Introduction to Objective Caml

Jason Hickey

DRAFT. DO NOT REDISTRIBUTE.

January 11, 2008

Contents

Preface

Objective Caml (OCaml) is a popular, expressive, high-performance dialect of ML developed by a research team at INRIA in France. This book presents a practical introduction and guide to the language, with topics ranging from how to write a program to the concepts and conventions that affect how affect how programs are developed in OCaml. The text can be divided into three main parts.

- The core language (Chapters 2–10).
- The module system (Chapters 11–13).
- Objects and class (Chapters 14–17).

This sequence is intended to follow the ordering of concepts needed as programs grow in size (though objects and classes can be introduced at any point in the development). It also happens to follow the history of Caml: many of the core concepts were present in Caml and Caml Light in the mid-1980s and early 1990s; Caml Special Light introduced modules in 1995; and Objective Caml added objects and classes in 1996.

This book is intended for programmers, undergraduate and beginning graduate students with some experience programming in a procedural programming language like C or Java, or in some other functional programming language. Some knowledge of basic data structures like lists, stacks, and trees is assumed as well.

The exercises vary in difficulty. They are intended to provide practice, as well as to investigate language concepts in greater detail, and occasionally to introduce special topics not present elsewhere in the text.

Acknowledgements This book grew out of set of notes I developed for Caltech CS134, an undergraduate course in compiler construction that I started teaching in 2000. My thanks first go to the many students who have provided comments and feedback over the years.

My special thanks go to Tim Rentsch, who provided the suggestion and impetus for turning my course notes into a textbook. Tim provided careful reading and comments on earlier forms of the text. In our many discussions, he offered suggestions on topics ranging from programming language design and terminology to writing style and punctuation. Tim's precision and clarity of thought have been indispensible, making a lasting impression on me about how to think and write about programming languages.

Heather Bergman at Cambridge University Press has been a most excellent editor, offering both encouragement and help. I also wish to thank Xavier Leroy and the members of the Caml team (project GALLIUM) at INRIA for developing OCaml, and also for their help reviewing the text and offering suggestions.

Finally, I would like to thank Takako and Nobu for their endless support and understanding.

Chapter 1

Introduction

This book is an introduction to ML programming, specifically for the Objective Caml (*OCaml*) programming language from INRIA [5, 10]. OCaml is a dialect of the ML (*Meta-Language*) family of languages, which derive from the Classic ML language designed by Robin Milner in 1975 for the LCF (*Logic of Computable Functions*) theorem prover [3].

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

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

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

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

- The ML type system is **polymorphic**, meaning that it is possible to write programs that work for values of any type. For example, it is straightforward to define generic data structures like lists, stacks, and trees that can contain elements of any type. In a language without polymorphism, the programmer would either have to write different implementations for each type (say, lists of integers *vs*. lists of floating-point values), or else use explicit coercions to bypass the type system.

```c
/*
 * A C function to
 * 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) != 0) {
      a = b;
      b = r;
   }
   return b;
}
```

```
(*
 * An OCaml function to
 * determine the greatest
 * common divisor of two
 * positive numbers a and b.
 * We assume a>b.
 *)
let rec gcd a b =
   let r = a mod b in
   if r = 0 then
      b
   else
      gcd b r
```

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

- ML implements a **pattern matching** mechanism that unifies case analysis and data destructors.
- ML includes an expressive **module system** that allows data structures to be specified and defined *abstractly*. The module system includes *functors*, which are are functions over modules that can be used to produce one data structure from another.
- OCaml is also the only widely-available ML implementation to include an **object system**. The module system and object system complement one another: the module system provides data abstraction, and the object system provides inheritance and re-use.
- OCaml includes a compiler that supports **separate compilation**. This makes the development process easier by reducing the amount of code that must be recompiled when a program is modified. OCaml actually includes two compilers: a *byte-code* compiler that produces code for the portable OCaml byte-code interpreter, and a *native-code* compiler that produces efficient code for many machine architectures.
- One other feature should be mentioned: all the languages in the ML family have a **formal semantics**, which means that programs have a mathematical interpretation, making the programming language easier to understand and explain.

1.1 Functional and imperative languages

The ML languages are mostly functional, meaning that the normal programming style is functional, but the language includes assignment and side-effects.

To compare ML with an imperative language, a comparison of two simple implementations of Euclid's algorithm is shown in Figure 1.1 (Euclid's algorithm computes the greatest common divisor of two nonnegative integers). In a language like C, the algorithm is normally implemented as a loop, and progress is made by modifying the state. Reasoning about this program requires that we reason about the program state: give an invariant for the loop, and show that the state makes progress on each step toward the goal.

In OCaml, Euclid's algorithm is normally implemented using recursion. The steps are the same, but there are no side-effects. The `let` keyword specifies a definition, the `rec` keyword specifies that the definition is recursive, and the `gcd a b` defines a function with two arguments a and b.

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

There are problems with taking too strong a stance in favor of pure 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 an intermediate approach. We use imperative code when necessary, but we encourage the use of pure functional approach whenever appropriate.

1.2 Organization

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

1.3 Additional Sources of Information

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

The OCaml reference manual [5] is available on the OCaml home page `http://www.ocaml.org/`.

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

Chapter 2

Simple Expressions

Most functional programming implementations include a runtime environment that defines a standard library and a garbage collector. They also often include an evaluator that can be used to interact with the system, called a *toploop*. OCaml provides a compiler, a runtime, and a toploop. By default, the toploop is called `ocaml`. The toploop prints a prompt (#), reads an input expression, evaluates it, and prints the result . Expressions in the toploop are terminated by a double-semicolon ';;'.

```
% ocaml
        Objective Caml version 3.10.0
# 1 + 4;;
- : int = 5
#
```

The toploop prints the type of the result (in this case, `int`) and the value (`5`). To exit the toploop, you may type the end-of-file character (usually Control-D when using a Unix[1] system, and Control-Z when using a Microsoft Windows system).

2.1 Comment convention

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

```
# 1 (* this is a comment *) + 4;;
- : int = 5
```

2.2 Basic expressions

OCaml is a *strongly typed* language. In OCaml every valid expression must have a type, and expressions of one type may not be used as expressions in another type.

[1] UNIX is a registered trademark of The Open Group.

Apart from polymorphism, which we discuss in Chapter 5, there are no implicit coercions. Normally, you do not have to specify the types of expressions. OCaml uses *type inference* [2] to figure out the types for you.

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

2.2.1 `unit`: the singleton type

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

2.2.2 `int`: the integers

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

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

$ddd\ldots$	an `int` literal specified in decimal.
`0o`$ooo\ldots$	an `int` literal specified in octal.
`0b`$bbb\ldots$	an `int` literal specified in binary.
`0x`$hhh\ldots$	an `int` literal specified in hexadecimal.

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

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

Here are some examples of integer expressions.

```
# 12345 + 1;;
- : int = 12346
# 0b1110 lxor 0b1010;;
- : int = 4
# 1 - - 2;;
- : int = 3
# 0x7fffffff;;
- : int = -1
# 0xffffffff;;
Characters 0-10:
  0xffffffff;;
  ^^^^^^^^^^
Integer literal exceeds the range of representable integers of type int
```

2.2.3 `float`: the floating-point numbers

The floating-point numbers provide dynamically scaled "floating point" numbers. The syntax of a floating point requires a decimal point, an exponent (base 10) denoted by an 'E' or 'e', or both. A digit is required before the decimal point, but not after. Here are a few examples:

0.2, 2e7, 3.1415926, 31.415926E-1, 2.

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

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

Here are some example floating-point expressions.

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

The final expression fails to type-check because the `int` operator `+` is used with the floating-point value `2.0`.

2.2.4 char: the characters

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

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

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

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

A decimal escape sequence must have exactly three decimal characters d; it specifies the ASCII character with the given decimal code. A hexadecimal escape sequence must have exactly two hexadecimal characters h.

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

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

```
# '\120';;
- : char = 'x'
# Char.code 'x';;
- : int = 120
# '\x7e';;
- : char = '~'
# Char.uppercase 'z';;
- : char = 'Z'
# Char.uppercase '[';;
- : char = '['
# Char.chr 33;;
- : char = '!'
```

2.2.5 string: character strings

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

The syntax for strings uses the double-quote symbol " as a delimiter. Characters in the string may be specified using the same escape sequences used for characters.

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

The operator ^ performs string concatenation.

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

Strings also support random access. The expression `s.[i]` returns the i'th from string s; and the expression `s.[i] <- c` replaces the i'th in string s by character c, returning a `unit` value. The `String` module (see Section 8.3) also defines many functions to manipulate strings, including the `String.length` function, which returns the length of a string; and the `String.sub` function, which returns a substring.

```
# "Hello".[1];;
- : char = 'e'
# "Hello".[0] <- 'h';;
- : unit = ()
# String.length "Ab\000cd";;
- : int = 5
# String.sub "Abcd" 1 2;;
- : string = "bc"
```

2.2.6 `bool`: the Boolean values

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

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

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

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

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

so does the expression "x = y". However it is still possible for "x = y" to be `true` even if "x == y" is not. In the OCaml implementation from INRIA, the expression "x == y"evaluates to `true` only if the two values x and y are exactly the same value, similar to the == operators in C/Java, or the function `eq?` operator in Scheme. The comparison == is a constant-time operation that runs in a bounded number of machine instructions; the comparison = is not.

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

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

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

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

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

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

2.3 Operator precedences

The precedences of operators on the basic types are as follows, listed in increasing order.

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

2.4 The OCaml type system

The ML languages are statically and strictly typed. In addition, every expression has a exactly one type. In contrast, C is a weakly-typed language: values of one type can usually be coerced to a value of any other type, whether the coercion makes sense or not. Lisp is a language that is dynamically and strictly typed: the compiler (or interpreter) will accept any program that is syntactically correct; the types are checked at run time. The strictly typed languages are safe; both Lisp and ML are *safe* languages, but 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 proving that every program that passes the type checker can never produce a machine fault, and Lisp guarantees it by checking for validity at run time. One surprising (some would say annoying) consequence is that ML has no `nil` or `NULL` values; these would potentially cause machine errors if used where a value is expected.

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

1. Every expression has exactly one type.
2. When an expression is evaluated, one of four things may happen:
   (a) it may evaluate to a *value* of the same type as the expression,
   (b) it may raise an exception (we'll discuss exceptions in Chapter 9),
   (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.

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

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

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

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

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

2.5 Compiling your code

You aren't required to use the toploop for all your programs. In fact, as your programs become larger, you will begin to use the toploop less, and rely more on the OCaml compilers. Here is a brief introduction to using the compiler; more information is given in the Chapter 11.

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

- To compile a single file, use `ocamlc -g -c` *`file`*`.ml`. This will produce a file *`file`*`.cmo`. The `ocamlopt` programs produces a file *`file`*`.cmx`. The `-g` option causes debugging information to be included in the output file.
- To link together several files into a single executable, use `ocamlc` to link the `.cmo` files. Normally, you would also specify the `-o` *`program_file`* option to specify the output file (the default is `a.out`). For example, if you have two program files `x.cmo` and `y.cmo`, the command would be:

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

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

2.6 Exercises

Exercise 2.1 For each of the following expressions, is the expression well-typed? If it is well-typed, does it evaluate to a value? If so, what is the value?

1. `1 - 2`
2. `1 - 2 - 3`
3. `1 - - 2`
4. `0b101 + 0x10`
5. `1073741823 + 1`
6. `1073741823.0 + 1e2`
7. `1 ^ 1`
8. `if true then 1`
9. `if false then ()`
10. `if 0.3 -. 0.2 = 0.1 then 'a' else 'b'`
11. `true || (1 / 0 >= 0)`
12. `1 > 2 - 1`
13. `"Hello world".[6]`
14. `"Hello world".[11] <- 's'`
15. `String.lowercase "A" < "B"`
16. `Char.code 'a'`
17. `(((())))`
18. `((((*1*))))`
19. `((*(((*))`

Chapter 3

Variables and Functions

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

`let` *identifier* `=` *expression*

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

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

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

`let` *identifier* `=` $expression_1$ `in` $expression_2$

The expression $expression_2$ is called the *body* of the `let`. The variable named *identifier* is defined as the value of $expression_1$ within the body. The *identifier* is defined only in the body $expression_2$ and not $expression_1$.

A `let` with a body is an expression; the value of a `let` expression is the value of the body.

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

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

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

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

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

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

3.1 Functions

Functions are defined with the keyword `fun`.

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

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

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

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

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

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

The keywords `begin` $\cdots$ `end` are equivalent to parentheses.

```
# increment begin 2 * 3 end;;
- : int = 7
```

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

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

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

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

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

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

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

`let` *identifier* v_1 v_2 $\cdots$ v_n `=` *expression*

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

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

3.1.1 Scoping and nested functions

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

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

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

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

In the `addi` function, the previous binding defines `i` as `5`. The second definition of `i` has no effect on the definition used for `addi`, and the application of `addi` to the argument `3` results in `3 + 5 = 8`.

3.1.2 Recursive functions

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

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

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

```
# let power_broken i x =
    if i = 0 then
      1.0
    else
      x *. (power_broken (i - 1) x);;
Characters 70-82:
      x *. (power_broken (i - 1) x);;
           ^^^^^^^^^^^^
Unbound value power_broken
```

Mutually recursive definitions (functions that call one another) can be defined using the and keyword to connect several `let` definitions.

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

3.1.3 Higher order functions

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

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

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

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

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

```
# f' 5.0;;
- : float = 75.0000594962
# f' 1.0;;
- : float = 3.00000024822
```

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

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

The second derivative, which we would expect to be $6x$, is way off! Ok, there are some numerical errors here.

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

3.2 Variable names

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

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

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

Note that the `*` operator requires space within the parenthesis. This is because of com-

ment conventions—comments start with `(*` and end with `*)`.

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

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

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

3.3 Labeled parameters and arguments

OCaml allows functions to have labeled and optional parameters and arguments. Labeled parameters are specified with the syntax *~label*`:` *pattern*. Labeled arguments are similar, *~label*`:` *expression*. Labels have the same syntactic conventions as variables: the label must begin with a lowercase letter or an underscore `_`.

```
# let f ~x:i ~y:j = i - j;;
val f : x:int -> y:int -> int = <fun>
# f ~y:1 ~x:2;;
- : int = 1
# f ~y:1;;
- : x:int -> int = <fun>
```

Within the type, a type expression of the form *label*`:` *type* specifies a labeled function parameter.

When all parameters are labeled, the order of the arguments does not matter, so the expression `f ~y:1` applies the function `f` to the argument labeled `~y` (the second argument), returning a function that expects an argument labeled `~x`.

Since labels are frequently the same as the parameter names, OCaml provides a shorthand where a parameter *~label* specifies both the parameter and label. Similarly, an argument *~label* represents both the label and the argument (a variable with the same name).

```
# let f ~x ~y = x - y;;
val f : x:int -> y:int -> int = <fun>
# let y = 1 in
  let x = 2 in
  f ~y ~x;;
- : int = 1
```

Optional parameters are like labeled parameters, using question mark `?` instead of a tilde `~` and specifying an optional value with the syntax `?(`*label* `=` *expression*`)`. Optional arguments are specified the same way as labeled arguments, or they may be omitted entirely.

```
# let g ?(x = 1) y = x - y;;
val g : ?x:int -> int -> int = <fun>
# g 1;;
- : int = 0
# g ~x:3 4;;
- : int = -1
```

3.3.1 Rules of thumb

Labeled, unlabeled, and optional arguments can be mixed in many different combinations. However, there are some rules of thumb to follow.

- An optional parameter should always be followed by a non-optional parameter (usually unlabeled).
- The order of labeled arguments does not matter, except when a label occurs more than once.
- Labeled and optional arguments should be specified explicitly for higher-order functions.

The reason for following an optional parameter with an unlabeled one is that, otherwise, it isn't possible to know when an optional argument has been omitted. The compiler produces a warning for function definitions with a final optional parameter.

```
# let f ~x ?(y = 1) = x - y;;
Characters 15-16:
Warning X: this optional argument cannot be erased.
  let f ~x ?(y = 1) = x - y;;
               ^
val f : x:int -> ?y:int -> int = <fun>
```

There is a slight difference between labeled and unlabeled arguments with respect to optional arguments. When an optional argument is followed only by labeled arguments, then it is no longer possible to omit the argument. In contrast, an unlabeled argument "forces" the omission.

```
# let add1 ?(x = 1) ~y ~z = x + y + z;;
val add1 : ?x:int -> y:int -> z:int -> int = <fun>
# add1 ~y:2 ~z:3;;
- : ?x:int -> int = <fun>
# let add2 ?(x = 1) ~y z = x + y + z;;
val add2 : ?x:int -> y:int -> int -> int = <fun>
# add2 ~y:2 3;;
- : int = 6
```

It is legal for a label to occur more than once in an argument list. If it does, then the arguments with that label are bound in the same order as the corresponding parameters.

```
# let h ~x:i ~x:j ?(y = 1) ~z =
    i * 1000 + j * 100 + y * 10 + z;;
val h : x:int -> x:int -> ?y:int -> z:int -> int = <fun>
# h ~z:3 ~x:4 ~y:5 ~x:6;;
- : int = 4653
```

For the final rule, explicit annotation for higher-order functions, consider the following definition of a function `apply`.

```
# let apply g = g ~x:1 2 + 3;;
val apply : (x:int -> int -> int) -> int = <fun>
```

Note that the compiler infers that the function ~g has a labeled, not an optional argument. The syntax `g ~x:1` is the same, regardless of whether the label `x` is labeled or optional, but the two are not the same.

```
# apply (fun ?(x = 0) y -> x + y);;
Characters 6-31:
  apply (fun ?(x = 0) y -> x + y);;
         ^^^^^^^^^^^^^^^^^^^^^^^^
This function should have type x:int -> int -> int
but its first argument is labeled ~?x
```

The compiler will always prefer to infer that an argument is labeled, not optional. If you want the other behavior, you can specify the type explicitly.

```
# let apply (g : ?x:int -> int -> int) = g ~x:1 2 + 3;;
val apply : (?x:int -> int -> int) -> int = <fun>
# apply (fun ?(x = 0) y -> x + y);;
- : int = 6
```

3.4 Exercises

Exercise 3.1 Which of the following let-expressions is legal? For each expression that is legal, give its type and the value that it evaluates to. Otherwise, explain why the expression is not legal.

1. `let x = 1 in x`
2. `let x = 1 in let y = x in y`
3. `let x = 1 and y = x in y`
4. `let x = 1 and x = 2 in x`
5. `let x = 1 in let x = x in x`
6. `let a' = 1 in a' + 1`
7. `let 'a = 1 in 'a + 1`
8. `let a'b'c = 1 in a'b'c`
9. `let x x = x + 1 in x 2`
10. `let rec x x = x + x in x 2`
11. ```
    let (++) f g x = f (g x) in
    let f x = x + 1 in
    let g x = x * 2 in
    (f ++ g) 1
    ```
12. `let (-) x y = y - x in 1 - 2 - 3`
13. `let rec (-) x y = y - x in 1 - 2 - 3`
14. `let (+) x y z = x + y + z in 5 + 6 7`
15. `let (++) x = x + 1 in ++x`

Exercise 3.2 What are the values of the following expressions?

1. `let x = 1 in let x = x + 1 in x`
2. ```
   let x = 1 in
   let f y = x in
   let x = 2 in
   f 0
   ```
3. ```
   let f x = x - 1 in
   let f x = f (x - 1) in
   f 2
   ```
   ```

4. ```
   let y = 2 in
   let f x = x + y in
   let f x = let y = 3 in f y in
   f 5
   ```
5. ```
   let rec factorial i =
      if i = 0 then 1 else i * factorial (i - 1)
   in
      factorial 5
   ```

Exercise 3.3 Write a function `sum` that, given two integer bounds `n` and `m` and a function `f`, computes a summation.

$$\texttt{sum n m f} = \sum_{i=n}^{m} f(i).$$

Exercise 3.4 Euclid's algorithm computes the greatest common divisor (GCD) of two integers. It is one of the oldest known algorithms, appearing in Euclid's *Elements* in roughly 300 BC. It can be defined in pseudo-code as follows, where $\leftarrow$ represents assignment.

```
gcd(n, m) =
   while m ≠ 0
      if n > m
         n ← n − m
      else
         m ← m − n
   return n
```

Write an OCaml function `%%` that computes the GCD using Euclid's algorithm (so `n %% m` is the GCD of the integers `n` and `m`). You should define it without assignment, as a recursive function. [Note, this is Euclid's original definition of the algorithm. More modern versions usually use a modulus operation instead of subtraction.]

Exercise 3.5 Suppose you have a function on integers `f : int -> int` that is monotonically increasing over some range of arguments from `0` up to `n`. That is, `f i < f (i + 1)` for any $\texttt{0} \leq \texttt{i} < \texttt{n}$. In addition `f 0 < 0` and `f n > 0`. Write a function `search f n` that finds the smallest argument `i` where $\texttt{f i} \geq \texttt{0}$.

Exercise 3.6 A *dictionary* is a data structure that represents a map from keys to values. A dictionary has three operations.

- `empty : dictionary`
- `add : dictionary -> key -> value -> dictionary`
- `find : dictionary -> key -> value`

The value `empty` is an empty dictionary; the expression `add dict key value` takes an existing dictionary `dict` and augments it with a new binding `key -> value`; and the expression `find dict key` fetches the value in the dictionary `dict` associated with the `key`.

One way to implement a dictionary is to represent it as a function from keys to values. Let's assume we are building a dictionary where the key type is `string`, the value type is `int`, and the empty dictionary maps every key to zero. This dictionary can be implemented abstractly as follows, where we write $\mapsto$ for the map from keys to values.

$$\begin{aligned} \mathit{empty} &= \mathit{key} \mapsto 0 \\ \mathit{add}(\mathit{dict}, \mathit{key}, v) &= \mathit{key}' \mapsto \begin{cases} v & \text{if } \mathit{key}' = \mathit{key} \\ \mathit{dict}(\mathit{key}) & \text{otherwise} \end{cases} \\ \mathit{find}(\mathit{dict}, \mathit{key}) &= \mathit{dict}(\mathit{key}) \end{aligned}$$

1. Implement the dictionary in OCaml.

2. Suppose we have constructed several dictionaries as follows.

```
let dict1 = add empty "x" 1
let dict2 = add dict1 "y" 2
let dict3 = add dict2 "x" 3
let dict4 = add dict3 "y" 4
```

   What are the values associated with `"x"` and `"y"` in each of the four dictionaries?

Exercise 3.7 Partial application is sometimes used to improve the performance of a multi-argument function when the function is to be called repeatedly with one or more of its arguments fixed. Consider a function $f(x, y)$ that is to be called multiple times with x fixed. First, the function must be written in a form $f(x, y) = h(g(x), y)$ from some functions g and h, where g represents the part of the computation that uses only the value x. We then write it in OCaml as follows.

```
let f x =
   let z = g(x) in
      fun y -> h(z, y)
```

Calling f on its first argument computes $g(x)$ and returns a function that uses the value (without re-computing it).

Consider one root of a quadratic equation $ax^2 + bx + c = 0$ specified by the quadratic formula $r(a, b, c) = \frac{-b+\sqrt{b^2-4ac}}{2a}$. Suppose we wish to to evaluate the quadratic formula for multiple values of a with b and c fixed. Write a function to compute the formula efficiently.

Exercise 3.8 A *stream* is an infinite sequence of values supporting an operation `hd(`s`)` that returns the first value in the stream s, and `tl(`s`)` that returns a new stream with the first element removed.

One way to implement a stream is to represent it as a function over the nonnegative integers. Given a stream `s : int -> int`, the first element is `(s 0)`, the second is `(s 1)`, *etc*. The operations are defined as follows.

```
let hd s = s 0
let tl s = (fun i -> s (i + 1))
```

For this exercise, we'll assume that we're working with streams of integers, so the type `stream` is `int -> int`. We'll write a stream as a sequence $(x_0, x_1, x_2, \ldots)$.

1. Define the following stream functions.

   - `(+:) : stream -> int -> stream`. Add a constant to a stream.
     $(x_0, x_1, x_2, \ldots)$ `+:` c `=` $(x_0 + c, x_1 + c, x_2 + c, \ldots)$.
   - `(-|) : stream -> stream -> stream`.
     Subtract two streams pointwise.
     $(x_0, x_1, x_2, \ldots)$ `-|` $(y_0, y_1, y_2, \ldots)$ `=` $(x_0 - y_0, x_1 - y_1, x_2 - y_2 \ldots)$.
   - `map : (int -> int) -> stream -> stream`.
     Apply a function to each element of the stream.
     `map` f $(x_0, x_1, x_2, \ldots)$ `=` $(f\ x_0, f\ x_1, f\ x_2, \ldots)$.

2. A "derivative" function can be defined as follows.

   ```
   let derivative s = tl s -| s
   ```

   Define a function `integral : stream -> stream` such that, for any stream s, `integral (derivative` s`) =` s `+:` c for some constant c.

Chapter 4

Basic pattern matching

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

match *expression* **with**
 | *pattern*$_1$ -> *expression*$_1$
 | *pattern*$_2$ -> *expression*$_2$
 ⋮
 | *pattern*$_n$ -> *expression*$_n$

The first vertical bar | is optional.

When a `match` expression is evaluated, the expression *expression* to be matched is first evaluated, and its value is compared with the patterns in order. If *pattern*$_i$ is the first pattern to match the value, then the expression *expression*$_i$ is evaluated and returned as the result of the `match`.

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

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

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

```
# fib 6;;
- : int = 8
```

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

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

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

4.1 Functions with matching

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

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

4.2 Pattern expressions

Larger patterns can be constructed in several different ways. The vertical bar | can be used to define a *choice* pattern $pattern_1$ | $pattern_2$ that matches many value matching $pattern_1$ or $pattern_2$. For example, we can write the Fibonacci function somewhat more succinctly by combining the first two cases.

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

The pattern *pattern* as *identifier* matches that same values as the pattern *pattern* and also binds the matched value to the identifier. For example, we can use this to shorten the Fibonacci definition further.

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

The keyword as has very low precedence; the patterns `(0 | 1) as i` and `0 | 1 as i` are the same.

Patterns can also be qualified by a predicate with the form *pattern* when *expression*. This matches the same values as the pattern *pattern*, but only when the predicate *expression* evaluates to `true`. The expression is evaluated within the context of the pattern; all variables in the pattern are bound to their matched values. Continuing with our Fibonacci example, we get yet another version.

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

4.3 Values of other types

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

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

```
# is_uppercase 'M';;
- : bool = true
# is_uppercase 'm';;
- : bool = false
```

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

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

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

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

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

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

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

```
# match 4.3 -. 1.2 with
     3.1 -> true
   | _ -> false;;
- : bool = false
```

4.4 Incomplete matches

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

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

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

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

Again, OCaml is fairly strict. In the case where the pattern does not match, it raises an *exception* (we'll see more about exceptions in Chapter 9). 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 because `i mod 2` is always `0` or `1`—it can't be 2 as the compiler suggests.

```
# 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 a wildcard case that raises an exception. The `Invalid_argument` exception is designed for this purpose. It takes a string argument that is usually used to identify the name

of the place where the failure occurred. You can generate an exception with the *raise* construction.

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

4.5 Patterns are everywhere

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

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

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

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

4.6 Exercises

Exercise 4.1 Which of the following expressions are legal in OCaml? For those that are legal, what is the type of the expression, and what does it evaluate to?

1. ```
   match 1 with
     1 -> 2
   | _ -> 3
   ```
2. ```
   match 2 with
     1 + 1 -> 2
   | _ -> 3
   ```
3. `let _ as s = "abc" in s ^ "def"`
4. `(fun (1 | 2) as i -> i + 1) 2`

Exercise 4.2 We have seen pattern matching for values of all the basic types with one notable exception—functions. For example, the following code is rejected.

```
# match (fun i -> i + 1) with
    (fun i -> i + 1) -> true;;
     ^^^
Syntax error
```

Why do you think the OCaml designers left out function matching?

Exercise 4.3 Suppose we have a crypto-system based on the following substitution cipher, where each plain letter is encrypted according to the following table.

Plain	A	B	C	D
Encrypted	C	A	D	B

For example, the string `BAD` would be encrypted as `ACB`.

Write a function `check` that, given a plaintext string s_1 and a ciphertext string s_2, returns `true` if, and only if, s_2 is the ciphertext for s_1. Your function should raise an exception if s_1 is not a plaintext string. You may wish to refer to the string operations on page 8. How does your code scale as the alphabet gets larger?

Chapter 5

Tuples, lists, and polymorphism

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

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

5.1 Polymorphism

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

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

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

function can even be applied to function arguments.

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

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

There may be times when the compiler infers a polymorphic type where one wasn't intended. In this case, the type can be constrained with the syntax `(`*s* `:` *type*`)`, where *s* can be a pattern or expression.

```
# let identity_int (i : int) = i;;
val identity_int : int -> int = <fun>
```

If the constraint is for the return type of the function, it can be placed after the final parameter.

```
# let do_if b i j = if b then i else j;;
val do_if : bool -> 'a -> 'a -> 'a = <fun>
# let do_if_int b i j : int = if b then i else j;;
val do_if_int : bool -> int -> int -> int = <fun>
```

5.1.1 Value restriction

What happens if we apply the polymorphic `identity` to a value with 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 might expect. Note the type assignment `identity' : '_a -> '_a`. The type variables `'_a` are now preceded by an underscore. These type variables specify that the `identity'` function takes an argument of *some* (as yet unknown) type, and returns a value of the same type. The `identity'` function is not truly polymorphic, because it can be used with values of only one type. When we apply the `identity'` function to a number, the type of the `identity'` function becomes `int -> int`, and it is no longer possible to apply it to a string.

This behavior is due to the *value restriction*: for an expression to be truly polymorphic, it must be an immutable value, which means 1) it is already fully evaluated, and 2) it can't be modified by assignment. For example, numbers and character 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`). Furthermore, mutable expressions like arrays and reference cells (Chapters 7–8) are not polymorphic.[1]

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

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

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

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

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

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

5.1.2 Other kinds of polymorphism

Polymorphism can be a powerful tool. In ML, a single identity function can be defined that works on values of any type. In a non-polymorphic language like C, a separate identity function would have to be defined for each type (unless the coercions are used to bypass the type system), in a style like the following.

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

[1] In the literature, the term *value* is usually restricted to immutable expressions, so the phrase "immutable value" is redundant. We prefer the redundant form, because it emphasizes the reason behind the restriction—that mutable data is not polymorphic.

```
struct complex { float real; float imag; };
struct complex complex_identity(struct complex x) { return x; }
```

Overloading

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

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

The expression `Adder.Add(5.5f, 7)` evaluates to `12.5`, the expression `Adder.Add("Hello ", "world")` evaluates to the string `"Hello world"`, and the expression `Adder.Add(5, 6)` is an error because of ambiguous overloading.

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

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

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

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

Subtype polymorphism and dynamic method dispatch

Subtype polymorphism and dynamic method dispatch are concepts used extensively in object-oriented programs. Both kinds of polymorphism are fully supported in OCaml. We discuss the object system in Chapter 14.

5.2 Tuples

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

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

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

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

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

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

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

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

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

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

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

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

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

However, it would be awkward to use tuples for defining database entries, like the following. For that purpose, records would be more appropriate. Records are defined in Chapter 8.

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

5.3 Lists

Lists are also used extensively in OCaml programs. A list is a sequence of values of the same type. There are two constructors: the `[]` expression is the empty list, and the e_1 `::` e_2 expression, called a *cons* operation, creates a *cons cell*—a new list where the first element is e_1 and the rest of the list is e_2. The shorthand notation [e_1`;` e_2`;` $\cdots$`;` e_n`]` is identical to e_1 `::` e_2 `::` $\cdots$ `::` `[]`.

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

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

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

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

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

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

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

Similarly, the map operation applies a function `f` to each element of a list `l` might be defined as follows (the `map` function is also defined in the standard library as `List.map`).

```
# let rec map f = function
   [] -> []
 | x :: l -> f x :: map f l;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun i -> (float_of_int i) +. 0.5) [1; 2; 3; 4];;
- : int list = [1.5; 2.5; 3.5; 4.5]
```

The function `map` takes a function `f` of type `'a -> 'b` (meaning the function takes a value of type `'a` and returns a value of type `'b`), and a list containing elements of type `'a`, and it returns a list containing elements of type `'b`.

Lists are also combined with tuple to represent sets of values, or a key-value relationships like the dictionaries in Exercise 3.6. The `List` library contains many list functions. For example, the `List.assoc` function returns the value associated with a key in a list of key-value pairs. This function might be defined as follows.

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

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

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

```
# let entry =
    [("name", "Jason");
     ("height", "6' 3''");
```

```
(* Non-tail-recursive *)
let rec fact1 i =
   if i = 0 then
      1
   else
      i * fact1 (i - 1)
```

```
(* Tail recursive *)
let fact2 i =
   let rec loop accum i =
      if i = 0 then
         accum
      else
         loop (i * accum) (i - 1)
   in
      loop 1
```

Figure 5.1: Two versions of a factorial function. The version on the right is tail-recursive. The version on the left is not.

```
      ("phone", "626-555-1212");
      ("salary", "50")];;
val entry : (string * string) list =
  ["name", "Jason"; "height", "6' 3''";
   "phone", "626-555-1212"; "salary", "50"]
# List.assoc "phone" entry;;
- : string = "626-555-1212"
```

Note that commas (`,`) separate the elements of the pairs in the list, and semicolons (`;`) separate the items of the list.

5.4 Tail recursion

We have seen several examples of recursive functions so far. A function is *recursive* if it calls itself. Recursion is the primary means for specifying looping and iteration, making it one of the most important concepts in functional programming.

Tail recursion is a specific kind of recursion where the value produced by a recursive call is returned directly by the caller without further computation. For example, consider the two implementations of a factorial function shown in Figure 5.1.

The factorial implementation `fact1` is not tail-recursive because value produced by the recursive call `fact1 (i - 1)` must be multiplied by `i` before it is returned by the caller.

The implementation `fact2` illustrates a standard "trick," where an extra argument, often called an *accumulator*, is used to collect the result of the computation. The function `loop` is tail-recursive because the result of the recursive call is returned directly by the caller.

5.4.1 Optimization of tail-recursion

Tail-recursion is important because it can be optimized effectively by the compiler. Consider the evaluation of a normal non-tail recursive call `i * fact1 (i - 1)` on a stack-based machine. First, the argument `(i - 1)` is evaluated and pushed on the stack together with the return address, and the function is called. When it returns, the argument and return address are removed from the stack, and the result of the call is

multiplied by `i`. In general, a non-tail-recursive function will require stack space linear in the number of recursive calls.

Now, consider the mechanics of a tail-call like `loop (i * accum) (i - 1)`. The result is to be returned directly by the caller, so instead of allocating stack space for the arguments, it is sufficient to overwrite the caller's state. That is, using ← to represent assignment, the compiler can translate the code as follows.

```
let rec loop accum i =
   if i = 0 then
      accum
   else
      accum <- i * accum;
      i <- i - 1;
      goto loop
```

By *goto loop* we mean that the function `loop` is restarted with the new values for its arguments—there is no `goto` instruction in OCaml.

The optimized function computes the same result as the original definition, but it requires only a constant amount of stack space, and it is usually much faster than the original.

5.4.2 Lists and tail recursion

Tail-recursion is especially important when programming with lists, because otherwise functions would usually take stack space linear in the length of the list. Not only would that be slow, but it would also mean that the list length is limited by the maximum stack size.

Fortunately, there are some standard techniques for writing tail-recursive functions. If the function is to return a list, one standard approach is to use an accumulator that collects the result in reverse order. For example, consider the following implementation of a function `map`.

```
let rec map f = function
   h :: t -> f h :: map f t
 | [] -> []
```

The function definition is simple, but it is not tail-recursive. To obtain a tail recursive version, we collect the result in an argument `accum`.

```
let rec rev accum = function
   h :: t -> rev (h :: accum) t
 | [] -> accum

let rec rev_map f accum = function
   h :: t -> rev_map f (f h :: accum) t
 | [] -> accum

let map f l = rev [] (rev_map f [] l)
```

Note that the result is collected in `accum` in *reverse* order, so it must be reversed (with the function `rev`) at the end of the computation. Still, traversing the list twice, once with `rev_map` and once with `rev`, is often faster than the non-tail-recursive implementation.

For clarity, we listed the `rev` function here. Normally, one would use the standard implementation `List.rev`.

5.5 Exercises

Exercise 5.1 The comma `,` that is used to separate the elements of a tuple has one of the lowest precedences in the language. How many elements do the following tuples have, and what do the expressions evaluate to?

1. `1 + 2, 3, - 5`
2. `"ABC", ( 1 , "def" ), ()`
3. `let x = 1 in x + 1, let y = 2 in y + 1, 4`

Exercise 5.2 What are the types of the following functions?

1. `let f (x, y, z, w) = x + z`
2. `let f (x, y, z, w) = (w, z, y, x)`
3. `let f [x; y; z; w] = x`
4. `let f [x; y] [z; w] = [x; z]`
5. `let f (x, y) (z, w) = [x; z]`

Exercise 5.3 One of the issues with tuples is that there is no general destructor function that takes a tuple and projects an element of it. Suppose we try to write one for triples.

```
let nth i (x, y, z) =
   match i with
      1 -> x
    | 2 -> y
    | 3 -> z
    | _ -> raise (Invalid_argument "nth")
```

1. What is the type of the `nth` function?
2. Is there a way to rewrite the function so that it allows the elements of the tuple to have different types?

Exercise 5.4 Suppose you are implementing a relational employee database, where the database is a list of tuples `name * phone * salary`.

```
let db =
   ["John", "x3456", 50.1;
    "Jane", "x1234", 107.3;
    "Joan", "unlisted", 12.7]
```

1. Write a function `find_salary : string -> float` that returns the salary of an employee, given the name.
2. Write a general function

```
select : (string * string * float -> bool) -> (string * string * float) list
```

that returns a list of all the tuples that match the predicate. For example the expression `select (fun (_, _, salary) -> salary < 100.0)` would return the tuples for John and Joan.

Exercise 5.5 We have seen that the identity function `(fun x -> x)` has type `'a -> 'a`. Are there any other functions with type `'a -> 'a`?

Exercise 5.6 In Exercise 3.7 we saw that partial application is sometimes used to improve performance of a function $f(x, y)$ under the following conditions:

- the function can be written in the form $f(x, y) = h(g(x), y)$, and
- f is to be called for multiple values of y with x fixed.

In this case, we code $f(x, y)$ as follows, so that $g(x)$ is computed when f is partially applied to its first argument.

```
let f x = h (g x)
```

Unfortunately, this technique doesn't always work in the presence of polymorphism. Suppose the original type of the function is `f : int -> 'a -> 'a`, and we want to compute the values of `(f 0 "abc")` and `(f 0 1.2)`.

```
let f' = f 0
let v1 = f' "abc"
let v2 = f' 1.2
```

What goes wrong? How can you compute both values without computing `g 0` twice?

Exercise 5.7 The function `append : 'a list -> 'a list -> 'a list` appends two lists. It can be defined as follows.

```
let rec append l1 l2 =
   match l1 with
      h :: t -> h :: append t l2
    | [] -> l2
```

Write a tail-recursive version of `append`.

Exercise 5.8 It is known that a welfare crook lives in Los Angeles. You are given lists for 1) people receiving welfare, 2) Hollywood actors, and 3) residents of Beverly Hills. The names in each list are sorted alphabetically (by <). A welfare crook is someone who appears in all three lists. Write an algorithm to find at least one crook.

Chapter 6

Unions

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

OCaml allows the definition of *exact* and *open* union types. The following syntax is used for an exact union type; we discuss open types later in Section 6.5.

```
type typename =
  | Identifier_1 of type_1
  | Identifier_2 of type_2
  .
  .
  .
  | Identifier_n of type_n
```

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

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

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

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

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

```
# let x = Real 3.2;;
val x : number = Real 3.2
```

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

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

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

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

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

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

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

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

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

6.1 Binary trees

Binary trees are often used for representing collections of data. For our purposes, a binary tree is an acyclic graph, where each node (vertex) has either zero or two nodes called *children*. If node n_2 is a child of n_1, then n_1 is called the *parent* of n_2. One node, called the *root*, has no parents; all other nodes have exactly one parent.

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

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

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

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

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

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

6.2 Unbalanced binary trees

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

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

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

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

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

6.3 Unbalanced, ordered, binary trees

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

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

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

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

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

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

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

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

6.4 Balanced red-black trees

In order to address the performance problem, we turn to an implementation of balanced binary trees. We'll use a functional implementation of red-black trees due to Chris Okasaki [9]. Red-black trees add a label, either `Red` or `Black`, to each non-leaf node. We will establish several new invariants.

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

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

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

```
type color =
   Red
 | Black

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

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

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

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

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

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

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

Note the use of nested patterns in the `balance` function. The `balance` function takes a 4-tuple, with a `color`, two `btrees`, and an element, and it splits the analysis into five cases: four of the cases are for the situation where invariant 2 needs to be re-established because `Red` nodes are nested, and the final case is the case where the tree does not need rebalancing.

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

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

6.5 Open union types (polymorphic variants)

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

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

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

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

Later, we might want extend the definition to include a constructor `‘Real` for floating-point values.

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

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

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

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

6.5.1 Type definitions for open types

Something strange comes up when we try to write a type definition using an open union type.

```
# type number = [> ‘Integer of int | ‘Real of float];;
Characters 4-50:
  type number = [> ‘Integer of int | ‘Real of float];;
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
A type variable is unbound in this type declaration.
In definition [> ‘Integer of int | ‘Real of float ] as ’a
the variable ’a is unbound
```

One way to think of the open type is that it contains the specified cases (`‘Integer of int` and `‘Real of float`) and potentially *more*—it also contains values like `‘Zero`, `‘Natural 17`, *etc*. In fact, any type constructor not explicitly mentioned in the definition is also part of the type.

Type theoretically then, any function defined over an open type must be polymorphic over the unspecified cases. Technically, this amounts to the same thing as saying that an open type is some type `’a` that includes at least the cases specified in the definition (and more). In order for the type definition to be accepted, we must write the type variable explicitly.

```
# type ’a number = [> ‘Integer of int | ‘Real of float] as ’a;;
type ’a number = ’a constraint ’a = [> ‘Integer of int | ‘Real of float ]
# let (zero : ’a number) = ‘Zero;;
val zero : [> ‘Integer of int | ‘Real of float | ‘Zero ] number = ‘Zero
```

The `as` form and the `constraint` form are two different ways of writing the same thing. In both cases, the type that is being defined is the type `’a` with an additional constraint that it includes at least the cases `‘Integer of int | ‘Real of float`. For example, it also includes the value `‘Zero`.

6.5.2 Closed union types

The notation `[> ··· ]` (with a >) is meant to suggest that the actual type includes the specified cases, and more. In OCaml, one can also write a closed type as `[< ···]` to mean that the type includes the specified values *and no more*.

