Answers to Questions

Chapter 1 (Starting Off)

1

The expression `17` is of type **int** and is a value already. The expression `1 + 2 * 3 + 4` is of type **int** and evaluates to the value `11`, since the multiplication is done first. The expression `800 / 80 / 8` has type **int**. It is the same as `(800 / 80) / 8` rather than `800 / (80 / 8)` and evaluates to `1`.

The expression `400 > 200` has type **bool** because this is the type of the result of the comparison operator `>`. It evaluates to `true`. Similarly, `1 <> 1` has type **bool** and evaluates to `false`. The expression `true || false` is of type **bool** and evaluates to `true` since one of the operands is true. Similarly, `true && false` evaluates to `false` since one of the operands is false. The expression `if true then false else true` evaluates to `false` since the first (`then`) part of the conditional expression is chosen, and takes the place of the entire expression.

The expression `'%'` is of type **char** and is already a value. The expression `'a' + 'b'` has no type – it gives a type error because the `+` operator does not operate on characters.

2

The `mod` operator is of higher precedence than the `+` operator. So `1 + 2 mod 3` and `1 + (2 mod 3)` are the same expression, evaluating to `1 + 2` which is `3`, but `(1 + 2) mod 3` is the same as `3 mod 3`, which is `0`.

3

The expression evaluates to `11`. The programmer seems to be under the impression that spacing affects evaluation order. It does not, and so this use of space is misleading.

4

The expression `max_int + 1` evaluates to a number equal to `min_int`. Likewise, `min_int - 1` evaluates to a number equal to `max_int`. The number line "wraps around". This leads to the odd situation that `max_int + 1 < max_int` evaluates to `true`. It follows that when writing programs, we must be careful about what happens when numbers may be very large or very small.

5

OCaml accepts the program, but complains when it is run:

```
        OCaml

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

We will talk about such *exceptions* later in the book. They are used for program errors which cannot necessarily be found just by looking at the program text, but are only discovered during evaluation.

6

For `x mod y`:

when $y = 0$, OCaml prints `Exception: Division_by_zero`

when $y <> 0$, $x < 0$, the result will be negative

when $y <> 0$, $x = 0$, the result will be zero

This illustrates how even simple mathematical operators require careful specification when programming – we must be explicit about the rules.

7

It prevents unexpected values: what would happen if an integer other than `1` and `0` was calculated in the program – what would it mean? It is better just to use a different type. We can then show more easily that a program is correct.

8

The lowercase characters are in alphabetical order, for example `'p' < 'q'` evaluates to `true`. The uppercase characters are similarly ordered. The uppercase letters are all "smaller" than the lowercase characters, so for example `'A' < 'a'` evaluates to `true`. For type **bool**, `false` is considered "less than" `true`.

Chapter 2 (Names and Functions)

1

Just take in an integer and return the number multiplied by ten. The function takes and returns an integer, so the type is **int** $\rightarrow$ **int**.

```
        OCaml

# let times_ten x = x * 10;;
val times_ten : int -> int = <fun>
```

2

We must take two integer arguments, and use the && and <> operators to test if they are both non-zero. So the result will be of type **bool**. The whole type will therefore be **int** $\rightarrow$ **int** $\rightarrow$ **bool**.

```
        OCaml
```

```
# let both_non_zero x y =
    x <> 0 && y <> 0;;
val both_non_zero : int -> int -> bool = <fun>
```

3

Our function should take an integer, and return another one (the sum). So, it is type will be **int** → **int**. The base case is when the number is equal to 1. Then, the sum of all numbers from $1 \ldots 1$ is just 1. If not, we add the argument to the sum of all the numbers from $1 \ldots (n-1)$.

```
        OCaml

# let rec sum n =
    if n = 1 then 1 else n + sum (n - 1);;
val sum : int -> int = <fun>
```

The function is recursive, so we used **`let rec`** instead of **`let`**. What happens if the argument given is zero or negative?

4

The function will have type **int** → **int** → **int**. A number to the power of 0 is 1. A number to the power of 1 is itself. Otherwise, the answer is the current n multiplied by n^{x-1}.

```
        OCaml

# let rec power x n =
    if n = 0 then 1 else
      (if n = 1 then x else
         x * power x (n - 1));;
val power : int -> int -> int = <fun>
```

Notice that we had to put one **`if ... then ... else`** inside the **`else`** part of another to cope with the three different cases. The parentheses are not actually required, though, so we may write it like this:

```
        OCaml

# let rec power x n =
    if n = 0 then 1 else
      if n = 1 then x else
        x * power x (n - 1);;
val power : int -> int -> int = <fun>
```

In fact, we can remove the case for `n = 1` since `power x 1` will reduce to `x * power x 0` which is just `x`.

5

The function `isconsonant` will have type **char** → **bool**. If a lower case character in the range `'a'...'z'` is not a vowel, it must be a consonant. So we can reuse the `isvowel` function we wrote earlier, and negate its result using the `not` function:

```
        OCaml

# let isconsonant c = not (isvowel c);;
val isconsonant : char -> bool = <fun>
```

6

The expression is the same as **let** `x = 1` **in** (**let** `x = 2` **in** `x + x`), and so the result is `4`. Both instances of `x` in `x + x` evaluate to `2` since this is the value assigned to the name `x` in the nearest enclosing **let** expression.

7

We could simply return `0` for a negative or zero argument:

```
        OCaml

# let rec factorial x =
    if x <= 0 then 0 else
      if x = 1 then 1 else
        x * factorial (x - 1);;
val factorial : int -> int = <fun>
```

Note that `factorial` can fail in other ways too – if the number gets too big and "wraps around". For example, on the author's computer, `factorial 40` yields `-2188836759280812032`.

Chapter 3 (Case by Case)

1

We can just pattern match on the boolean. It does not matter, in this instance, which order the two cases are in.

```
not : bool → bool

let not x =
  match x with
    true -> false
  | false -> true
```

2

Recall our solution from the previous chapter:

```
sum : int → int

let rec sum n =
  if n = 1 then 1 else n + sum (n - 1)
```

Modifying it to use pattern matching:

```
sum_match : int → int

let rec sum_match n =
  match n with
    1 -> 1
  | _ -> n + sum_match (n - 1)
```

3

Again, modifying our solution from the previous chapter:

```
power_match : int → int → int

let rec power_match x n =
  match n with
    0 -> 1
  | 1 -> x
  | _ -> x * power_match x (n - 1)
```

5

This is the same as

```
match 1 + 1 with =
  2 ->
    (match 2 + 2 with
      3 -> 4
    | 4 -> 5)
```

(A match case belongs to its nearest enclosing **match**). So the expression evaluates to 5.

6

We write two functions of type **char** → **bool** like this:

```
isupper : char → bool
islower : char → bool

let isupper c =
  match c with
    'A'..'Z' -> true
  | _ -> false

let islower c =
  match c with
    'a'..'z' -> true
  | _ -> false
```

Alternatively, we might write:

```
isupper : char → bool
islower : char → bool

let isupper c =
  match c with
    'A'..'Z' -> true
  | _ -> false

let islower c =
  not (isupper c)
```

These two solutions have differing behaviour upon erroneous arguments (such as punctuation). Can you see why?

Chapter 4 (Making Lists)

1

This is similar to `odd_elements`:

```
even_elements : α list → α list

let rec even_elements l =
  match l with
    [] -> []                                  the list has zero elements
  | [_] -> []                                 the list has one element – drop it
  | _::b::t -> b :: even_elements t           h is the head, t the tail
```

But we can perform the same trick as before, by reversing the cases, to reduce their number:

```
even_elements : α list → α list

let rec even_elements l =
  match l with
    _::b::t -> b :: even_elements t          drop one, keep one, carry on
  | _ -> []                                  otherwise, no more to drop
```

2

This is like counting the length of a list, but we only count if the current element is `true`.

