
    let line = input_line stdin in
      if not (Set.mem line already_read) then begin
        output_string stdout line;
        output_char stdout '\n';
        uniq (Set.add line already_read)
      end else
        unique already_read;;

(* Main program *)
try unique Set.empty with
    End_of_file ->
      ();;
```

Example run

```
% ocamlc -o unique unique.ml
% ./unique
> Adam Bede
Adam Bede
> A Passage to India
A Passage to India
> Adam Bede
> Moby Dick
Moby Dick
```

Signature definition

```
module type SetSig = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val mem : 'a -> 'a set -> bool
end;;
```

Structure definition

```
module Set : SetSig = struct
  type 'a set = 'a list
  let empty = []
  let add x l = x :: l
  let mem x l = List.mem x l
end;;
```

- `open` statements to open the namespace of another module
- `include` statements that include the contents of another module
- signature definitions
- nested structure definitions

Similarly, signatures may contain any of the declarations that might occur in an interface file, including any of the following.

- `type` declarations
- `exception` definitions
- `val` declarations
- `open` statements to open the namespace of another signature
- `include` statements that include the contents of another signature
- nested signature declarations

We have seen most of these constructs before. However, one new construct we haven't seen is `include`, which allows the entire contents of a structure or signature to be included in another. The `include` statement can be used to create modules and signatures that re-use existing definitions.

Signature definition

```
module type ChooseSetSig = sig
  include SetSig
  type 'a choice = Element of 'a | Empty
  val choose : 'a set -> 'a choice
end;;
```

Inferred type (from the toploop)

```
module type ChooseSetSig = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val mem : 'a -> 'a set -> bool
  type 'a choice = Element of 'a | Empty
  val choose : 'a set -> 'a choice
end;;
```

11.2.1 Using include to extend modules

Suppose we wish to defined a new kind of sets `ChooseSet` that have a `choose` function that returns an element of the set if one exists. Instead of re-typing the entire signature, we can use the `include` statement to include the existing signature, as shown in Figure 11.2.1. The resulting signature includes all of the types and declarations from `SetSig` as well as the new (transparent) type definition `'a choice` and function declaration `val choose`. For this example, we are using the toploop to display the infered signature for the new module.

11.2.2 Using include to extend implementations

The `include` statement can also be used in implementations. For our example, however, there is a problem. The straightforward approach in defining a module `ChooseSet` is to include the `Set` module, then define the new type `'a choice` and the new function `choose`. The result of this attempt is shown in Figure 11.2.2, where the toploop prints out an extensive error message (the toploop prints out the full signature, which we have elided in **sig** $\cdots$ **end**).

The problem is apparent from the last few lines of the error message—the `choose` function has type `'a list -> 'a choice`, not `'a set -> 'a choice` as it should. The issue is that we included the *abstract* module `Set`, where the type `'a set` is an abstract type, not a list.

Module definition

```
module ChooseSet : ChooseSetSig = struct
  include Set
  type 'a choice = Element of 'a | Empty
  let choose = function
    | x :: _ -> Element x
    | []-> Empty
end;;
```

Inferred type (from the toploop)

```
Signature mismatch:
Modules do not match:
  sig ... end
is not included in
  ChooseSetSig
Values do not match:
  val choose : 'a list -> 'a choice
is not included in:
  val choose : 'a set -> 'a choice
```

One solution is to manually copy the code from the `Set` module into the `ChooseSet` module. This has its drawbacks of course. We aren't able to re-use the existing implementation, our code base gets larger, etc. If we have access to the original non-abstract set implementation, there is another solution–we can just include the non-abstract set implementation, where it is known that the set is represented as a list.

Suppose we start with a non-abstract implementation `SetInternal` of sets as lists. Then the module `Set` is the same implementation, with the signature `SetSig`; and the `ChooseSet` includes the `SetInternal` module instead of `Set`. Figure 11.2.2 shows the definitions in this order, together with the types inferred by the toploop.

Note that for the module `Set` it is not necessary to use a **struct** ··· **end** definition because the `Set` module is *equivalent* to the `SetInternal` module, it just has a different signature. The modules `Set` and `ChooseSet` are "friends," in that they share internal knowledge of each other's implementation, while keeping their public signatures abstract.

11.3 Abstraction, friends, and module hiding

So far, we have seen that modules provide two main features, 1) the ability to divide a program into separate program units (modules) that each have a separate namespace, and 2) the ability to

Module definitions

```
module SetInternal = struct
  type 'a set = 'a list
  let empty = []
  let add x l = x :: l
  let mem x l = List.mem x l
end;;

module Set : SetSig = SetInternal

module ChooseSet : ChooseSetSig = struct
  include SetInternal
  type 'a choice = Element of 'a | Empty
  let choose = function
    | x :: _ -> Element x
    | []-> Empty
end;;
```

Inferred types (from the toploop)