```
# let string_of_number = function
     `Integer i -> string_of_int i
   | `Real x -> string_of_float x;;
val string_of_number : [< `Integer of int | `Real of float ] -> string = <fun>
# string_of_number `Zero;;
Characters 17-22:
  string_of_number `Zero;;
                   ^^^^^
This expression has type [> `Zero ] but is here used with type
  [< `Integer of int | `Real of float ]
The second variant type does not allow tag(s) `Zero
```

6.6 Some common built-in unions

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

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

The list type is similar, having the following effective definition. However, the `'a list` type is primitive in this case because `[]` is not considered a legal constructor name.

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

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

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

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

6.7 Exercises

Exercise 6.1 Suppose you are given the following definition of a list type.

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

1. Write a function `map : ('a -> 'b) -> 'a mylist -> 'b mylist`, where `map` f `[`x_0`;` x_1`;` $\cdots$`;` x_n`] = [`$f\ x_0$`;` $f\ x_1$`;` $\cdots$`;` $f\ x_n$`]`.
2. Write a function `append : 'a mylist -> 'a mylist -> 'a mylist`, where `append [`x_1`;` $\cdots$`;` x_n`] [`x_{n+1}`;` $\cdots$`;` x_{n+m}`] = [`x_1`;` $\cdots$`;` x_{n+m}`]`.

Exercise 6.2 A type of unary (base-1) natural numbers can be defined as follows,

```
type unary_number = Z | S of unary_number
```

where `Z` represents the number zero, and if i is a unary number, then `S` i is $i + 1$. For example, the number 5 would be represented as `S (S (S (S (S Z))))`.

1. Write a function to add two unary numbers. What is the complexity of your function?
2. Write a function to multiply two unary numbers.

Exercise 6.3 Suppose we have the following definition for a type of small numbers.

```
type small = Four | Three | Two | One
```

The builtin comparison `(<)` orders the numbers in reverse order.

```
# Four < Three;;
- : bool = true
```

1. Write a function `lt_small : small -> small -> bool` that orders the numbers in the normal way.
2. Suppose the type `small` defines n small integers. How does the size of your code depend on n?

Exercise 6.4 We can define a data type for simple arithmetic expressions as follows.

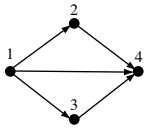
```
type unop = Neg
type binop = Add | Sub | Mul | Div
type exp =
   Constant of int
 | Unary of unop * exp
 | Binary of exp * binop * exp
```

Write a function `eval : exp -> int` to *evaluate* an expression, performing the calculation to produce an integer result.

Exercise 6.5 In Exercise 3.6 we defined the data structure called a *dictionary*. Another way to implement a dictionary is with tree, where each node in the tree has a label *and* a value. Implement a polymorphic dictionary, `('key, 'value) dictionary`, as a tree with the three dictionary operations.

```
empty : ('key, 'value) dictionary
add   : ('key, 'value) dictionary -> 'key -> 'value -> ('key, 'value) dictionary
find  : ('key, 'value) dictionary -> 'key -> 'value
```

Exercise 6.6 A *graph* (V, E) has a set of *vertices* V and a set of *edges* $E \subseteq V \times V$, where each edge (v_1, v_2) is a pair of vertices. In a *directed* graph, each edge is an arrow from one vertex to another. For example, for the graph below, the set of vertices is $V = \{1, 2, 3, 4\}$, and the set of edges is $\{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4)\}$.



One way to represent a graph is with a dictionary `(vertex, vertex list) dictionary` where each entry in the dictionary lists the outgoing edges from that vertex. Assume the following type definitions.

```
type vertex = int
type graph = (vertex, vertex list) dictionary
```

Write a function `reachable : graph -> vertex -> vertex -> bool`, where `reachable graph v1 v2` is `true` iff vertex `v2` is reachable from `v1` by following edges only in the forward direction. Your algorithm should terminate on all inputs.

Exercise 6.7 Consider the function `insert` for unbalanced, ordered, binary trees in Section 6.2. One potential problem with this implementation is that it uses the builtin comparison `(<)`. Rewrite the definition so the it is parameterized by a comparison function that, given two elements, returns on of three values `type comparison = LessThan | Equal | GreaterThan`. The expression `insert compare x tree` inserts an element `x` into the tree `tree`. The type is `insert : ('a -> 'a -> comparison) -> 'a -> 'a tree -> 'a tree`.

Exercise 6.8 A *heap* of integers is a data structure supporting the following operations.

- `makeheap : int -> heap`: create a heap containing a single element,
- `insert : heap -> int -> heap`: add an element to a heap; duplicates are allowed,
- `findmin : heap -> int`: return the smallest element of the heap.
- `deletemin : heap -> heap`: return a new heap that is the same as the original, without the smallest element.

- `meld : heap -> heap -> heap`: join two heaps into a new heap containing the elements of both.

A heap can be represented as a binary tree, where for any node a, if b is a child node of a, then $\mathit{label}(a) \leq \mathit{label}(b)$. The order of children does not matter. A *pairing heap* is a particular implementation where the operations are performed as follows.

- `makeheap i`: produce a single-node tree with `i` at the root.
- `insert h i = meld h (makeheap i)`.
- `findmin h`: return the root label.
- `deletemin h`: remove the root, and `meld` the subtrees.
- `meld h1 h2`: compare the roots, and make the heap with the larger element a subtree of the other.

1. Define a type `heap` and implement the five operations.
2. A *heap sort* is performed by inserting the elements to be sorted into a heap, then the values are extracted from smallest to largest. Write a function `heap_sort : int list -> int list` that performs a heap sort, where the result is sorted from largest to smallest.

Chapter 7

Reference cells and side-effects

Most of the values we have seen so far, like tuples and lists, are *immutable*. That is, once a value is created, it never changes. In this chapter we introduce operations and values for imperative programming—programming with assignment, side-effects, and mutable state.

The principal tool for imperative programming in OCaml is the *reference cell*, which can be viewed as a kind of "box." The box always holds a value, but the contents of the box can be replaced by assignment. The operations on reference cells are as follows.

```
val ref  : 'a -> 'a ref
val (:=) : 'a ref -> 'a -> unit
val (!)  : 'a ref -> 'a
```

Reference cells are created with the expression `ref e`, which takes the initial value `e` for the reference cell. The expression `!r` returns the value in the reference cell `r`, and the expression `r := e` replaces the contents of the reference cell with the value `e`.

For illustration, two imperative implementations of a factorial function are shown in Figure 7.1, one written in C++, and the other in OCaml. The structure of the code is similar: the variable `j` is initially defined to be `1` on line 2; then `j` is multiplied by each value $2, \ldots, i$ on the for-loop in lines 3 and 4 to produce the final value $j = 1 * 2 * \cdots * i = i!$.

Structurally, the programs look quite similar, but there is a key difference in regard to the variables. In C, the variable `j` is assigned to directly. In OCaml, variables are always immutable. Instead, the variable `j` refers to a reference cell and it is the contents of the reference cell that is modified. This also means that every time the value of the cell is needed, the expression `!j` is used.

The figure also shows examples of sequencing and looping. In OCaml, semicolons are used as separators to specify sequencing of evaluation $\mathit{expression}_1$ `;` $\mathit{expression}_2$. To evaluate the sequence, expression $\mathit{expression}_1$ is first evaluated, the value discarded, then the expression $\mathit{expression}_2$ is evaluated and the value returned as the result of the sequence. It should be emphasized that the semicolon `;` is not a terminator, like it is in C. For example, if line 6 of the OCaml factorial were followed by a semicolon, it

Imperative factorial in C++

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

Imperative factorial in OCaml

```
1  let factorial i =
2      let j = ref 1 in
3      for k := 2 to i do
4         j := !j * k
5      done;
6      !j
```

Figure 7.1: Imperative implementations of a factorial function.

would mean that the value of the expression `!j` should be discarded—and the value of the function is defined by whatever follows the semicolon.

For-loops and while-loops are specified in one of the following forms.

for *identifier* := *expression*$_1$ **to** *expression*$_2$ **do** *expression*$_3$ **done**
for *identifier* := *expression*$_1$ **downto** *expression*$_2$ **do** *expression*$_3$ **done**
while *expression*$_1$ **do** *expression*$_2$ **done**

In a for-loop, the body *expression*$_3$ of the loop is evaluated for each value of the identifier *identifier* between *expression*$_1$ and *expression*$_2$ inclusive; the `to` form counts upward by 1, and `downto` counts down. The expressions *expression*$_1$ and *expression*$_2$ are evaluated once, before the loop is entered.

A while-loop is evaluated by first evaluating the Boolean expression *expression*$_1$; if true, then the expression *expression*$_2$ is evaluated for its side-effect. The evaluation is repeated until *expression*$_1$ is false. For comparison, the factorial is implemented with a while-loop in Figure 7.2.

7.1 Pure functional programming

The one feature that is central to all functional programming languages is that functions are first class values. Functions can be passed as arguments and stored in data structures, just like any other kind of value. In fact, this is the only strict requirement for a language to be functional, and by this definition it can be argued that many languages are functional, including C, Javascript, OCaml, and others.

A related property is *purity*; a pure functional language does not include assignment or other side-effects. Haskell is an example of a pure functional language; OCaml and most Lisp dialects are impure, meaning that they allow side-effects of some form. One reason to prefer purity is that it simplifies reasoning about programs. In mathematics, a *function* is defined as a single-valued map, meaning that if f is a function and $f(x)$ is defined for some x, then there is only one value of $f(x)$. Now, consider the following "function" written in C.

```
int index = 1;
int g(int i) {
   index = index + 1;
   return i + index;
}
```

Imperative factorial in C++	Imperative factorial in OCaml
`int factorial(int i) {` `  int j = 1;` `  while(i > 0) {` `  j *= i;` `  i--;` `  }` `  return j;` `}`	`let factorial i =` `  let j = ref 1 in` `  let i = ref i in` `  while !i > 0 do` `  j := !j * !i;` `  i := !i - 1` `  done;` `  !j`

Figure 7.2: Imperative implementations of a factorial function using while-loops.

When called, the function modifies the variable `index` by side-effect. Mathematically speaking, it is not a function because an expression like `g(0)` can have many possible values—in fact, the expression `g(0) == g(0)` is always false!

Reasoning about imperative programs can be more difficult than it is for pure functional programs because of the need to analyze the program state. Pure functional programs don't have a global mutable state, which simplifies their analysis. More precisely, pure functional programs provide *referential transparency*, meaning that if two expressions evaluate to the same value, then one can be substituted for the other without affecting the result of the computation. For example, consider an expression `f(0) + f(0)`. If the expression `f(0)` is referentially transparent, then the two occurrences can be replaced with the same value, and the expression `let x = f(0) in x + x` is equivalent (and probably more efficient).

Pure functional programming and referential transparency have many benefits. Data structures are persistent, meaning that once a data value is created, it cannot be changed or destroyed. Programs can be easier to reason about than equivalent imperative programs, and the compiler is given a great deal of latitude in optimizations.

However, purity does have its drawbacks. Side-effects can be useful in reducing code size. For example, suppose we have written a program containing a function `f(x)`. During testing, we might want to know whether `f` is ever called. With side-effects, this is easy—we add a flag that gets set whenever the function is called. No other part of the program need be changed.

```
let f_was_called = ref false

let f x =
   f_was_called := true;
   ...
```

Without side-effects, this would be difficult to do without changing other parts of the program as well.

Another, deeper, issue with purity is that it becomes impossible to construct some cyclic data structures, ruling out some commonly-used data representations for graphs, circular lists, *etc*. The problem is that, when a data value like a tuple or list is constructed, the values it refers to must already exist. In particular, a value being constructed may not, in general, refer to itself. In imperative programming, this is not an

issue, because data values can be constructed, then later mutated if a cyclic structure is desired.

A related issue is that, for some algorithms, the best known implementations are imperative. For these reasons, and perhaps others, OCaml supports side-effects, including operations on mutable data and imperative input/output operations. In the next few sections, we'll look at some common imperative data representations using reference cells.

7.1.1 Value restriction

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

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

The value restriction requires that polymorphism be restricted to immutable values. Values include functions, constants, constructors with fields that are values, and other fully-evaluated immutable expressions. A function application is *not* a value, and a mutable reference cell is not a value. By this definition, the x and one_shot variables cannot be polymorphic, as the type constants '_a indicate.

7.2 Queues

A simple imperative *queue* is a data structure supporting three operations.

```
val create : unit -> 'a queue
val add    : 'a queue -> 'a -> unit
val take   : 'a queue -> 'a
```

In a first-in-first-out (FIFO) queue, the queue contains a sequence of elements, and the first element added to the queue is the first one taken out.

One way to represent a queue would be with a list, but this would mean that one of the operations `add` or `take` would be required to scan to the end of the list. An alternative commonly used implementation is to represent the queue with two lists $(front, back)$ where elements are added to the list *front*, taken from the list *back*, and the entire sequence of elements is *`front`* `@ (List.rev` *`back`*`)`—that is, the list *back* is represented in reverse order. The first two queue functions can be implemented as follows.

```
type 'a queue = ('a list * 'a list) ref

let create () =
   ref ([], [])

let add queue x =
   let (front, back) = !queue in
   queue := (x :: front, back)
```

In the empty queue, both *front* and *back* are empty; the function `add` simply adds the element to *front* and sets the queue reference to the new value.

The function `take` is only a little more complicated. By default, values are taken from the list *back*. If the list *back* is empty, the elements in *front* are "shifted" to *back* by reversing the list.

```
let rec take queue =
   match !queue with
      (front, x :: back) ->
          queue := (front, back);
          x
    | ([], []) ->
          raise (Invalid_argument "queue is empty")
    | (front, []) ->
          queue := ([], List.rev front);
          take queue
```

Note the recursive call to `take` in the final clause, which restarts the operation once the elements have been shifted. The worst-case complexity of the function `take` is $O(n)$ where n is the number of elements in the queue. However, if we consider the amortized cost, "charging" the one extra unit of time to each `add` call to account for the list reversal, then all operations take constant $O(1)$ amortized time.

7.3 Doubly-linked lists

In the builtin type `'a list`, a list element contains a value of type `'a` and a link to the next element of the list. This means that list traversal is always ordered front-to-back.

A doubly-linked list supports efficient traversal in both directions by adding a additional link, as shown in Figure 7.3. In addition to a link to the next element, each element also has a link to the previous element. This is a cyclic data structure, so it will necessarily be imperative.

To implement the list, we'll split the operations into two parts: operations on list elements, and operations on lists. Given a list element, there are operations to get

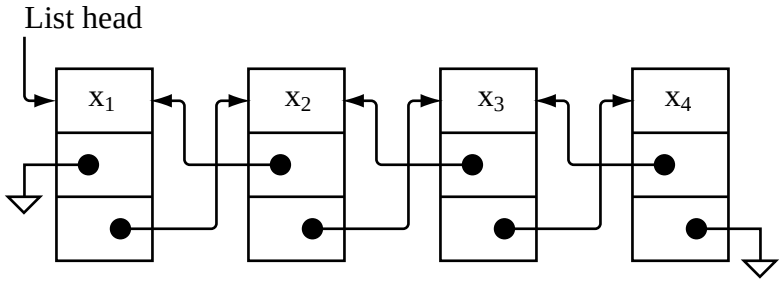


Figure 7.3: Doubly-linked list.

the next and previous elements; and there are operations for inserting and removing elements from a list.

```
(* The type of elements in the list *)
type 'a elem

val nil_elem    : 'a elem
val create_elem : 'a -> 'a elem
val get         : 'a elem -> 'a
val prev_elem   : 'a elem -> 'a elem
val next_elem   : 'a elem -> 'a elem

(* The type of lists with elements of type 'a element *)
type 'a dllist

val create : unit -> 'a dllist
val insert : 'a dllist -> 'a elem -> unit
val remove : 'a dllist -> 'a elem -> unit
```

The implementation starts with the list elements. A link to an element can be null (at the end of the list), or it can point to a real element. Instead of having two kinds of links, we can fold the two cases directly into the element type. The list is designed to be modified in place, so the links should be references to elements.

```
type 'a elem =
   Nil
 | Elem of 'a * 'a elem ref * 'a elem ref
```

With this type definition in place, the implementation of elements is fairly straightforward. A fragment is shown below; the omitted functions are similar.

```
let nil_elem = Nil
let create_elem x = Elem (x, ref Nil, ref Nil)

let get = function
   Elem (x, _, _) -> x
 | Nil -> raise (Invalid_argument "get")

let prev_elem = function
   Elem (_, prev, _) -> !prev
 | Nil -> raise (Invalid_argument "prev_elem")
```

Next, to implement the doubly-linked list, the list itself can be just be a reference to the first element. The function `create` constructs an empty list, and `insert` adds an element at the head of the list. The list operations are similar to what we would see on other imperative languages; the links are modified in place as the list is changed.

```
type 'a dllist = 'a elem ref

let create () = ref Nil

let insert list elem =
   match elem, !list with
      Elem (_, prev, next), Nil ->
         prev := Nil;
         next := Nil;
         list := elem
    | Elem (_, prev1, next1), (Elem (_, prev2, _) as head) ->
         prev1 := Nil;
         next1 := head;
         prev2 := elem;
         list := elem
    | Nil, _ ->
         raise (Invalid_argument "insert")
```

In the first case, the new element is added to an empty list; so both pointers are set to `Nil`. In the second case, the list has a head element, which is modified so that its previous link is now the new element.

Removing an element from the list is similar. If the element to be removed is the head element, then its previous link is `Nil`, and the list must be updated to refer to the next element. Otherwise, the forward-link of the previous element must be adjusted. Similarly, the back-link of the next element must be adjusted.

```
let remove list elem =
   match elem with
      Elem (_, prev, next) ->
         (match !prev with
             Elem (_, _, prev_next) -> prev_next := !next
           | Nil -> list := !next);
         (match !next with
             Elem (_, next_prev, _) -> next_prev := !prev
           | Nil -> ())
    | Nil ->
         raise (Invalid_argument "remove")
```

7.4 Memoization

Sometimes references and side-effects are used to improve performance without otherwise changing the behavior of a program. Memoization is an example of this, where a record is made of function applications as a kind of run-time optimization. If a function f is pure, and $f(e)$ is computed for some argument e, then any future computation $f(e)$ will return the same result. When a function f is memoized, the results of function applications are stored in a table, and the function is called at most once for any

argument.

In OCaml, we can implement this in a generic way. For immediate purposes, we'll represent the memo (the table) as an association list. The memoization itself can be implemented as a single higher-order function.

```
val memo : ('a -> 'b) -> ('a -> 'b)
```

That is, the function `memo` takes a function and returns a function with the same type. The intent is that the result should be a function that is equivalent, but perhaps faster. The memo can be implemented as follows.

```
1  let memo f =
2     let table = ref [] in
3     let rec find_or_apply entries x =
4        match entries with
5           (x', y) :: _ when x' = x -> y
6         | _ :: entries -> find_or_apply entries x
7         | [] ->
8            let y = f x in
9            table := (x, y) :: !table;
10           y
11    in
12    (fun x -> find_or_apply !table x)
```

The memo table is defined as a mutable table on line 2; the table is filled in with argument/result pairs as the function is called. Given an argument `x`, the function `find_or_apply` is called to search the table. If a previous entry is found (line 5), the previous value is returned. Otherwise the function is called (line 8), the result is saved in the table (line 9) by side-effect, and the value is returned. Let's try it on a computation of Fibonacci numbers.

```
let rec fib = function
   0 | 1 as i -> i
 | i -> fib (i - 1) + fib (i - 2)
```

To measure it, we can use the function `Sys.time` to measure the CPU time taken by the process.

```
# let time f x =
   let start = Sys.time () in
   let y = f x in
   let finish = Sys.time () in
   Printf.printf "Elapsed time: %f seconds\n" (finish -. start);
   y
;;
val time : ('a -> 'b) -> 'a -> 'b = <fun>
# let memo_fib = memo fib;;
val memo_fib : int -> int = <fun>
# time memo_fib 40;;
Elapsed time: 14.581937 seconds
- : int = 102334155
# time memo_fib 40;;
Elapsed time: 0.000009 seconds
- : int = 102334155
```

In the the first call, the computation of `fib 40` took roughly 15 seconds, while the second call was nearly instantaneous.

It should be noted that, although this kind of memoization does make use of side-effects, it has no effect on results of the computation (as long as it is applied to pure functions). In fact, if a function f is referentially transparent, so is `memo` f—it behaves as the original function, except that it trades space for time.

7.5 Graphs

Let's finish this chapter with a classic algorithm from graph theory: Kruskal's algorithm for minimum spanning trees. Given a connected, undirected graph, a *spanning tree* is a subset of edges that forms a tree and includes every vertex. If edges are assigned weights, the minimum spanning tree is the spanning tree with the lowest weight. The computation of minimum spanning trees has many practical purposes. One of the original uses was by Czech scientist Oscar Boruvka in 1923 to minimize the cost of electrical coverage in Bohemia.

Kruskal's algorithm is specified as follows, for a weighted graph (V, E) with vertices V and edges E.

1. Sort the edges E by weight.
2. For each edge in order of increasing weight, include the edge in the spanning tree if it would not form a cycle with the edges already in the tree, otherwise discard it.

The interesting part of the algorithm is step 2. An example is shown in Figure 7.4 for a small graph. First, the two edges with weight 5 are added to the spanning tree. Next, the edges with weights 6 and 7 are added, but the edge with weight 8 is discarded because it would produce a cycle. Similarly, the edge with weight 9 is included, but the edge with weight 11 is discarded, and the final edge 23 is discarded as well.

We can think of the algorithm as working over connected components of the graph. When an edge (v_1, v_2) is included in the tree, the connected component containing v_1 is merged with the component containing v_2. In order to be included in the spanning tree, the vertices v_1 and v_2 must belong to separate components—otherwise the edge would form a cycle.

We can implement this with a *union-find* data structure, which supports two operations.

- `find : vertex -> vertex`, where `find` v returns a canonical vertex of the set (connected component) containing v. Two vertices v_1 and v_2 are in the same component if `find` v_1 `=` `find` v_2.
- `union : vertex -> vertex -> unit`, where `union` u_1 u_2 takes the union of the sets containing canonical elements u_1 and u_2 (by side-effect).

Step 2 of Kruskal's algorithm can be written as follows, where the expression `List.iter f edges` calls the function `f` for each edge in the list of edges.

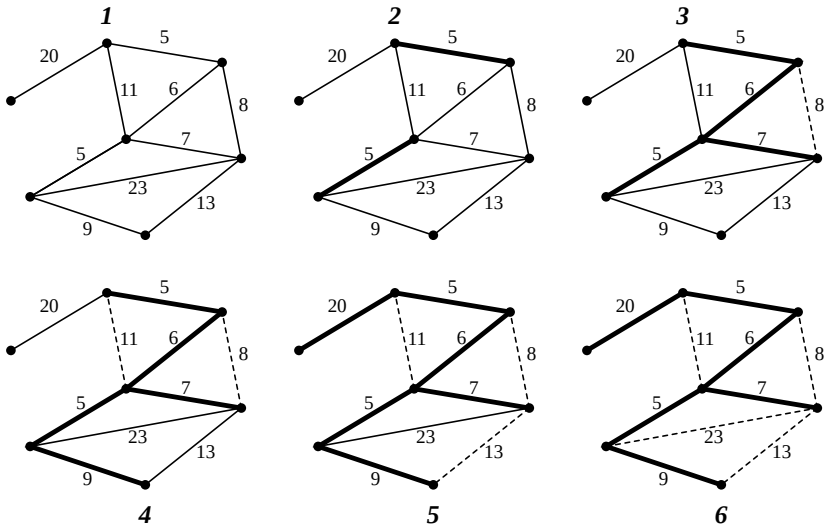


Figure 7.4: An example run of Kruskal's algorithm for computing the minimum spanning tree of a weighted undirected graph.

```
(* An edge is a triple (weight, vertex1, vertex2) *)
type 'a edge = float * 'a vertex * 'a vertex

(* A list of edges, sorted by increasing weight *)
let kruskal edges =
   let spanning_tree = ref [] in
   List.iter (fun ((_, v1, v2) as edge) ->
      let u1 = find v1 in
      let u2 = find v2 in
      if u1 != u2 then begin
         (* v1 and v2 belong to different components *)
         spanning_tree := edge :: !spanning_tree;
         union u1 u2
      end) edges;
   !spanning_tree
```

What remains is to specify the type for vertices, and to implement the functions `find` and `union`. One simple way to do so is to organize the vertices in each connected component into a tree, where each vertex has a pointer to its parent. The root of the tree is the canonical element returned by the function `find`. Given roots `u1` and `u2`, the union operation simply makes one a child of the other. For efficiency, we use the following heuristics.

- When performing a `union`, the smaller tree should become a child of the larger. We'll save the size of the tree at the root.
- When performing a `find` v, first find the root u, then traverse the path from v to u a second time, updating all parent pointers to point to u.

Both heuristics will tend to make the tree fatter. The second, called *path compression*, decreases the cost of subsequent `find` operations.

We'll represent a vertex as a pair (*`label`*, *`parent`*) where *label* is the label of the vertex, and *parent* is the parent link: either `Root i` for the root vertex, or `Parent p` if not. The `union` operation can be written as follows.

```
type 'a parent =
   Root of int
 | Parent of 'a vertex

and 'a vertex = 'a * 'a parent ref

let union ((_, p1) as u1) ((_, p2) as u2) =
   match !p1, !p2 with
      Root size1, Root size2 when size1 > size2 ->
         p2 := Parent u1;
         p1 := Root (size1 + size2)
    | Root size1, Root size2 ->
         p1 := Parent u2;
         p2 := Root (size1 + size2)
    | _ ->
         raise (Invalid_argument "union: not roots")
```

The `find` operation is implemented in two parts: the actual find operation `simple_find`, and the path compression `compress`.

```
let rec compress root (_, p) =
   match !p with
      Root _ -> ()
    | Parent v -> p := Parent root; compress root v

let rec simple_find ((_, p) as v) =
   match !p with
      Root _ -> v
    | Parent v -> simple_find v

let find v =
   let root = simple_find v in
   compress root v;
   root
```

It can be shown that with these heuristics, the complexity of step 2 of Kruskal's algorithm is $O((n+m)\alpha(n))$ where n is the number of vertices $n = |V|$, m is the number of edges $m = |E|$, and $\alpha(n)$ is the inverse of Ackermann's function. For any practical n, $\alpha(n)$ is at most 4, so Kruskal's algorithm effectively takes linear time.

The complexity argument is fairly long (see Kozen [4] for a good explanation), but the key benefit comes from path compression. Suppose path compression were not used. The first heuristic would still ensure that path lengths are at most $\log n$ because the tree at least doubles in size along each edge in the path from a vertex to the root of the tree, leading to a time complexity of $O((n+m)\log n)$. However, when path compression is used, the result of each `find` operation is effectively memoized, and the practical cost of the `find` operation becomes constant.

The principal motivation behind this example is to show that, in some cases, im-

perative programming can be both simple and efficient. While it is always possible to recode these algorithms in a pure functional language, it is not always natural, and equivalent performance may not be achievable.

7.6 Exercises

Exercise 7.1 What is the value of the following expressions?

1. `let x = ref 1 in let y = x in y := 2; !x`
2. `let x = ref 1 in let y = ref 1 in y := 2`
3. ```
   let x = ref 1 in
   let y = ref x in
   !y := 2;
   !x
   ```
4. ```
   let fst (x, _) = x in
   let snd (_, x) = x in
   let y = ref 1 in
   let x = (y, y) in
   fst x := 2;
   !(snd x)
   ```
5. ```
   let x = ref 0 in
   let y = ref [5; 7; 2; 6] in
   while !y <> [] do
      x := !x + 1;
      y := List.tl !y
   done;
   !x
   ```

Exercise 7.2 A *lazy* value is a computation that is deferred until it is needed; we say that it is *forced*. A forced value is memoized, so that subsequent forcings do not reevaluate the computation. The OCaml standard library already provides an implementation of lazy values in the `Lazy` module, but we can also construct them ourselves using reference cells and functions.

```
type 'a deferred
val defer : (unit -> 'a) -> 'a deferred
val force : 'a deferred -> 'a
```

Implement the type `'a deferred` and the functions `defer` and `force`.

Exercise 7.3 A lazy list is a list where the tail of the list is a deferred computation (a lazy list is also called a *stream*). The type can be defined as follows, where the type `deferred` is defined as in Exercise 7.2.

```
type 'a lazy_list =
   Nil
 | Cons of 'a * 'a lazy_list
 | LazyCons of 'a * 'a lazy_list deferred
```

Define the following functions on lazy lists.

```
val nil       : 'a lazy_list
val cons      : 'a -> 'a lazy_list -> 'a lazy_list
val lazy_cons : 'a -> (unit -> 'a lazy_list) -> 'a lazy_list
val is_nil    : 'a lazy_list -> bool
val head      : 'a lazy_list -> 'a
val tail      : 'a lazy_list -> 'a lazy_list
val (@@)      : 'a lazy_list -> 'a lazy_list -> 'a lazy_list
```


```

The expression $l_1$ `@@` $l_2$ appends two lazy lists in constant time.

**Exercise 7.4** The FIFO queues described in Section 7.2 are imperative; whenever a value is added to or taken from the queue, the queue is modified by side-effect. Implement a *persistent* queue with the following operations.

```
val empty : 'a queue
val add   : 'a queue -> 'a -> 'a queue
val take : 'a queue -> 'a * 'a queue
```

The expression `add queue e` produces a new queue, without affecting the contents of the original queue `queue`. The expression `take queue` returns an element of the queue `queue` and a new queue; again, the contents of the original queue `queue` are unaffected.

$\star$ Can you implement the queue so that any sequence of n `add` and `take` operations, in any order, take $O(n)$ time? Hint: consider using lazy lists (Exercise 7.3) to represent the queue, shifting the queue whenever the *front* is longer than the *back*. See Okasaki [8].

Exercise 7.5 One problem with the memoization function `memo : ('a -> 'b) -> ('a -> 'b)` in Section 7.4 is that it ignores recursive definitions. For example, the expression `memo fib` i still takes exponential time in i to compute.

To solve this, we'll need to modify the recursive definition for `fib` and perform an explicit memoization. Implement the following types and functions, where `fib = memo_fib (create_memo ())`. How fast is the Fibonacci function now?

```
type ('a, 'b) memo
val create_memo : unit -> ('a, 'b) memo
val memo_find   : ('a, 'b) memo -> 'a -> 'b option
val memo_add    : ('a, 'b) memo -> 'a -> 'b -> unit
val memo_fib    : (int, int) memo -> int -> int

let fib = memo_fib (create_memo ())
```

Exercise 7.6 One way to represent a directed graph is with an adjacency list stored directly in each vertex. Each vertex has a label and a list of out-edges; we also include a "mark" flag and an integer to be used by a depth-first-search.

```
type 'a vertex =
   (* Vertex (label, out-edges, dfs-mark, dfs-index) *)
   Vertex of 'a * 'a vertex list ref * bool ref * int option ref

type 'a directed_graph = 'a vertex list
```

Depth-first search and breadth-first search are two highly useful graph algorithms. A depth-first search (DFS) traverses the graph, assigning to each vertex a DFS index and marking a vertex v when all out-edges of v have been explored. The DFS search is performed as follows.

Choose an unmarked vertex u in the graph, push out-edges (u, v) onto a stack. Assign u DFS index 0, set the DFS counter c to 1, and then repeat the following until the stack is empty.

1. Pop an edge (u, v) from the stack, and classify it according to the following table.

Condition	Edge type for (u, v)
v does not have a DFS index	tree edge
$DFS(u) < DFS(v)$	forward edge
$DFS(u) > DFS(v)$ and v not marked	back edge
$DFS(u) > DFS(v)$ and v marked	cross edge

2. If (u, v) is a tree edge, assign v the current DFS index c, increment c, and push all edges (v, w) onto the stack.

3. When all edges (u, v) have been considered, mark the vertex u.

Repeat until all vertices have been marked. A graph is *cyclic* iff the DFS search found any back-edges.

Implement a DFS search. You can assume that all vertices are initially unmarked and their DFS index is `None`.

Exercise 7.7 One issue with graph data structures is that some familiar operations are hard to implement. For example, consider the following representation (similar to the previous exercise) for a directed graph.

```
(* Vertex (label, out-edges) *)
type 'a vertex = Vertex of 'a * 'a vertex list ref
type 'a directed_graph = 'a vertex list
```

Suppose we want to define a polymorphic map function on graphs.

```
val graph_map : ('a -> 'b) -> 'a directed_graph -> 'b directed_graph
```

Given an arbitrary function `f : 'a -> 'b` and a graph `g`, the expression `graph_map f g` should produce a graph isomorphic to `g`, but where `f` has been applied to each label. Is the function `graph_map` definable? If so, describe the implementation. If not, explain why not. Is there another implementation of graphs where `graph_map` can be implemented efficiently?

Chapter 8

Records, Arrays, and String

Records, arrays, and strings are aggregate data types. A record is like a tuple where the elements of the tuple are named. Arrays and strings are fixed-length sequences of data supporting random access to the elements in constant time. All three types support mutation of elements, by assignment.

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. Fields 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 record 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  = "626-555-1212";
       salary = 50.0
     };;
val jason : db_entry =
  {name="Jason"; height=6.25;
   phone="626-555-1212"; salary=50}
```

There are two ways to access 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 = "626-555-1212"
```

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`. Not all of the fields have to be included. Note that the binding occurrences of the variables n and h occur to the *right* of the equality symbol in their fields.

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

8.1.1 Functional and imperative record updates

There is a functional update operation that produces a copy of a record with new values for the specified fields. The syntax for 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="626-555-1212"; salary=50}
# jason;;
- : db_entry = {name="Jason"; height=6.25;
                phone="626-555-1212"; salary=50}
```

Record fields can also be modified by assignment, but only if the record field is declared as mutable in the type definition for the record. 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 need to declare the field `salary` as `mutable`.

```
# type db_entry =
    { name : string;
      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 = "626-555-1212";
      salary = 50.0
    };;
val jason : db_entry =
  {name="Jason"; height=6.25; phone="626-555-1212"; 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="626-555-1212"; salary=150}
```

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

```
# let dave = { jason with name = "Dave" };;
val dave : db_entry =
  {name="Dave"; height=6.25; phone="626-555-1212"; salary=150}
# dave.salary <- 180.0;;
- : unit = ()
# dave;;
- : db_entry = {name="Dave"; height=6.25; phone="626-555-1212"; salary=180}
# jason;;
- : db_entry = {name="Jason"; height=6.25; phone="626-555-1212"; salary=150}
```

8.1.2 Field label namespace

One important point: the namespace for toplevel record field labels is flat. 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 = "626-555-1212" };;
val dave : rec2 = {name="Dave"; phone="626-555-1212"}
# 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 in the type definition for `rec2`. 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 12.

8.2 Arrays

Arrays are fixed-size vectors of values. All of the values must have the same type. The fields in the array can be accessed and modified in constant time. Arrays can be created with the syntax `[|` e_1`;` e_2`;` $\cdots$`;` e_n `|]`, which creates an array of length n initialized with the values computed from the expressions $e_1, \ldots, e_n$.

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

Fields can be accessed with the `a.(i)` construction. Array indexes start from 0, and runtime checking is used to ensure that indexes are within the array bounds.

```
# 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 destructively 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|]
```

8.3 Strings

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`.

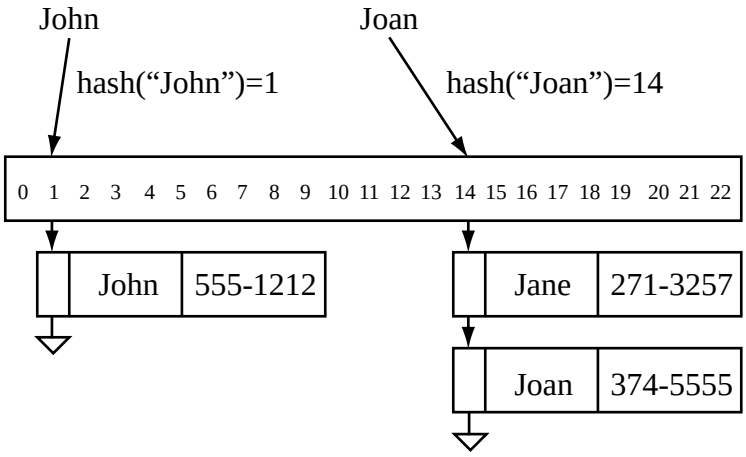


Figure 8.1: An open hash table.

```
# 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\027x\000\000"
# String.create 10;;
- : string = "\196\181\027x\001\000\000\000\000\000"
```

8.4 Hash tables

To illustrate these types in action, let's implement (yet another) dictionary, this time in the form of a hash table. A *hash table* provides the usual map from keys to values, but this time the expected running time for lookup and insertion is constant. The hash table works by computing an integer *hash* of a key that serves as an index into an array of dictionary entries. Insertion is performed by adding a new entry to the table at the hash index of the key; lookup is performed by searching for an entry with a matching key at the key's hash index. An example of a hash table is shown in Figure 8.1.

In the usual case, the space of indices is smaller than the space of keys, so hash

collisions may occur where two keys hash to the same index. Hash collisions can have a significant impact on performance. The hash table in the figure shows a so-called "chained" implementation, where entries with the same hash are stored in a list associated with that index.

For our example, we'll implement a simple hash table where the keys are strings, and the table is polymorphic over the type of values. One approach to producing a fast, fairly good hash is called a *s-box* (for *substitution box*), which uses a table of randomly-generated numbers.

```
1  let random_numbers =
2     [|0x04a018c6; 0x5ba7b0f2; 0x04dcf08b; 0x1e5a22cc; 0x2523b9ea; ···|]
3  let random_length = Array.length random_numbers
4
5  type hash_info = { mutable hash_index : int; mutable hash_value : int }
6
7  let hash_char info c =
8     let i = Char.code c in
9     let index = (info.hash_index + i + 1) mod random_length in
10    info.hash_value <- (info.hash_value * 3) lxor random_numbers.(index);
11    info.hash_index <- index
```

The record type `hash_info` has two fields: the `hash_index` is an index into the random number array, and `hash_value` is the partially computed hash. The function `hash_char` uses the character to update the `hash_index` and updates the `hash_value` by taking the exclusive-or with a random integer. The hash of a string is computed one character at a time.

```
12 let hash s =
13    let info = { hash_index = 0; hash_value = 0 } in
14    for i = 0 to String.length s - 1 do
15       hash_char info s.[i]
16    done;
17    info.hash_value
```

Note that the bounds in the for-loop on line 3 are *inclusive*; the index of the first character in the string is `0`, and the final character has index `String.length s - 1`.

The hash table itself is an array of key/value pair lists (called *buckets*), as shown in the following code.

```
18 type 'a hash_entry = { key : string; value : 'a }
19 type 'a hash_table = 'a hash_entry list array
20
21 (* create : unit -> 'a hash_table *)
22 let create () =
23    Array.create 101 []
24
25 (* add : 'a hash_table -> string -> 'a -> unit *)
26 let add table key value =
27    let index = (hash key) mod (Array.length table) in
28    table.(index) <- { key = key; value = value } :: table.(index)
29
30 (* find : 'a hash_table -> string -> 'a *)
31 let rec find_entry key = function
32    { key = key'; value = value } :: _ when key' = key -> value
```

```
33    | _ :: entries -> find_entry key entries
34    | [] -> raise Not_found
35
36  let find table key =
37      let index = (hash key) mod (Array.length table) in
38      find_entry key table.(index)
```

The function `add : 'a hash_table -> string -> 'a -> unit` adds a new entry to the table by adding the key/value pair to the table at the hash index for the key. The function `find : 'a hash_table -> string -> 'a` searches the table for the entry containing the key.

8.5 Exercises

Exercise 8.1 Reference cells are a special case of records, with the following type definition.

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

Implement the operations on reference cells.

```
val ref  : 'a -> 'a ref
val (!)  : 'a ref -> 'a
val (:=) : 'a ref -> 'a -> unit
```

Exercise 8.2 Consider the following record type definition.

```
type ('a, 'b) mpair = { mutable fst : 'a; snd : 'b }
```

What are the types of the following expressions?

1. `[|[]|]`
2. `{ fst = []; snd = [] }`
3. `{ { fst = (); snd = 2 } with fst = 1 }`

Exercise 8.3 Records can be used to implement abstract data structures, where the data structure is viewed as a record of functions, and the data representation is hidden. For example, a type definition for a functional dictionary is as follows.

```
type ('key, 'value) dictionary =
   { insert : 'key -> 'value -> ('key, 'value) dictionary;
     find   : 'key -> 'value
   }

val empty : ('key, 'value) dictionary
```

Implement the empty dictionary `empty`. Your implementation should be pure, without side-effects. You are free to use any internal representation of the dictionary.

Exercise 8.4 Records can also be used to implement a simple form of object-oriented programming. Suppose we are implementing a collection of geometric objects (blobs), where each blob has a position, a function (called a *method*) to compute the area covered by the blob, and methods to set the position and move the blob. The following record defines the methods for a generic object.

```
type blob =
   { get    : unit -> float * float;
     area   : unit -> float;
     set    : float * float -> unit;
     move   : float * float -> unit
   }
```

An actual object like a rectangle might be defined as follows.

```
let new_rectangle x y w h =
   let pos = ref (x, y) in
   let rec r =
      { get  = (fun () -> !pos);
        area = (fun () -> w *. h);
        set  = (fun loc -> pos := loc);
        move = (fun (dx, dy) ->
                   let (x, y) = r.get () in
                   r.set (x +. dx, y +. dy))
      }
   in
   r
```

The rectangle record is defined recursively so that the method `move` can be defined in terms of `get` and `set`.

Suppose we have created a new rectangle `rect1`, manipulated it, and now we want to fix it in position. We might try to do this by redefining the method `set`.

```
let rect1 = new_rectangle 0.0 0.0 1.0 1.0 in
rect1.move 1.2 3.4; ···
let rect2 = { rect1 with set = (fun _ -> ()) }
```

1. What happens to `rect2` when `rect2.move` is called? How can you prevent it from moving?
2. What happens to `rect2` when `rect1.set` is called?

Exercise 8.5 Write a function `string_reverse : string -> unit` to reverse a string in-place.

Exercise 8.6 What problem might arise with the following implementation of an array blit function? How can it be fixed?

```
let blit src src_off dst dst_off len =
   for i = 0 to len - 1 do
      dst.(dst_off + i) <- src.(src_off + i)
   done
```

Exercise 8.7 *Insertion sort* is a sorting algorithm that works by inserting elements one-by-one into an array of sorted elements. Although the algorithm takes $O(n^2)$ time to sort an array of n elements, it is simple, and it is also efficient when the array to be sorted is small. The pseudo-code is as follows.

```
insert(array a, int i)
    x <- a[i]
    j <- i - 1
    while j >= 0 and a[j] > x
        a[j] <- a[j - 1]
        j = j - 1
    a[j + 1] <- x

insertion_sort(array a)
    i <- 1
```

```
while i < length(a)
    insert(a, i)
    i <- i + 1
```

Write this program in OCaml.

Chapter 9

Exceptions

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

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

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

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

The first line of this program defines a new exception, declaring `Fail` as an exception with a string argument. The `head` function uses pattern matching—the result is `h` if the list has first element `h`; otherwise, there is no first element, and the `head` function raises a `Fail` exception. The expression `(Fail "head: the list is empty")` is a value of type exn; the `raise` function is responsible for aborting the current computation.

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

```
Exception: Fail "message".
```

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

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

When an exception is raised, the current computation is aborted, and control is passed directly to the currently active exception handler, which in this case is the toploop itself.

