

# Chapter 1

## Introduction

This document is an introduction to ML programming, specifically for the OCaml implementation. In CS134a, we used C (and variations) to specify processes, monitors, semaphores, device drivers and more. The topic of CS134a was operating systems, and the C language provides an appropriate machine model for programming the machine.

For the last part of CS134a, we covered a few of the newer operating systems, like Mach, Amoeba, Spin, Vino, etc. One running theme of these systems is that the operating system *kernel* should be reduced to minimum functionality; additional functionality is defined through user modules, or by inserting code into the kernel to be run in kernel space.

One result of this is that more work is being shifted into the compiler. For example, the Spin kernel requires that extensions be written in Modula-3 so that *safety* is guaranteed by the compiler. The Vino system retains the unsafe C compiler, but uses software fault isolation to guarantee the safety of the inserted code. In the Amoeba system, the compiler participates in the distribution and management of resources.

A shift like this is a natural part of any successful process in computer science. In general, the initial effort are focused on getting a *working* system. When operating systems were first being developed, the goal was to provide a safe, generic interface. the emphasis was on simplicity. C was developed as a language to make programming the machine easier, and the control that C allows over the representation and implementation of programs provided tight programmer control over the efficiency of programs and utilities. Another goal of these systems was to provide *portability* for a limited class of machines. We now have *working*, generic, systems.

Later, as a process evolves, it is faced with the standard problems of growth, technically termed *scalability*. As operating systems have evolved, more functionality is added to the kernel, and kernel sizes continue to grow. Scalability presents three kinds of problems.

1. Problems with managing a growing number of computing resources. This includes growth in the number of different kinds of machines and hardware devices, as well as collecting *distributed* resources into a manageable whole.
2. Problems with *designing* a large system. As a system grows, it must be divided into manageable pieces that distinct design teams can tackle.
3. Problems *maintaining* an ever-growing system. Monolithic systems increase the amount of effort in maintaining the system, because, in the worst case, *every* part of the system must evolve together with the entire system.

Traditional systems, like Linux and Windows have addressed part of the problem through methods like dynamically linking new functionality (modules) into a running kernel. The development of modules can be performed separately from kernel development. This technique is workable if the kernel extensions conform to one of the predefined interfaces, and if the set of kernel extensions are trusted. No support is provided for allowing an arbitrary user to add a new filesystem, for instance. In addition, many users needing performance are frustrated at being required to use a general-purpose, low-performance system API.

The design language traditional operating systems has evolved as well. Libraries for threads, process synchronization, and message passing have been implemented to augment the C/C++ programming languages with more modern features. The C design model has been stressed by this process. Some examples: threads have been grafted onto its serial programming model; pointers are not valid on multiple machines in a distributed system, and programmers have to be careful to copy data when appropriate; message passing primitives tend to be scattered through the code, making it difficult to change communication protocols.

Perhaps the biggest strain has been scaling C programs to extremely large sizes. The language and compiler provide only limited support for enforcing modularity, and in general a new component may interfere with the parts of an existing system (there are no guarantees about *safety*). The machine model provided by C is not always convenient for expressing high-level data structures, like filesystems or communication protocols. C programs tend to be tied closely with the machine architecture, making it difficult to adapt an existing program to a new model (say, converting a program that uses shared-memory to a program using message-passing).

The design problem is hard. No programming language seems appropriate for *all* programs; if it were, the language and compiler would themselves would have problems with scalability. At present, there is a pressing need for languages that address specific issues. The following are just a few examples.

- Languages for *graphics and simulation*, where the properties of simulation *models* can be separated from numerical algorithms and display

representations.

- Languages for large (thousands to millions of processors) systems, including distributed databases and compute servers. Any distributed system of this magnitude must be tolerant to massive failures in parts of the system.
- Languages for real-time systems like automotive, aircraft, and space applications.
- Languages for adding functionality *safely* to existing systems.

As researchers and developers in computer science, we strive to find simple elegant solutions to difficult problems. One route that is sometimes, but not always, appropriate is to adapt the programming language to the task at hand. The language is the basis for designing, understanding, and maintaining a system. The function of the compiler is to help us in the design process by providing useful functionality, catching errors, and (of course) translating a source program into a program for a specific machine architecture.

In implementing a compiler, we have to answer two questions. what is that language we are compiling? When language do we use to write the compiler? This two are not necessarily the same (although they often are). The task of compiling is a process of transformation, and many representations of the program are used in the process. The compiler should be implemented in a language where its data structures and algorithms have the most natural structure.

We'll be using OCaml. OCaml is a dialect of the ML ("Meta-Language") languages, which were designed for the LCF ("Logic of Computable Functions") theorem prover. ML was designed as a tool for reasoning about programs and languages. In particular, it is an ideal language for implementing a compiler.

## 1.1 Functional and imperative languages

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, here is how Euclid's algorithm would normally be written in an imperative language like C.

```
/*  
 * Determine the greatest common divisor of two positive  
 * numbers a and b. We assume a>b.  
 */  
int gcd(int a, int b)
```

```
{
  int r;

  while(r = a % b) {
    a = b;
    b = r;
  }
  return b;
}
```

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*.

```
let rec gcd a b =
  let r = a mod b in
  if r = 0 then
    b
  else
    gcd b r
```

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. Imperative code may be used where it is natural, but imperative code *may not* be used if there is a more elegant functional version.

## 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 you have programming experience in an imperative language like C; I'll point out the differences between ML and C in the cases that seem appropriate.





## Chapter 2

# Simple Expressions

Many functional programming implementations includes a significant runtime component including libraries and a garbage collector. They often include a “toploop” which can be used to interact with the system. OCaml provides a compiler and a toplevel. By default, the toplevel is called `ocaml`. The toplevel prints a prompt (`#`), read an input expression, and evaluates it. Expressions in the toplevel must be terminated by a double-semicolon `;;`. My machine name is `kenai`.

```
<kenai 113>ocaml
      Objective Caml version 2.04

# 1 + 4;;
- : int = 5
#
```

The toplevel prints the *type* of the result (in this case, `int`), and the value (5). To exit the toplevel, you may type `^D` (the end-of-file character).

## 2.1 Basic expressions

OCaml is a *strongly typed* language: every expression must have a type, and expressions of one type may not be used as expressions in another type. There are no implicit coercions.

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

### 2.1.1 `unit`: the “unit” type

The simplest type in OCaml is the `unit` type, which contains one element: `()`. This seems to be a rather silly type. It corresponds to the `void` type in C:

the element `()` is commonly used as the value of a procedure that computes by side-effect.

### 2.1.2 `int`: the “integers”

The `int` type is the type of numbers:  $\dots, -2, -1, 0, 1, 2, \dots$ . There are the usual expressions `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `mod` (remainder).

In addition there are the normal shifting and masking operators on the binary representations of the numbers.

- `i lsl j`: logical shift left  $i * 2^j$ .
- `i lsr j`: logical shift right  $i/2^j$  ( $i$  is treated as an unsigned two's-complement number).
- `i asl j`: arithmetic shift left  $i * 2^j$ .
- `i asr j`: arithmetic shift right  $i/2^j$  (the sign of  $i$  is preserved).
- `i land j`: bitwise-and.
- `i lor j`: bitwise-or.

### 2.1.3 `float`: the floating-point numbers

The floating-point numbers provide fractional numbers. The syntax of a floating point includes either a decimal point, or an exponent (base 10) denoted by an `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 `'`: `+. , -. , *. ,` and `/.` There are also coercion functions `int_of_float` and `float_of_int`.

```
<kenai 114>!!
```

```
ocaml
```

```
Objective Caml version 2.04
```

```
# 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
```

### 2.1.4 char: the characters

The character type is currently implemented as characters from the ASCII character set. The syntax for a character uses single quotes.

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

The numerical specification is in *decimal*, so `'\120'` is the ASCII character 'x', not 'P'.

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. (We'll get back to the naming of these functions in the next chapter).

The `lowercase` and `uppercase` functions give the equivalent lower or uppercase characters.

### 2.1.5 string: character strings

Character strings are a built-in type. Unlike strings in C, character strings do not use `'\000'` as a termination character. The `String.length` function computes the length of the string. The syntax for strings uses double-quotes. Characters in the string may use the `\ddd` syntax.

```
"Hello", " world\000 is not a terminator\n", ""
```

The `^` operator performs string concatenation.

```
# "Hello " ^ "world\000Not a terminator\n";;
- : string = "Hello world\000Not a terminator\n"
# Printf.printf "%s" ("Hello " ^ "world\000Not a terminator\n");;
Hello worldNot a terminator
- : unit = ()
```

Strings also allow random access. The `s.[i]` operator gets character *i* from string *s*, and the command `s.[i] <- c` replaces character *i* in string *s* by character *c*.

### 2.1.6 bool: the Boolean values

There are only two Boolean values: `true` and `false`. Every *relation* returns a Boolean value. Logical negation is performed by the `not` function. The standard binary relations take two values of equal types and compare them in the normal way.

- $x = y$ : equality
- $x <> y$ : *x* is not equal to *y* (equivalent to `not (x = y)`).

- $x < y$ :  $x$  is less than  $y$
- $x \leq y$ :  $x$  is no more than  $y$
- $x \geq y$ :  $x$  is no less than  $y$
- $x > y$ :  $x$  is greater than  $y$

These relations operate on values of *arbitrary* type. For the base types in this chapter, the comparison is what you would expect. For values of other types, the value is implementation-dependent.

The logical operators are also defined: `&&` is conjunction, and `||` is disjunction. Both operators are the “short-circuit” versions: the second clause is not evaluated if the result can be determined from the first clause. For example, in the following expression, the clause `3 > 4` is not evaluated (which makes no difference at this point, but will make a difference when we add side-effects).

```
# 1 < 2 || 3 > 4;;
- : bool = true
```

There is also a conditional operator `if  $b$  then  $e_1$  else  $e_2$` .

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

## 2.2 Compiling your code

OCaml also includes two compilers. If you wish to compile your code, you should place it in a file with the `.ml` suffix. There are two compilers: `ocamlc` compiles to byte-code, and `ocamlopt` compiles to native machine code. The native code is roughly three times faster, but compile time is a lot longer. The usage is a lot like `cc`. The double-semicolon terminators are not necessary in the 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`.
- 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 to program files `x.cmo` and `y.cmo`, the command would be:

```
<kenai 165>ocamlc -g -o program x.cmo y.cmo
<kenai 166>./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.

## 2.3 The OCaml type system

The ML languages are *strictly* typed. In addition, every expression has a exactly one type. In contrast, C is a weakly-typed language: values of one type can be coerced to a value of any other type, whether the coercion makes sense or not. Lisp is an “untyped” 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 a valid program will never fault because of an invalid machine operation. All memory accesses will be valid. ML guarantees safety by guaranteeing that every correctly-typed value is valid value, and Lisp guarantees it by checking for validity at run time. One surprising (some would say “annoying”) consequence is that ML has no “nil” values; again, all correctly type values are valid.

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 which programs are bogus. If you make a change, the type checker will help track down all 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 `chapter_exceptions`),
  - (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>pr -n -t 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, character 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 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 issue: the `else` branch is not required in a conditional. If it is omitted, the conditional is treated as if `else` case returns the `()` value. The following code has a type error.