```
count_true : bool list → int

let rec count_true l =
  match l with
    [] -> []                                 no more
  | true::t -> 1 + count_true t              count this one
  | false::t -> count_true t                 but not this one
```

We can use an accumulating argument in an auxiliary function to make a tail recursive version:

```
count_true_inner : int → bool list → int
count_true : bool list → int

let rec count_true_inner n l =
  match l with
    [] -> n                                  no more; return the accumulator
  | true::t -> count_true_inner (n + 1) t    count this one
  | false::t -> count_true_inner n t         but not this one

let count_true l =
  count_true_inner 0 l                       initialize the accumulator with zero
```

3

To make a palindrome from any list, we can append it to its reverse. To check if a list is a palindrome, we can compare it for equality with its reverse (the comparison operators work over almost all types).

```
mk_palindrome : α list → α list
is_palindrome : α list → bool

let mk_palindrome l =
  l @ rev l

let is_palindrome l =
  l = rev l
```

4

We pattern match with three cases. The empty list, where we have reached the last element, and where we have yet to reach it.

```
drop_last : α list → α list

let rec drop_last l =
  match l with
    [] -> []
  | [_] -> []                                  it is the last one, so remove it
  | h::t -> h :: drop_last t                   at least two elements remain
```

We can build a tail recursive version by adding an accumulating list, and reversing it when finished (assuming a tail recursive `rev`, of course!)

```
drop_last_inner : α list → α list → α list
drop_last : α list → α list

let rec drop_last_inner a l =
  match l with
    [] -> rev a                                return the reversed accumulator
  | [_] -> rev a                               same, ignoring the last element
  | h::t -> drop_last_inner (h :: a) t         at least two elements remain

let drop_last l =
  drop_last_inner [] l
```

5

The empty list cannot contain the element; if there is a non-empty list, either the head is equal to the element we are looking for, or if not, the result of our function is just the same as the result of recursing on the tail.

Note that we are using the property that the `||` operator only evaluates its right hand side if the left hand side is false to limit the recursion – it really does stop as soon as it finds the element.

```
member : α → α list → bool

let rec member e l =
  match l with
    [] -> false
  | h::t -> h = e || member e t
```

6

If a list is empty, it is already a set. If not, either the head exists somewhere in the tail or it does not; if it does exist in the tail, we can discard it, since it will be included later. If not, we must include it.

```
make_set : α list → α list

let rec make_set l =
  match l with
    [] -> []
  | h::t -> if member h t then make_set t else h :: make_set t
```

For example, consider the evaluation of `make_set [4; 5; 6; 5; 4]`:

$$
\begin{aligned}
& \underline{\texttt{make\_set [4; 5; 6; 5; 4]}} \\
\Longrightarrow\quad & \underline{\texttt{make\_set [5; 6; 5; 4]}} \\
\Longrightarrow\quad & \underline{\texttt{make\_set [6; 5; 4]}} \\
\Longrightarrow\quad & \texttt{6 :: }\underline{\texttt{make\_set [5; 4]}} \\
\Longrightarrow\quad & \texttt{6 :: 5 :: }\underline{\texttt{make\_set [4]}} \\
\Longrightarrow\quad & \texttt{6 :: 5 :: 4 :: }\underline{\texttt{make\_set []}} \\
\Longrightarrow\quad & \underline{\texttt{6 :: 5 :: 4 :: []}} \\
\overset{*}{\Longrightarrow}\quad & \texttt{[6; 5; 4]}
\end{aligned}
$$

7

The first part of the evaluation of `rev` takes time proportional to the length of the list, processing each element once. However, when the lists are appended together, the order of the operations is such that the first argument becomes longer each time. The `@` operator, as we know, also takes time proportional to the length of its first argument. And so, this accumulating of the lists takes time proportional to the square of the length of the list.

$$
\begin{aligned}
& \underline{\texttt{rev [1; 2; 3; 4]}} \\
\Longrightarrow\quad & \underline{\texttt{rev [2; 3; 4]}}\texttt{ @ [1]} \\
\Longrightarrow\quad & \underline{\texttt{(rev [3; 4] @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{((rev [4] @ [3]) @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{(((rev [] @ [4]) @ [3]) @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{((([] @ [4]) @ [3]) @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{(([4] @ [3]) @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{([4, 3] @ [2]) @ [1]}} \\
\Longrightarrow\quad & \underline{\texttt{[4, 3, 2] @ [1]}} \\
\Longrightarrow\quad & \texttt{[4; 3; 2; 1]}
\end{aligned}
$$

By using an additional accumulating argument, we can write a version which operates in time proportional to the length of the list.

```
rev_inner : α list → α list → α list
rev : α list → α list

let rec rev_inner a l =
  match l with
    [] -> a
  | h::t -> rev_inner (h :: a) l

let rev l =
  rev_inner [] l
```

For the same list:

```
      rev [1; 2; 3; 4]
⟹    rev_inner [] [1; 2; 3; 4]
⟹    rev_inner [1] [2; 3; 4]
⟹    rev_inner [2; 1] [3; 4]
⟹    rev_inner [3; 2; 1] [4]
⟹    rev_inner [4; 3; 2; 1] []
⟹    [4; 3; 2; 1]
```

Chapter 5 (Sorting Things)

1

Simply add an extra **let** to define a name representing the number we will take or drop:

```
msort : α list → α list

let rec msort l =
  match l with
    [] -> []                                  we are done if the list is empty
  | [x] -> [x]                                and also if it only has one element
  | _ ->
      let x = length l / 2 in
        let left = take x l in                          get the left hand half
          let right = drop x l in                     and the right hand half
            merge (msort left) (msort right)          sort and merge them
```

2

The argument to `take` or `drop` is `length l / 2` which is clearly less than or equal to `length l` for all possible values of `l`. Thus, `take` and `drop` always succeed. In our case, `take` and `drop` are only called with `length l` is more than 1, due to the pattern matching.

3

We may simply replace the `<=` operator with the `>=` operator in the `insert` function.

```
insert : α → α list → α list

let rec insert x l =
  match l with
    [] -> [x]
  | h::t ->
      if x >= h
        then x :: h :: t
        else h :: insert x t
```

The `sort` function is unaltered.

4

We require a function of type α **list** $\rightarrow$ **bool**. List of length zero and one are, by definition, sorted. If the list is longer, check that its first two elements are in sorted order. If this is true, also check that the rest of the list is sorted, starting with the second element.

