CamlPDF

A PDF library for Objective Caml

John Whitington
Coherent Graphics Ltd

Request For Comment

This is a CamlPDF, an OCaml library for reading, writing and manipulating Adobe portable document files. It is presented as a literate program in the manner of Knuth. Since this is an implementation of someone else's technology (unlike the rest of our work, which is new), we have decided to release it as open source software.

CamlPDF consists of a set of low level modules for representing, reading and writing the basic structure of PDF together with a higher level API. The auxiliary libraries Utility, Io, Units and Paper are not directly related to PDF.

Five examples (Pdfhello, Pdfdecomp, Pdfmerge, Pdfdraft, Pdftest) are provided (and presented in the appendices of this document).

CamlPDF is released under a BSD licence with special exceptions. See the LICENCE file in the source for details.

Please advise of the following:

- Files which cannot be read or written, or any other runtime error;
- Suggestions as to the extension of the higher level APIs;
- Instances of particularly slow or resource-hungry scenarios.

Please be aware that PDF is a highly complex format and that many files are malformed. We will incorporate support for malformed files if Acrobat reads them.

John Whitington
john@coherentgraphics.co.uk

Cambridge, England, November 2007

Contents

Introduction

This book contains the code and documentation for CamlPDF, a library for reading, processing and writing Adobe PDF files. It is written using *ocamlweb*. Documentation (in LaTeX) is interwoven with code (in Objective Caml).

This PDF is produced from source by ocamlweb, PDFLaTeX and BibTeX using the command `make literate`.

Scope

This document does not contain the interface files. This information can be found in the `ocamldoc`-generated literature supplied. However, functions which appear in the interface (either fully, or as abstract types) are indicated by a $\triangleright$ in the margin. This also helps to demarcate the sections of auxiliary functions in the code which lead up to each exposed function.

Nor does this document contain C files (but you can see the `external` declarations which reference them), nor does it contain makefiles and the like.

Order of Reading

The code is presented in compilation order save for the Utility, IO, Units and Paper modules which, being of a general nature, are at the end. The Utility module defines a number of commonly used functions and redefines a number of functions from Pervasives (including infix symbols) which are used without comment in the main parts of the program. The Io module provides for generic input and output to channels and bytestreams. It is therefore worth familiarising oneself with the appendices first. The Utility module is a somewhat arbitrary set of common functions used at Coherent Graphics — not all functions defined are used in CamlPDF.

Part I

Ancillary Libraries

1 Module Utility
Common functions

This module contains general-purpose functions used in many modules. Typically a module will use the **open** keyword to bring these definitions up to top level, so their names are considered reserved words in other modules.

All functions in this module are tail-recursive. All are exposed in the interface.

Print something and then flush standard output.

```
let flprint s =
  print_string s; flush stdout
```

Debug printing

```
let dp_print = ref false
```

```
let dpr s = if !dp_print then flprint s
```

1.1 Functions on lists

xxx is a tail-recursive version of List.*xxx*. See List module for details.

```
let sort = List.sort
```

```
let hd = List.hd
```

```
let tl = List.tl
```

```
let rev = List.rev
```

```
let iter = List.iter
```

```
let iter2 = List.iter2
```

```
let rec iter3 f a b c =
  match a, b, c with
  | [], [], [] → ()
  | ah :: a', bh :: b', ch :: c' →
      f ah bh ch;
      iter3 f a' b' c'
  | _ → raise (Invalid_argument "iter3")
```

```
let append a b =
  List.rev_append (rev a) b
```

```
let ( @ ) = append

let flatten lists =
  let rec flatten out = function
    | [] → out
    | l :: ls → flatten (append l out) ls
  in
    flatten [] (rev lists)

let rev_map = List.rev_map

let map f l =
  rev (List.rev_map f l)

let map2 f a b =
  rev (List.rev_map2 f a b)

let split l =
  let rec split_inner (l1, l2) = function
    | [] → rev l1, rev l2
    | (a, b) :: t → split_inner (a :: l1, b :: l2) t
  in
    split_inner ([], []) l

let split3 l =
  let rec split3_inner (l1, l2, l3) = function
    | [] → rev l1, rev l2, rev l3
    | (a, b, c) :: t → split3_inner (a :: l1, b :: l2, c :: l3) t
  in
    split3_inner ([], [], []) l

let split8 l =
  let rec split8_inner (l1, l2, l3, l4, l5, l6, l7, l8) = function
    | [] → rev l1, rev l2, rev l3, rev l4, rev l5, rev l6, rev l7, rev l8
    | (a, b, c, d, e, f, g, h) :: t →
        split8_inner (a :: l1, b :: l2, c :: l3, d :: l4, e :: l5, f :: l6, g :: l7, h ::
l8) t
  in
    split8_inner ([], [], [], [], [], [], [], []) l

let combine a b =
  let pairs = ref [] in
    try
      List.iter2 (fun x y → pairs := (x, y) ::!pairs) a b;
      rev !pairs
    with
      Invalid_argument _ → raise (Invalid_argument "Utility.combine")

let combine3 a b c =
  let pairs = ref [] in
    try
      iter3 (fun x y z → pairs := (x, y, z) ::!pairs) a b c;
      rev !pairs
    with
      Invalid_argument _ → raise (Invalid_argument "Utility.combine3")
```

```
let fold_left f b l = List.fold_left f b l

let fold_right f l e =
  List.fold_left (fun x y → f y x) e (rev l)

let length = List.length

let rec rev_map3_inner f a b c outputs =
  match a, b, c with
  | [], [], [] → outputs
  | ha :: ta, hb :: tb, hc :: tc →
      rev_map3_inner f ta tb tc (f ha hb hc :: outputs)
  | _ → raise (Invalid_argument "map3")

let rev_map3 f a b c =
  rev_map3_inner f a b c []

let map3 f a b c =
  rev (rev_map3 f a b c)

let rec rev_map4_inner f a b c d outputs =
  match a, b, c, d with
  | [], [], [], [] → outputs
  | ha :: ta, hb :: tb, hc :: tc, hd :: td →
      rev_map4_inner f ta tb tc td (f ha hb hc hd :: outputs)
  | _ → raise (Invalid_argument "map4")

let rev_map4 f a b c d =
  rev_map4_inner f a b c d []

let map4 f a b c d =
  rev (rev_map4 f a b c d)

let rec rev_map5_inner f a b c d e outputs =
  match a, b, c, d, e with
  | [], [], [], [], [] → outputs
  | ha :: ta, hb :: tb, hc :: tc, hd :: td, he :: te →
      rev_map5_inner f ta tb tc td te (f ha hb hc hd he :: outputs)
  | _ → raise (Invalid_argument "map5")

let rev_map5 f a b c d e =
  rev_map5_inner f a b c d e []

let map5 f a b c d e =
  rev (rev_map5 f a b c d e)
```