```
<kenai 236>pr -n -t y.ml
  1 if 1 < 2 then
  2   1
<kenai 237>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.

## 2.4 Comment convention

In OCaml, comments are enclosed in matching `(*` and `*)` pairs. Comments may be nested, and the comment content is treated as white space.

## Chapter 3

# Variables and Functions

So far, we have only considered simple expressions, not involving variables. In ML, variables are *names* for values. In a purely functional setting, it is not possible to tell the difference between a variable from the value it stands for.

In OCaml, variable bindings are introduced with the `let` keyword. The syntax of a simple top-level declaration 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 arbitrarily using the `in` form.

```
# 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): the value of a variable is defined by the innermost `let` definition for the variable. The variable is bound only in the body of the `let`; or, for toplevel definitions, the rest of the file.

```
# let x = 1 in
  let x = 2 in
    let y = x + x in
      x + y;;
- : int = 6
```

What is the value of `z` in the following definition?

```
# let x = 1
  let z =
    let x = 2 in
      let x = x + x in
        x + x
```

### 3.1 Functions

Functions are defined with the `fun` keyword. The `fun` is followed by a sequence of variables that name the arguments, and `->` separator, and the the body of the function. By default, functions are not *named*. In ML, functions are values like any other. They may be constructed, passed as arguments, and applied to specific arguments. Like any other value, they may be named by using a `let`.

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

The `->` is a *function type*. The type before the arrow is the type of the function's argument, and the value after the arrow is the type of the value. The `incr` function takes an integer argument, and returns an integer.

The syntax for function application (function call) is concatenation: the function is followed by its arguments.

```
# incr 2;;
- : int = 3
```

Functions may also be defined with multiple arguments. For example, 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* of the `sum`: `int -> int -> int`. The arrow associates to the right, so this could also be written `int -> (int -> int)`. That is, the `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. The application of a function to only some of its arguments 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 next to the function name, before the equality.

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

### 3.1.1 Scoping and nested functions

In ML, functions may be arbitrarily nested. They may also be defined and 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. All arguments, and `let`-bound variables are visible in nested functions. For example, another way to define the summation 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 sum i j =
    let apply f k =
        f k
    in
    let i = 5 in
    let sum2 l =
```

```

        i + 1
    in
        apply sum2 j;;
val sum : 'a -> int -> int = <fun>

```

You can ignore the 'a type for the moment. What is the value of this function? The `apply` function takes a function `f` and an argument `j`, and it applies `f` to `j`. In the next step, `i` is bound to 5. In `sum2`, is the value of `i` 5, or is it determined by the formal parameter `i`. Trying this out we find the following.

```

# sum 3 10;;
- : int = 15

```

Apparently, the value of `i` is 5, not 3 as we passed in the argument. Static (lexical) scoping determines this: the value of `i` in `sum2` must be 5, because the closest definition of `i` is 5—even though `sum2` is evaluated within the `apply` function where the value of `i` is determined by the argument `i`.

### 3.1.2 Recursive functions

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

```

# let rec power i x =
    if i = 0 then
        1
    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 very different (note the use of the type variables 'a and 'b, which we describe in the section on polymorphic types).

```

# let power i x =
    x;;
val power : 'a -> 'b -> 'b = <fun>
# let power i x =
    if i = 0 then
        1
    else

```

```

    x *. (power (i - 1) x);;
val power : int -> float -> float = <fun>
# power 5 2.0;;
- : float = 4

```

Mutually recursive definitions (functions that call one another) can be defined use 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 another definition where a function is passed as an argument, and another function is returned. 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 +. f (x +. dx)) /. 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 this to the power function defined above.

```

# 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
# 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

```

As we would expect, the derivative of  $x^3$  is approximately  $3x^2$ . 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 `'` character 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.

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

```

# 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 number 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
```



## Chapter 4

# Basic Pattern Matching

One of the more powerful features of ML is that it uses *pattern matching* to define functions by case analysis. Pattern matching is performed by the **match** expression, which has the following general form.

```
match expr with  
  patt1 -> expr1  
  | patt2 -> expr2  
  ...  
  | pattn -> exprn
```

A *pattern* is an expression made of constants and variables. When the pattern matches the argument, the variables are given the values that match.

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

```
# let rec fib i =  
  match i with  
    0 -> 1  
  | 1 -> 1  
  | j -> fib (j - 2) + fib (j - 1);;  
val fib : int -> int = <fun>  
# fib 1;;  
- : int = 1  
# fib 2;;  
- : int = 2  
# fib 3;;  
- : int = 3  
# fib 6;;  
- : int = 13
```

In this code, the argument  $i$  is compared against the constants 1 and 2. If either of these cases match, the return value is 1. 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  $\text{fib } (j - 2) + \text{fib } (j - 1)$  computes the returned value.

This form of matching, where the function body is a **match** expression applied to the function argument, is quite common in ML programs. OCaml defines an equivalent syntactic form to handle this case, using the **function** keyword (instead of **fun**). A **function** definition is just like a **fun**, except that multiple patterns are allowed. The `fib` definition using **function** is as follows.

```
# let rec fib = function
    0 -> 1
  | 1 -> 1
  | i -> fib (i - 1) + fib (i - 2);;
val fib : int -> int = <fun>
# fib 1;;
- : int = 1
# fib 6;;
- : int = 13
```

Patterns can also be used the other basic types, like characters, strings, and Boolean values. In addition, multiple patterns *without variables* can be used for a single body. For example, one way to check for capital letters is 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 pattern that matches anything. It is not a variable (so it can’t be used in an expression). By convention, the `is_uppercase` function would more appropriately be written using the wildcard pattern.

```

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

```

## 4.1 Incomplete matches

You might wonder about what happens if all the cases are not considered. 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`) is verbose about inexhaustive patterns. It warns that the pattern is non-exhaustive, and it even suggests 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 7). 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 identifies 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")
```

## 4.2 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 patt = expr
fun patt -> expr
```

These aren't much use with constants because the pattern match will always be non-exhaustive (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

```
include Base_theory
```

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 and functions. 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 implementation we need *aggregate types*.

OCaml provides a rich set of aggregate types, including tuples, lists, disjoint union (also called tagged unions, or variant records), records, and arrays. In this chapter, we'll look at the simplest part: tuples and lists. We'll discuss unions in Chapter 6, and we'll leave the remaining types for Chapter 8, when we introduce side-effects.

### 5.1 Polymorphism

At this point, it is also appropriate to introduce *polymorphism*. The ML languages provide *parametric polymorphism*. That is, types may be parameterized by type variables. For example, the identity function in ML can be expressed 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 `int`, then it returns an `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* type, and returns a value of the same type. This is a form of delayed polymorphism. When we apply the `identity'` function to a number, the type `'_a` is assigned to be `int`; the `identity'` function can no longer be applied 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 expressions that evaluate to themselves. For example, all numbers, characters, and string constants 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`).

the normal way to get around the value restriction is to use “eta-expansion,” which is the technical term for adding extra arguments to the function. We know that `identity'` is a function; we can add its argument explicitly.

```
# let identity' 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, but now it is properly polymorphic. 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; it would not be necessary in a purely functional language. We'll revisit this in Chapter 8.

### 5.1.2 Comparison with other languages

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;
}

char *string_identity(char *s)
{
    return s;
}
```

Another kind of polymorphism is *overloading* (also called *ad hoc* polymorphism). Overloading allows several functions to have the same name, but different types. When that function is applied, the compiler selects the appropriate function by checking 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 `+` expression is already overloaded).

```
class Adder {
    int Add(int i, int j) {
        return i + j;
    }
    float Add(float x, float y) {
        return x + y;
    }
}
```

```

    }
    String Add(String s1, String s2) {
        return s1.concat(s2);
    }
}

```

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

OCaml does *not* provide overloading. There are probably two main reasons. One is technical: 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 may become undecidable. *That* would be a problem.

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

I'm not sure I buy this argument. 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!

## 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 type. The syntax for a tuple is a sequence of expression 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 the type of the components, separated by a `*`. In this case, the type of the pair is `int * string`.

Tuples can be “deconstructed” using 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 function `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"
```

Note that these functions are polymorphic. The `fst` and `snd` functions 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.

```
# type coord = int * int;;
type coord = int * int
# 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 8.

```
# (* Name, Height, Phone, Salary *)
type db_entry = string * float * string * float;;
type db_entry = string * float * string * float
# let name_of_entry (name, _, _, _) = name;;
val name_of_entry : 'a * 'b * 'c * 'd -> 'a = <fun>
# let jason = ("Jason", 6.25, "626-395-6568", 50.0);;
val jason : string * float * string * float =
  "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 contains 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 is the “cons” of expression  $e_1$  onto the list  $e_2$ .

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

The bracket syntax  $[e_1; \dots; e_n]$  is an alternate syntax for the list containing the values computed by  $e_1, \dots, e_n$ .

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

Lists can be destructured 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

```

These functions can also be polymorphic. The function to check if a value  $x$  is in a list  $l$  could 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 "Jason" ["Hello"; "World"];;
- : bool = false
# mem "Dave" ["I'm"; "afraid"; "Dave"; "I"; "can't"; "do"; "that"];;
- : bool = true

```

This function takes an argument of any type  $'a$ , and checks if the element is in the  $'a$  list.

The standard “map” function, like `List.map`, is 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 map function takes a *function* of type  $'a \rightarrow 'b$  (this argument function takes a value of type  $'a$  and returns a value of type  $'b$ ), and a list containing elements of type  $'a$ . It returns a  $'b$  list. Equivalently,

$$\text{map } f [v_1; \dots; v_n] = [f v_1; \dots; f v_n].$$

Lists are commonly used to represent sets of values, or key-value relationships. The `List` library contains many list functions. The `List.assoc` function returns the value for a key in a list.

```

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

```

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

**Id:** 276469997

**end**

**Implementation:**

**begin**

## Chapter 6

# Unions

```
include Base_theory
```

Disjoint unions, also called *tagged unions* or *variant records*, are ubiquitous in OCaml programs. A disjoint union represents a set of *cases* of a particular type. For example, we may say that a binary tree contains nodes that are either *interior* nodes, or *leaves*. Suppose that the interior nodes have a label of type 'a. In OCaml, this would be expressed with the following type definition.