```
is_sorted : α list → bool

let rec is_sorted l =
  match l with
    [] -> true
  | [x] -> true
  | a::b::t -> a <= b && is_sorted (b :: t)
```

We can reverse the cases to simplify:

```
is_sorted : α list → bool

let rec is_sorted l =
  match l with
    a::b::t -> a <= b && is_sorted (b :: t)
  | _ -> true
```

5

Lists are compared starting with their first elements. If the elements differ, they are compared, and that is the result of the comparison. If both have the same first element, the second elements are considered, and so on. If the end of one list is reached before the other, the shorter list is considered smaller. For example:

```
[1] < [2] < [2; 1] < [2; 2]
```

These are the same principles you use to look up a word in a dictionary: compare the first letters – if same, compare the second etc. So, when applied to the example in the question, it has the effect of sorting the words into alphabetical order.

6

The **`let rec`** construct can be nested just like the **`let`** construct:

```
sort : α list → α list

let rec sort l =
  let rec insert x s =
    match s with
      [] -> [x]
    | h::t ->
        if x <= h
          then x :: h :: t
          else h :: insert x t
  in
    match l with
      [] -> []
    | h::t -> insert h (sort t)
```

We have renamed the second argument of the `insert` function to avoid confusion.

Chapter 6 (Functions upon Functions upon Functions)

1

Our function will have type **char list** → **char list**. We just match on the argument list: if it is empty, we are done. If it starts with an exclamation mark, we output a period, and carry on. If not, we output the character unchanged, and carry on:

```
calm : char list → char list

let rec calm l =
  match l with
    [] -> []
  | '!'::t -> '.' :: calm t
  | h::t -> h :: calm t
```

To use `map` instead, we write a simple function `calm_char` to process a single character. We can then use `map` to build our main function:

```
calm_char : char → char
calm : char list → char list

let calm_char x =
  match x with '!' -> '.' | _ -> x

let calm l =
  map calm_char l
```

This avoids the explicit recursion of the original, and so it is easier to see what is going on.

2

The `clip` function is of type **int** → **int** and is easy to write:

```
clip : int → int

let clip x =
  if x < 1 then 1 else
    if x > 10 then 10 else x
```

Now we can use map for the `cliplist` function:

```
cliplist : int list → int list

let cliplist l =
  map clip l
```

3

Just put the body of the clip function inside an anonymous function:

```
cliplist : int list → int list

let cliplist l =
  map
    (fun x ->
       if x < 1 then 1 else
         if x > 10 then 10 else x)
    l
```

4

We require a function `apply f n x` which applies function `f` a total of `n` times to the initial value `x`. The base case is when `n` is zero.

```
apply : (α → α) → int → α → α

let rec apply f n x =
  if n = 0
    then x                                          just x
    else f (apply f (n - 1) x)                      reduce problem size by one
```

Consider the type:

$$\overbrace{(\alpha \to \alpha)}^{\text{function f}} \to \overbrace{\textbf{int}}^{\text{n}} \to \overbrace{\alpha}^{\text{x}} \to \overbrace{\alpha}^{\text{result}}$$

The function `f` must take and return the same type α, since its result in one iteration is fed back in as its argument in the next. Therefore, the argument `x` and the final result must also have type α. For example, for $\alpha = \textbf{int}$, we might have a power function:

```
power : int → int → int

let power a b =
  apply (fun x -> x * a) b 1
```

So `power a b` calculates a^b.

5

We can add an extra argument to the `insert` function, and use that instead of the comparison operator:

```
insert : (α → α → bool) → α → α list → α list

let rec insert f x l =                              add extra argument f
  match l with
    [] -> [x]
  | h::t ->
      if f x h
        then x :: h :: t
        else h :: insert f x t                      remember to add f here too
```

Now we just need to rewrite the `sort` function.

```
sort : (α → α → bool) → α list → α list

let rec sort f l =
  match l with
    [] -> []
  | h::t -> insert f h (sort f t)
```

6

We cannot use map here, because the result list will not necessarily be the same length as the argument list. The function will have type $(\alpha \rightarrow \textbf{bool}) \rightarrow \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list}$.

```
filter : (α → bool) → α list → α list

let rec filter f l =
  match l with
    [] -> []
  | h::t ->
     if f h
       then h :: filter f t
       else filter f t
```

For example, `filter (fun x -> x mod 2 = 0) [1; 2; 4; 5]` evaluates to `[2; 4]`.

7

The function will have type $(\alpha \rightarrow \textbf{bool}) \rightarrow \alpha\ \textbf{list} \rightarrow \textbf{bool}$.

```
for_all : (α → bool) → α list → bool

let rec for_all f l =
  match l with
    [] -> true
  | h::t -> f h && for_all f t            true for this one, and all the others
```

For example, we can see if all elements of a list are positive: `for_all (fun x -> x > 0) [1; 2; -1]` evaluates to `false`. Notice that we are relying on the fact that && only evaluates its right hand side when the left hand side is true to limit the recursion.

8

The function will have type $(\alpha \rightarrow \beta) \rightarrow \alpha\ \textbf{list}\ \textbf{list} \rightarrow \beta\ \textbf{list}\ \textbf{list}$. We use `map` on each element of the list.

```
mapl : (α → β) → α list list → β list list

let rec mapl f l =
  match l with
    [] -> []
  | h::t -> map f h :: mapl f t
```

We have used explicit recursion to handle the outer list, and `map` to handle each inner list.

Chapter 7 (When Things Go Wrong)

1

The function `smallest_inner` takes a currently smallest found integer, a boolean value `found` indicating if we have found any suitable value or not, and the list of integers. It is started with `max_int` as the current value, so that any number is smaller than it, and `false` for `found` because nothing has been found yet.

```
smallest_inner : int → bool → int list → int
smallest : int list → int

let rec smallest_inner current found l =
  match l with
    [] ->
      if found then current else raise Not_found
  | h::t ->
      if h > 0 && h < current
        then smallest_inner h true t
        else smallest_inner current found t

let smallest l =
  smallest_inner max_int false l
```

Thus, the function raises an exception in the case of an empty list, or one which is non-empty but contains no positive integer, and otherwise returns the smallest positive integer in the list.

2

We just surround the call to `smallest` with an exception handler for `Not_found`.

```
smallest_or_zero : int list → int

let smallest_or_zero l =
  try smallest l with Not_found -> 0
```

3

We write a function `sqrt_inner` which, given a test number `x` and a target number `n` squares `x` and tests if it is more than `n`. If it is, the answer is `x - 1`. The test number will be initialized at `1`. The function `sqrt` raises our exception if the argument is less than zero, and otherwise begins the testing process.

```
sqrt_inner : int → int → int
sqrt : int → int

let rec sqrt_inner x n =
  if x * x > n then x - 1 else sqrt_inner (x + 1) n

exception Complex

let sqrt n =
  if n < 0 then raise Complex else sqrt_inner 1 n
```

4

We wrap up the function, handle the `Complex` exception and return.