```
module SetInternal : sig
  type 'a set = 'a list
  val empty : 'a list
  val add : 'a -> 'a list -> 'a list
  val mem : 'a -> 'a list -> bool
end;;

module Set : SetSig

module ChooseSet : ChooseSetSig
```

assign signatures that make each structure partially or totally abstract. In addition, as we have seen in the previous example, a structure like `SetInternal` can be given more than one signature (the module `Set` is equal to `SetInternal` but it has a different signature).

Another frequent use of modules uses nesting to define multiple levels of abstraction. For example, we might define a module container in which several modules are defined and implementation are visible, but the container type is abstract. This is akin to the C++ notion of "friend" classes, where a set of friend classes may mutually refer to class implementations, but the publicly visible fields remain protected.

In our example, there isn't much danger in leaving the `SetInternal` module publicly accessible. A `SetInternal.set` can't be used in place of a `Set.set` or a `ChooseSet.set`, because the latter types are abstract. However, there is a cleaner solution that nests the `Set` and `ChooseSet` structures in an outer `Sets` module. The signatures are left unconstrained within the `Sets` module, allowing the `ChooseSet` structure to refer to the implementation of the `Set` structure, but the signature of the `Sets` module is constrained. The code for this is shown in Figure11.3.

There are a few things to note of this definition.

1. The `Sets` module uses an *anonymous* signature (meaning that the signature has no name). Anonymous signatures and **struct** implementations are perfectly acceptable any place where a signature or structure is needed.

2. Within the `Sets` module the `Set` and `ChooseSet` modules are not constrained, so that their implementations are public. This allows the `ChooseSet` to refer to the `Set` implementation directly (so in this case, the `Set` and `ChooseSet` modules are firends). The signature for the `Sets` module makes them abstract.

11.3.1 Using include with incompatible signatures

In our current example, it might seem that there isn't much need to have two separate modules `ChooseSet` (with `choice`) and `Set` (without `choice`). In practice it is perhaps more likely that we would simply add a `choice` function to the `Set` module. The addition would not affect any existing code, since any existing code doesn't refer to the `choice` function anyway.

Module definitions

```
module Sets : sig
  module Set : SetSig
  module ChooseSet : ChooseSetSig
end = struct
  module Set = struct
    type 'a set = 'a list
    let empty = []
    let add x l = x :: l
    let mem x l = List.mem x l
  end
  module ChooseSet = struct
    include Set
    type 'a choice = Element of 'a | Empty
    let choose = function
      | x :: _ -> Element x
      | []-> Empty
  end
end;;
```

Inferred types (from the toploop)

```
module Sets : sig
  module Set : SetSig
  module ChooseSet : ChooseSetSig
end
```

Signature

```
module type Set2Sig = sig
  type 'a set
  val empty : 'a set
  val add : 'a set -> 'a -> 'a set
  val mem : 'a -> 'a set -> bool
end
```

Implementation

```
module Set2 : Set2Sig = struct
  include Set
  let add l x = Set.add x l
end;;
```

Surprisingly, this kind of example occurs in practice more than it might seem, due to programs being developed with incompatible signatures. For example, suppose we are writing a program that is going to make use of two independently-developed libraries. Both libraries have their own `Set` implementation, and we decide that we would like to use a single `Set` implementation in the combined program. Unfortunately, the signatures are incompatible—in the first library, the `add` function was defined with type `val add : 'a -> 'a set -> 'a set`; but in the second library, it was defined with type `val add : 'a set -> 'a -> 'a set`. Let's say that the first library uses the desired signature. Then, one solution would be to hunt through the second library, finding all calls to the `Set.add` function, reordering the arguments to fit a common signature. Of course, the process is tedious, and it is unlikely we would want to do it.

An alternative is to *derive* a wrapper module `Set2` for use in the second library. The process is simple, 1) `include` the `Set` module, and 2) redefine the `add` to match the desired signature; this is shown in Figure 11.3.1.

The `Set2` module is just a wrapper. Apart from the `add` function, the types and values in the `Set` and `Set2` modules are the same, and the `Set2.add` function simply reorders the arguments before calling the `Set.add` function. There is little or no performance penalty for the wrapper—in most cases the native-code OCaml compiler will *inline* the `Set2.add` function (in other words, it will perform the argument reordering at compile time).

11.4 Sharing constraints

Module definition

```
module Set2 : Set2Sig with type 'a set = 'a Set.set
= struct
  include Set
  let add l x = Set.add x l
end
```

Toploop

```
# let s = Set2.add Set.empty 1;;
val s : int Set2.set = <abstr>
# Set.mem 1 s;;
- : bool = true
```

There is one remaining problem with this example. In the combined program, the first library uses the original `Set` module, and the second library uses `Set2`. It is likely that we will want to pass values, including sets, from one library to the other. However, as defined, the `'a Set.set` and `'a Set2.set` types are distinct abstract types, and it is an error to use a value of type `'a Set.set` in a place where a value of type `'a Set2.set` is expected, and *vice-versa*. The following error message is typical.

```
# Set2.add Set.empty 1;;
This expression has type 'a Set.set
   but is here used with type 'b Set2.set
```

Of course, we might want the types to be distinct. But in this case, it is more likely that we want the definition to be transparent. We know that the two kinds of sets are really the same—`Set2` is really just a wrapper for `Set`. How do we establish the equivalence of `'a Set.set` and `'a Set2.set`.

The solution is called a *sharing constraint*. The syntax for a sharing constraint uses the `with` keyword to specify a type equivalence for a module signature in the following form.

signature ::= *signature* **with type** *typename* **=** *type*.

In this particular case, we wish to say that the `'a Set2.set` type is equal to the `'aSet.set` type, which we can do by adding a sharing constraint when the `Set2` module is defined, as shown in Figure 11.4.

The constraint specifies that the types `'a Set2.set` and `'a Set.set` are the same. In other words, they *share* a common type. Since the two types are equal, set values can be freely passed between the two set implementations.

11.5 Summary

JYH: still to write.

- Simple modules
- Modules with multiple signatures
- Sharing constraints

11.6 Exercises

1. One could argue that sharing constraints are never necessary for unparameterized modules like the ones in this chapter. In the example of Figure 11.4, there are at least two other solutions that allow the `Set2` and `Set` modules to share values, without having to use sharing constraints. Present two alternate solutions without sharing constraints.

2. In OCaml 3.08.3, signatures can apparently contain multiple declarations for the same value.