It is also possible to define explicit exception handlers. Exception handlers have the same form as a match pattern match, but using the try keyword instead. The syntax is as follows.

```
try expression_t with
 | pattern_1 -> expression_1
 | pattern_2 -> expression_2
 ⋮
 | pattern_n -> expression_n
```

First, $expression_t$ is evaluated. If it does not raise an exception, its 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 $pattern_1, \ldots, pattern_n$. If the first pattern to match the exception is $pattern_i$, the expression $expression_i$ is evaluated and returned as the result of the entire try expression. Unlike a match expression, there is no requirement that the pattern matching be complete. If no pattern matches, the exception is not caught, and it is propagated to the next exception handler (which may be the toploop).

For example, suppose we wish to define a function head_default, similar to head, but returning a default value if the list is empty. One way would be to write a new function from scratch, but we can also choose to handle the exception from head.

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

In this case, if evaluation of head l raises an exception Fail, the value default is returned.

9.1 Nested exception handlers

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

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

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

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

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

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

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

```
# map (fun i ->
          if i = 0 then
             raise Empty
```

```
          else
            i + 1) [3; 5; 0; 7; 0; 9];;
- : int list = [4; 6]
```

9.2 Examples of uses of exceptions

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

9.2.1 The exception Not_found

The OCaml standard library uses exceptions for several different purposes. One of the most common exceptions is `Not_found`, which is raised by functions that perform searching or lookup. There are many such functions in OCaml. One is the function `List.assoc`, which searches for a key-value pair in an association. For example, suppose we were implementing a grading program where the grades are represented as a list of name/grade pairs.

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

In typical programs, `Not_found` exceptions routinely occur and can be expected to happen during normal program operation.

9.2.2 Invalid_argument and Failure

An `Invalid_argument` exception means that some kind of runtime error occurred, like an array bounds violation. The string argument describes the error.

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

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

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

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

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

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

```
let rec find_grade name (names, grades) =
   match names, grades with
      name' :: _, grade :: _ when name' = name -> grade
    | _ :: names', _ :: grades' -> find_grade name (names', grades')
    | [], [] -> raise (Failure ("student is not enrolled: " ^ name))
    | (_ :: _), [] | [], (_ :: _) -> raise (Invalid_argument "corrupted database")
```

The function `find_grade` searches the lists, returning the first match if there is one. If the lists have different lengths, an `Invalid_argument` exception is raised because, 1) the implementation assumes that the lists have the same length, so the error violates a program invariant, and 2) there is no easy way to recover. The pattern `[], []` corresponds to the case where the student is not found, but the lists have the same length. This is expected to occur during normal operation, so the appropriate exception is `Failure` (or `Not_found` since this is a search function).

As a matter of style, it's considered bad practice to catch `Invalid_argument` exceptions (in fact, some early OCaml implementations did not even allow it). In contrast, `Failure` exceptions are routinely caught in order to recover from correctable errors.

9.2.3 Pattern matching failure

When a pattern matching is incompletely specified, the OCaml compiler issues a warning (and a suggestion for the missing pattern). At runtime, if the matching fails because it is incomplete, the `Match_failure` exception is raised with three values: the name of the file, the line number, and the character offset within the line where the match failed. It is often considered bad practice to catch the `Match_failure` exception because the failure indicates a programming error (proper programming practice would dictate that all pattern matches be complete).

```
# let f x =
     match x with
        Some y -> y;;
```

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

9.2.4 Assertions

Another common use of exceptions is for checking runtime invariants. The `assert` operator evaluates a Boolean expression, raising an `Assert_failure` exception if the value is `false`. For example, in the following version of the factorial function, an assertion is used to generate a runtime error if the function is called with a negative argument. The three arguments represent the file, line, and character offset of the failed assertion. As with `Invalid_argument` and `Match_failure`, it is considered bad programming practice to catch the `Assert_failure` exception.

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

9.2.5 Memory exhaustion exceptions

The two exceptions `Out_of_memory` and `Stack_overflow` indicate that memory resources have been exhausted. The `Out_of_memory` exception is raised by the garbage collector when there is insufficient memory to continue running the program. The `Stack_overflow` exception is similar, but it is restricted to just stack space. The `Stack_overflow` exception is often caused by an infinite loop, or excessively deep recursion, for example, using the function `List.map` on a list with more than a few thousand elements.

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

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

9.3 Other uses of exceptions

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

9.3.1 Decreasing memory usage

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

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

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

One problem with this function is that the entire list is copied needlessly when the element is not found, potentially increasing the space needed to run the program. Exceptions provide a convenient way around this problem. By raising an exception in the case where the element is not found, we can avoid reconstructing the entire list. In the following function, when the `Unchanged` exception is raised, the `remove` function returns the original list `l`.

```
exception Unchanged

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

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

9.3.2 Break statements

OCaml provides both "for" and "while" loops, but there is no "break" statement as found in languages like C and Java. Instead, exceptions can be used to abort a loop prematurely. To illustrate this, suppose we want to define a function `cat` that prints out all the lines from the standard input channel. We discuss input/output in more detail in Section 10, but for this problem we can just use the standard functions `input_char` to read a character from the input channel, and `output_char` to write it to the output channel. The `input_char` function raises the exception `End_of_file` when the end of the input has been reached.

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

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

9.3.3 Unwind-protect (finally)

In some cases where state is used, it is useful to define a "finally" clause (similar to an "unwind-protect" as seen in Lisp languages). The purpose of a "finally" clause is to execute some code (usually to clean up) after an expression is evaluated. In addition, the finally clause should be executed even if an exception is raised. A generic `finally` function can be defined using a wildcard exception handler. In the following function, the type `result` is used to represent the result of executing the function `f` on argument `x`, returning a `Success` value if the evaluation was successful, and `Failed` otherwise. Once the result is computed, the `cleanup` function is called, and 1) the result is returned on `Success`, or 2) the exception is re-raised on `Failed`.

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

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

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

```
let process in_channel = ···

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

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

successful.

9.3.4 The exn type

We close with a somewhat unorthodox use of exceptions completely unrelated to control flow. Exceptions (values of the exn type) are first-class values; they can be passed as arguments, stored in data structures, *etc*. The values in the exn type are specified with `exception` definitions. One unique property of the exn type is that it is *open* so that new exceptions can be declared when desired. This mechanism can be used to provide a kind of dynamic typing, somewhat like the polymorphic variants discussed in Section 6.5.

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

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

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

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

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

9.4 Exercises

Exercise 9.1 Which of the following are legal expressions?

1. `exception A`
2. `exception b`
3. `exception C of string`
4. `exception D of exn`
5. `exception E of exn let x = E (E (E Not_found))`
6. `let f () = exception F raise F`

Exercise 9.2 What is the result of evaluating the following programs?

1. ```
   exception A
   try raise A with
      A -> 1
   ```
2. ```
   exception A of int
   let f i =
      raise (A (100 / i));;
   let g i =
      try f i with
         A j -> j;;
   g 100
   ```
3. ```
   exception A of int
   let rec f i =
      if i = 0 then
         raise (A i)
      else
         g (i - 1)
   and g i =
      try f i with
         A i -> i + 1;;
   g 2
   ```

Exercise 9.3 In the following program, the function `sum_entries` sums up the integer values associated with each name in the list `names`. The `List.assoc` function finds the value associated with the name, raising the `Not_found` exception if the entry is not found. For example, the expression `sum_entries 0 ["a"; "c"]` would evaluate to `35`, and the expression `sum_entries 0 ["a"; "d"]` would raise the `Not_found` exception.

```
let table = [("a", 10); ("b", 20); ("c", 25)]
let rec sum_entries total (names : string list) =
   match names with
      name :: names' ->
         sum_entries (total + List.assoc name table) names'
    | [] ->
         total
```
```

Suppose we wish to catch the exception, arbitrarily assigning a value of 0 to each unknown entry. What is the difference between the following two functions? Which form is preferable?

1.
```
let table = [("a", 10); ("b", 20); ("c", 25)]
let rec sum_entries total (names : string list) =
   match names with
      name :: names' ->
         (try sum_entries (total + List.assoc name table) names' with
             Not_found ->
                sum_entries total names')
    | [] ->
         total
```

2.
```
let table = [("a", 10); ("b", 20); ("c", 25)]
let rec sum_entries total (names : string list) =
   match names with
      name :: names' ->
         let i =
            try List.assoc name table with
               Not_found  ->
                  1
         in
            sum_entries (total + i) names'
    | [] ->
         total
```

Exercise 9.4 Suppose we are given a `table` as in the last exercise, and we wish to call some function `f` on one of the entries, or returning 0 if the entry is not found. That is, we are given the function `f`, and a name, and we wish to evaluate `f (List.assoc table name)`. What is the difference between the following functions?

1.
```
let callf f name =
   try f (List.assoc table name) with
      Not_found ->
         0
```

2.
```
let callf f name =
   let i =
      try Some (List.assoc table name) with
         Not_found ->
            None
   in
      match i with
         Some j -> f j
       | None -> 0
```

Exercise 9.5 The expression `input_line stdin` reads a line of text from standard input, returning the line as a string, or raising the exception `End_of_file` if the end of the file has been reached. Write a function `input_lines` to read all the lines from the

channel `stdin`, returning a list of all the lines. The order of the lines in the list does not matter.

Chapter 10

Input and Output

The I/O library in OCaml is fairly expressive, including a library (the `Unix` library) that implements a set of system calls that are portable across the platforms on which OCaml runs. In this chapter, we'll cover many of the standard built-in I/O functions.

The I/O library starts by defining two data types: the type `in_channel` specifies an I/O channel from which characters can be read, and the type `out_channel` specifies 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; `stdin` is the standard input stream, `stdout` is the standard output stream, and `stderr` is the standard output stream for error messages.

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

10.1 File opening and closing

There are two functions to open an output file: the function `open_out` opens a file for writing text data, and the function `open_out_bin` opens a file for writing binary data. These two functions are identical on a Unix system. On some Macintosh and Microsoft Windows systems, the `open_out` function performs line termination translation, while the `open_out_bin` function writes the data exactly as written. These functions raise the exception `Sys_error` if the file can't be opened; otherwise they return an `out_channel`.

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

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

The open_out_gen and open_in_gen functions can be used to perform more kinds of file opening. The function requires an argument of type open_flag list that describes how to open the file. The flags mimic the oflags bitmask given to the int open(const char *path, int oflags, ...) system call in Unix; on other platforms the behavior is similar.

```
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 functions open_in_gen and open_out_gen have the following types.

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

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

Channels are not closed automatically. The closing operations close_out and close_in are used for explicitly closing the channels.

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

10.2 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 fail unpredictably.

```
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
```

10.3 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 known 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. The characters on an `out_channel` may not be completely written until the channel is closed. To force the writing on the buffer, use the `flush` function.

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

10.4 String buffers

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

```
val create : unit -> Buffer.t
```

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

```
val contents : Buffer.t -> string
val length : Buffer.t -> int
val clear : Buffer.t -> unit
val reset : Buffer.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 from a channel and appends it to the buffer.

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

For example, the following code sequence produces the string `"Hello world!\n"`

```
# let buf = Buffer.create 20;;
val buf : Buffer.t = <abstr>
# Buffer.add_string buf "Hello";;
# Buffer.add_char buf ' ';;
# Buffer.add_string buf "world!\n";;
# Buffer.contents buf;;
- : string = "Hello world!\n"
```

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

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

10.5 Formatted output with Printf

The regular functions for I/O are fairly low-level, and they can be awkward to use. OCaml also implements a `printf` function similar to the `printf` in the standard library for the C programming language. These functions are defined in the library module `Printf`. The general form is given by the function `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 Printf format string. For example, the following statement prints a line containing an integer `i` and a string `s`.

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

The strange typing of the `fprintf` function is because the OCaml `fprintf` function is type-safe. The OCaml compiler analyzes the the format string to determine the type of the arguments. For example, the following format string specifies that the `fprintf` function takes a `float`, `int`, and `string` argument.

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

The OCaml format specification is similar to format specifications in ANSI C. Normal characters (not `%`) are copied verbatim from the input to the output. Conversions are introduced by the character `%`, which is followed in sequence by optional width and length specifiers, and a conversion specifier. The conversion specifiers include the following.

- `d` or `i`: print an integer argument as a signed decimal value.
- `u`: print an integer argument as an unsigned decimal value.
- `o`: print an integer argument as an unsigned octal value.
- `x`: print an integer argument as an unsigned hexadecimal value, using lowercase letters.
- `X`: print an integer argument as an unsigned hexadecimal value, using uppercase letters.
- `s`: print a string argument.
- `c`: print a character argument.
- `f`: print a floating-point argument using decimal notation, in the style *dddd.ddd*.
- `e` or `E`: print a floating-point argument using decimal notation, in the style *d.ddd e+-dd* (mantissa and exponent).
- `g` or `G`: print a floating-point argument using decimal notation, in the style *f*, *e*, or *E*, whichever is more compact.

- `b`: print a Boolean argument as the string `true` or `false`.
- `a`: The argument should be a user-defined printer that takes two arguments, applying 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 inserted into the output of `fprintf` at the current point.
- `t`: This is mostly the 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 flushes the output channel.
- `%`: takes no argument and outputs one `%` character.

There may be more format conversions in your version of OCaml; see the reference manual for additional cases.

Most format specifications accept width and precision specifiers with the following syntax, where the square brackets indicate that the field is optional.

```
% [-] [width] [.precision] specifier
```

If specified, the *width* indicates the minimum number of characters to output. If the format contains a leading minus sign `-`, the output is left-justified; otherwise it is right-justified. For numeric arguments, if the width specifier begins with a zero, the output is padded to fit with width by adding leading zeros. The *precision* is used for floating-points values to specify how many fractional digits to print after the decimal point. Here are some examples.

```
# open Printf;;
# printf "///%8.3f///" 3.1415926;;
///   3.142///
# printf "///%-8.3f///" 3.1415926;;
///3.142   ///
# printf "///%8s///" "abc";;
///     abc///
# printf "///%8s///" "abcdefghijk";;
///abcdefghijk///
# printf "///%x///" 65534;;
///fffe///
# printf "///0x%08x///\n" 65534;;
///0x0000fffe///

# printf "///%a///" (fun buf (x, y) ->
     fprintf buf "x = %d, y = %g" x y) (17, 231.7);;
///x = 17, y = 231.7///
# printf "x = %d, y = %g" 17 231.7e35;;
x = 17, y = 2.317e+37
```

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

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
```

10.6 Formatted input with Scanf

The Scanf module is similar to Printf, but for input instead of output. The types are as follows.

```
val fscanf : in_channel -> ('a, Scanning.scanbuf, 'b) format -> 'a -> 'b
val sscanf : string -> ('a, Scanning.scanbuf, 'b) format -> 'a -> 'b
val scanf  : ('a, Scanning.scanbuf, 'b) format -> 'a -> 'b
```

The fscanf function reads from an input channel; the sscanf function reads from a string; and the scanf function reads from the standard input.

Once again, the types are somewhat cryptic. In actual use, the scanf functions take a format string and a function to process the values that are scanned. The format specifier uses a syntax similar to the printf format specification. For scanf, there are two main kinds of scanning actions.

- A plain character matches the same literal character on the input. There is one exception, a single space character matches any amount of whitespace in the input, including tabs, spaces, newlines, and carriage returns.
- A conversion specifies the format of a value in the input streams. For example, the conversion %d specifies that a decimal integer is to be read from the input channel.

Here are some examples.

```
# open Scanf;;
# sscanf "ABC   345" "%s %d" (fun s i -> s, i);;
- : string * int = ("ABC", 345)
# sscanf "ABC 345" "%s%d" (fun s i -> s, i);;
Exception: Scanf.Scan_failure "scanf: bad input at char number 4:  ".
# sscanf "ABC 345" "%4s %d" (fun s i -> s, i);;
- : string * int = ("ABC", 345)
# sscanf "ABC DEF 345" "%s %_s %f" (fun s x -> s, x);;
- : string * float = ("ABC", 345.)
# sscanf "123456" "%3d%3d" (fun i1 i2 -> i1 + i2);;
- : int = 579
# sscanf "0x123 -0b111" "%i %i" (fun i1 i2 -> i1, i2);;
- : int * int = (291, -7)
```

10.7 Exercises

Exercise 10.1 Write a "Hello world" program, which prints the line `Hello world!` to the standard output.

Exercise 10.2 The input functions raise the `End_of_file` exception when the end of file is reached, which dictates a style where input functions are always enclosed in exception handlers. The following function is not tail-recursive (see Section 5.4), which means the stack may overflow if the file is big.

```
let read_lines chan =
   let rec loop lines =
      try loop (input_line chan :: lines) with
         End_of_file -> List.rev lines
   in
   loop []
```

1. Why isn't the function `read_lines` tail-recursive?
2. How can it be fixed?

Exercise 10.3 Exceptions can have adverse interactions with input/output. In particular, unexpected exceptions may lead to situations where files are not closed. This isn't just bad style, on systems where the number of open files is limited, this may lead to program failure. Write a function `with_in_file : string -> (in_channel -> 'a) -> 'a` to handle this problem. When the expression `with_in_file filename f` is evaluated, the file with the given `filename` should be opened, and the function `f` called with the resulting `in_channel`. The channel should be closed when `f` completes, even if it raises an exception.

Exercise 10.4 You are given two files `a.txt` and `b.txt`, each containing a single character. Write a function `exchange` to exchange the values in the two files. Your function should be robust to errors (for example, if one of the files doesn't exist, or can't be opened).

Is it possible to make the `exchange` operation atomic? That is, if the operation is successful the contents are exchanged, but if the operation is unsuccessful the files are left unchanged?

Exercise 10.5 Suppose you are given a value of the following type, and you want to produce a string representation of the value.

```
type exp =
   Int of int
 | Id of string
 | List of exp list
```

The representation is as follows.

- `Int` and `Id` values print as themselves.

- `List` values are enclosed in parentheses, and the elements in the list are separated by a single space character.

Write a function `print_exp` to produce the string representation for a value of type `exp`. The following gives an example.

```
# print_exp (List [Int 2; Id "foo"]);;
(2 foo)
```

Exercise 10.6 You are given an input file `data.txt` containing lines that begin with a single digit `1` or `2`. Write a function using the `Buffer` module to print the file, without leading digits, in de-interleaved form.

`data.txt`	⟶	output
`2Is`		`This`
`1This`		`Is`
`2File`		`A`
`1A`		`File`

For example, given the input on the left, your program should produce the output on the right.

Exercise 10.7 Suppose you are given three values `(x, y, z) : string * int * string`. Using `printf`, print a single line in the following format.

- The string `x` should be printed left-justified, with a minimum column width of 5 characters.
- The integer `y` should be printed in hex with the prefix `0x`, followed by 8 hexadecimal digits, followed by a single space.
- The third word should be printed right-justified, with a minimum column width of 3 characters.
- The line should be terminated with a newline `\n`.

Exercise 10.8 Suppose you are given a list of pairs of strings (of type `(string * string) list`. Write a program to print out the pairs, separated by white space, in justified columns, where the width of the first column is equal to the width of the longest string in the column. For example, given the input `[("a", "b"); ("ab", "cdef")]` the width of the first column would be 2. Can you use `printf` to perform the formatting?

```
# print_cols ["a", "b"; "abc", "def"];;
a   b
abc def
```

Exercise 10.9 Consider the following program. The exception `Scan_failure` is raised when the input cannot be scanned because it doesn't match the format specification.

```
try scanf "A%s" (fun s -> s) with
   Scan_failure _ ->
      scanf "B%s" (fun s -> s)
```

What is the behavior of the this program when presented with the following input?

1. `AA\n`
2. `B\n`
3. `AB\n`
4. `C\n`
5. `ABC\n`

Chapter 11

Files, Compilation Units, and Programs

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

11.1 Single-file programs

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

One of the simplest implementations is to use a list to keep track of which lines have been read. The program can be implemented as a single recursive function that 1) reads a line of input, 2) compares it with lines that have been previously read, and 3) outputs the line if it has not been read. The entire program is implemented in the single file `unique.ml`, shown in Figure 11.1 with an example run.

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

11.1.1 Where is the main function?

Unlike C programs, OCaml program do not have a "`main`" function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated in or-

File: unique.ml

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

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

Example run

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

Figure 11.1: A program to print only unique lines.

der. In general, implementation files can contain arbitrary expressions, not just function definitions. For this example, the main program is the essentially `try` expression in the `unique.ml` file, which gets evaluated when the `unique.cmo` file is evaluated. We say "essentially" because the main function is really the entire program, which is evaluated starting from the beginning when the program is executed.

11.1.2 OCaml compilers

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

- Compile time is shorter with the `ocamlc` compiler. Compiled byte-code is portable to any operating system and architecture supported by OCaml, without the need to recompile. Some tasks, like debugging, work only with byte-code executables.
- Compile time is longer with the `ocamlopt` compiler, but program execution is usually faster. Program executables are not portable, and `ocamlopt` is supported on fewer operating systems and machine architectures than `ocamlc`.

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

11.2 Multiple files and abstraction

OCaml uses files as a basic unit for providing data hiding and encapsulation, two important properties that can be used to strengthen the guarantees provided by the implementation. We will see more about data hiding and encapsulation in Chapter 12, but for now the important part is that each file can be assigned a *interface* that declares types for all the accessible parts of the implementation, and everything not declared is inaccessible outside the file.

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

1. Decide how to divide the program into separate files. Each part will be implemented in a separate compilation unit.

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

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

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

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

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

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

The new project is shown in Figure 11.2. We have split the set operations into a file called `set.ml`, and instead of using the `List.mem` function we now use the `Set.mem` function. The way to refer to a definition f in a file named *filename* is by capitalizing the filename and using the infix `.` operator to project the value. The `Set.mem` expression refers to the `mem` function in the `set.ml` file. In fact, the `List.mem` function is the same way. The OCaml standard library contains a file `list.ml` that defines a function `mem`.

Compilation now takes several steps. In the first step, the `set.ml` and `unique.ml` files are compiled with the `-c` option, which specifies that the compiler should produce an intermediate file with a `.cmo` suffix. These files are then linked to produce an executable with the command `ocamlc -o unique set.cmo unique.cmo`.

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

11.2.1 Defining an interface

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

File: set.ml

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

File: unique.ml

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

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

Example run

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

Figure 11.2: Factoring the program into two separate files.

File: set.mli

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

File: set.ml

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

Example run (with lists)

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

Figure 11.3: Adding an interface to the `Set` implementation.

An interface should declare types for each of the values that are publicly accessible in a module, as well as any needed type declarations or definitions. For our purposes, we need to define a polymorphic type of sets `'a set` abstractly. That is, in the interface we will declare a type `'a set` without giving a definition, preventing other parts of the program from knowing, or depending on, the particular representation of sets we have chosen. The interface also needs to declare types for the public values `empty`, `add`, and `mem` values, as a declaration with the following syntax.

val *identifier* : *type*

The complete interface is shown in Figure 11.3. The implementation remains mostly unchanged, except that a specific, concrete type definition must be given for the type `'a set`.

Now, when we compile the program, we first compile the interface file `set.mli`, then the implementations `set.ml` and `unique.ml`. Note that, although the `set.mli` file must be compiled, it does not need to be specified during linking `ocamlc -o unique set.cmo unique.cmo`.

At this point, the `set.ml` implementation is fully abstract, making it easy to replace the implementation with a better one (for example, the implementation of sets using red-black trees in Section 6.4).

11.2.2 Transparent type definitions

In some cases, abstract type definitions are too strict. There are times when we want a type definition to be *transparent*—that is, visible outside the file. For example, suppose we wanted to add a `choose` function to the set implementation, where, given a set s, the expression (`choose` s) returns some element of the set if the set is non-empty, and nothing otherwise. One possible way to write this function is to define a union type `choice` that defines the two cases, as shown in Figure 11.4.

The type definition for `choice` must be transparent (otherwise there isn't much point in defining the function). For the type to be transparent, the interface simply

Interface file: set.mli

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

Implementation file: set.ml

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

Figure 11.4: Extending the `Set` implementation.

provides the definition. The implementation must contain the *same* definition.

11.3 Some common errors

As you develop programs with several files, you will undoubtably encounter some errors.

11.3.1 Interface errors

When an interface file (with a `.mli` suffix) is compiled successfully with `ocamlc` or `ocamlopt`, the compiler produces a compiled representation of the file, having a `.cmi` suffix. When an implementation is compiled, the compiler compares the implementation with the interface. If a definition does not match the interface, the compiler will print an error and refuse to compile the file.

Type errors

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

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

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

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

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

Missing definition errors

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

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

Type definition mismatch errors

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

Interface file: set.mli

```
type 'a set
type 'a choice =
   Element of 'a
 | Empty
...
```

Implementation file: set.ml

```
type 'a set = 'a list
type 'a choice =
   Empty
 | Element of 'a
...
```

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

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

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

Compile dependency errors

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

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

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

11.4 Using open to expose a namespace

Using the full name *`Filename.identifier`* to refer to the values in a module can get tedious. The statement `open` *`Filename`* can be used to "open" an interface, allowing the use of unqualified names for types, exceptions, and values. For example, the `unique.ml` module can be somewhat simplified by using the `open` directive for the `Set` module. In the following listing, the underlined variables refer to values from the Set implementation (the underlines are for illustration only, they don't exist in the program files).

File: unique.ml

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

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

Sometimes multiple opened files will define the same name. In this case, the *last* file with an open statement will determine the value of that symbol. Fully qualified names (of the form *`Filename.identifier`*) may still be used even if the file has been opened.

Fully qualified names can be used to access values that may have been hidden by an open statement.

11.4.1 A note about open

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

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

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

11.5 Debugging a program

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

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

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

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

```
% ocamldebug ./unique
        Objective Caml Debugger version 3.08.3
(ocd) help
List of commands: cd complete pwd directory kill help quit shell run reverse
step backstep goto finish next start previous print display source break
delete set show info frame backtrace bt up down last list load_printer
install_printer remove_printer
```

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

- `run`: Start or continue execution of the program.
- `break @ module linenum`: Set a breakpoint on line `linenum` in module `module`.
- `list`: display the lines around the current execution point.
- `print expr`: Print the value of an expression. The expression must be a variable.
- `goto time`: Execution of the program is measured in time steps, starting from 0. Each time a breakpoint is reached, the debugger prints 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 `unique` program, we need to know the line numbers. Let's set a breakpoint in the `unique` function, which starts in line 1 in the `Unique` module. We'll want to stop at the first line of the function.

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

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

```
(ocd) list
```

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

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

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

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

We can continue from here, examining the remaining functions and variables. You may wish to explore the other features of the debugger. Further documentation can be found in the OCaml reference manual.

11.6 Exercises

Exercise 11.1 Consider a file `f.ml` with the following contents.

```
type t = int
let f x = x
```

Which of the following are legal `f.mli` files?

1. `f.mli` is empty.
2. `f.mli`:

   ```
   val f : 'a -> 'a
   ```

3. `f.mli`:

   ```
   val f : ('a -> 'b) -> ('a -> 'b)
   ```

4. `f.mli`:

   ```
   val f : t -> t
   ```

5. `f.mli`:

   ```
   type t
   val f : t -> t
   ```

6. `f.mli`:

   ```
   type s = int
   val f : s -> s
   ```

Exercise 11.2 Consider the following two versions of a list reversal function.

rev.mli

```
val rev : 'a list -> 'a list
```

rev.ml (version 1)

```
let rev l =
   let rec rev_loop l1 l2 =
      match l2 with
         x :: l2 ->
            loop (x :: l1) l2
       | [] ->
            l1
   in
      rev_loop [] l
```

rev.ml (version 2)

```
let rec rev_loop l1 l2 =
   match l2 with
      x :: l2 ->
         loop (x :: l1) l2
    | [] ->
         l1

let rev l = rev_loop [] l
```

1. Is there any reason to prefer one version over the other?
2. In the second version, what would happen if we defined the `rev` function as a partial application?

```
(* let rev l = rev_loop [] l *)
let rev = rev_loop []
```

Exercise 11.3 When a program is begin developed, it is sometimes convenient to have the compiler produce a `.mli` file automatically, using the `-i` option to `ocamlc`. For example, suppose we have an implementation file `set.ml` containing the following definitions.

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

Inferring types, we obtain the following output. The output can then be edited to produce the desired `set.mli` file.

```
% ocamlc -i set.ml
type 'a set = 'a list
val empty : 'a list
val add : 'a -> 'a list -> 'a list
val mem : 'a -> 'a list -> bool
```

1. The output produced by `ocamlc -i` is not abstract—the declarations use the type `'a list`, not `'a set`. Instead of editing all the occurrences by hand, is there a way to get `ocamlc -i` to produce the right output automatically?

2. In some cases, `ocamlc -i` produces illegal output. What is the inferred interface for the following program? What is wrong with it? Can it be fixed?

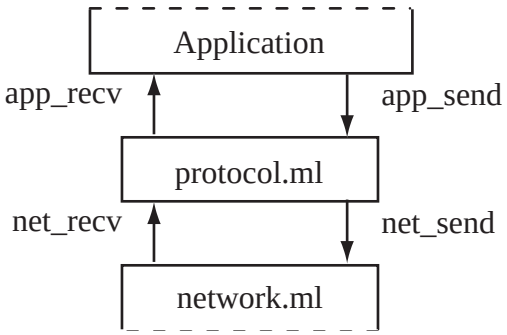
```
let cell = ref []
let push i = cell := i :: !cell
let pop () =
   match !cell with
      [] -> raise (Invalid_argument "pop")
    | i :: t ->
        cell := t;
        i
```

Exercise 11.4 One issue we discussed was the need for duplicate type definitions. If a `.mli` provides a definition for a type `t`, the the `.ml` file must specify exactly the same definition. This can be annoying if the type definition is to be changed.

One solution we discussed is to place the type definition in a separate file, like `types.ml`, with no interface file `types.mli`. It is also legal to place the type definition in a file `types.mli` with no implementation `types.ml`.

Is it ever preferable to use the second form (where `types.mli` exists, but `types.ml` doesn't)?

Exercise 11.5 The strict-ordering requirement during linking can potentially have a major effect on the software design. For example, suppose we were designing a bi-directional communication protocol, as shown in the following diagram.



With this design, the `Network` component calls `Protocol.net_recv` when a message arrives, and the `Protocol` component calls `Network.net_send` to send a message. However, this is not possible if the implementations are in separate files `protocol.ml` and `network.ml` because that would introduce a cyclic dependency.

Describe a method to circumvent this problem, without placing the code for the two components into a single file.

Chapter 12

The OCaml Module System

As we saw in the previous chapter, programs can be divided into parts that can be implemented in files, and each file can be given an interface that specifies what its public types and values are. Files are not the only way to partition a program, OCaml also provides a *module system* that allows programs to be partitioned even within a single file. There are three key parts in the module system: *signatures*, *structures*, and *functors*, where signatures correspond to interfaces, structures correspond to implementations, and functors are functions over structures. In this chapter, we will discuss the first two; we'll leave discussion of functors to Chapter 13.

There are several reasons for using the module system. Perhaps the simplest reason is that each structure has its own namespace, so name conflicts are less likely when modules are used. Another reason is that abstraction can be specified explicitly within a file by assigning a signature to a structure. Let's begin with naming.

12.1 Structures and signatures

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

module *ModuleName* = **struct** *implementation* **end**

The module name *ModuleName* must begin with an uppercase letter. The *implementation* can include any definition that might occur in a `.ml` file.

In the discussion of records (page 79), we noted that the space of record label names is flat; if two record types are defined with the same label names in the same file, the first definition is lost. Modules solve this problem by allowing the name space to be partitioned. Here is an example using records; the same principle applies to unions and constructor names.

```
module A = struct
    type t = { name : string; phone : string }
end
module B = struct
```

File: unique.ml

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

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

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

Figure 12.1: Gathering the `Set` implementation into a module.

```
    type t = { name : string; salary : float }
end
# let jason = { A.name = "Jason"; A.phone = "626-555-1212" };;
val jason : A.t = {A.name = "Jason"; A.phone = "626-555-1212"}
# let bob = { B.name = "Bob"; B.salary = 180.0 };;
val bob : B.t = {B.name = "Bob"; B.salary = 180.}
```

A simple *pathname*, or *fully-qualified* name, has the syntax *ModuleName.identifier*, where the *identifier* is the name of a module component (a type, value, record label, a nested module, etc.), and the *ModuleName* is the module containing the component. In the example, the pathname `B.salary` is the fully-qualified name of a field in the `B.t` record type (which is also a fully-qualified name).

Let's return to the `unique.ml` example from the previous chapter, using a simple list-based implementation of sets. This time, instead of defining the set data structure in a separate file, let's define it as a module, called `Set`, using an explicit module definition. The program is shown in Figure 12.1. The file is compiled and executed using the usual methods.

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

Signature definition

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

Structure definition

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

Figure 12.2: Defining an explicit signature for the `Set` module.

Moby Dick

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

In many ways, structures are like files. If we wish to hide the `Set` implementation, we can specify an explicit signature to hide parts of the implementation. A named signature is defined with a `module type` definition.

module type *ModuleName* = **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. For our example, the signature should include an abstract type declaration for the `'a set` type and `val` declarations for each of the values. The `Set` module's signature is constrained by specifying the signature after a colon in the module definition `module Set : SetSig = struct ··· end`, as shown in Figure 12.2.

12.2 Module definitions

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

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

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

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

12.2.1 Modules are not first-class

Modules/structures/signatures are not *first-class*, meaning in particular that they are not expressions. Modules cannot be passed as arguments to a function nor can they be returned as results. Normally, modules are defined as top-level outermost components in a program, hence all type and exception definitions are also top-level.

There are several reasons why modules are not first-class. Perhaps the most important is that modules and module types are complicated expressions involving types and type abstraction; if modules could be passed as values, type inference would become undecidable. Another reason is called the *phase-distinction*. From the point of view of the compiler, a program has two phases in its lifetime: compile-time and run-time. The objective of the phase distinction is to ensure that module expressions can be computed at compile-time. If modules were expressions, the phase-distinction would be difficult to ensure. In any case, the phase distinction is simply a guideline for the language designers. The effect on programmers is that modules can only be defined in a few chosen locations, but there is no performance penalty for using them.

12.2.2 The `let module` expression

One exception to top-level module definitions is the `let module` expression, which has the following syntax.

`let module` *ModuleName* `=` *module_expression* **`in`** *body_expression*

The *module_expression* is the definition of a module (often a `struct` $\cdots$ `end` block), and the module is defined in the *body_expression*. The `let module` expression is frequently used for renaming modules locally.

```
module ModuleWithALongName
...
let f x =
   let module M = ModuleWithALongName in
   ...
```

Similarly, it can be useful sometimes to redefine an existing module locally. In the following example, a `String` module is defined so that pathnames `String.`*`identifier`* in the *`function body`* refer to the locally-defined module, not the standard library (in this case, it is for debugging purposes).

```
let f x =
   let module String = struct
      let create n =
          eprintf "Allocating a string of length %d\n%!" n;
          String.create n
      ...
   end in
   function body
```

One other use of `let module` expression is to allow types and exceptions to be defined locally, not only at the top-level. For example, in the following program, the exception `Error` is local. It is guaranteed to be different from every other exception in the program.

```
let f x =
   let module M = struct exception Abort end in
   let g y =
      ...
      if done then raise M.Abort
   in
   try map g x with
      M.Abort message -> ...
```

In this particular case, the local definition of the `M.Abort` exception means that the `map` function (not shown) is not able to catch the exception except generically, by catching all exceptions.

12.3 Recursive modules

It is also possible to define modules that are defined recursively. That is, several modules that refer to one another are defined at once. The syntax for a recursive definition uses the `module rec` form.

```
module rec Name_1 : Signature_1 = struct_expression_1
and Name_2 : Signature_2 = struct_expression_2
⋮
and Name_n : Signature_n = struct_expression_n
```

The signatures $Signature_i$ are *required*.

For example, let's build a kind of nonempty trees with unbounded branching, and a function `map` with the standard meaning. The `map` function could be defined as part of a single module, but it is also possible to use recursive modules to split the program into two parts 1) mapping over single tree nodes, and 2) mapping over lists of trees.

```
type 'a ubtree = Node of 'a * 'a ubtree list
```

```
module rec Tree : sig
   val map : ('a -> 'b) -> 'a ubtree -> 'b ubtree
end = struct
   let map f (Node (x, children)) =
      Node (f x, Forest.map f children)
end

and Forest : sig
   val map : ('a -> 'b) -> 'a ubtree list -> 'b ubtree list
end = struct
   let map f l =
      List.map (Tree.map f) l
end;;
```

This definition is necessarily recursive because the `Tree` module refers to `Forest.map`, and `Forest` refers to `Tree.map`.

The signatures are required—one way to think of it is that the types of the mutual references are needed so that type checking can be performed for each module in isolation.[1] Stylistically, recursive modules are used infrequently for simple structure definitions; they become much more useful when used with functors, which allow the module bodies to be placed in separate files.

12.4 The `include` directive

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

12.4.1 Using include to extend modules

Suppose we wish to define a new kind of sets `ChooseSet` that has a `choose` function that returns an element of the set if one exists. Instead of re-typing the entire signature, we can use the `include` statement to include the existing signature, as shown in Figure 12.3. The resulting signature includes all of the types and declarations from `SetSig` as well as the new function declaration `val choose`. For this example, we are using the toploop to display the inferred signature for the new module.

12.4.2 Using include to extend implementations

The `include` statement can also be used in implementations. For our example, however, there is a problem. The straightforward approach in defining a module `ChooseSet` is to include the `Set` module, then define the new function `choose`. The result of this attempt is shown in Figure 12.4, where the toploop prints out an extensive error message (the toploop prints out the full signature, which we have elided in `sig ··· end`).

[1] Recursive modules are relatively new to OCaml; these requirements may change.

Signature definition

```
module type ChooseSetSig = sig
   include SetSig
   val choose : 'a set -> 'a option
end;;
```

Inferred type

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

Figure 12.3: Extending a signature with `include`.

Structure definition

```
module ChooseSet
 : ChooseSetSig =
struct
   include Set
   let choose = function
      x :: _ -> Some x
    | [] -> None
end;;
```

Inferred type (from the toploop)

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

Figure 12.4: A failed attempt to extend the `Set` implementation.

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

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

Suppose we start with a non-abstract implementation `SetInternal` of sets as lists. Then the module `Set` is the same implementation, with the signature `SetSig`; and the `ChooseSet` includes the `SetInternal` module instead of `Set`. Figure 12.5 shows the definitions in this order, together with the types inferred by the toploop.

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

Structure definitions

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

module Set : SetSig = SetInternal

module ChooseSet
 : ChooseSetSig =
 struct
   include SetInternal
   let choose = function
      x :: _ -> Some x
    | [] -> None
end;;
```

Inferred types (from the toploop)

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

module Set : SetSig

module ChooseSet : ChooseSetSig
```

Figure 12.5: Extending the `Set` using an internal specification.

12.5 Abstraction, friends, and module hiding

So far, we have seen that modules provide two main features, 1) the ability to divide a program into separate program units (modules) that each have a separate namespace, and 2) the ability to assign signatures that make each structure partially or totally abstract. In addition, as we have seen in the previous example, a structure like `SetInternal` can be given more than one signature (the module `Set` is equal to `SetInternal` but it has a different signature).

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

In our example, there isn't much danger in leaving the `SetInternal` module publicly accessible. A `SetInternal.set` can't be used in place of a `Set.set` or a `ChooseSet.set`, because the latter types are abstract. However, there is a cleaner solution that nests the `Set` and `ChooseSet` structures in an outer `Sets` module. The signatures are left unconstrained within the `Sets` module, allowing the `ChooseSet` structure to refer to the implementation of the `Set` structure, but the signature of the `Sets` module is constrained. The code for this is shown in Figure12.6.

There are a few things to note for this definition.

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

2. Within the `Sets` module the `Set` and `ChooseSet` modules are not constrained, so

Module definitions

```
module Sets : sig
   module Set : SetSig
   module ChooseSet : ChooseSetSig
end = struct
   module Set = struct
      type 'a set = 'a list
      let empty = []
      let add x l = x :: l
      let mem x l = List.mem x l
   end
   module ChooseSet = struct
      include Set
      let choose = function
         x :: _ -> Some x
       | [] -> None
   end
end;;
```

Inferred types (from the toploop)

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

Figure 12.6: Defining `ChooseSet` and `Set` as friends.

that their implementations are public. This allows the `ChooseSet` to refer to the `Set` implementation directly (so in this case, the `Set` and `ChooseSet` modules are friends). The signature for the `Sets` module makes them abstract.

12.5.1 Using include with incompatible signatures

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

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

An alternative is to *derive* a wrapper module `Set2` for use in the second library. The process is simple, 1) `include` the `Set` module, and 2) redefine the `add` to match the desired signature; this is shown in Figure 12.7.

The `Set2` module is just a wrapper. Apart from the `add` function, the types and

Signature

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

Implementation

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

Figure 12.7: Wrapping a module to use a new signature.

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

12.6 Sharing constraints

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

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

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

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

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

In this particular case, we wish to say that the `'a Set2.set` type is equal to the `'a Set.set` type, which we can do by adding a sharing constraint when the `Set2` module is defined, as shown in Figure 12.8.

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

Module definition

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

Toploop

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

Figure 12.8: Defining a sharing constraint.

12.7 Exercises

Exercise 12.1 Which of the following are legal programs? Explain your answers.

1. ```
   module A : sig
      val x : string
   end = struct
      let x = 1
      let x = "x"
   end
   ```
2. ```
   module A : sig
      val x : string
      val x : string
   end = struct
      let x = "x"
   end
   ```
3. ```
   module a = struct
      let x = 1
   end;;
   ```
4. ```
   module M : sig
      val f : int -> int
      val g : string -> string
   end = struct
      let g x = x
      let f x = g x
   end
   ```
5. ```
   let module X = struct let x = 1 end in X.x
   ```
6. ```
   module M = struct
      let g x = h x
      let f x = g x
      let h x = x + 1
   end
   ```
7. ```
   module rec M : sig
      val f : int -> int
      val h : int -> int
   end = struct
      open M
      let g x = h x
      let f x = g x
      let h x = x + 1
   end
   ```
8. ```
   module rec M : sig
      val f : int -> int
   end = struct
      let f = M.f
   end
   ```
9. ```
   type 'a t = { set : 'a -> unit; get : unit -> 'a }
   let f x =
      let cell = ref x in
      let module M = struct
         let s i = cell := i
         let g () = !cell
         let r = { set = s; get = g }
      end
   ```
   ```

```
   in
      M.r
```

10.
```
let f x =
   let cell = ref x in
   let module M = struct
      type 'a t = { set : 'a -> unit; get : unit -> 'a }
      let s i = cell := i
      let g () = !cell
      let r = { set = s; get = g }
   end
   in
      M.r
```

11.
```
module type ASig = sig type s  val f : int -> s end
module type BSig = sig type t  val g : t -> int end
module C : sig
   module A : ASig
   module B : BSig with type t = A.s
end = struct
   type u = string
   module A = struct type s = u  let f = string_of_int end
   module B = struct type t = u  let g = int_of_string end
end
include C
let i = B.g (A.f ())
```

12.
```
module type ASig = sig type t end
module type BSig = sig val x : int end
module A : ASig with type t = int
   = struct type t = int end
module B : BSig = struct let x = 1 end
module C : sig
   include ASig
   val x : t
end = struct
   include A
   include B
end
```

Exercise 12.2 In OCaml, programs are usually written "bottom-up," meaning that programs are constructed piece-by-piece, and the last function is a file is likely to be the most important. Many programmers prefer a top-down style, where the most important functions are defined first in a file, and supporting definitions are placed later in the file. Can you use the module system to allow top-down programming?

Exercise 12.3 One could argue that sharing constraints are never necessary for unparameterized modules like the ones in this chapter. In the example of Figure 12.8, there are at least two other solutions that allow the `Set2` and `Set` modules to share values, without having to use sharing constraints. Present two alternate solutions without sharing constraints.

Exercise 12.4 In OCaml, signatures can apparently contain multiple declarations for the same value.