Calculate the cumulative sum of a list given a base e.g $cumulative\_sum\ 5[1;2;3] = [6;\ 8;\ 11]$

```
let cumulative_sum b l =
  let rec cumulative_sum prev bse = function
    | [] → rev prev
    | h :: t → cumulative_sum ((bse + h) :: prev) (bse + h) t
  in
    cumulative_sum [] b l
```

Split a list into a list of lists at every point where p is true

```
let rec split_around_inner p prev curr = function
  | [] → if curr = [] then (rev prev) else (rev (rev curr :: prev))
  | h :: t →
      if p h
        then split_around_inner p (rev curr :: prev) [] t
        else split_around_inner p prev (h :: curr) t

let split_around p l =
  split_around_inner p [] [] l
```

Count the number of elements matching a predicate.

```
let rec lcount_inner p c = function
  | [] → c
  | h :: t →
      if p h
        then lcount_inner p (c + 1) t
        else lcount_inner p c t

let lcount p l =
  lcount_inner p 0 l
```

Find the position of the first element matching a predicate. The first element is number one.

```
let rec index_inner n p = function
  | [] → dpr "b"; raise Not_found
  | h :: t when p h → n
  | _ :: t → index_inner (n + 1) p t

let index n p = index_inner 1 n p
```

1.2 Functions on Strings

```
let firstchar (s : string) =
  try Some s.[0] with Invalid_argument _ → dpr "3R"; None

let lastchar (s : string) =
  try Some s.[String.length s − 1] with Invalid_argument _ → dpr "3S"; None
```

Make a list of characters from a string, preserving order.

```
let explode s =
  let l : char list ref = ref [] in
    if String.length s = 0 then !l else
      begin
        for p = 1 to String.length s do
          l := s.[p − 1] ::!l
        done;
        rev !l
      end
```

Make a string from a list of characters, preserving order.

```
let implode l =
  let b = Buffer.create (List.length l) in
    List.iter (Buffer.add_char b) l;
    Buffer.contents b
```

String of character.

```
let string_of_char c =
  implode [c]
```

1.3 Long-integer function abbreviations

```
let i32ofi = Int32.of_int

let i32toi = Int32.to_int

let i32tof = Int32.to_float

let i32add = Int32.add

let i32sub = Int32.sub

let i32mul = Int32.mul

let i32div = Int32.div

let lsr32 = Int32.shift_right_logical

let lsl32 = Int32.shift_left

let lor32 = Int32.logor

let land32 = Int32.logand

let lxor32 = Int32.logxor

let i32succ = Int32.succ

let i32pred = Int32.pred

let i32max = Pervasives.max

let i32min = Pervasives.min

let i64ofi = Int64.of_int

let i64toi = Int64.to_int

let i64add = Int64.add

let i64sub = Int64.sub

let i64mul = Int64.mul

let i64div = Int64.div

let lsr64 = Int64.shift_right_logical

let lsl64 = Int64.shift_left

let lor64 = Int64.logor
```

```
let i64succ = Int64.succ
let i64pred = Int64.pred
let i64max = Pervasives.max
let i64min = Pervasives.min
```

1.4 Byte streams

IF-OCAML
Type abbreviation for byte-addressable arrays.

```
type bytestream =
  (int, Bigarray.int8_unsigned_elt, Bigarray.c_layout) Bigarray.Array1.t
```

Make a stream of a given size.

```
let mkstream =
  Bigarray.Array1.create Bigarray.int8_unsigned Bigarray.c_layout
```

Find the size of a stream.

```
let stream_size = Bigarray.Array1.dim

let sset s n v =
  s.{n} ← v

let sget s n =
  s.{n}
```

ENDIF-OCAML
For lexing / parsing byte streams, keep the position. Starts at zero.

```
type stream =
  {mutable pos : int;
   mutable data : bytestream}
```

Fill a stream with a value.

```
let fillstream v s =
  for x = 0 to stream_size s − 1 do sset s x v done

let print_stream s =
  if stream_size s > 0 then
    for x = 0 to stream_size s − 1 do
      Printf.printf "%i " (sget s x)
    done
```

Make a bytestream from a string, with no terminator.

```
let bytestream_of_string s =
  let l = String.length s in
    let stream = mkstream l in
      if l > 0 then
        for k = 0 to l − 1 do
          sset stream k (int_of_char s.[k])
        done;
      stream
```

Make a byte stream from an integer list.

```
let bytestream_of_list l =
  let length = length l in
    if length = 0 then mkstream 0 else
      let s = mkstream length and l = ref l in
        for pos = 0 to length - 1 do
          sset s pos (hd !l);
          l := tl !l
        done;
        s
```