```
# module type ASig = sig
   val x : int
   val x : bool
  end;;
module type ASig = sig val x : int val x : bool end
```

   However, these declarations are really just an illusion, only the first declaration counts, any others are ignored. Based on what you know, is this behavior expected? If multiple declarations are allowed, which one should be the "real" declaration?

3. Unlike `val` declarations, `type` declarations must have distinct names in any structure or signature.

```
# module type ASig = sig
     type t = int
     type t = bool
  end;;
Multiple definition of the type name t.
Names must be unique in a given structure or signature.
```

While this particular example may seem silly, the real problem is that all modules included with **include** must have disjoint type names.

```
# module type XSig = sig
    type t
    val x : t
  end;;
# module A : XSig = struct
    type t = bool
    let x = false
  end;;
# module B : XSig = struct
    type t = int
    let x = 0
  end;;
# module C = struct
    include A
    include B
  end;;
Multiple definition of the type name t.
Names must be unique in a given structure or signature.
```

Is this a problem? If it is not, argue that conflicting includes should not be allowed in practice. If it is, propose a possible solution to the problem.

Chapter 12

Functors

Modules often refer to other modules. The modules we saw in Chapter 11 referred to other modules by name. Thus, all the module references we've seen up to this point have been to specific, constant modules.

It's also possible in OCaml to write modules that take one or more module parameters. These parameterized modules, called *functors*, might be thought of as "module skeletons." To be used, functors are instantiated by supplying actual module arguments for the functor's module parameters (similar to supplying arguments in a function call).

To illustrate the use of a parameterized module, let's return to the set implementation we have been using in the previous two chapters. One of the problems with that implementation is that the elements are compared using the OCaml built-in equality function `=`. For example, we might want a set of strings where equality is case-insensitive, or we might want a set of floating-point numbers where equality is to within a small constant. Rather than re-implementing a new set for each of these cases, we can implement it as a functor, where the equality function is provided as a parameter. An example is shown inf Figure 12.1.

In this example, the module `MakeSet` is a functor that takes another module `Equal` with signature `EqualSig` as an argument. The `Equal` module provides two things—a type of elements, and a function `equal` to compare two elements. The body of the functor `MakeSet` is much the same as the previous set implementations we have seen, except now the elements are compared using the

Set functor

```
module type EqualSig = sig
  type t
  val equal : t -> t -> bool
end;;

module MakeSet (Equal : EqualSig) =
struct
  open Equal
  type elt = Equal.elt
  type t = elt list
  let empty = []
  let rec mem x = function
    | x' :: l -> equal x x' || mem x l
    | []-> false
  let add x l = x :: l
  let rec find x = function
    | x' :: l when equal x x' -> x'
    | _ :: l -> find x l
    | []-> raise Not_found
end;;
```

Building a specific set

```
module StringCaseEqual = struct
  type t = string
  let equal s1 s2 =
    String.lowercase s1 = String.lowercase s2
end;;
module SSet = MakeSet (StringCaseEqual);;
```

Using the set

```
# let s = SSet.add "Great Expectations" SSet.empty;;
val s : string list = ["Great Expectations"]
# SSet.mem "great eXpectations" s;;
- : bool = true
# SSet.find "great eXpectations" s;;
- StringCaseEqual.t = "Great Expectations"
```

function `equal x x'` instead of the builtin-equality `x =x'`.

To construct a specific set, we first build a module that implements the equality function (in this case, the module `StringCaseEqual`), then apply the `MakeSet` functor module to construct the set module.

In many ways, functors are just like functions at the module level, and they can be used just like functions. However, there are a few things to keep in mind.

1. A functor parameter, like `(Equal : EqualSig)` must be a module, or another functor. It is not legal to pass non-module values (like strings, lists, or integers).

2. Syntactically, module and functor identifiers must always be capitalized. Functor parameters, like `(Equal : EqualSig)`, must be enclosed in parentheses, and the signature is required. For functor applications, like `MakeSet (StringCaseEqual)`, the argument must be enclosed in parenthesis.

3. Modules and functors are not first class. That is, they can't be stored in data structures or passed as arguments like other values, and module definitions cannot occur in function bodies. Technically speaking, the primary reason for this restriction is that type checking would become undecidable. Another reason is that module constructions and functor applications are normally computed at compile time, so it would not be legal to have a function compute a module.

Another point to keep in mind is that the new set implementation is no longer polymorphic—it is now defined for a specific type of elements defined by the `Equal` module. This loss of polymorphism occurs frequently when modules are parameterized, because the goal of parameterizing is to define different behaviors for different types of elements. While the loss of polymorphism is inconvenient, in practice, it is rarely an issue because modules can be constructed for each specific type of parameter by using a functor application.

12.1 Sharing constraints

In the `MakeSet` example of Figure 12.1, we omitted the signature for sets. This leaves the set implementation visible (for example, the `SSet.add` function returns a string list). As usual, it would be wise define a signature that hides the implementation, preventing the rest of the program from depending on the implementation details.

Functor signatures are defined the usual way, by specifying the signature after a colon, as shown in Figure 11.1.

Unfortunately, in this attempt, the `SSet` module is actually useless because of type abstraction. In the `SetSig` signature, the type `elt` is abstract, and since the `MakeSet` functor returns a module with signature `SetSig`, the type `SSet.elt` is also abstract. While we know that the type `SSet.elt` is really `string`, we can't make use of the fact.

One solution might be to define a transparent type `type elt = string` in the `SetSig` module, but this would mean that we could only construct sets of strings. Instead, the proper way to fix the problem is to add a constraint on the functor that specifies that the `elt` type produced by the functor is the *same* as the `Equal.elt` type in the argument.

The solution is simpleTo do this, we can use the *sharing constraints* introduced in Section 12.1. The corrected definition of the `MakeSet` functor uses a sharing constraint to specify that the `elt` types of the argument and result modules are the same.

module MakeSet (Equal : EqualSig)
  : SetSig **with type** elt = Equal.t
  = **struct** $\cdots$ **end**;;

The toploop now displays the correct element specification. When we redefine the `SSet` module, we get a working version of finite sets of integers.

Set functor