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

The cases are separated by a vertical dash (the | char). Each has a name and an optional set of values. The name must begin with an uppercase letter. In this case, the type of the definition is 'a btree, and the interior node Node has three values: a label of type 'a, a left child of type 'a btree, and a right child of type 'a btree. The type btree is parameterized by the type argument 'a.

The tags (like Node and Leaf), are called *constructors*. Technically, the constructors can be viewed as functions that *inject* values into the disjoint union. Thus, the Node constructor would be a function of type ('a \* 'a btree \* 'a btree) -> 'a btree. Note that OCaml does not allow constructors to be passed as arguments.

```
# Leaf;;  
- : 'a btree = Leaf  
# Node (1, Leaf, Leaf);;  
- : int btree = Node (1, Leaf, Leaf)
```

A value with a union type is a value having *one of* the cases. The value can be recovered through pattern matching. For example, a function that counts the number of interior nodes in a value of type 'a btree would be defined as follows.

```
# 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
```

## 6.1 Unbalanced binary trees

To see how this works, let's build a simple data structure for unbalanced binary trees that represent 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 by induction on the tree: 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
```

## 6.2 Unbalanced, ordered, binary trees

One problem with the unbalanced tree is that the complexity of the membership operation is  $O(n)$ , where  $n$  is cardinality of the set. We can improve this slightly by *ordering* the nodes in the tree. The invariant we 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 need to 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 does build balanced trees. If elements are inserted in order, the tree will be maximally unbalanced, with all the elements inserted along the right branch.

For the membership function, we can take advantage of the set ordering during the search.

```
# let rec mem x = function
  Leaf -> false
  | Node (y, left, right) ->
    if x < y then
      mem x left
    else if x > y then
      mem x right
    else (* x = y *)
      true;;
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.

Comment!section

red-black trees}

To correct the problem with linear complexity, we can use *balanced* trees. One common data structure is red-black trees, which add a label, either Red or Black to each of the interior nodes. Several new invariants are maintained:

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 btree =
  Node of color * 'a btree * 'a * 'a btree
  | Leaf
```

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

```
let rec mem x = function
  Leaf -> false
  | Node (_, a, y, b) ->
    if x < y then mem x a
    else if x > y then mem x b
    else true
```

The `insert` function is made a little more difficult because the invariants have to be maintained during the insertion. We can do this 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. If this happens, the `balance` function migrates the Red label upward in the tree.

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

let insert x s =
  let rec ins = function
    Leaf -> Node (Red, Leaf, x, Leaf)
  | Node (color, a, y, b) as s ->
    if x < y then balance (color, ins a, y, b)
    else if x > y then balance (color, a, y, ins b)
    else s
  in
    match ins s with (* guaranteed to be non-empty *)
      Node (_, a, y, b) -> Node (Black, a, y, b)
    | Leaf -> raise (Invalid_argument "insert");;
val balance : color * 'a btree * 'a * 'a btree -> 'a btree = <fun>
val insert : 'a -> 'a btree -> 'a btree = <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 violations of invariant 2 (nested Red nodes), 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 constant time. This means that the `insert` and `mem` functions both take time  $O(\log n)$ .

```
# let empty = Leaf;;
val empty : 'a btree = Leaf
# 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; 9; 5; 7; 11];;
val s : int btree =
  Node
    (Black, Node (Black, Node (Red, Leaf, 3, Leaf), 5, Leaf), 7,
      Node (Black, Node (Red, Leaf, 9, Leaf), 11, Leaf))
# mem 5 s;;
- : bool = true
# mem 6 s;;
- : bool = false

```

### 6.3 Some common built-in unions

A few of the types we have already seen are defined as unions. The built-in Boolean type is defined as a union (the `true` and `false` keywords are treated as capitalized identifiers).

```

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

```

The list type is similar: one again, the compiler treats the `[]` and `::` identifiers as capitalized identifiers for the scope of the definition.

```

# type 'a list = [] | :: of 'a * 'a list;;
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 in this case.

```

# 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-default values.

**Id:** 1018510896

**end**

**Implementation:**

**begin**



## Chapter 7

# Exceptions

Exceptions are used in OCaml as a control mechanism, either to signal errors, or to control the flow of execution. When an exception is raised, the current execution is aborted, and control is thrown to the innermost exception handler, which may choose to handle the exception, or pass it through to the next exception handler.

Exceptions were designed as a more elegant alternative to explicit error handling in more traditional languages. In Unix/C, for example, most system calls return `-1` on failure, and `0` on success. System code tends to be cluttered with explicit error handling code that obscures the intended operation of the code. In the OCaml `Unix` module, the system call stubs raise an exception on failure, allowing the use of a single error handler for a block of code. In some ways, this is like the `setjmp/longjmp` interface in C, but OCaml exceptions are safe.

To see how this works, consider the `List.assoc` function, which is defined as follows.

```
# let rec assoc key = function
  (k, v) :: l ->
    if k = key then
      v
    else
      assoc key l
  | [] ->
    raise Not_found;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# let l = [1, "Hello"; 2, "World"];;
val l : (int * string) list = [1, "Hello"; 2, "World"]
# assoc 2 l;;
- : string = "World"
```

```
# assoc 3 1;;
Uncaught exception: Not_found
# "Hello" ^ assoc 2 1;;
- : string = "HelloWorld"
# "Hello" ^ assoc 3 1;;
Uncaught exception: Not_found
```

In the first case, `assoc 2 1`, the key is found in the list and its value is returned. In the second case, `assoc 3 1`, the key 3 does not exist in the list, and the exception `Not_found` is raised. There is no explicit exception handler, and the top loop default handler is invoked.

Exceptions are declared with the `exception` keyword, and their syntax has the same form as a constructor declaration in a union type. Exceptions are raised with the `raise` function.

```
# exception Abort of int * string;;
exception Abort of int * string
# raise (Abort (1, "GNU is not Unix"));
Uncaught exception: Abort(1, "GNU is not Unix")
```

Exception handlers have the same form as a match pattern match, using the `try` keyword. The syntax is as follows:

```
try e with
  p1 -> e1
| p2 -> e2
:
| pn -> en
<
```

First,  $e$  is evaluated. If it does not raise an exception, the value is returned as the result of the `try` statement. Otherwise, if an exception is raised during evaluation of  $e$ , the exception is matched against the patterns  $p_1, \dots, p_n$ . If the exception matches pattern  $p_i$ , the expression  $e_i$  is evaluated and returned as the result. Otherwise, if no pattern matches, the exception is propagated to the next exception handler.

```
# try "Hello" ^ assoc 2 1 with
  Abort (i, s) -> s
| Not_found -> "Not_found";;
- : string = "HelloWorld"
# try "Hello" ^ assoc 3 1 with
  Abort (i, s) -> s
| Not_found -> "Not_found";;
- : string = "Not_found"
# try "Hello" ^ assoc 3 1 with
  Abort (i, s) -> s;;
Uncaught exception: Not_found
```

Exceptions are also used to manage control flow. For example, consider the binary trees in the previous chapter.

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

Suppose we wish to build a `replace` function that replaces a value in the set. The expression `replace x y s` should replace value `x` with `y` in tree `s`, or raise the exception `Not_found` if the value does not exist.

```
# let rec replace x y = function
  Leaf -> raise Not_found
  | Node (left, z, right) ->
    let left, mod_left =
      try replace x y left, true with
        Not_found -> left, false
    in
    let right, mod_right =
      try replace x y right, true with
        Not_found -> right, false
    in
    if z = x then
      Node (left, y, right)
    else if mod_left || mod_right then
      Node (left, x, right)
    else
      raise Not_found;;
val replace : 'a -> 'a -> 'a btree -> 'a btree = <fun>
```

In this function, the left and right subtrees are recursively modified. The `mod_left` and `mod_right` flags are set iff the corresponding branches were modified. If neither branch is modified, the `Not_found` exception is raised.



## Chapter 8

# Records, Arrays, Exceptions, and Side-Effects

In this chapter we discuss the remaining data types, all of which allow side-effects. The record type can be viewed as the type of tuples where the components are labeled. An array is a fixed-size vector of items with constant time access to each element. There are operations to modify some of the fields in a record, and any of the fields in an array.

### 8.1 Records

A record is a labeled collection of values of arbitrary types. The syntax for a record type is a set of field type definitions surrounded by braces, and separated by semicolons. Field types are declared as `label : type`, where the label is an identifier that must begin with a lowercase letter or an underscore. For example, the following record redefines the database entry from Chapter 5.

```
# type db_entry =  
  { name : string;  
    height : float;  
    phone : string;  
    salary : float  
  };;  
type db_entry = { name: string; height: float; phone: string; salary: float }
```

The syntax for a value is similar to the type declaration, but the fields are defined as `label = expr`. Here is an example database entry.

```
# let jason =
  { name = "Jason";
    height = 6.25;
    phone = "626-395-6568";
    salary = 50.0
  };;
val jason : db_entry =
  {name="Jason"; height=6.25; phone="626-395-6568"; salary=50}
```

There are two ways to get access to the fields in a record. The projection operation `r.l` returns the field labeled `l` in record `r`.

```
# jason.height;;
- : float = 6.25
# jason.phone;;
- : string = "626-395-6568"
```

Pattern matching can also be used to access the fields of a record. The syntax for a pattern is like a record value, but the fields contain a label and a pattern `label = patt`.

```
# let { name = n; height = h } = jason;;
val n : string = "Jason"
val h : float = 6.25
```

There are two update operations: a functional version, and an imperative version. The functional version produces a copy of a record with new values for the specified fields. The syntax for a functional update uses the `with` keyword in a record definition.

```
# let dave = { jason with name = "Dave"; height = 5.9 };;
val dave : db_entry =
  {name="Dave"; height=5.9; phone="626-395-6568"; salary=50}
# jason;;
- : db_entry = {name="Jason"; height=6.25;
               phone="626-395-6568"; salary=50}
```

### 8.1.1 Record updates

Record fields can also be modified by assignment, but *only if the record field is declared as **mutable***. The syntax for a mutable field uses the `mutable` keyword before the field label. For example, if we wanted to allow salaries to be modified, we would redeclare the record entry as follows.

```
# type db_entry =
  { name : string;
    mutable height : float;
```

```

        phone : string;
        mutable salary : float
    };;
type db_entry =
  { name: string;
    height: float;
    phone: string;
    mutable salary: float }
# let jason =
  { name = "Jason";
    height = 6.25;
    phone = "626-395-6568";
    salary = 50.0
  };;
val jason : db_entry =
  {name="Jason"; height=6.25; phone="626-395-6568"; salary=50}

```

The syntax for a field update is `r.label <- expr`. For example, if we want to give `jason` a raise, we would use the following statement.

```

# jason.salary <- 150.0;;
- : unit = ()
# jason;;
- : db_entry = {name="Jason"; height=6.25; phone="626-395-6568"; salary=150}

```

Note that the assignment statement itself returns the canonical unit value `()`. That is, it doesn't really return a value, unlike the functional update. A functional update creates a completely new copy of a record; assignments to the copies will be independent.