Convert a character list to a stream.

```
let bytestream_of_charlist cs =
  let length = length cs in
    if length = 0 then mkstream 0 else
      let s = mkstream length and cs = ref cs in
        for pos = 0 to length - 1 do
          sset s pos (int_of_char (hd !cs));
          cs := tl !cs
        done;
        s

let bytestream_of_arraylist l =
  let totalsize = fold_left ( + ) 0 (map Array.length l) in
    let output = mkstream totalsize
    and pos = ref 0 in
      iter
        (fun a ->
           for x = 0 to Array.length a - 1 do
             sset output !pos a.(x); incr pos
           done)
        l;
      output

let string_of_bytestream s =
  let l = stream_size s in
    let buf = Buffer.create l in
      for x = 0 to l - 1 do
        Buffer.add_char buf (char_of_int (sget s x))
      done;
      Buffer.contents buf

let stream_of_int_array a =
  let s = mkstream (Array.length a) in
    for i = 0 to stream_size s - 1 do
      sset s i a.(i)
    done;
    s
```

```
let int_array_of_stream s =
  let a = Array.make (stream_size s) 0 in
    for i = 0 to Array.length a − 1 do
      a.(i) ← sget s i
    done;
    a
```

Copy a stream.

```
let copystream s =
  let l = stream_size s in
    let s′ = mkstream l in
      if l > 0 then
        for k = 0 to l − 1 do
          sset s′ k (sget s k)
        done;
      s′
```

```
let int_array_of_string s =
  Array.init (String.length s) (fun i → int_of_char s.[i])
```

```
let string_of_int_arrays arrays =
  let len = fold_left ( + ) 0 (map Array.length arrays) in
    let buf = Buffer.create len in
      iter (Array.iter (fun v → Buffer.add_char buf (char_of_int v))) arrays;
      Buffer.contents buf
```

```
let string_of_int_array a =
  string_of_int_arrays [a]
```

Perform computation c until an exception is raised, with the dummy return value r, of the type of the expression evaluted when the exception is caught.

```
let until_exception r c =
  while true do c () done; r
```

Set each element of array a to value v.

```
let set_array a v =
  Array.fill a 0 (Array.length a) v
```

Evaluate v (), evaluate and ignore f (), return v (), in that order.

```
let do_return v f =
  let r = v () in ignore (f ()); r
```

Call f () some number of times.

```
let rec do_many f = function
  | n when n < 0 → raise (Invalid_argument "do_many")
  | 0 → ()
  | n → f (); do_many f (n − 1)
```

Interleave an element among a list, so that *interleave* 0 [1; 2; 3] yields [1; 0; 2; 0; 3]. An empty or singleton list is unchanged.

```
let interleave e l =
  let rec interleave_inner result elt = function
    | [] → rev result
    | [e] → interleave_inner (e :: result) elt []
    | h :: t → interleave_inner (elt :: h :: result) elt t
  in
    interleave_inner [] e l
```

Interleave two same-length lists together, taking from the first list first.

```
let interleave_lists a b =
  let rec interleave_lists_inner r a b =
    match a, b with
    | [], [] → rev r
    | h :: t, h′ :: t′ → interleave_lists_inner (h′ :: h :: r) t t′
    | _ → raise (Invalid_argument "interleave_lists")
  in
    interleave_lists_inner [] a b
```

Cons on list references

```
let ( =| ) r e =
  r := e ::!r
```

Append on list references

```
let ( =@ ) r l =
  r := l @ !r
```

Functions on characters.

```
let isdigit = function
  | '0'..'9' → true
  | _ → false
```

Be sure to open Utility after Bigarray if using both libraries since this name clashes with Bigarray.int.

Abbreviation.

```
let toint x = int_of_float x
```

Invert a predicate.

```
let notpred f =
  function e → ¬ (f e)
```

Prefix equality

```
let eq = ( = )
```

```
let neq = ( ≠ )
```

Map on a list of lists

```
let map_lol f =
  map (map f)
```

Raise x to the power i.

```
let rec pow i x =
  match i with
  | 0 → 1
  | 1 → x
  | i → pow (i / 2) (x × x) × (if i mod 2 = 0 then 1 else x)
```

1.5 Queues

Efficient Queues (F.W. Burton, 1982)

type α *queue* $=$ α *list* $\times$ α *list*

Make an empty queue

let $q\_mk \ = \ (([\,],\ [\,])\ :\ \alpha\ queue)$

let $list\_of\_q\ (a,\ b)\ =$
$\quad a\ @\ rev\ b$

Put a queue into normal form.

let $q\_norm\ =$ **function**
$\quad |\ [\,],\ r\ \rightarrow\ rev\ r,\ [\,]$
$\quad |\ q\ \rightarrow\ q$

Enqueue

let $q\_enq\ (f,\ r)\ e\ =$
$\quad q\_norm\ (f,\ e :: r)$

let $q\_of\_list\ l\ =$
$\quad fold\_left\ q\_enq\ q\_mk\ l$

Null predicate

let $q\_null\ =$ **function**
$\quad |\ (([\,],\ [\,])\ :\ \alpha\ queue)\ \rightarrow$ **true**
$\quad |\ \_\ \rightarrow$ **false**

Raised when an attempt is made to peek or dequeue on an empty queue.

exception EmptyQueue

Peek at the head

let $q\_hd\ =$ **function**
$\quad |\ ((h :: \_,\ \_)\ :\ \alpha\ queue)\ \rightarrow\ h$
$\quad |\ \_\ \rightarrow\ dpr$ `"c"`; $raise$ EmptyQueue