```
module type EqualSig = sig
  type t
  val equal : t -> t -> bool
end;;

module type SetSig = sig
  type t
  type elt
  val empty : t
  val mem : elt -> t -> bool
  val add : elt -> t -> t
  val find : elt -> t -> elt
end;;

module MakeSet (Equal : EqualSig) : SetSig =
struct
  type elt = Equal.elt
  type t = elt list
  let empty = []
  ...
end;;
```

Building a specific set

```
module StringCaseEqual = struct
  type t = string
  let equal s1 s2 =
    String.lowercase s1 = String.lowercase s2
end;;
module SSet = MakeSet (StringCaseEqual);;
```

Using the set

```
# SSet.empty;;
- : StringSet.t = <abstr>
# let s = SSet.add "Great Expectations" SSet.empty;;
This expression has type string
  but is here used with type
  StringSet.elt = MakeSet(StringCaseEqual).elt
```

```
# module SSet = MakeSet (StringCaseCompare);;
module SSet :
  sig
    type elt = StringCaseCompare.t
    and t = MakeSet(Int).t
    val empty : t
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val find : elt -> t -> elt
  end
# SSet.empty;;
- : IntSet.t = <abstr>
# open SSet;;
# let s = add "Great Expectations" empty;;
val s : SSet.t = <abstr>
# mem "great eXpectations" s;;
- : bool = true
# find "great eXpectations" s;;
- : string = "Great Expectations"
```

12.2 Module re-use using functors

Now that we have successfully constructed the `MakeSet` functor, let's move on to another frequently-used data structure called a *map*. A map is a table that associates a value with each element in a set. The data structure provides a function `add` to add an element and its value to the table, as well as a function `find` that retrieves that value associated with an element (or raises the exception `Not_found` if the element is not in the table).

The *map* and *set* data structures are very similar. Since we have implemented sets already, is it possible to re-use the implementation for maps. Once again, we can use functors for this purpose. In this case, we will write a functor that produces a *map* data structure given a comparison function. The code is shown in Figure 12.2.

The `MakeMap` functor takes two parameters, a `Equal` module to compare keys, and a `Value` module that specifies the type of values stored in the table. The functor itself first constructs a `Set` module for `(key, value)` pairs, where the comparison is limited to the keys. Once the `Set` module is constructed, the `Map` functions are simple wrappers around the `Set` functions.

```
module type ValueSig = sig
  type value
end;;

module type MapSig = sig
  type t
  type key
  type value
  val empty : t
  val add : t -> key -> value -> t
  val find : t -> key -> value
end;;
```

A (string, int) map

```
module IntValue = struct
  type value = int
end;;
module StringIntTable =
  MakeMap (EqualString) (IntValue);;
```