```
safe_sqrt : int → int

let safe_sqrt n =
  try sqrt n with Complex -> 0
```

Chapter 8 (Looking Things Up)

1

Since the keys must be unique, the number of different keys is simply the length of the list representing the dictionary – so we can just use the usual `length` function.

2

The type is the same as for the `add` function. However, if we reach the end of the list, we raise an exception, since we did not manage to find the entry to replace.

```
replace : α → β → (α × β) list → (α × β) list

let rec replace k v l =
  match l with
    [] -> raise Not_found                          could not find it; fail
  | (k', v')::t ->
      if k = k'
        then (k, v) :: t                              found it – replace
        else (k', v') :: replace k v t              keep it, and keep looking
```

3

The function takes a list of keys and a list of values and returns a dictionary. So it will have type α **list** $\rightarrow$ β **list** $\rightarrow$ $(\alpha \times \beta)$ **list**.

```
mkdict : α list → β list → (α × β) list

let rec mkdict keys values =
  match keys, values with
    [], [] -> []
  | [], _ -> raise (Invalid_argument "mkdict")          unequal length
  | _, [] -> raise (Invalid_argument "mkdict")                  ditto
  | k::ks, v::vs -> (k, v) :: mkdict ks vs      make one pair, and move on
```

4

This will have the type $(\alpha$ **list** $\times$ β **list**$) \rightarrow (\alpha \times \beta)$ **list**. For the first time, we need to return a pair, building up both result lists element by element. This is rather awkward, since we will need the tails of both of the eventual results, so we can attach the new heads. We can do this by pattern matching.

```
mklists : (α × β) list → α list × β list

let rec mklists l =
  match l with
    [] -> ([], [])                                  build the empty pair
  | (k, v)::more ->                  we have at least one key-value pair
      match mklists more with                                make the rest
        (ks, vs) -> (k :: ks, v :: vs)                  and attach k and v
```

Here's a sample evaluation (we cannot really show it in the conventional way, so you must work through it whilst looking at the function definition):

```
                mklists [(1, 2); (3, 4); (5, 6)]
⟹                      mklists [(3, 4); (5, 6)]
⟹                              mklists [(5, 6)]
⟹                                    mklists []
⟹                                      ([], [])
⟹                                    ([5], [6])
⟹                            ([3; 5], [4; 6])
⟹                      ([1; 3; 5], [2; 4; 6])
```

Since the inner pattern match has only one form, and is complete, we can use **let** instead:

```
mklists : (α × β) list → α list × β list

let rec mklists l =
  match l with
    [] -> ([], [])                                  build the empty pair
  | (k, v)::more ->                          we have at least one key-value pair
      let (ks, vs) = mklists more in                              make the rest
        (k :: ks, v :: vs)                                  and attach k and v
```

5

We can use our `member` function which determines whether an element is a member of a list, building up a list of the keys we have already seen, and adding to the result list of key-value pairs only those with new keys.

```
dictionary_of_pairs_inner : α list → (α × β) list → (α × β) list
dictionary_of_pairs : (α × β) list → (α × β) list

let rec dictionary_of_pairs_inner keys_seen l =
  match l with
    [] -> []
  | (k, v)::t ->
      if member k keys_seen
        then dictionary_of_pairs_inner keys_seen t
        else (k, v) :: dictionary_of_pairs_inner (k :: keys_seen) t

let dictionary_of_pairs l =
  dictionary_of_pairs_inner [] l
```

How long does this take to run? Consider how long `member` takes.

6

We pattern match on the first list – if it is empty, the result is simply `b`. Otherwise, we add the first element of the first list to the union of the rest of its elements and the second list.

```
union : (α × β) list → (α × β) list → (α × β) list

let rec union a b =
  match a with
    [] -> b
  | (k, v)::t -> add k v (union t b)
```

We can verify that the elements of dictionary `a` have precedence over the elements of dictionary `b` by noting that `add` replaces a value if the key already exists.

Chapter 9 (More with Functions)

1

The function `g a b c` has type $\alpha \to \beta \to \gamma \to \delta$ which can also be written $\alpha \to (\beta \to (\gamma \to \delta))$. Thus, it takes an argument of type α and returns a function of type $\beta \to (\gamma \to \delta)$ which, when you give it an argument of type β returns a function of type $\gamma \to \delta$ which, when you give it an argument of type γ returns something of type δ. And so, we can apply just one or two arguments to the function `g` (which is called partial application), or apply all three at once. When we write **`let`** `g a b c = ...` this is just shorthand for **`let`** `g = ` **`fun`** `a -> ` **`fun`** `b -> ` **`fun`** `c -> ...`

2

The type of `member` is $\alpha \to \alpha$ **list** $\to$ **bool**, so if we partially apply the first argument, the type of `member x` must be α **list** $\to$ **bool**. We can use the partially-applied `member` function and `map` to produce a list of boolean values, one for each list in the argument, indicating whether or not that list contains the element. Then, we can use `member` again to make sure there are no `false` booleans in the list.

```
member_all : α → α list list → bool

let member_all x ls =
  let booleans = map (member x) ls in
    not (member false booleans)
```

We could also write:

```
member_all : α → α list list → bool

let member_all x ls =
  not (member false (map (member x) ls))
```

Which do you think is clearer? Why do we check for the absence of `false` rather than the presence of `true`?

3

The function `( / ) 2` resulting from the partial application of the `/` operator is the function which divides two by a given number, not the function which divides a given number by two. We can define a reverse divide function…

```
let rdiv x y = y / x
```

…which, when partially applied, does what we want.

4

The function `map` has type $(\alpha \to \beta) \to \alpha$ **list** $\to \beta$ **list**. The function `mapl` we wrote has type $(\alpha \to \beta) \to \alpha$ **list list** $\to \beta$ **list list**. So the function `mapll` will have type $(\alpha \to \beta) \to \alpha$ **list list list** $\to \beta$ **list list list**. It may be defined thus:

```
mapll : (α → β) → α list list list → β list list list

let mapll f l = map (map (map f)) l
```

But, as discussed, we may remove the `ls` too:

```
mapll : (α → β) → α list list list → β list list list

let mapll f = map (map (map f))
```

It is not possible to write a function which would map a function `f` over a list, or list of lists, or list of lists of lists depending upon its argument, because every function in OCaml must have a single type. If a function could map `f` over an α **list list** it must inspect its argument enough to know it is a list of lists, thus it could not be used on a β **list** unless $\beta = \alpha$ **list**.

5

We can write a function to truncate a single list using our `take` function, being careful to deal with the case where there is not enough to take, and then use this and `map` to build `truncate` itself.