Dequeue

let $q\_deq\ =$ **function**
$\quad |\ \_ :: t,\ r\ \rightarrow\ q\_norm\ (t,\ r)$
$\quad |\ \_\ \rightarrow\ dpr$ `"d"`; $raise$ EmptyQueue

let $q\_len\ ((a,\ b)\ :\ \alpha\ queue)\ =$
$\quad length\ a\ +\ length\ b$

1.6 Dictionaries implemented as association lists

Look something up in a dictionary.

```
let rec lookup k' = function
  | [] → None
  | (k, v) :: t → if k = k' then Some v else lookup k' t
```

Same, but no *option* type.

```
let rec lookup_failnull k' = function
  | [] → dpr "e"; raise Not_found
  | (k, v) :: t → if k = k' then v else lookup_failnull k' t
```

Add something to a dictionary, replacing it if it's already there.

```
let add k' v d =
  let rec add_inner r k' v = function
    | [] → (k', v) :: r
    | (k, _) :: t when k = k' → r @ ((k', v) :: t)
    | h :: t → add_inner (h :: r) k' v t
  in
    add_inner [] k' v d
```

Replace something in a dictionary, failing if it doesn't exist.

```
let replace k' v l =
  let rec replace_inner r k' v = function
    | [] → dpr "f"; raise Not_found
    | (k, _) :: t when k = k' → r @ ((k', v) :: t)
    | h :: t → replace_inner (h :: r) k' v t
  in
    replace_inner [] k' v l
```

Remove something from a dictionary.

```
let remove k' l =
  let rec remove_inner r k' = function
    | [] → r
    | (k, _) :: t when k = k' → r @ t
    | h :: t → remove_inner (h :: r) k' t
  in
    remove_inner [] k' l
```

Merge two dictionaries, prefering elements in the second in the case of clashes.

```
let rec mergedict d = function
  | [] → d
  | (k, v) :: es → mergedict (add k v d) es
```

An infix operator for the composition of functions.

```
let ( <| ) a b = a b
```

Opposite version of @

```
let ( @@ ) a b = b @ a
```

In order to return pairs of list from recursive functions without recourse to accumulating arguments.

```
let conspair ((x, y), (xs, ys)) = x :: xs, y :: ys
```

The same with options determining whether or not each element is included in the output list.

```
let conspairopt ((xo, yo), (xs, ys)) =
  (match xo with None → xs | Some x → x :: xs),
  (match yo with None → ys | Some y → y :: ys)
```

1.7 Functions on lists

Make consecutive elements of an even-length list into a list of pairs.

```
let pairs_of_list l =
  let rec pairs_of_list_inner r = function
    | [] → rev r
    | [_] → raise (Invalid_argument "pairs_of_list")
    | h :: h' :: t → pairs_of_list_inner ((h, h') :: r) t
  in
    pairs_of_list_inner [] l

let charlist_of_bytestream s =
  let l = ref [] in
    for x = stream_size s − 1 downto 0 do
      l =| char_of_int (sget s x)
    done;
    !l
```

Return a list identical to the input but with any item true under predicate p replaced with o.

```
let replaceinlist p o l =
  let rec replaceinlist_inner r p o = function
    | [] → rev r
    | h :: t →
        if p h
          then replaceinlist_inner (o :: r) p o t
          else replaceinlist_inner (h :: r) p o t
  in
    replaceinlist_inner [] p o l
```

Produce a list of overlapping pairs of elements in a list in order, producing the empty list if on singleton input.

```
let pairs l =
  let rec pairs_inner r = function
    | [] | [_] → rev r
    | a :: b :: rest → pairs_inner ((a, b) :: r) (b :: rest)
  in
    pairs_inner [] l
```

Predicate to test if x is a member of a list.

let $mem\ =\ $ List.mem

The same, with reversed arguments.

let rec $mem'\ l\ x\ =\ mem\ x\ l$

Setify. Does not preserve order.

let $setify\ l\ =$
  let rec $setify\_inner\ r\ =$ **function**
    $|\ [\,]\ \rightarrow\ r$
    $|\ h :: t\ \rightarrow$
      if $mem\ h\ t$
        then $setify\_inner\ r\ t$
        else $setify\_inner\ (h :: r)\ t$
  in
    $setify\_inner\ [\,]\ l$

The first instance of an element survives

let $setify\_preserving\_order\ l\ =$
  $setify\ (rev\ l)$

Remove all elts of l' from l if l, l' sets.

let $setminus\ l\ l'\ =$
  let rec $setminus\_inner\ r\ l\ l'\ =$
    match l **with**
    $|\ [\,]\ \rightarrow\ r$
    $|\ h :: t\ \rightarrow$
      if $mem\ h\ l'$
        then $setminus\_inner\ r\ t\ l'$
        else $setminus\_inner\ (h :: r)\ t\ l'$
  in
    $setminus\_inner\ [\,]\ l\ l'$

let $setminus\_preserving\_order\ l\ l'\ =$
  $rev\ (setminus\ l\ l')$

Return a list of the heads of a list of lists.

let $heads\ l\ =$
  let rec $heads\_inner\ r\ =$ **function**
    $|\ [\,]\ \rightarrow\ rev\ r$
    $|\ h :: t\ \rightarrow\ heads\_inner\ (hd\ h :: r)\ t$
  in
    $heads\_inner\ [\,]\ l$

Return a list of the tails of a list of lists, failing if any of them are the empty list.

let $tails\ l\ =$
  let rec $tails\_inner\ r\ =$ **function**
    $|\ [\,]\ \rightarrow\ rev\ r$
    $|\ h :: t\ \rightarrow\ tails\_inner\ (tl\ h :: r)\ t$
  in
    $tails\_inner\ [\,]\ l$

Take a list of lists of equal length, and turn into a list of lists, the first containing all the first elements of the original lists, the second the second, and so on.