```
module MakeMap (Equal : EqualSig) (Value : ValueSig)
  : MapSig
    with type key = Equal.t
    with type value = Value.value
= struct
  type key = Equal.t
  type value = Value.value
  module EqualKey = struct
    type t = key * value
    let equal (key1, _) (key2, _) =
      Equal.equal key1 key2
  end;;
  module Set = MakeSet (EqualKey);;
  type t = Set.t
  let empty = Set.empty
  let add map key value = Set.add (key, value) map
  let find map key = snd (Set.find map key)
end;;
```

12.3 Higher-order functors

A *higher-order* functor is a functor that takes a functor as an argument. While higher-order functors are rarely used in practice, there are times when they can be useful.

For example, in relation to our running example, the `MakeMap` functor is tied to a specific definition of the `MakeSet` functor. If we have multiple ways to build sets (for example, as lists, trees, or some other data structure), we may want to be able to use any of these sets when building a map. The solution is to pass the `MakeSet` functor as a parameter to `MakeMap`.

The type of a functor is specified using the **functor** keyword, where $signature_2$ is allowed to depend on the argument `Arg`.

functor (Arg : $signature_1$) `->` $signature_2$

When passing the `MakeSet` functor to `MakeMap`, we need to specify the functor type with its sharing constraint. The `MakeMap` definition changes as follows; the structure definition itself doesn't change.

module MakeMap (Compare : CompareSig) (Value : ValueSig)
  (MakeSet : **functor** (CompareElt : CompareSig) `->`
    SetSig **with type** elt = CompareElt.elt)
  : MapSig
    with type key = Compare.elt
    with type value = Value.value
= **struct** $\cdots$ **end**

These types can get complicated! Certainly, it can get even more complicated with the ability to specify a functor argument that itself takes a functor. However, as we mentioned, higher-order functors are used fairly infrequently in practice, partly because they can be hard to understand. In general, it is wise to avoid gratuitious use of higher-order functors.

12.4 TODO

- Recursive modules
- Module sharing constraints

Chapter 13

The OCaml Object System

OCaml includes a unique object system with classes, parameterized classes, and objects, and the usual features of inheritance and subclassing. Objects and classes provide a mechanism for extensibility and code re-use, while preserving all the features we have come to expect from OCaml, including strong typing, type inference, and first-class functions.

13.1 Simple classes

Let's begin by defining class that implements a pseudo-random number generator. One of the simplest of these computes a *linear congruential sequence* of numbers $\langle x_n \rangle$ obtained from the following formula.

$$x_{n+1} = (ax_n + c) \textbf{ mod } m$$

There are four special numbers:

m	the modulus	$0 < m$,
a	the multiplier	$0 \leq a < m$,
c	the increment	$0 \leq c < m$,
x_0	the starting value, or seed	$0 \leq x_0 < m$.

For the moment, let's choose the values $a = 314159262$, $c = 1$, $X_0 = 1$, and $m = 2^{30}$. The following program defines a class that provides a method `next_int` to compute the next integer in the sequence.

```
class linear_congruential_rng1 =
object
    val mutable x = 1
    method next_int =
        x <- (x * 314159262 + 1) land 0x3fffffff;
        x
end;;
```

In OCaml, a *class* defines an object, which has a collection of values (defined with the keyword `val`), and methods (defined with `method`). In this example, the value x represents a number on the random sequence. The method `next_int` computes the next number of the sequence, setting x to the new value, and returns the result. For efficiency, and numerical reasons, instead of computing the result modulo 2^{30}, the result is masked with the integer `0x3fffffff` ($2^{30} - 1$).

Before the generator can be used, it must be *instantiated* using the `new` operation.

```
# let rng = new linear_congruential_rng1;;
val rng : linear_congruential_rng1 = <obj>
# rng#next_int;;
- : int = 314159263
# rng#next_int;;
- : int = 149901859
# rng#next_int;;
- : int = 494387611
```

The `new` operation builds an *object* from the class. Methods in the object are invoked with the operator `#` and the method name.

13.1.1 Objects vs. classes

In OCaml, objects and classes are not the same. A class defines a template for constructing an object, but it is not an object itself. In addition, every class has a name, while objects can be defined and used anonymously.

```
# (object method next_int = 31 end)#next_int;;
- : int = 31
```

For the moment, the existence of a name has little significance. However, as we will see in the next chapter, the name is required for defining inheritance. That is, it is possible to inherit from classes, but not objects. For this reason, we will usually be defining classes, rather than anonymous objects.

13.2 Parameterized classes

The class `linear_congruential_rng1` is somewhat limited, because the parameters for the random sequence are hard-coded. It is also possible to parameterize a class. The syntax is much the same as for defining a function; the parameters are listed after the class name.

```
# class linear_congruential_rng a c seed =
object
    val mutable x = seed
    method next_int =
        x <- (x * a + c) land 0x3fffffff;
        x
end;;
class linear_congruential_rng :
int -> int -> int ->
   object val mutable x : int method next_int : int end
```

A parameterized class is essentially a function that computes a class. For example, we can obtain a class that is equivalent to the original generator by applying the parameterized class to the original arguments.

```
# class linear_congruential_rng1 = linear_congruential_rng 314159262 1 1;;
class linear_congruential_rng1 : linear_congruential_rng
# let rng = new linear_congruential_rng1;;
val rng : linear_congruential_rng1 = <obj>
# rng#next_int;;
- : int = 314159263
# rng#next_int;;
- : int = 149901859
```

When given a parameterized class, the `new` operator returns a function that computes an object given arguments for the parameters.

```
# new linear_congruential_rng;;
- : int -> int -> int -> linear_congruential_rng = <fun>
# let rng = new linear_congruential_rng 31415926 1 1;;
val rng : linear_congruential_rng = <obj>
# rng#next_int;;
- : int = 31415927
# rng#next_int;;
- : int = 575552731
```

The function produced by `new` is the same as any other function; it is a value that can be passed as an argument, stored in a data structure, or partially applied. For example, the `linear_congruential_rng` takes three arguments, a, c, and the initial seed. If we want a particular generator with fixed values for a and c, and only allow the seed to vary, we can perform a partial application.

```
# let rng_from_seed = new linear_congruential_rng 314159262 1;;
val rng_from_seed : int -> linear_congruential_rng = <fun>
# let rng = rng_from_seed 17355;;
val rng : linear_congruential_rng = <obj>
# rng#next_int;;
- : int = 846751563
```

```
# rng#next_int;;
- : int = 411455563
```

13.3 Self references and private methods

So far, we have been dealing with objects that have one method. It is possible, of course, to define objects with more than one method. For example, in addition to generating integers, we might also want to generate floating-point numbers uniformly distributed between 0 and 1. It seems easy enough—we can define a new method `next_float` that computes the next random number, and divides it by the modulus m.