```

# let dave = { jason with name = "Dave" };;
val dave : db_entry =
  {name="Dave"; height=6.25; phone="626=395-6568"; salary=150}
# dave.salary <- 180.0;;
- : unit = ()
# dave;;
- : db_entry = {name="Dave"; height=6.25; phone="626=395-6568"; salary=180}
# jason;;
- : db_entry = {name="Jason"; height=6.25; phone="626=395-6568"; salary=150}

```

### 8.1.2 Field label namespace

One important point: the namespace for record field labels is flat. That is, all labels are defined at the toplevel in the same namespace. This is important if you intend to declare records with the same field names. If you do, the original labels will be lost! For example, consider the following sequence.

```

# type rec1 = { name : string; height : float };;
type rec1 = { name: string; height: float }

# let jason = { name = "Jason"; height = 6.25 };;
val jason : rec1 = {name="Jason"; height=6.25}

# type rec2 = { name : string; phone : string };;
type rec2 = { name: string; phone: string }

# let dave = { name = "Dave"; phone = "626-395-6568" };;
val dave : rec2 = {name="Dave"; phone="626-395-6568"}

# jason.name;;
Characters 0-5:
This expression has type rec1 but is here used with type rec2

# dave.name;;
- : string = "Dave"

# let bob = { name = "Bob"; height = 5.75 };;
Characters 10-41:
The label height belongs to the type rec1
but is here mixed with labels of type rec2

```

In this case, the name field was redefined. At this point, the original `rec1.name` label is lost, making it impossible to access the name field in a value of type `rec1`, and impossible to construct new values of type `rec1`. It is, however, permissible to use the same label names in separate files, as we will see in Chapter 11.

Comment!section

Mutable variables are common enough in OCaml programs that a special form is defined just for this case. Mutable fields use the `ref` function.

```

# let i = ref 1;;
val i : int ref = {contents=1}

```

The built-in type `'a ref` is defined using a regular record definition; the normal operations can be used on this record.

```

type 'a ref = { mutable contents : 'a }

```

Dereferencing uses the `!` operator, and assignment uses the `:=` infix operator.

```

# i := 17;;
- : unit = ()
# !i;;
- : int = 17

```

Don't get confused with the `!` operator in C here. the following code can be confusing.

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

You may be tempted to read `if !flag then ...` as testing if the `flag` is false. This is *not* the case; the `!` operator is more like the `*` operator in C.

### 8.1.3 Value restriction

As we mentioned in Section ??, side-effect interact with type inference. We considered, for example, an “identity” function that saves a value on its first call, and returns that value on all future calls. This function is not properly polymorphic. We can illustrate this using a single variable.

```
# 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 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 values. Values include constants, and constructors with fields that are values, and non-mutable records with fields that are values. An application is *not* a value, and a record with mutable fields is not a value. By this definition, the `x` and `one_shot` variables cannot be polymorphic, as the type constants `'a` indicate.

## 8.2 Arrays and strings

Arrays are fixed-size vectors of values. All of the values must have the same type. The fields in the array may be accessed and modified in constant

time. Arrays can be created with the `[|e1; ...; en|]` syntax, which creates an array of length  $n$  initialized with the values computed from the expressions  $e_1, \dots, e_n$ .

```
# let a = [|1; 3; 5; 7|];;
val a : int array = [|1; 3; 5; 7|]
```

Fields can be accessed with the `a.(i)` construction. Array indices start from 0; array bounds are checked.

```
# a.(0);;
- : int = 1
# a.(1);;
- : int = 3
# a.(5);;
Uncaught exception: Invalid_argument("Array.get")
```

Fields are updated with the `a.(i) <- e` assignment statement.

```
# a.(2) <- 9;;
- : unit = ()
# a;;
- : int array = [|1; 3; 9; 7|]
```

The `Array` library module defines additional functions on arrays. Arrays of arbitrary length can be created with the `Array.create` function, which requires a length and initializer argument. The `Array.length` function returns the number of elements in the array.

```
# let a = Array.create 10 1;;
val a : int array = [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1|]
# Array.length a;;
- : int = 10
```

The `Array.blit` function can be used to copy elements from one array to another. The `blit` function requires five arguments: the source array, a starting offset into the array, the destination array, a starting offset into the destination array, and the number of elements to copy.

```
# Array.blit [| 3; 4; 5; 6 |] 1 a 3 2;;
- : unit = ()
# a;;
- : int array = [|1; 1; 1; 4; 5; 1; 1; 1; 1; 1|]
```

In OCaml, strings are a lot like packed arrays of characters. The access and update operations use the syntax `s.[i]` and `s.[i] <- c` operators.

```
# let s = "Jason";;
val s : string = "Jason"
# s.[2];;
- : char = 's'
# s.[3] <- 'y';;
- : unit = ()
# s;;
- : string = "Jasyn"
```

The `String` module defines additional functions, including the `String.length` and `String.blit` functions that parallel the corresponding `Array` operations. The `String.create` function does not require an initializer. It creates a string with arbitrary contents.

```
# String.create 10;;
- : string = "\000\011\000\000,\200\027@\000\000"
# String.create 10;;
- : string = "\196\181\027@\001\000\000\000\000\000"
```

## 8.3 Sequential execution and looping

Sequential execution is not useful in a functional language—why compute a value and discard it? In an imperative language, including a language like OCaml, sequential execution is needed to compute by side-effect.

Sequential execution is defined using the semicolon operator. The expression  $e_1;e_2$  evaluates  $e_1$ , discards the result (it probably has a side-effect), and evaluates  $e_2$ . Note that the semicolon is a *separator* (as in Pascal), not a *terminator* (as in C). The compiler produces a warning if expression  $e_1$  does not have type `unit`. As usual, heed these warnings! The `ignore : 'a -> unit` function can be used if you really want to discard a non-unit value.

There are two kinds of loops in OCaml, a `for` loop, and a `while` loop. The `while` loop is simpler; we'll start there.

### 8.3.1 while loops

The `while` loop has the syntax `while  $e_1$  do  $e_2$  done`. The expression  $e_1$  must have type `bool`. When a `while` loop is evaluated, the expression  $e_1$  is evaluated first. If it is false, the `while` loop terminates. Otherwise,  $e_2$  is evaluated, and the loop is evaluated again.

Here is an example to check if a value  $x$  is in an array  $a$ .

```
# let array_mem x a =
  let len = Array.length a in
  let flag = ref false in
```

```

let i = ref 0 in
  while !flag = false && !i < len do
    if a.(!i) = x then
      flag := true;
      i := !i + 1
    done;
  !flag;;
val array_mem : 'a -> 'a array -> bool = <fun>
# array_mem 1 [| 3; 5; 1; 6|];;
- : bool = true
# array_mem 7 [| 3; 5; 1; 6|];;
- : bool = false

```

### 8.3.2 for loop

The for loop iterates over a finite range of integers. There are two forms, one to count up, and one to count down. The syntax of these two operations is as follows.

```

for v = e1 to e2 do e3 done
for v = e1 downto e2 do e3 done

```

The for loops first evaluate  $e_1$  and  $e_2$ , which must have type `int`. The `to` form evaluates the body  $e_3$  for values of  $v$  counting up from  $e_1$  to  $e_2$ , and the `downto` form evaluates the body for values counting down from  $e_1$  to  $e_2$ . Note that the final value  $e_2$  is *included* in the evaluation.

The following code is a simpler expression for computing membership in an array (although it is somewhat less efficient).

```

# let array_mem x a =
  let flag = ref false in
    for i = 0 to Array.length a - 1 do
      if a.(i) = x then
        flag := true
      done;
  !flag;;
val array_mem : 'a -> 'a array -> bool = <fun>
# array_mem 1 [| 3; 5; 1; 6|];;
- : bool = true
# array_mem 7 [| 3; 5; 1; 6|];;
- : bool = false

```

## 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 the same 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 with the corresponding 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,out}_gen` functions have type

```
open_flag list -> int -> string -> {in,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 *ddd.ddd*
- e or E** convert a floating-point argument to decimal notation, in the style *d.dd 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 than 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` functions 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

One of the principles of modern programming is *data hiding* using *encapsulation*. An *abstract data type* (ADT) is a program unit that defines a data type and functions (also called *methods*) that operate on that data type. An ADT has two parts: a *signature* that declares the accessible data structures and methods, and an *implementation* that defines concrete implementations of the objects declared in the signature. The *implementation* is hidden: all access to the ADT must be through the methods defined in the signature.

There are several ideas behind data hiding using ADTs. First, by separating a program into distinct program units (called *modules*), the program may be easier to understand. Ideally, each module encapsulates a single *concept* needed to address the problem at hand.

Second, by hiding the *implementation* of a program module, dependencies between program modules become tightly controlled. Since all interactions must be through a module's methods, the implementation of the module can be changed without affecting the correctness of the program (as long as the behavior of the methods is preserved).

OCaml provides a *module system* that makes it easy to use the concepts of encapsulation and data hiding. In fact, in OCaml every program file acts as a abstract module, called a *compilation unit* in the OCaml terminology. A *signature* for the file can be defined in a `.mli` file with the same name. If there is no `.mli` file, the default signature includes all type and functions defined in the `.ml` file.

## 10.1 Signatures

In OCaml, a *signature* contains type definitions and function declarations for the visible types and methods in the module. To see how this works, let's revisit the binary trees we defined in Chapter 6. A binary tree defines a simple, distinct concept, and it is an ideal candidate for encapsulation.

A module signature usually contains three parts:

1. Data types used by the module.
2. Exceptions used by the module.
3. Method type declarations for all the externally visible methods defined by the module.

For the binary tree, the signature will need to include a *type* of binary trees, and type declarations for the methods for operating on the tree. First, we need to choose a filename for the compilation unit. The filename should reflect the *function* of the data structure defined by the module. For our purposes, the binary tree is a data structure used for defining a finite *set* of values, and the appropriate filename for the signature would be `fset.ml.i`.

The set data structure defines a type of sets, and three methods: an empty set, a `mem` membership function, and a `insert` insertion function. The complete signature is defined below; we'll discuss each of the parts in the following sections.

```
(* The abstract type of sets *)
type 'a t

(* Empty set *)
val empty : 'a t

(* Membership function *)
val mem : 'a -> 'a t -> bool

(* Insertion is functional *)
val insert : 'a -> 'a t -> 'a t
```

### 10.1.1 Type declarations

Type declarations in a signature can be either *transparent* or *abstract*. An abstract type declaration declares a type without giving the type definition; a transparent type declaration includes the type definition.

For the binary tree, the declaration `type 'a t` defines an *abstract* type `'a t` of sets because the type definition is left unspecified. In this case, the type definition won't be visible to other program units; they will be forced to use the methods if they want to operate on the data type. Note that

the abstract type definition is polymorphic: it is parameterized by the type variable 'a.

Alternatively, we could have chosen a transparent definition that would make the type visible to other program modules. For example, if we intend to use the unbalanced tree representation, we might include the following type declaration in the signature.

