Answers to Questions

Hints may be found on page 189.

Chapter 1 (Unravelling "Fold")

1

This can be achieved by folding the subtraction operator over the deductions, with the starting accumulator set to the budget:

```
deduct : int → int list → int

let deduct budget expenses =
  List.fold_left ( - ) budget expenses

let deduct = List.fold_left ( - )
```

Partial application can be used, as in the second definition.

2

We can use an accumulator starting at zero, and increment it once for each element processed:

```
length : α list → int

let length l =
  List.fold_left (fun a _ -> a + 1) 0 l
```

Since we ignore the element itself, the function is polymorphic.

3

If the list is empty, we return `None`, otherwise we use a left fold which simply replaces the accumulator with each successive element from the list. We must initialize the accumulator, so we pick the first element for that (we have already eliminated the case where there are no elements, so `List.hd` will succeed).

```
last : α list → α option

let last l =
  match l with
    [] -> None
  | _ -> Some (List.fold_left (fun _ e -> e) (List.hd l) l)
```

Not quite idiomatic.

4

If we start from the left, consing each element to the accumulator (which is initially the empty list), the list will be reversed.

```
rev : α list → α list

let rev l =
  List.fold_left (fun a e -> e :: a) [] l
```

Since the accumulator is the empty list (which has type α list), the function remains polymorphic, having the type we would expect.

5

The accumulator begins set to `false`. For each element, we calculate the logical OR of the element tested for equality and the accumulator. If at least one `true` occurs, the result will be `true`.

```
member : α → α list → bool

let member x l =
  List.fold_left (fun a e -> e = x || a) false l
```

Note that this is less efficient than `List.mem` because there is no early exit – the whole list is processed in every case.

6

This is a classic problem. We either need to

- add a space after each word except for the last; or
- add a space before each word except for the first.

With `fold_left` we can detect when we are at the first word, by inspecting the accumulator, and use the second method.

```
sentence : string list → string

let sentence words =
  List.fold_left
    (fun a e -> if a = "" then e else a ^ " " ^ e)
    ""
    words
```

Note that the requirement that the words be non-empty is important here. The efficiency is poor, however, since each string concatenation builds a new string. The Standard Library **Buffer** module is a better approach here.

7

We can use the built-in `max` function to update the accumulator.

```
max_depth : α tree → int

let max_depth l =
  List.fold_tree (fun _ l r -> 1 + max l r) 0 l
```

The current element is ignored.

8

We can compare the speed of `List.mem` and `member` with the help of the `Unix.gettimeofday` function, as shown in Figure A.1. On the Author's machine, this results in:

```
Our member took 2.513232 seconds
List.mem took 1.162159 seconds
```

There is a significant speed penalty in our version of the member function, at least for this scenario.

Chapter 2 (Being Lazy)

1

This is similar to the `lseq` function in the text, but we double every time instead of adding one.

```
ldouble : int → int lazylist
thedoubles : int lazylist

let rec ldouble n =
  Cons (n, fun () -> ldouble (n * 2))

let thedoubles = ldouble 1
```

```
l : int list
t : float
t' : float
t'' : float

let l = [1; 2; 3; 2; 1; 2; 2; 56; 32; 2; 34; 4; 2]

let t = Unix.gettimeofday ()

let _ =
  for x = 1 to 10_000_000 do ignore (member 56 l) done

let t' = Unix.gettimeofday ()

let _ =
  for x = 1 to 10_000_000 do ignore (List.mem 56 l) done

let t'' = Unix.gettimeofday ()

let _ =
  Printf.printf "Our member took %f seconds\n" (t' -. t);
  Printf.printf "List.mem took %f seconds\n" (t'' -. t')
```

Figure A.1

Having written the function which, given a number, doubles from that point, we then just code the list itself by starting at 1.

2

If we are asked to fetch the 0th element, we already have it – as the head of the lazy list. If not, we force evaluation of the tail, and recurse.

```
lnth : α lazylist → int → α

let rec lnth (Cons (h, tf)) n =
  match n with
    0 -> h
  | _ -> lnth (tf ()) (n - 1)
```

Of course, this does not terminate on bad inputs (when $n < 0$). Error detection should be added.

3

Below, the function `lrepeating_inner` takes the current list `c` and the original list `l`. Matching on `c`, we build the lazy list. If we reach the last element of the input list, we start again, with the original list, which is always retained.

```
lrepeating_inner : α list → α list → α lazylist
lrepeating : α tree → α lazylist

let rec lrepeating_inner c l =
  match c with
    [] -> raise (Invalid_argument "lrepeating: empty list")
  | [x] -> Cons (x, fun () -> lrepeating_inner l l)
  | h::t -> Cons (h, fun () -> lrepeating_inner t l)

let lrepeating l = lrepeating_inner l l
```

Note that we cannot build from an empty list, since there could be no head.

4

The first two fibonacci numbers are defined to be 0 and 1. Thereafter, we keep the current and previous number, and generate the lazy list.

```
fibonacci_inner : int → int → int lazylist
fibonacci : int lazylist

let rec fibonacci_inner x y =
  Cons (x, fun () -> fibonacci_inner y (x + y))

let fibonacci = fibonacci_inner 0 1
```

5

This is slightly delicate. We must force the tail twice, to reveal new elements for the heads of the two output lists, and the final tail for the next time each list is forced.

```
unleave : α lazylist → α lazylist × α lazylist

let rec unleave (Cons (h, tf)) =
  let Cons (h', tf') = tf () in
    let t = tf' () in
      (Cons (h, fun () -> fst (unleave t)),
       Cons (h', fun () -> snd (unleave t)))
```

Note that we cannot hoist the calculation of `unleave t` into a **`let`** so as to do it once, since to do so would not delay evaluation.

6

If we write a function which, given a number, gives the correct string, then the lazy list itself is easy to build with `lmap` and `lseq`.

```
letter_string : int → string
alphas : string lazylist

let rec letter_string n =
  if n <= 26 then
    Char.escaped (char_of_int (n + 64))
  else
    letter_string ((n - 1) / 26) ^
    letter_string (((n - 1) mod 26) + 1)

let alphas =
  lmap letter_string (lseq 1)
```

Chapter 3 (Named Tuples with Records)

1

Since a reference is just a record with a mutable field `contents`, we can use the `<-` construct:

```
        OCaml

# let x = ref 0;;
val x : int ref = {contents = 0}
# x.contents <- 1;;
- : unit = ()
# x;;
- : int ref = {contents = 1}
```

There is no reason to use this rather than `:=` of course.

2

The function `Unix.time` returns the current time. The function `Unix.localtime` builds from this a record of type **Unix.tm**. First we will need two ancillary functions:

```
string_of_month : int → string
string_of_day : int → string

let string_of_month m =
  match m with
    0 -> "January"
  | 1 -> "February"
  | 2 -> "March"
  | 3 -> "April"
  | 4 -> "May"
  | 5 -> "June"
  | 6 -> "July"
  | 7 -> "August"
  | 8 -> "September"
  | 9 -> "October"
  | 10 -> "November"
  | 11 -> "December"
  | _ -> raise (Invalid_argument "string_of_month")

let string_of_day d =
  match d with
    0 -> "Sunday"
  | 1 -> "Monday"
  | 2 -> "Tuesday"
  | 3 -> "Wednesday"
  | 4 -> "Thursday"
  | 5 -> "Friday"
  | 6 -> "Saturday"
  | _ -> raise (Invalid_argument "string_of_day")
```

Now the main function, which both uses the shortened form of a record pattern, and names only the fields we need:

```
print_time : unit → unit

let string_of_time () =
  let
    {Unix.tm_min;
     Unix.tm_hour;
     Unix.tm_mday;
     Unix.tm_mon;
     Unix.tm_year;
     Unix.tm_wday}
  =
    Unix.localtime (Unix.time ())
  in
    "It is "
    ^ string_of_int tm_hour
    ^ ":"
    ^ string_of_int tm_min
    ^ " on "
    ^ string_of_day tm_wday
    ^ " "
    ^ string_of_int tm_mday
    ^ " "
    ^ string_of_month tm_mon
    ^ " "
    ^ string_of_int (tm_year + 1900)
```

Note that the names bound by the pattern do not include the `Unix` prefix.

3

The construct **`type`** `t = {x : int ref}` is a record type containing a reference to an integer. We can build it using an existing reference, and can extract the reference for use elsewhere. We can share a single reference between two or more instances of this data type. The construct **`type`** `t = {`**`mutable`** `x : int}` is a record with a single, mutable field. We must use `<-` rather than `:=` to mutate it, and it may not be shared.

4

We can use multiple type parameters (which are written with parentheses and commas) and then use these types for the fields in the appropriate way:

```
type ('a, 'b, 'c) t =
  {a : 'a;
   b : 'a;
   c : 'b;
   d : 'b;
   e : 'c;
   f : 'c}
```

5

For the first part, we can use the function `Gc.stat` to return a **Gc.stat** record. Then we can write various components to the given file (here, we have chosen just a few):

```
write_gc_summary : string → unit

let write_gc_summary filename =
  let ch = open_out filename in
    let
      {Gc.minor_words;
       Gc.promoted_words;
       Gc.major_words;
       Gc.minor_collections;
       Gc.major_collections}
    =
      Gc.stat ()
    in
      output_string ch "Minor Words: ";
      output_string ch (string_of_float minor_words);
      output_string ch "\nPromoted Words: ";
      output_string ch (string_of_float promoted_words);
      output_string ch "\nMajor Words: ";
      output_string ch (string_of_float major_words);
      output_string ch "\nMinor Collections: ";
      output_string ch (string_of_int minor_collections);
      output_string ch "\nMajor Collections: ";
      output_string ch (string_of_int major_collections);
      close_out ch
```

For the second part, we define names for the magic numbers given in the documentation. We can then write a function `change_verbosity` which adds them up to make the new flags field, and uses `Gc.get` together with the **`with`** syntax to build a new record to be passed to `Gc.set`:

```
start_of_major : int
minor_collection : int
heap_grow_shrink : int
stack_resizing : int
heap_compaction : int
change_parameters : int
compute_slice_size : int
call_finalisation : int
bytecode_exe_search : int
change_verbosity : int list → unit

let start_of_major = 0x001
let minor_collection = 0x002
let heap_grow_shrink = 0x004
let stack_resizing = 0x008
let heap_compaction = 0x010
let change_parameters = 0x020
let compute_slice_size = 0x040
let call_finalisation = 0x080
let bytecode_exe_search = 0x100

let change_verbosity vs =
  let n = List.fold_left ( + ) 0 vs in
    Gc.set {(Gc.get ()) with Gc.verbose = n}
```

Chapter 4 (Generalized Input/Output)

1

This is a simple modification of `input_of_string` from the text, replacing string operators with array ones.

```
input_of_array : char array → input

let input_of_array a =
  let pos = ref 0 in
    {pos_in = (fun () -> !pos);
     seek_in =
       (fun p ->
          if p < 0 then raise (Invalid_argument "seek < 0");
            pos := p);
     input_char =
       (fun () ->
         if !pos > Array.length a - 1
           then raise End_of_file
           else (let c = a.(!pos) in pos := !pos + 1; c));
     in_channel_length = Array.length a}
```

Note that an array of characters like this requires much more space as a string.