let $zipn\ l\ =$
  let rec $zipn\_inner\ r\ =$ **function**
    $|\ []\ |\ []::\_\ \rightarrow\ rev\ r$
    $|\ l\ \rightarrow\ zipn\_inner\ (heads\ l :: r)\ (tails\ l)$
  in
    $zipn\_inner\ []\ l$

Remove the second, fourth etc elements from a list, saving the last element (if of even length) e.g $drop\_evens\ [1;2;3;4;5;6]\ is\ [1;3;5;6]$.

let $drop\_evens\ l\ =$
  let rec $drop\_evens\_inner\ r\ =$ **function**
    $|\ h :: \_ :: h'' :: t\ \rightarrow\ drop\_evens\_inner\ (h :: r)\ (h'' :: t)$
    $|\ h :: h' :: []\ \rightarrow\ rev\ (h' :: h :: r)$
    $|\ [x]\ \rightarrow\ rev\ (x :: r)$
    $|\ \_\ \rightarrow\ rev\ r$
  in
    $drop\_evens\_inner\ []\ l$

Same, but don't save the last even one.

let $really\_drop\_evens\ l\ =$
  let rec $really\_drop\_evens\_inner\ r\ =$ **function**
    $|\ []\ \rightarrow\ rev\ r$
    $|\ [h]\ \rightarrow\ really\_drop\_evens\_inner\ (h :: r)\ []$
    $|\ h :: h' :: more\ \rightarrow\ really\_drop\_evens\_inner\ (h :: r)\ more$
  in
    $really\_drop\_evens\_inner\ []\ l$

Remove the first, third etc. The last odd element is not saved. e.g $drop\_odds\ [1;2;3;4;5;6;7]\ is\ [2;4;6]$.

let $drop\_odds\ l\ =$
  let rec $drop\_odds\_inner\ r\ =$ **function**
    $|\ \_ :: h' :: t\ \rightarrow\ drop\_odds\_inner\ (h' :: r)\ t$
    $|\ \_\ \rightarrow\ rev\ r$
  in
    $drop\_odds\_inner\ []\ l$

tl but silent failure.

let $tail\_no\_fail\ =$ **function**
  $|\ []\ \rightarrow\ []$
  $|\ \_ :: t\ \rightarrow\ t$

Couple the elements of a list l using function f. For instance, $couple\ (+)\ [[1;\ 3;\ 5]]$ $\Longrightarrow$ [4; 8]. The two elements are applied to f in the order in which they appear in the input list.

let $couple\ f\ l\ =$
  let rec $couple\_inner\ r\ f\ =$ **function**
    $|\ x :: x' :: xs\ \rightarrow\ couple\_inner\ (f\ x\ x' :: r)\ f\ (x' :: xs)$
    $|\ \_\ \rightarrow\ rev\ r$
  in
    $couple\_inner\ []\ f\ l$

As above, but an extra function g is applied to any last (odd) element.

```
let couple_ext f g l =
  let rec couple_ext_inner r f g = function
    | x :: x′ :: xs → couple_ext_inner (f x x′ :: r) f g (x′ :: xs)
    | x :: [] → couple_ext_inner (g x :: r) f g []
    | [] → rev r
  in
    couple_ext_inner [] f g l
```

Apply *couple* repeatedly until only one element remains. Return that element.

```
let rec couple_reduce f = function
  | [] → raise (Invalid_argument "Utility.couple_reduce")
  | [a] → a
  | l → couple_reduce f (couple f l)
```

A similar function to *couple*, but the coupling is non-overlapping.

```
let pair f l =
  let rec pair_inner r f = function
    | [] → rev r
    | [a] → pair_inner (a :: r) f []
    | a :: b :: t → pair_inner (f a b :: r) f t
  in
    pair_inner [] f l
```

A version of *pair* which adds a unary function for the singleton, much like *couple_ext*.

```
let rec pair_ext f g l =
  let rec pair_ext_inner r f g = function
    | [] → rev r
    | [a] → pair_ext_inner (g a :: r) f g []
    | a :: b :: t → pair_ext_inner (f a b :: r) f g t
  in
    pair_ext_inner [] f g l
```

As *couple_reduce* is to *couple*, so this is to *pair*.

```
let rec pair_reduce f = function
  | [] → raise (Invalid_argument "Utility.pair_reduce")
  | [a] → a
  | l → pair_reduce f (pair f l)
```

List.*filter* has a confusing name, so we define *keep* and *lose* to avoid error.

```
let keep = List.filter

let rec lose_inner prev p = function
  | [] → rev prev
  | h :: t →
    if p h
      then lose_inner prev p t
      else lose_inner (h :: prev) p t
```

```
let lose p = lose_inner [] p
```

Make a list of length n with each element equal to x.

```
let many x n =
  Array.to_list (Array.make n x)
```

A version where we need to apply unit each time, for instance when producing a list of random numbers. Result is ordered.

```
let manyunique f n =
  let rec manyunique_inner r f n =
    if n = 0
      then rev r
      else manyunique_inner (f () :: r) f (n − 1)
  in
    manyunique_inner [] f n
```

Take n elements from the front of a list l, returning them in order.

```
let take l n =
  if n < 0 then raise (Invalid_argument "Utility.take") else
  let rec take_inner r l n =
    if n = 0 then rev r else
      match l with
      | [] → raise (Invalid_argument "Utility.take")
      | h :: t → take_inner (h :: r) t (n − 1)
  in
    take_inner [] l n

let take′ n l = take l n
```

Take from the list l while the predicate p is true.

```
let takewhile p l =
  let rec takewhile_inner r p l =
    match l with
    | [] → rev r
    | h :: t → if p h then takewhile_inner (h :: r) p t else rev r
  in
    takewhile_inner [] p l
```