```
type 'a t =
  Node of 'a t * 'a * 'a t
  | Leaf
```

By doing this, we would make the binary tree structure *visible* to other program components; they can now use the type definition to access the binary tree directly. This would be undesirable for several reasons. First, we may want to change the representation later (by using red-black trees for example). If we did so, we would have to find and modify all the other modules that accessed the unbalanced structure directly. Second, we may be assuming that there are some invariants on values in the data structure. For example, we may be assuming that the nodes in the binary tree are ordered. If the type definition is visible, it would be possible for other program modules to construct trees that violate the invariant, leading to errors that may be difficult to find.

### 10.1.2 Method declarations

The method declarations include all the functions and values that are *visible* to other program modules. For the `fset` module, the visible methods are the `empty`, `mem`, and `insert` methods. The signature gives only the type declarations for these methods.

It should be noted that *only* these methods will be visible to other program modules. If we define helper functions in the implementation, these functions will be *private* to the implementation and inaccessible to other program modules.

## 10.2 Implementations

The module implementation is defined in a `.ml` file with the same name as the signature file. The implementation contains parts that correspond to each of the parts in the signature.

1. Data types used by the module.
2. Exceptions used by the module.

### 3. Method definitions.

The definitions do not have to occur in the same order as this in the signature, but there must be a definition for every item in the signature.

## 10.2.1 Type definitions

In the implementation, definitions must be given for each of the types in the signature. The implementation may also include other types. These types will be *private* to the implementation; they will not be visible outside the implementation.

for the `fset` module, let's use the red-black implementation of balanced binary trees. We need two type definitions: the definition of the `Red` and `Black` labels, and the tree definition itself.

```
type color =
  Red
| Black

type 'a t =
  Node of color * 'a t * 'a * 'a t
| Leaf
```

The `color` type is a private type, the `'a t` type gives the type definition for the abstract type declaration type `'a t` in the signature.

## 10.2.2 Method definitions

In the implementation we need to implement each of the methods declared in the signature. The empty method is easy: the `Leaf` node is used to implement the empty set.

```
let empty = Leaf
```

The `mem` method performs a search over the binary tree. The nodes in the tree are ordered, and we can use a binary search.

```
let rec mem x = function
  Leaf -> false
| Node (_, a, y, b) ->
  if x < y then mem x a
  else if x > y then mem x b
  else true
```

The `insert` method we needed two methods: one is the actual `insert` function, and another is the helper function `balance` that keeps the tree balanced. We can include both functions in the implementation. The `balance` function will be private, since it is not declared in the signature.

```

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

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

```

## 10.3 Building a program

Once a compilation unit is defined, the types and methods can be used in other files by prefixing the names of the methods with the *capitalized* file name. For example, the `empty` set can be used in another file with the name `Set.empty`.

Let's define another module to test the `fset` implementation. This will be a simple program with an input loop where we can type in a string. If the string is not in the set, it is added; otherwise, the loop should print out a message that the string is already added. To implement this program, we need to add another file; we'll call it `test`.

The `test` compilation unit has no externally visible types or methods. By default, the `test.mli` file should be empty. The `test` implementation should contain a function that recursively:

1. prints a prompt
2. reads a line from `stdin`
3. checks if the line is already in the set

4. if it is, then print a message

5. repeat

We'll implement this as a loop method.

```
let loop () =
  let set = ref Fset.empty in
  try
    while true do
      output_string stdout "set> ";
      flush stdout;
      let line = input_line stdin in
      if Fset.mem line !set then
        Printf.printf "%s is already in the set\n" line
      else
        Printf.printf "%s added to the set\n" line;
        set := Fset.insert line !set
    done
  with
    End_of_file ->
      ()

let _ = loop ()
```

There are a few things to note. First, we need to catch the `End_of_file` exception that is raised when the end of the input file is reached. In this case, we exit without comment. To run the loop, we include the line `let _ = loop ()`. The `let _ = ...` may seem strange: it tells the OCaml parser that this is a new top level expression. Another way to accomplish this is by adding the `;;` terminator after the last `()` expression in the loop function.

## 10.4 Compiling the program

Once the files for the program are defined, the next step is to compile them using `ocamlc`. The usage of `ocamlc` is much like `cc`. Normally, the files are compiled separately and linked into an executable. Signatures must be compiled first, followed by the implementations.

For the `fset` module, the signature can be compiled with the following command.

```
% ocamlc -c fset.mli
```

If there are no errors in the signature, this step produces a file called `fset.cmi`.

The implementations are compiled with the following command.

```
% ocamlc -c fset.ml
% ocamlc -c test.ml
```

If this step is successful, the compiler produces the files `fset.cmo` and `test.cmo`.

The modules can now be linked into a complete program using the `ocamlc` linker. The command is as follows.

```
% ocamlc -o test fset.cmo test.cmo
```

The linker requires all of the `.cmo` files to be included in the program. The order of these files is important! Each module in the link line can refer only to the modules listed *before* it. If we reverse the order of the modules on the link line, we will get an error.

```
% ocamlc -o test test.cmo fset.cmo
Error while linking test.cmo: Reference to undefined global 'Fset'
Exit 2
```

Once the program is linked, we can run it.

```
% ./test
set> hello
hello added to the set
set> world
world added to the set
set> hello
hello is already in the set
set> x
x added to the set
set> world
world is already in the set
```

#### 10.4.1 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 files in the program are evaluated in the order specified on the link line. Program files contain type and method definitions. They can also contain arbitrary expressions to be evaluated. The `let _ = loop ()` statement in the `test.ml` file is an example: it evaluates the `loop` function. Informally, this is the main loop; it is the last expression to be executed in the program.

### 10.4.2 Some common errors

When a `.ml` file is compiled, the compiler compares the implementation with the signature in the `.cmi` 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 `Fset.insert` function so that the set argument is first.

```
let insert s x =
  ...
```

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 fset.ml
The implementation fset.ml does not match the interface fset.cmi:
Values do not match:
  val insert : 'a t -> 'a -> 'a t
is not included in
  val insert : 'a -> 'a t -> 'a t
Exit 2
```

#### Missing definition errors

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

```
% ocamlc -c fset.ml
The implementation fset.ml does not match the interface fset.cmi:
The field 'insert' is required but not provided
Exit 2
```

#### 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. Suppose we were to export the definition for type `'a t`. We need to include exactly the same definition in the implementation. A correct `fset.mli` file would contain the following definition.

```
type color
```

```
type 'a t =
  Node of color * 'a t * 'a * 'a t
  | Leaf
```

Note that we must include a type definition for `color`, since it is used in the definition of the set type `'a t`. The type definition for `color` may be transparent or abstract.

Now, suppose we reorder the constructors in the interface definition for `'a t` by placing the `Leaf` constructor first.

```
type color

type 'a t =
  Leaf
  | Node of color * 'a t * 'a * 'a t
```

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

```
% ocamlc -c fset.mli
% ocamlc -c fset.ml
The implementation fset.ml does not match the interface fset.cmi:
Type declarations do not match:
  type 'a t = | Node of color * 'a t * 'a * 'a t | Leaf
is not included in
  type 'a t = | Leaf | Node of color * 'a t * 'a * 'a t
Exit 2
```

### Compile dependency errors

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

```
% ocamlc -c fset.mli
% ocamlc -c test.ml
% ocamlc -o test fset.cmo test.cmo
Files test.cmo and fset.cmo make inconsistent assumptions over interface Fset
Exit 2
```

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

## 10.5 Using open to expose a namespace

The *module\_name.method\_name* to refer to the methods in a module can get tedious. The open *module\_name* can be used to “open” a module interface, which will allow the use of unqualified names for types, exceptions, and methods. For example, the `test.ml` module can be somewhat simplified by using the open statements for the `Printf` and `Fset` modules.

```
open Printf
open Fset

let loop () =
  let set = ref empty in
  try
    while true do
      output_string stdout "set> ";
      flush stdout;
      let line = input_line stdin in
      if mem line !set then
        printf "%s is already in the set\n" line
      else
        printf "%s added to the set\n" line;
        set := insert line !set
    done
  with
  End_of_file ->
  ()

let _ = loop ()
```

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

### 10.5.1 A note about open

Be careful with the use of open. In general, fully qualified names are 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 `Fset` and `List` modules both define a `mem` function. In the `Test` module we just defined,

it may not be immediately obvious to a programmer that the `mem` symbol refers to `Fset.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` does not include an `open Mod` statement; this is done automatically by the compiler. 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 is an annoying feature of OCaml. But it preserves a simple semantics: the implementation must provide a definition for each declaration in the signature.

## 10.6 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 fset.mli
% ocamlc -c -g fset.ml
% ocamlc -c -g test.ml
% ocamlc -o test -g fset.cmo test.cmo
```

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

```
% ocamldebug ./test
Objective Caml Debugger version 2.04
```

```
(ocd) help
List of commands :
cd complete pwd directory kill help quit 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
```

(ocd)

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 `test` program, we need to know the line numbers. Let's set a breakpoint in the `loop` function, which starts in line 27 in the `Test` module. We'll want to stop at the first line of the function.

```
(ocd) break @ Test 28
Loading program... done.
Breakpoint 1 at 24476 : file Test, line 28 column 4
(ocd) run
Time : 7 - pc : 24476 - module Test
Breakpoint : 1
28   <|b|>let set = ref Fset.empty in
(ocd) n
Time : 8 - pc : 24488 - module Test
29   <|b|>try
(ocd) p set
set : string Fset.t ref = {contents=Fset.Leaf}
```

Next, let's set a breakpoint after the next input line is read and continue execution to that point.

```
(ocd) list
27 let loop () =
28   let set = ref Fset.empty in
29   <|b|>try
30     while true do
31       output_string stdout "set> ";
32       flush stdout;
33       let line = input_line stdin in
34       if Fset.mem line !set then
35         Printf.printf "%s is already in the set\n" line
36       else
37         Printf.printf "%s added to the set\n" line;
38         set := Fset.insert line !set
39     done
```

```
(ocd) break @ 34
```

```
Breakpoint 2 at 24600 : file Test, line 33 column 40
```

```
(ocd) run
```

```
set> hello
```

```
Time : 22 - pc : 24604 - module Test
```

```
Breakpoint : 2
```

```
34   <|b|>if Fset.mem line !set then
```

```
(ocd) p line
```

```
line : string = "hello"
```

When we run the program, the evaluation prompts us for an input line, and we can see the value of the line in the line variable. Let's continue and view the set after the line is added.