```
# module type ASig = sig
```

```
    val f : 'a -> 'a
    val f : int -> int
  end;;
module type ASig = sig val f : 'a -> 'a val f : int -> int end
```

In any structure that is given this signature, the function `f` must have *all* the types listed. If `f` is not allowed to raise an exception, what is the only sensible definition for it?

Exercise 12.5 Unlike `val` declarations, `type` declarations must have distinct names in any structure or signature.

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

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

```
# module type XSig = sig
    type t
    val x : t
  end;;
# module A : XSig = struct
    type t = int
    let x = 0
  end;;
# module B : XSig = struct
    type t = int
    let x = 1
  end;;
# module C = struct
    include A
    include B
  end;;
Multiple definition of the type name t.
Names must be unique in a given structure or signature.
```

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

Chapter 13

Functors

Modules often refer to other modules. The modules we saw in Chapter 12 referred to other modules by name. Thus, all the module references we've seen up to this point have been to specific, constant modules.

It's also possible in OCaml to write modules that are parameterized by other modules. To be used, functors are instantiated by supplying actual module arguments for the functor's module parameters (similar to supplying arguments in a function call).

To illustrate the use of a parameterized module, let's return to the set implementation we have been using in the previous two chapters. One of the problems with that implementation is that the elements are compared using the OCaml built-in equality function =. What if, for example, we want a set of strings where equality is case-insensitive, or a set of floating-point numbers where equality is to within a small constant? Rather than re-implementing a new set for each of new case, we can implement it as a functor, where the equality function is provided as a parameter. An example is shown in Figure 13.1, where we represent the set as a list of elements.

In this example, the module `MakeSet` is a functor that takes another module `Equal` with signature `EqualSig` as an argument. The `Equal` module provides two things—a type of elements, and a function `equal` to compare two elements. The body of the functor `MakeSet` is much the same as the previous set implementations we have seen, except now the elements are compared using the function `equal x x'` instead of the builtin-equality `x = x'`. The expression `List.exists f s` is true iff the predicate `f` is true for some element of the list `s`. The `List.find f s` expression returns the first element of `s` for which the predicate `f` is true, or raises `Not_found` if there is no such element.

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

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

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

Set functor

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

module MakeSet (Equal : EqualSig) = struct
   open Equal
   type elt = Equal.t
   type t = elt list
   let empty = []
   let mem x s = List.exists (equal x) s
   let add = (::)
   let find x s = List.find (equal x) s
end;;
```

Building a specific case

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

Using the set

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

Figure 13.1: An implementation of sets based on lists.

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

3. Modules and functors are not first class. That is, they can't be stored in data structures or passed as arguments like other values, and module definitions cannot occur in function bodies.

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

13.1 Sharing constraints

In the `MakeSet` example of Figure 13.1, we omitted the signature for sets. This leaves the set implementation visible (for example, the `SSet.add` function returns a string list). We can define a signature that hides the implementation, preventing the rest of the program from depending on these details. Functor signatures are defined the usual way, by specifying the signature after a colon, as shown in Figure 13.2.

The *sharing constraint* `SetSig with type elt = Equal.t` is an important part of the construction. In the `SetSig` signature, the type `elt` is abstract. If the `MakeSet` functor were to return a module with the plain signature `SetSig`, the type `SSet.elt` would be abstract, and the set would be useless. If we repeat the construction without the sharing constraint, the compiler produces an error message when we try to use it.

```
# module MakeSet (Equal : EqualSig) : SetSig = struct ... end
# module SSet = MakeSet (StringCaseCompare);;
# SSet.add "The Magic Mountain" SSet.empty;;
Characters 9-29:
  SSet.add "The Magic Mountain" SSet.empty;;
           ^^^^^^^^^^^^^^^^^^^^
This expression has type string but is here used with type
  SSet.elt = MakeSet(StringCaseEqual).elt
```

The message indicates that the types `string` and `SSet.elt` are not the same—and in fact, the only property known is that the types `SSet.elt` and `MakeSet(StringCaseEqual).elt` are equal.

13.2 Module sharing constraints

The sharing constraints that we have seen so far apply to types in a module definition. It is also possible to specify sharing constraints on entire modules. The effect of a module

Set signature

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

module MakeSet (Equal : EqualSig)
 : SetSig with type elt = Equal.t =
struct
   type elt = Equal.t
   ...
end;;
```

Building a specific case

```
module StringCaseEqual = struct ··· end;;
module SSet = MakeSet (StringCaseEqual);;
```

Using the set

```
# SSet.empty;;
- : SSet.t = <abstr>
# open SSet;;
# let s = add "Paradise Lost" empty;;
val s : SSet.t = <abstr>
# mem "paradise lOst" s;;
- : bool = true
# find "paradise loSt" s;;
- : string = "Paradise Lost"
```

Figure 13.2: Assigning a signature to the `MakeSet` functor.

sharing constraint is to equate two modules, including all the types contained within the modules. This can be a tremendous benefit when many types must be constrained as part of a new module definition.

To see how this works, let's redefine the `MakeSet` functor using a sharing constraint on the `Equal` module. The first step is to revise the signature `SetSig` to include the module. In this new module signature, the type `elt` is now defined as `Equal.t`.

```
module type SetSig = sig
   module Equal : EqualSig

   type t
   type elt = Equal.t
   val empty : t
   val mem : elt -> t -> bool
   val add : elt -> t -> t
   val find : elt -> t -> elt
end;;
```

The next step is to revise the `MakeSet` functor to include the `Equal` module and the corresponding sharing constraint.

```
module MakeSet (EqArg : EqualSig)
 : SetSig with module Equal = EqArg =
struct
   module Equal = EqArg

   type elt = Equal.t
   type t = elt list
   let empty = []
   ...
end;;
```

The effect of the sharing constraint is to specify that the types `elt` and `Equal.t` are equivalent. In this example, there is only one type, so there isn't much benefit. In general, however, the modules being constrained can include many types, and the module constraint provides both clarity and a reduction in code size.

13.3 Module re-use using functors

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

The *map* and *set* data structures are very similar. Since we have implemented sets already, we will try to re-use the implementation for maps. In this case, we will write a functor that produces a *map* data structure given a comparison function. The code is shown in Figure 13.4.

The `MakeMap` functor takes two parameters, a `Equal` module to compare keys, and a `Value` module that specifies the type of values stored in the table. The constructed

Signature definitions

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

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

Figure 13.3: Signatures for the map module.

```
module MakeMap (Equal : EqualSig) (Value : ValueSig)
 : MapSig
   with type key = Equal.t
   with type value = Value.value
= struct
   type key = Equal.t
   type value = Value.value
   type item = Key of key | Pair of key * value

   module EqualItem = struct
      type t = item
      let equal (Key key1 | Pair (key1, _)) (Key key2 | Pair (key2, _)) =
         Equal.equal key1 key2
   end;;
   module Set = MakeSet (EqualItem);;
   type t = Set.t

   let empty = Set.empty
   let add map key value =
      Set.add (Pair (key, value)) map
   let find map key =
      match Set.find (Key key) map with
         Pair (_, value) -> value
       | Key _ ->
           raise (Invalid_argument "find")
end;;
```

Figure 13.4: Constructing a map (a dictionary) from a set.

module has three parts: 1) type definitions, 2) the construction of a `Set` module, and 3) the implementation of the functions and values for the `Map`.

The `Set` contains values of type `item`, which is defined as either a `Key` or a `Pair`. The set itself always contains key/value pairs `Pair (key, value)`. The `Key key` form is defined so that the `find` function can be implemented without requiring a dummy value.

13.4 Higher-order functors

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

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

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

`functor (`*FunctorName* `:` $signature_1$`) -> `$signature_2$

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

```
module MakeMap (Equal : EqualSig) (Value : ValueSig)
   (MakeSet : functor (Equal : EqualSig) ->
      SetSig with type elt = Equal.t)
 : MapSig
   with type key = Equal.t
   with type value = Value.value
= struct ··· end
```

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

13.5 Recursive modules and functors

In Section 12.3 we saw how modules can be defined recursively, as long as they are all defined in the same file. With functors, the situation becomes much better, because the modules can now be implemented in separate files as functors (although the final recursive construction must still be in a single file).

The syntax for recursive functors remains the same as before, using the syntax `module rec ⋯ and ⋯`, where some of the definitions are functor applications.

To illustrate, let's build a more general versions of sets, where the set elements may be integers or—in addition—other sets. So, for instance, sets like $\{1, 2\}$, $\{\{1, 2\}\}$, and $\{17, \{21\}, \{-3, \{88\}\}\}$ should all be representable.

The immediate problem is that, if we wish to re-use the `MakeSet` functor we already defined to build the set, there appears to be no way to define the type of set elements.

```
type elt = Int of int | Set of ???
```

What should be the type of sets to use in this type definition? Certainly, it should be the type `MakeSet(SetEqual).t`, but the `SetEqual` module must be able to compare sets before the type is constructed.

To solution is to define the `SetEqual` and the corresponding `Set` module recursively. For now, let's just use the builtin equality = to compare sets.

```
type 'set element = Int of int | Set of 'set

module rec SetEqual
 : EqualSig with type t = Set.t element =
struct
    type t = Set.t element
    let equal = (=)
end

and Set : SetSig with type elt = SetEqual.t = MakeSet (SetEqual)
```

The construction is necessarily recursive—the `SetEqual` module refers to the `Set` module and *vice versa*.

Recursive module definitions are often more verbose than non-recursive definitions because the modules are *required* to have signatures. This often means that it is usually necessary to specify types twice, once in the module body, and again in the signature. In this example, the type `t` is defined twice as `type t = Set.t element`. Another issue is that it isn't possible to define a sharing constraint of the form `EqualSig with type t = Int of int | Set of Set.t` because that would be a type definition, not a type constraint. The actual type definition must be given beforehand, and since the type of sets is not known at that point, a type variable `'set` must be introduced to stand for the type of sets.

In practice, recursive modules are used much less frequently than simple non-recursive modules. They are, however, a powerful tool that adds significant expressivity to the module system. Without recursion, the `Set` module could not be constructed using the `MakeSet` functor. When used judiciously, recursion is an important part in maintaining modularity in programs.

13.6 A complete example

For simplicity, we have been using a list representation of sets. However, this implementation is not practical except for very small sets, because the set operations take time linear in the size of the set. Let's explore a more practical implementation based

on a tree representation, where the set operations are logarithmic in the size of the set. For this, we turn to the red-black trees discussed in Section 6.4.

Red-black trees (and binary search trees in general) are labeled trees where the nodes are *in-order*; that is, given a node `n` with label `l`, the left children of `n` have labels smaller than `l`, and the right children have labels larger than `l`. The module `Equal` that we have been using is too coarse. We need a more general comparison function, so the first step is to define a module signature for it.

```
type comparison = LT | EQ | GT

module type CompareSig = sig
   type t
   val compare : t -> t -> comparison
end
```

The `SetSig` signature is unchanged for the most part, but we include a `compare` function so that it will be easy to construct sets of sets.

```
module type SetSig = sig
   module Compare : CompareSig

   type t
   type elt = Compare.t
   val empty   : t
   val add     : elt -> t -> t
   val mem     : elt -> t -> bool
   val find    : elt -> t -> elt
   val compare : t -> t -> comparison
end
```

The rest of the construction is now to define the `MakeSet` functor in terms of red-black trees. The module sketch is as follows, where our objective is to fill in the ellipses $\cdots$ with the actual implementation.

```
module MakeSet (Compare : CompareSig)
 : SetSig with module Compare = Compare =
struct
    module Compare = Compare
    type elt = Compare.t
    type color = Red | Black
    type t = Leaf | Node of color * elt * t * t
    let empty = Leaf
    let add x s = ···
    let mem x s = ···
    let find x s = ···
    let compare s1 s2 = ···
end
```

The definition `module Compare = Compare` may seem a little silly at first, but it defines the placeholder that allows the sharing constraint to be expressed—namely, that the type of elements in the set is `Compare.t`.

To implement the final four functions, we can use the implementation of red-black trees, modified to take the comparison into account. Let's start with the `find` function, which traverses the tree, looking for a matching element. This is the usual recursive

inorder traversal.

```
let rec find x = function
   Leaf -> raise Not_found
 | Node (_, y, left, right) ->
      match Compare.compare x y with
         LT -> find x left
       | GT -> find x right
       | EQ -> y
```

The mem function is similar. It is so similar in fact, that we can simply implement it in terms of find, using the exception to determine membership. The expression ignore (find x s); true discards the result of find x s, returning true.

```
let mem x s =
   try ignore (find x s); true with
      Not_found -> false
```

The function add is the same as the function insert from page **??**, modified to use the generic function compare. The balance function is as before.

```
let add x s =
   let rec insert = function
      Leaf -> Node (Red, x, Leaf, Leaf)
    | Node (color, y, a, b) as s ->
        match Compare.compare x y with
           LT -> balance (color, y, insert a, b)
         | GT -> balance (color, y, a, insert b)
         | EQ -> s
   in
      match insert s with  (* guaranteed to be non-empty *)
         Node (_, y, a, b) -> Node (Black, y, a, b)
       | Leaf -> raise (Invalid_argument "insert");;
```

Finally, we must implement a compare function on sets. The builtin equality (=) is not appropriate. First, we should be using the supplied comparison Compare instead, and second, there may be many different red-black trees that represent the same set. The shape of the tree is determined partly by the order in which elements are added to the set, while we want true set equality: two sets are equal *iff* they have the same elements.

There are many ways to define a set equality. We will implement a simple one that first converts the sets to lists, then performs a lexicographic ordering on the lists. This is not the most efficient comparison, but it is adequate, taking time $O(max(n, m))$ when comparing two sets of size n and m.

```
let rec to_list l = function
   Leaf -> l
 | Node (_, x, left, right) ->
      to_list (x :: to_list l right) left

let rec compare_lists l1 l2 =
   match l1, l2 with
      [], [] -> EQ
    | [], _ :: _ -> LT
    | _ :: _, [] -> GT
    | x1 :: t1, x2 :: t2 ->
```

```
            match Compare.compare x1 x2 with
               EQ -> compare_lists t1 t2
             | LT | GT as cmp -> cmp

let compare s1 s2 =
   compare_lists (to_list [] s1) (to_list [] s2)
```

The expression `to_list [] s` produces a list of elements of the set in sorted order—this is important, because it means that two equal sets have the same list representation. The function `compare_lists` defines the lexicographic ordering of two lists, and the function `compare s1 s2` ties it together.

The `MakeSet` functor is now finished. For a final step, let's repeat the recursive definition of a set of sets, from Section 13.5.

```
type 'set element = Int of int | Set of 'set

module rec Compare
 : CompareSig with type t = Set.t element =
struct
   type t = Set.t element
   let compare x1 x2 =
      match x1, x2 with
         Int i1, Int i2 ->
            if i1 < i2 then LT else if i1 > i2 then GT else EQ
       | Int _, Set _ -> LT
       | Set _, Int _ -> GT
       | Set s1, Set s2 -> Set.compare s1 s2
end

and Set : SetSig with module Compare = Compare = MakeSet (Compare)
```

The function `Compare.compare` compares two elements; we choose to use the usual ordering on integers, the `Set` ordering on sets, and integers are always smaller than sets. Note that the module definition is now recursive not only in the types being defined, but also in the definition of the `compare` function.

13.7 Exercises

Exercise 13.1 Which of the following are legal programs? Explain your answers.

1. ```
   module type XSig = sig val i : int end
   module F (X : XSig) = X
   ```
2. ```
   module type S = sig end
   module Apply (F : functor (A : S) -> S) (A : S) = F (A)
   ```
3. ```
   module type ISig = sig val i : int end
   module F (I : ISig) : ISig = struct let i = I.i + 1 end
   let j = F(struct let i = 1 end).i
   ```
4. ```
   module X = struct type t = int end
   module F (X) = struct type t = X.t end
   ```
5. ```
   module F (X : sig type t = A | B end) : sig type t = A | B end = X
   ```
6. ```
   module F (X : sig type t = A | B end) : sig type t = A | B end =
   struct type t = A | B end
   ```
7. ```
   module F (X : sig type t = A | B end) : sig type t = A | B end =
   struct type t = X.t end
   ```

Exercise 13.2 Consider the following well-typed program.

```
module type T = sig type t val x : t end
module A = struct type t = int let x = 0 end
module B = struct type t = int let x = 0 end
module C = A
module F (X : T) = X
module G (X : T) : T = X
module D1 = F (A)
module D2 = F (B)
module D3 = F (C)
module E1 = G (A)
module E2 = G (B)
module E3 = G (C)
```

Which of the following expressions are legal? Which have type errors?

1. `D1.x + 1`
2. `D1.x = D2.x`
3. `D1.x = D3.x`
4. `E1.x + 1`
5. `E1.x = E2.x`
6. `E1.x = E3.x`
7. `D1.x = E1.x`

Exercise 13.3 How many lines of output does the following program produce?

```
module type S = sig val x : bool ref end

module F (A : S) =
struct
   let x = ref true;;
   if !A.x then begin
```
```

```
        print_string "A.x is true\n";
        A.x := false
    end
end

module G = F (F (F (struct let x = ref true end)))
```

Exercise 13.4 It is sometimes better to define a data structure as a record instead of a module. For example, the record type for the finite sets in this chapter might be defined as follows, where the type `'elt t` is the set representation for sets with elements of type `'elt`.

```
type 'elt t = ···
type 'elt set =
  { empty : 'elt t;
    add   : 'elt -> 'elt t -> 'elt t;
    mem   : 'elt -> 'elt t -> bool;
    find  : 'elt -> 'elt t -> 'elt
  }
```

1. Write a function `make_set : ('elt -> 'elt -> bool) -> 'elt set` that corresponds to the `MakeSet` functor on page 140 (the argument to `make_set` is the equality function). Can you hide the definition of the type `'elt t` from the rest of the program?

2. Is it possible to implement sets two different ways such that both implementations use the same `'elt set` type, but different `'elt t` representations?

3. Consider an alternative definition for sets, where the record type is also parameterized by the set representation.

   ```
   type ('elt, 't) set =
     { empty : 't;
       add   : 'elt -> 't -> 'elt;
       mem   : 'elt -> 't -> bool;
       find  : 'elt -> 't -> 'elt
     }
   ```

   Write the function `make_set` for this new type. What is the type of the `make_set` function?

4. What are some advantages of using the record representation? What are some advantages of using the functor representation?

Exercise 13.5 Suppose you wish to write a program that defines two mutually-recursive functions `f : int -> int` and `g : int -> int`. To keep the design modular, you wish to write the code for the two functions in separate files `f.ml` and `g.ml`. Describe how to use recursive modules to accomplish the task.

Exercise 13.6 In Unix-style systems[1] a *pipeline* is a series of processes p_1 | p_2 | $\cdots$ | p_n that interact through communication channels, where the input of process p_{i+1} is the output of process p_i.

We can use a similar architecture within a program to connect modules, which we will call *filters*, giving them the signature `Filter`. The pipeline itself is given the signature `Pipeline`, where the type of elements passed into the pipeline have type `Pipeline.t`.

```
module type Pipeline = sig
   type t
   val f : t -> unit
end

module type Filter = functor (P : Pipeline) -> Pipeline
```

For example, the following pipeline `CatFile` prints the contents of a file to the terminal, one line at a time.

```
module Print = struct
   type t = string
   let f s = print_string s; print_char '\n'
end

module Cat (Stdout : Pipeline with type t = string) =
struct
   type t = string

   let f filename =
      let fin = open_in filename in
      try
         while true do Stdout.f (input_line fin) done
      with End_of_file -> close_in fin
end

module CatFile = Cat (Print)
```

1. Write a `Uniq` filter that, given a sequence of input lines, discards lines that are equal to their immediate predecessors. All other lines should be passed to the output.

2. Write a `Grep` filter that, given a regular expression and a sequence of input lines, outputs only those lines that match the regular expression. For regular expression matching, you can use the `Str` library. The function `Str.regexp : string -> regexp` compiles a regular expression presented as a string; the expression `Str.string_match r s 0` tests whether a string s matches a regular expression r.

3. Write a function `grep : string -> string -> unit`, where the expression `grep regex filename` prints the lines of the file `filename` that match the pattern specified by the string `regex`, using the pipeline construction and the module `Grep` from the previous part.

[1] UNIX® is a registered trademark of The Open Group.

4. Sometimes it is more convenient for filters to operate over individual characters instead of strings. For example, the following filter translates a character stream to lowercase.

```
module Lowercase (Stdout with type t = char) =
struct
   type t = char
   let f c = Stdout.f (Char.lowercase c)
end
```

   Write a filter `StringOfChar` that converts a character-based pipeline to a string-based pipeline.

```
StringOfChar : functor (P : Pipeline with type t = char) ->
   Pipeline with type t = string
```

5. The pipeline signatures, as defined, seem to require that pipelines be constructed from the end toward the beginning, as a module expression of the form `P1 (P2 ··· (Pn) ···)`. Write a functor `Compose` that takes two filters and produces a new one that passes the output of the first to the second. What is the signature of the `Compose` functor? (Hint: consider redefining the signature for filters.)

Chapter 14

Objects

Object-oriented programming is a programming model based on "objects" and their interactions. The OCaml object system, like many other object-oriented languages, includes various concepts like objects, classes, inheritance, subtype polymorphism, *etc*. OCaml's object system also differs from other languages in several ways. It is quite expressive, and it is *structural* rather than *nominal*, meaning that the names of objects and classes make very little difference. We'll point out some of these differences as we go along. For now, let's start with simple objects, without classes. We'll look at classes and inheritance in Chapter 15, and we'll cover polymorphic classes in Chapter 17.

To begin, it is simplest to think of an object as a collection of data together with functions to operate on that data. The data are called *fields* of the object, and the functions are called *methods*. For example, the following object represents a polygon that includes a method `draw` to draw it on the screen (this examples uses the OCaml `Graphics` package to perform the drawing).

```
# #load "graphics.cma";;  (* Load the Graphics module into the toploop *)
# let poly =
  object
     val vertices = [|(46, 70); (54, 70); (60, 150); (40, 150)|]
     method draw = Graphics.fill_poly vertices
  end;;
val poly : < draw : unit > = <obj>
```

The syntax for an object uses the keywords `object` $\cdots$ `end` as delimiters. Fields are defined with the keyword `val`, and methods are defined with the keyword `method`.

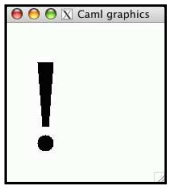
The type of an object is similar to a record type, but it uses angle brackets `<` $\cdots$ `>` as delimiters. The object type includes the method types, but not the types of the fields, so the type of the polygon is simply `< draw : unit >`. It is easy to define other objects with the same type.

```
# let circle =
  object
     val center = (50, 50)
     val radius = 10
     method draw =
```

```
        let x, y = center in
        Graphics.fill_circle x y radius
  end;;
val circle : < draw : unit > = <obj>
```

Methods are invoked by with the syntax *object#method-name* (this is often called *sending a message* to the object). The following sequence of operations opens the graphics window, draws the two objects, and waits for a button to be pressed. The display is shown on the right.

```
Graphics.open_graph " 200x200";;
poly#draw;;
circle#draw;;
ignore (Graphics.wait_next_event [Graphics.Button_down]);;
```



This example illustrates a property of object-oriented programming called *dynamic lookup*. [1] For an expression `obj#m`, the actual method `m` that gets called is determined dynamically by the object `obj`, not by some static property of the program. A polygon draws itself one way, a circle draws itself in another way, but the implementation is the responsibility of the object, not the client.

14.1 Encapsulation and polymorphism

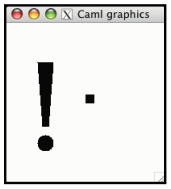
Another important feature of object-oriented programming is *encapsulation*, also called *abstraction*. An object encapsulates some data with methods for operating on the data; it isn't necessary to know how an object is implemented in order to use it. In our example, the polygon and the circle have a single method `draw`, so they have the same type, and they can be used in the same ways. Let's define a function to draw a list of objects.

```
# let draw_list items =
    List.iter (fun item -> item#draw) items;;
val draw_list : < draw : unit; .. > list -> unit = <fun>
# draw_list [poly; circle];;
- : unit = ()
```

Note the type of the function `draw_list`, which specifies that it takes a list of objects of type `< draw : unit; .. >`. The ellipsis `..` in this type stands for "other" methods. That is, the function `draw_list` takes a list of objects having at least a method `draw : unit`, and possibly some other methods. Suppose we defined a new kind of object that represents a square that, in addition to having a method `draw`, also defines a method `area` to compute the area.

[1] Dynamic lookup is often called *polymorphism* in object-oriented circles, but that conflicts with the term *polymorphism* that we use for ML. The two are not at all the same. Following Mitchell [6], we'll use the term polymorphism to refer to *parametric polymorphism* (type polymorphism), and *dynamic lookup* to refer to object polymorphism.

```
# let square =
  object
     val lower_left = (100, 100)
     val width = 10
     method area = width * width
     method draw =
        let (x, y) = lower_left in
        Graphics.fill_rect x y width width
  end;;
val square : < area : int; draw : unit > = <obj>
# draw_list [square];;
- : unit = ()
```



If we had used the simpler type `draw_list_exact : < draw : unit > list -> unit`, the list drawing function would work only with objects having *exactly one* method, the method `draw`. The expression `draw_list_exact [square]` produces a type error, because the object `square` has an extra method `area`.

```
# let draw_list_exact (items : < draw : unit > list) =
    List.iter (fun item -> item#draw) items;;
val draw_list_exact : < draw : unit > list -> unit = <fun>
# draw_list_exact [square];;
Characters 17-23:
  draw_list_exact [square];;
                  ^^^^^^
This expression has type < area : int; draw : unit >
but is here used with type < draw : unit >
The second object type has no method area
```

Technically speaking, an occurrence of an ellipsis `..` in an object type is called a *row variable*, and the scheme for typing is called *row polymorphism*. It might not look like it, but the type is really polymorphic, as we'll see if we try to write a type definition.

```
# type blob = < draw : unit; ..>;;
Characters 4-30:
  type blob = < draw : unit; ..>;;
      ^^^^^^^^^^^^^^^^^^^^^^^^^^
A type variable is unbound in this type declaration.
In definition < draw : unit; .. > as 'a the variable 'a is unbound
```

The issue is that an ellipsis `..` is like a type variable, standing for the types of all the "other" methods. Unfortunately, it doesn't look like a type variable, and it doesn't make sense to write `type (..) blob = < draw : unit; .. >`. The error message is a little cryptic, but it suggests the solution, which is to introduce a type variable `'a` that stands for the type of the entire object. The `as` form and the `constraint` form are equivalent; you can write it either way.

```
# type 'a blob = < draw : unit; .. > as 'a;;
type 'a blob = 'a constraint 'a = < draw : unit; .. >
# let draw_list_poly : 'a blob list -> unit = draw_list;;
val draw_list_poly : < draw : unit; .. > blob list -> unit = <fun>
# draw_list_poly [square];;
- : unit = ()
```

14.2 Transformations

An important feature of any 2D graphics library is the ability to transform objects by scaling, rotation, or translation. The cleanest way to do this is to make use of so-called *homogeneous coordinates*, where the 2D coordinates (x, y) are represented as triples $(x, y, 1)$, and transformations are represented as 3×3 *transformation matrices*. A 3×3 matrix has 9 values t_{ij}, written as follows.

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

The product of a matrix and a vector is computed as follows, where ab represents the product of a and b.

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} t_{11}x + t_{12}y + t_{13}z \\ t_{21}x + t_{22}y + t_{23}z \\ t_{33}x + t_{23}y + t_{33}z \end{pmatrix}$$

The product of two matrices is computed as follows,

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{pmatrix} \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{pmatrix}$$

where $u_{ij} = s_{i1}t_{1j} + s_{i2}t_{2j} + s_{i3}t_{3j}$.

14.2.1 Basis transformations

The basis transformation matrices are specified as follows.

Scale by (s_x, s_y)	Rotate by θ	Translate by (dx, dy)
$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix}$

Transformations are composed by multiplying their matrices. The application of a transformation to a point is also a matrix multiplication, treating the coordinate as a column vector. The following formula represents a scaling by (s_x, s_y) followed by a translation by (dx, dy) (for a formula $(T_n \cdots T_2T_1)p$, the matrix T_1 is the first transformation, and T_n is the last).

$$\left(\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \right) \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} s_x & 0 & dx \\ 0 & s_y & dy \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} s_x x + dx \\ s_y y + dy \\ 1 \end{pmatrix}$$

14.2.2 Functional update

Let's implement an object that represents a transformation matrix. We could implement three separate functions to produce the basis transformations, but that would require duplicating the object definition. It will be easier to implement them as methods instead.

Let's start with the basis transformations. Since the last row in a transformation is always $(0\ 0\ 1)$, we'll just omit it and use a flattened 6-tuple to represent the matrix.

Matrix		Flattened representation as a 6-tuple
$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ 0 & 0 & 1 \end{pmatrix}$	$\Longrightarrow$	$(x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23})$

First, let's write the methods `new_scale`, `new_rotate`, and `new_translate` that construct the basis transformations. For the moment, we're omitting the implementations of the methods `transform` and `multiply`.

```
# let transform =
object
   val matrix = (1., 0., 0., 0., 1., 0.)
   method new_scale sx sy =
      {< matrix = (sx, 0., 0., 0., sy, 0.) >}
   method new_rotate theta =
      let s, c = sin theta, cos theta in
      {< matrix = (c, -.s, 0., s, c, 0.) >}
   method new_translate dx dy =
      {< matrix = (1., 0., dx, 0., 1., dy) >}
   method transform (x, y) = ···
   method multiply matrix2 = ···
end;;
val transform :
  < new_scale : float -> float -> 'a;
    new_rotate : float -> 'a;
    new_translate : float -> float -> 'a;
    transform : float * float -> int * int;
    multiply : ... > as 'a = <obj>
```

The expression `{< ··· >}` represents a *functional update*. This kind of update produces a new object that is the same as the current object, except for the specified changes; the original object is not affected. For example, an expression `{< >}` would produce an identical copy of the current object. In the `transform` object, the expression `{< matrix =` *expression* `>}` produces a new transform object with new values for the field `matrix`. In our example, the canonical object `transform` is the identity transformation, and each basis method `new_`... produces a new object by discarding the original value of the field `matrix`, and replacing it with new values that implement the desired transformation. In effect, each method `new_`... is a *constructor* that constructs a new object, using the current one as a template.

In general, there are two things to keep in mind when using a functional update. First, the expression form `{< ··· >}` can be used only in a method body. Second, the

update can be used only to update fields, not methods—method implementations are fixed at the time the object is created.

14.3 Binary methods

Let's return to the implementation and fill in the remaining methods. The method `transform` is just a matrix multiplication, which we write out by hand.

```
method transform (x, y) =
   let (m11, m12, m13, m21, m22, m23) = matrix in
   (m11 *. x +. m12 *. y +. m13,
    m21 *. x +. m22 *. y +. m23)
```

The `multiply` method is a little harder. The problem is that in OCaml, unlike some other object-oriented languages, fields are private to an object. The `multiply` method is called a *binary method* because it takes another object of the same type as an argument. A binary method cannot directly access the fields of the object passed as an argument.

There are several approaches to dealing with binary methods, but the easiest one here is to add a method `representation` that exposes the internal representation of the object.[2] The remaining part of the implementation is as follows.

```
# let transform =
object
   ...
   method representation = matrix
   method multiply matrix2 =
      let (x11, x12, x13, x21, x22, x23) = matrix in
      let (y11, y12, y13, y21, y22, y23) = matrix2#representation in
      {< matrix =
            (x11 *. y11 +. x12 *. y21,
             x11 *. y12 +. x12 *. y22,
             x11 *. y13 +. x12 *. y23 +. x13,
             x21 *. y11 +. x22 *. y21,
             x21 *. y12 +. x22 *. y22,
             x21 *. y13 +. x22 *. y23 +. x23)
      >}
end;;
val transform : < ... > as 'a = <obj>
# let ( ** ) t1 t2 = t1#multiply t2;;
val ( ** ) : < multiply : 'a -> 'b; .. > -> 'a -> 'b = <fun>
```

14.4 Object factories

Now that we have defined the transformation, let's return to the graphical objects, where we now want to add a new method `transform` to apply a transformation to the object. It can get tedious to define an entire object each time it is created, so instead we will write functions that create new objects (functions that create new objects

[2] Of course, this is not always desirable. In Section 15.4, we describe a way to use the module system to improve abstraction.

are often called *factories*). In addition, we now represent the coordinates as pairs of floating-point numbers. Here is the implementation of the factory `new_poly`; the code for circles is similar.[3]

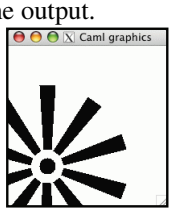
```
# let int_coord (x, y) = (int_of_float x, int_of_float y);;
val int_coord : float * float -> int * int = <fun>
# let new_poly vertices =
object
   val vertices = vertices
   method draw = Graphics.fill_poly (Array.map int_coord vertices)
   method transform matrix = {< vertices = Array.map matrix#transform vertices >}
end;;
val new_poly :
  (float * float) array ->
  (< draw : unit;
     transform : < transform : float * float -> float * float; .. > -> 'a >
   as 'a) = <fun>
```

Note the type of the `transform` method, which is more subtle than it might seem. Remember that the ellipsis `..` is a row variable—it is like a type variable, so the type of polygons really contains two type variables. We'll revisit this issue in the next section.

Finally, to illustrate our graphics library in action, let's draw a few transformed and rotated objects.

```
let poly = new_poly [|(-0.05, 0.2); (0.05, 0.2); (0.1, 1.0); (-0.1, 1.0)|] in
let circle = new_circle (0.0, 0.0) 0.1 in
let matrix1 =
   (transform#new_translate 50.0 50.0) ** (transform#new_scale 100.0 100.0) in
   for i = 0 to 9 do
      let matrix2 = matrix1 ** (transform#new_rotate (0.628 *. float_of_int i)) in
      (poly#transform matrix2)#draw
   done;
   (circle#transform matrix1)#draw;;
```

This program starts with two objects, `poly` and `circle`, centered at the origin. The initial transformation `matrix1` scales by 100 and centers the image on $(50, 50)$. The transformation `matrix2` draws a ray rotated about the point $(50, 50)$ to form a kind of star. The following image shows the output.



14.5 Imperative objects

Next, it is natural to want to define a collection of items that acts like a single drawable object. This time, let's define it imperatively, so that the collection includes a method

[3]Note that the function to draw circles `Graphics.fill_circle` takes the circle's radius. In general, however, a transformed circle should be an ellipse.

add that adds an item to the collection by side-effect. The syntax for a field that can be modified is `val mutable` *identifier* `=` *expression*.

```
# let new_collection () =
object
   val mutable items = []
   method add item = items <- item :: items
   method draw = List.iter (fun item -> item#draw) items
   method transform matrix =
      {< items = List.map (fun item -> item#transform matrix) items >}
end;;
val new_collection :
  unit ->
  (< add : (< draw : unit; transform : 'c -> 'b; .. > as 'b) -> unit;
     draw : unit; transform : 'c -> 'a >
   as 'a) = <fun>
```

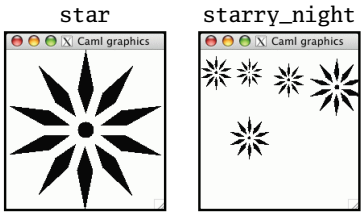
Apart from the inferred type, the definition is reasonably simple. The field `items` is declared as mutable, and the method `add` modifies it by side-effect (using `<-` for assignment). Let's build a star.

```
let star =
   let poly = new_poly [|(0.0, 0.2); (0.1, 0.5); (0.0, 1.0); (-0.1, 0.5)|] in
   let star = new_collection () in
   star#add (new_circle (0.0, 0.0) 0.1);
   for i = 0 to 9 do
      let trans = transform#new_rotate (0.628 *. (float_of_int i)) in
      star#add (poly#transform trans)
   done;
   star;;
```

Since the `star` object is also a drawable object, we can also build a collection with multiple stars.

```
let starry_night =
   let starry_night = new_collection () in
   let add_star (x, y, scale) =
      let trans = (transform#new_translate x y)
         ** (transform#new_scale scale scale) in
      starry_night#add (star#transform trans) in
   List.iter add_star
      [0.35, 0.50, 0.15;
       0.12, 0.95, 0.12;
       0.35, 0.95, 0.10;
       0.62, 0.90, 0.12;
       0.95, 0.85, 0.20];
   starry_night
```

The images for the objects `star` and `starry_night` are shown below.



14.6 self: referring to the current object

Suppose we wish to define a method that is recursive, or a method that calls another method in the same object. In OCaml, the fields of an object can be referred to directly by name, but methods must always use the syntax *object#method-name*. If we wish to call a method in the current object, the object must first be named with the following syntax, where the name occurs in parenthesis after the `object` keyword.

```
object (pattern) ··· end
```

The pattern can use any identifier, but by convention the current object is usually named `self`. It is often specified with a type as well. The following form is conventional, where the name `self` refers to the current object, and `'self` is its type.

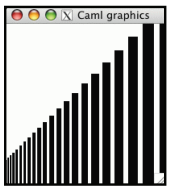
```
object (self : 'self) ··· end
```

Now that we have a name for the current object, let's define a collection method `add_multiple n trans item` that adds `n` of the `item` to the collection, transforming each copy. The easiest way to define the method is to make it recursive.

```
let new_collection () =
   object (self : 'self)
      val mutable items = []
      method add item = items <- item :: items
      method add_multiple n matrix item =
         if n > 0 then begin
            self#add item;
            self#add_multiple (n - 1) matrix (item#transform matrix)
         end
      method draw = List.iter (fun item -> item#draw) items
      method transform matrix = ···
   end;;
```

The expression `self#add item` is a method call that adds the item to the current collection, and `self#add_multiple` is a recursive call.

```
let line =
   new_poly [|(0., 0.); (2., 0.); (2., 30.); (0., 30.)|];;
let xform =
   transform#new_translate 3. 0. ** transform#new_scale 1.1 1.1;;
let image = new_collection ();;
image#add_multiple 25 xform line;;
image#draw;;
```



14.7 Initializers; private methods

There is an important rule to keep in mind when constructing objects.

—Field expressions may not refer to other fields, nor to *self*.—

Here is an example.

```
# object
     val x = 1
     val x_plus_1 = x + 1
  end;;
Characters 38-39:
     val x_plus_1 = x + 1
                    ^
The instance variable x
cannot be accessed from the definition of another instance variable
```

The technical reason for this is that the object doesn't exist when the field values are being computed, so it is an error to refer to the object or any of its fields and methods.

As one way of addressing this problem, objects can contain an *initializer*, written `initializer` *expression*. The initializer expression is evaluated just after the object is created, but before it is used. The object exists at initialization, so it is legal to refer to its fields and methods in the initializer.

```
# object
     val x = 1
     val mutable x_plus_1 = 0
     initializer
        x_plus_1 <- x + 1
  end;;
```

Initializers are especially useful when an object has an invariant to be maintained, or when the value of one field is derived from another. For a more realistic example along these lines, let's write a version of the polygon object that allows the polygon to be transformed in place (by side-effect).

```
let new_imp_poly vertices =
   object
      val mutable vertices = vertices
      method draw = Graphics.fill_poly (Array.map int_coord vertices)
      method transform matrix = {< >}#transform_in_place matrix
      method transform_in_place matrix =
         vertices <- Array.map matrix#transform vertices
   end;;
```

One potential source of inefficiency in this object is that the vertices are represented with floating-point coordinates, but must be drawn with integer coordinates. The method `draw` performs the conversion each time the object is drawn.

An alternative is to maintain two versions of the vertices, `float_vertices` and `int_vertices`, with the invariant that `int_vertices = Array.map int_coord float_vertices`. When one field is derived from another like this, it is usually best to define a method that handles

changes to the fields. In this case, the method `set_vertices` updates the object with new coordinates.

```
# let new_imp_poly vertices =
object (self : 'self)
   val mutable float_vertices = [||]
   val mutable int_vertices = [||]
   method draw = Graphics.fill_poly int_vertices
   method transform matrix = {< >}#transform_in_place matrix
   method transform_in_place matrix =
      self#set_vertices (Array.map matrix#transform vertices)
   method private set_vertices vertices =
      float_vertices <- vertices;
      int_vertices <- Array.map int_coord float_vertices
   initializer
      self#set_vertices vertices
end;;
val new_imp_poly : (float * float) array ->
  < draw : unit;
    transform : (< transform : float * float -> float * float; .. > as 'a) -> unit;
    transform_in_place : 'a -> unit > = <fun>
```

The method `set_vertices` is called in two places, 1) by the `initializer` to set the initial values of the vertices, and 2) by the method `transform_in_place`, which computes new values for the vertices. The object reference `self` is legal in the initializer, allowing the method call `self#set_vertices`.

This example also contains a *private method*. Private methods are defined with the syntax `method private` *identifier* `=` *expression*. They are used just like normal (public) methods, but they are not visible outside the object—they don't even appear in the object type.

14.8 Object types, coercions, and subtyping

The types of the objects we have been creating are getting pretty complicated. To make sense of it all, let's make some type definitions. We don't really care about giving the most general polymorphic types, so let's use exact non-polymorphic types instead. We'll call a drawable object a `blob`.

```
type coord = float * float
type transform = < transform : coord -> coord >
type blob = < draw : unit; transform : transform -> blob >
type collection =
   < add : blob -> unit;
     draw : unit;
     transform : transform -> collection
   >
```

Note that the type `collection` differs from `blob` in two ways. A collection has an extra method `add`, and the method `transform` returns another collection, not a blob.

We can now annotate the object creation functions to get simpler types (the object definitions are the same as before).

```
# let new_poly (vertices : coord array) : blob = object ··· end;;
val new_poly : coord array -> blob = <fun>
# let new_circle (center : coord) radius : blob = object ··· end;;
val new_circle : coord -> float -> blob = <fun>
# let new_collection () : collection = object ··· end;;
val new_collection : unit -> collection = <fun>
```

Now that the types are simplified, we run into a new issue: the actual object types do not match the expected exact types. For example, the method `transform` now expects an object with exactly one method (the method is also called `transform`), but the real object has many more methods. Here is what we get if we try to perform a transformation.