2

We create a **Buffer.t**, attempt to read the specified number of characters, and then return the contents. If one of the calls to `i.input_char ()` raises the `End_of_file` exception, we return the buffer contents anyway – it will contain the characters read so far.

```
input_string : input → int → string

let input_string i n =
  let b = Buffer.create 100 in
    try
      for x = 0 to n - 1 do
        Buffer.add_char b (i.input_char ())
      done;
      Buffer.contents b
    with
      End_of_file -> Buffer.contents b
```

3

We add a function of the expected type to the `input`. The option `None` will represent the end of the file.

```
type input =
  {pos_in : unit -> int;
   seek_in : int -> unit;
   input_char : unit -> char;
   input_char_opt : unit -> char option;
   in_channel_length : int}
```

Now for a new `input_of_channel`. We try to build the `Some` option, returning `None` if the `End_of_file` exception is raised.

```
input_of_channel : in_channel → input

let input_of_channel ch =
  {pos_in = (fun () -> pos_in ch);
   seek_in = seek_in ch;
   input_char = (fun () -> input_char ch);
   input_char_opt =
     (fun () ->
        try Some (input_char ch) with End_of_file -> None);
   in_channel_length = in_channel_length ch}
```

The `input_of_string` function is another similar modification. This time, there is no need for exception handling.

```
input_of_string : string → input

let input_of_string s =
  let pos = ref 0 in
    {pos_in = (fun () -> !pos);
     seek_in =
       (fun p ->
         if p < 0 then raise (Invalid_argument "seek < 0");
         pos := p);
     input_char =
       (fun () ->
         if !pos > String.length s - 1
           then raise End_of_file
           else (let c = s.[!pos] in pos := !pos + 1; c));
     input_char_opt =
       (fun () ->
         if !pos > String.length s - 1
           then None
           else (let c = s.[!pos] in pos := !pos + 1; Some c));
     in_channel_length = String.length s}
```

4

We add the function of type **unit** → **int**.

```
type input =
  {pos_in : unit -> int;
   seek_in : int -> unit;
   input_char : unit -> char;
   input_byte : unit -> int;
   in_channel_length : int}
```

Now, having defined as a convenience, the name `no_more` for the -1, we can modify `input_of_channel` and `input_of_string` easily:

```
no_more : int
input_of_channel : in_channel → input
input_of_string : string → input

let no_more = (-1)

let input_of_channel ch =
  {pos_in = (fun () -> pos_in ch);
   seek_in = seek_in ch;
   input_char = (fun () -> input_char ch);
   input_byte =
     (fun () ->
       try int_of_char (input_char ch) with End_of_file -> no_more);
   in_channel_length = in_channel_length ch}

let input_of_string s =
  let pos = ref 0 in
    {pos_in = (fun () -> !pos);
     seek_in =
       (fun p ->
          if p < 0 then raise (Invalid_argument "seek < 0");
          pos := p);
     input_char =
       (fun () ->
          if !pos > String.length s - 1
            then raise End_of_file
            else (let c = s.[!pos] in pos := !pos + 1; c));
     input_byte =
       (fun () ->
          if !pos > String.length s - 1
            then no_more
            else
              (let c = s.[!pos] in pos := !pos + 1; int_of_char c));
     in_channel_length = String.length s}
```

These functions have none of the advantages of the exception-raising or option-returning ones, but they are very fast indeed.

5

We alter `input_of_channel` to check for a newline and raise `End_of_file` in that case.

```
single_line_input_of_channel : in_channel → input

let single_line_input_of_channel ch =
  {pos_in = (fun () -> pos_in ch);
   seek_in = seek_in ch;
   input_char =
     (fun () ->
       match input_char ch with '\n' -> raise End_of_file | c -> c);
   in_channel_length = in_channel_length ch}
```

Now we can create one of these special inputs from standard input, and use our `input_string` function to build a string from the user's input, ending when the return key is pressed. For example:

```
# input_string (single_line_input_of_channel stdin) max_int;;
Some input
- : string = "Some input"
```

6

Ideal functions to use when defining the new type already exist in the **Buffer** module. Then, we can build an example, where we give a name to the buffer, build an input from it and process it, retrieving its contents afterwards.

```
output_of_buffer : Buffer.t → output
build_buffer : unit → string

let output_of_buffer b =
  {output_char = Buffer.add_char b;
   out_channel_length = fun () -> Buffer.length b}

let build_buffer () =
  let b = Buffer.create 20 in
    let o = output_of_buffer b in
      o.output_char 'A';
      o.output_char 'B';
      o.output_char 'C';
      Buffer.contents b
```

Chapter 5 (Streams of Bits)

1

We can read quickly if the number of bits wanted is 8 and we happen to be at the beginning of a byte. In this case we can call the underlying output's `input_char` function directly.

```
getval_fast : input_bits → int → int

let getval_fast b n =
  if n = 8 && b.bit = 0
    then b.input.input_char ()
    else getval b n
```

If the conditions are not met, we fall back to the old `getval` function. This will be faster if we frequently read data aligned-byte-by-aligned-byte, but we still need the flexibility of a stream of bits when required.

2

We simply replace the **int** operators with those for **Int32.t**:

```
getval_32 : input_bits → int → Int32.t

let getval_32 b n =
  if n < 0 then raise (Invalid_argument "getval_32") else
    if n = 0 then 0l else
      let r = ref Int32.zero in
        for x = n - 1 downto 0 do
          let num = Int32.of_int (if getbit b then 1 else 0) in
            r := Int32.logor !r (Int32.shift_left num x)
        done;
        !r
```

3

Our precondition this time is that the number of bits to be written is 8 and the `obit` field is set to its initial value, 7. Then we can use the `output_char` function of the underlying `output`.

```
putval_fast : output_bits → int → int → unit

let putval_fast o v l =
  if l = 8 && o.obit = 7
    then o.output.output_char v
    else putval o v l
```

Otherwise, we fall back to the old `putval` function.

4

The integer functions are replaced by ones from the **Int32** module:

```
putval_32 : output_bits → Int32.t → int → unit

let putval_32 o v l =
  for x = l - 1 downto 0 do
    putbit o
      (Int32.to_int
        (Int32.logand v (Int32.shift_left (Int32.of_int x) 1)))
  done
```

5

We add a field `rewind` which will move the position in the output backwards one byte, if possible. This is the new type:

```
type output =
  {output_char : char -> unit;
   rewind : unit -> unit;
   out_channel_length : unit -> int}
```

Now we can rewrite, for example, `output_of_bytes` for this new type:

```
output_of_bytes : bytes → output

let output_of_bytes b =
  let pos = ref 0 in
    {output_char =
       (fun c ->
          if !pos < Bytes.length b
            then (Bytes.set b !pos c; pos := !pos + 1)
            else raise End_of_file);
     rewind =
       (fun () ->
          if !pos > 0
            then pos := !pos - 1
            else raise (Failure "rewind"));
     out_channel_length =
       (fun () -> Bytes.length b)}
```

We alter `putbit` appropriately:

```
putbit : output_bits → int → unit

let rec putbit o b =
  if o.obit = (-1) then
    begin
      o.obyte <- 0;
      o.obit <- 7;
      putbit o b
    end
  else
    begin
      if b <> 0 then o.obyte <- o.obyte lor (1 lsl o.obit);
      o.output.output_char (char_of_int o.obyte);
      o.output.rewind ();
      o.obit <- o.obit - 1
    end
```

Chapter 6 (Compressing Data)

1

We will need to look through the input list of integers (bytes), finding same and different runs and building a new list. For clarity, we will produce the output runs using a new type, producing the actual bytes later:

```
type run = Same of int * int | Diff of int list
```

The `Same` case holds a (length, value) pair and `Diff` just a list of bytes. Now we can write a function `get_same` which, given the first value, the current count, and the list, returns a pair of the final count of like characters, and the remaining list:

```
get_same : α → int → α list → int × α list

let rec get_same x n l =
  match l with
    h::t when h = x -> get_same x (n + 1) t
  | _ -> (n, l)
```

Similarly, we can define a function to read a different run into an accumulator, returning the run and the remaining list. This function will be called only when `get_same` returned a run of length one.

```
get_different : α → α list → α list × α list

let rec get_different a l =
  match l with
    [] -> (List.rev a, [])
  | h::t ->
      if a = [] then get_different [h] t
      else if h <> List.hd a then get_different (h :: a) t
      else (List.rev (List.tl a), List.hd a :: l)
```

Now we can write a function which uses both of these functions to get a single run, creating an instance of our new data type:

```
getrun : int list → run × int list

let getrun l =
  match l with
    [] -> raise (Invalid_argument "getrun")
  | h::_ ->
    match get_same h 0 l with
      1, _ ->
        let diff, rest = get_different [] l in (Diff diff, rest)
    | n, rest -> (Same (n, h), rest)
```

Now, we use the defined rules to build a function which makes a list of bytes from a run:

```
chars_of_run : run → int list

let chars_of_run r =
  match r with
    Same (length, c) -> [257 - length; c]
  | Diff chars -> List.length chars - 1 :: chars
```

With all this done, it is easy to define the compression function itself, which accumulates runs, concatenating them when all the data has been processed. We must be sure to add the EOD marker at the end.

```
compress_inner : run list → int list → int list
compress : int list → int list

let rec compress_inner a l =
  match l with
    [] -> List.concat (List.map chars_of_run (List.rev a))
  | _ ->
    let run, rest = getrun l in
      compress_inner (run :: a) rest

let compress l = compress_inner [] l @ [128]
```

Decompression is rather simpler, making use of our `take` and `drop` functions from the Util module described on page xiii. We read each run and expand it to a list of bytes, accumulating them until end of data.

```
decompress_inner : int list list → int list → int list
decompress : int list → int list

let rec decompress_inner a l =
  match l with
    [128] -> List.concat (List.rev a)
  | [] | [_] -> raise (Invalid_argument "decompress_inner")
  | h::t::t' ->
      if h < 127 then
        let bytes = Util.take (t :: t') (h + 1) in
        let rest = Util.drop (t :: t') (h + 1) in
          decompress_inner (bytes :: a) rest
      else if h > 128 then
        decompress_inner
          (Array.to_list (Array.make (257 - h) t) :: a) t'
      else decompress_inner a []

let decompress l = decompress_inner [] l
```

Note a little cheat – we use functions from the **Array** module to expand the `Same` run.