```
truncate_l : int → α list → α list
truncate : int → α list list → α list list

let truncate_l n l =
  if length l >= n then take n l else l

let truncate n ll =
  map (truncate n) ll
```

Here we have used partial application of `truncate` to build a suitable function for `map`. Note that we could use exception handling instead of calling length, saving time:

```
truncate_l : int → α list → α list
truncate : int → α list list → α list list

let truncate_l n l =
  try take n l with Invalid_argument "take" -> l

let truncate n ll =
  map (truncate n) ll
```

You might, however, reflect on whether or not this is good style.

6

First, define a function which takes the given number and a list, returning the first element (or the number if none). We can then build the main function, using partial application to make a suitable function to give to `map`:

```
firstelt : α → α list → α
firstelts : α → α list list → α list

let firstelt n l =
  match l with [] -> n | h::_ -> h

let firstelts n l =
  map (firstelt n) l
```

Chapter 10 (New Kinds of Data)

1

We need two constructors – one for squares, which needs just a single integer (the length of a side), and one for rectangles which needs two integers (the width and height, in that order):

```
type rect =
  Square of int
| Rectangle of int * int
```

The name of our new type is rect. A rect is either a `Square` or a `Rectangle`. For example,

```
s : rect
r : rect

let s = Square 7

let r = Rectangle (5, 2)                                  width 5, height 2
```

2

We pattern match on the argument:

```
area : rect → int

let area r =
  match r with
    Square s -> s * s
  | Rectangle (w, h) -> w * h
```

3

This will be a function of type rect → rect. Squares remain unaltered, but if we have a rectangle with a bigger width than height, we rotate it by ninety degrees.

```
rotate : rect → rect

let rect r =
  match r with
    Rectangle (w, h) ->
      if w > h then Rectangle (h, w) else r
  | Square _ -> r
```

4

We will use `map` to perform our rotation on any rects in the argument list which need it. We will then use the sorting function from the previous chapter which takes a custom comparison function so as to just compare the widths.

```
width_of_rect : rect → int
rect_compare : rect → rect → bool
pack : rect list → rect list

let width_of_rect r =
  match r with
    Square s -> s
  | Rectangle (w, _) -> w

let rect_compare a b =
  width_of_rect a < width_of_rect b

let pack rects =
  sort rect_compare (map rotate rects)
```

For example, packing the list of rects

```
[Square 6; Rectangle (4, 3); Rectangle (5, 6); Square 2]
```

will give

```
[Square 2; Rectangle (3, 4); Rectangle (5, 6); Square 6]
```

5

We follow the same pattern as for the **list** type, being careful to deal with exceptional circumstances:

```
take : int → α sequence → α sequence
drop : int → α sequence → α sequence
map : (α → β) → α sequence → β sequence

let rec take n l =
  if n = 0 then Nil else
    match l with
      Nil -> raise (Invalid_argument "take")
    | Cons (h, t) -> Cons (h, take (n - 1) t)

let rec drop n l =
  if n = 0 then l else
    match l with
      Nil -> raise (Invalid_argument "drop")
    | Cons (_, l) -> drop (n - 1) l

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

6

We can use our `power` function from earlier:

```
type expr =
  Num of int
| Add of expr * expr
| Subtract of expr * expr
| Multiply of expr * expr
| Divide of expr * expr
| Power of expr * expr

evaluate : expr → int

let rec evaluate e =
  match e with
    Num x -> x
  | Add (e, e') -> evaluate e + evaluate e'
  | Subtract (e, e') -> evaluate e - evaluate e'
  | Multiply (e, e') -> evaluate e * evaluate e'
  | Divide (e, e') -> evaluate e / evaluate e'
  | Power (e, e') -> power (evaluate e) (evaluate e')
```

7

We can just wrap up the previous function:

```
evaluate_opt : expr → int option

let evaluate_opt e =
  try Some (evaluate e) with Division_by_zero -> None
```

Chapter 11 (Growing Trees)

1

Our function will have type $\alpha \to \alpha$ tree $\to$ **bool**. It takes a element to look for, a tree holding that kind of element, and returns `true` if the element is found, or `false` otherwise.

```
member_tree : α → α tree → bool

let rec member_tree x tr =
  match tr with
    Lf -> false
  | Br (y, l, r) -> x = y || member_tree x l || member_tree x r
```

Note that we have placed the test `x = y` first of the three to ensure earliest termination upon finding an appropriate element.

2

Our function will have type α tree $\to \alpha$ tree. A leaf flips to a leaf. A branch has its left and right swapped, and we must recursively flip its left and right sub-trees too.

```
flip_tree : α tree → α tree

let rec flip_tree tr =
  match tr with
    Lf -> Lf
  | Br (x, l, r) -> Br (x, flip_tree r, flip_tree l)
```

3

We can check each part of both trees together. Leaves are considered equal, branches are equal if their left and right sub-trees are equal.

```
equal_shape : α tree → α tree → bool

let rec equal_shape tr tr2 =
  match tr, tr2 with
    Lf, Lf ->
      true
  | Br (_, l, r), Br (_, l2, r2) ->
      equal_shape l l2 && equal_shape r r2
  | _, _ ->
      false
```

We can build a more general one which can compare trees of differing types to see if they have the same shape by using the `tree_map` function to make two trees, each with type **int** tree with all labels set to `0`, and then compare them using OCaml's built-in equality operator:

```
equal_shape : α tree → β tree → bool

let equal_shape tr tr2 =
  tree_map (fun _ -> 0) tr = tree_map (fun _ -> 0) tr2
```

4

We can use the tree insertion operation repeatedly:

```
tree_of_list : (α × β) list → (α × β) tree

let rec tree_of_list l =
  match l with
    [] -> Lf
  | (k, v)::t -> insert (tree_of_list t) k v
```

There will be no key clashes, because the argument should already be a dictionary. If it is not, earlier keys are preferred since `insert` replaces existing keys.

5

We can make list dictionaries from both tree dictionaries, append them, and build a new tree from the resultant list.

```
tree_union : (α × β) tree → (α × β) tree → (α × β) tree

let tree_union t t' =
  tree_of_list (list_of_tree t @ list_of_tree t')
```

The combined list may not be a dictionary (because it may have repeated keys), but `tree_of_list` will prefer keys encountered earlier. So, we put entries from `t'` after those from `t`.

6

We will use a list for the sub-trees of each branch, with the empty list signifying there are no more i.e. that this is the bottom of the tree. Thus, we only need a single constructor.

```
type 'a mtree = Branch of 'a * 'a mtree list
```

So, now we can define `size`, `total`, and `map`.

```
size : α mtree → int
total : α mtree → int
map_mtree : (α → β) → α mtree → β mtree

let rec size tr =
  match tr with
    Branch (e, l) -> 1 + sum (map size l)

let rec total tr =
  match tr with
    Branch (e, l) -> e + sum (map total l)

let rec map_mtree f tr =
  match tr with
    Branch (e, l) -> Branch (f e, map (map_mtree f) l)
```

In fact, when there is only one pattern to match, we can put it directly in place of the function's argument, simplifying these definitions:

```
size : α mtree → int
total : α mtree → int
map_mtree : (α → β) → α mtree → β mtree

let rec size (Branch (e, l)) =
  1 + sum (map size l)

let rec total (Branch (e, l)) =
  e + sum (map total l)

let rec map_mtree f (Branch (e, l)) =
  Branch (f e, map (map_mtree f) l)
```

Chapter 12 (In and Out)

