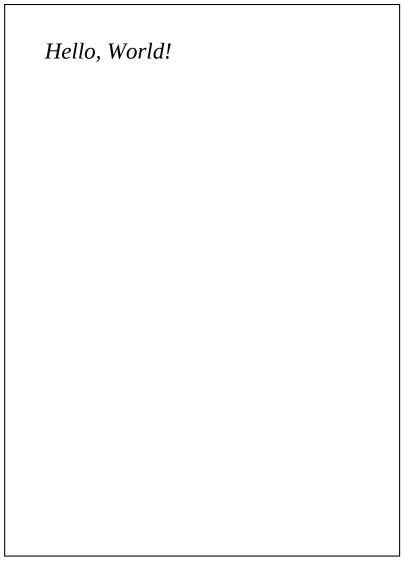
Chapter 13

Representing Documents

In this chapter we will define a data type for representing a PDF document. PDF is a structured format for describing a wide variety of graphical and textual data. The PDF file format itself is large and complex, but we will introduce only the parts required for our examples. It is relatively easy to write PDF files but rather harder to read them, so we will concentrate on creating PDF data structures in memory, and then writing them to valid files. Here is an example PDF, as it might be displayed in a PDF viewer:



Here is the corresponding code, which you would see if you opened the PDF file in a plain text editor:

```
%PDF-1.1
1 0 obj
<</Type /Page
  /Parent 3 0 R
  /Resources
```

```
    <</Font
       <</F0
          <</Type /Font /Subtype /Type1 /BaseFont /Times-Italic>>>>>>
       /MediaBox [0 0 595.275590551 841.88976378]
       /Rotate 0 /Contents [4 0 R] >>
endobj
2 0 obj
<</Type /Catalog /Pages 3 0 R>>
endobj
3 0 obj
<</Type /Pages /Kids [1 0 R] /Count 1>>
endobj
4 0 obj
<</Length 53>>
stream
1 0 0 1 50 770 cm BT /F0 36 Tf (Hello, World!) Tj ET
endstream
endobj
xref
0 5
0000000000 65535 f
0000000015 00000 n
0000000200 00000 n
0000000245 00000 n
0000000296 00000 n
trailer
<</Size 5 /Root 2 0 R>>
startxref
397
%%EOF
```

Rather complicated, as we can see. Our first job is to define a pleasant OCaml data type for PDF documents, which can then be flattened to the format above when written to a file.

The main body of a PDF file is a set of numbered objects – there are four in the example above, from `1 0 obj` to `4 0 obj`. Each one contains some structured data, such as the *dictionary* `<</Type /Pages /Kids [1 0 R] /Count 1>>` which associates the keys `/Type`, `/Kids`, and `/Count` to the name `/Pages`, the array `[1 0 R]`, and the integer `1` respectively. Before and after the main body is some ancillary data, most of which we do not need to hold in our data structure – it is generated upon writing. Here are all the kinds of data we will be using:

- Booleans, like `true` and `false`
- Integers, such as 4, 256, -1
- Floating-point numbers such as 1.585
- Strings, like `(a string)` which are sequences of characters within parentheses
- Names, like `/Name`
- Ordered arrays of objects such as `[1 2 4]`
- Dictionaries, which are unordered collections of key-value pairs, where the keys are names. For example, `<</One 1 /Two 2 /Three 3>>`.

- Streams, like object 4 in the example above, which are arbitrary sequences of bytes.
- Indirect references, like `4 0 R` which point to another object by its number (here, object 4).

Here is an OCaml data type to hold such data:

```
type pdfobject =
  Boolean of bool
| Integer of int
| Float of float
| String of string
| Name of string
| Array of pdfobject list
| Dictionary of (string * pdfobject) list
| Stream of pdfobject * string
| Indirect of int
```

Note that it is recursive, mirroring the structure of the data. For example, object 3 in the example above, that is `<</Type /Pages /Kids [1 0 R] /Count 1>>`, will be represented as:

```
Dictionary
  [("/Type", Name "/Pages"); ("/Kids", Array [Indirect 1]); ("/Count", Integer 1)]
```

Now, we need a type to represent the whole document, which contains a list of these objects, the PDF version number (1.1 in the example above), and the *trailer dictionary* (`<</Size 5 /Root 2 0 R>>` above). Everything else is generated upon writing. It is traditional to name the main type of a module t:

```
type t =
  {version : int * int;
   objects : (int * pdfobject) list;
   trailer : pdfobject}
```