```
# let circle = new_circle (0.0, 0.0) 0.1;;
val circle : blob = <obj>
# circle#transform (transform#new_translate 100.0 100.0);;
Characters 17-54:
  circle#transform (transform#new_translate 100.0 100.0);;
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type
  < multiply : ... > as 'a
but is here used with type transform
The second object type has no method multiply
```

The problem here is that the expression `(transform#new_translate 100.0 100.0)` produces an actual object with many methods, while the method `circle#transform` expects an object having exactly one method. In principle, it should be fine to pass an object with more methods to one that expects fewer—all the extra methods can simply be ignored.

14.8.1 Coercions

In OCaml, such coercions are in fact legal, but they are not automatic. This is in accord with the usual OCaml policy that all coercions should be explicit; for example, integers are never coerced automatically to floating-point values, the function `float_of_int` must be written explicitly.

An explicit object coercion can be written two ways, as a "single coercion" or as a "double coercion."

(*object* `:>` *object-type*)	single coercion
(*object* `:` *object-type* `:>` *object-type*)	double coercion

The single coercion expression (`e :> t`) coerces the object e to have type t (if legal). The double coercion expression (`e : t1 :> t2`) means to consider first that e has type t_1, then coerce it to an object of type t_2. In most cases, a single coercion is sufficient.

```
# circle#transform (transform#new_translate 100.0 100.0 :> transform);;
- : blob = <obj>
```

For another example, consider the collection object, which contains a list of simple blobs. If we want to add any other kind of object, it must be coerced.

```
# let image = new_collection ();;
val image : collection = <obj>
# image#add (new_circle (0.0, 0.0) 0.1);;
- : unit = ()
# image#add star;;
Characters 10-14:
  image#add star;;
            ^^^^
This expression has type collection but is here used with type blob
The second object type has no method add
```

That is as we expected, but the single coercion doesn't work either.

```
# image#add (star :> blob);;
Characters 11-15:
  image#add (star :> blob);;
             ^^^^
This expression cannot be coerced to type
  blob = < draw : unit; transform : transform -> blob >;
...
This simple coercion was not fully general. Consider using a double coercion.
```

The real error message is quite long, most of it has been elided. The last line suggests using a double coercion, so we try that. Finally, success.

```
# star;;
- : collection = <obj>
# image#add (star : collection :> blob);;
- : unit = ()
```

Why does the single coercion sometimes work, but at other times the double coercion is required? The complete technical explanation is complicated and has to do with the specific algorithm used for type inference. The simplest rule is this: if the compiler complains about a single coercion, try replacing it with a double coercion. If you would like to know the real reason, a bit of explanation might be helpful.

Internally, the compiler uses only double coercions. Whenever the compiler encounters a single coercion (`(`e `:>` t_2`)`) it constructs a double coercion (`(`e `:` t_1 `:>` t_2`)`) by inferring the *most general expected type* t_1. However, this fails if there is no unique most general expected type. The general guidelines can be stated as follows.

A single coercion (`(`e `:>` t_2`)`) may fail if:

- the type t_2 is recursive, or
- the type t_2 has polymorphic structure.

If either condition holds, use a double coercion (`(`e `:` t_1 `:>` t_2`)`).

In our example, the single coercion (`transform#new_translate 100.0 100.0 :> transform`) is successful because the type `transform` is neither recursive nor polymorphic. However, the coercion (`star :> blob`) fails because the type `blob` is recursive. The compiler doesn't consider the actual type of the expression, so even though we know `star` has type `collection`, it is still necessary to write the double coercion (`star : collection :> blob`).

14.8.2 Subtyping

That's not the entire story of course, because not all coercions (`(`e `:` t_1 `:>` t_2`)`) are legal. There are two necessary conditions: first, expression e should have type t_1; and second, type t_1 must be a subtype of t_2.

We say that a type t_1 is a *subtype* of t_2, written t_1 `<:` t_2, if values of type t_1 can be used where values of type t_2 are expected. It may be confusing that the symbols `:>` (the coercion operator) and `<:` (the subtyping relation) look like they point in opposite directions. It may be helpful to remember that the former is an operator, and the latter is a relation not belonging to the syntax of the language.

Consider the following type definitions: an `animal` eats, and a `dog` also barks.

```
type animal = < eat : unit >
type dog = < eat : unit; bark : unit >
```

The subtyping relation `dog <: animal` holds because a `dog` object has all the methods that an animal has with the same type, and so a dog object e can be used wherever an `animal` object is expected (so the coercion (`(`e `: dog :> animal``)`) is legal).

Width and depth subtyping

Subtyping for object types takes two forms, called *width* and *depth* subtyping. Width subtyping means that an object type t_1 is a subtype of object type t_2 if t_1 implements all the methods (and possibly more) of t_2 with the same method types. The order of methods in an object type doesn't matter, so we can write this as follows, where we use the notation $f_{1..n}$ `:` $t_{1..n}$ to represent n method declarations f_i `:` t_i for $i \in \{1, 2, \ldots, n\}$.

`<` $f_{1..n}$ `:` $t_{1..n}$`,` $g_{1..m}$ `:` $s_{1..m}$ `> <: <` $f_{1..n}$ `:` $t_{1..n}$ `>`

The subtyping relation `dog <: animal` follows from width subtyping, because class type `dog` implements the `eat` method, the only method in the `animal` class type.

```
< eat : unit; bark : unit > <: < eat : unit >
```

Depth subtyping means that an object type t_1 is a subtype of t_2 if the two types have the same methods, but the method types in t_1 are subtypes of the corresponding types in t_2. This rule is usually written as follows, as an *inference rule*, where the subtyping properties above the horizontal line imply the subtyping property below the line. We read it informally as follows, "If each method type s_i is a subtype of method type t_i, then the object type `<` $f_{1..n}$ `:` $s_{1..n}$ `>` is a subtype of the object type `<` $f_{1..n}$ `:` $t_{1..n}$ `>`."

$$\frac{s_i \texttt{ <: } t_i \quad \text{(for each } i \in \{1, \ldots, n\})}{\texttt{< } f_{1..n} \texttt{ : } s_{1..n} \texttt{ > <: < } f_{1..n} \texttt{ : } t_{1..n} \texttt{ >}}$$

In general, the method types may include various type constructors for tuples, lists, records, functions, other objects, *etc*. Each type constructor in OCaml has its own subtyping rules describing how the type construction varies in terms of its component types. These variances can be *covariant*, meaning that the construction varies in the

same way as a component type; *contravariant*, meaning the construction varies oppositely to a component type; or *invariant*, which means that it is neither purely covariant nor purely contravariant.

For example, consider the tuple type t_1 `*` t_2, which is covariant in both types t_1 and t_2. If we have two dogs `dog * dog`, then we also have two animals `animal * animal` (so `dog * dog <: animal * animal`). The inference rule for pairs is specified as follows.

$$\frac{s_1 <: t_1 \quad s_2 <: s_2}{(s_1 * s_2) <: (t_1 * t_2)}$$

Function subtyping

Nearly all type constructors in OCaml are covariant over all their component types, but there are two exceptions. One is that types that specify mutable values are always invariant. The other exception is more interesting, for the function type. A function type t_1 `->` t_2 is covariant in its range type t_2, but *contravariant* in the domain type t_1. This property is written as follows.

$$\frac{t_1 <: s_1 \quad s_2 <: t_2}{(s_1 \texttt{->} s_2) <: (t_1 \texttt{->} t_2)}$$

The contravariance in function types is the source of many problems in the design of object-oriented programming languages, and it can be difficult to understand. To get some intuition, consider a function `feed` for feeding an animal.

```
# let feed (x : animal) = x#eat;;
val feed : animal -> unit = <fun>
```

When calling the function, we can pass it an `animal` object or a `dog` object—both support the `eat` method. Thus, if we like, we can coerce the function to have type `dog -> unit`.

```
# let feed_dog = (feed : animal -> unit :> dog -> unit);;
val feed_dog : dog -> unit = <fun>
```

Now consider an barking function for dogs.

```
# let do_bark (x : dog) = x#bark;;
val do_bark : dog -> unit = <fun>
```

We can't pass a plain `animal` object to `do_bark`, because animals do not bark in general. In general, we *cannot* use a function of type `dog -> unit` in places where a function of type `animal -> unit` is expected.

```
# (do_bark : dog -> unit :> animal -> unit);;
Characters 0-41:
  (do_bark : dog -> unit :> animal -> unit);;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Type dog -> unit is not a subtype of type animal -> unit
Type animal = < eat : unit > is not a subtype of type
  dog = < bark : unit; eat : unit >
```

Subtyping of recursive object types

Finally, let's consider subtyping for recursive object types. In this case, the subtyping rule is circular: first, assume that the subtyping relationship holds, then prove that it holds. Don't worry—in this particular case the circular argument is sound.

For example, consider the argument that a `collection` is a `blob`. The types are recursive, so we first assume that the relation holds, and then prove it. Here is the argument.

- Assume: `collection <: blob`.
- Show:

```
< add : blob -> unit;                          <:    < draw : unit;
  draw : unit;                                       transform : transform -> blob
  transform : transform -> collection                >
>
```

- From width subtyping, we can ignore the `add` method.

```
< draw : unit;                                 <:    < draw : unit;
  transform : transform -> collection                transform : transform -> blob
>                                                    >
```

- The `draw` methods have the same type. The `transform` method is justified by depth subtyping. We must show the following.

```
transform -> collection <: transform -> blob
```

- Functions are covariant in the range type. The final goal is the following.

```
collection <: blob
```

  This follows by assumption.

14.9 Narrowing

In object-oriented programming, *narrowing* is the ability to coerce an object to one of its subtypes—the opposite of a normal coercion. This is commonly used when the actual type of an object has been lost due to a coercion somewhere else in the program. For example, suppose we have defined some types for cats and dogs. If we want a list that contains both cats and dogs, the elements of the list must coerced to a common type, in this case the supertype `animal`.

```
type animal = < eat : unit >
type dog = < eat : unit; bark : unit >
type cat = < eat : unit; meow : unit >

let fido : dog = ···
let daphne : cat = ···

let animals = [(fido :> animal); (daphne :> animal)]
```

Because of the coercion, the methods bark and meow have been lost, which can be a potential problem. Languages that support narrowing usually include a "typecase" to perform a case analysis on an object's actual type. The following function illustrates how a typecase would be written, presented in OCaml-style pseudo-code.

```
(* THIS IS NOT LEGAL OCAML CODE *)
let chorus (animals : animal list) =
   List.iter (fun animal ->
      if animal instanceof dog then (animal :> dog)#bark
      else if animal instanceof cat then (animal :> cat)#meow) animals
```

Narrowing is **not permitted** in OCaml. Period. First, we'll give some arguments why narrowing is undesirable, then we'll describe how to do it anyway.

14.9.1 Why narrowing is bad

The usual argument against narrowing is that it is unsound. Actually, it is probably more accurate to say that it isn't known whether there is a useful, general form of narrowing that is compatible with the OCaml type system.

We might leave it at that, but there are good *design* principles that also argue against narrowing. One of the principal benefits of object-oriented programming is that the responsibility of implementation is shifted to the object, away from the client. The need for case analysis is reduced because of dynamic lookup, which ensures that the code that is executed is always appropriate to the object.

Looking at our example, the problem is that a single concept, let's call it "speak," is named in two different ways, bark and meow. A better implementation would use the same name in both cases, perhaps also keeping the species-specific name.

```
type animal = < eat : unit; speak : unit >
type dog = < eat : unit; speak : unit; bark : unit >
type cat = < eat : unit; speak : unit; meow : unit >
type lizard = < eat : unit; speak : unit; sleep : unit >
let fido : dog = object (self) method speak = self#bark ··· end
let daphne : cat = object (self) method speak = self#meow ··· end
let fred : lizard = object (self) method speak = () ··· end
let animals = [(fido :> animal); (daphne :> animal); (fred :> animal)]

let chorus (animals : animal list) =
   List.iter (fun animal -> animal#speak) animals
```

No case analysis is necessary; it is the responsibility of each animal to decide how it will speak.

In other cases, the abstraction must be changed to avoid the lossy coercion. For example, one might decide that, out of a collection of animals, the dogs should bark but all other animals should remain silent. The solution in that case is to avoid the coercion in the first place. If it is important to know which of the animals are dogs and which are cats, then a single list of animals is not appropriate. Multiple lists should be used instead.

14.9.2 Implementing narrowing

If, despite these arguments, you still wish to use narrowing, it is fairly easy to implement. There are two things we need: a runtime "tag" that indicates what the actual type of an object is, and a "typecase" for case analysis over the tag. The tags should be *open* so that new subclasses can be added, which leaves us with two choices: polymorphic variants (Section 6.5), or exceptions. We'll use exceptions for illustration because the types are easier to write down. The main idea is to define a method `actual` that returns the actual object.

```
type narrowable_object = < actual : exn >
type animal = < actual : exn; eat : unit >
type dog = < actual : exn; eat : unit; bark : unit >
type cat = < actual : exn; eat : unit; meow : unit >
exception Dog of dog
exception Cat of cat

let fido : dog = object (self) method actual = Dog self ··· end
let daphne : cat = object (self) method actual = Cat self ··· end

let animals = [(fido :> animal); (daphne :> animal)]

let chorus (animals : animal list) =
   List.iter (fun animal ->
      match animal#actual with
         Dog dog -> dog#bark
       | Cat cat -> cat#meow
       | _ -> ()) animals
```

The idea here is to define a constructor for each actual type of object (defined here as the exceptions `Dog` and `Cat`). The method `actual` returns a tagged object, and the case analysis uses a `match` expression. The use of exceptions is a technicality; Exercise 14.5 discusses narrowing using polymorphic variants.

14.10 Alternatives to objects

The objects we have seen in this chapter are simple, serving mainly to encapsulate some data together with functions that operate on that data. In many ways, they are similar to abstract data types. In fact, much of what we have done in this chapter can also be done using the module system. However, there are two key differences: 1) with objects, the type of the data is entirely hidden, and 2) objects are first-class values, while modules are not. For example, a module to implement a polygon might be written as follows.

```
module type PolySig = sig
   type poly
   val create : (float * float) array -> poly
   val draw : poly -> unit
   val transform : poly -> transform -> poly
end;;

module Poly : PolySig =
   type t = (float * float) array
```

```
    let create vertices = vertices
    let draw vertices = Graphics.fill_poly (Array.map int_coord vertices)
    let transform matrix = Array.map matrix#transform vertices
end;;
```

The main problem with this approach is that, even though the data type `Poly.poly` is abstract, it is explicit. A polygon has type `Poly.poly`; a circle would have a similar type like `Circle.circle`, *etc*. This means, for example, that one cannot create a list containing both polygons and circles without further effort.

In Exercise 8.4, we suggested that a simplified object could be represented as a record of methods. In fact, this is quite similar to what we have seen in this chapter, except for functional update. Note the recursive call to the function `new_poly` in the following implementation.

```
type blob =
  { draw : unit -> unit;
    transform : transform -> blob
  }

let rec new_poly vertices =
  { draw = Graphics.fill_poly (Array.map int_coord vertices);
    transform = (fun matrix -> new_poly (Array.map matrix#transform vertices))
  }
```

We can build many object features from records and other parts of the language, but the fact is that the simple objects we have seen in this chapter provide a simple, useful programming model. However, we are still missing one of the most important features, *inheritance*, the topic of the next chapter.

14.11 Exercises

Exercise 14.1 In Section 14.2 we implemented the three factory functions `new_scale`, `new_rotate`, and `new_translate` as methods, claiming that it would avoid code duplication. Write one of the factory functions as a normal function (not a method). How can you avoid code duplication?

Exercise 14.2 In Section 14.4 the factory functions include some apparently silly field definitions. For example, the function `new_poly` includes the field `val vertices = vertices`. What is the purpose of the field definition? What would happen if it were omitted?

Exercise 14.3 Suppose we wish to enforce the fact that a program contains only one copy of an object. For example, the object may be an accounting object, and we wish to make sure the object is never copied or forged.

The standard library module `Oo` contains a function that copies any object.

```
val copy : (< .. > as 'a) -> 'a
```

Modify the following object so that it refuses to work after being copied.

```
let my_account =
object
   val mutable balance = 100
   method withdraw =
      if balance = 0 then
         raise (Failure "account is empty");
      balance <- balance - 1
end
```

Exercise 14.4 For each of the following instances of types t_1 and t_2, determine whether t_1 is a subtype of t_2—that is, whether $t_1 <: t_2$. Assume the following class declarations and relations.

Subtyping relations
`dog <: animal`
`cat <: animal`

1. `type` t_1 `= animal -> cat`
   `type` t_2 `= dog -> animal`
2. `type` t_1 `= animal ref`
   `type` t_2 `= cat ref`
3. `type 'a cl = < f : 'a -> 'a >`
   `type` t_1 `= dog cl`
   `type` t_2 `= animal cl`
4. `type 'a cl = < x : 'a ref >`
   `type` t_1 `= dog cl`
   `type` t_2 `= animal cl`
5. `type 'a c1 = < f : 'a -> unit; g : unit -> 'a >`
   `type 'a c2 = < f : 'a -> unit >`
   `type` t_1 `= animal c1`
   `type` t_2 `= cat c2`

6. `type` t_1 `= ((animal -> animal) -> animal) -> animal`
   `type` t_2 `= ((cat -> animal) -> dog) -> animal`

Exercise 14.5 Let's reimplement the narrowing example from page 172 in terms of polymorphic variants instead of exceptions. The type definitions can be given as follows.

```
type 'a animal = < actual : 'a; eat : unit >
type 'a dog = < actual : 'a; eat : unit; bark : unit >
type 'a cat = < actual : 'a; eat : unit; meow : unit >

type 'a tag = [> `Dog of 'a tag dog | `Cat of 'a tag cat ] as 'a
```

1. Implement the rest of the example, including the function `chorus`.
2. What does the type variable `'a` represent?
3. What must be changed when a new type of animals is added, say `'a lizard`, for lizards that eat but don't vocalize?

Exercise 14.6 The narrowing technique on page 172 skirts an important problem—what if the inheritance hierarchy has multiple levels? For example, we might have the following relationships.

```
hound <: dog <: animal
tabby <: cat <: animal
```

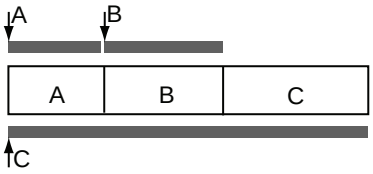
In a naïve implementation, typecases would have to be updated whenever a new tag is added. For example, the `chorus` function might require at least four cases.

```
let chorus (animals : animal list) =
   List.iter (fun animal ->
      match animal#actual with
         Dog dog -> dog#bark
       | Hound hound -> hound#bark
       | Cat cat -> cat#meow
       | Tabby tabby -> tabby#meow
       | _ -> ()) animals
```

This is undesirable of course, since the `chorus` function cares only about the general cases `dog` and `cat`.

Modify the implementation so that the method `actual` takes a list of acceptable tags as an argument. For example, for a hound `hound`, the expression `hound#actual [CatTag; DogTag]` would evaluate to `Dog hound`; but `hound#actual [HoundTag; DogTag; CatTag]` would evaluate to `Hound hound`.

Exercise 14.7 In OCaml, an object of type t_1 can be coerced to any supertype t_2, regardless of whether type t_2 has a name. This differs from some other languages. For example, in C++, an object can safely be coerced to any of its superclasses, but arbitrary supertypes are not allowed. This is mainly because objects in C++ are represented as a sequence of fields and methods (for space efficiency, methods are usually represented in a separate array called a *vtable*). For instance, if class C is a subclass of two independent classes A and B, their representations are laid out in order.



The object A is laid out first, followed by B, then any additional fields in C. A pointer to a C object is also a pointer to an A, so this coercion has no runtime cost. The coercion from C to B is also allowed with a bit of pointer arithmetic.

In OCaml, the situation is quite different. The order of methods and fields in an object doesn't matter, coercions to arbitrary supertypes are allowed, and coercions never have a runtime cost. To help understand how this works, let's build a model of objects using polymorphic variants.

Abstractly, an object is just a thing that reacts to messages that are sent to it—in other words, it is a function on method names. Given an object o with method names $m_1, m_2, \ldots, m_n$, the names are given variant labels `‘L_m1`, `‘L_m2`, ..., `‘L_mn`. The object becomes a pattern match over method labels, and the fields become let-definitions. Here is a dog object together with the corresponding model.

Object	Model
`type dog =` `  < eat : unit;` `  bark : unit;` `  chase : string -> unit` `  >` `let dog s =` `object (self)` `  val name = s` `  method eat = printf "%s eats\n" name` `  method bark = printf "%s barks\n" name` `  method chase s =` `  self#bark;` `  printf "%s chases %s\n" name s` `end`	`! type model_dog =` `!  l:[‘L_eat | ‘L_bark | ‘L_chase] ->` `!  (match l with` `!  ‘L_eat | ‘L_bark -> unit` `!  | ‘L_chase -> (string -> unit))` `let model_dog s =` `  let name = s in` `  let rec self = function` `  ‘L_eat -> printf "%s eats\n" name` `  | ‘L_bark -> printf "%s barks\n" name` `  | ‘L_chase -> (fun s ->` `  self ‘L_bark;` `  printf "%s chases %s\n" name s)` `  in` `  self`

The recursive function `self` represents the object. It takes a method label, and returns the method value. The type `model_dog` can't be defined in OCaml, because it is a *dependent* type. Informally it says that a `model_dog` is a function that takes a label `l`. If `l` is `‘L_eat` or `‘L_bark`, then the result type is `unit`. If the label is `‘L_chase`, the result type is `string -> unit`.

1. Suppose an animal object is defined as follows.

```
type animal = < eat : unit >
let animal s = object method eat = printf "%s eats\n" s end
```

   Write the model `model_animal` for an animal object.

2. Given a `model_dog` e, how is a coercion (`(`e `: model_dog :> model_animal)`) implemented?

3. How is a coercion (`(`e `: dog :> < chase : string -> unit >)`) implemented in the model?

Suppose that, instead of representing fields as individual let-definitions, the fields of an object are collected in a record, with the compiler inserting the appropriate projections. For example, here is a revised `model_dog`.

```
type dog_fields = { name : string }

let model_dog s =
   let rec self fields = function
      `L_eat -> printf "%s eats\n" fields.name
    | `L_bark -> printf "%s barks\n" fields.name
    | `L_chase -> (fun s ->
          self fields `L_bark;
          printf "%s chases %s\n" fields.name s)
   in
   self { name = s }
```

4. In this revised version, how is a functional update implemented? Explain your answer by giving the model for a new method

   ```
   method new_dog s = {< name = s >}.
   ```

5. What is the complexity of method dispatch? Meaning, given an arbitrary method label, how long does it take to perform the pattern match?

Suppose the pattern match is hoisted out of the object into a separate function `vtable`.

```
let model_dog s =
   let vtable = function
      `L_eat -> (fun self fields -> printf "%s eats\n" fields.name)
    | `L_bark -> (fun self fields -> printf "%s barks\n" fields.name)
    | `L_chase -> (fun self fields s ->
          self fields `L_bark;
          printf "%s chases %s\n" fields.name s)
   in
   let rec self fields label =
      vtable label self fields
   in
   self { name = s }
```

6. What are the advantages of the separate `vtable`? What are some disadvantages?

Chapter 15

Classes and inheritance

The simple objects that we have seen so far provide abstraction, but they provide little in the way of software re-use, which is one of the key benefits of object-oriented programming. What exactly is object-oriented programming? Mitchell [6] points out four fundamental properties.

- *Abstraction*: the details of the implementation are hidden in the object; the interface is just the set of publically-accessible methods.
- *Subtyping*: if an object `a` has all the functionality of an object `b`, then we may use `a` in any context where `b` is expected.
- *Dynamic lookup*: when a message is sent to an object, the method to be executed is determined by the implementation of the object, not by some static property of the program. In other words, different objects may react to the same message in different ways.
- *Inheritance*: the definition of one kind of object can be re-used to produce a new kind of object.

We have seen a little about first three features; we now look at inheritance. In OCaml, like many other languages, inheritance arises from classes, where a *class* is a *template* that describes how to build an object. Inheritance is the ability to create new classes (and thus new objects) from existing ones by adding, removing, and modifying methods and fields.

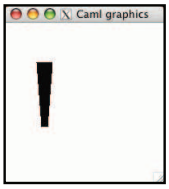
15.1 Class basics

Let's begin by defining a class. The simplest form of a class definition looks like the definition of an object, but using the keyword `class` instead of `let`. Classes are not objects in OCaml, and a class definition is not an expression. Every class definition must occur at the top level.

```
# class poly =
  object
     val vertices = [|(46, 70); (54, 70); (60, 150); (40, 150)|]
     method draw = Graphics.fill_poly vertices
  end;;
class poly : object val vertices : (int * int) array method draw : unit end
```

To create an object from a class, the keyword `new` is used with the name of the class.

```
# let p = new poly;;
val p : poly = <obj>
# p#draw;;
- : unit = ()
```



15.1.1 Class types

There are a number of things happening here, so let's look at the parts. First, we defined a class called `poly` with field `vertices` and a method `draw`. The class `poly` has a *class type* that specifies the type of its methods and fields.

```
object
   val vertices : (int * int) array
   method draw : unit
end
```

Class types are something you may not have seen before, even if you are familiar with object-oriented programming. However, class types arise naturally in languages that include both classes and modules. In OCaml, every definition that can appear in a module must have a type. A class is not a type (because it contains code), so it must have a type. Consider a module that defines the blobs of the previous chapter. The module `Blobs` contains the class definition, and the signature `BlobsSig` declares the class and its type.

```
module Blobs : BlobsSig = struct
   class poly =
   object
      val vertices = [||]
      method draw =
         Graphics.fill_poly vertices
   end

   let p = new poly
end;;
```

```
module type BlobsSig = sig
   class poly :
   object
      val vertices : (int * int) array
      method draw : unit
   end

   val p : poly
end;;
```

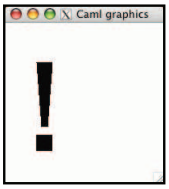
Another thing to point out is that the polygon `p` has type `val p : poly`. In this context, the class name `poly` stands for the type of polygon objects—that is, `type poly = < draw : unit >`. In general, whenever a class name appears in the context of a type expression, it stands for an object type. There is nothing special about the class name. Two classes that have methods with the same types stand for the same object type, as the following example illustrates.

```
# class gunfighter =
  object
     method draw = print_string "Bang!\n"
  end;;
class gunfighter : object method draw : unit end
# let p : gunfighter = new poly;;
val p : gunfighter = <obj>
```

15.1.2 Parameterized classes

The current `poly` class is not very useful because the vertices are fixed. If we want more than one polygon, we can define a *parameterized class*. A parameterized class definition looks like a class definition that takes arguments; the arguments are passed to `new` at object creation time.

```
# class poly vertices =
  object
     val vertices = vertices
     method draw = Graphics.fill_poly vertices
  end;;
class poly : (int * int) array ->
  object val vertices : (int * int) array method draw : unit end
# let p1 =
     new poly [|(46, 70); (54, 70); (60, 150); (40, 150)|];;
val p1 : poly = <obj>
# let p2 = new poly [|(40, 40); (60, 40); (60, 60); (40, 60)|];;
val p2 : poly = <obj>
# p1#draw; p2#draw;;
- : unit = ()
```



In OCaml, there is no specific language feature called an object constructor. Instead, the class definition serves as its only constructor, and there is only one way to construct an object from a class—by using `new`.

In a class definition, any class expression can be used as the definition. For example, the following definition specifies that the class `rectangle` is a specific kind of `poly`.

```
# class rectangle (x1, y1) (x2, x2) =
     poly [|(x1, y1); (x2, y1); (x2, y2); (x1, y2)|];;
class rectangle : float * float -> float * float -> poly
```

15.1.3 Classes with let-expressions

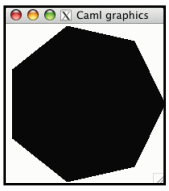
Classes can be defined with leading let-definitions, which are evaluated before a new object is created. The let-definitions have the standard form. For example, the following class defines a regular polygon with `n` sides. The vertices of the polygon are computed before the object is created.

```
class regular_poly n radius =
   let () = assert (n > 2) in
   let vertices = Array.create n (0, 0) in
```

```
  let step = 6.28 /. float_of_int n in
  let () =
     for i = 0 to n - 1 do
        let theta = float_of_int i *. step in
        let x = int_of_float (cos theta *. radius) in
        let y = int_of_float (sin theta *. radius) in
        vertices.(i) <- (x + 100, y + 100)
     done
  in
  object
     method draw = Graphics.fill_poly vertices
  end;;
```

Syntactically, each leading expression must be a let-definition, so any computations that operate by side-effect are written with a dummy let in the form `let () = ··· in`. The assertion ensures that the polygon has at least 3 sides.

```
# let p = new regular_poly 7 100.0;;
val p : regular_poly = <obj>
# p#draw;;
- : unit = ()
```



15.1.4 Type inference

If you have tried defining classes of your own, you may be running into a problem where OCaml is inferring class types that are "too polymorphic." Class types can be polymorphic, as we will see in Chapter 17, but the polymorphism must be written explicitly. Consider the following class definition, which is rejected by the compiler.

```
# class cell x =
  object
     method get = x
  end;;
...
Some type variables are unbound in this type:
  class cell : 'a -> object method get : 'a end
The method get has type 'a where 'a is unbound
```

The problem is that the argument x has polymorphic type, so the class also has a polymorphic type. We won't bother much with polymorphic classes at this point, except to say that if you really want one, then 1) you must write the type variables in square brackets before the class name, and 2) you should read Chapter 17 before you do so.

```
# class ['a] cell (x : 'a) =
  object
     method get = x
  end;;
class ['a] cell : 'a -> object method get : 'a end
```

In many cases, polymorphic classes are inadvertant. Suppose we copy the definition of the polygon object from the previous chapter.

```
# class poly vertices =
object
   val vertices = vertices
   method draw = Graphics.fill_poly (Array.map int_coord vertices)
   method transform matrix = {< vertices = Array.map matrix#transform vertices >}
end;;
...
The method transform has type
  (< transform : float * float -> float * float; .. > as 'b) -> 'a
where 'b is unbound
```

This class definition is rejected because the method `transform` takes a `matrix` that has an open, thus polymorphic, method type. We really didn't mean to write a polymorphic class, it is just that the type that was inferred is polymorphic. There are two easy solutions: constrain the type so that it is not polymorphic, or use a polymorphic method type. Constraining the type so that it is not polymorphic is easy.

```
# type coord = float * float;;
type coord = float * float
# class poly vertices =
object
   val vertices = vertices
   method draw = Graphics.fill_poly (Array.map int_coord vertices)
   method transform (matrix : < transform : coord -> coord >) =
      {< vertices = Array.map matrix#transform vertices >}
end;;
class poly : coord array -> ...
```

Using a polymorphic method type is almost as easy. The method `transform` has to be written with the type right after the method name, using an open object type `< transform : fcoord -> fcoord; .. > as 'a` for the `matrix` argument.

```
class poly vertices =
object (self : 'self)
   val vertices = vertices
   method draw = Graphics.fill_poly (Array.map int_coord vertices)
   method transform : 'a. (< transform : coord -> coord; .. > as 'a) -> 'self =
      (fun matrix -> {< vertices = Array.map matrix#transform vertices >})
end;;
```

The leading `'a.` is a type quantifier. It indicates that the type variable belongs specifically to the method `transform`, not to the class as a whole. We'll see more about polymorphic methods in Section 15.5.

15.2 Inheritance

Generally speaking, inheritance is the ability to define new classes by re-using existing ones. In the normal case, a new class is created by adding methods and fields to an existing class, or by changing its method implementations, or both. When a class B inherits from a class A, we say that B is a *subclass* of A, and A is a *superclass* of B. When B also defines a subtype of A (so that an object of class B can be used anywhere than an object of class A is expected), we say that the relationship is an

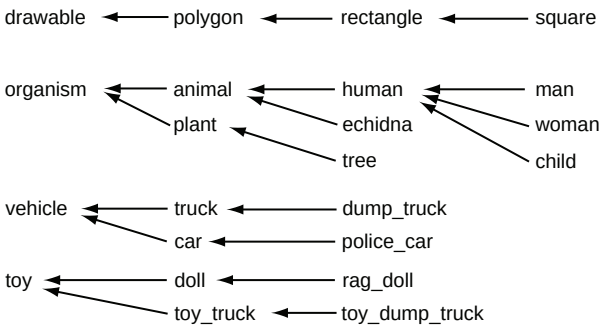
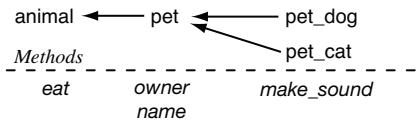


Figure 15.1: Example inheritance hierarchies.

"is-a" relationship. Is-a relationships are the most common form of inheritance, and in standard programming practice most, if not all, inheritance relationships are is-a relationships.

This notion leads to a programming model where an inheritance hierarchy is conceived in terms of the is-a relationship. Some examples are shown in Figure 15.1, where the arrows point from subclass to immediate superclass. For example the class `square` inherits from the class `rectangle` (and a square is-a rectangle).

Concretely, a class inherits from another with the directive `inherit` *class-expression*, which effectively includes the entire class *class-expression* within the current one. To illustrate, let's build a simple model of a part of the animal kingdom. The following diagram lists the class hierarchy and methods: every animal eats; a pet is an animal with an owner and a name; and a pet dog is a pet that barks.



At the root of the hierarchy is the class `animal`, which has a single method `eat`.

```
# class animal species =
  object
     method eat = Printf.printf "A %s eats.\n" species
  end;;
class animal : string -> object method eat : unit end
```

A pet is an animal with an owner and a name. The class `pet` defines methods `owner` and `name`, and it also *inherits* from the class `animal`. The effect of the inheritance is like inclusion, as if the methods of class `animal` were included in the class `pet`.

```
# class pet ~species ~owner ~name =
  object
     inherit animal species
     method owner : string = owner
     method name  : string = name
  end;;
```

```
class pet : species:string -> owner:string -> name:string ->
  object method name : string method owner : string method eat : unit end
```

The class pet_dog is a particular kind of pet that barks. Once again, the result of the inheritance is exactly like inclusion. The class pet_dog includes the methods of class pet, which in turn includes the methods of the class animal. The result is a class that includes all the methods of all the ancestor classes.

```
# class pet_dog ~owner ~name =
  object
     inherit pet ~species:"dog" ~owner ~name
     method speak = Printf.printf "%s barks!\n" name
  end;;
class pet_dog : owner:string -> name:string ->
  object
    method speak : unit
    method name : string
    method owner : string
    method eat : unit
  end
# let clifford = new pet_dog ~name:"Clifford" ~owner:"Emily";;
val clifford : pet_dog = <obj>
# clifford#speak;;
Clifford barks!
# clifford#eat;;
A dog eats.
```

15.2.1 Method override

These previous two examples of inheritance are both forms of specialization where the inheriting class includes the behavior of the superclass but does not modify it. It is also possible for a subclass to modify the behavior by redefining its methods. For example, since a pet has a name, we may wish to use the pet's name instead of the species name when it eats. We can do this by redefining the method eat.

```
# class pet ~species ~owner ~name =
object
   inherit animal species
   method owner : string = owner
   method name : string = name
   method eat = Printf.printf "%s eats.\n" name
end;;
class pet : species:string -> owner:string -> name:string ->
  object method name : string method owner : string method eat : unit end
```

Some dogs are protective about their food. We can capture this by further redefining the method eat.

```
# class pet_dog ~owner ~name =
  object (self : 'self)
     inherit pet ~species:"Dog" ~owner ~name as super
     method speak = Printf.printf "%s barks!\n" name
     method prepare_to_eat =
        Printf.printf "%s growls menacingly.\n" name
```

```
    method eat =
      self#prepare_to_eat;
      super#eat
  end;;
class pet_dog : ...
# let clifford = new pet_dog ~owner:"Emily" ~name:"Clifford";;
val clifford : pet_dog = <obj>
# clifford#eat;;
Clifford growls menacingly.
Clifford eats.
```

The syntax `inherit` *class-expression* `as` *identifier* gives a name to the superclass, which allows the superclass methods to be invoked. This is useful mainly when the superclass's methods are overridden in the subclass. In this case, the method `eat` in the class `pet_dog` is a kind of "wrapper" method. It overrides the `pet`'s `eat` method by first preparing to eat, then calling the pet-eat method with the expression `super#eat`.

15.2.2 Class types

The design of the inheritance hierarchy can have a major impact on the ease of programming a system. This is particularly true for languages based on nominal typing, like C++ or Java, where subtyping is constrained by the hierarchy. For example, in a nominal type scheme, the type of an object corresponds to the class name, so a `pet_dog` can be coerced to a `pet`, and then to an `animal`, but no other coercions are allowed.

This can be overly restrictive of course. As the design proceeds we might discover we need new classes. For the animal example, we might want a class `farm_animal`, where animals have owners but might not have names; or a class `vocal_animal` for animals that can vocalize, but might be wild. A `dog` can be a `pet`, a `farm_animal`, and also a `vocal_animal`.

In the worst case, the inheritance hierarchy can become complicated because the number of feature sets is combinatorial. Languages with interfaces, like Java, try to combat this problem by allowing a class to satisfy multiple interface definitions.

```
interface farm_animal { void eat(); String owner(); };
interface vocal_animal { void eat(); void speak(); }
class dog extends pet implements farm_animal, vocal_animal { ··· }
```

One drawback of this approach is that the interfaces must be specified at class definition time, requiring the designer to predict what the useful interfaces will be.

In OCaml, the situation is different. The inheritance hierarchy constrains the way in which classes are specified and implemented, but it has no effect on subtyping. Here is how we could extend the animal example to support farm animals and vocal animals.

```
# class type farm_animal =
  object
    inherit animal
    method owner : string
  end;;
class type farm_animal = object method owner : string method eat : unit end
# class type vocal_animal =
  object
```

```
    inherit animal
    method speak : unit
  end;;
class type vocal_animal = object method speak : unit method eat : unit end
# (clifford : pet_dog :> farm_animal);;
- : farm_animal = <obj>
```

Note that in the `inherit` clause, the class type does not take arguments. In this context, the name `animal` stands for the class type, and the `inherit` directive again acts as textual inclusion.

The main advantage of structural subtyping is that an object can be coerced to any compatible type, the types do not have to be specified ahead of time. The disadvantage of course, is that subtyping may be overly permissive; some coercions might not make sense semantically.

```
# class cat ~owner ~name =
  object
    inherit pet ~species:"cat" ~owner ~name
    method speak = Printf.printf "%s meows.\n" name
  end;;
class cat : owner:string -> name:string ->
  object
    method speak : unit
    method name : string
    method owner : string
    method eat : unit
  end
# let my_cat = (clifford :> cat);;
val my_cat : cat = <obj>
# my_cat#speak;;
Clifford barks!
```

15.2.3 Class type constraints and hiding

The type of a class can be constrained with the following syntax, where *class-type* is a class type.

class *class-name parameter*$_1$ $\cdots$ *parameter*$_n$: *class-type* = *class-expression*

It is also possible to place constraints on the object type, with an explicit constraint of the form `constraint 'self =` *object-type*, or equivalently using the syntax *object-type* `as 'self`.

```
class cat ~name ~owner =
object (self : 'self)
   constraint 'self = < eat : unit; speak : unit; .. >
   inherit pet ~name ~owner
   method speak = Printf.printf "%s meows.\n" name
end
```

As a programmer, you might want to write a type constraint to ensure that your implementation matches a specified interface. However, there are other reasons for using a type constraint. Unlike the rest of OCaml, type constraints on classes can change how

the program behaves. Here is what you can (and cannot) do with a constraint.

1. A constraint *cannot* be used hide public methods.
2. A constraint can be used to turn a private method into a public method.
3. A constraint can be used to hide fields and private methods.

The restriction on public methods is the same as it is for object types. If an object has a public method, the method must appear in the type.

```
# (object method x = 1 method y = 2 end : < x : int >);;
...
This expression has type < x : int; y : int > but is here used with type
  < x : int >
The second object type has no method y
```

The second property, where private methods are made public, may be a little surprising. It is often used in cases where a superclass defines a private method that a subclass would like to be made public.

```
# class foo =
  object (self : 'self)
     constraint 'self = < x : int; .. >
     method private x = 1
  end;;
class foo : object method x : int end
```

The third kind of constraint, used to hide field and private methods, requires some discussion. When a class hides a field, or a private method, that field becomes inaccessible to subclasses, and it also means that the field or method cannot be overridden. Consider the following class definitions.

```
# class a =
object (self)
   method private f_private = print_string "aaaa\n"
   method test_a = self#f_private
end;;
# class b =
object (self)
   inherit a
   method private f_private = print_string "bbbb\n"
   method test_b =
      self#test_a;
      self#f_private
end;;
# (new b)#test_b;;
bbbb
bbbb
```

The method `f_private` is private, but it is the same method in both classes. The definition of `f_private` in class `b` *overrides* the definition in class `a`, so the result is to print `bbbb` twice. Now let's consider what happens when `f_private` is hidden by a type constraint.

```
# class type a_type = object method test_a : unit end;;
# class a : a_type =
  object (self)
     method private f_private = print_string "aaaa\n"
     method test_a = self#f_private
  end;;
# class b = ···same as before···;;
# (new b)#test_b;;
aaaa
bbbb
```

In this case, the result is different. By hiding the method `f_private` in class `a`, it can't be used or overridden in subclass `b`, and the behavior of the program is changed.

The property is the same for fields. When a field is hidden y a type constraint, it is no longer accessible to subclasses. New field definitions with the same name in subclasses are independent.

15.2.4 Classes and class types as object types

As we have mentioned, a class is not a type, but the class name stands for an object type when used in the context of a type expression. The same holds for class types—a class type is the type of a class, but it also stands for the type of an object when used in context of a type expression. The object type is formed from a class type by omitting the field types. The following table lists some examples of equivalent types.

Class type	Object type	Simplified
`class type t1 =` `object` `  val x : int` `  method y : int` `end`	`type t1 = < y : int >`	
`class type t2a =` `object ('self)` `  val x : int` `  method f1 : t2a` `  method f2 : 'self` `end`	`type t2a =` `  < f1 : t2a;` `  f2 : 'self` `  > as 'self`	`type t2a =` `  < f1 : t2a;` `  f2 : t2a` `  >`
`class type t2b =` `object ('self)` `  inherit t2a` `  method f3 : unit` `end`	`type t2b =` `  < f1 : t2a;` `  f2 : 'self;` `  f3 : unit` `  > as 'self`	`type t2b =` `  < f1 : t2a;` `  f2 : t2b;` `  f3 : unit` ` >

The role of `'self` in these types is subtle and important. In class type `t2a`, the method `f1` returns an object of type `t2a`, which means that it is an object with exactly two methods, `f1` and `f2`. The method `f2` returns an object of the same type as `self`—which, for an object of type `t2a`, has the same methods `f1` and `f2`.

However, when the subclass `t2b` is formed, a new method `f3` is added. The method `f1` returns an object of type `t2a`, having two methods `f1` and `f2`; but the method `f2` returns an object of type `'self`, now having three methods `f1`, `f2`, and `f3`.

There are actually two kinds of object types that are formed from class names: exact and open types. The type expression *class-name* stands for the type of objects having exactly the methods of the class *class-name* and the type expression *#class-name* stands for objects having those methods, and possibly more. In other words, a type expression *#class-name* stands for an open object type.

Type expression	Object type
`t1`	`< y : int >`
`#t1`	`< y : int; .. >`
`t2a`	`< f1 : t2a; f2 : 'self > as 'self`
`#t2a`	`< f1 : t2a; f2 : 'self; .. > as 'self`
`#t2b`	`< f1 : t2a; f2 : 'self; f3 : unit; .. > as 'self`

Just like an open object type, a type expression *#class-name* is polymorphic.