Drop n elements from the front of a list, returning the remainder in order.

```
let rec drop_inner n = function
  | [] → raise (Invalid_argument "drop")
  | _ :: t → if n = 1 then t else drop_inner (n − 1) t

let drop l n =
  if n < 0 then raise (Invalid_argument "drop") else
  if n = 0 then l else
    drop_inner n l

let drop′ n l = drop l n

let rec dropwhile p = function
  | [] → []
  | h :: t → if p h then dropwhile p t else (h :: t)
```

Split a list l into two parts, the first part containing n elements.

```
let cleave l n =
  let rec cleave_inner l left n =
    if n = 0 then rev left, l else
      match l with
      | [] → raise (Invalid_argument "cleave:  not enough elements")
      | _ → cleave_inner (tl l) (hd l :: left) (n − 1)
  in
    if n < 0
      then raise (Invalid_argument "cleave:  negative argument")
      else cleave_inner l [] n
```

Returns elements for which p is true, until one is not, paired with the remaining list. The same as *takewhile p l*, *dropwhile p l*, but requiring only one pass over the list.

```
let cleavewhile p l =
  let rec cleavewhile_inner p l elts =
    match l with
    | [] → rev elts, []
    | e :: es →
        if p e
          then cleavewhile_inner p es (e :: elts)
          else rev elts, l
  in
    cleavewhile_inner p l []

let cleavewhile_unordered p l =
  let rec cleavewhile_unordered_inner p l elts =
    match l with
    | [] → elts, []
    | e :: es →
        if p e
          then cleavewhile_unordered_inner p es (e :: elts)
          else elts, l
  in
    cleavewhile_unordered_inner p l []
```

Isolate a central section of a list, from the first element after the element for which predicate p is true, to the element before p' is first true.

```
let isolate p p' l =
  let _, during_and_after = cleavewhile (notpred p) l in
    match during_and_after with
    | [] → []
    | _ :: t → fst (cleavewhile (notpred p') t)
```

Collate a list into a list of lists based upon a comparison function by which it has already been sorted. e.g *collate* $[1;\ 2;\ 2;\ 3;\ 3]$ calculates [[1]; [2;2]; [3;3]].

```
let collate cmp l =
  let rec collate_inner prev = function
    | [] → rev prev
    | h :: t →
        let x, y = cleavewhile (fun a → cmp h a = 0) (h :: t) in
          collate_inner (x :: prev) y
  in
    collate_inner [] l
```

Split a list into some lists of length n (and possibly a final one of length $< n$).

```
let splitinto n l =
  let rec splitinto_inner a n l =
    if l = [] then rev a else
      if length l < n then rev (l :: a) else
        let h, t = cleave l n in
          splitinto_inner (h :: a) n t
  in
    splitinto_inner [] n l
```

Split a list l at the given points. Point 1 means after the first element.

```
let rec splitat_inner prev l = function
  | [] → begin match l with [] → rev prev | _ → rev (l :: prev) end
  | h :: t →
      let this, rest = cleave l h in
        splitat_inner (this :: prev) rest t

let splitat points l =
  splitat_inner [] l (couple (fun a b → b − a) (0 :: points))
```

Select the nth element in a list (first is element 1)

```
let select n l =
  try hd (drop l (n − 1)) with
    Invalid_argument "drop" | Failure "hd"→ raise (Invalid_argument "select")
```

Simple list utilities.

```
let isnull = function [] → true | _ → false

let notnull = function [] → false | _ → true
```

Find the last element of a list.

```
let rec last = function
  | [] → raise (Invalid_argument "Utility.last")
  | x :: [] → x
  | _ :: xs → last xs
```

Produce a list containing all but the last element of a list

```
let all_but_last = function
  | [] | [_] → []
  | l → rev (tl (rev l))
```

Find the first and last element of a list. If the list has one element, that is returned twice.

let $extremes$ = **function**
  | [] → $raise$ (Invalid_argument "Utility.extremes")
  | x :: [] → x, x
  | x :: xs → x, $last\ xs$

Return the first, middle and last elements of a list which has length at least two.

let $extremes\_and\_middle$ = **function**
  | [] | [_] →
      $raise$ (Invalid_argument "extremes_and_middle")
  | h :: t →
      let m, l = $cleave\ t\ (length\ t - 1)$ **in**
        h, m, $hd\ l$

1.8 References

Set a boolean reference.

let $set\ r$ =
  r := **true**

Clear a boolean reference.

let $clear\ r$ =
  r := **false**

Change the value of a boolean reference.

let $flip\ r$ =
  $r := \neg\, !r$

Increment and decrement integer references r by an integer n.

let (+=) $r\ n$ =
  $r := !r + n$

let (−=) $r\ n$ =
  $r := !r - n$

let (/=) $r\ n$ =
  $r := !r\ /\ n$

let (∗=) $r\ n$ =
  $r := !r \times n$

Similar functions on floating-point references.

let (+.=) $r\ n$ =
  $r := !r +. n$

let (−.=) $r\ n$ =
  $r := !r -. n$

let (/.=) $r\ n$ =
  $r := !r\ /.\ n$

let (∗.=) $r\ n$ =
  $r := !r *. n$

1.9 Vectors and geometry

Vectors in two dimensions.

type *vector* = *float* × *float*

Make a vector from a point $(x0,\ y0)$ to a point $(x1,\ y1)$.

let *mkvector* $(x0,\ y0)\ (x1,\ y1)\ =\ x1\ -.\ x0,\ y1\ -.\ y0$

Invert a vector.

let *invert* $(a,\ b)$ = ~-.a, ~-.b

Offset a point $(px,\ py)$ by a vector $(x,\ y)$.

let *offset_point* $(x,\ y)\ (px,\ py)\ =\ px\ +.\ x,\ py\ +.\ y$

Find the vector $\pi/2$ anticlockwise from the given one.