We put these two types into `pdf.ml` and `pdf.mli`. Here is how we might build an instance of this data type representing our example PDF:

```
let objects =
  [(1,
     Pdf.Dictionary
       [("/Type", Pdf.Name "/Page");
        ("/Parent", Pdf.Indirect 3);
        ("/Resources",
           Pdf.Dictionary
             [("/Font",
              Pdf.Dictionary
                [("/F0",
                  Pdf.Dictionary
                    [("/Type", Pdf.Name "/Font");
                     ("/Subtype", Pdf.Name "/Type1");
                     ("/BaseFont", Pdf.Name "/Times-Italic")])])]);
        ("/MediaBox",
          Pdf.Array
            [Pdf.Float 0.; Pdf.Float 0.;
```

```
             Pdf.Float 595.275590551; Pdf.Float 841.88976378]);
        ("/Rotate", Pdf.Integer 0);
        ("/Contents", Pdf.Array [Pdf.Indirect 4])]);
   (2,
     Pdf.Dictionary
       [("/Type", Pdf.Name "/Catalog");
        ("/Pages", Pdf.Indirect 3)]);
   (3,
     Pdf.Dictionary
       [("/Type", Pdf.Name "/Pages");
        ("/Kids", Pdf.Array [Pdf.Indirect 1]);
        ("/Count", Pdf.Integer 1)]);
   (4,
     Pdf.Stream
       (Pdf.Dictionary [("/Length", Pdf.Integer 53)],
        "1 0 0 1 50 770 cm BT /F0 36 Tf (Hello, World!) Tj ET"))]

let hello =
  {Pdf.version = (1, 1);
   Pdf.objects = objects;
   Pdf.trailer =
   Pdf.Dictionary
    [("/Size", Pdf.Integer 5);
     ("/Root", Pdf.Indirect 2)]}
```

The advantage of using this data structure as opposed to generating the PDF file directly is that it may be programmatically manipulated with ease, using pattern matching and other standard OCaml techniques. Note that the content of the page itself `"1 0 0 1 50 770 cm BT /F0 36 Tf (Hello, World!) Tj ET"` remains a plain string. We shall look at this separate language soon.

In the next three chapters, we will learn how to write this representation to a file, and add our own text and graphics to the page.

Questions

1. Draw the graph of the relationships, via indirect references such as `3 0 R`, of the objects 1, 2, 3, 4 and the trailer dictionary.
2. Represent the following PDF objects using our data type:

   ```
   /Name

   (Quartz Crystal)

   <</Type /ObjStm /N 100 /First 807 /Length 1836 /Filter /FlateDecode>>

   [1 2 1.5 (black)]

   [1 2 0 R]
   ```

3. PDF files can contain arbitrary objects, which will be ignored by a PDF reader if they are not understood. Design a way of representing items of the following type using one or more PDF objects:

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

4. Write a function of type pdfobject $\rightarrow$ pdfobject which, given an object, replaces the value of any dictionary entry with key `/Rotate` to `90`.

Chapter 14

Writing Documents

We have built a representation for PDF documents to be held in memory, and defined an example document. Now, we must build functions to write this to file. Recall our example file from the last chapter:

```
%PDF-1.1
1 0 obj
<</Type /Page
  /Parent 3 0 R
  /Resources
    <</Font
       <</F0
          <</Type /Font /Subtype /Type1 /BaseFont /Times-Italic>>>>>>
       /MediaBox [0 0 595.275590551 841.88976378]
       /Rotate 0 /Contents [4 0 R] >>
endobj
2 0 obj
<</Type /Catalog /Pages 3 0 R>>
endobj
3 0 obj
<</Type /Pages /Kids [1 0 R] /Count 1>>
endobj
4 0 obj
<</Length 53>>
stream
1 0 0 1 50 770 cm BT /F0 36 Tf (Hello, World!) Tj ET
endstream
endobj
xref
0 5
0000000000 65535 f
0000000015 00000 n
0000000200 00000 n
0000000245 00000 n
0000000296 00000 n
trailer
<</Size 5 /Root 2 0 R>>
```

```
startxref
397
%%EOF
```

It consists, we remember, of a *header*, then some *objects* (here, four), and a *trailer*. We will need four functions:

1. The function `string_of_pdfobject` to make a string from a Pdf.pdfobject, for example making the string "`<</Type /Pages /Kids [1 0 R] /Count 1>>`" from the pdfobject `Dictionary [("/Type", Name "/Pages"); ("/Kids", Array [Indirect 1]); ("/Count", Integer 1)];`
2. The function `write_header` to write the header (i.e. everything before the objects);
3. The function `write_trailer` to write the trailer (i.e. everything after the objects); and
4. The main function `pdf_to_file` which uses these three functions to write a Pdf.t to a file under the given file name.