```
# class linear_congruential_rng a c seed =
  object
    val mutable x = seed
    method next_int =
        x <- (x * a + c) land 0x3fffffff;
        x
    method next_float =
        x <- (x * a + c) land 0x3fffffff;
        (float_of_int x) /. (float_of_int 0x3fffffff)
  end;;
class linear_congruential_rng : ...
# let rng = new linear_congruential_rng 314159262 1 1;;
val rng : linear_congruential_rng = <obj>
# rng#next_float;;
- : float = 0.292583613928950936
# rng#next_float;;
- : float = 0.139606985393545574
```

This is suboptimal of course. We see that the `next_int` and `next_float` methods are duplicating the code for generating random numbers. What we should do is move the shared code into a shared

method, called `next`, that computes the next number in the sequence.

To do so, we will need to give the object a name, so that the `next` method can be called from the `next_int` and `next_float` methods. Syntactically, this is performed by specifying the object name in parentheses after the **object** keyword (the name can be an arbitrary lowercase identifier, but the usual choice is **self**). Let's rewrite the new generator.

```
class linear_congruential_rng a c seed =
object (self)
    val mutable x = seed
    method next =
        x <- (x * a + c) land 0x3fffffff
    method next_int =
        self#next;
        x
    method next_float =
        self#next;
        (float_of_int x) /. (float_of_int 0x3fffffff)
end;;
```

As a final step, the shared method `next` is really a "private" method, used to implement `next_int` and `next_float`. It is unlikely that we intend it to be called directly. Methods of this kind can be marked with the keyword **private** after the **method** keyword, to make them inaccessible outside the object.

```
# class linear_congruential_rng a c seed =
  object (self)
      val mutable x = seed
      method private next =
          x <- (x * a + c) land 0x3fffffff
      method next_int = self#next; x
      method next_float = self#next; ...
  end;;
```

```
class linear_congruential_rng : ...
# rng#next_float;;
- : float = 0.292583613928950936
# rng#next;;
This expression has type linear_congruential_rng
It has no method next
```

13.4 Class initializers

Unlike many other object-oriented languages, OCaml does not provide explicit constructors. With parameterized classes, there is less of a need, since the initial object can be often computed from the parameters. However, there are times when it is useful or necessary to perform a computation at object creation time.

There are two ways to specify initializers, as **let**-definitions that are evaluated before the object is created, or as anonymous **initializer** methods that are evaluated after the object is created.

13.4.1 Let-initializers

Let-initializers are defined as the initial part of a class definition. Using our example, suppose we wish to define a random number generator that produces either 1) a canonical sequence starting from a standard seed, or 2) a sequence with a random initial seed. Our new generator will take a Boolean argument, and use a let-definition to choose between the cases. For the latter case, we'll use the current time of day as the seed. [1]

```
# class new_rng randomize_seed =
  let a, c, m = 314159262, 1, 0x3fffffff in
  let seed =
      if randomize_seed
```

[1] Note that this example uses the `Unix.gettimeofday` function. To run the example in the toploop, you need to pass the `Unix` library using the command `ocaml unix.cma`.

```
      then int_of_float (Unix.gettimeofday ())
      else 1
  in
  let normalize x = (float_of_int x) /. (float_of_int m) in
  object (self)
      val mutable x = seed
      method private next =
          x <- (x * a + c) land m
      method next_int =
          self#next;
          x
      method next_float =
          self#next;
          normalize x
  end;;
class new_rng : ...
# let rng = new new_rng true;;
val rng : new_rng = <obj>
# rng#next_int;;
- : int = 1025032669
```

Notice that we are also defining the initial parameters a, c, and m symbolically, as well as a normalization function for producing the floating-point results.

13.4.2 Anonymous initializer methods

Let-initializers are evaluated before an object is created. Sometimes it is also useful to evaluate an initializer after the object is created. For example, supposed we wish to skip an initial prefix of the random number sequence, and we are given the length of the initial prefix. While we could potentially pre-compute the initial values for the generator, it is much easier to construct the generator without skipping, and then remove the initial prefix before returning the object.

```
class skip_rng skip =
  let a, c, m, seed = 314159262, 1, 0x3fffffff, 1 in
  object (self)
      val mutable x = seed
      method private next =
          x <- (x * a + c) land m
      method next_int = self#next; x
      initializer
          for i = 1 to skip do
              self#next
          done;
          Printf.printf "rng state: %d\n" x
  end;;
class skip_rng : ...
# let rng = new skip_rng 10;;
rng state: 888242763
val rng : skip_rng = <obj>
# rng#next_int;;
- : int = 617937483
# let rng11 = new skip_rng 11;;
rng state: 617937483
val rng11 : skip_rng = <obj>
```

13.5 Polymorphism

Classes and objects may also include polymorphic values and methods. As we have seen in the examples so far, the types of methods and values are automatically inferred. Very little changes when polymorphism is introduced, but it will be necessary to introduce a small number of annotations.

One common application of random number generators is to choose from a finite set of values. That is, instead of returning a number, the generator should return a value, chosen randomly, from

a prespecified set. The type of elements of the set is unimportant to the choice of element, of course, so the generator should be polymorphic.

As an initial attempt, we can define a generator that takes an array of elements as a parameter. The `choose` method will then select from this set of elements.