1

A first attempt might be:

```
print_integers : int list → unit

let print_integers l =
  print_string "[";
  iter (fun i -> print_int i; print_string "; ") l;
  print_string "]"
```

However, there are two problems:

OCaml

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# print_integers [1; 2; 3];;
[1; 2; 3; ]- : unit = ()
```

There is an extra space after the last element, and a semicolon too. We can fix this, at the cost of a longer program:

```
print_integers_inner : int list → unit
print_integers : int list → unit

let rec print_integers_inner l =
  match l with
    [] -> ()
  | [i] -> print_int i
  | h::t -> print_int h; print_string "; "; print_integers_inner t

let print_integers l =
  print_string "[";
  print_integers_inner l;
  print_string "]"
```

Now, the result is correct:

OCaml

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# print_integers [1; 2; 3];;
[1; 2; 3]- : unit = ()
```

2

We must deal with the exception raised when `read_int` attempts to read something which is not an integer, as before. When that exception is caught, we try again, by recursively calling ourselves. The function ends when three integers are input correctly, returning them as a tuple.

```
read_three : unit → int × int × int

let rec read_three () =
  try
    print_string "Type three integers, pressing Enter after each";
    print_newline ();
    let x = read_int () in
      let y = read_int () in
        let z = read_int () in
          (x, y, z)
  with
    Failure "int_of_string" ->
      print_string "Failed to read integers; please try again";
      print_newline ();
      read_three ()
```

You may wonder why we used nested **let** ... **in** structures rather than just writing `(read_int (), read_int (), read_int ())` – the evaluation order of a tuple is not specified and OCaml is free to do what it wants.

3

We ask the user how many dictionary entries will be entered, eliminating the need for a special "I have finished" code. First, a function to read a given number of integer–string pairs, dealing with the usual problem of malformed integers:

```
read_dict_number : int → (int × string) list

let rec read_dict_number n =
  if n = 0 then [] else
    try
      let i = read_int () in
        let name = read_line () in
          (i, name) :: read_dict_number (n - 1)
    with
      Failure "int_of_string" ->
        print_string "This is not a valid integer."
        print_newline ();
        print_string "Please enter integer and name again."
        print_newline ();
        read_dict_number n
```

And now, asking the user how many entries there will be, and calling our first function:

```
read_dict : unit → (int × string) list

exception BadNumber

let rec read_dict () =
  print_string "How many dictionary entries to input?";
  print_newline ();
  try
    let n = read_int () in
      if n < 0 then raise BadNumber else read_dict_number n
  with
    Failure "int_of_string" ->
      print_string "Not a number. Try again";
      print_newline ();
      read_dict ()
  | BadNumber ->
      print_string "Number is negative. Try again";
      print_newline ();
      read_dict ()
```

Notice that we defined, raised, and handled our own exception `BadNumber` to deal with the user asking to read a negative number of dictionary entries – this would cause `read_dict_number` to fail to return.

4

If we write a function to build the list of integers from 1 to n (or the empty list if n is zero):

```
numlist : int → int list

let rec numlist n =
  match n with
    0 -> []
  | _ -> numlist (n - 1) @ [n]
```

We can then write a function to output a table of a given size to an output channel.

```
write_table_channel : in_channel → int → unit

let write_table_channel ch n =
  iter
    (fun x ->
       iter
         (fun i ->
           output_string ch (string_of_int i);
           output_string ch "\t")
         (map (( * ) x) (numlist n));
       output_string ch "\n")
    (numlist n)
```

Look at this carefully. We are using nested calls to `iter` to build the two-dimensional table from one-dimensional lists. Can you separate this into more than one function? Which approach do you think is more readable?

We can test `write_table_channel` most easily by using the built-in output channel `stdout` which just writes to the screen:

```
        OCaml

# write_table_channel stdout 5;;
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
- : unit = ()
```

Now we just need to wrap it in a function to open an output file, write the table, and close the output, dealing with any errors which may arise.

```
table : string → int → unit

exception FileProblem

let table filename n =
  if n < 0 then raise (Invalid_argument "table") else
    try
      let ch = open_out filename in
        write_table_channel ch n;
        close_out ch
    with
      _ -> raise FileProblem
```

In addition to raising `Invalid_argument` in the case of a negative number, we handle all possible exceptions to do with opening, writing to, and closing the file, re-raising them as our own, predefined one. Is this good style?

5

We write a simple function to count the lines in a channel by taking a line, ignoring it, and adding one to the result of taking another line; our recursion ends when an `End_of_file` exception is raised – it is caught and 0 ends the summation.

The main function `countlines` just opens the file, calls the first function, and closes the file. Any errors are caught and re-raised using the built-in `Failure` exception.

```
countlines_channel : in_channel → int
countlines : string → int

let rec countlines_channel ch =
  try
    let _ = input_line ch in
      1 + countlines_channel ch
  with
    End_of_file -> 0

let countlines file =
  try
    let ch = open_in file in
      let result = countlines_channel ch in
        close_in ch;
        result
  with
    _ -> raise (Failure "countlines")
```

6

As usual, let us write a function to deal with channels, and then deal with opening and closing files afterward. Our function takes an input channel and an output channel, adds the line read from the input to the output, follows it with a newline character, and continues. It only ends when the `End_of_file` exception is raised inside `input_line` and caught.

```
copy_file_ch : in_channel → out_channel → unit

let rec copy_file_ch from_ch to_ch =
  try
    output_string to_ch (input_line from_ch);
    output_string to_ch "\n";
    copy_file_ch from_ch to_ch
  with
    End_of_file -> ()
```

Now we wrap it up, remembering to open and close both files and deal with the many different errors which might occur.

```
copy_file : string → string → unit

exception CopyFailed

let copy_file from_name to_name =
  try
    let from_ch = open_in from_name in
      let to_ch = open_out to_name in
        copy_file_ch from_ch to_ch;
        close_in from_ch;
        close_out to_ch
  with
    _ -> raise CopyFailed
```

Chapter 13 (Putting Things in Boxes)

1

Two references, `x` and `y`, of type **int ref** have been created. Their initial values are 1 and 2. Their final values are 2 and 4. The type of the expression is **int** because this is the type of `!x + !y`, and the result is 6.