The function `string_of_pdfobject` is best expressed as a set of mutually-recursive functions introduced with the **`let rec`** `...` **`and`** `...` construct. To give it as a single function is possible, but would be rather large, and so harder to read and edit. Let us present it piece-by-piece and then all at once.

First, the main part. Given an object we wish to produce its string, assuming that `string_of_array`, `string_of_dictionary`, and `string_of_stream` exist:

```
string_of_pdfobject : Pdf.pdfobject → string

let rec string_of_pdfobject obj =
  match obj with
    Pdf.Boolean b -> string_of_bool b
  | Pdf.Integer i -> string_of_int i
  | Pdf.Float f -> string_of_float f
  | Pdf.String s -> "(" ^ s ^ ")"
  | Pdf.Name n -> n
  | Pdf.Array a -> string_of_array a
  | Pdf.Dictionary d -> string_of_dictionary d
  | Pdf.Stream (dict, data) -> string_of_stream dict data
  | Pdf.Indirect i -> Printf.sprintf "%i 0 R" i
```

The cases are all simple, except for the ones we have put aside to be implemented separately. Strings must be put between parentheses, indirect references are printed as `2 0 R` etc.

To build a string from an `Array`, we need to start with an open square bracket, add the string for each Pdf.pdfobject in the array (which may be arbitrarily complex, of course), and put spaces between them. Then we end with a close square bracket:

```
string_of_array : Pdf.pdfobject → string

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

It is natural to use the **Buffer** module from the Standard Library to collect these strings together. It is also more efficient than using string concatenation. Notice we do not add an initial space after the square bracket, but do before the closing square bracket, for symmetry. The `string_of_dictionary` function is somewhat similar:

```
string_of_dictionary : Pdf.pdfobject → string

let rec string_of_dictionary d =
  let b = Buffer.create 100 in
    Buffer.add_string b "<<";
    List.iter
      (fun (k, v) ->
         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
```

Now for `string_of_stream`. A stream in PDF is written like this:

stream dictionary
`stream`
stream data
`endstream`

This is simple with the **Buffer** module too:

```
string_of_stream : Pdf.pdfobject → string → string

let rec string_of_stream dict data =
  let b = Buffer.create 100 in
    List.iter (Buffer.add_string b)
      [string_of_pdfobject dict; "\nstream\n"; data; "\nendstream"];
    Buffer.contents b
```

The code for these four functions is collected together in Figure 14.1. Now that we can build a string from any Pdf.pdfobject, we can proceed to build the `write_header`, `write_trailer`, and `write_objects` functions, so that we have everything we need for the final `pdf_to_file` function.

The PDF header consists of `%%PDF-`*m*`.`*n* where *m* and *n* are the major and minor version numbers. This is easy to build with `Printf.sprintf`:

```
write_header : out_channel → Pdf.t → unit

let write_header o {Pdf.version = (major, minor)} =
  output_string o
    (Printf.sprintf "%%PDF-%i.%i\n" major minor)
```

The `write_objects` function, which is given a list of (**int** × Pdf.pdfobject) pairs, sorts the objects by their number, and then outputs each object using `string_of_pdfobject`. For example, the pair

```
(3,
  Dictionary
    [("/Type", Name "/Pages"); ("/Kids", Array [Indirect 1]); ("/Count", Integer 1)])
```

will be written to file like this:

```
3 0 obj
<< /Type /Pages /Kids [1 0 R] /Count 1 >>
endobj
```

The function collects and returns a list of the byte offsets of the objects written, since this will be needed to write the trailer section, which we shall describe in a moment.

```
write_objects : out_channel → (int × Pdf.pdfobject) → int list

let write_objects o objs =
  let offsets = ref [] in
    List.iter
      (fun (objnum, obj) ->
         offsets := pos_out o :: !offsets;
         output_string o (Printf.sprintf "%i 0 obj\n" objnum);
         output_string o (string_of_pdfobject obj);
         output_string o "\nendobj\n")
      (List.sort objs);
    List.rev !offsets
```

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

let rec string_of_array a =
  let b = Buffer.create 100 in
    Buffer.add_string b "[";
    List.iter
      (fun s ->
         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) ->
         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

and string_of_stream dict data =
  let b = Buffer.create 100 in
    List.iter (Buffer.add_string b)
      [string_of_pdfobject dict; "\nstream\n"; data; "\nendstream"];
    Buffer.contents b

and string_of_pdfobject obj =
  match obj with
    Pdf.Boolean b -> string_of_bool b
  | Pdf.Integer i -> string_of_int i
  | Pdf.Float f -> string_of_float f
  | Pdf.String s -> "(" ^ s ^ ")"
  | Pdf.Name n -> n
  | Pdf.Array a -> string_of_array a
  | Pdf.Dictionary d -> string_of_dictionary d
  | Pdf.Stream (dict, data) -> string_of_stream dict data
  | Pdf.Indirect i -> Printf.sprintf "%i 0 R" i
```