```
# class choose_rng elements =
    let a, c, m, seed = 314159262, 1, 0x3fffffff, 1 in
    let length = Array.length elements in
    object (self)
        val mutable x = seed
        method private next =
            x <- (x * a + c) land m
        method choose =
            self#next;
            elements.(x mod length)
    end;;
Some type variables are unbound in this type:
  class choose_rng : 'a array -> ...
```

Unfortunately, this definition is rejected by the compiler because "Some type variables are unbound." There are two rules to follow when defining a polymorphic object.

1. All type parameters must be listed between square brackets after the **class** keyword (for example, as `['a]`).

2. Explicit types must be specified for methods that return values of polymorphic type.

In our example, the `elements` array is polymorphic, and the `choose` method returns a value of polymorphic type, so the example can be fixed as follows.

```
class ['a] choose_rng elements =
  let a, c, m, seed = 314159262, 1, 0x3fffffff, 1 in
  let length = Array.length elements in
  object (self)
```

```
      val mutable x = seed
      method private next =
          x <- (x * a + c) land m
      method choose : 'a =
          self#next;
          elements.(x mod length)
  end;;
class ['a] choose_rng : ...
# let rng = new choose_rng [|"Red"; "Green"; "Blue"|];;
val rng : string choose_rng = <obj>
# rng#choose;;
- : string = "Red"
# rng#choose;;
- : string = "Green"
# rng#choose;;
- : string = "Blue"
# rng#choose;;
- : string = "Green"
# let rng = new choose_rng [|1.1; 2.2; 3.14; 4.4; 5.5|];;
val rng : float choose_rng = <obj>
# rng#choose;;
- : float = 5.5
# rng#choose;;
- : float = 3.14
```

13.5.1 Polymorphic methods

A small complication arises for methods where the arguments are polymorphic. For example, instead of defining the set of elements are a class parameter, suppose we pass the element array as an argument to the `choose` method. Following the rules given in the previous section, we will have

to specify a type for the `choose` method.

```
class ['a] choose_rng =
  let a, c, m, seed = 314159262, 1, 0x3fffffff, 17 in
  object (self)
      val mutable x = seed
      method private next =
          x <- (x * a + c) land m
      method choose (elements : 'a array) : 'a =
          self#next;
          elements.(x mod Array.length elements)
  end;;
# let rng = new choose_rng;;
val rng : '_a choose_rng = <obj>
# rng#choose [|1; 2; 3|];;
- : int = 1
# rng#choose [|1; 2; 3|];;
- : int = 2
# rng;;
- : int choose_rng = <obj>
# rng#choose [|"Red"; "Green"; "Blue"|];;
This expression has type string array but is here used with type int array
```

Unfortunately, the object is not polymorphic in the way that we want. The type `'_a choose_rng` specifies that the generator can be used with some type `'_a` of elements. When we use the `rng` with an array of integers, the type becomes `int choose_rng`, and any attempt to use it with any other type (such as an array of strings) results in a type error.

The problem here is that it isn't the *object* that should be polymorphic, is the the *method*. In other words, the `choose` method should be polymorphic, having type `'a array-> 'a` for any type `'a`, but the object itself is not polymorphic. OCaml provides a way to specify this directly, using explicit type quanitification. The method `choose` gets the type `'a . 'a array -> 'a`, where the `'a .` prefix specifies that polymorphism is restricted to the `choose` method, as presented in the

following example.

```
class choose_rng =
  let a, c, m, seed = 314159262, 1, 0x3fffffff, 17 in
  object (self)
      val mutable x = seed
      method private next =
          x <- (x * a + c) land m
      method choose : 'a. 'a array -> 'a = fun elements ->
          self#next;
          elements.(x mod Array.length elements)
  end;;
class choose_rng : ...
# let rng = new choose_rng;;
val rng : choose_rng = <obj>
# rng#choose [|1; 2; 3|];;
- : int = 1
# rng#choose [|"Red"; "Green"; "Blue"|];;
- : string = "Green"
```

Chapter 14

Inheritance

`JYH: this is currently a very rough draft.`

Inheritance, in a general sense, is the the ability for one part of a program to re-use code in another part of the program by specifying the code to be re-used as well as any modifications that are needed. In the context of object-oriented languages, inheritance usually means the ability for one class to acquire methods and other attributes from another class—in other words the first class inherits from the second—simply by referring to the inherited class. Normally inheritance will be transitive; if C inherits from B and B inherits from A, then C also inherits (indirectly) from A.

Object-oriented programming languages that use static typing (not all do) need also to describe the typing rules for objects that may be influenced by the inheritance relationships in the program. Normally, this takes the form of a *subtyping* relationship, written $B <: A$, which specifies that a value of type B may be used anywhere where a value of type A is expected. In OCaml, as in many other object-oriented languages, inheritance and subtyping are the same. That is, if class B inherits from class A, then $B <: A$, and an object of class B may be used anywhere where an object of class A is expected.

Furthermore, the dual role of classes as definitions for objects and classes as (or producing) types for object expressions has caused some object-oriented languages to distinguish *implementation inheritance* and *interface inheritance*. *Implementation inheritance* refers to inheriting of attribute

definitions: instance variables, methods, and sometimes other structural elements. *Interface inheritance* refers to inheriting of attribute *specifications*: types for methods (and sometimes instance variables) and a requirement that definitions for the specified elements be present.

The OCaml object system provides extensive support for inheritance, including both implementation inheritance and interface inheritance, and explicit control for cases where methods have parameters that might be affected by inheritance. To ensure that programs be type-safe, the object system includes type-safe constructions for doing type conversion up and down the inheritance hierarchy.

In this chapter we will cover the language constructs in OCaml that support inheritance, and show code examples for standard patterns that normally arise in programs that make use of inheritance—abstract classes and methods, access to "super," sending messages up and down the inheritance hierarchy. The latter part of the chapter will cover these same items again for *multiple inheritance*, where classes inherit from more than one "parent" class.

14.1 Simple inheritance

Let's return to the example of random number generators, introduced in the previous chapter. All the examples in that chapter used the linear congruential method for computing pseudo-random sequences. The linear method isn't the only method for generating pseudo-random sequences, of course. Suppose we wish to use a new quadratic method, say $x_{n+1} = x_n(x_n + 1)$ **land** m, to build a new class `quadratic_rng`. Only one method (the `next` method) needs to be redefined, as shown in Figure 14.1.

The class `quadratic_rng` inherits from the class `linear_rng`, which means that it gets all the methods and instance variables from `linear_rng`. In the figure, the `quadratic_rng` also redefines the `next` method to use a quadratic formula. The new definition *overrides* the previous definition; when the `self#next` method is invoked, it now refers to the quadratic computation, not the linear.

