Introduction to Objective Caml

(Answers to exercises)

Jason Hickey

November 30, 2007

Contents

Chapter 1

Answers to exercises

2 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`

   Well typed. The value is `-1`.

2. `1 - 2 - 3`

   Well typed. Subtraction is left-associative, so the value is `-4`.

3. `1 - - 2`

   Well typed. The value is `3`.

4. `0b101 + 0x10`

   Well typed. The value is `0x15` (`21` in decimal).

5. `1073741823 + 1`

Well typed. On a 32-bit machine, `1073741823` is the maximum integer, so the value is `-1073741824`. On a 64-bit machine, the addition does not overflow, so the result is `1073741824`.

6. *`1073741823.0 + 1e2`*

   Ill typed. The operator `+` is for integer addition only.

7. *`1 ^ 1`*

   Ill typed. The operator `^` is string concatenation.

8. *`if true then 1`*

   Ill typed. The missing `else` branch has type `unit`, which is not compatible with `1`.

9. *`if false then ()`*

   Well typed. The result is `()`.

10. *`if 0.3 -. 0.2 = 0.1 then 'a' else 'b'`*

    Well-typed. On most machines, `0.3 -. 0.2` is very close to, but different from, `0.1`, so the result is `'b'`.

11. *`true || (1 / 0 >= 0)`*

    Well-typed. The value is `true` (since disjunction `||` is a short-circuit operator).

12. *`1 > 2 - 1`*

    Well typed, because `-` has higher precedence than `>`. The result is `false`.

13. *`"Hello world".[6]`*

    Well typed. The value is `'w'`.

14. *`"Hello world".[11] <- 's'`*

Well typed, but the index `11` is out of bounds, so the expression does not evaluate to a value.

15. `String.lowercase "A" < "B"`

    Well typed. The value is `false`.

16. `Char.code 'a'`

    Well typed. The ASCII character code for `'a'` is `97`.

17. `((((()))))`

    Well typed. The value is the unit `()`.

18. `((((*1*))))`

    Well typed. The value is `()`.

19. `((*(((()*))`

    Well typed. The value is `()`.

3 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`

   Legal. The value is `1 : int`.

2. `let x = 1 in let y = x in y`

   Legal. The value is `1 : int`.

3. `let x = 1 and y = x in y`

   Illegal. The variable `x` is undefined in the definition `y = x`.

4. `let x = 1 and x = 2 in x`

   Illegal. The variable x is defined twice.

5. `let x = 1 in let x = x in x`

   Legal. The value is `1 : int`.

6. `let a' = 1 in a' + 1`

   Legal. The value is `2 : int`.

7. `let 'a = 1 in 'a + 1`

   Illegal. Identifiers may not begin with a single quote `'`.

8. `let a'b'c = 1 in a'b'c`

   Legal. The value is `1 : int`.

9. `let x x = x + 1 in x 2`

   Legal. The identifier x represents the argument in the function body `x + 1` and the function itself in `x 2`. The value is `3 : int`.

10. `let rec x x = x + x in x 2`

    Legal. The `rec` modifier has no effect here. The value is `4 : int`.

11. ```
    let (++) f g x = f (g x) in
    let f x = x + 1 in
    let g x = x * 2 in
    (f ++ g) 1
    ```

    Legal. The identifier `++` represents function composition, so the value is `(1 * 2) + 1` `(3 : int)`.

12. `let (-) x y = y - x in 1 - 2 - 3`

    Legal. The value is `3 - (2 - 1)` `(1)`.
    ```

13. `let rec (-) x y = y - x in 1 - 2 - 3`

    Legal. Evaluation does not terminate because the operator - is defined in terms of itself.

14. `let (+) x y z = x + y + z in 5 + 6 7`

    Illegal. Application has higher precedence than +, so the expression in the body is really `5 + (6 7)`, which is ill-typed.

15. `let (++) x = x + 1 in ++x`

    Illegal. All operators starting with a + sign are binary operators.

Exercise 3.2 *What are the values of the following expressions?*

1. `let x = 1 in let x = x + 1 in x`

   The value is `2 : int`.

2. ```
   let x = 1 in
   let f y = x in
   let x = 2 in
   f 0
   ```

   The value of `x` in the function `f` is `1`, so the result is `2 : int`.

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

   The expression `f (x - 1)` refers to the first definition of `f`, so the result is `0 : int`.

4. ```
   let y = 2 in
   let f x = x + y in
   let f x = let y = 3 in f y in
   f 5
   ```

   The second definition `let y = 3` does not affect the first, so the result is `3 + 2 = 5`.
   ```

5.
```
let rec factorial i =
   if i = 0 then 1 else i * factorial (i - 1)
in
   factorial 5
```

The value is $5!$ or 60.

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

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

The easiest way to write this function is as a simple recursive definition.

```
let rec sum n m f =
   if n > m then
      0
   else
      f n + sum (n + 1) m f
```

This function is not tail-recursive. A simple way to fix it is to define an auxiliary function `loop` that collects the intermediate result `x` in an *accumulator*.

```
let sum n m f =
   let rec loop i x =
      if i > m then
         x
      else
         loop (i + 1) (f i + x)
   in
      loop n
```

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.]

The simplest translation of the GCD function is as follows.

```
let rec (%%) n m =
   if m = 0 then
      n
   else if n > m then
      (n - m) %% m
   else
      n %% (m - n)
```

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 $0 \leq$ `i` $<$ `n`. In addition `f 0 < 0` and `f n > 0`. Write a function `search f n` that finds the smallest argument `i` where `f i` $\geq$ `0`.*

A simple solution is to search for the first positive value starting from 0, taking $O(n)$ iterations to find the answer.

```
let search f n =
   let rec loop i =
      if f i >= 0 then
         i
      else
         loop (i + 1)
   in
      loop 0
```

However, it is easy to write a more efficient implementation. An algorithm to find the answer in $O(\log n)$ iterations using a binary search can be defined as follows.

```
let search f n =
   let rec loop i j =
      if i < j - 1 then
         let k = (i + j) / 2 in
            if f k >= 0 then
               loop i k
            else
               loop k j
      else
         j
   in
```

```
loop 0 n
```

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.*
   Answer

```
let empty = fun key -> 0
let add dict key v = fun key' -> if key' = key then v else dict key
let find dict key = dict key
```

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?*

Answer

```
dict1 "x"  =  1
dict2 "x"  =  1
dict3 "x"  =  3
dict4 "x"  =  3
dict1 "y"  =  0
dict2 "y"  =  2
dict3 "y"  =  2
dict4 "y"  =  4
```

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.

For efficiency, we should precompute as much as possible.

```
let r b c =
   let minusb = -. b in
   let bsquared = b *. b in
   let fourc = -4.0 *. c in
      fun a -> (minusb +. sqrt (bsquared -. fourc *. a)) /. (2.0 *. a)
```

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 `=` $(x_0 + c, x_1 + c, x_2 + c, \ldots)$.
     Answer.

     ```
     let (+:) s c = (fun i -> s i + c)
     ```

   - `(-|) : stream -> stream -> stream`*.*

     Subtract two streams pointwise.

     $(x_0, x_1, x_2, \ldots)$ `-|` $(y_0, y_1, y_2, \ldots)$ `=` $(x_0 - y_0, x_1 - y_1, x_2 - y_2 \ldots)$.
     Answer.

     ```
     let (-|) s1 s2 = (fun i -> s1 i - s2 i)
     ```

   - `map : (int -> int) -> stream -> stream`*.*

Apply a function to each element of the stream.

`map` $f\ (x_0, x_1, x_2, \ldots)$ = $(f\ x_0, f\ x_1, f\ x_2, \ldots)$.
Answer.

```
let map f s = (fun i -> f (s i))
```

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.
Answer.

```
let integral s i =
   let rec loop sum j =
      if j = i then
         sum
      else
         loop (sum + s j) (j + 1)
   in
      loop 0
```

4 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
   ```

   Legal. The expression evaluates to `1 : int`.

2. ```
   match 2 with
      1 + 1 -> 2
    | _ -> 3
   ```

   Not legal. The expression `1 + 1` is not a constant, and it is not a pattern.

3. ```
   let _ as s = "abc" in s ^ "def"
   ```

   Legal. This expression is the same as `let s = "abc" in s ^ "def"`; the value is `"abcdef" : string`.
   ```

4. `(fun (1 | 2) as i -> i + 1) 2`

   Legal. The pattern matching `(1 | 2)` is legal but not exhaustive. The value is `3 : int`.

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?

To implement pattern matching for any class of values, one must decide how to define equality on the values being matched. For simple types like integers, strings, *etc*., equality is well-defined. The problem with functions is that function equality is undecidable in general.

As an alternative, it might be possible to compare the program *text* that defines the functions, rather then the functions themselves. However, this would mean than variations in the text, even insignificant ones, would affect the pattern matching. The result would not be very useful.

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* **??**. *How does your code scale as the alphabet gets larger?*

There are two general approaches. One is to write a function to encrypt a string, then compare encrypted strings. The other is to compare the strings character-by-character. For this solution, we'll write the former. This solution grows linearly with the size of the alphabet.