let *perpendicular* $(a,\ b)$ = ~-.b, a

Find the length of a vector.

let *sqr* $x\ =\ x\ *.\ x$

let *veclength* $(x,\ y)$ =
  sqrt $(sqr\ x\ +.\ sqr\ y)$

Scale a vector to a length l.

let *scalevectolength* $l\ (a,\ b)$ =
  let *currentlength* = *veclength* $(a,\ b)$ **in**
    if *currentlength* = 0. **then** $(a,\ b)$ **else**
      let *factor* = l /. *currentlength* **in**
        $a\ *.\ factor,\ b\ *.\ factor$

Make a unit vector from s to e

let *mkunitvector* $s\ e$ =
  scalevectolength 1. (*mkvector* $s\ e$)

Find the point equidistant between two others.

let *between* $(x,\ y)\ (x',\ y')$ =
  $(x\ +.\ x')\ /.\ 2.,\ (y\ +.\ y')\ /.\ 2.$

The cartesian distance between two points.

let *distance_between* $(px,\ py)\ (px',\ py')$ =
  sqrt $(sqr\ (px\ -.\ px')\ +.\ sqr\ (py'\ -.\ py))$

1.10 Functions on the integers

The largest power of two by which n is exactly divisible.

let *largest_pow2_divisible* n =
  let rec s *test* n =
    if n **mod** *test* = 0 **then** s (*test* × 2) n
    else *test* / 2
  in
    $s\ 1\ n$

Find the largest power of two smaller or equal to an integer t.

```
let pow2lt t =
  let rec pow2lt_i target current =
    if current × 2 > target
      then current
      else pow2lt_i target (current × 2)
  in
    pow2lt_i t 1
```

Find the largest power of two greater or equal to an integer t.

```
let pow2gt t =
  let lt = pow2lt t in
    if lt = t then t else lt × 2
```

Find the integer base two logarithm of a number.

```
let log2of t =
  let rec log2of_i target num =
    if num × 2 > target
      then 0
      else let n = log2of_i target (num × 2) in n + 1
  in
    log2of_i t 1
```

Integer compare function — saves the cost of polymorphic comparisons.

```
let compare_i (a : int) b =
  if a < b then −1 else if a > b then 1 else 0
```

Reverse comparison

```
let rev_compare a b =
  −(compare a b)
```

The integer range between $[s..e]$ inclusive.

```
let ilist s e =
  if e < s then raise (Invalid_argument "Utility.ilist") else
    let nums = ref [] in
      let rec ilist s e =
        if s = e
          then nums =| e
          else (nums =| s; ilist (s + 1) e)
      in
        ilist s e;
        rev !nums
```

Same, but return null list for ilist x x rather than x

```
let ilist_null s e =
  if s = e then [] else ilist s e
```

Same, but upon failure just return null.

```
let ilist_fail_null s e =
  if s > e then [] else ilist_null s e
```

A common case: Make indexes for a (non-null) list

```
let indx l =
  ilist 1 (length l)

let indx0 l =
  ilist 0 (length l - 1)

let indxn n l =
  ilist n (n + length l - 1)
```

1.11 Miscellaneous functions

Even/odd predicates. Work for positive, negative and zero values.

```
let even x = x mod 2 = 0

let odd = notpred even
```

Exclusive Or of a and b.

```
let ( |&| ) a b =
  (a ∨ b) ∧ ¬ (a ∧ b)
```

The identity function.

```
let ident x = x
```

An array analog of List.*iter2*.

```
let array_iter2 f a b =
  if Array.length a = Array.length b then
    if Array.length a = 0 then () else
      for x = 0 to (Array.length a) - 1 do
        f (Array.get a x) (Array.get b x)
      done
  else
    raise (Invalid_argument "Utility.array_iter2")

let array_map2 f a b =
  if Array.length a = Array.length b then
    Array.init (Array.length a) (function i → f a.(i) b.(i))
  else
    raise (Invalid_argument "Utility.array_map2")
```

Find the number of bytes in n megabytes.

```
let megabytes n = n × 1024 × 1024
```

Some simple functions for working with the *option* type.

```
let some = function None → false | _ → true

let none = function None → true | _ → false

let unopt = function
  | Some x → x
  | None → failwith "unopt"
```

let $option\_map\ f\ l\ =$
  $map\ unopt\ (lose\ (eq\ \mathsf{None})\ (map\ f\ l))$

Integer-specialised minimum and maximum functions for speed, overriding *Pervasives.min* and *Pervasives.max*.

let $min\ (a\ :\ int)\ b\ =$ **if** $a\ <\ b$ **then** a **else** b
and $max\ (a\ :\ int)\ b\ =$ **if** $a\ >\ b$ **then** a **else** b

Floating point ones.

let $fmin\ (a\ :\ float)\ b\ =$ **if** $a\ <\ b$ **then** a **else** b
and $fmax\ (a\ :\ float)\ b\ =$ **if** $a\ >\ b$ **then** a **else** b

let $fabs\ x\ =\ abs\_float\ x$

The union of two rectangles, each defined by its minimum and maximum coordinates

let $box\_union\ (xmin0,\ xmax0,\ ymin0,\ ymax0)\ (xmin1,\ xmax1,\ ymin1,\ ymax1)\ =$
  $min\ xmin0\ xmin1,\ max\ xmax0\ xmax1,\ min\ ymin0\ ymin1,\ max\ ymax0\ ymax1$

The union of two rectangles, each defined by its minimum and maximum coordinates

let $box\_union\_float\ (xmin0,\ xmax0,\ ymin0,\ ymax0)\ (xmin1,\ xmax1,\ ymin1,\ ymax1)\ =$
  $fmin\ xmin0\ xmin1,\ fmax\ xmax0\ xmax1,\ fmin\ ymin0\ ymin1,\ fmax\ ymax0\ ymax1$