2

The tree carries no information in its branches (since no code can be a prefix of another code). Leaves can carry or not carry information, depending on whether there is a code there. In order to avoid **option** types, we can just split into three cases, `Lf` for empty leaves, `Code` for full ones, and `Br` for a branch, where 0 goes left and 1 goes right.

```
type tree = Lf | Code of int | Br of tree * tree
```

Now, we can define the function to add a code to an existing tree. For example, `add_elt Lf ([0; 1],`

`67)` adds the code `[0; 1]` for the run length 67. This will build the tree `Br (Br (Lf, Code 67), Lf)`. We match on the code, building the tree as we go left or right.

```
add_elt : tree → int list × int → tree

let rec add_elt tr (l, n) =
  match l with
    0::m ->
      begin match tr with
        Lf -> Br (add_elt Lf (m, n), Lf)
      | Br (left, right) -> Br (add_elt left (m, n), right)
      | Code x -> raise (Failure "collision")
      end
  | 1::m ->
      begin match tr with
        Lf -> Br (Lf, add_elt Lf (m, n))
      | Br (left, right) -> Br (left, add_elt right (m, n))
      | Code x -> raise (Failure "collision")
      end
  | [] -> Code n
  | _ -> raise (Failure "bad code")
```

Now, to build the whole tree, we just use repeated insertion with `fold_left`, having built the (code, run length) pairs:

```
make_tree : int list array → int list → tree

let make_tree arr numbers =
  List.fold_left
    add_elt
    Lf
    (List.combine (Array.to_list arr) numbers)
```

Now, for example, we can build the white terminating codes as a tree:

```
white_terminating_tree : tree

let white_terminating_tree =
  make_tree
    white_terminating_codes
    (Util.from 0 (Array.length white_terminating_codes - 1))
```

(`Util.from a b` gives the list of numbers starting at `a` and ending at `b` in ascending order). The function succeeds, verifying that there are no collisions, and yields:

```
Br
 (Br
   (Br
     (Br
       (Br
         (Br (Br (Lf, Br (Code 29, Code 30)),
           Br (Br (Code 45, Code 46), Code 22)),
         Br (Br (Code 23, Br (Code 47, Code 48)), Code 13)),
       Br
        (Br (Br (Code 20, Br (Code 33, Code 34)),
          Br (Br (Code 35, Code 36), Br (Code 37, Code 38))),
        Br (Br (Code 19, Br (Code 31, Code 32)), Code 1))),
     Br
      (Br (Br (Code 12, Br (Br (Code 53, Code 54), Code 26)),
        Br (Br (Br (Code 39, Code 40), Br (Code 41, Code 42)),
         Br (Br (Code 43, Code 44), Code 21))),
      Br
       (Br (Br (Code 28, Br (Code 61, Code 62)),
         Br (Br (Code 63, Code 0), Lf)),
       Code 10))),
   Br
    (Br
      (Br (Code 11,
        Br (Br (Code 27, Br (Code 59, Code 60)), Br (Lf, Code 18))),
       Br
        (Br (Br (Code 24, Br (Code 49, Code 50)),
          Br (Br (Code 51, Code 52), Code 25)),
        Br (Br (Br (Code 55, Code 56), Br (Code 57, Code 58)), Lf))),
     Br (Lf, Code 2))),
 Br
  (Br (Br (Code 3, Br (Lf, Code 8)),
    Br (Br (Code 9, Br (Code 16, Code 17)), Code 4)),
   Br (Br (Code 5, Br (Br (Code 14, Code 15), Lf)), Br (Code 6, Code 7))))
```

3

Compressing our data with `compress_string input_data 1680 1` instead of `compress_string input_data 80 21` generates a string of length 110 bytes rather than 120 bytes. This is because we could generate one run for the white section at the end of one line followed by a white section at the beginning of the next, instead of splitting at line boundaries.

If we try to re-compress this data, with `compress_string compressed 880 1`, the data size increases to 197 bytes. This is unsurprising, since the job of the white and black codes is to be information-dense and the compression algorithm works best on data which is information-sparse.

4

We can re-use the `read_up_to` function to build our histogram. Given white and black arrays, each of length 1792 and with elements initialized to zero, the input bits and the width and height, we can repeatedly call `read_up_to`. We must maintain a count of how many pixels are left to be read, and an

additional count of how many are left in this line, so the correct width can be passed to the `read_up_to` function.

```
build_histogram : int array → int array → input_bits → int → int → unit

let build_histogram a_white a_black i w h =
  let toread = ref (w * h) in
  let wleft = ref w in
    while !toread > 0 do
      let n, v = read_up_to (peekbit i) i 0 !wleft in
        let a = if v then a_black else a_white in
          a.(n) <- a.(n) + 1;
          toread := !toread - n;
          wleft := !wleft - n;
          if !wleft = 0 then wleft := w
    done
```

Now it is easy to build two histograms – one for white and one for black, and return them:

```
histogram_of_input : input → int → int → (int array × int array)

let histogram_of_input i w h =
  let white = Array.make 1792 0 in
    let black = Array.make 1792 0 in
      build_histogram white black (input_bits_of_input i) w h;
      (white, black)
```

We can define a simple function to print the histogram, eliding any zero counts:

```
print_histogram : int array → unit

let print_histogram =
  Array.iteri
    (fun x n ->
       if n > 0 then Printf.printf "%i runs of length %i\n" n x)
```

Here is the histogram for white runs on our example data:

```
# print_histogram white;;
5 runs of length 1
7 runs of length 2
15 runs of length 3
15 runs of length 4
34 runs of length 5
20 runs of length 6
4 runs of length 7
```

```
4 runs of length 8
4 runs of length 9
3 runs of length 10
3 runs of length 11
7 runs of length 12
1 runs of length 13
1 runs of length 14
1 runs of length 15
3 runs of length 16
1 runs of length 17
1 runs of length 20
1 runs of length 21
3 runs of length 35
1 runs of length 39
1 runs of length 46
1 runs of length 47
1 runs of length 73
3 runs of length 80
- : unit = ()
```

And here is the histogram for black runs:

```
# print_histogram black;;
12 runs of length 1
55 runs of length 2
38 runs of length 3
5 runs of length 4
2 runs of length 5
1 runs of length 6
1 runs of length 7
1 runs of length 8
2 runs of length 9
2 runs of length 10
- : unit = ()
```

Chapter 7 (Labelled and Optional Arguments)

1

If α is **int** then the first and second argument can be confused. We can fix this by adding labels and calling `Array.make`. Notice the use of punning here.

```
map : len:int → elt:α → α array

let make ~len ~elt =
  Array.make len elt
```

Now the function can be called without confusion:

```
        OCaml

# make ~len:5 ~elt:4;;
- : int array = [|4; 4; 4; 4; 4|]
```

Of course, it can still be called without labels.

2

We can define separate types for the start and length so that their names must be mentioned when calling the function.

```
fill : α array → start  → length → α → unit
filled : unit → string array

type start = Start of int

type length = Length of int

let fill a (Start s) (Length l) v =
  for x = s to s + l - 1 do a.(x) <- v done

let filled () =
  let a = Array.make 100 "x" in
    fill a (Start 20) (Length 40) "y";
    a
```

Not nearly as convenient as labels, though.

3

There are three functions where confusion may arise, and we can label them with simple wrappers. They are the functions where multiple arguments have the same type, and so may be confused.

```
sub : Buffer.t → off:int → len:int → string
blit : Buffer.t → srcoff:int → string → dstoff:int → len:int → unit
add_substring : Buffer.t → string → ofs:int → len:int → unit

let sub b ~off ~len =
  Buffer.sub b off len

let blit src ~srcoff dst ~dstoff ~len =
  Buffer.blit src srcoff dst dstoff len

let add_substring b s ~ofs ~len =
  Buffer.add_substring b s ofs len
```

4

We can make the accumulator an optional argument. Now the caller can call the function as if it were the same as `List.map`.

```
map : ?a:α list → (β → α) → β list → α list

let rec map ?(a = []) f l =
  match l with
    [] -> List.rev a
  | h::t -> map ~a:(f h :: a) f t
```

The optional argument must still appear in the interface, of course, so we might still prefer the old approach of wrapping it up and only exposing the wrapper.

Chapter 8 (Formatted Printing)

1

We can use `Printf.bprintf` to accumulate the individual parts, making sure to deal with the final element specially. The outer function sets everything up.

```
cycle_of_points_inner : Buffer.t → (int × int) list → string
cycle_of_points : (int × int) list → string

let rec cycle_of_points_inner b l =
  match l with
    [] ->
      Buffer.contents b
  | [(x, y)] ->
      Printf.bprintf b "(%i, %i)" x y;
      Buffer.contents b
  | (x, y)::t ->
      Printf.bprintf b "(%i, %i) --> " x y;
      cycle_of_points_inner b t

let cycle_of_points l =
  match l with
    [] -> ""
  | h::t ->
      cycle_of_points_inner (Buffer.create 256) (h :: t @ [h])
```

2

Again, `Printf.bprintf` is the key. This time, we can calculate the initial buffer size exactly.

```
hex_of_string : string → string

let hex_of_string s =
  let b = Buffer.create (String.length s * 2) in
    String.iter
      (fun c -> Printf.bprintf b "%02X" (int_of_char c))
      s;
    Buffer.contents b
```

We have used a width specifier of `2` and the `0` flag to make sure that characters with code $0..15$ are padded with a zero.

3

The format string for `Printf.printf` must be known at compile time. The solution for printing the result of `mkstring` using `printf` is the `%s` format specification:

```
Printf.printf "%s" (mkstring ())
```

4

The $*$ character can be used as a width or precision specifier, to indicate that the width or precision is given as an argument. We use $*$ for the width, and pass in 10.

```
Printf.sprintf "(%*i)" 10 1
```

So, the result is:

```
(         1)
```

We can use `List.iter` to print a table by applying this to each of a list of numbers in turn.

```
print_integers : int → int list → unit

let print_integers w ns =
  List.iter (Printf.printf "(%*i)" w) ns
```

Chapter 9 (Searching for Things)

1

For the first part, where all matches are considered, we can rewrite `search` with an extra argument to count the matches, restructuring its logic so as not to finish upon the first match.

```
string' : int → int → string → string → int
string : string → string → int

let rec search' matches n ss s =
  if String.length ss > String.length s - n then matches else
    if at ss 0 s n (String.length ss)
      then search' (matches + 1) (n + 1) ss s
      else search' matches (n + 1) ss s

let search = search' 0 0
```

There is no need to rewrite the `at` function. Now, for the version which considers only non-overlapping matches, we just jump by the length of the pattern `ss` upon a match:

```
string' : int → int → string → string → int
string : string → string → int

let rec search' matches n ss s =
  if String.length ss > String.length s - n then matches else
    if at ss 0 s n (String.length ss)
      then search' (matches + 1) (n + String.length ss) ss s
      else search' matches (n + 1) ss s

let search = search' 0 0
```

2

It is simple to write a function which returns the length of the longest matching prefix at the beginning of a list:

```
prefix : α list → α list → int

let rec prefix p l =
  match p, l with
    ph::pt, lh::lt -> if ph = lh then 1 + prefix pt lt else 0
  | _ -> 0
```

Now, we can write a function which keeps track of the position and length of the longest prefix found, returning them when the whole list has been searched.

```
longest_prefix_inner : int → int → int → α list → α list → int × int
longest_prefix : α list → α list → int × int

let rec longest_prefix_inner currpos bestpos bestlen p l =
  match l with
    [] -> (bestpos, bestlen)
  | h::t ->
      let prelen = prefix p l in
        if prelen > bestlen then
          longest_prefix_inner (currpos + 1) currpos prelen p t
        else
          longest_prefix_inner (currpos + 1) bestpos bestlen p t

let longest_prefix p l =
  longest_prefix_inner 0 0 0 p l
```

Here, `currpos` is the current position, `bestpos` the position of the longest matching prefix found so far, and `bestlen` the length of the longest prefix so far.