```
(ocd) n
```

```
Time : 24 - pc : 24628 - module Test
```

```
34   if Fset.mem line !set<|a|> then
```

```
(ocd) n
```

```
Time : 25 - pc : 24672 - module Test
```

```
37   <|b|>Printf.printf "%s added to the set\n" line;
```

```
(ocd) n
```

```
Time : 135 - pc : 24700 - module Test
```

```
37   Printf.printf "%s added to the set\n" line<|a|>;
```

```
(ocd) n
```

```
Time : 141 - pc : 24728 - module Test
```

```
38   set := Fset.insert line !set<|a|>
```

```
(ocd) n
```

```
Time : 142 - pc : 24508 - module Test
```

```
31   <|b|>output_string stdout "set> ";
```

```
(ocd) p set
```

```
set : string Fset.t ref =
```

```
{contents=Fset.Node (<abstr>, Fset.Leaf, "hello", Fset.Leaf)}
```

```
(ocd)
```

This value seems to be correct. Next, suppose we want to go back a descend into the `Fset.mem` function. We can go back to time 22 (the time just before the `Fset.mem` function is called), and use the `step` command to descend into the membership function.

```
(ocd) goto 22
set> hello
Time : 22 - pc : 24604 - module Test
Breakpoint : 7
34          <|b|>if Fset.mem line !set then
(ocd) s
Time : 23 - pc : 22860 - module Fset
39  Leaf -> <|b|>>false
(ocd) s
Time : 24 - pc : 24628 - module Test
34          if Fset.mem line !set<|a|> then
(ocd)
```

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 sometime before the call to `input_line`, and the program resumption requires another input value.

When we step into the `Fset.mem` function, we see that the membership is false (the set is the `Leaf` empty 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

The compilation units discussed in the Chapter 10 are not the only way to create modules. OCaml provides a general module system where modules can be created explicitly using the `module` keyword. There are three key parts in the module system: *signatures*, *structures*, and *functors*.

Module signatures correspond to the signatures defined in a `.mli` file, and module *structures* correspond to the implementations defined in a `.ml` file. There are several differences however. One obvious difference is that a compilation unit can contain multiple structures and signatures. Another, perhaps more important difference, is that a single signature can be used to specify multiple structures; and a structure can have multiple *signatures*.

This ability to *share* signatures and structures can have important effects on code re-use. For example, in Chapter 6, we introduced three implementations of finite sets (using unbalanced, ordered, and balanced binary trees). All three of these implementations can be expressed as structures having the *same* signature. Any of the three implementations can be used in a context that requires an implementation of finite sets.

The ability to assign multiple signatures to a structure becomes useful in larger programs composed of multiple components each spread over multiple files. The structures within a program component may make their implementations visible to one another (like a “friends” declaration in a C++ class, or a `protected` declaration for a Java method). Outside the program component, a new signature may be assigned to hide the implementation details (making them “private”).

The OCaml module system also includes *functors*, or *parameterized* structures. A functor is a *function* that computes a structure given a structure argument. Functors provide a simple way to generalize the implementation of a structure.

In the following sections, we’ll describe the three different parts of the module system by developing the finite set example in the context of the

module system.

## 11.1 Module signatures

A module signature is declared with a `module type` declaration.

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

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, including any of the following.

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

Signatures can be defined in an interface, implementation, or in the OCaml toplevel. A signature is like a type declaration—if a `.mli` file defines a signature, the *same* signature must also be defined in the `.ml` file.

For the finite sets example, the signature should include a type declaration for the sets, and method declarations for the `empty`, `mem`, and `insert` methods. For this example, we'll return to the OCaml toplevel, which will display the types of the modules we define.

```
# module type FsetSig =
  sig
    type 'a t
    val empty : 'a t
    val mem : 'a -> 'a t -> bool
    val insert : 'a -> 'a t -> bool
  end;;
module type FsetSig =
  sig
    type 'a t
    val empty : 'a t
    val mem : 'a -> 'a t -> bool
    val insert : 'a -> 'a t -> bool
  end
```

The `include` statements can be used to create a new signature that *extends* and existing signature. For example, suppose we would like to define a signature for finite sets that includes a `delete` function to remove an element of a set. One way to be to re-type the entire signature for finite sets followed by the `delete` declaration. The `include` statement performs this inclusion automatically.

```
# module type FsetDSig =
  sig
    include Fset
    val delete : 'a -> 'a t -> 'a t
  end;;
module type FsetDSig =
  sig
    type 'a t
    val empty : 'a t
    val mem : 'a -> 'a t -> bool
    val insert : 'a -> 'a t -> bool
    val delete : 'a -> 'a t -> 'a t
  end
```

## 11.2 Module structures

Module structures are defined with the `module` keyword.

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

Once again, the module *name* must begin with an uppercase letter. The *implementation* is exactly the same as the contents of a `.ml` file. It can include any of the following.

- type definitions
- exception definitions
- method definitions, using the `let` keyword
- open statements to open the namespace of another module
- `include` statements that include the contents of another module
- signature declarations
- nested structure definitions

Let's try this with the balanced binary tree example (the complete definitions for the `balance` and `insert` functions are given in Section 10.2.2.

```

# module Fset =
  struct
    type color =
      Red
      | Black

    type 'a t =
      Node of color * 'a t * 'a * 'a t
      | Leaf

    let empty = Leaf

    let rec mem x = function
      Leaf -> false
      | Node (_, a, y, b) ->
          if x < y then mem x a
          else if x > y then mem x b
          else true

    let balance = ...

    let insert x s = ...
  end;;
module Fset :
  sig
    type color = | Red | Black
    and 'a t = | Node of color * 'a t * 'a * 'a t | Leaf
    val empty : 'a t
    val mem : 'a -> 'a t -> bool
    val balance : color * 'a t * 'a * 'a t -> 'a t
    val insert : 'a -> 'a t -> 'a t
  end
# Fset.empty;;
- : 'a Fset.t = Fset.Lean
# Fset.balance;;
- : Fset.color * 'a Fset.t * 'a * 'a Fset.t -> 'a Fset.t = <fun>

```

### 11.2.1 Assigning a signature

Note that the default signature assigned to the structure exposes *all* of the types and functions in the structure, including the type definitions for the `color` and `'a t` types, as well as the `balance` function. To assign a signature to a structure, we include a type constraint using a `:` modifier of the following form.

```

module Name : SigName = struct implementation end

```

In the finite set example, we want to assign the `FsetSig` signature to the module.

```
# module Fset =
  struct
    type color =
      Red
    | Black

    type 'a t =
      Node of color * 'a t * 'a * 'a t
    | Leaf

    let empty = Leaf
    let rec mem x = ...
    let balance = ...
    let insert x s = ...
  end;;
module Fset : FsetSig
# Fset.empty;;
- : 'a Fset.t = <abstr>
# Fset.balance;;
Characters 0-12:
Unbound value Fset.balance
```

When we assign this signature, the type definition for `'a t` becomes *abstract*, and the `balance` function is no longer visible outside the module definition.

## 11.3 Functors

One problem with the implementation of finite sets that we have been using is the use of the built-in `<` comparison operation to compare values in the set. The definition of the `<` operator is implementation-specific, and it may not always define the exact ordering that we want.

To fix this problem, we can define our own comparison function, but we will need to define a separate finite set implementation for each different element type. For this purpose, we can use *functors*. A functor is a *function* on modules; the function requires a module argument, and it produces a module. Functors can be defined with the `functor` keyword, or with a more common alternate syntax.

```
module Name = functor (ArgName : ArgSig) ->
  struct implementation end
module Name (Arg : ArgSig) =
  struct implementation end
```

For the finite set example, we'll need to define an argument structure that includes a type `elt` of elements, and a comparison function `compare`. We'll have the `compare` function return one of three cases:

1. a *negative* number if the first argument is smaller than the second,
2. *zero* if the two arguments are equal,
3. a *positive* number if the first argument is larger than the second.

```
module type EltSig =
sig
  type elt
  val compare : elt -> elt -> int
end
```

The finite set signature `FsetSig` must also be modified to used a specific element type `elt`. Note that the set itself is no longer polymorphic, it is defined for a specific type of elements.

```
module type FsetSig =
sig
  type elt
  type t

  val empty : t
  val mem : elt -> t -> bool
  val insert : elt -> t -> t
end
```

Next, we redefine the set implementation as a functor. The implementation must be modified to include a type definition for the `elt` type, and the `mem` and `insert` functions must be modified to make use of the comparison function.

```
# module MakeFset (Elt : EltSig) =
struct
  type elt = Elt.elt
  type color = ...
  type t =
    Node of color * t * elt * t
    | Leaf

  let empty = Leaf

  let rec mem x = function
    Leaf -> false
    | Node (_, a, y, b) ->
```

```

    let i = Elt.compare x y in
      if i < 0 then mem x a
      else if i > 0 then mem x b
      else true

let balance = ...

let insert x s =
  let rec ins = function
    Leaf -> Node (Red, Leaf, x, Leaf)
  | Node (color, a, y, b) as s ->
    let i = Elt.compare x y in
      if i < 0 then balance (color, ins a, y, b)
      else if i > 0 then balance (color, a, y, ins b)
      else s
  in
    match ins s with (* guaranteed to be non-empty *)
      Node (_, a, y, b) -> Node (Black, a, y, b)
    | Leaf -> raise (Invalid_argument "insert")
end;;
module MakeFset :
  functor(Elt : EltSig) ->
    sig
      type elt = Elt.elt
      and color = | Red | Black
      and t = | Node of color * t * elt * t | Leaf
      val empty : t
      val mem : Elt.elt -> t -> bool
      val balance : color * t * elt * t -> t
      val insert : elt -> t -> t
    end
end

```

Note the return type. The argument type is right: the functor takes an argument module `Elt` with signature `EltSig`. But the returned module make the implementation fully visible. To fix this problem, we need to add a type constraint using the `:` modifier.

```

# module MakeFset (Elt : EltSig) : FsetSig =
  struct
    type elt = Elt.elt
    type color = ...
    type t = ...
    let empty = ...
    let rec mem x = ...
    let balance = ...
    let insert x s = ...
  end

```

```

end;;
module MakeFset : functor(Elt : EltSig) -> FsetSig

```

### 11.3.1 Using a functor

To *use* the module produced by the functor, we need to *apply* it to a specific module implementation of the `EltSig` signature. Let's define a comparison function for a finite set of integers. The comparison function is simple—we can just subtract the two arguments.

```

# module Int =
  struct
    type elt = int
    let compare = (-)
  end;;
module Int : sig type elt = int val compare : int -> int -> int end
# Int.compare 3 5;;
- : int = -2

```

Note that a type constraint would not be appropriate here. In the `EltSig` signature, the `elt` type is *abstract*. The `Int` module *satisfies* the `EltSig` signature, but we want to keep the `elt` definition visible.