2

The expression `[ref 5; ref 5]` is of type **int ref list**. It contains two references each containing the integer 5. Changing the contents of one reference will not change the contents of the other. The expression **let** `x = ref 5` **in** `[x; x]` is also of type **int ref list** and also contains two references to the integer 5. However, altering one will alter the other:

```
        OCaml

# let r = let x = ref 5 in [x; x];;
val r : int ref list = [{contents = 5}; {contents = 5}]
# match r with h::_ -> h := 6;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
- : unit = ()
# r;;
- : int ref list = [{contents = 6}; {contents = 6}]
```

3

We can write a function `forloop` which takes a function of type **int** $\rightarrow \alpha$ (where alpha would normally be **unit**), together with the start and end numbers:

```
forloop : (int → α) → int → int → unit

let rec forloop f n m =
  if n <= m then
    begin
      f n;
      forloop f (n + 1) m
    end
```

For example:

```
        OCaml

# forloop print_int 2 10;;
2345678910- : unit = ()
# forloop print_int 2 2;;
2- : unit = ()
```

4

`[|1; 2; 3|]` : **int array**

`[|true; false; true|]` : **bool array**

`[|[|1|]|]` : (**int array**) **array** which is **int array array**

`[|[1; 2; 3]; [4; 5; 6]|]` : **int list array**

`[|1; 2; 3|].(2)` : **int**, has value 2

`[|1; 2; 3|].(2) <- 4` : **unit**, updates the array to `[|1; 2; 4|]`

5

We use a **`for`** construct:

```
array_sum : int array → int

let array_sum a =
  let sum = ref 0 in
    for x = 0 to Array.length a - 1 do
      sum := !sum + a.(x)
    done;
    !sum
```

Note that this works for the empty array, because a **`for`** construct where the second number is less than the first never executes its expression.

6

Since we wish to reverse the array in place, our function will have type α **array** $\rightarrow$ **unit**. Our method is to proceed from the first element to the half-way point, swapping elements from either end of the array. If the array has odd length, the middle element will not be altered.

```
array_rev : α array → unit

let array_rev a =
  if a <> [||] then
    for x = 0 to Array.length a / 2 do
      let t = a.(x) in
        a.(x) <- a.(Array.length a - 1 - x);
        a.(Array.length a - 1 - x) <- t
    done
```

Note that we must check for the case where the array is empty; otherwise there would be an invalid attempt to access element zero inside the **`for`** loop.

7

We will represent the **int array array** as an array of columns so that `a.(x).(y)` is the element in column `x` and row `y`.

```
table : int → int array array

let table n =
  let a = Array.make n [||] in
    for x = 0 to n - 1 do
      a.(x) <- Array.make n 0
    done;
    for y = 0 to n - 1 do
      for x = 0 to n - 1 do
        a.(x).(y) <- (x + 1) * (y + 1)
      done
    done;
    a
```

Note that the result is correct for `table 0`.

8

The difference between the codes for `'a'` and `'A'`, or `'z'` and `'Z'` is 32, so we add or subtract as appropriate. Codes not in those ranges are unaltered.

```
uppercase : char → char
lowercase : char → char

let uppercase x =
  if int_of_char x >= 97 && int_of_char x <= 122
    then char_of_int (int_of_char x - 32)
    else x

let lowercase x =
  if int_of_char x >= 65 && int_of_char x <= 90
    then char_of_int (int_of_char x + 32)
    else x
```

9

Periods, exclamation marks and question marks may appear in multiples, leading to a wrong answer. The number of characters does not include newlines. It is not clear how quotations would be handled. Counting the words by counting spaces is inaccurate – a line with ten words will count only nine.

Chapter 14 (The Other Numbers)

1

We calculate the ceiling and floor, and return the closer one, being careful to make sure that a point equally far from the ceiling and floor is rounded up.

```
round : float → float

let round x =
  let c = ceil x in
    let f = floor x in
      if c -. x <= x -. f then c else f
```

The behaviour with regard to values such as `infinity` and `nan` is fine, since it always returns the result of either `floor` or `ceil`.

2

The function returns another point, and is simple arithmetic.

```
between : float × float → float × float → float × float

let between (x, y) (x', y') =
  ((x +. x') /. 2., (y +. y') /. 2.)
```

3

The whole part is calculated using the built-in `floor` function. We return a tuple, the first number being the whole part, the second being the original number minus the whole part. In the case of a negative number, we must be careful – `floor` always rounds downward, not toward zero!

```
parts : float → float × float

let rec parts x =
  if x < 0. then
    let a, b = parts (-. x) in
      (-. a, b)
    else
      (floor x, x -. floor x)
```

Notice that we are using the unary operator `-.` to make the number positive.

4

We need to determine at which column the asterisk will be printed. It is important to make sure that the range $0 \ldots 1$ is split into fifty equal sized parts, which requires some careful thought. Then, we just print enough spaces to pad the line, add the asterisk, and a newline character.

```
star : float → unit

let star x =
  let i = int_of_float (floor (x *. 50.)) in
    let i' = if i = 50 then 49 else i in
      for x = 1 to i' - 1 do print_char ' ' done;
      print_char '*';
      print_newline ()
```

5

We use a reference to hold the current value, starting at the beginning of the range, and then loop until the we are outside the range.

```
plot : (float → float) → float → float → float → unit

let plot f a b dy =
  let pos = ref a in
    while !pos <= b do
      star (f !pos);
      pos := !pos +. dy
    done
```

No allowance has been made here for bad arguments (for example, `b` smaller than `a`). Can you extend our program to move the zero-point to the middle of the screen, so that the sine function can be graphed even when its result is less than zero?

Chapter 15 (The OCaml Standard Library)

1

A non-tail-recursive one is simple:

```
concat : α list list → α list

let rec concat l =
  match l with
    [] -> []
  | h::t -> h @ concat t
```

To make a tail-recursive one, we can use an accumulator, reversing each list as we append it, and reversing the result. `List.rev` is tail-recursive already.

```
concat_tail : α list → α list list → α list
concat : α list list → α list

let rec concat_tail a l =
  match l with
    [] -> List.rev a
  | h::t -> concat_tail (List.rev h @ a) t

let concat l =
  concat_tail [] l
```

2

We can use `List.mem`, partially applied, to map over the list of lists. We then make sure that `false` is not in the resultant list, again with `List.mem`.

```
all_contain_true : bool list list → bool

let all_contain_true l =
  not (List.mem false (List.map (List.mem true) l))
```

3

The `String.iter` function calls a user-supplied function of type **char** → **unit** on each character of the string. We can use this to increment a counter when an exclamation mark is found.

```
count_exclamations : string → int

let count_exclamations s =
  let n = ref 0 in
    String.iter (function '!' -> n := !n + 1 | _ -> ()) s;
    !n
```

The contents of the counter is then the result of the function.

4

We can use the `String.map` function, which takes a user-supplied function of type **char** → **char** and returns a new string, where each character is the result of the mapping function on the character in the same place in the old string.

```
calm : string → string

let calm =
  String.map (function '!' -> '.' | x -> x)
```

Notice that we have taken advantage of partial application to erase the last argument as usual.

5

Looking at the documentation for the `String` module we find the following:

```
val concat : string -> string list -> string

      String.concat sep sl concatenates the list of strings sl, inserting the separator string sep between each.
```

So, by using the empty string as a separator, we have what we want:

```
concat: string list → string

let concat =
  String.concat ""
```

6

We can use the functions `create`, `add_string`, and `contents` from the `Buffer` module together with the usual list iterator `List.iter`:

```
concat : string list → string

let concat ss =
  let b = Buffer.create 100 in
    List.iter (Buffer.add_string b) ss;
    Buffer.contents b
```

The initial size of the buffer, 100, is arbitrary.