3

We can write a simple profiling function which, given a search function, measures its running time. This allows us to compare the naive and better versions of `search` we wrote for searching in strings:

```
profile : (string → string → α) → float

let profile f =
  let t = Unix.gettimeofday () in
    for x = 1 to 1_000_000 do
      ignore (f "ABA" "Somewhere in here is the pattern ABBBBAABA.")
    done;
    Unix.gettimeofday () -. t
```

Compiling with `ocamlc` on the Author's machine:

```
Naive version took 7.608291 seconds
Better version tool 3.388830 seconds
```

Now, compiling with `ocamlopt`:

```
Naive version took 2.966211 seconds
Better version tool 0.450546 seconds
```

4

We can add a case to the main search. We must check the character following the backslash for a match, assuming there is such a next character. If so, we move two positions in the pattern and one in the string. Otherwise, the match has failed.

```
| '\\'->
    if
      sp < String.length s &&
      ssp < String.length ss - 1 &&
      ss.[ssp + 1] = s.[sp]
    then
      Some (2, 1)
    else
      None
```

5

The Standard Library function `String.uppercase` can be used, in conjunction with the optional boolean argument:

```
search : ?nocase:bool → string → string → bool

let search ?(nocase = false) ss s =
  if nocase then
    search' 0 (String.uppercase ss) (String.uppercase s)
  else
    search' 0 ss s
```

Both the pattern and string must be upper case, of course.

Chapter 10 (Finding Permutations)

1

The combinations of a list can be generated by calculating the combinations of the tail. Then consider two possibilities – the head is included in this combination, or it is not:

```
combinations : α list → α list list

let rec combinations l =
  match l with
    [] -> [[]]
  | h::t ->
      let cs = combinations t in
        List.map (fun x -> h :: x) cs @ cs
```

Note the base case is the list containing the empty list, not just the empty list.

2

We build this from `perms` and the `combinations` function we just wrote:

```
permicombinations : α list → α list list

let permicombinations l =
  List.concat (List.map perms (combinations l))
```

We used the tail-recursive version of `perms`, of course.

3

This has roughly the same shape as `combinations`, with two differences: we keep a counter to make sure the computation ends, and we always add something to the list – either `true` or `false`.

```
bool_lists : int → bool list list

let rec bool_lists n =
  match n with
    0 -> [[]]
  | _ ->
     let ls = bool_lists (n - 1) in
       (List.map (fun l -> true :: l) ls) @
       (List.map (fun l -> false :: l) ls)
```

4

We repeatedly swap elements from opposite ends of the sub-array, given an array, offset and length:

```
array_rev : α array → int → int → unit

let array_rev a o l =
  for x = 0 to l / 2 - 1 do
    swap a (o + x) (o + l - x - 1)
  done
```

It is important to make sure it works for the empty range, an even-length sub-array and an odd-length sub-array. You could add detection of invalid arguments to this function.

5

The two functions `first` and `last` turn out to be even more awkward than their imperative counterparts. The function `first`, given a list, returns a tuple of three things: the elements before the "first" item, the first item itself, and those afterward. This is done by reversing the input list and looking for the first appropriate item, since this is easier than looking for the last appropriate item in the original input:

```
first_inner : α list → α list → (α list × α × α list)
first : α list → (α list × α × α list)

let rec first_inner before l =
  match l with
    [] -> raise (Invalid_argument "first_inner")
  | [x] -> (List.rev before, x, [])
  | a::b::t ->
      if b < a
        then (List.rev t, b, (a :: before))
        else first_inner (a :: before) (b :: t)

let first l =
  first_inner [] (List.rev l)
```

The `last` function is still more verbose: we locate the correct item by sorting and finding the smallest item greater than `f`. Then we can call `split_at` to return the item before and after the instance of `f`.

```
split_at_inner : α list → α → α list → α list × α list
split_at : α → α list → α list × α list
last : α → α list → α list × α × α list

let rec split_at_inner before n l =
  match l with
    [] -> (List.rev before, [])
  | h::t ->
      if h = n
        then (List.rev before, t)
        else split_at_inner (h :: before) n t

let split_at n l =
  split_at_inner [] n l

let last f l =
  match List.filter (fun x -> x > f) (List.sort compare l) with
    [] -> raise (Invalid_argument "last")
  | h::t ->
      let before, after = split_at h l in
        (before, h, after)
```

Now, the `next_permutation` function calls `first` and `last`, and stitches everything together:

```
next_permutation : α list → α list

let next_permutation l =
  let before_f, f, after_f = first l in
    let before_c, c, after_c = last f after_f in
      before_f @ [c] @ List.rev (before_c @ [f] @ after_c)
```

Here is the equivalent `non_increasing` function, which is simple:

```
non_increasing : α list → bool

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

The final `all_permutations` function is now easy.

```
all_permutations_inner : α list list → α list → α list list
all_permutations : α list → α list list

let rec all_permutations_inner a l =
  if non_increasing l then List.rev a else
    let next = next_permutation l in
      all_permutations_inner (next :: a) next

let all_permutations l =
  l :: all_permutations_inner [] l
```

Conclusion: converting an imperative algorithm mechanically to a functional style is not always useful.

Chapter 11 (Making Sets)

1

It is not possible to measure directly the memory used by an OCaml data structure (though one could calculate it by reading the section in the OCaml manual about data representation), but we can use the **Gc** module to measure the number of words allocated whilst building the structure, by using the data in `Gc.counters` before and after building each structure, and the given formula "memory used since start of program = minor words + major words - promoted words":

Set representation	Memory used (words)
List	150006
Tree	3911613
Red-Black tree	8193038
Hash table	265559

Notice that a huge amount more is required for the tree structures, because every time a new element is inserted, part of the tree is rewritten. In the case of the Red-Black tree, rotations involve allocating new memory too.

2

We can add the type for the `union` function to our signature:

```
module type SetType2 =
  sig
    type 'a t
    val set_of_list : 'a list -> 'a t
    val list_of_set : 'a t -> 'a list
    val insert : 'a -> 'a t -> 'a t
    val size : 'a t -> int
    val member : 'a -> 'a t -> bool
    val union : 'a t -> 'a t -> 'a t
  end
```

For lists, the `union` function is easy, we just insert each element of `b` into the list `a`. Duplicates will be removed correctly:

```
let union a b = List.fold_left (fun x y -> insert y x) a b
```

For trees and Red-Black trees, we must turn `b` into a list first, so `fold_left` can be used, but the solution is broadly the same.

```
let union a b = List.fold_left (fun x y -> insert y x) a (list_of_set b)
```

For hash tables, to preserve the previous tables, we must build lists from both sets, and then build a new set from the concatenation of those too lists:

```
let union a b = set_of_list (list_of_set a @ list_of_set b)
```

The built-in **Set** module, which is considered in Question 3, provides a particularly efficient union operation.

3

We write a version of our set signature which is specialized to integers. Then, we use the syntax given in the question to build the module `S`. This contains the type `S.t`, the value `S.empty` and the functions `S.elements`, `S.add`, `S.mem`, and `S.cardinal`, which we can use to write the functions to match our signature.

```
module IntSet :
  sig
    type t
    val set_of_list : int list -> t
    val list_of_set : t -> int list
    val insert : int -> t -> t
    val size : t -> int
    val member : int -> t -> bool
  end
=
  struct
    module S =
       Set.Make (struct type t = int let compare = compare end)

    type t = S.t

    let list_of_set s = S.elements s

    let set_of_list l = List.fold_right S.add l S.empty

    let member = S.mem

    let insert = S.add

    let size = S.cardinal
  end
```

Now the benchmarking for insertion and membership is simple:

```
benchmark_intset : string → int list → unit

let benchmark_intset name ns =
  let a = Unix.gettimeofday () in
    let set = IntSet.set_of_list ns in
      let b = Unix.gettimeofday () in
        List.iter (fun x -> ignore (IntSet.member x set)) ns;
        let c = Unix.gettimeofday () in
          Printf.printf
            "For %s, insertion took %f, membership %f\n"
            name (b -. a) (c -. b)

let _ =
  benchmark_intset "ordered" nums;
  benchmark_intset "unordered" rand
```

Here is the output:

```
For ordered, insertion took 0.056586, membership 0.019593
For unordered, insertion took 0.087148, membership 0.021295
```

4

We can change the type thus, with `BrR` for red and `BrB` for black:

```
type 'a t =
  Lf
| BrR of 'a t * 'a * 'a t
| BrB of 'a t * 'a * 'a t
```

Now, the solutions are tedious to write out, but not difficult. The result is Figure A.2.

Chapter 12 (Playing Games)

1

We already know that O wins 131184 times. By a similar use of `num_wins` we find that X wins 77904 times. So, as expected, going first is an advantage. We must write another function to find how many draws there are. A board is drawn if it is full but does not contain a winning configuration of either X or O:

```
drawn : tree → int

let rec drawn (Move (b, bs)) =
    (if
       empty b = [] &&
       not (won (List.map (( = ) O) b)) &&
       not (won (List.map (( = ) X) b))
     then 1 else 0)
  +
    List.fold_left ( + ) 0 (List.map drawn bs)
```

This tells us that there are 46080 drawn games. Since each game must be either won or drawn, the total number of possible games is $131184 + 77904 + 46080 = 255168$. We can check this by writing a function to find the terminal nodes directly:

```
terminals : tree → int

let rec terminals (Move (b, bs)) =
  (if bs = [] then 1 else 0) +
  List.fold_left ( + ) 0 (List.map terminals bs)
```

This gives 255168 too.

```
module SetRedBlack : sig include SetType end =
  struct
    type 'a t =
      Lf
    | BrR of 'a t * 'a * 'a t
    | BrB of 'a t * 'a * 'a t

    let rec list_of_set s =
      match s with
        Lf -> []
      | BrR (l, x, r) | BrB (l, x, r) ->
          x :: list_of_set l @ list_of_set r

    let balance t =
      match t with
        BrB (BrR (BrR (a, x, b), y, c), z, d)
      | BrB (BrR (a, x, BrR (b, y, c)), z, d)
      | BrB (a, x, BrR (BrR (b, y, c), z, d))
      | BrB (a, x, BrR (b, y, BrR (c, z, d))) ->
          BrR (BrB (a, x, b), y, BrB (c, z, d))
      | BrR (a, b, c) -> BrR (a, b, c)
      | BrB (a, b, c) -> BrB (a, b, c)
      | Lf -> Lf

    let rec insert_inner x s =
      match s with
        Lf -> BrR (Lf, x, Lf)
      | BrR (l, y, r) ->
          if x < y
            then balance (BrR (insert_inner x l, y, r))
            else if x > y then balance (BrR (l, y, insert_inner x r))
            else BrR (l, y, r)
      | BrB (l, y, r) ->
          if x < y
            then balance (BrB (insert_inner x l, y, r))
            else if x > y then balance (BrB (l, y, insert_inner x r))
            else BrB (l, y, r)

    let insert x s =
      match insert_inner x s with
        BrR (l, y, r) | BrB (l, y, r) -> BrB (l, y, r)
      | Lf -> assert false

    let rec set_of_list l =
      match l with
        [] -> Lf
      | h::t -> insert h (set_of_list t)

    let rec size s =
      match s with
        Lf -> 0
      | BrR (l, _, r) | BrB (l, _, r) -> 1 + size l + size r

    let rec member x s =
      match s with
        Lf -> false
      | BrR (l, y, r) | BrB (l, y, r) ->
          x = y || if x > y then member x r else member x l
  end
```

Figure A.2

2

We need make only two small changes. Delaying evaluation in the type…

```
type tree = Move of turn list * (unit -> tree list)
```

…and altering `next_moves` to insert that delay:

```
next_moves : turn → turn list → tree

let rec next_moves turn board =
  let next =
    fun () ->
      if
        won (List.map (( = ) O) board) ||
        won (List.map (( = ) X) board)
      then
        []
      else
        List.map
          (next_moves (flip_turn turn))
          (List.map (replace turn board) (empty board))
  in
    Move (board, next)
```

Now, we can carefully write a function `select_case` which, given a starting board such as `[E; E; E; E; O; E; E; E; E]` and the game tree, returns the portion of the game tree matching that board. Due to laziness, the rest of the tree is now not explored.

We can now alter `num_wins` easily for the delayed case, and write a function `pos_wins` which returns the number of wins starting from a position like `[E; E; E; E; O; E; E; E; E]`.