```
# type s2a = #t2a;;
Characters 4-15:
  type s2a = #t2a;;
      ^^^^^^^^^^^
A type variable is unbound in this type declaration.
In definition #t2a as 'a the variable 'a is unbound
# type 'a s2a = #t2a as 'a;;
type 'a s2a = 'a constraint 'a = #t2a
```

15.3 Inheritance is not subtyping

Let's summarize the operations that can be performed through inheritance. One may:

- add new fields and new private methods,
- add new public methods,
- override fields or methods, but the type can't be changed.

Fields and private methods don't appear in the object type, and method override isn't allowed to change the method's type. So from a typing perspective, if a class B is a subclass of A, it might have more methods than A, but everything else is unchanged. By width subtyping (Section 14.8.2), this will usually mean that the object type for B is a subtype of the object type for A. In fact, in many languages, subclassing and subtyping are the same, and it isn't possible to have one without the other.

Unfortunately, the correspondence is sound only in the case where the object type is covariant in `'self`; in other words, when the object has no binary methods. Consider a class type `comparable` for objects that can be compared.

```
(* less-than: negative; equal: zero; greater-than: positive *)
type comparison = int

class type comparable =
object ('self)
   method compare : 'self -> comparison
end
```

The method `compare` is a binary method; it takes another object of type `'self` and performs a comparison. The implementations use the usual technique of adding a method `representation` to expose the internal representation so that the comparison can be implemented.[1]

```
class int_comparable (i : int) =
object
   method representation = i
   method compare (j : 'self) =
      i - j#representation
end
```

```
class string_comparable (s : string) =
object
   method representation = s
   method compare (s2 : 'self) =
      String.compare s s2#representation
end
```

We might later decide to implement some subclasses for printable, comparable objects.

```
class int_print_comparable i =
object (_ : 'self)
   inherit int_comparable i
   method print = print_int i
end
```

```
class string_print_comparable s =
object (_ : 'self)
   inherit string_comparable s
   method print = print_string s
end
```

We might expect that a printable and comparable integer is also a comparable integer. However, the expected coercions fail.

```
# (new int_comparable 1 :> comparable);;
... This expression cannot be coerced ...
# (new int_print_comparable 1 :> int_comparable);;
... This expression cannot be coerced ...
```

For an intuitive explanation, consider what would happen if the subtype relation `int_comparable <: comparable` held. The relation `string_comparable <: comparable` would hold as well, allowing us to perform an unsound comparison of strings and integers.

```
(* !!FAKE--THESE OPERATIONS ARE UNSOUND!! *)
# let i = (new int_comparable 1 :> comparable);;
i : comparable = < obj >
# let s = (new string_comparable "Hello" :> comparable);;
s : comparable = < obj >
# i#compare s;;
???
```

The real problem is the the method `compare` is *contravariant* in the type `'self` because it takes a value of type `'self` as an argument. This is the opposite of what it normally is. Here are the object types that correspond to the classes and class types.

```
comparable = < compare : 'self -> bool > as 'self
```

[1] Note that subtraction can only be used to compare small integers.

```
int_comparable =
   < representation : int;
     compare : 'self -> bool > as 'self
int_print_comparable =
   < representation : int;
     compare : 'self -> bool;
     print : unit > as 'self
```

Let's try to carry out a proof of the subtyping relation `int_comparable <: comparable` to see where it goes wrong. The type `'self` in each case corresponds to the type name, so we are try to show the following.

```
< representation : int; compare : int_comparable -> bool >
<: < compare : comparable -> bool >
```

Attempted proof:

- The types are recursive, so we first assume that subtyping holds on the type names `int_comparable <: comparable`.
- By width subtyping, the method `representation` can be dropped. Show:

  ```
  < compare : int_comparable -> bool > <: < compare : comparable -> bool >
  ```

- By depth subtyping, show:

  ```
  (int_comparable -> bool) <: (comparable -> bool)
  ```

- By function subtyping, show:

  ```
  comparable <: int_comparable
  ```

- This is the opposite of the assumption, so the proof fails.

Of course, this doesn't mean that we can't write generic functions of comparable objects, it simply means that the functions should use the open type `#comparable`, not the exact non-polymorphic type `comparable`.

```
# let sort (l : #comparable list) = List.sort (fun e1 e2 -> e1#compare e2) l;;
val sort : (#comparable as 'a) list -> 'a list = <fun>
# let l = List.map (new int_print_comparable) [9; 1; 6; 4];;
val l : int_print_comparable list = [<obj>; <obj>; <obj>; <obj>]
# List.iter (fun i -> i#print) (sort l);;
1469
```

We might be willing to accept the fact that there are no useful objects with exact type `comparable`, but what about the relation between the types `int_print_comparable` and `int_comparable`? If it seems that the subtyping relation should hold, the solution is to avoid the use of binary methods. The method `compare` doesn't really need an argument of type `'self`, it just needs an object with the same representation. The classes can be reimplemented to use a constrained argument type.

```
class int_comparable (i : int) =
object
   method representation = i
   method compare : 'a. (< representation : int; .. > as 'a) -> comparison =
      (fun j -> Pervasives.compare i j#representation)
end

class int_print_comparable i =
object
   inherit int_comparable i
   method print = print_int i
end
```

It now holds that `int_print_comparable` is a subtype of `int_comparable`, but there are several drawbacks. One is that the new objects no longer have type `#comparable`, so it isn't possible to write truly generic functions. Another problem is that the types get quickly complicated. Still, with some effort, we get what we want.

```
# let compare_int (e1 : #int_comparable) (e2 : #int_comparable) =
     e1#compare e2;;
val compare_int : #int_comparable -> #int_comparable -> comparison = <fun>
# let sort_int (l : #int_comparable list) =
     List.sort compare_int l;;
val sort_int : (#int_comparable as 'a) list -> 'a list = <fun>
# let l = List.map (new int_print_comparable) [9; 1; 3; 2];;
val l : int_print_comparable list = [<obj>; <obj>; <obj>; <obj>]
# List.iter (fun i -> i#print) (sort_int l);;
1239
```

15.4 Modules and classes

OCaml includes two major systems for modularity and abstraction: the module system and the object system. In many respects, the two systems are quite similar. Both provide mechanisms for abstraction and encapsulation, for subtyping (by omitting methods in objects, and omitting fields in modules), and for inheritance (objects use `inherit`; modules use `include`, Section 12.4). However, the two systems are not comparable. On the one hand, objects have an advantage: objects are first-class values, and modules are not—in other words, modules do not support dynamic lookup. On the other hand, modules have an advantage: modules can contain type definitions, and objects cannot.

We have already suggested that modules are handy for hiding internal representations in the definition of binary methods. For example, let's use the module to hide the representation of a comparable object.

```
module type IntComparableSig =
sig
   type rep

   class int_comparable : int ->
   object ('self)
      method representation : rep
```

```
      method compare : 'self -> comparison
   end
end

module IntComparable : IntComparableSig =
struct
   type rep = int

   class int_comparable (i : int) =
   object (_ : 'self)
      method representation = i
      method compare (j : 'self) =
         Pervasives.compare i j#representation
   end
end
```

This method of hiding the representation also has the side-effect that it is no longer possible to construct an object of type `int_comparable` without using the class `int_comparable`. We can use this to our advantage. In the animal example, if we wish to ensure that dogs cannot be turned into cats, we can give the cats a "certificate" that "justifies" their authenticity.

```
module type CatSig =
sig
   type cert

   class cat : owner:string -> name:string ->
   object
      inherit pet
      method cert : cert
      method speak : unit
   end
end

module Cat : CatSig =
struct
   type cert = unit

   class cat ~owner ~name =
   object
       inherit pet ~species:"cat" ~owner ~name
       method cert = ()
       method speak = Printf.printf "%s meows.\n" name
   end
end
```

It doesn't matter what the certificate is, just that it is abstract. Of course, this technique mainly prevents accidental coercions. If a dog really wants to become a cat, it can take the certificate from one of them.

Functors can also be useful in class construction. For example, there are many specific kinds of comparable objects, integers, floating-point values, strings, pairs of `comparable` objects, *etc*. All we need to build one of these is a function to compare the values. The generic class is described the usual way.

```
(* less-than: negative; equal: zero; greater-than: positive *)
```

```
type comparison = int

class type comparable =
object ('self)
   method compare : 'self -> comparison
end;;
```

To build a specific kind of comparable items, we need a function to compare the values. We can build a generic module as a functor that takes the type of items and the comparison as an argument, and produces a new class of comparable items.

```
module type CompareSig =
sig
   type t
   val compare : t -> t -> comparison
end;;

module type ComparableSig =
sig
   type t
   type rep

   class element : t ->
   object ('self)
       method representation : rep
       method compare : 'self -> comparison
   end
end;;

module MakeComparable (Compare : CompareSig)
: ComparableSig with type t = Compare.t =
struct
   type t = Compare.t
   type rep = Compare.t

   class element (x : t) =
   object (self : 'self)
      method representation = x
      method compare (item : 'self) =
         Compare.compare x item#representation
   end
end;;
```

The representation type `rep` is abstract, but the type of element values `t` is not. The method `compare` simply uses the provided function `Compare.compare` to compare the representations.

15.5 Polymorphic methods and virtual classes

Let's turn to another example of object-oriented programming, this time more computational. A *collection*, in general, is like a set of elements. There are many kinds of collections, some based on implementations like arrays or lists, and others based on access patterns, like stacks and queues. Figure 15.2 shows a partial inheritance

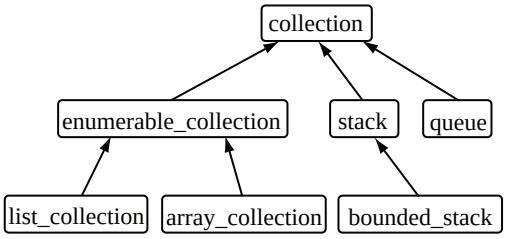


Figure 15.2: Hierarchy of collections

hierarchy.

It won't matter much what the elements are, so for now we'll just assume the elements can be printed. It might also be useful to build polymorphic collections, but we'll leave that topic for the Chapter 17.

```
class type element = object method print : unit end
```

At the root of the inheritance hierarchy is the type `collection`, which represents the features that all collections have in common. What are those features? It can't be implementation or access pattern, because those features change for different kinds of collections. In fact, the type `collection` by itself is pretty useless. We'll assume it has a method `length` that returns the number of elements in the collection.

```
class type collection =
object
   method length : int
end
```

At the middle of the hierarchy is the "`enumerable_collection`," which represents collections where the elements can be enumerated, using functions like `iter` to iterate over all the elements, or `fold` to compute a value from all the elements. The method types correspond to the function types in the standard libraries `List` and `Array`. We still haven't implemented a concrete collection, so this is still a class type.

```
class type enumerable_collection =
object
   inherit collection
   method iter : (element -> unit) -> unit
   method fold : ('a -> element -> 'a) -> 'a -> 'a
end;;
```

15.5.1 Polymorphic methods

Next, `list` and `array` are two concrete kinds of collections that differ mainly in how they are accessed; a `list` is a stack, while an `array` allows random access. However, when we try to provide a concrete list implementation, we encounter an error because the method `fold` is polymorphic.

```
# class broken_list_collection =
```

```
  object
     ...
     method fold f x = List.fold_left f x elements
  end;;
Characters 5-328: ...
Some type variables are unbound in this type:
...
The method fold has type ('a -> element -> 'a) -> 'a -> 'a where 'a
is unbound
```

The problem here is that OCaml assumes that the *class* must be polymorphic, not the *method* (which is what we intend). OCaml allows methods to be polymorphic, but they must be annotated explicitly using the following syntax.

method *identifier* : '*type-variable* ··· '*type-variable*. *type* = *expression*

The type expression '*type-variable* ··· '*type-variable*. *type* is called a *universally quantified type*, and the type variables before the . are bound specifically in the method type *type*, not for the class as a whole. Note that the type constraint is required to occur right after the method name, so functions must be written explicitly. Here is a corrected definition, which is now accepted by the top-loop. Note that in the class type, the type variable is not required.

```
# class list_collection =
  object
     val mutable elements : element list = []
     method length = List.length elements

     method add x = elements <- x :: elements
     method remove = elements <- List.tl elements

     method iter f = List.iter f elements
     method fold : 'a. ('a -> element -> 'a) -> 'a -> 'a =
        (fun f x -> List.fold_left f x elements)
  end;;
class list_collection :
object
   ...
   method fold : ('a -> element -> 'a) -> 'a -> 'a
end
```

Continuing with our example, the `array_collection` has a similar definition. Some differences are that the size of the array is determined at instantiation time, and the `Array` module doesn't provide a `fold` function, so we have to code it manually.

```
class array_collection size init =
object
   val elements = Array.create size init
   method length = size
   method set i x = elements.(i) <- x
   method get i = elements.(i)
   method iter f = Array.iter f elements
   method fold : 'a. ('a -> element -> 'a) -> 'a -> 'a =
      (fun f x ->
         let rec loop i x =
```

```
            if i = size then x else loop (i + 1) (f x elements.(i))
          in
          loop 0 x)
end;;
```

15.5.2 Virtual (abstract) classes and methods

At this point, we have two class types, `collection` and `enumerable_collection`; and two concrete classes, `list_collection` and `array_collection`. The reason for using class types for the former is because there are no actual instances of `collection` or `enumerable_collection`; the only actual instances are of the subclasses.

Now, suppose we wanted to be able to print a collection. We could implement a method `print` for each of the concrete classes `list` and `array`, but in fact the implementations can be written the same way.

```
method print = self#iter (fun element -> element#print)
```

A better way to implement it is to "lift" the implementation into the `enumerable_collection` superclass. The problem is that `enumerable_collection` is a class type, not a class, so it can't contain code.

The solution is to define `enumerable_collection` as a *virtual class*, which is a class where some or all of the method implementations are omitted. In our example, the methods `iter` and `fold` are to be implemented in subclasses, so they are omitted, but the method `print` can be implemented. A method where the implementation is omitted is called a *virtual method*, and it is declared with the syntax

```
method virtual identifier : type.
```

Any class that contains a virtual method must also be declared virtual, with the syntax `class virtual`. For completeness in our example, we also declare the class `collection` as virtual, so that it can be inherited by the `collection` class.

```
class virtual collection =
object
   method virtual length : int
end;;

class virtual enumerable_collection =
object (self : 'self)
   inherit collection
   method virtual iter : (element -> unit) -> unit
   method virtual fold : 'a. ('a -> element -> 'a) -> 'a -> 'a
   method print = self#iter (fun element -> element#print)
end;;
```

Virtual classes cannot be instantiated. However, once the methods have been implemented (by a subclass), the virtual status of a class can be removed.

```
# class list_collection =
  object
     inherit enumerable_collection
```

```
    val mutable elements : element list = []
    method length = List.length elements
    method add x = elements <- x :: elements
    method remove = elements <- List.tl elements
    method iter f = List.iter f elements
    method fold : 'a. ('a -> element -> 'a) -> 'a -> 'a =
      (fun f x -> List.fold_left f x elements)
  end;;
class list_collection :
  object
    val mutable elements : element list
    method add : element -> unit
    method fold : ('a -> element -> 'a) -> 'a -> 'a
    method iter : (element -> unit) -> unit
    method length : int
    method print : unit
    method remove : unit
  end
```

15.5.3 Terminology

Classes with omitted implementations are a standard feature of many object-oriented languages. In mainstream terminology, they are called *abstract* classes, and a "virtual method" is a method that is resolved using dynamic lookup (as opposed to a "static method" that uses static lookup).

The non-standard terminology OCaml uses can be quite confusing, especially to those not familiar with OCaml, or those just learning the language. However, there is a good argument that OCaml uses the correct terms, and mainstream terminology is inaccurate. Here are selected definitions of the terms "abstract" and "virtual," taken from the American-Heritage dictionary of the English Language [1].

ab·stract . . . 4. Thought of or stated without reference to a specific instance . . .

vir·tu·al 1. Existing or resulting in essence or effect though not in actual fact, form, or name. . .

Put more loosely, if something is abstract, it means it exists but it is not entirely specified or defined; if something is virtual, it appears to exist, but doesn't in actual fact. In this sense, virtual is the more appropriate term, because a virtual class can be used in all ways like a normal class—except for one: it can't be instantiated because it doesn't fully exist.

15.5.4 Stacks

Returning to our example, let's move back up the hierarchy, and consider the `stack` class. Stacks are defined by their behavior: elements are pushed onto the top of the stack, and taken from the top of the stack, in last-in-first-out (LIFO) order. A generic stack is a virtual class with two virtual methods: `push : element -> unit` pushes an

element onto the top of the stack, and `pop : element` removes and returns the top element. In addition, we include two derived methods: `dup : unit` duplicates to topmost element of the stack, and `swap : unit` swaps the top two elements.

```
class virtual stack =
object (self : 'self)
   inherit collection
   method virtual push : element -> unit
   method virtual pop  : element
   method dup =
      let x = self#pop in
      self#push x;
      self#push x
   method swap =
      let x1 = self#pop in
      let x2 = self#pop in
      self#push x1;
      self#push x2
end;;
```

Let's implement a real subclass `bounded_stack` in terms of arrays. The `bounded_stack` inherits from the virtual class `stack`, and implements the methods `push` and `pop` in terms of array operations.

```
class bounded_stack size =
let dummy = object method print = () end in
object
   inherit stack

   val data = new array_collection size dummy
   val mutable index = 0

   method push x =
      if index = size then
         raise (Failure "stack is full");
      data#set index x;
      index <- index + 1
   method pop =
      if index = 0 then
         raise (Failure "stack is empty");
      index <- index - 1;
      data#get index
   method length = data#length
end;;
```

The class `bounded_stack` stores its values in an array, initialized to a dummy value. The method `push` adds an element if there is room, and the method `pop` returns an element if the stack is nonempty. The methods `dup` and `swap` are implemented by the superclass `stack`.

15.5.5 Lists and stacks

The class `bounded_stack` demonstrates two relationships: it *is-a* stack, so it inherits from the class `stack`; and it *has-a* array, which it includes as a field that it uses to

implement the virtual methods needed to implement a `stack`.

Another way to build a stack is in terms of a list, but in this case the stack and list are so similar, we might want to say that a stack *is-a* list, and inherit from the class `list_collection` directly. OCaml supports multiple inheritance; we simply `inherit` from each superclass.

```
# class unbounded_stack =
  object (self : 'self)
     inherit list_collection
     inherit stack

     method push x = self#add x
     method pop =
        let x = self#head in
        self#remove;
        x
  end;;
class unbounded_stack :
  object
    val mutable elements : element list
    method add : element -> unit
    method dup : unit
    method fold : ('a -> element -> 'a) -> 'a -> 'a
    ...
  end
```

The resulting class has the methods and fields of both superclasses, so in addition to being a stack, it is also an `enumerable_collection` (and a `list_collection`).

Is this construction appropriate? It depends on whether it is acceptable to view the stack as a list. For example, the class `unbounded_stack` has two methods to add an element to the stack: `push` and `add`, and they are the same (`push` calls `add`). If it is acceptable for subclasses to override one of the methods and not the other, then the multiple inheritance is acceptable; otherwise it is not. Certainly, it is not appropriate for the `bounded_collection` to inherit from the `array_collection` because the array's unrestricted `set` and `get` operations are not appropriate for a stack. We'll see more about multiple inheritance in the next chapter.

15.6 Exercises

Exercise 15.1 What are the class types for the following classes?

1. ```
   class c1 =
   object
      val x = 1
      method get = x
   end
   ```
2. ```
   class c2 =
   object
      method copy = {< >}
   end
   ```
3. ```
   class c3 y =
   object (self1)
      method f x =
         object (self2)
            val x = x
            method h = self1#g + x
         end
      method g = y
   end
   ```
4. ```
   class c4 =
   object (self : < x : int; .. > as 'self)
      method private x = 1
   end
   ```

Exercise 15.2 What does the following program print out?

```
class a (i : int) =
let () = print_string "A let\n" in
object
   initializer print_string "A init\n"
end;;

class b (i : int) =
let () = print_string "B let\n" in
object
   inherit a i
   initializer print_string "B init\n"
end;;

new b 0;;
```

Exercise 15.3 Normally, we would consider a square to be a subtype of rectangle. Consider the following class `square` that implements a square,

```
class square x y w =
object
   val x = x
   val y = y
   method area = w * w
   method draw = Graphics.fill_rect x y w w
   method move dx dy = {< x = x + dx; y = y + dy >}
end
```

Write a class `rectangle` that implements a rectangle by inheriting from `square`. Is it appropriate to say that a `rectangle` is a `square`?

Exercise 15.4 A mutable list of integers can be represented in object-oriented form with the following class type.

```
class type int_list =
object
    method is_nil : bool
    method hd : int
    method tl : int_list
    method set_hd : int -> unit
    method set_tl : int_list -> unit
end
```

1. Define classes `nil` and `cons` that implement the usual list constructors.

   ```
   class nil : int_list
   class cons : int -> int_list -> int_list
   ```

2. The class type `int_list` is a recursive type. Can it be generalized to the following type?

   ```
   class type gen_int_list =
   object ('self)
       method is_nil : bool
       method hd : int
       method tl : 'self
       method set_hd : int -> unit
       method set_tl : 'self -> unit
   end
   ```

3. The class type `int_list` should also include the usual list functions.

   ```
   class type int_list =
   object
       method is_nil : bool
       method hd : int
       method tl : int_list
       method set_hd : int -> unit
       method set_tl : int_list -> unit
       method iter : (int -> unit) -> unit
       method map  : (int -> int) -> int_list
       method fold : 'a. ('a -> int -> 'a) -> 'a -> 'a
   end
   ```

   Implement the methods `iter`, `map`, and `fold` for the classes `nil` and `cons`.

Exercise 15.5 Consider the following definition of a stack of integers, implemented using the imperative lists of Exercise 15.4.

```
class int_stack =
object
```

```
    val mutable items = new nil
    method add x = items <- new cons x items
    method take =
        let i = items#hd in
        items <- items#tl;
        i
end
```

1. Define a class `int_queue` that implements a queue, by inheriting from the class `int_stack`, without overriding the method `take`.

2. Is it appropriate to say that a queue is-a stack?

Exercise 15.6 The following type definition uses polymorphic variants to specify an open type for simple arithmetic expressions with variables.

```
type 'a exp =
 [> `Int of int
  | `Var of string
  | `Add of 'a exp * 'a exp
  | `Sub of 'a exp * 'a exp ] as 'a
```

1. Build an object-oriented version of expressions, where class type exp includes an evaluator that computes the value of the expression.

   ```
   class type env =
   object ('self)
      method add : string -> int -> 'self
      method find : string -> int
   end

   class type exp =
   object
      method eval : 'a. (#env as 'a) -> int
   end
   ```

   The classes should have the following types.

   ```
   class int_exp : int -> exp
   class var_exp : string -> exp
   class add_exp : #exp -> #exp -> exp
   class sub_exp : #exp -> #exp -> exp
   ```

2. Implement a new kind of expression `` `Let of string * exp * exp ``, where `` `Let (v, e1, e2) `` represents a let-expression `let v = e1 in e2`.

3. Suppose that, in addition to being able to evaluate an expression, we wish to check whether it is *closed*, meaning that it has no undefined variables. For the polymorphic variant form, the definition can be expressed concisely.

   ```
   let rec closed defined_vars = function
      `Int _ -> true
    | `Var v -> List.mem v defined_vars
   ```

```
| `Add (e1, e2)
| `Sub (e1, e2) -> closed defined_vars e1 && closed defined_vars e2
| `Let (v, e1, e2) ->
    closed defined_vars e1 && closed (v :: defined_vars) e2
```

Implement a method `closed : bool` for the expression classes. Any new classes should be defined by inheriting from the existing ones. How many new classes need to be defined?

Exercise 15.7 Object-oriented programming originated in the Simula, a language designed by Dahl and Nygaard [7] for the purpose of simulation. In this exercise, we'll build a simple circuit simulator using objects in OCaml.

A logic circuit is constructed from *gates* and *wires*. A gate has one or more inputs and an output that is a computed as a Boolean function of the inputs. A wire connects the output of a gate to one or more input *terminals*, where a terminal has a method `set : bool -> unit` to set the value of the terminal. Here are the definitions of the classes `terminal` and `wire`.

```
type terminal = < set : bool -> unit >

class wire =
object
   val mutable terminals : terminal list = []
   val mutable value = false
   method add_terminal t = terminals <- t :: terminals
   method set x =
      if x <> value then (value <- x; List.iter (fun t -> t#set x) terminals)
end

let dummy_wire = new wire
```

There are many kinds of gates, so we'll build an inheritance hierarchy. A generic gate has a single output, connected to a wire. It also has a virtual method `compute_value` that defines the function computed by the gate.

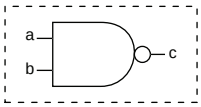
```
class virtual gate =
object (self : 'self)
   val mutable output_wire = dummy_wire
   method connect_output wire = output_wire <- wire
   method private set_output =  output_wire#set self#compute_value
   method private virtual compute_value : unit -> bool
end
```

A `two_input_gate` is a gate that has two inputs.

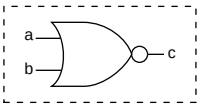
```
class virtual two_input_gate =
object (self : 'self)
   inherit gate
   val mutable a = false
   val mutable b = false
   method private set_input_a x = a <- x; self#set_output
   method private set_input_b x = b <- x; self#set_output
   method connect_input_a wire = ···
   method connect_input_b wire = ···
```

```
    end
```

With the boilerplate defined, we can build some standard gates.



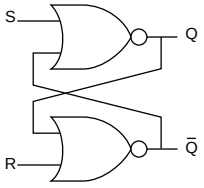

```
class nand2 =
object
   inherit two_input_gate
   method compute_value = not (a && b)
end
```



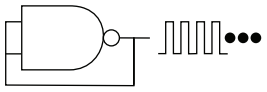

```
class nor2 =
object
   inherit two_input_gate
   method compute_value = not (a || b)
end
```

1. Fill in the definitions of the methods `connect_input_a` and `connect_input_b`.
2. Define a class `three_input_gate` (for gates with three inputs) by inheriting from `two_input_gate`.
3. Would the definition be simpler if the type `terminal` were a function instead of an object (where `type terminal = bool -> unit`)?
4. What is the purpose of the conditional `if x <> value then` $\cdots$ in the class `wire`?
5. Write a program for the following circuit, called a *SR latch*.

S	R	Action
0	0	Keep state
0	1	$Q = 0$
1	0	$Q = 1$
1	1	$Q = 0, \overline{Q} = 0$



Exercise 15.8 The simulator in Exercise 15.7 has a problem with some cyclic circuits. For example, the following circuit, called a *ring oscillator*, oscillates indefinitely, overflowing the stack during simulation.



The simulation can be executed in constant stack space by implementing an *event-driven simulator*. In the circuit context, an *event* occurs whenever the value on a terminal is set. An event-driven simulator uses a scheduler to manage events. When the value of a terminal is set, the terminal is scheduled, but not executed yet. When scheduled, the terminal is removed from the scheduling queue and executed.

Define an event driven simulator by implementing a scheduler. You can use a scheduling policy of your choice, but it should be fair, meaning that if a terminal is scheduled, it will eventually be executed.

The scheduler should include a method `main : unit` that runs until there are no more events (perhaps forever). The type `terminal` should be defined as follows. The method `set` schedules the terminal, and the method `execute` executes it.

```
type terminal = < set : bool -> unit; execute : unit >
```


Chapter 16

Multiple inheritance

Multiple inheritance is the ability to inherit from more than one superclass. In OCaml, the mechanism is simple, any class that contains more than one `inherit` directive inherits from each of them.

Object-oriented programming has received much attention over the years, and multiple inheritance is one of the more controversial areas. Some claims against it are that it is complicated, that it requires extensive training to understand, or that all of its useful features can be captured with interfaces. Many of these claims are baloney—in fact, multiple inheritance is often one of the simplest and most elegant way to combine features and abstractions. However, there are two main issues that give rise to these claims:

- *shadowing*: what happens when two ancestor classes define a method with the same name?
- *repeated inheritance*: what happens when a class is inherited more than once?

As we will see, OCaml's model of inheritance is quite simple. It is essentially equivalent to textual inclusion, and shadowing follows the normal rule: if a method is defined more than once, the last definition wins. First though, let's turn to some examples.

16.1 Examples of multiple inheritance

Multiple inheritance arises whenever an object is described as a collection of features. Examples in the real world abound.

- A *clock radio* is a *timepiece* and a *radio*.
- An *ambulance* is a *vehicle* and an *emergency medical facility*.
- A *mobile home* is a *vehicle* and a *home*.
- A *spork* is a *spoon* and a *fork*.

- A *graduate student* is a *graduate* and a *student*.
- A *Swiss army knife* is a *knife* and *scissors* and *pliers* and a *screwdriver* and...

If we consider these examples, there are really two kinds of combinations: those of independent features like *clock* and *radio*; and those of related features. For example, spoons and forks are both utensils, graduates and students are kinds of persons, *etc*.

16.1.1 Inheriting from multiple independent classes

From a programming perspective, inheritance from independent classes is simpler. When the classes to be combined use disjoint names, the result of combining them is simply to produce an object with all the parts. For example, consider the following sketches of the classes `clock` and `radio`.

```
class clock =
object
   val mutable now = ···
   method gettimeofday = now
   method private tick = ···
end;;

class radio =
object
   val mutable frequency = 89.3e6
   val mutable volume = 11.0
   method tune freq = ···; frequency <- freq
   method set_volume vol = ···; volume <- vol
end;;
```

The combined class `clock_radio` simply inherits from both. The resulting class has the methods and fields of both superclasses.

```
# class clock_radio =
  object
     inherit clock
     inherit radio
  end;;
class clock_radio :
  object
    val mutable frequency : float
    val mutable now : float
    val mutable volume : float
    method gettimeofday : float
    method set_volume : float -> unit
    method private tick : unit
    method tune : float -> unit
  end
```

16.1.2 Inheriting from multiple virtual classes

Sometimes inheritance is used as a mechanism to add functionality by inheriting from a partially virtual superclass. Let's take a look at an example, based on the class

comparable we introduced on page 190. We'll define two virtual classes.

- A comparable value can be compared to values of the same type.
- number is a class of numbers that can be compared, having a zero, and a negation function.

The virtual class comparable declares a virtual method compare, and derives a method less_than from it.

```
(* less-than: negative; equal: zero; greater-than: positive *)
type comparison = int

class virtual comparable =
object (self : 'self)
   method virtual compare : 'self -> comparison
   method less_than (x : 'self) = compare self x < 0
end;;
```

For the class number, we require that the subclass provide methods zero, neg (negate), and compare. From that, a method abs (absolute value) can be derived.

```
class virtual number =
object (self : 'self)
   method virtual zero : 'self
   method virtual neg : 'self
   method virtual compare : 'self -> comparison
   method abs =
      if self#compare self#zero < 0 then
         self#neg
      else
         self
end;;
```

Finally, an actual concrete class of numbers inherits from both classes comparable and number, implementing the virtual methods, and inheriting the derived methods. The following class also implements the methods less_than and abs because it inherits them from the virtual superclasses.

```
class float_number x =
object (self : 'self)
   inherit comparable
   inherit number

   val number = x
   method representation = number
   method zero = {< number = 0.0 >}
   method neg = {< number = -. number >}
   method compare y = Pervasives.compare x y#representation
end;;
```

16.1.3 Mixins

A *mixin* is a class that is used to add augment the functionality of or change the behavior of a subclass, but there is no explicit is-a relationship. According to folklore, the term

mix-in was inspired by Steve's Ice Cream Parlor in Somerville, Massachusetts, where extra items like nuts, chocolate sprinkles, *etc*., were mixed into a base flavor of ice cream like chocolate or vanilla.

Let's implement some ice cream with two arbitrary mixed-in flavors. We'll use the drawable objects of Chapter 14. A `blob` is an object that can be drawn, including the classes `circle` and `poly`, and a `collection` is a collection of blobs.

```
class vanilla_ice_cream = object ··· end

class virtual mixed_ice_cream =
object (self)
   inherit vanilla_ice_cream
   inherit collection

   method virtual mixin1 : unit
   method virtual mixin2 : unit

   method stir = self#map (fun item ->
      let t = recenter ** transform#new_rotate (Random.float 6.28)
         ** transform#new_translate (Random.float 10.) (sqrt (Random.float 1e4))
      in item#transform t)

   initializer self#mixin1; self#mixin2; self#stir
end
```

Vanilla ice cream is featureless, but the mixed ice cream contains a collection of extra items. The virtual methods `mixin1` and `mixin2` add the extra items to the collection. To implement a particular kind of mixed ice cream, we implement two classes, one for each mixin.

```
class drop = circle ~color:0x880022 (0., 0.) 10.
class sprinkle = poly ~color:0x220044 [|(0., 0.); (10., 0.); (10., 3.); (0., 3.)|]

class virtual drop_mixin1 =
object (self)
   method virtual add : blob -> unit
   method mixin1 = for i = 0 to 4 do self#add (new drop) done
end

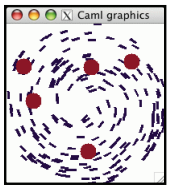
class virtual sprinkle_mixin2 =
object (self)
   method virtual add : blob -> unit
   method mixin2 = for i = 0 to 200 do self#add (new sprinkle) done
end
```

For the final product, we just mix the ice cream with its ingredients. The mixin classes implement the virtual methods of the ice cream class, and the result is a flavored ice cream.

```
class my_favorite_ice_cream =
object
   inherit mixed_ice_cream
   inherit drop_mixin1
   inherit sprinkle_mixin2
end
```

This use of inheritance is not a specialization. We certainly don't mean that `my_favorite_ice_cream` is-a `sprinkle`, or a collection of sprinkles, or anything of the sort. The purpose of the inheritance in this case is simply for the mixin to specify an item that is to be added to the ice cream.

```
(new my_favorite_ice_cream)#draw;;
```



In all the examples of this section, the inheritance is used to combine classes that are really independent. In each case, there may be several declarations of a method as virtual, but there only one implementation. Let's turn to the more general case where methods and fields may be defined multiple times, starting with the topic of *shadowing*.

16.2 Overriding and shadowing

What happens when an object defines a name twice? For comparison, let's first consider a similar question: what happens in a module when a name is defined twice? We know the answer—the definition exported by the module is the last one.

```
# module M =
  struct
     let x = 1;;
     Printf.printf "x = %d!\n" x;;
     let x = 2;;
  end;;
x = 1!
# M.x;;
- : int = 2
```

Objects are a little different from modules, but the naming is similar. Consider the following objects that contain duplicate definitions. OCaml warns about the duplication, but it is instructive to see the results. First, let's override a method.

```
# let a =
  object
     method get = 1
     method get = 2
  end;;
Warning M: the method get is overriden in the same class.
val a : < get : int > = <obj>
# a#get;;
- : int = 2
```

As would be expected, the last method definition is the one that is used by the object. We can try the same thing with fields.

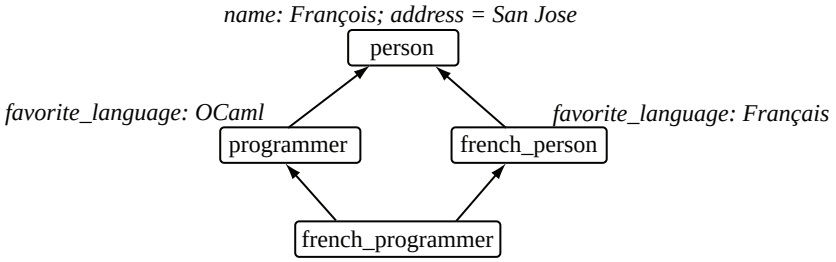
```
# let b =
  object
```

```
    val x = 1
    method get = x
    val x = 2
  end;;
Warning V: the instance variable x is overriden.
# b#get;;
- : int = 2
```

Field override seems to be frowned upon by the compiler. Still, the result is the same, it is the last definition that is used by the object. Any preceding definitions for the same name are ignored.

16.3 Repeated inheritance

Repeated inheritance occurs when a class inherits from another along multiple paths in the inheritance hierarchy, perhaps more commonly known as the "diamond problem." For example, we might say that a French programmer is both French and a programmer, and both of these are more generally persons. That means that a French programmer inherits from person twice. The following diagram illustrates the class relationship, with a hypothetical programmer in italics.



In OCaml, the `inherit` directive is nearly equivalent to textual inclusion—the result is the same as if the `inherit` clause were replaced with the text of the class being inherited from. This means that a `programmer` includes the program text for a `person`, so does a `french_person`, and the `french_programmer` contains the program text twice.

When does this make a difference? For any given method or field, remember the rule: it is the *last* definition that matters, so perform the textual expansion mentally and look for the final definition. Consider a case where a class `a` is inherited twice.

```
# class a =
  object
    method x = 1
  end;;
# class b =
  object
    inherit a
    method x = 2
    inherit a
  end;;
Warning: ... x ... is overriden by the class a
# (new b)#x;;
```

```
13  - : int = 1
```

The result is `1` because the final definition for the method `x` comes from the inherit clause on line 9 (which defines `x` as `1`).

The diamond problem is similar. Consider the French programmer example, here drawn in the shape of a diamond (don't type it in this way).

```
class person =
object
    method name = "Francois"
    method address = "San Jose"
end

class programmer =
object
   inherit person
   method lang = "OCaml"
end

class french_person =
object
   inherit person
   method lang = "Francais"
end

class french_programmer =
object
    inherit programmer
    inherit french_person
end
```

Let's think how the text will be expanded for class `french_programmer`. If we perform the expansion, here is the order in which the methods are defined: `person#name,address`, `programmer#lang`, `person#name,address`, `french_person#lang`. Keeping only the final definition for each method, we obtain the following class, equivalent to `french_programmer`.

```
class french_programmer_flattened =
object
   method name = "Francois"
   method lang = "Francais"
   method address = "San Jose"
end
```

Field override works the same as method override for fields that are visible. If we had defined the values as fields instead of methods, the result would be the same: the favorite language is Français.

However, fields that are hidden (because of typing) are not overridden; they are duplicated. Suppose we define a class a that defines a mutable field `x`, which is then hidden using a type constraint.

```
class type a_type =
object
   method set : int -> unit
   method get : int
end;;

class a : a_type =
object
   val mutable x = 0
```

```
    method set y = x <- y
    method get = x
end;;
```

Repeated inheritance *duplicates* the hidden field x, which means that operations on one copy of a do not effect the others.

```
# class b =
  object
     inherit a as super1
     inherit a as super2

     method test =
        super1#set 10;
        super2#get
  end;;
class b :
  object method get : int method set : int -> unit method test : int end
# (new b)#test;;
- : int = 0
```

16.4 Avoiding repeated inheritance

The previous section points out the issue with multiple inheritance, which is that there are at least two different policies for repeated inheritance: override and copying. There is no single policy that is best. For example, the French programmer may wish to go by the same name in all contexts, but he might wish to use a different address for his occupation than he uses as a French citizen. That is, the repeated field `name` should refer to the same value, but the `address` field should be copied.

As pointed out, OCaml uses the following policy: visible fields and methods use the override policy, fields and methods that are hidden use the copy policy. This policy can be difficult for a programmer to adhere to, "All the fields that are hidden are copied." The semantics of multiple inheritance might be simple enough to state, "it is just textual expansion," but the problem is that it might be necessary to know the text of all repeated superclasses. For this reason and others, it is natural to want to avoid repeated inheritance, at least in some cases.

16.4.1 Is-a *vs*. has-a

For an example, suppose we have classified vehicles in two different ways: according to where they are used, and by how they are powered. In the following diagram, the class `vehicle` is a superclass of all the others; for example it might contain methods to move forward or stop, *etc*. A `car` is a vehicle that travels roads, a `boat` water; an `electric_vehicle` needs to be charged occasionally, *etc*.

vehicle	
car	electric_vehicle
boat	gasoline_vehicle
submarine	rocket
spacecraft	pedaled_vehicle
	sailed_vehicle
	nuclear_vehicle

A particular kind of vehicle can constructed by combining a property from the left column and another from the right. We have gasoline powered boats, nuclear submarines, and electric cars. Some combinations don't make much sense, like pedaled spacecraft.

On the surface, this classification may seem reasonable. We classify vehicles by two orthogonal properties, and use multiple inheritance to define classes for the combinations that make sense. However, this will involve repeated inheritance, which might be a problem if we don't know how the class `vehicle` is defined.

There is another way to classify vehicles that is just as natural, but avoids the repeated inheritance. That is, we can continue to classify vehicles by where they travel, but instead of classifying powered vehicles, we classify power sources directly.

vehicle
car
boat
submarine
spacecraft

power_source
electric_motor
gasoline_engine
rocket_engine
pedals
sails
nuclear_plant

assembled_vehicle
electric_car = car + electric_motor
nuclear_submarine = submarine + nuclear_plant
sailboat = boat + sails
rocket = spacecraft + rocket_engine
...

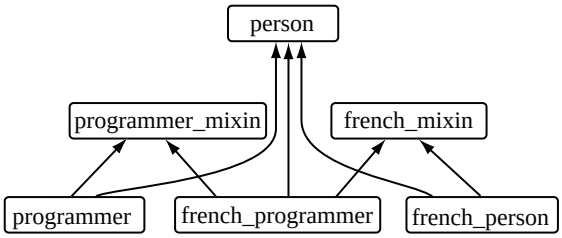
Now we have two independent concepts, vehicles and power sources, and an assembled vehicle needs one of each. The assemblies can be created with multiple inheritance, which has the advantage that bogus assemblies (like rocket-powered submarines) can be omitted from the collection.

An alternative to multiple inheritance is for the vehicle class to take a power source as an argument and include it as a field. There are two potential advantages. One is that it is not necessary to write down every possible assembly. The other is that it might make more sense semantically—a vehicle has-a power source, but it isn't-a power source, at least not normally. A disadvantage is that it isn't easy to rule out combinations that don't make sense.

16.4.2 Mixins revisited

In many ways, the approach to the vehicle example is really just a mixin. Power sources aren't really useful until they are harnessed, which happens when an assembled vehicle is created by mixing in a power source into a vehicle.

This suggests that one approach to avoiding repeated inheritance is to delay the use of multiple inheritance as much as possible. Returning to the French programmer, there were four classes with a repeated base class `person`. Another way to design the hierarchy is to partition the properties, where the programming and French properties are split from the `person` and become mixins.



The mixin classes `programmer_mixin` and `french_mixin` are now standalone classes. They can still refer to the properties of being a person through virtual methods and fields, but they don't make much sense alone until combined with the `person` class.

As with any programming style, it isn't always appropriate to program this way. There are more classes; the relationship between classes is not as well defined—there is nothing that says that a `programmer_mixin` is to be combined with a `person`; and it is easy to forget about maintaining the relationship between a class and its mixins. Of course, in many cases this *is* a useful technique, and it can lead to simpler, shorter programs.

At this point we have covered objects, classes with single inheritance, and classes with multiple inheritance. The object system is quite powerful, and we have many tools at our disposal. There is one remaining important topic: polymorphic classes, which we discuss in the next chapter.

16.5 Exercises

Exercise 16.1 Assume there is a class name that represents the name of a person. We would normally say that a person is-a human and has-a name,

```
class person (n : name) = object inherit human val name = n ··· end
```

Suppose that instead, the class person inherits from both.