```
(* Encrypt a string *)
let encrypt_char = function
   'A' -> 'C'
 | 'B' -> 'A'
 | 'C' -> 'D'
 | 'D' -> 'B'
 | _ -> raise (Invalid_argument "encrypt_char")

(* Encrypt the string in place *)
let encrypt_string s =
   let len = String.length s in
   let rec loop i =
      if i < len then
         let () = s.[i] <- encrypt_char s.[i] in
         loop (i + 1)
   in
   loop 0

(* Compare the strings *)
let check s1 s2 =
   let () = encrypt_string s1 in
   s1 = s2
```

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`

   The tuple is `(3, 3, -5)` of arity 3.

2. `"ABC", ( 1 , "def" ), ()`

   The tuple has arity 3; it evaluates to itself. The expression `(1, "def")` is a sub-tuple. The expression `()` is the value of type `unit` (though it can be thought of like a tuple of arity zero).

3. `let x = 1 in x + 1, let y = 2 in y + 1, 4`

   The tuple has arity 2; the value is `(2, (3, 4))`.

Exercise 5.2 *What are the types of the following functions?*

1. `let f (x, y, z, w) = x + z`

   The type is `f : int * 'a * int * 'b -> int`.

2. `let f (x, y, z, w) = (w, z, y, x)`

   The type is `f : 'a * 'b * 'c * 'd -> 'd * 'c * 'b * 'a`.

3. `let f [x; y; z; w] = x`

   The type is `f : 'a list -> 'a`.

4. `let f [x; y] [z; w] = [x; z]`

   The type is `f : 'a list -> 'a list -> 'a list`.

5. `let f (x, y) (z, w) = [x; z]`

   The type is `f : 'a * 'b -> 'a * 'c -> 'a list`.

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?*

   The type is `nth : int -> 'a * 'a * 'a -> 'a`

2. *Is there a way to rewrite the function so that it allows the elements of the tuple to have different types?*

   If the elements of the tuple had different types, the function `nth` would return values of different types for different argument `i`. This is not expressible in the type system we have seen. Another argument is that, since the branches of a conditional (like `match`) must have the same type, the elements of the tuple must have the same type.

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.*

```
let find_salary name =
   let rec search = function
       (name', _, salary) :: rest when name' = name -> salary
     | _ :: rest -> search rest
     | [] -> raise (Invalid_argument "find_salary")
   in
      search db
```

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.

   We can implement the function `select` by examining each tuple in order, collecting the matching tuples in an accumulator `found`.

```
let select pred =
   let rec search found = function
      tuple :: rest when pred tuple ->
          search (tuple :: found) rest
```

```
  | _ :: rest -> search found rest
  | [] -> found
in
    search db
```

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`*?*

There are only two kinds of functions with type `'a -> 'a`: the identity function (and all equivalent functions), and functions that does not terminate, for example `(fun x -> raise (Invalid_argument "error"))`. One way to think about it is that the function takes an argument of some arbitrary type `'a`. Since the actual type is not known, the function can do only one of two things: 1) it can return it without change, or 2) it can fail to terminate.

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?*

The problem is the value restriction. The expression `f 0` has type `'_a -> '_a`, so it is not properly polymorphic. The eta-expansion is not a good solution either, because it does not optimize the computation of `g 0`.

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

The only real solution is to "lift" the computation `g x` out of the function `f`, so that it may be computed explicitly.

```
let z = g 0
let v1 = h z "abc"
let v2 = h z 1.2
```

However, this may not always be desirable because it exposed the implementation of the function `f`.

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*.*

In contrast to the `map` function, which reverses the result, it is easier to reverse the argument for the append function.

```
let append l1 l2 =
   let rec rev_append l2 = function
      x :: l1 -> rev_append (x :: l2) l1
    | [] -> l2
   in
      rev_append l2 (List.rev l1)
```

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.*

Since the lists are sorted, we can search them from first to last, skipping a name when it doesn't match the other lists. We'll use `l1`, `l2`, and `l3` for the lists.

```
let rec find_crook l1 l2 l3 =
   match l1, l2, l3 with
      (h1 :: t1), (h2 :: t2), (h3 :: t3) ->
```

```
      if      h1 < h2 || h1 < h3 then find_crook t1 l2 l3
      else if h2 < h1 || h2 < h3 then find_crook l1 t2 l3
      else if h3 < h1 || h3 < h2 then find_crook l1 l2 t3
      else (* all three names h1,h2,h3 are the same *) h1
  | _ ->
      raise (Invalid_argument "no crooks")
```

6 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`].`

   This is a non-tail-recursive version of `map`.

   ```
   let rec map f = function
      Nil -> Nil
    | Cons (h, t) -> Cons (f h, map f t)
   ```

   For a tail-recursive version, we collect the list in reverse order.

   ```
   let rec rev accum = function
      Nil -> accum
    | Cons (h, t) -> rev (Cons (h, accum)) t

   let map f l =
      let rec loop accum = function
         Nil -> rev Nil accum
       | Cons (h, t) -> loop (Cons (f h, accum)) t
      in
         loop Nil l
   ```

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}`].`
   We give the tail-recursive version, using the function `rev` as defined above.

   ```
   let append l1 l2 =
      let rec loop l2 = function
         Cons (h, t) -> loop (Cons (h, l2)) t
       | Nil -> l2
      in
         loop l2 (rev l1)
   ```

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?*

   To add two numbers we use the following equivalences.

$$\begin{aligned} i + 0 &= i \\ i + (j + 1) &= (i + 1) + j \end{aligned}$$

   This gives us the following loop.

```
let rec add i = function
   S j -> add (S i) j
 | Z -> i
```

   The time complexity of the expression `add` n m is $O(m)$.

2. *Write a function to multiply two unary numbers.*

   We can multiply two numbers by repeated summing.

```
let mul n m =
   let rec loop sum = function
      Z -> sum
    | S m -> loop (add sum n) m
   in
      sum Z m
```

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.*

   The comparison can be implemented as a pattern matching on the pair of numbers to be compared.

```
let lt_small i j =
   match i, j with
      Zero, (One | Two | Three)
    | One, (Two | Three)
    | Two, Three ->
         true
    | _ ->
         false
```

   However, this implementation has poor style because of the wildcard matching `_ -> false`. If another number, like `Five`, is added to the type, the implementation of `lt_small` must be changed. It is better to avoid the use of wildcards.

```
let lt_small i j =
   match i, j with
      Zero, (One | Two | Three)
    | One, (Two | Three)
    | Two, Three ->
         true
    | Zero, Zero
    | One, (Zero | One)
    | Two, (Zero | One | Two)
    | Three, (Zero | One | Two | Three) ->
         false
```

2. *Suppose the type* `small` *defines* n *small integers. How does the size of your code depend on* n*?*

   With the implementation style above, the code is quadratic $O(n^2)$.

   One way to reduce the code size in this case is to map the type onto a linear subrange of the integers, then use the builtin comparison.

```
let index_of_small = function
   Four -> 4
 | Three -> 3
```

```
  | Two -> 2
  | One -> 1

let lt_small i j = index_of_small i < index_of_small j
```

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.*

```
let rec eval = function
   Constant i -> i
 | Unary (Neg, e) -> -(eval e)
 | Binary (e1, op, e2) ->
      let i = eval e1 in
      let j = eval e2 in
         match op with
            Add -> i + j
          | Sub -> i - j
          | Mul -> i * j
          | Div -> i / j
```

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
```

The central difference between a set and a dictionary is that a node has both a `'key` and a `'value`. The `add` function is similary to `insert`, and the `find` function is similar to `mem`.

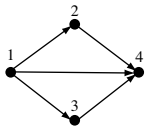
```
type ('key, 'value) dictionary =
   Node of 'key * 'value * ('key, 'value) dictionary * ('key, 'value) dictionary
 | Leaf
```

```
let empty = Leaf

let rec add dict key value =
   match dict with
      Leaf -> Node (key, value, Leaf, Leaf)
    | Node (key', value', left, right) ->
        if key < key' then
           Node (key', value', add left key value, right)
        else if key > key' then
           Node (key', value', left, add right key value)
        else (* key = key' *)
           Node (key, value, left, right)

let rec find dict key =
   match dict with
      Leaf ->
         raise Not_found
    | Node (key', value, left, right) ->
        if key < key' then
           find left key
        else if key > key' then
           find right key
        else (* key = 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.*

To ensure that the algorithm terminates, we need to keep track of which vertices have already been visited using a set `visited` of vertices that have already been visited. The reachability test can be performed by a depth-first-search.

```
let reachable graph v1 v2 =
   let rec search visited v =
      if v = v2 then
         true
      else if set_mem visited v then
         false
      else
         search_list (set_insert visited v) (dict_find graph v)
   and search_list visited = function
      v :: vl -> search visited v || search_list visited vl
    | [] -> false
   in
      search set_empty v1
```

Exercise 6.7 *Consider the function* `insert` *for unbalanced, ordered, binary trees in Section* **??**. *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`.

The principal change is to use pattern matching instead of the conditional.

```
let rec insert compare x = function
   Leaf -> Node (x, Leaf, Leaf)
 | Node (y, left, right) as node ->
     match compare x y with
        LessThan -> Node (y, insert compare x left, right)
      | Equal -> node
      | GreaterThan -> Node (y, left, insert compare x right)
```

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 label$(a) \leq$ 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.*

```
type heap = Node of int * heap list

let makeheap i = Node (i, [])

let meld (Node (i1, c1) as h1) (Node (i2, c2) as h2) =
   if i1 < i2 then
      Node (i1, h2 :: c1)
   else
      Node (i2, h1 :: c2)

let insert h i =
   meld h (makeheap i)

let findmin (Node (i, _)) =
   i

let deletemin = function
   Node (_, []) -> raise (Invalid_argument "deletemin")
```

```
| Node (_, [h]) -> h
| Node (_, h :: t) ->
    let rec meld_all h = function
       x :: t -> meld_all (meld h x) t
     | [] -> h
    in
       meld_all h t
```

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.*

```
let insert_list heap = function
   i :: elements -> insert_list (insert heap i) elements
 | [] -> heap

let heap_sort = function
   [] -> []
 | i :: elements ->
    let rec loop sorted heap =
       match heap with
          Node (i, []) -> i :: sorted
        | _ -> loop (findmin heap :: sorted) (deletemin heap)
    in
       loop [] (insert_list (makeheap i) elements)
```

7 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`

   The variables x and y refer to the same reference cell, so the result is 2.

2. `let x = ref 1 in let y = ref 1 in y := 2`

   The variables x and y refer to different reference cells, so the result is 1.

3. ```
   let x = ref 1 in
   let y = ref x in
   !y := 2;
   !x
   ```
   ```

Since y refers to x, assigning to !y is the same as assigning to x. The final value !x is 2.

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)
```

Both elements of the pair (y, y) refer to the same cell, so the assignment fst x := 2 effects both parts; the value !(snd x) is 2.

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
```

The variable x is incremented for each element of the list y, so the final value !x is 4.

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*.*

The important part is that the forcing is memoized. We can use a reference cell to save to computation.

```
type 'a deferred_value =
   Deferred of (unit -> 'a)
 | Forced of 'a

type 'a deferred = 'a deferred_value ref

let defer f = ref (Deferred f)
```

```
let force cell =
   match !cell with
      Deferred f ->
         let result = f () in
         cell := Forced result;
         result
    | Forced result ->
         result
```

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.*

The implementations are as follows.