```
select_case : turn list → tree → tree
num_wins : turn → tree → int
pos_wins : turn → turn list → int

let select_case board (Move (_, f)) =
  match List.filter (fun (Move (b, _)) -> b = board) (f ()) with
    [Move (b, g)] -> g ()
  | _ -> raise (Failure "select_case")

let rec num_wins turn (Move (b, bs)) =
  (if won (List.map (( = ) turn) b) then 1 else 0) +
  List.fold_left ( + ) 0 (List.map (num_wins turn) (bs ()))

let pos_wins turn pos =
  List.fold_left ( + ) 0
    (List.map (num_wins turn) (select_case pos game_tree))
```

Similarly, we can modify the `drawn` function to work with the new lazy structure, and write a new function `draws` to count the drawn positions from a given starting board such as `[E; E; E; E; O; E; E; E; E]`:

```
drawn : tree → int
draws : turn list → int

let rec drawn (Move (b, bs)) =
    (if
       empty b = [] &&
       not (won (List.map (( = ) O) b)) &&
       not (won (List.map (( = ) X) b))
     then 1 else 0)
  +
    List.fold_left ( + ) 0 (List.map drawn (bs ()))

let draws pos =
  List.fold_left ( + ) 0
    (List.map drawn (select_case pos game_tree))
```

Now we can define starting boards for the centre spot, the middle of a side, and a corner. We can now use `pos_wins` and `draws`, taking account of symmetry to enumerate all the cases:

```
let centre = [E; E; E; E; O; E; E; E; E]

let side = [E; O; E; E; E; E; E; E; E]

let corner = [O; E; E; E; E; E; E; E; E]

let centre_x_wins = pos_wins X centre

let centre_o_wins = pos_wins O centre

let centre_drawn = draws centre

let side_x_wins = pos_wins X side * 4

let side_o_wins = pos_wins O side * 4

let side_drawn = draws side * 4

let corner_x_wins = pos_wins X corner * 4

let corner_o_wins = pos_wins O corner * 4

let corner_drawn = draws side * 4
```

This gives the following:

```
val centre_x_wins : int = 5616
val centre_o_wins : int = 15648
val centre_drawn : int = 4608
val side_x_wins : int = 40704
val side_o_wins : int = 56928
val side_drawn : int = 20736
val corner_x_wins : int = 31584
val corner_o_wins : int = 58608
val corner_drawn : int = 20736
```

The total is 255168, of course.

3

The strategy of using the magic square representation is to build a problem which is *isomorphic* (has the same essential characteristics – literally *the same shape*) to the original one, but is easier to work with. Before building the tree tree, we will need five little functions:

- `sum`, which checks if a list sums to 15;
- `threes`, which finds all the combinations of numbers from a list of numbers which are of length three (the `combinations` function is from Chapter 10);
- `won`, which uses `threes` to check if a list of integers contains a combination of three numbers which sum to 15;
- `drawn` which, given the integer lists for X and O, works out if the game has been drawn; and
- `possibles` which, given all the non-empty squares, lists the empty ones.

```
sum : int list → bool
threes : α list → α list list
won : int list → bool
drawn : α list → β list → bool
possibles : int list → int list

let sum l = List.fold_left ( + ) 0 l = 15

let threes l = List.filter (fun l -> length l = 3) (combinations l)

let won l = List.mem true (List.map sum (threes l))

let drawn l l' = length l + length l' = 9

let possibles all =
  List.filter
    (fun x -> not (List.mem x all))
    [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Now, the type contains a list of X positions, a list of O positions, and the list of child nodes:

```
type tree = Move of int list * int list * tree list
```

We do not need a type for the turn this time – we can just use a boolean. The function `next_moves` follows the usual pattern – if the game is won or drawn, the list of child nodes is empty. Otherwise, we build child nodes for each possible position the next player could place his piece.

```
next_moves : int list → int list → bool → tree
game_tree : tree

let rec next_moves xs os o_is_playing =
  let next =
    if won xs || won os || drawn xs os then [] else
      if o_is_playing
        then
          List.map
            (fun new_os -> next_moves xs new_os (not o_is_playing))
            (List.map (fun q -> q :: os) (possibles (xs @ os)))
        else
          List.map
            (fun new_xs -> next_moves new_xs os (not o_is_playing))
            (List.map (fun q -> q :: xs) (possibles (xs @ os)))
  in
    Move (xs, os, next)

let game_tree = next_moves [] [] true
```

The game tree is shown in Figure A.3. We can write a simple `xwins` function to test our new function returns the same result as the original.

```
xwins : tree → int

let rec xwins (Move (xs, os, cs)) =
  (if won xs then 1 else 0) +
  List.fold_left ( + ) 0 (List.map xwins cs)
```

Chapter 13 (Representing Documents)

1

Writing T for the trailer dictionary, we have the graph $\mathtt{T} \to 2 \to 3 \longleftrightarrow 1 \to 4$.

2

These can be written out by reference to the data structure:

```
val game_tree : tree =
  Move ([], [],
   [Move ([], [1],
     [Move ([2], [1],
       [Move ([2], [3; 1],
         [Move ([4; 2], [3; 1],
           [Move ([4; 2], [5; 3; 1],
             [Move ([6; 4; 2], [5; 3; 1],
               [Move ([6; 4; 2], [7; 5; 3; 1], []);
                Move ([6; 4; 2], [8; 5; 3; 1],
                 [Move ([7; 6; 4; 2], [8; 5; 3; 1], []);
                  Move ([9; 6; 4; 2], [8; 5; 3; 1], [])]);
                Move ([6; 4; 2], [9; 5; 3; 1], [])]);
              Move ([7; 4; 2], [5; 3; 1],
               [Move ([7; 4; 2], [6; 5; 3; 1],
                 [Move ([8; 7; 4; 2], [6; 5; 3; 1],
                   [Move ([8; 7; 4; 2], [9; 6; 5; 3; 1], [])]);
                  Move ([9; 7; 4; 2], [6; 5; 3; 1], [])]);
                Move ([7; 4; 2], [8; 5; 3; 1],
                 [Move ([6; 7; 4; 2], [8; 5; 3; 1], []);
                  Move ([9; 7; 4; 2], [8; 5; 3; 1], [])]);
                Move ([7; 4; 2], [9; 5; 3; 1], [])]);
              Move ([8; 4; 2], [5; 3; 1],
               [Move ([8; 4; 2], [6; 5; 3; 1],
                 [Move ([7; 8; 4; 2], [6; 5; 3; 1],
                   [Move ([7; 8; 4; 2], [9; 6; 5; 3; 1], [])]);
                  Move ([9; 8; 4; 2], [6; 5; 3; 1], [])]);
                Move ([8; 4; 2], [7; 5; 3; 1], []);
                Move ([8; 4; 2], [9; 5; 3; 1], [])]);
              Move ([9; 4; 2], [5; 3; 1], [])]);
            ...]);
          ...]);
        ...]);
      ...]);
    ...])
```

Figure A.3

- `Name "/Name"`
- `String "Quartz Crystal"`
- `Dictionary [("/Type", Name "/ObjStm"); ("/N", Integer 100); ("/First", Integer 807); ("/Last", Integer 1836); ("/Filter", Name "/FlateDecode")]`
- `Array [Integer 1; Integer 2; Float 1.5; String "black"]`
- `Array [Integer 1; Indirect 2]`

In the last two examples, we assumed integer where appropriate. Notice that in the last example, the parsing is somewhat ambiguous – we would need to read all the way to the `R` to be sure it was not an array of several integers.

3

Consider the tree `Br (Br (Lf, 1, Lf), 2, Br (Lf, 3, Lf))`. We will represent it by using nested PDF dictionaries. A branch will have a `/Type` key with value `/Br`. It will have `/Left` and `/Right` entries for the sub-trees. A leaf is indicated simply by `/Lf`. In the PDF file we would write it like this:

```
<</Type /Br
  /Value 2
  /Left <</Type /Br /Value 1 /Left /Lf /Right /Lf>>
  /Right <</Type /Br /Value 3 /Left /Lf /Right /Lf>>>>
```

To construct it from our data type in OCaml:

```
let tree =
  Pdf.Dictionary
    [("/Type", Pdf.Name "/Br");
     ("/Value", Pdf.Integer 2);
     ("/Left",
        Pdf.Dictionary
          [("/Type", Pdf.Name "/Br");
           ("/Value", Pdf.Integer 1);
           ("/Left", Pdf.Name "/Lf");
           ("/Right", Pdf.Name "/Lf")]);
     ("/Right",
         Pdf.Dictionary
           [("/Type", Pdf.Name "/Br");
            ("/Value", Pdf.Integer 3);
            ("/Left", Pdf.Name "/Lf");
            ("/Right", Pdf.Name "/Lf")])]
```

4

We must search for dictionary entries inside `Pdf.Dictionary`, of course, but also inside `Pdf.Stream`, which contains a dictionary, and `Pdf.Array` which may do so too. Two mutually-recursive functions will do:

```
rotate_90 : Pdf.pdfobject → Pdf.pdfobject
rotate_90_dict : string × Pdf.pdfobject → string × Pdf.pdfobject

let rec rotate_90 obj =
  match obj with
    Pdf.Array objs ->
      Pdf.Array (List.map rotate_90 objs)
  | Pdf.Dictionary objs ->
      Pdf.Dictionary (List.map rotate_90_dict objs)
  | Pdf.Stream (dict, str) ->
      Pdf.Stream (rotate_90 dict, str)
  | x -> x

and rotate_90_dict (k, v) =
  match k with
    "/Rotate" -> ("/Rotate", Pdf.Integer 90)
  | _ -> (k, rotate_90 v)
```

Chapter 14 (Writing Documents)

1

We can alter the functions `string_of_array` and `string_of_dictionary` to simply not output the space preceding the closing bracket. To remove the initial space requires a little trick. We inspect the length of the buffer to determine if we are about to output the first item. In that case, no space is written. This is shown in Figure A.4.

2

The full code can be found in the online resources. We make the following changes:

- Change `/Count` from `1` to `3`.
- Move the `/Resources` to its own object, number `5`.
- Write two new pages, objects `6` and `7`.
- Write two new page content streams, `8` and `9`.
- Change the `/Size` from `5` to `10` to account for the new objects.

3

The full code can be found in the online resources. See the hint for a little more help.