Figure 14.1: Our mutually recursive functions to build a string from a Pdf.pdfobject

Now, we can write the trailer. That is to say, this section:

```
xref
0 5
0000000000 65535 f
0000000015 00000 n
0000000230 00000 n
0000000279 00000 n
0000000338 00000 n
trailer
<< /Size 5 /Root 2 0 R >>
startxref
440
%%EOF
```

This consists largely of a list of the byte offsets of the four objects in the file (15, 230, 279, 338), the trailer dictionary, and so on. It is not necessary to understand it in detail. This is the first section which would be read by a PDF reader, to find the actual objects in the file.

```
write_trailer : out_channel → Pdf.t → int list → unit

let write_trailer o pdf offsets =
  let startxref = pos_out o in
    output_string o "xref\n";
    output_string o
      (Printf.sprintf "0 %i\n" (List.length pdf.Pdf.objects + 1));
    output_string o "0000000000 65535 f \n";
    List.iter
      (fun offset ->
         output_string o (Printf.sprintf "%010i 00000 n \n" offset))
      offsets;
    output_string o "trailer\n";
    output_string o (string_of_pdfobject pdf.Pdf.trailer);
    output_string o "\nstartxref\n";
    output_string o (string_of_int startxref);
    output_string o "\n%%EOF"
```

Now the main function is simple. We open the file, write the header, objects, and trailer, and close the file. In the event of an exception, we close the file to clean up, and re-raise it.

```
pdf_to_file : Pdf.t → string → unit

let pdf_to_file pdf filename =
  let output = open_out_bin filename in
    try
      write_header output pdf;
      let offsets = write_objects output pdf.Pdf.objects in
        write_trailer output pdf offsets;
        close_out output
    with
      e -> close_out output; raise e
```

This code goes into `pdfwrite.ml`. The interface `pdfwrite.mli` is very simple. We expose only `pdf_to_file`:

```
val pdf_to_file : Pdf.t -> string -> unit
```

Here is the output of our program on the example Pdfhello.hello:

```
%PDF-1.1
1 0 obj
<< /Type /Page /Parent 3 0 R /Resources << /Font << /F0 << /Type /Font /Subtype /
    Type1 /BaseFont /Times-Italic >> >> >> /MediaBox [ 0. 0. 595.275590551
    841.88976378 ] /Rotate 0 /Contents [ 4 0 R ] >>
endobj
2 0 obj
<< /Type /Catalog /Pages 3 0 R >>
endobj
3 0 obj
<< /Type /Pages /Kids [ 1 0 R ] /Count 1 >>
endobj
4 0 obj
<< /Length 53 >>
stream
1 0 0 1 50 770 cm BT /F0 36 Tf (Hello, World!) Tj ET
endstream
endobj
xref
0 5
0000000000 65535 f
0000000015 00000 n
0000000230 00000 n
0000000279 00000 n
0000000338 00000 n
trailer
<< /Size 5 /Root 2 0 R >>
startxref
```

```
440
%%EOF
```

This loads correctly into a PDF viewer. Notice that some of the spacing is different from the specimen PDF we had at the top of the chapter – this does not matter. Now we can move on to putting more interesting content on the page of our PDF documents.

Questions

1. Modify the code so that arrays and dictionaries do not have spaces at either end of them (these spaces are not required). For example, we should see `[2 0 R]` rather than `[ 2 0 R ]`.
2. Build an example PDF with three pages. This will have three entries in its `/Kids` array, and three page objects. The page contents may be shared by the three pages if desired. You will need to number the additional objects carefully.
3. Modify the program to use our byte-by-byte compression algorithm from Chapter 6 to compress the content stream in the "Hello, World" program. You will need to fix the `/Length` and add `/Filter /RunLengthDecode` to the stream dictionary. Now modify the program to use our bit-by-bit compression algorithm from the same chapter with `/Filter /CCITTFaxDecode`. You will need to add the following entry to the stream dictionary, in addition to altering the `/Filter` entry:

   ```
   ("/DecodeParms",
       Pdf.Dictionary
         [("/Rows", Pdf.Integer 1);
          ("/Columns", Pdf.Integer (52 * 8));
          ("/BlackIs1", Pdf.Boolean true);
          ("/EndOfBlock", Pdf.Boolean false)])
   ```

   The number 52 is the number of bytes of data in our example.