7

We repeatedly check if the string we are looking for is right at the beginning of the string to be searched. If not, we chop one character off the string to be searched, and try again. Every time we find a match, we increment a counter.

```
occurrences : string → string → int

let occurrences ss s =                                  occurrences of ss in s
  if ss = "" then 0 else                                        defined as zero
    let num = ref 0 in                                  occurrences found so far
      let str = ref s in                                          current string
        while
          String.length ss <= String.length !str && !str <> ""
        do
          if String.sub !str 0 (String.length ss) = ss then
            num := !num + 1;
          str := String.sub !str 1 (String.length !str - 1)
        done;
        !num
```

You might consider that writing this function with lists of characters rather than strings would be easier. Unfortunately, it would be slow, and these kinds of searching tasks are often required to be very fast.

Chapter 16 (Building Bigger Programs)

1

First, we extend the `Textstat` module to allow frequencies to be counted and expose it through the interface, shown in Figures 16.6 and 16.7. Then the main program is as shown in Figure 16.8.

2

We can write two little functions – one to read all the lines from a file, and one to write them. The main function, then, reads the command line to find the input and output file names, reads the lines from the input, reverses the list of lines, and writes them out. If a problem occurs, the exception is printed out. If the command line is badly formed, we print a usage message and exit. This is shown in Figure 16.9.

Note that there is a problem if the file has no final newline – it will end up with one. How might you solve that?

```
type stats

val lines : stats -> int

val characters : stats -> int

val words : stats -> int

val sentences : stats -> int

val frequency : stats -> char -> int

val stats_from_file : string -> stats
```

Figure 16.6: `textstat.mli`

3

We can simply do something (or nothing) a huge number of times using a **`for`** loop.

```
(* A program which takes sufficiently long to run that we can distinguish
between the ocamlc and ocamlopt compilers *)

for x = 1 to 10000000 do
  ()
done
```

On many systems, typing `time` followed by a space and the usual command will print out on the screen how long the program took to run. For example, on the author's computer:

```
$ ocamlc bigloop.ml -o bigloop
$ time ./bigloop

real  0m1.896s
user  0m1.885s
sys   0m0.005s

$ ocamlopt bigloop.ml -o bigloop
$ time ./bigloop

real  0m0.022s
user  0m0.014s
sys   0m0.003s
```

You can see that, when compiled with `ocamlc`, it takes 1.9s to run, but when compiled with `ocamlopt` just 0.22s.

```
(* Text statistics *)
type stats = int * int * int * int * int array

(* Utility functions to retrieve parts of a stats value *)
let lines (l, _, _, _, _) = l

let characters (_, c, _, _, _) = c

let words (_, _, w, _, _) = w

let sentences (_, _, _, s, _) = s

let frequency (_, _, _, _, h) x = h.(int_of_char x)

(* Read statistics from a channel *)
let stats_from_channel in_channel =
  let lines = ref 0 in
  let characters = ref 0 in
  let words = ref 0 in
  let sentences = ref 0 in
  let histogram = Array.make 256 0 in
    try
      while true do
        let line = input_line in_channel in
          lines := !lines + 1;
          characters := !characters + String.length line;
          String.iter
            (fun c ->
               match c with
               '.'  | '?'  | '!' -> sentences := !sentences + 1
             | ' ' -> words := !words + 1
             | _ -> ())
            line;
          String.iter
            (fun c ->
               let i = int_of_char c in
                 histogram.(i) <- histogram.(i) + 1)
            line
      done;
      (0, 0, 0, 0, [||]) (* Just to make the type agree *)
    with
      End_of_file -> (!lines, !characters, !words, !sentences, histogram)

(* Read statistics, given a filename. Exceptions are not handled *)
let stats_from_file filename =
  let channel = open_in filename in
    let result = stats_from_channel channel in
      close_in channel;
      result
```

Figure 16.7: `textstat.ml`

```
let print_histogram stats =
  print_string "Character frequencies:\n";
  for x = 0 to 255 do
    let freq = Textstat.frequency stats (char_of_int x) in
      if freq > 0 then
        begin
          print_string "For character '";
          print_char (char_of_int x);
          print_string "' (character number ";
          print_int x;
          print_string ") the count is ";
          print_int freq;
          print_string ".\n"
          end
        done
in
  try
    begin match Sys.argv with
      [|_; filename|] ->
        let stats = Textstat.stats_from_file filename in
          print_string "Words: ";
          print_int (Textstat.words stats);
          print_newline ();
          print_string "Characters: ";
          print_int (Textstat.characters stats);
          print_newline ();
          print_string "Sentences: ";
          print_int (Textstat.sentences stats);
          print_newline ();
          print_string "Lines: ";
          print_int (Textstat.lines stats);
          print_newline ();
          print_histogram stats
    | _ ->
         print_string "Usage: stats <filename>\n"
    end
  with
    e ->
      print_string "An error occurred: ";
      print_string (Printexc.to_string e);
      print_newline ();
      exit 1
```

Figure 16.8: `stats.ml`

```
(* Reverse the lines in a file *)

let putlines lines filename =
  let channel = open_out filename in
    List.iter
      (fun s ->
         output_string channel s;
         output_char channel '\n')
      lines;
    close_out channel

let getlines filename =
  let channel = open_in filename in
    let lines = ref [] in
      try
        while true do
          lines := input_line channel :: !lines
        done;
        []
      with
        End_of_file ->
          close_in channel;
          List.rev !lines

let _ =
  match Sys.argv with
    [|_; infile; outfile|] ->
      begin
        try
          let lines = List.rev (getlines infile) in
            putlines lines outfile
        with
          e ->
            print_string "There was an error. Details follow:\n";
            print_string (Printexc.to_string e);
            print_newline ();
            exit 1
      end
  | _ ->
      print_string "Usage: reverse input_filename output_filename\n";
      exit 1
```

Figure 16.9: `reverse.ml`

4

We can get all the lines in the file using our `getlines` function from question two. The main function simply calls `string_in_line` on each line, printing it if `true` is returned.

The interesting function is `string_in_line`. To see if `term` is in `line` we start at position 0. The condition for the term having been found is a combination of boolean expressions. The first ensures that we are not so far through the string that the expression could not possibly fit at the current position. The second checks to see if the term is found at the current position by using the function `String.sub` from the OCaml Standard Library. If not, we carry on. This is illustrated in Figure 16.10.

```
let rec string_in_line term line pos =
    pos + String.length term <= String.length line
  &&
    (String.sub line pos (String.length term) = term
    || string_in_line term line (pos + 1))

let getlines filename =
  let channel = open_in filename in
    let lines = ref [] in
      try
        while true do
          lines := input_line channel :: !lines
        done;
        []
      with
        End_of_file ->
          close_in channel;
          List.rev !lines

let _ =
  match Sys.argv with
    [|_; searchterm; filename|] ->
      begin
        try
          List.iter
            (fun line ->
               if string_in_line searchterm line 0 then
                 begin
                   print_string line;
                   print_newline ()
                 end)
            (getlines filename)
        with
          e ->
            print_string "An error occurred:\n";
            print_string (Printexc.to_string e);
            print_newline ()
      end
  | _ ->
    print_string "Usage: search search_term filename\n"
```

Figure 16.10: `search.ml`