Chapter 15 (Pretty Pictures)

1

Since we are working with circles, let us define π:

```
string_of_array : Pdf.pdfobject list → string
string_of_dictionary : (string ×Pdf.pdfobject) list → string

let rec string_of_array a =
  let b = Buffer.create 100 in
    Buffer.add_string b "[";
    List.iter
      (fun s ->
        if Buffer.length b > 1 then Buffer.add_char b ' ';
        Buffer.add_string b (string_of_pdfobject s))
      a;
    Buffer.add_string b "]";
    Buffer.contents b

and string_of_dictionary d =
  let b = Buffer.create 100 in
    Buffer.add_string b "<<";
    List.iter
      (fun (k, v) ->
        if Buffer.length b > 2 then Buffer.add_char b '';
        Buffer.add_string b k;
        Buffer.add_char b ' ';
        Buffer.add_string b (string_of_pdfobject v))
      d;
    Buffer.add_string b ">>";
    Buffer.contents b
```

Figure A.4

```
pi : float

let pi = 4. *. atan 1.
```

We can write a simple function to return the point at a given angle, distance `r` from point (`x`, `y`):

```
point : float → float → float → float → float × float

let point x y r angle =
  (cos angle *. r +. x, sin angle *. r +. y)
```

Now, we can build a list of all these points, remembering to stop before we have been around the whole circle. The argument `step` is the angle between successive points.

```
points : float → float → float → float → (float × float) list

let points x y r step =
  let n = ref 0
  and points = ref [] in
    while float_of_int !n *. step < 2. *. r *. pi do
      points := point x y r (float_of_int !n *. step) :: !points;
      n := !n + 1
    done;
    !points
```

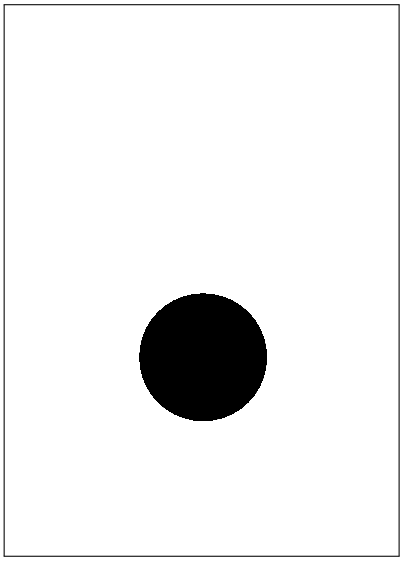
Finally, we generate a `Move` to the first points, `Line`s to the rest, and a final `Close`.

```
circle : float → float → float → Pdfpage.t list
circle_filled : Pdfpage.t list

let circle x y r =
  match points x y r (pi /. 20.) with
    (x, y)::lines ->
      Pdfpage.Move (x, y) ::
      List.map (fun (x, y) -> Pdfpage.Line (x, y)) lines @
      [Pdfpage.Close]
  | _ ->
      assert false          the points function should never return an empty list

let circle_filled =
  circle 300. 300. 100. @ [Pdfpage.Fill]
```

For our example, we made a filled circle of radius 100 centred at $(300, 300)$ with `Pdfpage.Fill`:



2

First, a function to build a pseudo-random circle somewhere on our page, making sure to overlap the edges:

```
random_circle : unit → Pdfpage.t list

let random_circle () =
  let x = Random.float 700. -. 50.
  and y = Random.float 1000. -. 50.
  and r = Random.float 100. +. 20. in
    circle x y r
```

Now, a little utility function to build a list of n things when given a function which generates them, such as `random_circle`:

```
many : (unit → α) → int → α list

let rec many f n =
  match n with
    0 -> []
  | _ -> f () :: many f (n - 1)
```

Now it is simple to build a hundred random grey filled circles and append them all together with `List.concat`:

```
many_circles : Pdfpage.t list

let many_circles =
  List.concat
    (List.map
      (fun l ->
        List.append
          (Pdfpage.FillColour (Random.float 1.) :: l)
          [Pdfpage.Fill])
      (many random_circle 100))
```

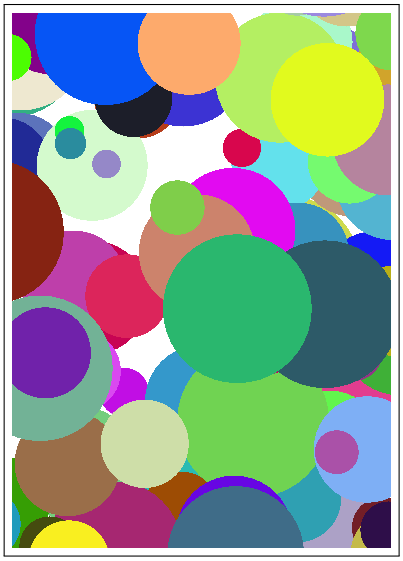
3

We can add `FillColourRGB of float * float * float` and `StrokeColourRGB of float * float * float` to the data type and associated functions, and then modify our previous example:

```
many_circles_colour : Pdfpage.t list

let many_circles_colour =
  List.concat
    (List.map
      (fun l ->
         List.append
           (Pdfpage.FillColourRGB
              (Random.float 1., Random.float 1., Random.float 1.)
               :: l)
           [Pdfpage.Fill])
      (many random_circle 100))
```

If you are reading the PDF ebook version of this book, the following image is in colour:



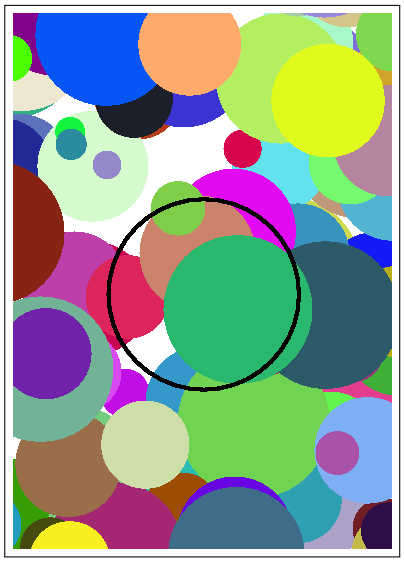
4

We add `LineWidth of float` to the data type and associated functions. Now we set the line width and stroke colour and draw the large circle:

```
big_circle : Pdfpage.t
circle_over_circles : Pdfpage.t

let big_circle =
    [Pdfpage.LineWidth 5.; Pdfpage.StrokeColour 0.]
  @ circle 300. 400. 150.
  @ [Pdfpage.Stroke]

let circle_over_circles =
  many_circles_colour @ big_circle
```



5

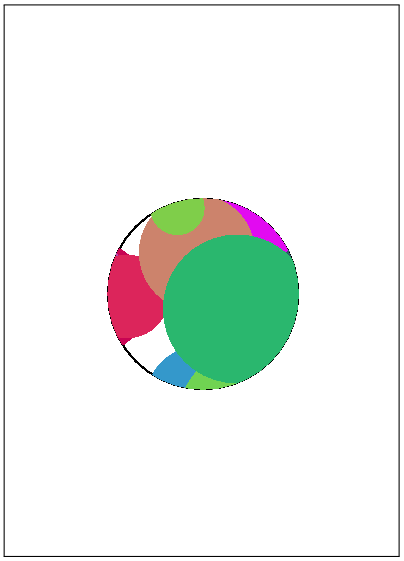
We add `SetClip` to the data type and associated functions. Then we set the clip before stroking.

```
big_clipping_circle : Pdfpage.t list
clipped : Pdfpage.t list

let big_clipping_circle =
  [Pdfpage.LineWidth 1.; Pdfpage.StrokeColour 0.] @
  circle 300. 400. 150. @
  [Pdfpage.SetClip; Pdfpage.Stroke]

let clipped =
  big_clipping_circle @ many_circles_colour
```

The single path is used both for clipping and for the stroke:



Chapter 16 (Adding Text)

1

We can pull out font size, line spacing and margin easily, defining them at the top of the page. The text width is derived from the page width and margin. The maximum number of characters in a line can be calculated using the formula given in the question:

```
let font_size = 10.0

let line_spacing = 1.1

let margin = 40.0

let text_width = page_width -. margin -. margin

let max_chars = int_of_float (text_width /. font_size *. (5. /. 3))
```

We can now insert these new names into `typeset_line_at`:

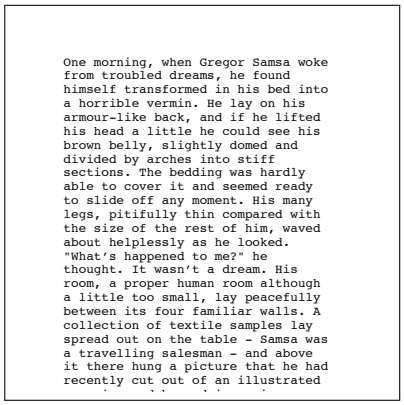
```
Pdfpage.SetTextPosition (margin, y);
Pdfpage.SetFontAndSize ("/F0", font_size);
```

We pass `max_chars` to `clean_lines` and alter our call to `downfrom`:

```
let ls = clean_lines (lines max_chars words) in
```

```
downfrom
  (font_size *. line_spacing)
  (page_height -. margin -. line_spacing) (List.length ls) 0
```

Here is an example of the new program, with text typeset on a much smaller page (it runs off the bottom, of course):



2

We can implement this by manually inserting space characters into the buffer in `lines_inner` following every newline (i.e. at the beginning of the second and subsequent paragraphs). This is shown in Figure A.5. Notice the use of an optional argument in `lines`. We can now alter one line in `typeset_page`:

```
let ls = clean_lines (lines max_chars ~indent:8 words) in
```

The result is shown in Figure A.6.

3

The page height can be calculated easily:

```
let text_height = page_height -. margin -. margin
```

Now, we can use `Pdfpage.SetCharacterSpacing` in `typeset_line_at`, adding an extra argument for the spacing:

```
lines_inner : Pdfpage.t list
lines : float → ?indent:int → string list → string list list

let rec lines_inner ls b width indent words =
  match words with
    [] ->
      if Buffer.length b > 0 then
        List.rev (Partial (Buffer.contents b) :: ls)
      else
        List.rev ls
  | "\n"::t ->
      let b' = Buffer.create width in
        for x = 1 to indent do Buffer.add_char b' ' ' done;
        lines_inner
          (Partial (Buffer.contents b) :: ls) b' width indent t
  | word::t ->
      if Buffer.length b = 0 && String.length word > width then
        lines_inner
          (Full word :: ls) (Buffer.create width) width indent t
      else if String.length word + Buffer.length b < width then
        begin
          Buffer.add_string b word;
          if Buffer.length b < width then Buffer.add_char b ' ';
          lines_inner ls b width indent t
        end
      else
        lines_inner
          (Full (Buffer.contents b) :: ls)
          (Buffer.create width) width indent (word :: t)

let lines width ?(indent=0) words =
  lines_inner [] (Buffer.create width) width indent words
```

Figure A.5

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked.

"What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer.

Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before.

"Oh, God", he thought, "what a strenuous career it is that I've chosen! Travelling day in and day out. Doing business like this takes much more effort than doing your own business at home, and on top of that there's the curse of travelling, worries about making train connections, bad and irregular food, contact with different people all the time so that you can never get to know anyone or become friendly with them. It can all go to Hell!" He felt a slight itch up on his belly; pushed himself slowly up on his back towards the headboard so that he could lift his head better; found where the itch was, and saw that it was covered with lots of little white spots which he didn't know what to make of; and when he tried to feel the place with one of his legs he drew it quickly back because as soon as he touched it he was overcome by a cold shudder.