```
# let rng = new quadratic_rng;;
val rng : quadratic_rng = <obj>
# rng#next_int;;
```

```
class linear_rng =
  object (self)
      val a = 314159262
      val c = 1
      val m = 0x3fffffff
      val mutable x = 2
      method private next =
          x <- (x * a + c) land m
      method next_int =
          self#next;
          x
      method next_float =
          self#next;
          float_of_int x /. float_of_int m
  end;;
```

```
class quadratic_rng =
   object
       inherit linear_rng
       method private next =
           x <- (x * (x + 1)) land m
   end;;

class quadratic_rng :
  object
    val mutable x : int
    val m : int
    val c : int
    val a : int
    method private next : unit
    method next_float : float
    method next_int : int
  end
```

```
- : int = 6
# rng#next_int;;
- : int = 42
```

14.1.1 Type equality

Now that we have defined a quadratic generator, we would expect that it can be used in all the same places that a linear generator can be used—after all, that two classes have the same methods with the same types. For example, let's redefine a `choose` function that selects an element from an array. Here we specify explicitly that the `choose` function should take a `linear_rng` as an argument.

```
# let choose (rng : linear_rng) elements () =
      elements.(rng#next_int mod Array.length elements);;
val choose : linear_rng -> 'a array -> unit -> 'a = <fun>
# let g = choose (new quadratic_rng) [|"Red"; "Green"; "Blue"|];;
val g : unit -> int = <fun>
# g ();;
- : string = "Red"
...
# g ();;
- : string = "Green"
# g ();;
- : string = "Blue"
```

In this case, the reason why the `quadratic_rng` is accepted as a `linear_rng` is because the generator classes have types that are exactly equal—they have the same methods, and each method has the same type.

14.1.2 Subtyping

In general, of course, the class type may change during inheritance. Suppose, for example, that we decide to give the quadratic generator an extra method.

```
# class quadratic_rng =
   object
       inherit linear_rng
       method private next =
           x <- (x * (x + 1)) land m
       method print =
           print_string ("x = " ^ string_of_int x)
   end;;

# let choose (rng : linear_rng) elements () =
    elements.(rng#next_int mod Array.length elements);;

# let g = choose (new quadratic_rng) [|"Red"; "Green"; "Blue"|];;
This expression has type quadratic_rng but
    is here used with type linear_rng
Only the first object type has a method print
```

Here, the class types are no longer the same, because the class `quadratic_rng` has an extra method. The OCaml compiler rejects use of a quadratic generator because of a type-mismatch. In fact, the error message mentions the name of the extra method.

OCaml takes a strict approach to subtyping. The type `quadratic_rng` is a subtype of `linear_rng`, but coercions must be explicit. That is, we must explicitly *coerce* the `quadratic_rng` to a `linear_rng` using the `:>` operator, as follows.

```
# let g = choose (new quadratic_rng :> linear_rng) [|"Red"; "Green"; "Blue"|];;
val g : unit -> string = <fun>
# g ();;
- : string = "Red"
```

The `:>` operator *casts* its argument, which must have an object type, to a supertype. In cases

where the argument type can't be inferred, a ternary form may be used. For example, the following function defines a cast from `quadratic_rng` to `linear_rng`.

```
# let linear_of_quadratic_rng rng =
     (rng : quadratic_rng :> linear_rng);;
val linear_of_quadratic_rng : quadratic_rng -> linear_rng = <fun>
```

14.2 Abstract classes

```
Outline for the rest of single-inheritance.
1. Abstract classes:
   a. Define an abstract superclass rng.
2. Variance annotations.
3. Interface inheritance
4. Lack of downcasting.
```

Bibliography

[1] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[2] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[3] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2002. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

[4] Chris Okasaki. Red-black trees un a functional setting. *Journal of Functional Programming*, 9(4):471–477, May 1999.

[5] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.