```

# module Int' = (Int : EltSig);;
module Int' : EltSig
# Int'.compare 3 5;;
Characters 13-14:

```

This expression has type `int` but is here used with type `Int'.elt`

A functor is applied to an argument with the syntax *functor\_name* (*arg\_name*). To build a finite set of integers, we apply the `MakeFset` functor to the `Int` module.

```

# module IntSet = MakeFset (Int);;
module IntSet :
  sig
    type elt = MakeFset(Int).elt
    and t = MakeFset(Int).t
    val empty : t
    val mem : elt -> t -> bool
    val insert : elt -> t -> t
  end
# IntSet.empty;;
- : IntSet.t = <abstr>

```

Note the type definitions for `elt` and `t`: the OCaml compiler notes that the two types are computed from an application of the `MakeFset` functor.

### 11.3.2 Sharing constraints

In its current state, the `IntSet` module is actually useless. Once again, the problem is with type abstraction: the `elt` type is defined as an *abstract* type in the `FsetSig` signature. The OCaml compiler remembers that the type of elements `elt` is produced by an application of the functor, but it doesn't know that the argument type in the `Int` module was `int`.

```
# IntSet.insert 5 IntSet.empty;;
```

Characters 14-15:

This expression has type `int` but is here used with type

```
IntSet.elt = MakeFset(Int).elt
```

To fix this problem, we can't add a type definition in the `FsetSig` module, since we want to be able to construct finite sets with multiple different element types. The only way to fix this problem is to add a constraint on the functor type that specifies that the `elt` type produced by the functor is the *same* as the `elt` type in the argument.

These kind of type constraints are called *sharing constraints*. The argument and value of the `MakeFset` functor “share” the same `elt` type. Sharing constraints are defined by adding a `with type` constraint to a signature. The corrected definition of the `MakeFset` functor is as follows.

```
# module MakeFset (Elt : EltSig)
  : FsetSig with type elt = Elt.elt =
  struct
    type elt = Elt.elt
    type color = ...
    type t = ...
    let empty = ...
    let rec mem x = ...
    let balance = ...
    let insert x s = ...
  end;;
module MakeFset :
  functor(Elt : EltSig) ->
  sig
    type elt = Elt.elt
    and t
    val empty : t
    val mem : elt -> t -> bool
    val insert : elt -> t -> t
  end
```

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

```
# module IntSet = MakeFset (Int);;
module IntSet :
  sig
    type elt = Int.elt
    and t = MakeFset(Int).t
    val empty : t
    val mem : elt -> t -> bool
    val insert : elt -> t -> t
  end
# IntSet.empty;;
- : IntSet.t = <abstr>
# open IntSet;;
# let s = insert 3 (insert 5 (insert 1 empty));;
val s : IntSet.t = <abstr>
# mem 5 s;;
- : bool = true
# mem 4 s;;
- : bool = false
```

## Chapter 12

# 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 are perhaps not as frequently used in OCaml as in other languages like C++ or Java, because the module system provides similar features for code re-use. However, classes and objects are often appropriate in programs where extensibility is desirable.

### 12.1 The basic object system

The OCaml object system differs in one major way from the classes defined in many other languages: the object system includes both class *types* as well as class *expressions*. The two are separate, just as module signatures are separate from module structures. There are three constructs in the OCaml object system: class types are signatures for classes, classes are initial specifications for objects, and objects are instances of classes created with the `new` keyword.

#### 12.1.1 Class types

A class type is defined using a `class` type definition. The syntax of a class type declaration is as follows.

```
class type name = object declarations end
```

The *name* of the class type should begin with a lowercase letter or an underscore. The declarations can include any of the following.

- Inheritance directives with the `inherit` keyword.

- Values, declared with the `val` keyword.
- Methods, declared with the `method` keyword.
- Type constraints, declared with the `constraint` keyword.

To illustrate the object system, let's use the canonical object example: a one-dimensional movable point. The point should have methods to return and modify the position of the point.

The class type for the point includes two methods: one to `get` the position of the point, and another to `set` the position of the point. We will also include `areset` function to return the point to the origin.

```
# class type point_type =
  object
    method get : int
    method set : int -> unit
    method reset : unit
  end;;
class type point_type = object
  method get : int
  method set : int -> unit
  method reset : unit
end
```

### 12.1.2 Class expressions

A class expression gives the definition of a class. The syntax for an class expression uses the `class` keyword.

`object implementation end`

The implementation can include any of the following.

- Values, defined with the `val` keyword.
- Methods, defined with the `method` keyword.
- Type constraints, defined with the `constraint` keyword.
- Initializers, defined with the `initializer` keyword.

We can build a class of the `point_type` class type by implementing each of the fields in the class type. To implement the point, we will need to include a `pos` field that specifies the position of the point. The `get` method should return the `pos` value, and the `move` method should add an offset to the position.

```
# class point =
  object
    val mutable pos = 0
    method get = pos
    method set pos' = pos <- pos'
    method reset = pos <- 0
  end;;
class point : object
  val mutable pos : int
  method get : int
  method reset : unit
  method set : int -> unit
end
```

The `pos <- pos + off` is a *side-effect*: the value of `pos` is updated by adding the offset argument.

Note that the `pos` field is *visible* in the class type. To get the correct class type, we need to add a type constraint.

```
# class point : point_type =
  object
    val mutable pos = 0
    method get = pos
    method set pos' = pos <- pos'
    method reset = pos <- 0
  end;;
class point : point_type
```

Class expressions are templates, like function bodies. The expressions in a class expression are not evaluated when the class is defined; they are evaluated when the class is instantiated as an object.

### 12.1.3 Objects

Objects are the values created from classes using the `new` keyword. The methods in the object can be accessed by using the `#` operator.

```
# let p = new point;;
val p : point = <obj>
# p#get;;
- : int = 0
# p#set 7;;
- : unit = ()
# p#get;;
- : int = 7
# let p2 = new point;;
```

```
val p2 : point = <obj>
# p2#get;;
- : int = 0
```

### 12.1.4 Parameterized class expressions

Class expressions can be parameterized in OCaml, using a `fun` expression. For example, suppose we want to specify the initial position of the point as an argument to the class expression.

```
# class make_point_class (initial_pos : int) =
  object
    val mutable pos = initial_pos
    method get = pos
    method set pos' = pos <- pos'
    method reset = pos <- 0
  end;;
class make_point_class : int ->
  object
    val mutable pos : int
    method get : int
    method reset : unit
    method set : int -> unit
  end
```

We have to constrain the argument `initial_pos` to be an `int`: otherwise the object would be polymorphic. Specific classes can be defined by application.

```
# class point7 = make_point_class 7;;
class point7 : make_point_class
# let p = new point7;;
val p : point7 = <obj>
# p#get;;
- : int = 7
# p#reset;;
- : unit = ()
# p#get;;
- : int = 0
```

A parameterized class can also include `let` definitions in the function body. For example, we can lift the `pos` field out of the class and use a reference cell instead.

```
# class make_point_class (initial_pos : int) =
  let pos = ref initial_pos in
  object
```

```

        method get = !pos
        method set pos' = pos := pos'
        method reset = pos := 0
    end;;
class make_point_class : int ->
  object
    method get : int
    method reset : unit
    method set : int -> unit
  end

```

The body of the `class` definition is not evaluated initially—it is evaluated at object instantiation time. All `point` objects will have separate positions.

```

# let p1 = new point7;;
val p1 : point7 = <obj>
# let p2 = new point7;;
val p2 : point7 = <obj>
# p1#set 5;;
- : unit = ()
# p2#get;;
- : int = 7

```

## 12.2 Polymorphism

Class types, class expressions, and methods can also be polymorphic. For example, consider the parameterized class `make_point_class` we just defined. If we do not constrain the type of argument, we get a type of reference cells. The syntax of a polymorphic class includes the type parameters in square brackets after the `class` keyword.

```

# class ['a] make_ref_cell (x : 'a) =
  object
    val mutable contents = x
    method get = contents
    method set x = contents <- x
  end;;
class ['a] make_ref_cell :
  'a ->
  object
    val mutable contents : 'a
    method get : 'a
    method set : 'a -> unit
  end
# class int_ref = [int] make_ref_cell 0;;
class int_ref : [int] make_ref_cell

```

```

# let p = new int_ref;;
val p : int_ref = <obj>
# p#set 7;;
- : unit = ()
# p#get;;
- : int = 7
# class string_ref = [string] make_ref_cell "";;
class string_ref : [string] make_ref_cell
# let s = new string_ref;;
val s : string_ref = <obj>
# s#set "Hello";;
- : unit = ()
# s#get;;
- : string = "Hello"

```

## 12.3 Inheritance

Inheritance allows classes to be defined by extension. For example, suppose we wish to define a new point class that includes a `move` method that moves the point by a relative offset. The `move` method can be defined using the `get` and `set` methods. To be able to access these methods, we need a *self* parameter (like the `this` object in C++ or Java).

The `self` parameter is defined after the `object` keyword. We make a new class `movable_point` using the `inherit` keyword to inherit the `point` class definition.

```

# class movable_point =
  object (self)
    inherit point
    method move off =
      self#set (self#get + off)
  end;;
class movable_point :
  object
    method get : int
    method move : int -> unit
    method reset : unit
    method set : int -> unit
  end
# let p = new movable_point;;
val p : movable_point = <obj>
# p#set 7;;
- : unit = ()
# p#get;;
- : int = 7

```

```
# p#move 5;;
- : unit = ()
# p#get;;
- : int = 12
```

### 12.3.1 Multiple inheritance

Classes can also be defined by inheriting from multiple classes. For example, let's define a point class that also has a color. The color class can be defined in the normal way.

```
# type color = Black | Red | Green | Blue;;
type color = | Black | Red | Green | Blue
# class color =
  object
    val mutable color = Black
    method get_color = color
    method set_color color' = color <- color'
    method reset = color <- Black
  end;;
class color :
  object
    val mutable color : color
    method get_color : color
    method reset : unit
    method set_color : color -> unit
  end
# let c = new color;;
val c : color = <obj>
# c#set_color Green;;
- : unit = ()
# c#get_color;;
- : color = Green
```

To define a colored point we inherit from both classes. Objects in this class will have the methods and values defined in both classes.

```
# class colored_point =
  object
    inherit point
    inherit color
  end;;
```

Characters 63-74:

```
Warning: the following methods are overridden
  by the inherited class: reset
class colored_point :
```

```

object
  val mutable color : color
  method get : int
  method get_color : color
  method reset : unit
  method set : int -> unit
  method set_color : color -> unit
end
# let cp = new colored_point;;
val cp : colored_point = <obj>
# cp#get;;
- : int = 0
# cp#get_color;;

```

Note that the compiler produced a warning message when the colored point is created. The `point` and `color` *both* define a method called `reset`. Which definition does the colored point use?