```
let nil = Nil
let cons h t = Cons (h, t)
let lazy_cons h f = LazyCons (h, defer f)

let is_nil = function
   Nil -> true
 | Cons _ | LazyCons _ -> false

let head = function
   Nil -> raise (Invalid_argument "head")
 | Cons (h, _)
 | LazyCons (h, _) -> h

let tail = function
   Nil -> raise (Invalid_argument "tail")
 | Cons (_, t) -> t
 | LazyCons (_, f) -> force f

let rec (@@) l1 l2 =
   match l1, l2 with
```

```
    Nil, l | l, Nil -> l
  | _ -> lazy_cons (head l1) (fun () -> tail l1 @@ l2)
```

Exercise 7.4 *The FIFO queues described in Section* **??** *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.*

⋆ *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 [2].*

The queue could simply be represented as a list, but then one of the operations `add` or `take` would take time $O(m)$ where m is the length of the queue.

The two list (`front, back`) representation is a little better. To make the data structure persistent, the functions `add` and `take` produce new queues.

```
1  type 'a queue = ('a list * 'a list) ref
2
3  let empty = ref ([], [])
4
5  let add queue x =
6     let (front, back) = !queue in
7     ref (x :: front, back)
8
9  let rec take queue =
10    match !queue with
11       [], [] -> raise (Invalid_argument "queue")
12    | front, x :: back ->
13       x, ref (front, back)
14    | front, [] ->
15       queue := ([], List.rev front);
16       take queue
```

The side effect on line 15 does not affect persistence because the queue membership

is preserved. However, efficiency is still not optimal. Consider a sequence of m adds, followed by m takes.

```
let q_1 = add empty 1
let q_2 = add q_1 2
...
let q_m = add q_{m-1} m
let x_m, _ = take q_m
let x_{m-1}, _ = take q_{m-1}
...
let x_1, _ = take q_1
```

Each operation take q_i shifts i elements of the queue, so the total time is $O(m^2)$.

Okasaki gives an efficient implementation, summarized below in OCaml. A function maybe_shift is used to shift the queue whenever the front becomes longer than the back. The list lengths are needed, so the queue is a 4-tuple (*front*, $|$*front*$|$, *back*, $|$*back*$|$).

```
type 'a queue = 'a lazy_list * int * 'a lazy_list * int

let empty = (Nil, 0, Nil, 0)

let insert (front, flen, back, blen) x =
   maybe_shift (cons x front) (flen + 1) back blen

let remove (front, flen, back, blen) =
   head back, maybe_shift front flen (tail back) (blen - 1)

let maybe_shift front flen back blen =
   if flen <= blen then
      (front, flen, back, blen)
   else
      (Nil, 0, shift front back [], flen + blen)

let shift front back l =
   if is_nil back then
      cons (head front) l
   else
      lazy_cons (head back) (fun () ->
         shift (tail front) (tail back) (cons (head front) l))
```

Exercise 7.5 *One problem with the memoization function* `memo : ('a -> 'b) -> ('a -> 'b)` *in Section* **??** *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 ())
```

The main task here is to separate the parts of the memoization. We'll use a simple association list for the memo table. The expression `fib` n is computed in quadratic time $O(n^2)$. A more efficient implementation of the dictionary would reduct this to no more than $O(n \log n)$ time.

```
type ('a, 'b) memo = ('a * 'b) list

let create_memo () = []

let rec memo_find table x =
   match table with
      (x', y) :: _ when x' = x -> Some y
    | _ :: table -> memo_find table x
    | [] -> None

let memo_fib table i =
   match memo_find table i with
      Some j -> j
    | None ->
        match i with
           0 | 1 -> i
         | _ ->
            let j = memo_fib table (i - 1) + memo_fib table (i - 2) in
            memo_add table i j;
            j
```

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`*.*

We'll keep a flag `cyclic` to indicate whether the graph is cyclic. First, all the mark bits and DFS counters are cleared, then the function `search` is called to perform the DFS search. It isn't necessary to keep an explicit stack of edges—in effect, the OCaml runtime stack is serving as the edge stack.

```
let rec dfs graph =
   let cyclic = ref false in
   let dfs_counter = ref 0 in
```

```
(* The main DFS search *)
let rec search (Vertex (_, u_edges, u_mark, u_index)) =
   if not !u_mark then begin
      let c = !dfs_counter in
      u_index := Some c;
      dfs_counter := c + 1;
      List.iter (fun (Vertex (_, _, v_mark, v_index) as v) ->
         match !v_index with
            Some index ->
               if index < c && not !v_mark then
                  cyclic := true
          | None ->
               search v) !u_edges;
      u_mark := true
   end
in

(* Reset the graph state *)
List.iter (fun (Vertex (_, _, mark, index)) ->
   mark := false;
   index := None) graph;

(* DFS over all the vertices in the graph *)
List.iter search graph
```

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?*

The function `graph_map` is definable, but it is difficult to implement it efficiently. The central issue is that, in order to preserve the structure of the graph, the function

graph_map must be memoized using physical equality. The following code gives a simple, but inefficient, implementation.

```
let graph_map f graph =
   let memo = ref [] in
   let rec map u = function
      (u', v) :: table when u' == u -> v
    | _ :: table -> map u table
    | [] ->
       let (label, out_edges) = u in
       let new_out_edges = ref [] in
       let new_u = (f label, new_out_edges) in
       memo := (u, new_u) :: !memo;
       new_out_edges := map_list !out_edges
   and map_list edges =
      List.map (fun v -> map v !memo) edges
   in
   map_list graph
```

The function map searches through the memo table in linear order, looking for an entry that matches with physical equality (==). If so, the previous result is returned. Otherwise, a new vertex is created and added to the memo table. Note that the new vertex new_u is added before following the out-edges recursively, preventing infinite looping on cyclic graphs.

Given a graph with n vertices and m edges, the function map examines n vertices worst case, so the total time complexity is of graph_map is $O(nm)$.

An alternative representation that would support an efficient map is to give vertices unique names, and refer to them by name rather than referring to them directly. The penalty is that edge traversal takes time $O(\log n)$ rather than constant time because of the dictionary lookup.

```
(* A name for a vertex *)
type name = int

(* A vertex is (label, out-edges) *)
type 'a vertex = 'a * name list ref

(* A graph is (vertices, vertex dictionary) *)
type 'a graph = name list * (name, 'a vertex) dictionary

let graph_map f (vertices, dict) =
   vertices, dictionary_map f dict
```

8 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
```

The operations are implemented with operations on records.

```
let ref x = { contents = x }
let (!) cell = cell.contents
let (:=) cell x = cell.contents <- x
```

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. `[|[]|]`

   The type is `[|[]|] : '_a list array`.

2. `{ fst = []; snd = [] }`

   Mutable fields are not values, so the field `fst` is not polymorphic because of the value restriction. The type is `{ fst = []; snd = [] } : ('_a list, 'b list) mpair`.

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

   During a functional update, it is legal for the types of polymorphic fields to change. The expression `{ fst = (); snd = 2 }` has type `(unit, int) mpair`, but the final value is `{ fst = 1; snd = 2 }` of type `(int, int) mpair`.

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.*

We'll use association lists. The function `insert` adds to the list, and `find` searches the list. The function `new_dictionary` is used to form a dictionary from an association list.

```
let empty =
   let rec find entries key =
      match entries with
         (key', value) :: _ when key' = key -> value
       | _ :: entries -> find entries key
       | [] -> raise Not_found
   in
   let rec new_dictionary entries =
      { insert = insert entries;
        find = find entries
      }
   and insert entries key value =
      new_dictionary ((key, value) :: entries)
   in
   new_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?*

The problem is that the method `move` refers to the definitions of `get` and `set` when the rectangle is first created. This is called *early binding*, where true object systems use *late* binding. Early binding means that the method `move` is not updated when `set` is changed.

1. The expression `rect2.move (dx, dy)` moves `rect2` by `(dx, dy)`. To prevent this from happening, the method `move` should be updated as well.

   ```
   let rect2 = { rect1 with set = (fun _ -> ()); move = (fun _ -> ()) }
   ```

2. The rectangles `rect2` and `rect1` refer to the same position `pos`, so setting the position of `rect1` also moves `rect2`.

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

```
let string_reverse s =
   let len = String.length s in
   for i = 0 to len / 2 - 1 do
      let c = s.[i] in
      s.[i] <- s.[len - i - 1];
      s.[len - i - 1] <- c
   done
```

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
```

There can be a problem if the `src` and `dst` arrays are the same, and the ranges to be copied overlap, and `dst_off > src_off`.

For example, the following expression duplicates the first element of the array instead of copying a subrange.

```
let data = [|1; 2; 3; 4; 5; 6; 7; 8; 9|];;
val data : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9|]
# blit data 0 data 1 5;;
- : unit = ()
# data;;
- : int array = [|1; 1; 1; 1; 1; 1; 7; 8; 9|]
```

An easy solution is to copy in reverse direction when `dst_off > src_off`.

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

Each of the constructs in the pseudo-code can be translated directly to OCaml. However, it is slightly more efficient to avoid the use of reference cells, and translate the while-loop as a recursive function.

```
let insert a i =
   let x = a.(i) in
   let rec loop j =
      if j >= 0 && a.(j) > x then begin
         a.(j) <- a.(j - 1);
         loop (j - 1)
      end
      else
         j
   in
   let j = loop (i - 1) in
   a.(j) <- x

and insertion_sort a =
   for i = 1 to Array.length a - 1 do
      insert a i
   done
```

9 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`

1. `exception A` is legal.
2. `exception b` is not legal because the name `b` must be capitalized.
3. `exception C of string` is legal.
4. `exception D of exn` is legal, it adds a recursive definition.
5. `exception E of exn let x = E (E (E Not_found))` is also legal, the value `x` has type exn.
6. `let f () = exception F raise F` is not legal, exceptions can only be declared at the top-level, not within function bodies.

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
```

1. 1

2. When g is called, f is called with the argument 100, raising the exception A 1, passing control back to g which returns 1.

3. The expression g i returns 1 for any value i ≥ 0. The function f always raises the exception A 0, which passes control to the *innermost* exception handler for g, which then returns 1. As the call stack unwinds, the return value 1 is passed unchanged.