He slid back into his former position. "Getting up early all the time", he thought, "it makes you stupid. You've got to get enough sleep. Other travelling salesmen live a life of luxury. For instance, whenever I go back to the guest house during the morning to copy out the contract, these gentlemen are always still sitting there eating their breakfasts. I ought to just try that with my boss; I'd get kicked out on the spot. But who knows, maybe that would be the best thing for me. If I didn't have my parents to think about I'd have given in my notice a long time ago, I'd have gone up to the boss and told him just what I think, tell him everything I would, let him know just what I feel. He'd fall right off his desk! And it's a funny sort of business to be sitting up there at your desk, talking down at your subordinates from up there, especially when you have to go right up close because the boss is hard of hearing. Well, there's still some hope; once I've got the money together to pay off my parents' debt to him - another five or six years I suppose - that's definitely what I'll do. That's when I'll make the big change. First of all though, I've got to get up, my train leaves at five."

Figure A.6

```
typeset_line_at : float → string → float → Pdfpage.t list

let typeset_line_at spacing line y =
  [Pdfpage.BeginText;
   Pdfpage.SetCharacterSpacing spacing;
   Pdfpage.SetTextPosition (margin, y);
   Pdfpage.SetFontAndSize ("/F0", font_size);
   Pdfpage.ShowText line;
   Pdfpage.EndText]
```

The spacing is calculated as prescribed, only for `Full` lines:

```
calculate_spacing : float → string → float

let calculate_spacing width line =
  match line with
    Full s ->
      float (width - String.length s) /.
      float (String.length s - 1) *. font_size *. (3. /. 5.)
  | Partial s -> 0.
```

In a revised `typeset_page`, we can calculate the correct spacings, passing them to `typeset_line_at`:

```
typeset_page : string → Pdfpage.t list

let typeset_page text =
  let words = words_of_input (Input.input_of_string text) in
  let ls = lines max_chars ~indent:8 words in
  let spacings = List.map (calculate_spacing max_chars) ls in
  let positions =
    downfrom
      (font_size *. line_spacing)
      (page_height -. margin -. line_spacing) (List.length ls) 0
  in
    List.concat
      (List.map3
         typeset_line_at spacings (clean_lines ls) positions)
```

Here is an example page. You can see that the lines are fully justified – flush to the left and right margins.

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked.

"What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer.

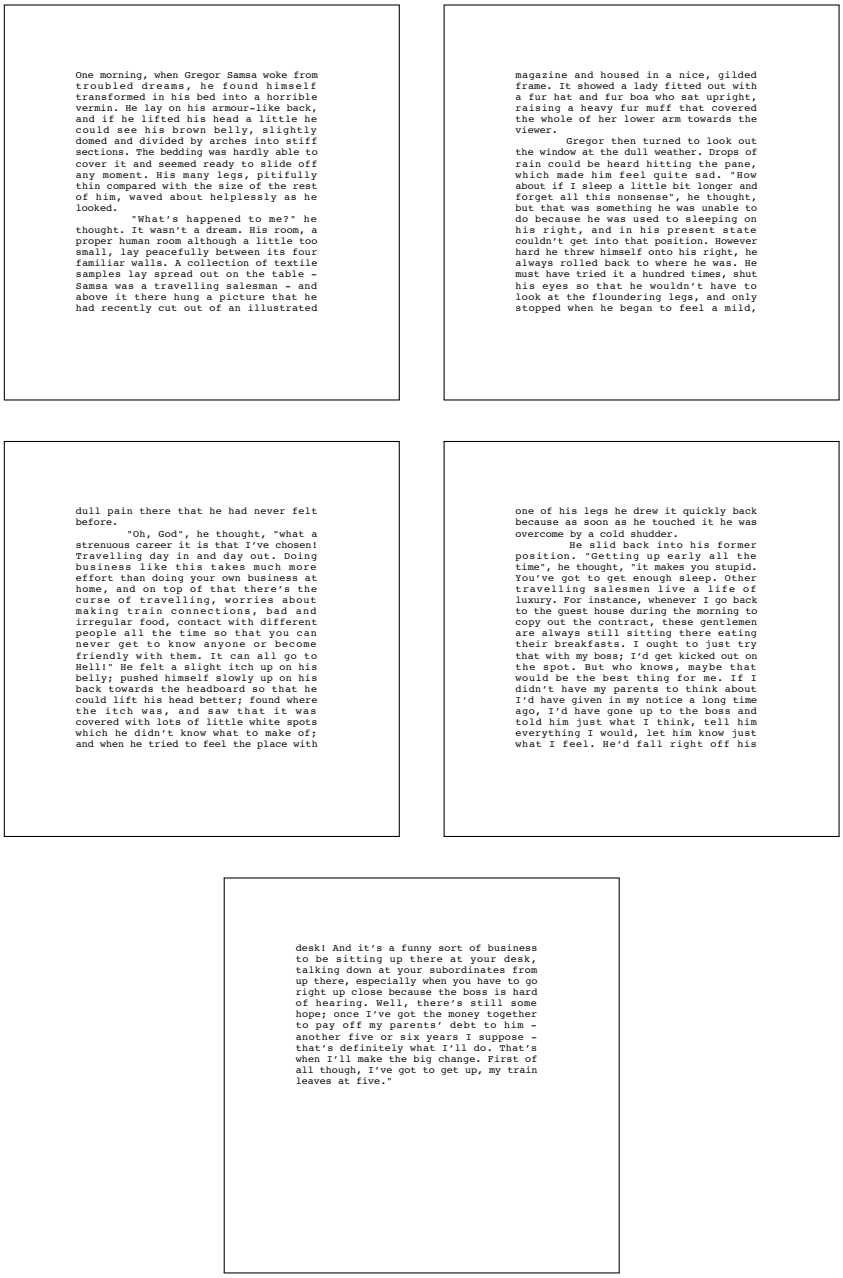
Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before.

"Oh, God", he thought, "what a strenuous career it is that I've chosen! Travelling day in and day out. Doing business like this takes much more effort than doing your own business at home, and on top of that there's the curse of travelling, worries about making train connections, bad and irregular food, contact with different people all the time so that you can never get to know anyone or become friendly with them. It can all go to Hell!" He felt a slight itch up on his belly; pushed himself slowly up on his back towards the headboard so that he could lift his head better; found where the itch was, and saw that it was covered with lots of little white spots which he didn't know what to make of; and when he tried to feel the place with one of his legs he drew it quickly back because as soon as he touched it he was overcome by a cold shudder.

He slid back into his former position. "Getting up early all the time", he thought, "it makes you stupid. You've got to get enough sleep. Other travelling salesmen live a life of luxury. For instance, whenever I go back to the guest house during the morning to copy out the contract, these gentlemen are always still sitting there eating their breakfasts. I ought to just try that with my boss; I'd get kicked out on the spot. But who knows, maybe that would be the best thing for me. If I didn't have my parents to think about I'd have given in my notice a long time ago, I'd have gone up to the boss and told him just what I think, tell him everything I would, let him know just what I feel. He'd fall right off his desk! And it's a funny sort of business to be sitting up there at your desk, talking down at your subordinates from up there, especially when you have to go right up close because the boss is hard of hearing. Well, there's still some hope; once I've got the money together to pay off my parents' debt to him - another five or six years I suppose - that's definitely what I'll do. That's when I'll make the big change. First of all though, I've got to get up, my train leaves at five."

4

The answer to this question is too long to be contained in the text. Consult the online resources for the program itself. An example multi-page output is shown in Figure A.7.


One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked.

"What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated

magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer.

Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild,

dull pain there that he had never felt before.

"Oh, God", he thought, "what a strenuous career it is that I've chosen! Travelling day in and day out. Doing business like this takes much more effort than doing your own business at home, and on top of that there's the curse of travelling, worries about making train connections, bad and irregular food, contact with different people all the time so that you can never get to know anyone or become friendly with them. It can all go to Hell!" He felt a slight itch up on his belly; pushed himself slowly up on his back towards the headboard so that he could lift his head better; found where the itch was, and saw that it was covered with lots of little white spots which he didn't know what to make of; and when he tried to feel the place with

one of his legs he drew it quickly back because as soon as he touched it he was overcome by a cold shudder.

He slid back into his former position. "Getting up early all the time", he thought, "it makes you stupid. You've got to get enough sleep. Other travelling salesmen live a life of luxury. For instance, whenever I go back to the guest house during the morning to copy out the contract, these gentlemen are always still sitting there eating their breakfasts. I ought to just try that with my boss; I'd get kicked out on the spot. But who knows, maybe that would be the best thing for me. If I didn't have my parents to think about I'd have given in my notice a long time ago, I'd have gone up to the boss and told him just what I think, tell him everything I would, let him know just what I feel. He'd fall right off his

desk! And it's a funny sort of business to be sitting up there at your desk, talking down at your subordinates from up there, especially when you have to go right up close because the boss is hard of hearing. Well, there's still some hope; once I've got the money together to pay off my parents' debt to him - another five or six years I suppose - that's definitely what I'll do. That's when I'll make the big change. First of all though, I've got to get up, my train leaves at five."

Figure A.7

Hints for Questions

Chapter 1
Unravelling "Fold"

1

What must the accumulator start at? What operation must we perform each time? Consider the `sum` function from the text as a starting point.

2

We need to ignore each element, just incrementing an accumulator when we see each one. The final value of the accumulator will then be the length of the list. What is the initial accumulator?

3

A good way to deal with the fact that an empty list has no last element would be to return an **option** type.

4

These sorts of folds producing lists have an accumulator which is a list type – you might start at the empty list.

5

The type will be the same as for `List.mem`. How do we keep track of whether or not we have seen a matching element? What is the initial accumulator?

6

This will involve a string accumulator and the string concatenation operator. How to we ensure there is not an excess space before or after the sentence?

7

The built-in `max` operator returns the larger of two values.

8

You can use the Standard Library function `Unix.gettimeofday` to help time the functions.

Chapter 2
Being Lazy

1

This is very similar to `lseq` in the text.

2

Construct by analogy to the same function on ordinary lists. Remember one needs to force evaluation to get at the tail.

3

How do we know when we have reached the end of the input list and must start again? What do we do if the input list is empty?

4

Consider the definition of fibonacci numbers. Split into a function which builds the lazy list given two numbers, and the construction of the list itself.

5

The type of the function will be α lazylist $\rightarrow \alpha$ lazylist $\times \alpha$ lazylist. You will need to force the evaluation of the tail of the input list twice to yield the heads of the two output lists, and then work out how to produce the tails.

6

Write a function which builds the alphabetic string from a number. Then, some of the generic lazy list handling functions from the text can be used to build the lazy list itself.

Chapter 3
Named Tuples with Records

1

What is a reference really?

2

Look at the functions `Unix.time` and `Unix.localtime`. You will need to deal with conversion of day-of-week and month from integers to strings yourself, but this is not difficult.

4

Records types can be parametrized, just like other data types.

5

See the documentation for a) `Gc.stat` and b) `Gc.set` and `Gc.get`.