```
class person (s : string) =
object
   inherit human
   inherit name s
   ...
end
```

What is the difference? Under what conditions would the different representations be preferred?

Exercise 16.2 Consider the following class, which implements a persistent reference-counted value stored in a file. When there are no more references, the file is removed.

```
class persistent_refcounted_value filename =
object (self)
    (* persistent_value *)
    val mutable x : int list =
       let fin = open_in_bin filename in
       let x = input_value fin in
       close_in fin;
       x
    method get = x
    method set y = x <- y; self#save
    method private save =
       let fout = open_out_bin filename in
       output_value fout x;
       close_out fout

    (* refcounted_value *)
    val mutable ref_count = 1
    method add_ref = ref_count <- ref_count + 1
    method rm_ref =
       ref_count <- ref_count - 1;
       if ref_count = 0 then
          Sys.remove filename
end
```

1. Partition the class into three classes: persistent_value implements persistent values stored in files, refcounted_value implements generic reference counted objects, and persistent_refcounted_value inherits from both.
2. What is the advantage in partitioning the class?

Exercise 16.3 In the French programmer example, the programmer has a field favorite_language and so does the french_person. Can the inheritance hierarchy be modified so that these are available as favorite_programming_language

and `favorite_natural_language`, without modifying the classes `programmer` and `french_person`?

Exercise 16.4 You are given the following functor that defines a class `cell` containing a value of type `T.t`.

```
module MakeCell (T : sig type t end) =
struct
    class cell x =
    object
        val mutable x : T.t = x
        method private get = x
        method private set y = x <- y
    end
end
```

Define a singly-linked list of integers by inheriting from the class `cell` twice. Your class should have the type `int_cons`.

```
class type int_cons =
object
   method hd : int
   method tl : int_cons option
   method set_hd : int -> unit
   method set_tl : int_cons option -> unit
end

type int_list = int_cons option
```

Exercise 16.5 Suppose we have several mutually-recursive functions f_1 `: int -> int`, ..., f_n `: int -> int` that we want to define in separate files. In Exercise 13.5 we did this with recursive modules. Do it with multiple inheritance instead. Is there any advantage to using classes over recursive modules?

Chapter 17

Polymorphic Classes

So far, we have seen many kinds of class and class type definitions. Classes can be fixed, or they can be parameterized by ordinary values. In addition, classes and class types can be *polymorphic*, meaning that they can be parameterized by types, just like other expressions and types in the language (except for module types).

This kind of generic object-oriented programming appears in other programming languages in various forms. The Eiffel programming language supports *genericity*; C++ has a construct called *templates*; Java has type-parameterized classes called *generics*.

In OCaml, polymorphism is not a new concept when applied to classes. It is the *same* concept that appears throughout the language, and it works the same way for classes and class types as it does for other constructs.

17.1 Polymorphic dictionaries

Let's start with an example based on the "map" data structure that we developed in Section 13.3 using functors. A map is a dictionary containing key-value pairs, parameterized by a function `compare` that defines a total order on keys. For brevity, we'll implement the map using association lists.

```
# type ordering = Smaller | Equal | Larger
type ordering = Smaller | Equal | Larger
# class ['key, 'value] map (compare : 'key -> 'key -> ordering) =
    let equal key1 (key2, _) = compare key1 key2 = Equal in
    object (self : 'self)
      val elements : ('key * 'value) list = []
      method add key value = {< elements = (key, value) :: elements >}
      method find key = snd (List.find (equal key) elements)
    end;;
class ['a, 'b] map : ('a -> 'a -> ordering) ->
  object ('self)
    val elements : ('a * 'b) list
    method add : 'a -> 'b -> 'self
    method find : 'a -> 'b
  end
```

The result type has been edited slightly for readability.

The class definition is parameterized by two types, `'key` and `'value`, written within square brackets before the class name as `['key, 'value]`. The square brackets are required even if there is only one type parameter to a class definition.

The entries of the map are stored in the list `elements`, of type `('key * 'value) list`, and the map's methods examine the list. This implementation of a map is pure, the method `add` produces a new object with the new entry added to the list, leaving the `self` object unchanged.

17.1.1 Free type variables in polymorphic classes

The type constraints, besides being good documentation, are required. Class definitions are not allowed to have free type variables, meaning that every type variable must be a parameter, or it must be bound somewhere else in the class definition. The following definition fails.

```
# class ['a] is_x x =   (* Does not work! *)
     object (self : 'self)
        method test y = (x = y)
     end;;
Some type variables are unbound in this type:
  class ['a] is_x : 'b -> object method test : 'b -> bool end
The method test has type 'a -> bool where 'a is unbound
```

The reason for the failure is that the type of the argument `x` is not specifically written as being of type `'a`, hence the method `test` has some type `'b -> bool`, where the type variable `'b` is not a parameter of the class. The solution is to constrain the types so that the method `test` has type `'a -> bool`.

```
# class ['a] is_x (x : 'a) =
     object (self : 'self)
        method test y = (x = y)
     end;;
class ['a] is_x : 'a -> object method test : 'a -> bool end
```

17.1.2 Instantiating a polymorphic class

Instantiating a class (to get an object), works the same it does with non-polymorphic classes. The `new` operator is used to instantiate the class. For example, here is how we might construct an actual map object where the keys are integers.

```
# let compare_int (i : int) (j : int) =
     if i < j then Smaller
     else if i > j then Larger
     else Equal;;
val compare_int : int -> int -> ordering = <fun>
# let empty_int_map = new map compare_int;;
val empty_int_map : (int, '_a) map = <obj>
# let one = empty_int_map#add 1 "One";;
val one : (int, string) map = <obj>
# empty_int_map;;
```

```
- : (int, string) map = <obj>
```

Note that the type for the empty map is `empty_int_map : (int, '_a) map`. That is, it does not have polymorphic type, hence it can be used only at one type. This is due to the *value restriction* (Section 5.1.1)—the expression `new map compare_int` is an application, so it is not a value, and so it is not polymorphic. For the most part, the practical consequences of the value restriction are minimal, it simply means that a new empty map must be created for each type of map value that is to be used in a program.

One additional step we might take is to define a class specifically for the case when the keys are integers. The new class is polymorphic over just one type, the type of values.

```
# class ['value] int_map = [int, 'value] map compare_int;;
class ['a] int_map : [int, 'a] map
```

Note that the type arguments are required on the right as part of the definition (in addition to the normal value argument `compare_int`). The syntax for type arguments is a sequence of comma-separated type expressions between square brackets, placed before the class name. We could, if we wish, further constrain the type.

```
# class int_map2 = [int, string * float] map compare_int;;
class int_map2 : [int, string * float] map
```

17.1.3 Inheriting from a polymorphic class

Inheriting from a polymorphic class works as usual, except that type arguments to polymorphic superclasses must be supplied explicitly. Let's define a new kind of map that supports a method `iter` that applies a function once to each entry in the map.

```
# class ['key, 'value] iter_map compare =
    object
      inherit ['key, 'value] map compare
      method iter f = List.iter (fun (key, value) -> f key value) elements
    end;;
class ['a, 'b] iter_map : ('a -> 'a -> ordering) ->
  object ('c)
    ...
    method iter : ('a -> 'b -> unit) -> unit
  end
```

The directive `inherit` takes the type arguments in addition to any normal arguments, but otherwise the directive works as expected.

Next, let's consider a new method `map` that applies a function to each of the values in the dictionary, returning a new map. Given a function `f : 'value -> 'value2`, what should be the type of the method `map`? Let's write the code.

```
# class ['key, 'value] map_map compare =
  object (self : 'self)
    inherit ['key, 'value] map compare
    method map f =
      {< elements = List.map (fun (key, value) -> key, f value) elements >}
```

```
  end;;
class ['a, 'b] map_map : ('a -> 'a -> ordering) ->
  object ('self)
    ...
    method map : ('b -> 'b) -> 'self
  end
```

Note the type of the method `map`, which requires that the function argument have type `'b -> 'b`. We might have expected a more general typing where, given an object of type `obj : ['key, 'value1] map_map` and a function `f : 'value1 -> 'value2`, that the expression `obj#map f` would have type `['key, 'value2] map_map`.

There is a good reason for the more restrictive typing. Suppose we decide to build a variation on a map where, instead of the method `find` raising an exception when an entry is not found, it returns some default value. The new class is easy to define.

```
class ['key, 'value] default_map compare (default : 'value) =
  object (self : 'self)
     inherit ['key, 'value] map_map compare as super
     method find key =
        try super#find key with
           Not_found -> default
  end;;
```

Objects of the class `default_map` have two places where values of type `'value` appear: in the list of `elements`, and the `default` value. It is not safe to change the type of `elements` without also changing the `default` value in the same way. In this case, the more general typing for the method `map` would be unsafe because it doesn't also change the `default` value.

Of course, OCaml does not try to predict how subclasses will be created. The only safe approach is for the object type to be invariant.

17.2 Polymorphic class types

Polymorphic classes have polymorphic class types, using the usual syntax where the type parameters are enclosed in square brackets.

```
# class type ['key, 'value] map_type =
    object ('self)
      method add  : 'key -> 'value -> 'self
      method find : 'key -> 'value
    end;;
class type ['a, 'b] map_type =
  object ('c) method add : 'a -> 'b -> 'c method find : 'a -> 'b end
# class ['key, 'value] map2 (compare : 'key -> 'key -> ordering)
  : ['key, 'value] map_type =
    let equal key1 (key2, _) = compare key1 key2 = Equal in
    object ··· end
class ['a, 'b] map2 : ('a -> 'a -> ordering) -> ['a, 'b] map_type
```

This implementation of the class `map2` is entirely self-contained. Let's look at an example of a recursive definition, based on the implementation of binary search trees in

Section 6.1. In that section, we defined the binary tree with the following polymorphic union type.

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

If we wish to take an object-oriented approach, we can implement each case of the union as a class that has a class type `['a] tree`, where the type `['a] tree` specifies the operations on a tree, but not its implementation. For our purposes, a tree supports a functional `add` operation to add an element to the tree, and a `mem` function to test for membership in the tree.

```
class type ['a] tree =
  object ('self)
    method add : 'a -> 'a tree
    method mem : 'a -> bool
  end;;
```

There are then two classes that implement a tree: a class `['a] leaf` that represents the empty tree, and a class `['a] node` that represents an internal node. Let's start with the internal node.

```
class ['a] node (compare : 'a -> 'a -> ordering)
    (x : 'a) (l : 'a tree) (r : 'a tree) =
  object (self : 'self)
    val label = x
    val left = l
    val right = r
    method mem y =
      match compare y label with
         Smaller -> left#mem y
       | Larger -> right#mem y
       | Equal -> true
    method add y =
      match compare y label with
         Smaller -> {< left = left#add y >}
       | Larger -> {< right = right#add y >}
       | Equal -> self
  end;;
```

An internal node has three fields: a label and two children, where the children have type `'a tree`. The method `mem` performs a binary search, and the method `add` performs a functional update, returning a new tree.

The class `['a] leaf` is simpler, the method `mem` always returns `false`, and the method `add` produces a new internal node.

```
class ['a] leaf (compare : 'a -> 'a -> ordering) =
  object (self : 'self)
    method mem (_ : 'a) = false
    method add x =
      new node compare x (new leaf compare) (new leaf compare)
  end;;
```

This implementation is adequate, but it is slightly inefficient because the method `add` creates entirely new leaves. Since all leaves are the same, we might consider using `self` instead, but we run into a type error because the type `'self` is not equivalent to `'a tree`.

```
# class ['a] leaf (compare : 'a -> 'a -> ordering) =
    object (self : 'self)
      method mem (_ : 'a) = false
      method add x = new node compare x self self
    end;;
Characters 151-155:
        new node compare x self self
                           ^^^^
This expression has type < add : 'a -> 'b; mem : 'a -> bool; .. >
but is here used with type 'a tree
Self type cannot be unified with a closed object type
```

The problem is that, in general, the type `'self` is a subtype of `'a leaf`, but the class `node` takes arguments of the exact type `'a tree`. One solution is to coerce `self` to have the appropriate type.

```
class ['a] leaf (compare : 'a -> 'a -> ordering) =
  object (self : 'self)
    method mem (_ : 'a) = false
    method add x =
      new node compare x (self :> 'a tree) (self :> 'a tree)
  end;;
```

The coercion works as we expect, and the definition is accepted. We investigate polymorphic coercions more in the following section.

17.2.1 Coercing polymorphic classes

Objects having polymorphic class types can be coerced just like those with non-polymorphic types, but the process requires more preparation when the type arguments also change during the coercion.

Before we begin the discussion, let's define an example that is smaller and easier to work with. We define a polymorphic class `mut_pair` that is like a mutable arity-2 tuple.

```
# class ['a, 'b] mut_pair (x0 : 'a) (y0 : 'b) =
    object (self : 'self)
      val mutable x = x0
      val mutable y = y0
      method set_fst x' = x <- x'
      method set_snd y' = y <- y'
      method value = x, y
    end;;
```

```
class ['a, 'b] mut_pair : 'a -> 'b ->
  object
    val mutable x : 'a
    val mutable y : 'b
    method set_fst : 'a -> unit
    method set_snd : 'b -> unit
    method value : 'a * 'b
  end
```

To test it, let's use the animal example. Each kind of animal should have its own class derived from a common super-class `animal` that characterizes all animals. Here are some example definitions.

```
# class virtual animal (name : string) =
    object (self : 'self)
      method eat = Printf.printf "%s eats.\n" name
    end;;
class virtual animal : string -> object method eat : unit end
# class dog (name : string) =
    object (self : 'self)
      inherit animal name
      method bark = Printf.printf "%s barks!\n" name
    end;;
class dog : string -> object method bark : unit method eat : unit end
# let dogs = new mut_pair (new dog "Spot") (new dog "Rover");;
val dogs : (dog, dog) mut_pair = <obj>
```

According to our definition, every animal has a name, all animals eat, and dogs also bark. The final value `dogs` is a pair of dogs, named Spot and Rover.

The class `animal` is marked as `virtual` only because we intend that every animal should belong to a particular class; there are no generic animals. Some operations, however, should work for animals generically. For example, let's build a function `eat2` that, given a pair of animals, calls the method `eat` for the two animals.

```
# let eat2 animals =
     let x, y = animals#value in x#eat; y#eat
val eat2 :
  < value : < eat : 'a; .. > * < eat : 'b; .. >; .. > -> 'b = <fun>
# eat2 dogs;;
Spot eats.
Rover eats.
```

Note the strange type for the function `eat2`; it takes an object with a method `value` that produces a pair of objects with methods `eat`. We might want to give it a simpler type by specifically stating that it takes a pair of animals.

```
# let eat2_both (animals : (animal, animal) mut_pair) =
     let x, y = animals#value in x#eat2; y#eat2;;
val eat2 : (animal, animal) mut_pair -> unit = <fun>
# eat2 dogs;;
```

```
Characters 15-19:
  eat2 dogs;;
               ^^^^
This expression has type
  (dog, dog) pair = ...
but is here used with type
  (animal, animal) mut_pair = ...
Type dog = < bark : unit; eat : unit > is not compatible with type
  animal = < eat : unit >
Only the first object type has a method bark
```

Here, we run into a problem—the function `eat2` expects a pair of animals, but we passed it a pair of dogs. Of course, every dog is an animal, so perhaps we just need to perform a type coercion to convert the pair to the right type.

```
# let animals = (dogs : (dog, dog) mut_pair :> (animal, animal) mut_pair);;
Characters 14-71:
  let animals = (dogs : (dog, dog) mut_pair :> (animal, animal) mut_pair);;
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
...
Type animal = < eat : unit > is not a subtype of type
  dog = < bark : unit; eat : unit >
```

Here we run into more trouble. The error message states that `animal` is not a subtype of `dog`. Given a single dog object like `spot`, we can coerce it explicitly with the expression `(spot : dog :> animal)`. However, for the pair `dogs` it seems that we need would to coerce the dog objects individually, which would not only be annoying, but possibly incorrect because a new copy of the pair must be created.

OCaml does provide a solution, but to understand it we need to look at *variance annotations*, which describe what coercions are legal for polymorphic classes.

17.2.2 Variance annotations

OCaml uses *variance annotations* on the parameters of a type definition to specify its subtyping properties. A parameter annotation `+'a` means that the definition is covariant in `'a`; an annotation `-'a` means that the definition is contravariant in `'a`; and the plain parameter `'a` means the definition is invariant in `'a`. When a type is defined, the compiler checks that the annotations are legal.

```
# type (+'a, +'b) pair' = 'a * 'b;;
type ('a, 'b) pair' = 'a * 'b
# type (+'a, +'b) func = 'a -> 'b;;
Characters 5-31:
  type (+'a, +'b) func = 'a -> 'b;;
       ^^^^^^^^^^^^^^^^^^^^^^^^^^
In this definition, expected parameter variances are not satisfied.
The 1st type parameter was expected to be covariant,
but it is contravariant
# type (-'a, +'b) func = 'a -> 'b;;
type ('a, 'b) func = 'a -> 'b
```

The toploop is erasing the annotations in the displayed output, but it is still checking that the annotations are legal.

Let's look at how this works in the context of classes and class types. Consider a class definition of an immutable pair.

```
# class [+'a, +'b] pair (x0 : 'a) (y0 : 'b) =
    object (self : 'self)
      val mutable x = x0
      val mutable y = y0
      method value : 'a * 'b = x, y
    end;;
class ['a, 'b] pair : 'a -> 'b -> object ... method value : 'a * 'b end
# let p = new pair (new dog "Spot") (new dog "Rover");;
val p : (dog, dog) pair = <obj>
```

As before, we might wish to coerce the pair of dogs to a pair of animals. This time the coercion works as expected.

```
# (p :> (animal, animal) pair);;
- : (animal, animal) pair = <obj>
```

The reason this works is because the class type for `pair` is covariant in the component types. Since `dog` is a subtype of `animal`, the type `(dog, dog) pair` is a subtype of `(animal, animal) pair`.

Let's try to specify similar annotations for the class of mutable pairs, `mut_pair`.

```
# class [+'a, +'b] mut_pair (x0 : 'a) (y0 : 'b) =
    object (self : 'self)
      val mutable x = x0
      val mutable y = y0
      method set_fst x' = x <- x'
      method set_snd y' = y <- y'
      method value = x, y
    end;;
Characters 5-185: .....
In this definition, expected parameter variances are not satisfied.
The 1st type parameter was expected to be covariant,
but it is invariant
```

Why isn't the new definition allowed? If we look back to the method types in the unannotated class, we find the following types for the `set_xxx` methods.

```
# class ['a, 'b] mut_pair (x0 : 'a) (y0 : 'b) = ···
class ['a, 'b] pair : 'a -> 'b ->
  object
    ...
    method set_fst : 'a -> unit
    method set_snd : 'b -> unit
    method value : 'a * 'b
  end
```

The problem is that the type parameters `'a` and `'b` occur to the left of an arrow, so the occurrences are contravariant. The other significant occurrences are in the type `'a * 'b`, where they are covariant. Since the variables have both contravariant and covariant occurrences, they must be invariant.

For some intuition, imagine that the covariant definition were allowed. Consider the following sequence of actions, where for illustration we refer to a subclass `cat` that

inherits from `animal`, but is not a dog.

```
# let dogs = new mut_pair (new dog "Spot") (new dog "Rover");;
dogs : (dog, dog) mut_pair
  (* Imagine if this were legal *)
# let animals = (dogs :> (animal, animal) mut_pair);;
animals : (animal, animal) mut_pair
# let fifi = new cat "Fifi";;
fifi : cat
# animals#set_fst (fifi :> animal);;
# let fifi' = fst dogs#value;;
fifi' : dog
# fifi'#bark;;
????
```

The steps 1) create a pair of dogs, 2) coerce it to a pair of animals, and 3) replace one of the dogs with a cat (which is also an animal). Since the modification is done in-place, the original dog pair now contains a cat. This is wrong because, among other things, cats do not bark.

17.2.3 Positive and negative occurrences

Mechanically speaking, the restrictions on variance annotations are not determined by whether a class has mutable fields or contains side-effects, it is purely based on the non-private method types.

Consider the following slightly different definition of a class `get_pair`.

```
# class [+'a, +'b] get_pair (x0 : 'a) (y0 : 'b) =
    object (self : 'self)
      val mutable x = x0
      val mutable y = y0
      method get_fst : ('a -> unit) -> unit = fun f -> f x
      method value = x, y
    end;;
class ['a, 'b] get_pair : 'a -> 'b ->
  object
    val mutable x : 'a
    val mutable y : 'b
    method get_fst : ('a -> unit) -> unit
    method value : 'a * 'b
  end
```

This class is accepted, with the covariant annotation `+'a`, even though `'a` occurs to the left of an arrow in the method `get_fst : ('a -> unit) -> unit`. How does this work?

There is a straightforward calculation for determining the variance of a variable in a type. First, for some occurrence of the type variable in question, we define a *left-nesting depth* with respect to the arrows in the type definition, where the left-nesting depth increases by one each time the type variable occurs to the left of an arrow in the fully-parenthesized type. Covariant constructors, like `*`, do not affect the depth. Type constructors, like `ref`, that specify mutable values require that the variable be invariant.

Here are some examples for the nesting depth of a type variable `'a`, where "*"

indicates that the type is invariant.

Type	Fully-parenthesized type	Depth
t_1 -> t_2 -> 'a	t_1 -> (t_2 -> 'a)	0
t_1 -> 'a -> t_2 -> t_3	t_1 -> ('a -> (t_2 -> t_3))	1
(t_1 -> 'a -> t_2) -> t_3	(t_1 -> ('a -> t_2)) -> t_3	2
(t_1 -> t_2 -> 'a) -> t_3	(t_1 -> (t_2 -> 'a)) -> t_3	1
((('a * t_1) -> t_2) -> t_3) -> t_4	same	3
'a ref	same	*
('a -> t_1) ref -> t_2	same	*

Next, consider the type variables that are not invariant. If the nesting depth is even, the occurrence is called *positive*; if it is odd, the occurrence is *negative*. Positive occurrences are covariant, negative occurrences are contravariant. For the method `get_fst : ('a -> unit) -> unit`, the nesting depth of `'a` is 2, which means that the occurrence is positive and covariant.

17.2.4 Coercing by hiding

Let's return to the class of mutable pairs `mut_pair`. Suppose we have pair of dogs, and we still wish to coerce the object to a pair of animals. The methods `set_fst : 'a -> unit` and `set_snd : 'b -> unit` prevent this, because of the negative occurrence of the type variables `'a` and `'b`. However, it is still possible to coerce the class, *provided* that the these methods are omitted.

```
# let dogs = new mut_pair (new dog "Spot") (new dog "Rover");;
val dogs : (dog, dog) mut_pair = <obj>
# (dogs :> (animal, animal) pair);;
- : (animal, animal) pair = <obj>
# (dogs : (dog, dog) mut_pair :> (animal, animal) mut_pair);;
...
Type animal = < eat : unit > is not a subtype of type
  dog = < bark : unit; eat : unit >
```

We can think of the the coercion to `(animal, animal) pair` as two steps: the first step coerces the object to type `(dog, dog) pair`, which simply means omitting the methods `set_fst` and `set_snd`; the next step coerces to `(animal, animal) pair`, which is legal because the class `pair` is covariant in its type parameters.

There are other examples where it may be useful to view the same object with different types. For example, suppose we have an object that behaves like a reference cell.

```
# class ['a] refcell (x0 : 'a) =
    object (self : 'self)
      val mutable x = x0
      method set y = x <- y
      method get = x
    end;;
class ['a] refcell : 'a ->
  object val mutable x : 'a method get : 'a method set : 'a -> unit end
```

We can give an object of this class two types, one covariant type with just the method `get`, and another with contravariant type having just the method `set`.

```
# class type [+'a] getcell = object method get : 'a end;;
class type ['a] getcell = object method get : 'a end
# class type [-'a] setcell = object method set : 'a -> unit end;;
class type ['a] setcell = object method set : 'a -> unit end
```

To test it, let's introduce a new class for guard dogs.

```
# class guard_dog name =
    object (self : 'self)
      inherit dog name
      method growl = Printf.printf "%s growls!\n" name
    end;;
class guard_dog : string ->
  object method bark : unit method growl : unit method eat : unit end
# let cell = new refcell (new dog "Spot");;
val cell : dog refcell = <obj>
# let read = (cell : (dog) refcell :> (animal) getcell);;
val read : animal getcell = <obj>
# let write = (cell : (dog) refcell :> (guard_dog) setcell);;
val write : guard_dog setcell = <obj>
# write#set (new guard_dog "Spike");;
- : unit = ()
# let spike = read#get;;
val spike : animal = <obj>
# spike#eat;;
Spike eats.
```

17.3 Type constraints

Previously, we defined a function `eat2` that called the `eat` methods for a pair of animals. Another way to do this is to define a class `animal_pair` specifically for pairs of animals. We would like to inherit from the class `pair`, but how can we specify that the components of the pair are animals? The solution is to use type constraints, previously introduced in Section 14.1. To keep our example small, we require that the two animals have the same type.

```
# class [+'a] animal_pair (x0 : 'a) (y0 : 'a) =
    object (self : 'self)
      inherit ['a, 'a] pair x0 y0
      constraint 'a = #animal
      method eat = x#eat; y#eat
    end;;
class [+'a] animal_pair : 'a -> 'a ->
  object
    constraint 'a = #animal
    method eat : unit
    ...
  end
# let dogs = new animal_pair (new dog "Spot") (new dog "Rover");;
val dogs : dog animal_pair = <obj>
# dogs#eat;;
```

```
Spot eats.
Rover eats.
```

The constraint `constraint 'a = #animal` means that the type `'a` must be a subtype of animals. We could have written an exact constraint instead, written `constraint 'a = animal`. The exact constraint would mean that the elements of the pair must exactly be `animal`, not any of its subtypes. Exact constraints may be appropriate in some places, but they would make this example much less useful. For example, suppose we wish to define a pair for dogs.

```
# class [+'a] dog_pair x0 y0 =
    object (self : 'self)
      inherit ['a] animal_pair x0 y0
      constraint 'a = #dog
      method bark = x#bark; y#bark
    end;;
class [+'a] dog_pair : 'a -> 'a ->
  object
    constraint 'a = #dog
    method bark : unit
    method eat : unit
    ...
  end
```

The constraint `#dog` is compatible with the constraint `#animal`, and the two together simplify to the single constraint `#dog`. Exact constraints wouldn't be compatible.

Next, to illustrate classes that contain polymorphic fields, let's define a class that represents a list of animal pairs.

```
# class ['a] animal_pairs =
  object (self : 'self)
    val mutable pairs : 'a animal_pair list = []
    method insert x0 y0 =
      pairs <- new animal_pair x0 y0 :: pairs
    method eat = List.iter (fun p -> p#eat) pairs
  end;;
class ['a] animal_pairs :
  object
    constraint 'a = #animal
    val mutable pairs : 'a animal_pair list
    method insert : 'a -> 'a -> unit
    method eat : unit
  end
```

Note that the toploop infers the constraint `'a = #animal`.

If a class contains type constraints, the constraints must also be included in the class type. It is of course legal to coerce objects to remove the type constraint because the type of the object has already been fixed, and the constraint has already been satisfied.

```
# class type [+'a] read_only_animal_pairs_type =
    object method eat : unit end;;
# let dogs = new animal_pairs;;
val dogs : _#animal animal_pairs = <obj>
# dogs#insert (new dog "Spot") (new dog "Fifi");;
# dogs#insert (new dog "Rover") (new dog "Muffin");;
```

Modules

```
module type DogSig = sig
   type t
   val create     : string -> t
   val name       : t -> string
   val eat        : t -> unit
   val bark       : t -> unit
   val bark_eat   : t -> unit
end;;

module Dog : DogSig = struct
   type t = string
   let create name = name
   let name dog = dog
   let eat dog =
      printf "%s eats.\n" (name dog)
   let bark dog =
      printf "%s barks!\n" (name dog)
   let bark_eat dog =
      bark dog; eat dog
end;;
```

Objects

```
class type dog_type =
  object ('self)
    method name       : string
    method eat        : unit
    method bark       : unit
    method bark_eat   : unit
  end;;


class dog name : dog_type =
  object (self : 'self)
    method name = name
    method eat =
      printf "%s eats.\n" self#name
    method bark =
      printf "%s barks!\n" self#name
    method bark_eat =
      self#bark; self#eat
  end;;
```

Figure 17.1: Implementations of dogs, using modules and objects.

```
# let animals = (dogs : dog animal_pairs :> animal read_only_animal_pairs_type);;
val animals : animal read_only_animal_pairs_type = <obj>
```

17.4 Comparing objects and modules

OCaml provides two significant tools for abstraction and re-use: the module system and the object system. Many tasks are supported equally well by both systems, but there are differences that will determine whether you use one system or the other. To finish this chapter, we'll explore these differences.

17.4.1 Late binding

Let's start with our example of animals, coding it in both systems. The example is very simple, but should illustrate some of the differences. In the example, we define a dog to be a thing that can bark and eat, shown in Figure 17.1.

The module defines an abstract data type: there is a type of dogs `Dog.t` (which is just a string for the name of the dog), and functions for creating a new instance of a dog, plus functions for having it bark and eat. The object definition is similar, except that dog creation is performed with the operator `new`; there is no need for a separate method.

So far, there is very little difference; which implementation to use is mainly a matter of preference.

Modules	Objects
`module Hound : DogSig = struct` `  include Dog` `  let bark dog =` `  printf "%s howls!\n" (name dog)` `end;;`	`class hound_n : dog_type =` `  object (self : 'self)` `  inherit dog_n` `  method bark =` `  printf "%s howls!\n" self#name` `  end;;`
`# let sam = Hound.create "Sam";;` *`val sam : string = "Sam"`* `# Hound.bark sam;;` *`Sam howls!`* `# Hound.bark_eat sam;;` *`Sam barks!`* *`Sam eats.`*	`# let sam = new hound "Sam";;` *`val sam : hound = <obj>`* `# sam#bark;;` *`Sam howls!`* `# sam#bark_eat;;` *`Sam howls!`* *`Sam eats.`*

Figure 17.2: Defining a subtype of dogs.

Next, let's consider what will happen should we wish to create a new kind of dog. We'll define a new implementation for hounds, which usually howl instead of barking. What we would like to do is replace the function/method `bark` with a new implementation that prints the appropriate message. The new implementations are shown in Figure 17.2, where the module definition uses `include` to include the `Dog` implementation. The object definition uses `inherit`.

The behavior of the two implementations differ. When the function `Hound.bark` is called, the dog howls as expected. However, when the function `Hound.bark_eat` is called, the hound barks (not howls), and then eats. The reason is that the function `bark_eat` was defined in the `Dog` module, and so it refers to the definition of `Dog.bark`. This is simply static scoping: an identifier refers to the nearest previous definition in the program text that is in scope.

In contrast, in the object definition, the method call `self#bark` refers to the *latest* definition of the `bark` method in the class. The latest definition is in the `hound` object, and so the dog always howls.

There are several names for this behavior. For objects it is called *late binding*, *dynamic method dispatch*, or *open recursion*. For modules it is called *early binding*, *static scoping*, or *closed recursion*. When late binding is desired, as it probably is in this example, objects are the preferred solution.

Another point to notice is that in the module implementation, the data is decoupled from the functions that use the data. In other words, the programmer must be sure to use the functions from the `Hound` module when dealing with hounds. This is enforced by the type checker because the type `Hound.t` is different from `Dog.t`. We could, if we wish, define a sharing constraint `Hound : DogSig with type t = Dog.t`. However, this would allow any of the functions from the module `Dog` to be applied to hounds, which may not be what we wish. In contrast, the class `hound` encapsulates the data with its methods; the programmer need not be concerned about whether the appropriate methods are being used.

17.4.2 Extending the definitions

It is frequently believed that object-oriented programs are easier to modify and extend than "normal" functional programs. In fact, this is not the always the case—the two styles are different and not exactly comparable. Let's consider an example where we define a calculator-style language with variables, together with an evaluator. In the functional approach, we'll define the language using a union type, and in the object-oriented approach we'll define a class of expressions. To handle variables, we'll use a version of the `Map` data structure that we developed earlier in this chapter, where we specifically represent variables as strings.

Modules	Objects
`module type EnvSig = sig` `  type 'a t` `  val empty : 'a t` `  val add : 'a t -> string -> 'a -> 'a t` `  val find : 'a t -> string -> 'a` `end;;` `module Env : EnvSig = struct` `  type 'a t = (string * 'a) list` `  let empty = []` `  let add env v x = (v, x) :: env` `  let find env v = List.assoc v env` `end;;`	`class type ['a] env_sig =` `  object ('self)` `  method add : string -> 'a -> 'self` `  method find : string -> 'a` `  end;;` `class ['a] env : ['a] env_sig =` `  object (self : 'self)` `  val env : (string * 'a) list = []` `  method add v x =` `  {< env = (v, x) :: env >}` `  method find v = List.assoc v env` `  end;;`

An "environment" is a map from variables to values. The implementations, as the module `Env` and the class `env` are very similar. Again, the choice is mainly stylistic.

Next, we define the language itself. We'll include constants, variables, some basic arithmetic, and a "let" binding construct, shown in Figure 17.3. On the left, we show a "standard" definition where expressions are specified with a disjoint union type. The evaluator is a single function, defined by pattern matching, that computes the value associated with each kind of expression.

On the right, we show a similar object-oriented implementation, where the class type `exp` describes a generic expression that has a method `eval` that produces a value given an environment. Each kind of expression is defined as a specific implementation of a class that has class type `exp`. Since there are five kinds of expressions in the language, there are five different classes.

The implementation using unions is somewhat smaller than the implementation using objects, but otherwise the implementations are much the same. The main reason for the object-oriented program being larger is that each of the classes must be named, and there is some overhead for each definition. In larger programs, it is likely that this overhead would be insignificant.

Adding a function

One way in which the implementations differ has to do with how they can be extended. Suppose we wish to add a new function `print` that prints out an expression. Again, the implementations are fairly straightforward. For the implementation with unions,

Unions

```
type exp =
   Int of int
 | Var of string
 | Add of exp * exp
 | If  of exp * exp * exp
 | Let of string * exp * exp

let rec eval env = function
   Int i -> i
 | Var v -> Env.find env v
 | Add (e1, e2) ->
    eval env e1 + eval env e2
 | If (e1, e2, e3) ->
    if eval env e1 <> 0
    then eval env e2
    else eval env e3
 | Let (v, e1, e2) ->
    let i = eval env e1 in
    let env' = Env.add env v i in
       eval env' e2
```

```
(* let x = 3 in x + 4 *)
# let e = Let ("x", Int 3,
     Add (Var "x", Int 4));;
val e : exp = Let ("x", Int 3,
   Add (Var "x", Int 4))
# let i = eval Env.empty e;;
val i : int = 7

(* Evaluation: objects *)
# let e =
    new let_exp "x" (new int_exp 3)
      (new add_exp (new var_exp "x")
        (new int_exp 4));;
val e : let_exp = <obj>
# let i = e#eval (new env);;
val i : int = 7
```

Objects

```
class type exp =
  object ('self)
    method eval : int env -> int
  end

class int_exp (i : int) =
  object (self : 'self)
    method eval (_ : int env) = i
  end

class var_exp v =
  object (self : 'self)
    method eval (env : int env) =
      env#find v
  end

class add_exp (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method eval env =
      e1#eval env + e2#eval env
  end

class if_exp
  (e1 : #exp) (e2 : #exp) (e3 : #exp) =
  object (self : 'self)
    method eval env =
      if e1#eval env <> 0
      then e2#eval env
      else e3#eval env
  end

class let_exp
  (v : string) (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method eval env =
      let i = e1#eval env in
      let env' = env#add v i in
        e2#eval env'
  end;;
```

Figure 17.3: Implementing an evaluator

we simply add a new function, defined by pattern matching, that describes how to print each of the kinds of expressions.

```
let rec print chan = function
   Int i -> fprintf chan "%d" i
 | Var v -> fprintf chan "%s" v
 | Add (e1, e2) ->
     fprintf chan "(%a + %a)" print e1 print e2
 ...
```

The object version is somewhat different. In this case, we must add a new method to each of the classes for each of the kinds of expressions. This means that either 1) we have to modify the source code for each class definition, or 2) we have to define new classes by inheritance that provide the new implementations—and then be sure to use the new definitions in all the places where we construct new expressions. Fortunately, the type checker will help us find all the code that needs to be changed. Let's use the latter form.

```
class type printable_exp =
  object ('self)
    inherit exp
    method print : out_channel -> unit
  end

class printable_add_exp
  (e1 : #printable_exp) (e2 : #printable_exp) =
  object (self : 'self)
    inherit add_exp e1 e2
    method print chan =
      fprintf chan "(%t + %t)" e1#print e2#print
  end
...
```

Updating the union implementation is clearly easier than updating the object implementation. To add a new function to the union implementation, we simply add it—none of the original code must be modified.

With objects, we have two options. If we have access to the original class definitions, each of the classes can (and must) be updated. Otherwise, we define new updated classes by inheritance, and each object creation with `new` must be updated to refer to the new classes. The updates may be scattered throughout the program, and it may take some time to find them.

Adding a new kind of expression

For another kind of example, let's consider what must be done if we add a new kind of expression. For example, suppose we wish to add an expression that represents the product of two expressions. This time, the object-oriented approach is easy, we just add a new object for products.

```
class printable_mul_exp (e1 : #printable_exp) (e2 : #printable_exp) =
  object (self : 'self)
    method eval env = e1#eval env * e2#eval env
```

```
    method print chan = fprintf chan "(%t + %t)" e1#print e2#print
  end;;
```

Here, none of the original code need be modified. Objects of type `printable_mul_exp` can be used anyplace where an expression is needed.

In contrast, updating the union definition is much more difficult. We *must* be able to update the original type definition to include the new case, and in addition, each of the functions must be updated to handle the new kind of expression.

```
type exp =
   ...
 | Mul of exp * exp

let rec eval env = function
   ...
 | Mul (e1, e2) ->
     eval env e1 * eval env e2

let rec print chan = function
   ...
 | Mul (e1, e2) ->
     fprintf chan "(%a * %a)" print e1 print e2
```

This problem is the dual of adding a new method in the object implementation. When a new kind of expression is added to the union, each of the functions must be updated, leading to a scattering of updates throughout the program. Fortunately, the type checker will help find each of the functions—each function to be updated will likely cause an "incomplete pattern match" warning.

We can summarize the differences in the following table.

	Unions	Objects
	One type definition, with a case for each kind of thing; one function for each operation.	One class for each kind of thing, one method for each operation.
Adding a function	Define the new function, the original code is unchanged.	Update *each* class definition. (However, see Exercise 17.11.)
Adding a case	Modify the type definition. Update *each* function. (However, see Exercise 17.12.)	Define the new class, the original code is unchanged.

There is no single good solution; modifications that are easy in one style may be difficult in the other style. The choice of which style to use should be based on what the desired properties are.

Pattern matching

Let's turn to a different kind of issue. One advantage of the disjoint union specification is that pattern matching is well-supported. Suppose we wish to write an "optimizer" for expressions based on the distributive law. We'll specifically use the following equivalences.

e_1 * e_2 + e_1 * e_3 = e_1 * (e_2 + e_3)
e_2 * e_1 + e_3 * e_1 = e_1 * (e_2 + e_3)

Evaluating the expression on the right is likely to be more efficient than evaluating the left expression because e_1 is computed only once.

The optimizer is a function from expressions to expressions that is intended to preserve the result of evaluation. Here is how we might implement it.

```
let rec optimize = function
   Add (e1, e2) ->
      (match optimize e1, optimize e2 with
          Mul (a, b), Mul (c, d)
        | Mul (b, a), Mul (d, c) when a = c ->
            Mul (a, Add (b, d))
        | e1, e2 ->
            Add (e1, e2))
 | If (e1, e2, e3) ->
     If (optimize e1, optimize e2, optimize e3)
 | ...
```

Implementing a similar operation with objects is more difficult. We specifically wish to consider cases where the subexpressions of an object of type `add_exp` have type `mul_exp`. However, the class `add_exp` is defined so that its subexpressions are of type `exp`, and otherwise there is no way to determine what they are. The general problem is an instance of narrowing, discussed previously in Section 14.9.

One solution is to define an explicit type of descriptions for the various kinds of objects. For example, the description of a `mul_exp` might be `Mul (e1, e2)`, where `e1` and `e2` are the subexpressions. In addition, we must add a `describe` method to each of the different expression classes.

```
type 'a description =
   Mul of 'a * 'a
 | Other

class virtual exp =
  object ('self)
    method virtual eval : int env -> int
    method virtual optimize : exp
    method describe : exp description = Other
  end

class add_exp (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    inherit exp
    method eval env = e1#eval env + e2#eval env
    method optimize =
      let e1 = e1#optimize in
```

```
      let e2 = e2#optimize in
        match e1#describe, e2#describe with
          Mul (a, b), Mul (c, d)
        | Mul (b, a), Mul (d, c) when a = c ->
            new mul_exp a (new add_exp b d)
        | _ ->
            new add_exp e1 e2
  end;;

class mul_exp (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method eval env = e1#eval env * e2#eval env
    method optimize = new mul_exp e1#optimize e2#optimize
    method describe = Mul (e1, e2)
  end
```

The type definitions in this example reflect the fact that we want only to implement the specifications based on the distributive law. If we expect to do general optimizations, it may be useful to describe all of the different kinds of expressions, so that the type `'a description` contains a case for each of the different kinds of expressions.

Of course, if we did, we would find that the type `'a description` would be nearly equivalent to the union type exp, and we would find that the object-oriented implementation contains a fragment of the alternative implementation.

One might argue that the object-oriented style of implementation is pointless because, in the end, it might still require implementing a fragment based on the "standard" functional representation. However, this is not the case. There are many good reasons to use objects, and the choice of style is based on the needs of the specific project.

One of the principal reasons why functional constructs appear in object-oriented programs is because OCaml is a functional programming language. Regardless of stylistic preferences, proficient OCaml programmers use the best tools possible, and this means using the constructs that are appropriate to the problem at hand. In some cases, this may mean traditional functional programming; in others, it may require extreme object-oriented programming. The beauty of OCaml is that one is not forced into a particular methodology, be it imperative, functional, object-oriented, or something else. Nearly any approach you might take in another language, you can take in OCaml—and, most likely, do it better.

17.5 Exercises

Exercise 17.1 The restriction about free type variables applies only to non-private method types. Which of the following definitions are legal? For those that are legal, give their types. For those that are not legal, explain why.

1. `class c1 = object val x = [] end;;`
2. `class c2 = object val x = ref [] end;;`
3. `class c3 x = object val y = x end`
4. `class c4 x = object val y = x method z = y end`
5. `class c5 x = object val y = x + 1 method z = y end`
6. `class c6 (x : 'a) = object constraint 'a = int method y = x end;;`

Exercise 17.2 Write an imperative version of a polymorphic map. A newly-created map should be empty. The class should have the following type.