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
```

The second form is preferable. The first version is not tail-recursive, and the depth of the exception stack is linear in the number of entries in the `names` list. The second version does not have these problems; it is properly tail-recursive.

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
```

In the first version, the function `f` is called within the exception handler, which

means that if `f` raises the `Not_found` exception, then `callf` will return 0, the same as if the `List.assoc` function raises `Not_found`.

The second version separates the calls. If the function `f` raises `Not_found`, the exception will be propagated through the calls to `callf`. The second form is preferable in situations where exceptions raised from `f` indicate an error, not normal operation.

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.*

The main problem with writing the `input_lines` function is in catching the `End_of_file` exception. The following program is inefficient, because the exception stack is linear in the length of the input file. For large files, the stack will likely overflow.

```
let rec input_lines stdin =
    try input_line stdin :: input_lines stdin with
       End_of_file ->
          []
```

The way to code this efficiently is to wrap the `input_line` function to catch the exception.

```
let maybe_input_line stdin =
   try Some (input_line stdin) with
      End_of_file ->
         None

let input_lines stdin =
   let rec input lines =
      match maybe_input_line stdin with
         Some line -> input (line :: lines)
       | None -> List.rev lines
   in
      input []
```

10 Exercises

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

The `output_string` function can be used to print the line.

```
# output_string stdout "Hello world!\n";;
Hello world
```

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* **??***), 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?*

1. The function `loop` is not tail-recursive because the recursive call is enclosed in the `try` block.

2. One solution that keeps the general style is to introduce a function `maybe_input_line` that produces a string option instead of raising an exception.

```
let maybe_input_line chan =
   try Some (input_line chan) with
      End_of_file -> None

let read_lines chan =
   let rec loop lines =
      match maybe_input_line chan with
         Some line -> loop (line :: lines)
       | None -> List.rev lines
   in
   loop []
```

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.*

If the function raises an exception, the exception can be caught with a wildcard exception handler, the channel closed, and the exception re-raised.

```
let with_in_file filename f =
   let in_chan = open_in filename in
   try
      let x = f in_chan in
      close_in in_chan;
      x
   with exn ->
      close_in in_chan;
      raise exn
```

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?*

A sensible approach is to read the characters, swap them, and write the results back. If an error occurs before the values are written, the operation can simply be aborted.

However, once the first file is modified, the second file must be modified as well. In the following code, if an error occurs while writing the second file (line 7), the error is caught and an attempt is made to write character `c1` back to the first file. An error on line 10 is fatal. In general is it not possible to make the `exchange` operation atomic.

```
let exchange file1 file2 =
   let c1 = with_in_file file1 input_char in
   let c2 = with_in_file file2 input_char in
   with_out_file file1 (fun chan -> output_char chan c2);
   (* Errors after this line must be handled *)
   try
      with_out_file file2 (fun chan -> output_char chan c1)
   with exn ->
      (* Try to write back c1 to file1 *)
      with_out_file file1 (fun chan -> output_char chan c1);
      raise exn
```

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)
```

Here is one version.

```
let rec print_exp = function
   Int i -> print_int i
 | String s -> printf "\"%s\"" s
 | List el -> print_char '('; print_exp_list el; print_char ')'

and print_exp_list = function
   [] -> ()
 | [e] -> print_exp e
 | e :: el ->
     print_exp e;
     print_char ' ';
     print_exp_list el
```

The `printf`-style functions provide a slightly more concise implementation.

```
let rec print_exp out_chan = function
   Int i -> output_int out_chan i
 | String s -> fprintf out "\"%s\"" s
 | List el -> fprintf out "(%a)" print_exp_list el

and print_exp_list out_chan = function
   [] -> ()
 | [e] -> print_exp out_chan e
 | e :: el -> fprintf out_chan "%a %a" print_exp e print_exp_list el
```

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.

The lines that begin with the digit `1` can be printed immediately, so we need only buffer the lines beginning with the digit `2`.

```
let deinterleave () =
   let inc = open_in "data.txt" in
   let buf = Buffer.create 256 in
   try
      while true do
         let line = input_line inc in
         let len = String.length line - 1 in
         if line.[0] = '1' then begin
            output stdout line 1 len;
            output_char stdout '\n'
         end
         else begin
            Buffer.add_substring buf line 1 len;
            Buffer.add_char buf '\n'
         end
   with End_of_file ->
      output_string stdout (Buffer.contents buf)
```

This code is a bit sloppy. It doesn't deal gracefully with blank lines, and it assumes that any line not beginning with the digit `1` must begin with a `2`.

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`.

```
let printline (x, y, z) =
   printf "%-5s 0x%08x %3s\n" x y z
```

Here are some examples.

```
# printline ("A", 10, "B");;
A     0x0000000a   B
# printline ("abcdefg", 255, "hijk");;
abcdefg 0x000000ff hijk
```

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
```

This seems like it would be straightforward with `printf`, we would just calculate the width of the first column, and produce the right format string. For example, if the

variable `words` contains the word list, and the width of the first column is computed to be 20 characters, we would use the following printer.

```
List.iter (fun (w1, w2) ->
   printf "%-20s %s\n" w1 w2) words
```

However, this doesn't work in general because the format string must be computed, and `printf` requires a literal string.

```
# let fmt = sprintf "%%-%ds %%s\n" 20;;
val fmt : string = "%-20s %s\n"
# List.iter (fun (w1, w2) ->
     printf fmt w1 w2) words;;
Characters 37-40:
     printf fmt w1 w2) words;;
%           ^^^
This expression has type string but is here used with type
  ('a -> 'b -> 'c, out_channel, unit) format
```

Instead, we can define a "padding" function, and use it to justify the output, using the `%a` format specifier.

```
let pad ochan i =
   for j = 1 to i do
      output_char ochan ' '
   done

let print_cols l =
   let width =
      List.fold_left (fun width (s, _) ->
          max width (String.length s)) 0 l
   in
      List.iter (fun (s1, s2) ->
         printf "%s%a %s\n" s1
            print_pad (width - String.length s1) s2) l
```

As it turns out, there *is* a way to use `printf` directly. If the width specifier is the character `*`, then `printf` expects the width specifier to be passed as an argument.

```
let print_cols l =
   let width =
      List.fold_left (fun width (s, _) ->
          max width (String.length s)) 0 l
   in
      List.iter (fun (s1, s2) ->
         printf "%-*s %s\n" width s1 s2) l
```

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`

1. The program returns the string `"A"`.
2. The program returns the empty string.
3. The program returns the string `"B"`.
4. The program raises the `Scan_failure` exception, removing the `C` from the input channel.
5. The program returns the string `"C"`. The first `scanf` fails, but removes the `A` from the input stream. The second `scanf` matches the `B`, and returns the string `"C"`.

11 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
```

1. It is always legal for a `.mli` file to be empty. However, this hides all definitions in the `.ml` file, so it has limited usefulness.

2. The specification `val f : 'a -> 'a` is legal.

3. The specification `val f : ('a -> 'b) -> ('a -> 'b)` is also legal (it is just a refinement of the type `'a -> 'a`).

4. The specification `val f : t -> t` is not legal because there is no definition for the type `t`.

5. The specification `type t val f : t -> t` is legal.

6. The specification `type s = int val f : s -> s` is not legal because the type `s` must also be defined in the `.ml` file.

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 []
```

1. There are differences, but it isn't clear that one version is better than the other. In version 1, the variable `l` is bound (and visible) in the `rev_loop` function. Some typographical errors (for example, if we had written `match l with ...` instead of `match l2 with ...`) will not be caught at compile time.

   Version 2 has a similar problem; the `rev_loop` function is visible in the rest of the file, so it might be used accidentally. Note however, that the signature `rev.mli` hides the definition from the rest of the program.

2. The partial application `let rev = rev_loop []` is not allowed, as it has type `'_a -> '_a`, not `'a -> 'a`. See Section **??**, which discusses the value restriction.

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
```

1. One solution is to add explicit type constraints to the source program. For example, if we revise the definition for add as follows, the correct type for it will be inferred.

```
let add x (s : 'a set) : 'a set = x :: s
```

2. The inferred type for this program is the following.

```
val cell : '_a list ref
val push : '_a -> unit
val pop : unit -> '_a
```

Syntactically, the problem with this signature is the type variable `'_a`, which is not allowed in an interface file. The real problem is that the type of the reference cell is unspecified, but it can only be used with one type. To fix the signature, we

must choose a specific type for the stack. For example, the following is a valid signature that specifies that the stack is a stack of integers.

```
val push : int -> unit
val pop : unit -> int
```

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.*

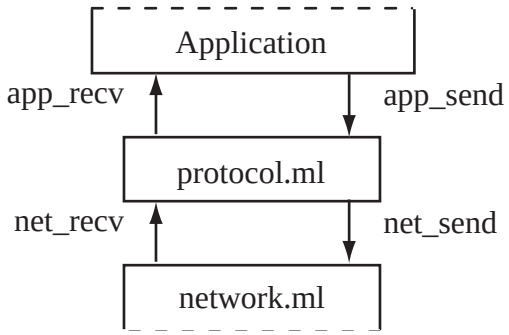
One solution we discussed is to place the type definition in a separate file, like `types.ml`*, with no interface file* `types.mli`*. It is also legal to place the type definition in a file* `types.mli` *with no implementation* `types.ml`*.*

Is it ever preferable to use the second form (where `types.mli` *exists, but* `types.ml` *doesn't)?*

The differences between the two include the following.

- A `.mli` file contains only declarations and type definitions. If the file contains a expression definition (like `let zero = 0`), then it must be placed in a `.ml` file.
- Components with only a `.mli` file need not be specified during linking. This is a slight benefit, but it can slightly simplify program construction in some cases.

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.*

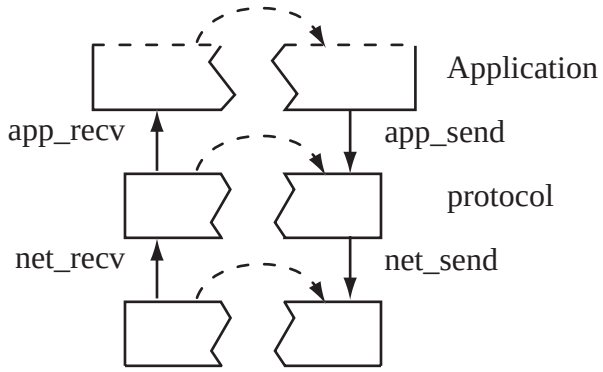


With this design, the `Network` *component calls* `Protocol.net_recv` *when a message arrives, and the* `Protocol` *component calls* `Network.net_send` *to send a message. However, this is not possible if the implementations are in separate files* `protocol.ml` *and* `network.ml` *because that would introduce a cyclic dependency.*

Describe a method to circumvent this problem, without placing the code for the two components into a single file.

There are a few potential solutions, including the use of recursive modules (discussed in Section **??**) or objects (Chapter **??**). In addition here are a few alternate ways.

1. If the design can be partitioned into two separate message streams without any cycles, the design can be programmed directly.



   However, this approach may not be possible, and the changes required may be unnatural.

2. A simpler solution is to introduce a third file containing references that can be set to refer each of the functions. This solution, while lacking elegance, is straightforward.

refs.mli

```
val net_recv_ref : (message -> unit) ref
val net_send_ref : (message -> unit) ref
val app_recv_ref : (message -> unit) ref
val app_send_ref : (message -> unit) ref
```

The cells are initialized with dummy values.