```

# cp#set 7;;
- : unit = ()
# cp#set_color Red;;
- : unit = ()
# cp#reset;;
- : unit = ()
# cp#get;;
- : int = 7
# cp#get_color;;
- : color = Black

```

As usual, the compiler chooses the *last* definition of the method.

The correct version of the colored point should call *both* the `point` and `color` `reset` functions. The `colored_point` method must override the definition. To do this, we need to include a name for the object in each of the `inherit` declarations.

```

class colored_point =
  object
    inherit point as p
    inherit color as c
    method reset =
      p#reset;
      c#reset
  end;;

```

Characters 64-69:

Warning: the following methods are overridden by the inherited class:

```

reset
class colored_point :

```

```

object
  val mutable color : color
  val mutable pos : int
  method get : int
  method get_color : color
  method reset : unit
  method set : int -> unit
  method set_color : color -> unit
end
# let cp = new colored_point;;
val cp : colored_point = <obj>
# cp#set 5;;
- : unit = ()
# cp#set_color Red;;
- : unit = ()
# cp#reset;;
- : unit = ()
# cp#get;;
- : int = 0
# cp#get_color;;
- : color = Black

```

The compiler still produces a warning message, but this time the `reset` method works correctly.

### 12.3.2 Virtual methods

Virtual methods can be used to postpone the implementation of methods for definition in subclasses. For example, suppose we wish to make a point that includes a method `move` to move the object by a relative offset. One way would be to inheritance to define a new class `movable_point` based on the `point` class. Another, more general, way is to define a separate `movable` class that can be combined by multiple inheritance with any class that implements the `get` and `set` methods. This class must be declared as `virtual` because it can't be instantiated (the `get` and `set` methods are not implemented).

```

# class virtual movable =
  object (self)
    method virtual get : int
    method virtual set : int -> unit
    method move off =
      self#set (self#get + off)
  end;;
class virtual movable :
  object

```

```

    method virtual get : int
    method move : int -> unit
    method virtual set : int -> unit
  end
# let m = new movable;;

```

Characters 8-19:

One cannot create instances of the virtual class movable

Now to create the class `movable_point`, we combine the classes by multiple inheritance.

```

# class movable_point =
  object
    inherit point
    inherit movable
  end;;
class movable_point :
  object
    val mutable pos : int
    method get : int
    method move : int -> unit
    method reset : unit
    method set : int -> unit
  end
# let p = new movable_point;;
val p : movable_point = <obj>
# p#set 7;;
- : unit = ()
# p#move 5;;
- : unit = ()
# p#get;;
- : int = 12

```

Note that a `virtual` method in OCaml does not mean the same thing as a `virtual` declaration in C++. In C++, the `virtual` declaration means that a method can be overridden in subclasses. In OCaml, all methods are virtual in this sense. The OCaml `virtual` declaration just means that the method definition is omitted.

### 12.3.3 Subclassing

The `inherit` declarations in a class definition define an inheritance hierarchy. In OCaml an object can be coerced to a class type of any of its ancestors. Coercions in OCaml must be made explicitly using the `:>` operator, which requires two class types: the type of the object, and the type of the object after the coercion.

```
# let p = (cp : colored_point :=> point);;
val p : point = <obj>
# p#get;;
- : int = 0
# p#get_color;;
Characters 0-1:
This expression has type point
It has no method get_color
```

If the class type can be inferred, the first type can be omitted.

```
# let p = (cp :=> point);;
val p : point = <obj>
```

In OCaml, objects can also be coerced to any class type that has fewer methods. For example, suppose we want a “read only” colored point without the set and set\_color methods.

```
# class type read_only_point =
  object
    method get : int
    method get_color : color
  end;;
class type read_only_point =
  object
    method get : int
    method get_color : color
  end
# let ro_p = (cp : colored_point :=> read_only_point);;
val ro_p : funny_point = <obj>
# ro_p#get;;
- : int = 0
# ro_p#get_color;;
- : color = Red
# ro_p#set 5;;
Characters 0-4:
This expression has type read_only_point
It has no method set
```

#### 12.3.4 Superclassing, or typecase

In OCaml, there is no operator to coerce an object to a superclass (there is no “typecase” operator, or instanceof predicate like in Java). So for instance, once we coerce a colored\_point to a point, there is no corresponding operator for recovering the colored\_point.

This kind of problem can arise frequently in some contexts, especially when *binary* functions are defined over two objects. For example, suppose

we wish to implement an equality relation on points. The `point_equal` function should take two objects. If both objects have type `point`, and both have the same position, the `point_equal` function should return `true`. If both are `colored_points`, and have the same position and color, it should also return `true`. Otherwise, it should return `false`.

How can we define this function? One thing is clear, the `point_equal` function must have type `point -> point -> bool` because the type of point is not known beforehand. If the argument is of type `point`, how can we tell if it is actually a `colored_point`?

The easiest way to solve this problem is to use a “trick.” For each class, we add a new method that uses an *exception* to return the actual value. We will call this method `typecase`, and it will have type `unit` (since it returns the result by exception). The `point` class implements the `typecase` method as follows.

```
# class type point_type =
  object
    method get : int
    method set : int -> unit
    method reset : unit
    method typecase : unit
  end;;
class type point_type =
  object
    method get : int
    method reset : unit
    method set : int -> unit
    method typecase : unit
  end
# exception Point of point_type;;
exception Point of point_type
# class point =
  object (self)
    val mutable pos = 0
    method get = pos
    method set pos' = pos <- pos'
    method reset = pos <- 0
    method typecase = raise (Point (self :> point_type))
  end;;
      class point :
object
  val mutable pos : int
  method get : int
  method reset : unit
  method set : int -> unit
  method typecase : unit
```

```
end
```

The `typecase` method raises the `Point` exception. Note that the `self` parameter must be coerced to `point_type`.

For the `colored_point`, we perform a similar operation. First, we define the type of colored points, and the exception.

```
# class type colored_point_type =
  object
    inherit point
    inherit color
  end;;
class type colored_point_type =
  object
    val mutable color : color
    val mutable pos : int
    method get : int
    method get_color : color
    method reset : unit
    method set : int -> unit
    method set_color : color -> unit
    method typecase : unit
  end
# exception ColoredPoint of colored_point_type;;
exception ColoredPoint of colored_point_type
```

Next, we define the class, and override the `typecase` method.

```
# class colored_point =
  object (self)
    inherit point as p
    inherit color as c
    method reset =
      p#reset;
      c#reset
    method typecase =
      raise (ColoredPoint (self :> colored_point_type))
  end;;
```

Characters 77-82:

Warning: the following methods are overridden by the inherited class:

```
reset
class colored_point :
  object
    val mutable color : color
    val mutable pos : int
    method get : int
    method get_color : color
```

```

method reset : unit
method set : int -> unit
method set_color : color -> unit
method typecase : unit
end

```

Now, the `typecase` method can be used to determine the class type of a point.

```

# let p1 = new point;;
val p1 : point = <obj>
# let p2 = new colored_point;;
val p2 : colored_point = <obj>
# let p3 = (p2 :> point);;
val p3 : point = <obj>
# p1#typecase;;
Uncaught exception: Point(_)
# p2#typecase;;
Uncaught exception: ColoredPoint(_)
# p3#typecase;;
Uncaught exception: ColoredPoint(_)

```

At this point, we can define the `point_print` printing function.

```

# let point_print p =
  try p#typecase with
    Point p ->
      printf "Point: position = %d\n" p#get
    | ColoredPoint p ->
      let color =
        match p#get_color with
          Black -> "black"
        | Red -> "red"
        | Green -> "green"
        | Blue -> "blue"
      in
        printf "ColoredPoint: position = %d, color = %s\n" p#get color
    | _ ->
      raise (Invalid_argument "point_print");;
val point_print : < typecase : unit; .. > -> unit = <fun>
# p1#set 7;;
- : unit = ()
# p2#set_color Green;;
- : unit = ()
# List.iter point_print [p1; (p2 :> point); p3];;
Point: position = 7
ColoredPoint: position = 0, color = green

```

```
ColoredPoint: position = 0, color = green
- : unit = ()
```

There are two things to note. First, the `point_print` function takes *any* object with a `typecase` method—no just points. Second, we include a default exception case: if the `typecase` method returns some other exception, the argument is invalid.

## 12.4 Functional objects

In all of the examples we have given so far, the methods work by side-effect. OCaml can also be used to implement *functional* objects, where method updates produce new values by copying the self object. The syntax for a functional update uses the

```
{< ... >}
```

notation to produce a copy of the current object with *the same type* as the current object, with updated fields. The use of the update operator is important—it is the only way to preserve the current object's type.

Let's build a functional version of points. We include the `pos` field, which the `set` method replaces.

```
# class point =
  object
    val pos = 0
    method get = pos
    method set pos' = {< pos = pos' >}
  end;;
class point :
  object ('a)
    val pos : int
    method get : int
    method set : int -> 'a
  end
# let p1 = new point;;
val p1 : point = <obj>
# p1#get;;
- : int = 0
# let p2 = p1#set 5;;
val p2 : point = <obj>
# p2#get;;
- : int = 5
```

Note the type of the `set` method: on an object of type `'a`, it takes an integer argument, and returns a new object of type `'a`.

The `color` class can also be modified so that it is functional.

```
# class color =
  object
    val color = Black
    method get_color = color
    method set_color color' = {< color = color' >}
    method reset = {< color = Black >}
  end;;
class color :
  object ('a)
    val color : color
    method get_color : color
    method reset : 'a
    method set_color : color -> 'a
  end
```

What about the `colored_point` example? For the `reset` function, we need to invoke the `reset` method from both the `point` and `color` super-classes. There is no syntax to do this directly; for this purpose, we will need to make use of private methods, so that we can name the `reset` functions.

```
# class colored_point =
  object (self)
    inherit point as p
    inherit color as c
    method private p_reset = p#reset
    method private c_reset = c#reset
    method reset = self#p_reset#c_reset
  end;;
```

Characters 75-80:

Warning: the following methods are overridden by the inherited class:

```
reset
class colored_point :
  object ('a)
    val color : color
    val pos : int
    method c_reset : 'a
    method get : int
    method get_color : color
    method private p_reset : 'a
    method reset : 'a
    method set : int -> 'a
    method set_color : color -> 'a
  end
```

The resulting object has the expected behavior.

```
# let p1 = new colored_point;;
```

```
val p1 : colored_point = <obj>
# let p2 = p1#set 7;;
val p2 : colored_point = <obj>
# let p3 = p2#set_color Blue;;
val p3 : colored_point = <obj>
# p3#get;;
- : int = 7
# p3#get_color;;
- : color = Blue
# p2#get_color;;
- : color = Black
# let p4 = p3#reset;;
val p4 : colored_point = <obj>
# p4#get;;
- : int = 0
# p4#get_color;;
- : color = Black
```