```
class ['a, 'b] imp_map : ('a -> 'a -> ordering) ->
  object
    method find   : 'a -> 'b
    method insert : 'a -> 'b -> unit
  end
```

Exercise 17.3 Reimplement the polymorphic map class from page 221 so that the class takes no arguments, and `compare` is a virtual method. Define a specific class `int_map` where the keys have type `int` with the usual ordering.

Exercise 17.4 In the class type definition `['a] tree` on page 225, the method `add` has type `'a -> 'a tree`. What would happen if we defined the class type as follows?

```
class type ['a] self_tree =
  object ('self)
    method add : 'a -> 'self
    method mem : 'a -> bool
  end
```

Exercise 17.5 In the implementations for the `['a] node` and `['a] leaf` classes in Section 17.2, the function `compare` is threaded through the class definitions. Implement a functor `MakeTree`, specified as follows.

```
type ordering = Smaller | Equal | Larger

module type CompareSig = sig
   type t
   val compare : t -> t -> ordering
end;;

class type ['a] tree =
  object ('self)
    method add : 'a -> 'a tree
    method mem : 'a -> bool
  end;;
```

```
module MakeTree (Compare : CompareSig)
  : sig val empty : Compare.t tree end =
struct ··· end
```

Exercise 17.6 Instead of defining a class type `class type ['a] tree`, we could have specified it as a virtual class like the following.

```
class virtual ['a] virtual_tree =
  object (self : 'self)
    method virtual add : 'a -> 'a virtual_tree
    method virtual mem : 'a -> bool
  end;;
```

Are there any advantages or disadvantages to this approach?

Exercise 17.7 Which of the following class definitions are legal? Explain your answers.

1. ```
   class [+'a] cl (x : 'a) =
     object (self : 'self)
       val f : 'a -> unit = fun x -> ()
       method value : unit -> 'a = fun () -> x
     end
   ```
2. ```
   class [+'a] cl =
     object (self : 'self)
       method f : 'a -> unit = fun x -> ()
     end
   ```
3. ```
   class [+'a] cl =
     object (self : 'self)
       method private f : 'a -> unit = fun x -> ()
     end
   ```
4. ```
   class [+'a] cl =
     object (self : 'a)
       method copy : 'a = {< >}
     end
   ```
5. ```
   class [-'a] cl (x : 'a) =
     object (self : 'self)
       val mutable y = x
       method f x = y <- x
     end;;
   ```
6. ```
   class foo = object end
   class ['a] cl (x : 'a) =
     object
       constraint 'a = #foo
       method value : #foo = x
     end
   ```
7. ```
   class foo = object end
   class [-'a] cl (x : #foo as 'a) =
     object
       method value : #foo = x
     end
   ```
   ```

Exercise 17.8 Consider the following class definitions.

```
class ['a] alt_animal_pair1 (p : 'a) =
  object (self : 'self)
    constraint 'a = ('b, 'b) #pair
    constraint 'b = #animal
    method sleep =
      let a1, a2 = p#value in
      a1#sleep; a2#sleep
  end;;
```

```
class ['a] alt_animal_pair2
  (a1 : 'b) (a2 : 'c) =
  object (self : 'self)
    inherit ['b, 'c] pair a1 a2
    constraint 'a = 'b * 'c
    constraint 'b = #animal
    constraint 'c = #animal
    method sleep =
      a1#sleep; a2#sleep
  end;;
```

1. The type variable `'b` is not a type parameter of `alt_animal_pair1`. Why is the definition legal?
2. Is the type `['a] alt_animal_pair1` covariant, contravariant, or invariant in `'a`?
3. Suppose we have a class `cat` that is a subtype of `animal`. What is the type of the following expression?

```
new alt_animal_pair2 (new dog "Spot") (new cat "Fifi");;
```

4. What happens if the line `constraint 'a = 'b * 'c` is left out of the class definition for `alt_animal_pair2`?
5. What if the line is replaced with `constraint 'a = 'b -> 'c`?
6. In principle, is it ever necessary for a class to have more than one type parameter?

Exercise 17.9 In the object implementation of the evaluator in Figure 17.3, the method `eval` takes an environment of exact type `int env`. Suppose we try to change it to the following definition.

```
class type exp =
  object ('self)
    method eval : int #env -> int
  end

class int_exp (i : int) : exp =
  object (self : 'self)
    method eval (_ : int #env) = i
  end;;
...
```

1. The new type definition is accepted, but the class definition `int_exp` is rejected. How can it be fixed?
2. Are there any advantages to the new definition?

Exercise 17.10 Consider the following class definition.

```
# class type ['a] c1 = object method f : c2 -> 'a end
  and c2 = object method g : int c1 end;;
class type ['a] c1 = object constraint 'a = int method f : c2 -> 'a end
and c2 = object method g : int c1 end
```

Unfortunately, even though the class type `['a] c1` should be polymorphic in `'a`, a type constraint is inferred that `'a = int`. The problem is that polymorphic type definitions are not polymorphic *within* a recursive definition.

1. Suggest a solution to the problem, where class type `c1` is truly polymorphic.

2. The following definition is rejected.

```
# class type ['a] c1 = object method f : c2 -> 'a end
  and c2 = object method g : 'a. 'a c1 -> 'a end;;
Characters 79-94:
and c2 = object method g : 'a. 'a c1 -> 'a end;;
                           ^^^^^^^^^^^^^^^
This type scheme cannot quantify 'a :
it escapes this scope.
```

   The problem arises from the same issue—the class `['a] c1` is not polymorphic within the recursive definition, so the type `'a. 'a c1 -> 'a` is rejected.

   Suggest a solution to this problem.

Exercise 17.11 As discussed in Section 17.4, one problem with object-oriented implementations is that adding a new functionality to a class hierarchy might require modifying all the classes in the hierarchy. *Visitor design patterns* are one way in which this problem can be addressed.

A *visitor* is defined as an object with a method for each of the kinds of data. For the type exp, a visitor would have the following type.

```
class type visitor =
  object ('self)
    method visit_int : int_exp -> unit
    method visit_var : var_exp -> unit
    method visit_add : add_exp -> unit
    method visit_if  : if_exp  -> unit
    method visit_let : let_exp -> unit
  end;;
```

The class type exp is augmented with a method `accept : visitor -> unit` that guides the visitor through an expression, visiting every subexpression in turn. Here is a fragment of the code.

```
class type exp =
  object ('self)
    method eval : int env -> int
    method accept : visitor -> unit
  end;;

class int_exp (i : int) =
  object (self : 'self)
    method eval (_ : int env) = i
```

```
    method accept visitor = visitor#visit_int (self :> int_exp)
  end

class add_exp (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method eval env = e1#eval env + e2#eval env
    method accept visitor =
      visitor#visit (self :> add_exp);
      e1#accept visitor;
      e2#accept visitor
  end
...
```

1. One problem with this approach is the order of definitions. For example, the class type `visitor` refers to the class `add_exp`, which refers back to the `visitor` type in the definition of the method `accept`.

   (a) We could simplify the types. Would the following definition be acceptable?

   ```
   class type exp =
     object ('self)
       method eval : int env -> int
       method accept : visitor -> unit
     end

   and visitor =
     object ('self)
       method visit_int : exp -> unit
       method visit_var : exp -> unit
       ...
     end
   ```

   (b) What is a better way to solve this problem?

2. The class type `visitor` has one method for each specific kind of expression. What must be done when a new kind of expression is added?

As defined, the visitor pattern is not very useful because the classes do not provide any additional information about themselves. Suppose we add a method `explode` that presents the contents of the object as a tuple. Here is a fragment.

```
class type exp = object ··· end
and visitor = object ··· end

and int_exp_type =
  object ('self)
    inherit exp
    method explode : int
  end

and add_exp_type =
  object ('self)
    inherit exp
    method explode : exp * exp
```

```
  end
...
```

3. Since the method explode exposes the internal representation, it isn't really necessary for the accept methods to perform the recursive calls. For example, we could make the following definition, and assume that the visitor will handle the recursive calls itself.

```
class add_exp (e1 : #env) (e2 : #env) : add_exp_type =
  object (self : 'self)
    method eval env = e1#eval env + e2#eval env
    method accept visitor = visitor#visit_add (self :> add_exp_type)
    method explode = e1, e2
  end
```

   What are the advantages of this approach? What are its disadvantages?

4. Another approach is, instead of passing the objects directly to the visitor, to pass the exploded values as arguments. Here is the new visitor type definition.

```
class type visitor =
  object ('self)
    method visit_int : int -> unit
    method visit_add : exp -> exp -> unit
    ...
  end
```

   What are the advantages of this approach? What are its disadvantages?

5. Write a visitor to print out an expression.

The visitors we have specified are imperative. It is also possible to write pure visitors that compute without side-effects. The visitor has a polymorphic class type parameterized over the type of values it computes. As discussed in Exercise 17.10, a recursive definition does not work, so we break apart the recursive definition.

```
class type ['a, 'exp] pre_visitor =
  object ('self)
    method visit_int : int -> 'a
    method visit_var : string -> 'a
    method visit_add : 'exp -> 'exp -> 'a
    method visit_if  : 'exp -> 'exp -> 'exp -> 'a
    method visit_let : string -> 'exp -> 'exp -> 'a
  end;;

class type exp =
  object ('self)
    method eval   : int env -> int
    method accept : 'a. ('a, exp) pre_visitor -> 'a
  end

class type ['a] visitor = ['a, exp] pre_visitor
```

6. Rewrite the class definitions to implement the new `accept` methods.

7. Write an evaluator as a pure visitor `eval_visitor`. The `eval_visitor` is not allowed to call the method `eval`, and it is not allowed to use assignment or any other form of side-effect.

Exercise 17.12 We also stated in Section 17.4 that one problem with the traditional functional representation is that it is hard to add a new case to a union, because each of the functions that operate on the data must also be updated.

One way to address this is through the use of *polymorphic variants*, discussed in Section 6.5. Polymorphic variants can be defined as "open" types that can be later extended. For the evaluator example, here is how we might define the initial type of expressions.

```
type 'a exp1 = 'a constraint 'a =
 [> `Int of int
  | `Var of string
  | `Add of 'a * 'a
  | `If  of 'a * 'a * 'a
  | `Let of string * 'a * 'a ]
```

The type `'a exp` is an open type that includes at least the cases specified in the type definition. The type of an evaluator is defined as follows, where the module `Env` is defined on page 236.

```
type 'a evaluator = int Env.t -> 'a -> int
```

1. Write an evaluator (of type `'a exp evaluator`).

We can extend the type of expressions by adding an additional constraint that specifies the new kinds of expressions. For example, this is how we might add products as a kind of expression.

```
type 'a exp2 = 'a
   constraint 'a = 'a exp1
   constraint 'a = [> `Mul of 'a * 'a ]
```

The next step is to define an evaluator of type `'a exp2 evaluator`. However, we don't want to reimplement it completely—we would like to be able to re-use the previous implementation. For this, we need a kind of "open recursion." Let's define a *pre-evaluator* as a function of the following type. That is, a pre-evaluator takes an evaluator as an argument for computing values of subterms.

```
type 'a pre_evaluator = 'a evaluator -> 'a evaluator

let pre_eval1 eval_subterm env = function
   `Add (e1, e2) -> eval_subterm env e1 + eval_subterm env e2
 | ...
```

The function has type `pre_eval1 : 'a exp1 pre_evaluator`.

2. Write the complete definition of `pre_eval1`.

3. Write a function `make_eval` that turns a pre-evaluator into an evaluator. Hint: this is a kind of "fixpoint" definition, explored in Exercise **??**.

```
val make_eval : 'a pre_evaluator -> 'a evaluator
```

4. The pre-evaluator `pre_eval2 : 'a exp2 pre_evaluator` can be implemented as follows.

```
let pre_eval2 eval_subterm env = function
   `Mul (e1, e2) -> eval_subterm env e1 * eval_subterm env e2
 | e -> pre_eval1 eval_subterm env e
```

   Implement the evaluator `eval2 : 'a exp2 evaluator` in terms of `pre_eval2`.

Syntax

Whenever learning a new language it is useful to have a guide to the language's syntax. The OCaml reference manual describes the syntax using a context-free grammar. The syntax we give here uses the same format, and it also serves as an index into the book, listing the pages where specific syntactical features are introduced. The OCaml language is still evolving; you should consider the reference manual to be the authoritative definition. However, it is likely that the syntax here will be very similar to the one you are using.

.1 Notation

The grammar is specified in standard notation called Backus-Naur Form (BNF), where a grammar consists of a start symbol, a set of nonterminal symbols, a set of terminal symbols, and a set of productions. The terminal symbols represent the basic words of input, like keywords, numbers, special symbols, *etc*. A production the form s ::= $s_1 s_2 \cdots s_n$, where s is a nonterminal, and $s_1 s_2 \cdots s_n$ is a sequence of symbols. For example, the following production says, informally, that an expression can be a conditional that starts with the keyword `if`, following by an expression, followed by the keyword `then`, *etc*.

expression `::=` **`if`** *expression* **`then`** *expression* **`else`** *expression*

We write terminal symbols in a fixed-width font `if`, and nonterminals in a slanted font *expression*.

By convention, a vertical bar on the right hand side of a production is shorthand for multiple productions. The following two grammars are equivalent.

Short form			Meaning		
d	::=	0 | 1	*d*	::=	0
			d	::=	1

For brevity, we'll also use meta-notation for some kinds of repetition. We use Greek letters for sequences of symbols.

Description	Meta-notation
Optional (zero or one)	$[\![\beta]\!]^?$
Repetition (zero or more)	$[\![\beta]\!]^*$
Repetition (one or more)	$[\![\beta]\!]^+$
Repetition with separator	$[\![\beta]\!]^*_{(sep=\alpha)}$
Repetition with separator and optional terminator	$[\![\beta]\!]^*_{(sep=\alpha]}$
Repetition with separator and optional prefix	$[\![\beta]\!]^*_{[sep=\alpha)}$
Choice	$[\![\alpha\|\beta]\!]$
Character choice	$[\![\texttt{a}\|\texttt{b}\|\texttt{c}]\!]$
Inverted character choice	$[\![\texttt{\^{}a}\|\texttt{b}\|\texttt{c}]\!]$
Character range	$[\![\texttt{0..9}]\!]$

When the meta-brackets enclose a single symbol $[\![s]\!]^+$, we will often omit them, writing s^+ instead.

To summarize the repetition forms, a superscript $\cdot^*$ means zero or more repetitions, a superscript $\cdot^+$ means zero or more, and a superscript $\cdot^?$ means zero or one. A subscript $\cdot_{(sep=s)}$ means that the repetitions are separated by a symbol s. The subscript $\cdot_{(sep=s]}$ means that the final element is optionally followed by the separator; and $\cdot_{[sep=s)}$ means that separator can also be used as a prefix to each element.

Choice is allowed, in $[\![\alpha|\beta]\!]$ the alternatives are α and β. When used with characters, a leading "hat" $[\![\texttt{\^{}a}|\texttt{b}|\texttt{c}]\!]$ means any character in the ASCII character set *except* `a`, `b`, or `c`. A character range $[\![c_1\texttt{..}c_2]\!]$ includes all characters with ASCII codes between c_1 and c_2, inclusive.

For some examples, consider the following hypothetical grammar.

$$
\begin{array}{rcl}
e & ::= & [\![\texttt{0..9}|\texttt{\_}]\!]^+ \\
 & | & \texttt{(}\ [\![\ e\ ]\!]^+_{(sep=\texttt{,})}\ \texttt{)} \\
 & | & \texttt{[}\ [\![\ e\ ]\!]^*_{(sep=\texttt{;}]}\ \texttt{]}
\end{array}
$$

The following table lists some sentences, where we use the term "legal" to mean that the sentence is in the language of this hypothetical grammar.

Legal sentences	Illegal sentences
`( 0, 1, 72_134 )`	`( )`
`[ ]`	`[;]`
`[ 3; 2; 6 ]`	`[3 2; 6]`
`[ 14; 55; 237; ]`	

The syntax of OCaml can be placed into several categories: expressions, type expressions, structure expressions, structure types, module expressions and types, and class expressions and types. We'll cover each of these, but first it is useful to describe the terminal symbols.

.2 Terminal symbols (lexemes)

The terminal symbols are the "words" that make up a program, including keywords, numbers, special symbols, and other things.

.2.1 Whitespace and comments

Whitespace includes the following characters: space, tab, carriage return, newline, and form feed. When it occurs outside of character and string literals, whitespace is used to separate the terminal symbols, but is otherwise ignored. Whitespace within character and string literals is treated as a constant.

Comments begin with the two-character sequence `(*` and end with the two-character sequence `*)`. Comments are treated as whitespace. The text within a comment is mostly unrestricted. However, comments may be nested, and comment delimiters that occur within text that appears like a string or character literal are ignored. The reason for this is to allow arbitrary OCaml code to be commented simply by enclosing it in comment delimiters without requiring additional editing.

The following comments are properly delimited.

```
(* This is (* a nested *) comment *)
(* let a = (* b in *) c *)
(* let a = "a delimiter (* in a string" in b *)
```

The following lines are not properly terminated comments.

```
(* This is not a (* nested comment *)
(* let a = "a delimiter" (* "in a string" in b *)
```

.2.2 Keywords

The following table lists the keywords in OCaml.

Keyword	Page	Keyword	Page	Keyword	Page
`and`	19	`if`	10	`of`	49
`as`	31, 56	`in`	15	`open`	117
`assert`	92	`include`	130	`or`	10
`asr`	6	`inherit`	184	`private`	165
`begin`	17	`initializer`	164	`rec`	18
`class`	179	`land`	6	`sig`	127
`constraint`	157	`lazy`	73	`struct`	125
`do`	62	`let`	15	`then`	10
`done`	62	`lor`	6	`to`	62
`downto`	62	`lsl`	6	`true`	9
`else`	10	`lsr`	6	`try`	88
`end`	17	`lxor`	6	`type`	49
`exception`	87	`match`	29	`val`	114, 155
`external`		`method`	155	`virtual`	198
`false`	9	`mod`	6	`when`	31
`for`	62	`module`	125	`while`	62
`fun`	16	`mutable`	78, 161	`with`	29
`function`	30	`new`	180		
`functor`	145	`object`	155		

The following symbols are also keywords.

Keyword	Page	Keyword	Page	Keyword	Page
`!=`	9	`..`	156	`??`	
`#`	163	`:`	38	`[`	42
`&`	10	`::`	42	`[<`	55
`&&`	10	`:=`	61	`[>`	55
`'`	37	`:>`	166	`[|`	80
`(`	17	`;`	42, 61	`]`	42
`)`	17	`;;`	5	`_`	32
`*`	6	`<`	9	`` ` ``	55
`+`	6	`<-`	80, 79, 80, 162	`{`	77
`,`	41	`=`	9	`{<`	159
`-`	6	`>`	9	`|`	29
`-.`	7	`>]`		`|]`	80
`->`	16	`>}`	159	`~`	55
`.`	77	`?`	21		

.2.3 Prefix and infix symbols

Prefix and infix symbols are special identifiers that start with a special character, and include a sequence of operator characters.

infixSymbol ::= `[[=|<|>|@|^|||&|+|-|*|/|$|%]]`*operatorChar**
prefixSymbol ::= `[[!|?|~]]`*operatorChar**
operatorChar ::= `[[!|$|%|&|*|+|-|.|/|:|<|=|>|?|@|^|||~]]`

.2.4 Integer literals

Integers can be specified in several radixes and sizes. An integer has four possible parts:

1. an optional leading minus sign (default nonnegative);
2. an optional radix specifier: `0x` for hexadecimal, `0o` for octal, or `0b` for binary (default decimal);
3. a sequence of digits that may contain optional non-leading underscores `_` (the underscores are ignored);
4. an optional size specifier: `l` for `int32`, `L` for `int64`, or `n` for `nativeint` (default `int`).

integerLiteral ::= (page 6)
| `-`$^?$ `[[0..9]][[0..9|_]]`*`[[l|L|n]]`$^?$
| `-`$^?$ `0[[x|X]][[0..9|a..f|A..F]][[0..9|a..f|A..F|_]]`*`[[l|L|n]]`$^?$
| `-`$^?$ `0[[o|O]][[0..7]][[0..7|_]]`*`[[l|L|n]]`$^?$
| `-`$^?$ `0[[b|B]][[0..1]][[0..1|_]]`*`[[l|L|n]]`$^?$

.2.5 Floating-point literals

Floating-point numbers are written in decimal. A floating-point literal has four possible parts.

1. an optional leading minus sign (default nonnegative);
2. a non-empty sequence of decimal digits;
3. a decimal point followed by an optional sequence of decimal digits;
4. an exponent e followed by a sequence of decimal digits.

A decimal point or exponent is required, but both are not necessary.

floatLiteral	::=	$\texttt{-}^?$ *decimal* . *_decimal*$^?$ *exponent*	(page 7)
	|	$\texttt{-}^?$ *decimal exponent*	
	|	$\texttt{-}^?$ *decimal* . *_decimal*$^?$	
_decimal	::=	$[\![\texttt{0..9}\|\texttt{\_}]\!]^+$	
decimal	::=	$[\![\texttt{0..9}]\!][\![\texttt{0..9}\|\texttt{\_}]\!]^*$	
exponent	::=	$[\![\texttt{e}\|\texttt{E}]\!][\![\texttt{-}\|\texttt{+}]\!]^?$*decimal*	

.2.6 Character literals

Characters are delimited by single quotes. A literal can be a single ASCII character, or it can be an escape sequence.

charLiteral	::=	'*normalChar*'	(page 8)
	|	'*escapeChar*'	
normalChar	::=	$[\![\texttt{\^{}'}\|\texttt{\textbackslash}]\!]$	
escapeChar	::=	\n	(newline)
	|	\r	(carriage return)
	|	\t	(tab)
	|	\b	(backspace)
	|	\ *space*	(space)
	|	\\	(backslash)
	|	\'	(single quote)
	|	\"	(double quote)
	|	\ddd	(decimal code ddd)
	|	\x$hh$	(hexadecimal code hh)
d	::=	$[\![\texttt{0..9}]\!]$	
h	::=	$[\![\texttt{0..9}\|\texttt{a..f}\|\texttt{A..F}]\!]$	

.2.7 String literals

A string is a sequence of characters delimited by double quotes.

stringLiteral	::=	"*stringChar**"	(page 8)
stringChar	::=	*normalStringChar* | *escapeChar*	
normalStringChar	::=	$[\![\texttt{\^{}'}\|\texttt{"}\|\texttt{\textbackslash}]\!]$	

There is no practical limit on the length of string literals.

.2.8 Identifiers

Identifiers come in two kinds, distinguished by the case of the first letter: a *lident* is an identifier that starts with a lowercase letter or an underscore _, and *Uident* is an identifier starting with an uppercase letter.

lident ::= ⟦a..z|_⟧⟦a..z|A..Z|0..9|_|'⟧* (page 15)
Uident ::= ⟦A..Z⟧⟦a..z|A..Z|0..9|_|'⟧*

Accented letters from the ISO Latin 1 set are also allowed (not shown in this grammar). There is no practical limit on identifier length.

.2.9 Labels

Labels are used for labeled parameters and arguments. A label starts with a tilde ~ (for a required argument), or a question mark ? (for an optional argument), followed by a lowercase identifier, followed by a colon.

label ::= ~*lident*: (page 21)
optlabel ::= ?*lident*:

.2.10 Miscellaneous

The program may also contain line number directives in C-preprocessor style.

lineNumber ::= #⟦0..9⟧⁺ *stringLiteral*?

Line number directives affect the reporting of warnings and errors, but otherwise they behave as whitespace.

.3 Names

The reference manual uses the term *identifier* to refer to the string of characters that spells out a name. A *name* is used to refer to some construct in the language, like a value, a constructor, *etc*. There are two kinds of identifiers, those that begin with a lowercase letter, and those that begin in uppercase. There are many kinds of names, classified by what they refer to.

.3.1 Simple names

A *value* is the result of evaluating an expression. The values include numbers, characters, strings, functions, tuples of values, variant values (elements of a union), and objects. A *valueName* is the name that can occur in a let-expression `let` *valueName* `=` *expression*. A value name is a *lident* or an operator name enclosed in parentheses.

valueName	::=	*lident*	(page 15)
	|	`(` ⟦*prefixSymbol*|*infixSymbol*|*infixOther*⟧ `)`	(page 20)
infixOther	::=	`*` | `=` | `or` | `&` | `:=` | `mod` | `land`	
	|	`lor` | `lxor` | `lsl` | `lsr` | `asr`	

There are several other kinds of names, all of them are either lowercase or uppercase identifiers. The capitalization of the nonterminal names corresponds to the capitalization of the name.

Language construct	nonterminal		case
Constructor	*ConstructorName*	::=	*Uident*
Polymorphic variants	*VariantName*	::=	*Uident*
Exceptions	*ExceptionName*	::=	*Uident*
Type constructors	*typeName*	::=	*lident*
Labels	*labelName*	::=	*lident*
Record fields	*fieldName*	::=	*lident*
Classes	*className*	::=	*lident*
Methods	*methodName*	::=	*lident*
Object fields	*objectFieldName*	::=	*lident*
Modules	*ModuleName*	::=	*Uident*
Module types	*ModuleTypeName*	::=	*Uident*|*lident*

.3.2 Path names

Certain kinds of names can be qualified using module path prefix. For example, the name `List.map` is the name of the function `map` in the `List` module. A module path is simply a list of module names separated by a period. We'll also use a form *OptModulePathPrefix* where the final module is followed by a period.

modulePath	::=	⟦*ModuleName*⟧$^{+}_{(sep=\texttt{.})}$
optModulePathPrefix	::=	⟦*ModuleName*`.`⟧*

The qualified names are as follows.

valuePath	::=	*optModulePathPrefix*	*valueName*
ConstructorPath	::=	*optModulePathPrefix*	*ConstructorName*
field	::=	*optModulePathPrefix*	*fieldName*
classPath	::=	*optModulePathPrefix*	*className*

When a type is named, the module path may also contain functor applications.

extendedModulePath ::=
  ModuleName
| *extendedModulePath*`.`*ModuleName*
| *extendedModulePath*`(`*extendedModulePath*`)`

optExtendedModulePathPrefix ::= ⟦*extendedModulePath*`.`⟧$^{?}$

Types can use an extended prefix.

typePath ::=
    optExtendedModulePathPrefix typeName
moduleTypePath ::=
    optExtendedModulePathPrefix ModuleTypeName

.4 Expressions

There are many kind of expressions in the language.

expression ::=

	valuePath	
|	*constant*	
|	`(` *expression* `)`	(page 17)
|	`begin` *expression* `end`	(page 17)
|	`(`*expression* `:` *typeExpression*`)`	(page 38)
|	$[\![$*expression*$]\!]^{+}_{(sep=,)}$	(page 41)
|	*ConstructorName expression*	(page 49)
|	*‘VariantName expression*	(page 55)
|	*expression* `::` *expression*	(page 42)
|	`[` $[\![$*expression*$]\!]^{*}_{(sep=;]}$ `]`	(page 42)
|	`[|` $[\![$*expression*$]\!]^{*}_{(sep=;]}$ `|]`	(page 80)
|	`{` $[\![$*fieldName* `=` *expression*$]\!]^{+}_{(sep=;]}$ `}`	(page 77)
|	`{` *expression* `with` $[\![$*fieldName* `=` *expression*$]\!]^{+}_{(sep=;]}$ `}`	(page 78)
|	*expression* $[\![$*argument*$]\!]^{+}$	(page 17)
|	*prefixSymbol expression*	(page 6)
|	*expression infixSymbol expression*	(page 6)
|	*expression* `.` *fieldPath*	(page 77)
|	*expression* `.` *fieldPath* `<-` *expression*	(page 79)
|	*expression* `.(` *expression* `)`	(page 80)
|	*expression* `.(` *expression* `)` `<-` *expression*	(page 80)
|	*expression* `.[` *expression* `]`	(page 80)
|	*expression* `.[` *expression* `]` `<-` *expression*	(page 80)
|	*expression*`;` *expression*	(page 61)
|	`if` *expression* `then` *expression* `else` *expression*	(page 10)
|	`while` *expression* `do` *expression* `done`	(page 62)
|	`for` *valueName* `=` *expression* $[\![$`to`|`downto`$]\!]$ *expression* `do` *expression* `done`	(page 62)
|	`match` *expression* `with` *patternMatching*	(page 29)
|	`try` *expression* `with` *patternMatching*	(page 88)
|	`function` *patternMatching*	(page 30)
|	`fun` *multipleMatching*	(page 16)
|	`let` `rec`$^{?}$ $[\![$*letBinding*$]\!]^{+}_{(sep=\texttt{and})}$ `in` *expression*	(page 15)

expression ::= ···

	`new` *classPath*	(page 180)
|	`object` *classBody* `end`	(page 155)
|	*expression* `#` *methodName*	(page 163)
|	*objectFieldName*	(page 155)
|	*objectFieldName* `<-` *expression*	(page 162)
|	`(`*expression* `:>` *typeExpression*`)`	(page 166)
|	`(`*expression* `:` *typeExpression* `:>` *typeExpression*`)`	(page 166)
|	`{<` $[\![$*objectFieldName* `=` *expression*$]\!]^{+}_{(sep=;]}$ `>}`	(page 159)
|	`assert` *expression*	(page 92)
|	`lazy` *expression*	(page 73)

An argument to an application can be labeled.

argument	::=	*expression*	
	|	*~labelName*	(page 21)
	|	*~labelName*`:` *expression*	
	|	`?`*labelName*	(page 21)
	|	`?`*labelName*`:` *expression*	

A pattern-matching is a list of cases separated by vertical bars. A leading vertical bar is optional. Each case can be conditioned on a predicate `when` *expression*.

patternMatching ::= $[\![$ *pattern* $[\![$ `when` *expression* $]\!]^{?}$ `->` *expression* $]\!]^{+}_{[sep=|)}$

A *multipleMatching* is used for `fun` and `let` expressions, which allow multiple parameters.

multipleMatching	::=	*parameter*$^{+}$ $[\![$ `when` *expression* $]\!]^{?}$ `->` *expression*
letBinding	::=	*pattern* `=` *expression*
	|	*valueName parameter** $[\![$`:` *typeExpression*$]\!]^{?}$ `=` *expression*

Parameters can be labeled, and they allow only limited pattern matching, not a full case analysis.

parameter ::=

	pattern	
|	*~labelName*	(page 21)
|	`~(`*labelName* $[\![$`:` *typeExpression*$]\!]^{?}$`)`	
|	*~labelName*`:` *pattern*	
|	`?`*labelName*	(page 21)
|	`?(`*labelName* $[\![$`:` *typeExpression*$]\!]^{?}$ $[\![$`=` *expression*$]\!]^{?}$`)`	
|	`?`*labelName*`:` *pattern*	
|	`?`*labelName*`: (`*pattern* $[\![$`:` *typeExpression*$]\!]^{?}$ $[\![$`=` *expression*$]\!]^{?}$`)`	

.4.1 Patterns

A pattern is a template that is used for matching.

pattern ::=		(page 29)
	_	(page 32)
|	*valueName*	
|	*constant*	
|	*pattern* `as` *valueName*	(page 31)
|	`(`*pattern* `:` *typeExpression*`)`	
|	*pattern* `|` *pattern*	(page 31)
|	*ConstructorPath pattern*$^{?}$	(page 50
|	`‘`*VariantName pattern*$^{?}$	(page 55)
|	*#typeName*	
|	$[\![pattern]\!]^{+}_{(sep=,)}$	(page 41)
|	`{` $[\![$*fieldPath* `=` *pattern*$]\!]^{+}_{(sep=;]}$ `}`	(page 78)
|	`[` $[\![pattern]\!]^{+}_{(sep=;]}$ `]`	(page 42)
|	*pattern* `::` *pattern*	(page 42)
|	`[|` $[\![pattern]\!]^{+}_{(sep=;]}$ `|]`	(page 80)

.4.2 Constants

The constant expressions include literals and simple constructors.

constantExpression	::=	*integerLiteral*
	|	*floatLiteral*
	|	*charLiteral*
	|	*stringLiteral*
	|	*ConstructorPath*
	|	`‘`*VariantName*
	|	`true`
	|	`false`
	|	`[]`
	|	`()`

.4.3 Precedence of operators

The following table lists the operator precedences from highest to lowest. The precedence of operator symbols is determined the longest prefix in the following table. For example, an operator `**@@` would have the precedence of `**`, but an operator `*@@` would have the precedence of `*`.

Operator	Associativity
`~`, `?`, `!`	none
`.`, `.[`, `.(`	none
function and constructor application, `assert`, `lazy`	left
`-`, `-.` (when used as a unary operator)	none
`**`, `lsl`, `lsr`, `asr`	left
`*`, `/`, `%`, `mod`, `land`, `lor`, `lxor`	left
`+`, `-`	left
`::`	right
`@`, `^`	right
`<`, `<=`, `=`, `==`, `!=`, `<>`, `>=`, `>` and other *infixSymbol* not listed	left
`&`, `&&`	left
`or`, `||`	left
`,`	none
`<-`, `:=`	right
`if`	none
`;`	right
`fun`, `function`, `let`, `match`, `try`	none

.5 Type expressions

Type expressions have the following grammar. Note that there are no type expressions for records and disjoint unions; those types must be named in a type definition before they can be used.

typeExpression ::=

	_	
|	*'lident*	(page 37)
|	`(` *typeExpression* `)`	
|	$[\![$ `?`$^{?}$ *labelName*`:` $]\!]^{?}$ *typeExpression* `->` *typeExpression*	(page 16)
|	$[\![$*typeExpression*$]\!]^{+}_{(sep=*)}$	
|	*typePath*	
|	*typeExpression typePath*	(page 37)
|	`(` $[\![$*typeExpression*$]\!]^{+}_{(sep=,)}$ `)` *typePath*	(page 37)
|	*typeExpression* `as` *'lident*	(pages 56, 157)
|	*variantType*	
|	`<` $[\![$*methodType*$]\!]^{*}_{(sep=;]}$ `>`	(page 155)
|	`<` $[\![$*methodType*$]\!]^{*}_{(sep=;)}$`; .. >`	(page 156)
|	`#` *classPath*	(page 189)
|	*typeExpression* `#` *classPath*	(page 221)
|	`(` $[\![$*typeExpression*$]\!]^{+}_{(sep=,)}$ `)` `#` *classPath*	

Methods (and record fields) can have polymorphic type.

methodType	::=	*methodName* `:` *polyTypeExpression*	(page 197)
polyTypeExpression	::=	$[\![$ *'lident*$]\!]^{+}$ `.` $]\!]^{?}$ *typeExpression*	

Types for polymorphic variants have several form that depending whether the type is exact, open, or closed.

variantType	::=		(page 55)
		`[` $[\![$*variantTagType*$]\!]^{+}_{[sep=\|)}$ `]`	
	|	`[>` $[\![$*variantTagType*$]\!]^{*}_{[sep=\|)}$ `]`	
	|	`[<` $[\![$*variantTagIntersectionType*$]\!]^{*}_{[sep=\|)}$ `]`	
variantTagType	::=	*typeExpression*	
	|	*'variantName* $[\![$`of` *typeExpression*$]\!]^{?}$	
variantTagIntersectionType	::=	*typeExpression*	
	|	*'variantName* $[\![$`of` $[\![$*typeExpression*$]\!]^{+}_{(sep=\&)}$$]\!]^{?}$	

The precedences of the type operators is given in the following table, from highest precedence to lowest.

Operator	Associativity
Application *typeExpression typePath*	none
`*`	none
`->`	right
`as`	none

.6 Type definitions

A type definition associates a type name with a type expression, forming an abbreviation; or it defines a record type or disjoint union; or it does both. Type definitions can be recursive; the type name being defined is always bound within its own definition. Mutually recursive types are separated with the keyword and.

typeDefinition ::= `type` $[\![$*typeDef*$]\!]^{+}_{(sep=\texttt{and})}$

typeDef ::= *typeParameters*$^{?}$ *typeName*
$[\![$`=` *typeExpression*$]\!]^{?}$
$[\![$`=` *typeRepresentation*$]\!]^{?}$
typeConstraint*

typeParameters ::= *typeParameter*
| `(` $[\![$*typeParameter*$]\!]^{+}_{(sep=\texttt{,})}$ `)`

typeParameter ::= `'`*lident* (page 37)
| `+` `'`*lident* (page 228)
| `-` `'`*lident* (page 228)

A *typeRepresentation* is the definition of a record type or a disjoint union.

typeRepresentation ::= $[\![$*constructorDecl*$]\!]^{+}_{[sep=\texttt{|})}$ (page 49)
| `{` $[\![$*fieldDecl*$]\!]^{+}_{(sep=\texttt{;}]}$ `}` (page 77)
constructorDecl ::= *constructorName* $[\![$ `of` $[\![$*typeExpression*$]\!]^{+}_{(sep=\texttt{*})}$ $]\!]^{?}$
fieldDecl ::= `mutable`$^{?}$ *fieldName* `:` *polyTypeExpression*

A type definition can include any number of type constraints.

typeConstraint ::= `constraint` `'`*lident* `=` *typeExpression* (page 157)

.7 Structure items and module expressions

A *structItem* is an item that can occur in a module definition (within a `struct` $\cdots$ `end` block), or in an implementation file. A *moduleExpression* represents a module.

structItem ::=

	`let rec`$^{?}$ $[\![$*letBinding*$]\!]^{+}_{(sep=\texttt{and})}$	(page 15)
|	`external` *valueName* `:` *typeExpression* `=` *stringLiteral*$^{+}$	
|	*typeDefinition*	
|	*exceptionDefinition*	(page 87)
|	*classDefinition*	(page 179)
|	*classTypeDefinition*	(page 180)
|	`module` *ModuleName moduleParameter**	(page 125)
	$[\![$`:` *moduleType*$]\!]^{?}$ `=` *moduleExpression*	
|	`module type` *ModuleTypeName* `=` *moduleType*	(page 127)
|	`open` *modulePath*	(page 117)
|	`include` *moduleExpression*	(page 130)

moduleExpression ::=

	`struct` $[\![$ *structItem* `;;`$^{?}]\!]^{*}$ `end`	(page 125)
|	`functor` *moduleParameter* `->` *moduleExpression*	(page 145)
|	*moduleExpression* `(`*moduleExpression*`)`	(page 139)
|	`(` *moduleExpression* `)`	
|	`(` *moduleExpression* `:` *moduleType* `)`	

moduleParameter ::=

	`(` *ModuleName* `:` *moduleType* `)`	(page 139)

Exceptions definitions are similar to disjoint unions.

exceptionDefinition ::= (page 87)

	`exception` *constructorName* $[\![$`of` $[\![$*typeExpression*$]\!]^{+}_{(sep=\texttt{*})}]\!]^{?}$
|	`exception` *constructorName* `=` *constructorPath*

.8 Signature items and module types

A *sigItem* is an item that can occur in a module type definition (within a `sig` $\cdots$ `end` block), or in an interface file. A *moduleTypeExpression* represents the type of a module.

sigItem ::=

	`val` *valueName* `:` *typeExpression*	(page 114)
|	`external` *valueName* `:` *typeExpression* `=` *stringLiteral*$^{+}$	
|	*typeDefinition*	
|	`exception` *constructorDecl*	(page 87)
|	*classSpecification*	(page 180)
|	*classTypeDefinition*	(page 180)
|	`module` *ModuleName moduleParameter** `:` *moduleType*	(page 125)
|	`module type` *ModuleTypeName* $[\![$`=` *moduleType*$]\!]^{?}$	(page 127)
|	`open` *modulePath*	(page 117)
|	`include` *moduleExpression*	(page 130)

moduleType ::=

	moduleTypePath	
|	`sig` $[\![$*sigItem* `;;`$^{?}]\!]^{*}$ `end`	(page 127)
|	`functor` *moduleParameter* `->` *moduleType*	(page 145)
|	*moduleType* `with` $[\![$*moduleConstraint*$]\!]^{+}_{(sep=\texttt{and})}$	(page 141)
|	`(` *moduleType* `)`	

moduleConstraint ::= (page 141)

	`type` *typeParameters*$^{?}$ *typePath* `=` *typeExpression*	
|	`module` *modulePath* `=` *extendedModulePath*	

.9 Class expressions and types

Objects, class expressions, and class types are discussed in chapters 14–17.

classExpression ::=

	classPath	
|	`[` $[\![$*typeExpression*$]\!]^{+}_{(sep=\texttt{,})}$ `]` *classPath*	(page 221)
|	`(` *classExpression* `)`	
|	`(` *classExpression* `:` *classType* `)`	
|	*classExpression argument*$^{+}$	(page 181)
|	`fun` *parameter*$^{+}$ `->` *classExpression*	(page 181)
|	`let` `rec`$^{?}$ $[\![$*letBinding*$]\!]^{+}_{(sep=\texttt{and})}$ `in` *classExpression*	(page 181)
|	`object` *selfBinder*$^{?}$ *classField** `end`	(page 155)

classField ::=

	`inherit` *classExpression* $[\![$`as` *valueName*$]\!]^{?}$	(page 184)
|	`val` `mutable`$^{?}$ *objectFieldName*	
	$[\![$`:` *typeExpression*$]\!]^{?}$ `=` *expression*	(page 155)
|	`val` `mutable`$^{?}$ `virtual` *objectFieldName* `:` *typeExpression*	(page 198)
|	`method` `private`$^{?}$ *methodName*	(page 165)
	parameter* $[\![$`:` *typeExpression*$]\!]^{?}$ `=` *expression*	
|	`method` `private`$^{?}$ *methodName*	
	`:` *polyTypeExpression* `=` *expression*	
|	`method` `private`$^{?}$ *methodName* `:` *polyTypeExpression*	
|	`constraint` *typeExpression* `=` *typeExpression*	(page 157)
|	`initializer` *expression*	(page 164)

selfBinder ::=

	(*pattern* $[\![$`:` *typeExpression*$]\!]^{?}$`)`	(page 163)

.9.1 Class types

classType ::=
    $[\![$ `?`$^{?}$ *labelName*`:`$]\!]^{?}$ *typeExpression* `->`$]\!]^{*}$ *classBodyType*

classBodyType ::=
    classPath
  | `[` $[\![$*typeExpression*$]\!]^{+}_{(sep=\texttt{,})}$ `]` *classPath*
  | `object` *selfType*$^{?}$ *classItemType** `end`

classItemType ::=
    `inherit` *classType* (page 186)
  | `val` `mutable`$^{?}$ `virtual`$^{?}$ *objectFieldName* `:` *typeExpression* (page 155)
  | `method` `private`$^{?}$ `virtual`$^{?}$ *methodName* `:` *polyTypeExpression* (page 197)
  | `constraint` *typeExpression* `=` *typeExpression* (page 157)

selfType ::=
    `(` *typeExpression* `)`

Bibliography

[1] *The American Heritage Dictionary of the English Language*. Houghton Mifflin, fourth edition, 2000.

[2] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[4] Dexter Kozen. *The Design and Analysis of Algorithms*. Springer-Verlag, New York, 1991.

[5] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2002. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

[6] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[7] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*. Academic Press, 1981.

[8] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[9] Chris Okasaki. Red-black trees un a functional setting. *Journal of Functional Programming*, 9(4):471–477, May 1999.

[10] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

Index