```
let net_recv_ref = ref (fun _ -> raise (Invalid_argument "not initialized"))
...
```

At startup time, the layers set the references to refer to the appropriate functions.

protocol.ml

```
let net_recv msg = ...
...
Refs.net_recv_ref := net_recv
```

3. Yet another option is to define a third file, say `link.ml` that builds the recursive definition in a single file. Each of the layer functions would take as arguments the functions that it expects to call.

```
let rec net_recv msg =
   Protocol.net_recv net_send app_recv msg
and app_send msg =
   Protocol.app_send net_send app_recv msg
and net_send msg =
   Network.net_send net_recv msg
and app_recv msg =
   Application.app_recv app_send msg
```

This solution is more complicated and may be slightly less efficient than the solution using reference cells.

12 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
```

1. Legal; the value associated with a variable is specified by the last definition in the module.

2. Legal; the duplicate type definition is legal. What would happen if the type definitions differ?

3. Not legal; the module name must begin with an uppercase letter.

4. Legal; within the structure body the function `g` has type `'a -> 'a`. The type constraint is applied *after* the structure is formed.

5. Legal; the value is `1`.

6. Not legal; the function `h` must be defined before `g`.

7. Legal; the forward reference to `h x` is allowed because the module is recursive.

8. Legal; however a application of the function `f` will fail at runtime because the value cannot be resolved.

9. Legal; `f` has type `f : 'a -> 'a t`.

10. Not legal; OCaml produces the error message "In this type, the locally bound module name M escapes its scope." This is because the the function `f` produces a value of type `M.t`, but `M` is defined only in the body of `f`.

11. Legal; the modules `A` and `B` share a common (abstract) type, so it is legal to pass the result of `A.f t B.g`.

12. Legal; the `include` directive is like textual inclusion, subject to signature constraints. After expansion, the module `C` has the following form.

```
module C : sig
   type t
   val x : t
end = struct
   type t = int (* sig type t = int *)
   let x = 1    (* sig val x : int *)
end
```

This is a legal module definition.

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?*

Recursive modules can be used for the top-down programming style. The code is wrapped in a module, and all forward references must be declared in the module signature. To illustrate, suppose we have a function `main` that calls functions `f` and `g`, using an intermediate type `t`.

```
module rec Body : sig
   type t = V of int
```

```
    val main : int -> int
    val f : int -> t
    val g : t -> int
end = struct
    open Body

    let main i = g (f i)
    let f i = V i
    let g (V i) = i
    type t = V of int
end
```

This approach has the usual disadvantage that types must be defined twice, once in the signature and once in the structure. However, the type definition can be defined before the module. This is common style even in 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* **??***, 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?*

Instead of using a sharing constraint, we can add the type definition directly in the signature.

```
module type Set2Sig = sig
    type 'a set = 'a Set.set
    val empty : 'a set
    ...
end
```

In fact, we don't even need to define a separate type.

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

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).

The type `'a -> 'a` is more general than the type `int -> int`. That is, any value with the former type can also be given the latter type; the inverse does not hold. This means that the value must have `'a -> 'a`. The only sensible value with this type is the

identity function.

```
module A : ASig = struct
   let f x = x
end
```

struct4

13 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
   ```

1. Legal; the functor `F` is the identity functor.

2. Legal; the module expression `Apply (F) (A)` is equivalent to `F (A)`.

3. Not legal; the module expression `F(struct let i = 1 end)` is not an expression.

4. Not legal; the module parameter `X` must have a signature (as in $F(X : XSig)$).

5. Legal; the module `X` has the specified signature `sig type t = A | B end`.

6. Legal; the structure `struct type t = A | B end` has signature `sig type t = A | B end`.

7. Not legal; the types `X.t` and `type t = X.t` are different types.
   ```

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`

The first three expressions are legal because, in each of the `D` modules, the type `t` is `int`. For the remaining four expressions, the functor `G` produces a module with signature `T`, where the type `t` is abstract. In part 6, the expressions have types `E1.x : G(A).t` and `E3.x : G(C).x`. The types `G(A).t` and $G(C).t$ are different, even though modules `A` and `C` are equal.

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)))
```

The body of a functor is not evaluated until the functor is applied. Thus, the program prints one line of output for each functor application (so there are three lines of output).

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* **??** *(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?*

1. The definition is a direct translation of the functor `MakeSet`.

```
type 'elt t = 'elt list
let make_set equal =
   { empty = [];
     add = (::);
     mem = (fun x s -> List.mem (equal x) s);
     find = (fun x s -> List.find (equal x) s)
   }
```

   To keep the type `'elt t` abstract, we must enclose the definition in a module.

```
module Set : sig
   type 'elt t
   type 'elt set = ···
   val make_set : ('elt -> 'elt -> bool) -> 'elt set
end = struct
   type 'elt t = 'elt list
   type 'elt set = ···
   let make_set equal = ···
end
```

2. No. The type `'elt t` is fixed for all sets (of type `'elt set`).

3. The implementation of the function `make_set` is unchanged. It has the following type.

```
type 'elt set_repr = 'elt list
···
val make_set : ('elt -> 'elt -> bool) -> ('elt, 'elt set_repr) set
```

   The type `set_repr` is the representation for sets; the type definition can be hidden the usual way.

4. The main advantage of the record representation is that it is *first class*, meaning that values of type `'elt set` can be passed as arguments, stored in data structures, etc. There are several disadvantages. Among them, type expressions are larger. In the worst case, the type definition requires a parameter for each type that would be abstract in the module. In addition, there is a slight performance

penalty for calling a function in the `set` record; the functor does not have this penalty penalty because references are resolved at compile time.

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.*

Each function can be defined in its own file using a functor, where the functor argument defines both functions.

```
module type RSig = sig
   val f : int -> int
   val g : int -> int
end

module FFun (R : RSig) = struct
   open R
   let rec f i = ···
end
```

The glue code can be placed in a third file, using recursive modules to "tie the knot."

```
module rec F : sig val f : int -> int end = FFun (R)
and F : sig val g : int -> int end = GFun (R)
and R : RSig = struct
   let f = F.f
   let g = G.g
end
```

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
```

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

```
   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.*

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.)*

1. For the `Uniq` filter, we can use a reference cell to keep track of lines as they are read. We "cheat" in this solution—since we know the input never contains a newline, we initialize the refcell to an impossible value. A better solution would be for the cell to be initialized to `None` (and thus be of type `string option`).

```
module Uniq (Stdout : Pipeline with type t = string)
 : Pipeline with type t = string =
struct
   type t = string
   let last_line = ref "\n"
   let f s =
      if s <> !last_line then
         Stdout.f s;
      last_line := s
end
```

2. The main problem in defining the `Grep` filter is that it takes a regular expression as an argument. It is possible to pass the regular expression in its own module,

but this will mean that the Grep module works for only one regular expression. (The next part illustrates another solution to this problem.)

```
module Grep
 (R : sig val regex : string end)
 (P : Pipeline with type t = string) =
struct
   type t = string
   let regexp = Str.regexp R.regex
   let f s =
      if Str.string_match regexp s 0 then
         P.f s
end
```

3. One easy way to define the grep function is to use a `let module` construction to define the pipeline within the function body.

```
let grep regex filename =
   let module P = Cat (Grep (struct let regex = regex end) (Print)) in
   P.f filename
```

4. The StringOfChar module simply iterates through each character of the input string.

```
module StringOfChar (P : Pipeline with type t = char)
 : Pipeline with type t = string =
struct
   type t = string
   let f s = String.iter P.f s
end
```

5. The Compose functor takes three arguments, the first filter F1, the second filter F2, and the rest of the pipeline P3, where `Compose (F1) (F2) (P3) = F1 (F2 (P3))`. Thus, a partial application `Compose (F1) (F2)` will yield a filter.

   The main issue with constructing the filter is with the constraints about compatibility of the filters' types. These sharing constraints are not obvious. Suppose filter F1 takes values of type t_1, filter F2 takes values of type t_2, and the rest of the pipeline P3 takes values of type t_3. For illustration, we would have the following constraints.

```
module Compose
 (F1 : (P : Pipeline with type t = t2) -> Pipeline with type t = t1)
 (F2 : (P : Pipeline with type t = t3) -> Pipeline with type t = t2)
 (P3 : Pipeline with type t = t3) = F1 (F2 (P3))
```

The proper types are then as follows t_3 = `P3.t`, t_2 = `F2(P3).t`, and t_1 = `F1(F2(P3)).t`. However, using these definitions directly is not possible because it would create forward references in the type definition. For example, the signature for `F1` would refer forward to the modules `F2` and `P3`. We could reorder the arguments to eliminate the forward references, but then the partial application would not work as we wish.

Arguably, the best solution is to redefine the signature for filters so that they specify the types for both input and output.

```
module type Filter = sig
   type t_in
   type t_out
   module F : functor (P : Pipeline with type t = t_out) ->
      Pipeline with type t = t_in
end
```

The filters themselves are not much different. For example, here is the filter `StringOfChar`.