Chapter 4
Generalized Input/Output

1

A simple modification of `input_of_string`.

2

Use a **Buffer.t** to accumulate the characters.

3

The new field will have type **unit** $\rightarrow$ **char option**.

4

The new field will have type **unit** $\rightarrow$ **int**. For `input_of_channel` and `input_of_string`, use exception handling to return the special value. Consider giving the special value -1 a name.

5

Check for a newline character on each `input_char`, raising `End_of_file`. Can you use `input_string` with the new channel you have created to return the user input?

6

The **Buffer** module already contains ideal functions for this.

Chapter 5
Streams of Bits

1

What is the test for being able to read a whole byte at a time? What do we do if the condition is met? If it is not?

2

Just replace the integer functions with ones from the **Int32** module.

3

What is the test for being able to write a whole byte at a time? What do we do if the condition is met? If it is not?

4

Just replace the integer functions with ones from the **Int32** module.

5

Try adding a `rewind` method to your `output` type, which goes back one byte. Now extend `output_of_bytes` appropriately.

Chapter 6
Compressing Data

1

Use list processing functions over a list of integers representing the data. For testing purposes, re-use our `int_list_of_string` and `string_of_int_list` functions.

2

Carefully define a suitable data type for the tree – the data will all be at the fringes of the tree, since no code is a prefix of another. Write a function to add a code to the tree. Now, repeated insertion can be used to build the whole tree.

4

We can re-use the `read_up_to` function, building up two histograms – one for white runs and one for black runs.

Chapter 7
Labelled and Optional Arguments

1

Consider what happens if α is **int**.

2

Perhaps define one or more new types.

3

Consider which functions have two or more arguments of the same type. We can add labels to those arguments only and wrap the function up.

4

We can use an optional argument for the accumulator.

Chapter 8
Formatted Printing

1

Consider `Printf.bprintf` as a way to accumulate the parts. What is special about the last element? What if there is no last element?

2

There is a format specifier for hexadecimal numbers. Consider what happens with very small character codes.

Chapter 9
Searching for Things

1

The `at` function from the chapter may be re-used here, in both parts of the question.

2

Start by writing a function which gives the length of the longest prefix of a pattern at the beginning of a list. Now wrap it in another which works through the whole list.

3

The expression `Unix.gettimeofday ()` gives a floating-point number representing the current time.

4

Add a case to the inner match. It must manually check the next letter in the pattern matches the next letter in the string (if it exists).

5

This does not require altering the search code itself – one can just preprocess the pattern and string. There is a function `String.uppercase` in the Standard Library.

Chapter 10
Finding Permutations

1

What is the base case? For the main case, first calculate the combinations of the tail by recursion. What can you do now?

2

You can build this from functions we have already written.

3

What is the base case? You will need to keep a counter to make sure we only generate lists of the given length.

4

Consider using the `swap` function we defined in the main text.

5

Instead of using indices for the "first" and "last", use the values themselves. Pull the list representing the old permutation apart, and rearrange it until you have the next.

Chapter 11
Making Sets

1

In the documentation for the **Gc** module, you will find the function `Gc.counters` and a description of how to calculate the total amount of memory allocated since the program began.

2

Add a `union` function to the signature, with the type α `t` $\rightarrow$ α `t` $\rightarrow$ α `t` and then a `union` function to the struct for each set representation.

3

You can include the given code inside the structure of the new module. Now, you can set the type `t` to be equal to `S.t`, the type of these new sets.

4

Alter the type first. The changes to each function then follow relatively easily.

Chapter 12
Playing Games

1

We already have code to find the number of X wins. What are the conditions for a game being drawn?

2

It is only necessary to alter the type `tree` and the function `next_moves`, and only a little. Now we need a function to extract the part of the tree which begins with O in the centre slot. Once we have this, simple modifications of our functions for counting wins and draws will do the job.

3

There is no need to worry about the order of the numbers in the square – the tree will have the same essential characteristics regardless.

Chapter 13
Representing Documents

3

A structure built from nested dictionaries would be suitable.

4

Consider everywhere a dictionary entry might appear. The function or functions will be recursive, following the general pattern of the data structure.

Chapter 14
Writing Documents

1

Removing the space at the end is simple. How can we detect if we are about to output the first item, and thus do not want a space?

2

Remember to alter the `/Size` entry, and to change `/Indirect` entries as required to reflect the new structure.

3

For the content stream in the "Hello, World!" file, one long row is sufficient. However, when encoding other example content streams, consult the PDF specification to ensure the file is still valid. To test, open in your favourite PDF reader.

Chapter 15
Pretty Pictures

1

We need to generate a set of points on the circle, and then they can be built into a path with `Move`, `Line`, and `Close`. This path can then be used with `Fill` to build the final page content.

2

First write a function to generate a path for a pseudo-random circle using a function we have previously written. Now `FillColour` and `Fill` can be added for each one, and the final list of operators produced.

3

We add `FillColourRGB` and `StrokeColourRGB` items to the data type and its associated functions. Then we can substitute them into the previous answer.

Chapter 16
Adding Text

1

You need to alter `typeset_line_at`, `clean_lines`, and `downfrom` once you have defined the new values.

2

Try just adding eight space characters after each new line in `lines_inner`.

3

If there are n characters, we must insert $n - 1$ pieces of space between them, each of equal size.

4

To make the construction of the multipage document simpler, consider choosing carefully the order (and hence numbering) of the objects in the file.

Coping with Errors

It is very hard to write even small programs correctly the first time. An unfortunate but inevitable part of programming is the location and fixing of mistakes. OCaml has a range of messages to help you with this process.

Here are descriptions of the common messages OCaml prints when a program cannot be accepted or when running it causes a problem (a so-called "run-time error"). We also describe warnings OCaml prints to alert the programmer to a program which, though it can be accepted for evaluation, might contain mistakes.

ERRORS

These are messages printed when an expression could not be accepted for evaluation, due to being malformed in some way. No evaluation is attempted. You must fix the expression and try again.

Syntax error

This error occurs when OCaml finds that the program text contains things which are not valid words (such as **`if`**, **`let`** etc.) or other basic parts of the language, or when they exist in invalid combinations – this is known as *syntax*. Check carefully and try again.

```
        OCaml

#1 +;;
Error: syntax error
```

OCaml has underlined where it thinks the error is. Since this error occurs for a wide range of different mistakes and problems, the underlining may not pinpoint the exact position of your mistake.

Unbound value ...

This error occurs when you have mentioned a name which has not been defined (technically "bound to a value"). This might happen if you have mistyped the name.

```
        OCaml
```

```
# x + 1;;
Error: Unbound value x
```

In our example x is not defined, so it has been underlined.

This expression has type … but an expression was expected of type …

You will see this error very frequently. It occurs when the expression's syntax is correct (i.e. it is made up of valid words and constructs), and OCaml has moved on to type-checking the expression prior to evaluation. If there is a problem with type-checking, OCaml shows you where a mismatch between the expected and actual type occurred.

```
        OCaml

# 1 + true;;
Error: This expression has type bool but an expression was expected of type
         int
```

In this example, OCaml is looking for an integer on the right hand side of the + operator, and finds something of type **bool** instead.

It is not always as easy to spot the real source of the problem, especially if the function is recursive. Nevertheless, a careful look at the program will often shine light on the problem – look at each function and its arguments, and try to find your mistake.

This function is applied to too many arguments

Exactly what it says. The function name is underlined.

```
        OCaml

# let f x = x + 1;;
val f : int -> int = <fun>
# f x y;;
Error: This function is applied to too many arguments;
maybe you forgot a `;'
```

The phrase "maybe you forgot a ';' " applies to imperative programs where accidently missing out a ';' between successive function applications might commonly lead to this error.

Unbound constructor …

This occurs when a constructor name is used which is not defined.

```
        OCaml

# type t = Roof | Wall | Floor;;
```

```
type t = Roof | Wall | Floor
# Window;;
Error: Unbound constructor Window
```

OCaml knows it is a constructor name because it has an initial capital letter.

The constructor … expects … argument(s), but is applied here to … argument(s)

This error occurs when the wrong kind of data is given to a constructor for a type. It is just another type error, but we get a specialized message.

```
        OCaml

# type p = A of int | B of bool;;
type p = A of int | B of bool
# A;;
Error: The constructor A expects 1 argument(s),
       but is applied here to 0 argument(s)
```

RUN-TIME ERRORS

In any programming language powerful enough to be of use, some errors cannot be detected before attempting evaluation of an expression (until "run-time"). The exception mechanism is for handling and recovering from these kinds of problems.

Stack overflow during evaluation (looping recursion?)

This occurs if the function builds up a working expression which is too big. This might occur if the function is never going to stop because of a programming error, or if the argument is just too big.

```
        OCaml

# let rec f x = 1 + f (x + 1);;
val f : int -> int = <fun>
# f 0;;
Stack overflow during evaluation (looping recursion?).
```

Find the cause of the unbounded recursion, and try again. If it is really not a mistake, rewrite the function to use an accumulating argument (and so, to be tail recursive).

Exception: Match_failure …

This occurs when a pattern match cannot find anything to match against. You would have been warned about this possibility when the program was originally entered. For example, if the following function `f` were defined as

```
let f x = match x with 0 -> 1
```

then using the function with `1` as an argument would produce:

```
        OCaml

# f 1;;
Exception: Match_failure ("//toplevel//", 1, 10).
```

In this example, the match failure occurred in the top level (i.e. the interactive OCaml we are using), at line one, character ten.

Exception: ...

This is printed if an un-handled exception reaches OCaml.

```
        OCaml

# exception Exp of string;;
exception Exp of string
# raise (Exp "Failed");;
Exception: Exp "Failed".
```

This can occur for built-in exceptions like `Division_by_Zero` or `Not_found` or ones the user has defined like `Exp` above.

WARNINGS

Warnings do not stop an expression being accepted or evaluated. They are printed after an expression is accepted but before the expression is evaluated. Warnings are for occasions where OCaml is concerned you may have made a mistake, even though the expression is not actually malformed. You should check each new warning in a program carefully.

This pattern-matching is not exhaustive

This warning is printed when OCaml has determined that you have missed out one or more cases in a pattern match. This could result in a `Match_failure` exception being raised at run-time.

```
        OCaml

# let f x = match x with 0 -> 1;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val f : int -> int = <fun>
```

Helpfully, it is able to generate an example of something the pattern match does not cover, so this should give you a hint about what has been missed out. You may ignore the warning if you are sure that, for other reasons, this case can never occur.

This match case is unused

This occurs when two parts of the pattern match cover the same case. In this situation, the second one could never be reached, so it is almost certain the programmer has made a mistake.

OCaml

```
# let f x = match x with _ -> 1 | 0 -> 0;;
Warning 11: this match case is unused.
val f : int -> int = <fun>
```

In this case, the first case matches everything, so the second cannot ever match.

This expression should have type unit

Sometimes when writing imperative programs, we ignore the result of some side-effect-producing function. However, this can indicate a mistake.

OCaml

```
# f 1; 2;;
Warning 10: this expression should have type unit.
- : int = 2
```

It is better to use the built-in `ignore` function in these cases, to avoid this warning:

OCaml

```
# ignore (f 1); 2;;
- : int = 2
```

The ignore function has type $\alpha \to$ **unit**. It has no side-effect.

Index