```
module StringOfChar
 : Filter with type t_in = string and type t_out = char =
struct
   type t_in = string
   type t_out = char
   module F (X : Pipeline with type t = char) = struct
      type t = string
      let f s = String.iter X.f s
   end
end
```

The sharing constraints for the `Compose` functor are now easy to express.

```
module Compose
 (F1 : Filter)
 (F2 : Filter with type t_in = F1.t_out)
 : Filter with type t_in = F1.t_in and type t_out = F2.t_out =
struct
   type t_in = F1.t_in
   type t_out = F2.t_out
   module F (P3 : Pipeline with type t = t_out) = struct
```

```
      module Pipe = F1.F (F2.F (P3))
      type t = t_in
      let f = Pipe.f
    end
end
```

The signature can be represented as follows.

```
Compose : functor (F1 : Filter) ->
  functor (F2 : Filter with type t_in = F1.t_out) ->
  Filter with type t_in = F1.t_in and type t_out = F2.t_out
```

14 Exercises

Exercise 14.1 *In Section* **??** *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?*

Let's implement `new_scale` as a normal function.

```
let new_scale sx sy =
   object
      val matrix = (sx, 0., 0., sy, 0., 0.)
      method transform (x, y) = ···
      method multiply matrix2 = ···
   end;;
```

If the other functions were implemented the same way, the code for the methods `transform` and `multiply` would be duplicated. We can avoid this by creating a generic constructor function that takes the entire matrix as an argument.

```
let new_transform matrix =
   object
      val matrix = matrix
      method transform (x, y) = ···
      method multiply matrix2 = ···
   end

let new_scale sx sy = new_matrix (sx, 0., 0., sy, 0., 0.)
let new_translate dx dy = new_matrix (1., 0., dx, 0., 1., dy)
let new_rotate theta =
   let s, c = sin theta, cos theta in
   new_transform (c, -.s, 0., s, c, 0.)
```

Exercise 14.2 *In Section* **??** *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?*

The purpose of the code is to allow the functional update `{< vertices = Array.map matrix#transform vertices >}`; for this to work, `vertices` must be a field of the object. If the field definition were omitted, the functional update would fail.

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
```

The object is initially created with a reference `self1` to itself, using the `self` parameter, and the value of `self` is checked (with pointer equality) before the withdraw operation is allowed. The value for `self1` has to be set in an initializer (when the value of `self` is known). For simplicity, we use the empty object `object end` for the initial value of `self1`.

```
let my_account =
object (self : 'self)
    val mutable balance = 100
    val mutable self1 : < > = object end
    initializer self1 <- (self :> < >)
    method withdraw =
        if (self :> < >) != self1 then
           raise (Failure "object has been copied");
        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`

1. t_1 <: t_2, because `dog <: animal` and `cat <: animal`.
2. $t_1 \not<: t_2$, because `'a refcell` is invariant in `'a`.
3. $t_1 \not<: t_2$, the class `'a cl` is invariant in `'a`.
4. $t_1 \not<: t_2$, because `'a refcell` is invariant in `'a`.
5. t_1 <: t_2, because `'a c2` is contravariant in `'a`.
6. t_1 <: t_2, because the type `(('a -> animal) -> 'b) -> animal` is contravariant in `'a` and `'b`.

Exercise 14.5 *Let's reimplement the narrowing example from page* **??** *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?*

1. The complete solution is mainly unchanged from the code that uses exceptions.

```
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

let fido : 'a tag dog =
object (self)
   method actual = `Dog self
   method eat = ()
   method bark = ()
end;;

let daphne : 'a tag cat =
object (self)
   method actual = `Cat self
   method eat = ()
   method meow = ()
end;;

let animals = [(fido :> 'a tag animal); (daphne :> 'a tag animal)];;

let chorus (animals : 'a tag animal list) =
   List.iter (fun animal ->
      match animal#actual with
         `Dog dog -> dog#bark
       | `Cat cat -> cat#meow
       | _ -> ()) animals
```

2. The type variable `'a` stands for the real type of tags. The tag type contains at least the tags `` `Dog `` and `` `Cat ``, but it is an open type, so the actual type `'a` may contain additional constructors.

3. Since lizards don't vocalize, we really don't need to change any of the existing code. We simple add the new object with a new tag.

```
let fred : 'a tag lizard =
object (self)
   method actual = `Lizard self
   method eat = ()
end;;
```

Exercise 14.6 *The narrowing technique on page* **??** *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`*.*

To keep it simple, we'll use exceptions both as tags and actual values.

```
type actual = exn list -> exn
type animal = < actual : actual; eat : unit >
type dog = < actual : actual; eat : unit; bark : unit >
type cat = < actual : actual; eat : unit; meow : unit >
type hound = < actual : actual; eat : unit; bark : unit; howl : unit >

exception DogTag
```

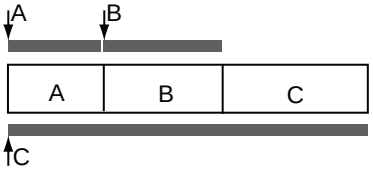
```
exception CatTag
exception HoundTag

exception Dog of dog
exception Cat of cat
exception Hound of hound

let fido : hound =
object (self)
   method actual tags =
      match tags with
         HoundTag :: _ -> Hound self
       | DogTag :: _ -> Dog (self :> dog)
       | _ :: tags -> self#actual tags
       | [] -> Not_found
   method eat = ()
   method bark = ()
   method howl = ()
end;;

let chorus (animals : animal list) =
   List.iter (fun animal ->
      match animal#actual [DogTag; CatTag] with
         Dog dog -> dog#bark
       | Cat cat -> cat#meow
       | _ -> ()) animals
```

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

```
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
```

Model

```
! 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?*

   1. ```
      let animal_model s =
          let rec self = function
              `L_eat -> printf "%s eats\n" s
          in
          self
      ```

   2. A `model_dog` has all the methods of a `model_animal` so it can be used as an animal unchanged. The coercion returns the dog without change. To be more precise, consider the types.

      ```
      type model_dog = l:[`L_eat | `L_bark | `L_chase] -> ···
      type model_animal = l:[`L_eat] -> ···
      ```

      The relation `model_dog <: model_animal` holds because `[`L_eat] <: [`L_eat | `L_bark | `L_chase]`.

   3. A `model_dog` can be used as a model-`< chase : string -> unit>` without change.

   4. A functional update becomes a record update. The new method `new_dog` is modeled as follows.

      ```
      let model_dog s =
          let rec self fields = function
              ···
            | `L_new_dog s ->
                self { fields with name = s }
          in
          self { name = s }
      ```

   5. The pattern match is really just a table lookup, so it can be implemented in $O(\log n)$ time, where n is the number of labels. However, the number of labels in the program is fixed, so the pattern match can be implemented in constant time.

   6. The advantage of hoisting the `vtable` is that it can be shared by multiple dog objects. The disadvantage is that it may be slightly more expensive because of the extra function call.
      ```

15 Exercises

Exercise 15.1 *What are the class types for the following classes?*

1. ```
   class c1 =
   object
      val x = 1
      method get = x
   end
   class c1 :
   object
      val x : int
      method get : int
   end
   ```

2. ```
   class c2 =
   object
      method copy = {< >}
   end
   class c2 :
   object ('self)
      method copy : 'self
   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
   class c3 : int ->
   object
      method f : int -> < h : int >
      method g : int
   end
   ```

4. ```
   class c4 =
   object (self : < x : int; .. > as 'self)
      method private x = 1
   end
   ```

   The type constraint removes the private status of the method x.

   ```
   class c4 :
   object
      method x : int
   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;;
```

The order of the let-expressions and initializers follows the textual order. Class a is nested within b, but before b's initializer. The sequence is the following.

```
B let
A let
A init
B init
```

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*?*

The class rectangle adds a new dimension h.

```
class rectangle x y w h =
object
   inherit square
   method area = w * h
   method draw = Graphics.fill_rect x y w h
end
```

It is appropriate to say that the *representation* of a rectangle includes the representation of a square. The is-a relationships in the program are defined by the programmer. They don't have to correspond to real-life relationships.

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
```

The class `nil` returns `true` for the method `is_nil`, and it raises an exception on all other operations.

```
class nil : int_list =
object (_ : #int_list as 'self)
   method is_nil = true
   method hd = raise (Invalid_argument "hd")
   method tl = raise (Invalid_argument "tl")
   method set_hd _ = raise (Invalid_argument "set_hd")
   method set_tl _ = raise (Invalid_argument "set_tl")
end
```

The class `cons` implements the mutable cons-cell. The constraint `_ : #int_list as 'self` is used to simplify the types.

```
class cons hd tl =
object (_ : #int_list as 'self)
   val mutable hd = hd
   val mutable tl = tl
   method is_nil = false
   method hd = hd
   method tl = tl
   method set_hd x = hd <- x
   method set_tl l = tl <- l
end
```

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
```

No, it is not possible to generalize the type, at least not easily. The problem is that the class `cons` takes the `tl` as an argument. If we try to implement it, we get the error "Self type cannot escape its class," because the argument `tl` has the same type as the class being defined.

```
class cons hd tl =
object (_ : #gen_int_list as 'self)
...
end
      This expression has type 'a but is here used with type
        < hd : int; is_nil : bool; set_hd : int -> unit; set_tl : 'b -> unit;
          tl : 'b; .. >
        as 'b
      Self type cannot escape its class
```

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`*.* Here are the complete class definitions.

```
class nil =
object (self : #int_list as 'self)
   method is_nil = true
   method hd = raise (Invalid_argument "hd")
   method tl = raise (Invalid_argument "tl")
   method set_hd (_ : int) = raise (Invalid_argument "set_hd")
   method set_tl (_ : int_list) = raise (Invalid_argument "set_tl")
   method iter f = ()
   method map f = (self :> int_list)
   method fold f x = x
end

class cons hd tl =
object (self : #int_list as 'self)
   val mutable hd = hd
```

```
    val mutable tl = tl
    method is_nil = false
    method hd = hd
    method tl = tl
    method set_hd x = hd <- x
    method set_tl l = tl <- l
    method iter f = f hd; tl#iter f
    method map f = ({< hd = f hd; tl = tl#map f >} :> int_list)
    method fold f x = tl#fold f (f x hd)
  end
```

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`*.*

   The queue can be defined by keeping track of the last element in the list of items, so that the method `add` adds the new element at the end of the list, instead of at the beginning.

```
class int_queue =
let nil = new nil in
object
   inherit int_stack

   val mutable last = nil

   method add i =
      let new_last = new cons i nil in
      if last#is_nil then
         items <- new_last
      else
         last#set_tl new_last;
      last <- new_last
end
```

2. *Is it appropriate to say that a queue is-a stack?* The two data structures have the same type but they are semantically different. A queue refines the stack

implementation, but it does not behave like a stack. We would not normally 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
```

The class definitions are pretty simple; they each implement an evaluator.

```
class int_exp i =
object (_ : #exp as 'self)
   method eval env = i
end

class binary_exp op (e1 : #exp) (e2 : #exp) =
object
   method eval : 'a. (#env as 'a) -> int =
      (fun env -> op (e1#eval env) (e2#eval env))
end

class add_exp e1 e2 = binary_exp (+) e1 e2
class sub_exp e1 e2 = binary_exp (-) e1 e2
```

```
class var_exp v =
object
   method eval : 'a. (#env as 'a) -> int = (fun env -> env#find v)
end
```

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`*.*

```
class let_exp v (e1 : #exp) (e2 : #exp) =
object
   method eval : 'a. (#env as 'a) -> int =
      (fun env -> e2#eval (env#add v (e1#eval env)))
end
```

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?*

Unfortunately, all the classes need to be extended.

```
class type exp2 =
object
   inherit exp
   method closed : string list -> bool
end

class int_exp2 i =
object (_ : #exp2 as 'self)
   inherit int_exp i
   method closed _ = true
end

class binary_exp2 op e1 e2 =
object (_ : #exp2 as 'self)
   inherit binary_exp op e1 e2
```

```
   method closed defined_vars =
      e1#closed defined_vars && e2#closed defined_vars
end

class add_exp2 e1 e2 = binary_exp2 (+) e1 e2
class sub_exp2 e1 e2 = binary_exp2 (-) e1 e2

class let_exp2 v e1 e2 =
object (_ : #exp2 as 'self)
   inherit let_exp v e1 e2
   method closed defined_vars =
      e1#closed defined_vars && e2#closed (v :: defined_vars)
end
```

Exercise 15.7 *Object-oriented programming originated in the Simula, a language designed by Dahl and Nygaard [1] 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.*

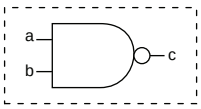
```
class virtual gate =
object (self : 'self)
   val mutable output_wire = dummy_wire
   method connect_output wire = output_wire <- wire
   method private set_output =  output_wire#set self#compute_value
```

```
   method private virtual compute_value : unit -> bool
end
```

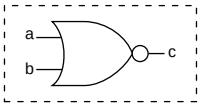
A `two_input_gate` *is a gate that has two inputs.*

```
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`.

   For the methods `connect_input_a` we need to construct a terminal object that performs the appropriate action.

   ```
   method connect_input_a (wire : wire) =
       wire#add_terminal (object method set x = self#set_input_a x end)
   ```

2. *Define a class* `three_input_gate` *(for gates with three inputs) by inheriting from* `two_input_gate`.

   ```
   class three_input_gate =
   object
      inherit two_input_gate
      val mutable c = false
      method private set_input_c x = c <- x; self#set_output
      method connect_input_c (wire : wire) =
          wire#add_terminal (object method set x = self#set_input_c x end)
   ```

3. *Would the definition be simpler if the type* `terminal` *were a function instead of an object (where* `type terminal = bool -> unit`*)?*

It would be slightly simpler because the input terminal could be set without the intermediate terminal object. The connect methods would have the following form.

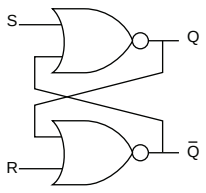
```
method connect_input_a (wire : wire) =
   wire#add_terminal self#set_input_a
```

4. *What is the purpose of the conditional* `if x <> value then` $\cdots$ *in the class* `wire`*?*

   The conditional prevents activity from propagating if it does not change the circuit values. It also means that the simulation will terminate, even for cyclic circuits, if the circuit becomes quiescent.

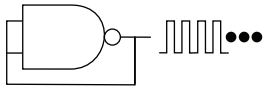
5. *Write a program for the following circuit, called a* SR latch.

S	*R*	*Action*
0	*0*	*Keep state*
0	*1*	$Q = 0$
1	*0*	$Q = 1$
1	*1*	$Q = 0, \overline{Q} = 0$




```
let gate1 = new nor2;;
let gate2 = new nor2;;
let wire1 = new wire;;
let wire2 = new wire;;
gate1#connect_output wire1;;
gate2#connect_output wire2;;
gate1#connect_input_b wire2;;
gate2#connect_input_a wire1;;
```

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 >
```

The scheduling queue can be implemented using the `Queue` standard library. This will use a FIFO policy, which is fair.

The scheduler has two methods. The method `main` runs the simulation until the queue is empty. The method `connect_input` takes a wire and a function, creates a terminal object, and add the object to the wire. When the terminal is set, it gets added to the scheduling queue. Note that the field `event_queue` is accessible in the inner terminal object.

```
type terminal = < set : bool -> unit; execute : unit >

class scheduler =
object (self : 'self)
   val event_queue : terminal Queue.t = Queue.create ()

   method main =
       while not (Queue.is_empty event_queue) do
          (Queue.take event_queue)#execute
       done
   method connect_input (wire : wire) (f : bool -> unit) =
      let term =
        object (term_self)
           val mutable value = false
           method set x =
              value <- x;
              Queue.add term_self event_queue
```

```
            method execute = f value
          end
      in
      wire#add_terminal term
end;;

let the_scheduler = new scheduler;;
```

The only other change to the simulation is to the methods `connect_input_?`.

```
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 : wire) =
      the_scheduler#connect_input wire self#set_input_a
   method connect_input_b (wire : wire) =
      the_scheduler#connect_input wire self#set_input_b
end
```

16 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?

In the former case, a person is not a name, and can't be used as a name. With multiple inheritance, the person *can* be used as a name. Multiple inheritance is more likely to be used in a situation where the name of a person and the person himself are

treated as the same thing. The former is more likely when the name is just a symbol for the person.

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?*

1. The class is simply split down the middle. The details of deletion must be implemented by the value, not the reference counting class, so the virtual method `delete` is used to connect the two objects.

```
class persistent_value filename =
object (self)
    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
    method private delete = Sys.remove filename
end

class virtual ref_value =
object (self)
    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 self#delete
    method private virtual delete : unit
end

class persistent_ref_value2 filename =
object
   inherit persistent_value filename
   inherit ref_value
end
```

2. The advantage of splitting the class is that we now have two more generic classes. For example, reference counting is general concept that can be re-used elsewhere in the program.

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`*?*

OCaml doesn't provide a way to rename fields, so the only thing we can do is to hide the field and use methods to access it. Suppose the class `programmer` is defined as follows.

```
class programmer =
object
   inherit person
   val favorite_language = "OCaml"
end
```

The renamed class `programmer'` could be defined as follows.

```
class type programmer_type' =
object
   val name : string
   val address : string
   method favorite_programming_language : string
end

class programmer' : programmer_type' =
object
   inherit programmer
   method favorite_programming_language = favorite_language
end
```

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
```

Inheriting from the `cell` twice would override the methods `get` and `set`, which is not what we want. We need to *rename* the methods first. For this list, it is sufficient to rename the methods in just one of the classes. Note that the cell value is hidden so that it is not overridden.

```
module IntCell =
struct
   module Cell = MakeCell (struct type t = int end);;
```

```
   class type cell_type =
   object
      method private get_int : int
      method private set_int : int -> unit
   end

   class cell i : cell_type =
   object (self)
      inherit Cell.cell i
      method private get_int = self#get
      method private set_int = self#set
   end
end

module ListCell = MakeCell (struct type t = int_list end);;

class cons i : int_cons =
object
    inherit IntCell.cell i as value
    inherit ListCell.cell None as link

    method hd = value#get_int
    method tl = link#get
    method set_hd = value#set_int
    method set_tl = link#set
end
```

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?*

Each function $f_i : int -> int$ would be defined in a separate class, where all the other functions are virtual.

```
class virtual fun_i =
object
   method virtual f_1 : int -> int
   ...
   method virtual f_{i-1} : int -> int
   method f_i i = ···
   ...
   method virtual f_n : int -> int
end
```

The final class would use multiple inheritance to tie the recursive knot.

```
class everything =
object
   inherit fun_1 ··· inherit fun_n
end
```

To reduce the amount of code, a single shared base class could be used to declare all the functions.

```
class virtual declarations =
object
   method virtual f_1 : int -> int
   ...
   method virtual f_n : int -> int
end

class virtual fun_i =
object
   inherit declarations
   method f_i i = ···
end
```

An advantage of the class representation is that the text required for the declarations is *much* smaller than needed for recursive modules.

17 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;;`

1. Legal; the type is `class c1 : object val x : 'a list end`

2. Legal; the type produced by the toploop is

   ```
   class c2 : object val x : 'a list ref end
   ```

   This typing seems a little strange, since `x` is not truly polymorphic (its type should be `'_a list ref`).

3. Legal; the type is `class c3 : 'a -> object val y : 'a end`.

4. Not legal; the method z has a polymorphic type 'a, which is not a parameter of the class definition.

5. Legal; the type is `class c5 : int -> object val y : int method z : int end`.

6. Legal; the constraint means the class type is not polymorphic. The type is `class c6 : int -> object method y : int 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
```

Here is one implementation.

```
class ['key, 'value] imp_map (compare : 'key -> 'key -> comparison) =
   let equal key1 (key2, _) = compare key1 key2 = Equal in
   object (self : 'self)
      val mutable elements : ('key * 'value) list = []
      method insert key value = elements <- (key, value) :: elements
      method find key = snd (List.find (equal key) elements)
   end;;
```

Exercise 17.3 *Reimplement the polymorphic map class from page* **??** *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.*

When `compare` is implemented as a method, the equality function `equal` must also be a method.

```
class virtual ['key, 'value] map =
  object (self : 'self)
    val elements : ('key * 'value)
    method add key value = {< elements = (key, value) :: elements >}
    method find key = snd (List.find (self#equal key) elements)

    method private equal key1 (key2, _) = compare key1 key2 = Equal
    method private virtual compare : 'key -> 'key -> ordering
  end;;

class ['value] int_map =
  object (self : 'self)
```

```
    inherit [int, 'value] map

    method private compare i j =
      if i < j then Smaller
      else if i > j then Larger
      else Equal
  end
```

Exercise 17.4 *In the class type definition* `['a] tree` *on page* **??**, *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
```

The alternate definition, using `'self` does not work because the class `leaf` must create a new internal node. The expression `new node x (self :> 'a tree) (self : 'a tree)` has type `'a tree`. It doesn't have type `'self`.

Exercise 17.5 *In the implementations for the* `['a] node` *and* `['a] leaf` *classes in Section* **??**, *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


module MakeTree (Compare : CompareSig)
  : sig val empty : Compare.t tree end =
struct
   open Compare
```

```
  type key = Compare.t
  type t = key tree

  class node x (l : t) (r : t) =
    object (self : 'self)
      val label = x
      val left = l
      val right = r
      method mem x =
        match compare x label with
           Smaller -> left#mem x
         | Larger -> right#mem x
         | Equal -> true
      method add x =
        match compare x label with
           Smaller -> {< left = left#add x >}
         | Larger -> {< right = right#add x >}
         | Equal -> self
  end;;

  class leaf =
    object (self : 'self)
      method mem _ = false
      method add x = new node x (self :> t) (self :> t)
    end;;

  let empty = new leaf;;
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?

The new definition is nearly the same. However, class definitions, even virtual ones, are not allowed in signatures or interfaces (`.mli` files), so there is a disadvantage to using the virtual class specification.

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
```

1. The annotation is legal, because fields are not part of the variance calculation.

2. The annotation is not legal, the type variable `'a` occurs negatively.

3. The annotation is legal, because private methods are not part of the variance calculation.

4. The annotation is legal.

5. The annotation is legal, because `'a` occurs only negatively in the type of the method `f`.

6. The annotation is legal.

7. The annotation is not legal, because the type `#foo` must be covariant.

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?*

1. The definition is legal because the type variable `'b` is included as a part of `'a`, so `'b` is not free in the definition.

2. Technically, all three annotations `+'a`, `-'a`, and `'a` are accepted. However, since the type `('b, 'b) pair` is covariant in `'b`, the definition is also covariant in `'a`.

3. The type is `(dog * cat) alt_animal_pair2`. Note that the type `dog * cat` is artificial, it has nothing to do with whether the class represents a pair.

4. If the constraint is left out, the type variables `'b` and `'c` become free in the class definition, so the definition is rejected.

5. The constraint `constraint 'a = 'b -> 'c` is also legal, but it means that the definition is no longer covariant in `'a`.

6. Strictly speaking, it isn't necessary. Suppose we have a class type `[-'a1, ..., -'an, +'b1, ..., +'bm] cl`. We can replace it with a single constraint `['c] cl` and the following constraint.

   ```
   constraint 'c = ('a1 * ··· * 'an) -> ('b1 * ··· * 'bn)
   ```

   We can't specify the variance of `'c`, but the constraint enforces the variance.

Exercise 17.9 *In the object implementation of the evaluator in Figure* **??***, 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?*

1. The problem is that the type #env is polymorphic. We can fix the definition by using a polymorphic method type.

```
class int_exp (i : int) : exp =
  object (self : 'self)
    method eval : 'a. (int #env as 'a) -> int = (fun _ -> i)
  end;;
```

2. There isn't really any reason to use the new definition, because an environment of type `int #env` is no more useful than an environment of type `int env`. The only savings is in the outermost call to the evaluator, where is may be possible to omit a coercion.

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.

The solution in both cases is to break apart the recursive definition by adding a type parameter to the class type `c1` that represents the class `c2`.

```
class type ['a, 'c2] pre_c1 = object method f : 'c2 -> 'a end
```

The solutions are then as follows.

1. ```
   class type c2 = object method g : (int, c2) pre_c1 end
   class type ['a] c1 = ['a, c2] pre_c1
   ```
2. ```
   class type c2 = object method g : 'a. ('a, c2) pre_c1 -> 'a end
   class type ['a] c1 = ['a, c2] pre_c1
   ```

Exercise 17.11 *As discussed in Section* **??**, *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.*

1. (a) The simplified class type `visitor` is not acceptable because the visitor methods are called with a plain expression `exp`, which isn't enough to do deconstruction.

   (b) A better way is to define class types for each of the kinds of expressions. For example, an object of class `add_exp` might have a class type definition `add_exp_type` that includes a method `subterms` to allow the visitor to decompose the sum.

```
class type add_exp_type =
  object ('self)
    inherit exp
    method subterms : exp * exp
  end

class add_exp (e1 : exp) (e2 : exp) =
  object (self : 'self)
    ...
    method subterms = e1, e2
  end
```

2. When a new kind of expression is added, the class type `visitor` must be extended with a new method definition, and all visitors must be updated to implement the new method. This is the same issue that appears when adding a new expression to the union representation.

3. The advantages of the approach is that the visitor can choose how to visit, and in what order. For example, the visitor might choose to visit an expression from the bottom up, or from the top down.

   The disadvantages are that the burden of traversal is shifted onto the visitor, which means 1) each kind visitor must duplicate the traversal code, and 2) the

traversal code may become out of date as the original definitions are modified.

One intermediate approach is to define virtual classes that provide code for the common traversals. Specific visitors would become subclasses of some version of a traversal class.

4. Some advantages are that the new code may be slightly more efficient because the visitor doesn't have to call the method `explode` to decompose the object. In many cases, the code may also be easier to write. The principal disadvantage is the the visitor no longer has access to the original object, which may be required if, for example, the visitor wishes to modify the object in-place.

5. Here is an example printer, based on the exploded visitor definition from part 3.

```
class print_visitor : visitor =
  object (self : 'self)
    method visit_int (e : int_exp_type) =
       print_int e#explode
    method visit_var (e : var_exp_type) =
       print_string e#explode
    method visit_add (e : add_exp_type) =
       let e1, e2 = e#explode in
       print_string "(";
       e1#accept (self :> visitor);
       print_string " + ";
       e2#accept (self :> visitor);
       print_string ")"
    method visit_if (e : if_exp_type) =
       let e1, e2, e3 = e#explode in
       print_string "(if ";
       e1#accept (self :> visitor);
       print_string " then ";
       e2#accept (self :> visitor);
       print_string " else ";
       e3#accept (self :> visitor);
       print_string ")"
    method visit_let (e : let_exp_type) =
       let v, e1, e2 = e#explode in
       printf "(let %s = " v;
       e1#accept (self :> visitor);
       print_string " in ";
       e2#accept (self :> visitor);
       print_string ")"
  end;;
```

6. Here are the definitions of the expression classes. We change the type of the

method `accept` slightly so that it is possible to pass a subtype of a `visitor` without coercing. The method `eval` has been omitted (we'll define it as a visitor in the next part).

```
class type ['a, 'exp] 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 accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a
  end

class int_exp (i : int) =
  object (self : 'self)
    method accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a =
      fun visitor -> visitor#visit_int i
  end

class var_exp v =
  object (self : 'self)
    method accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a =
      fun visitor -> visitor#visit_var v
  end

class add_exp (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a =
      fun visitor -> visitor#visit_add e1 e2
  end

class if_exp (e1 : #exp) (e2 : #exp) (e3 : #exp) =
  object (self : 'self)
    method accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a =
      fun visitor -> visitor#visit_if e1 e2 e3
  end

class let_exp (v : string) (e1 : #exp) (e2 : #exp) =
  object (self : 'self)
    method accept : 'a 'b. (('a, exp) #visitor as 'b) -> 'a =
      fun visitor -> visitor#visit_let v e1 e2
  end;;
```

7. The evaluator object needs to include the environment so that variables can be evaluated. Here is a completely pure implementation.

```
class eval_visitor : [int, exp] visitor =
  object (self : 'self)
```

```
  val env = new env

  method visit_int (i : int) =
     i
  method visit_var v =
     env#find v
  method visit_add (e1 : exp) (e2 : exp) =
     e1#accept self + e2#accept self
  method visit_if (e1 : exp) (e2 : exp) (e3 : exp) =
     if e1#accept self <> 0
     then e2#accept self
     else e3#accept self
  method visit_let v (e1 : exp) (e2 : exp) =
     e2#accept {< env = env#add v (e1#accept self) >}
end;;
```

Note that this code is nearly as concise as the version defined over the union representation of expressions.

Exercise 17.12 *We also stated in Section* **??** *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* **??**. *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* **??**.

```
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`*.*

1. Here is a complete definition. Since the type is open, there is a wildcard case for unknown expressions.

```
let rec eval1 env = function
   `Int i -> i
 | `Var v -> Env.find env v
 | `Add (e1, e2) -> eval1 env e1 + eval1 env e2
 | `If (e1, e2, e3) ->
      if eval1 env e1 <> 0
      then eval1 env e2
      else eval1 env e3
 | `Let (v, e1, e2) ->
      let i = eval1 env e1 in
      let env' = Env.add env v i in
      eval1 env' e2
 | _ ->
      raise (Failure "eval")
```

2. The pre-evaluator `pre_eval1` is very similar to `eval1`, but it is not directly recursive.

```
let pre_eval1 eval_subterm env = function
   `Int i -> i
 | `Var v -> Env.find env v
 | `Add (e1, e2) ->
      eval_subterm env e1 + eval_subterm env e2
 | `If (e1, e2, e3) ->
      if eval_subterm env e1 <> 0
      then eval_subterm env e2
      else eval_subterm env e3
 | `Let (v, e1, e2) ->
      let i = eval_subterm env e1 in
      let env' = Env.add env v i in
      eval_subterm env' e2
 | _ ->
      raise (Failure "eval")
```

3. The function `make_eval` wraps the pre-evaluator in a fixpoint definition.

```
let rec make_eval pre_eval env e =
   pre_eval (make_eval pre_eval) env e
```

4. The evaluator can be constructed using the function `make_eval`.

```
let eval2 env e = make_eval pre_eval2 env e
```

Bibliography

[1] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*. Academic Press, 1981.

[2] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.