The intersection rectangle of two rectangles defined by integers. $x0,\ y0$ etc refer to the top left, $x1,\ y1$ etc. to the bottom right.

let $box\_overlap\ ax0\ ay0\ ax1\ ay1\ bx0\ by0\ bx1\ by1\ =$
  if $ax0\ >\ bx1\ \lor\ ay0\ >\ by1\ \lor\ ax1\ <\ bx0\ \lor\ ay1\ <\ by0$
    then None
    else Some $(max\ ax0\ bx0,\ max\ ay0\ by0,\ min\ ax1\ bx1,\ min\ ay1\ by1)$

The same for floating point coordinates.

let $box\_overlap\_float\ ax0\ ay0\ ax1\ ay1\ bx0\ by0\ bx1\ by1\ =$
  if $ax0\ >\ bx1\ \lor\ ay0\ >\ by1\ \lor\ ax1\ <\ bx0\ \lor\ ay1\ <\ by0$
    then None
    else Some $(fmax\ ax0\ bx0,\ fmax\ ay0\ by0,\ fmin\ ax1\ bx1,\ fmin\ ay1\ by1)$

Apply a function f n times to initial argument arg.

let rec $applyn\ f\ n\ arg\ =$
  if $n\ =\ 0$ **then** arg **else** $applyn\ f\ (n\ -\ 1)\ (f\ arg)$

The type of binary trees.

type $\alpha\ tree\ =$ Lf | Br **of** $\alpha\ \times\ \alpha\ tree\ \times\ \alpha\ tree$

Define π.

let $pi\ =\ 4.\ *.\ atan\ 1.$

Define $\sqrt{2}$.

let $root2\ =\ sqrt\ 2.$

Radians of degrees.

```
let rad_of_deg a = a *. pi /. 180.
```

Degrees of radians.

```
let deg_of_rad a = a *. 180. /. pi
```

Constant boolean predicates

```
let always _ = true
and never _ = false
```

A null hash table.

```
let null_hash () =
  Hashtbl.create 0
```

IF-OCAML

```
let tryfind table k =
  try
    Some (Hashtbl.find table k)
  with
    Not_found → None
(*ENDIF-OCAML*)
```

Extract all (key, value) pairs from a hash table.

```
let list_of_hashtbl t =
  let contents = ref [] in
    Hashtbl.iter
      (fun k v → contents =| (k, v))
      t;
  !contents
```

Build a hashtable from a dictionary

```
let hashtable_of_dictionary pairs =
  let table = Hashtbl.create (length pairs × 2) in
    iter (fun (k, v) → Hashtbl.add table k v) pairs;
    table
```

Round a number.

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

Render a float normal by replacing anything abnormal by 0.

```
let safe_float f =
  match classify_float f with
  | FP_nan | FP_infinite | FP_zero (*IF-OCAML*)| FP_subnormal(*ENDIF-OCAML*) → 0.
  | _ → f
```

Build a tuple

```
let tuple x y = x, y
```

Make a unit function.

```
let mkunit f x = fun () → f x
```

Swap two elements of an array.

```
let swap a i j =
  let t = a.(i) in
    a.(i) ← a.(j);
    a.(j) ← t
```

Print floats, integers or int32 values with spaces between them.

```
let print_floats fs =
  iter (fun x → print_float x; print_string " ") fs;
  print_newline ()

let print_ints is =
  iter (fun x → print_int x; print_string " ") is;
  print_newline ()

let print_int32s is =
  iter (fun x → Printf.printf "%li " x) is;
  print_newline ()

let digest =
  (*IF-OCAML*)
  Digest.string
  (*ENDIF-OCAML*)

let slash =
  (*IF-OCAML*)
  match Sys.os_type with
  | "Win32" → "\\"
  | _ → "/"
  (*ENDIF-OCAML*)

let leafnames_of_dir d =
  (*IF-OCAML*)
  Array.to_list (Sys.readdir d)
  (*ENDIF-OCAML*)
```

2 Module PDFIo
General Input and Output

open Utility

We use 64-bit sized files as standard.
IF-OCAML

open LargeFile
($*$ENDIF-OCAML$*$)

2.1 Defining and creating input and output functions

IF-OCAML

type $pos\ =\ int64$
let $pos\_succ\ =\ i64succ$
let $pos\_pred\ =\ i64pred$
let $pos\_max\ =\ i64max$
let $possub\ =\ i64sub$
let $posadd\ =\ i64add$
let $posofi\ x\ =\ i64ofi\ x$
let $postoi\ x\ =\ i64toi\ x$
let $postoi64\ x\ =\ x$
let $posofi64\ x\ =\ x$
($*$ENDIF-OCAML$*$)

let $no\_more\ =\ -1$

▷ A general type for input functions. This allows paramaterization over channels, strings, bigarrays etc.

type $input\ =$
$\{pos\_in\ :$ unit $\rightarrow$ $pos;$
$seek\_in\ :\ pos\ \rightarrow$ unit;
$input\_char\ :$ unit $\rightarrow$ char option;
$input\_byte\ :$ unit $\rightarrow$ int;
$in\_channel\_length\ :$ unit $\rightarrow$ $pos;$
$set\_offset\ :\ pos\ \rightarrow$ unit$\}$

▷ A general type for output functions, allowing parameterisation as above.

```
type output =
  {pos_out : unit → pos;
   seek_out : pos → unit;
   output_char : char → unit;
   output_byte : int → unit;
   out_channel_length : unit → pos}
```

▷ Create input functions from a channel.

```
let input_of_channel ch =
  let offset = ref (posofi 0) in
    {pos_in =
       (fun () → possub (pos_in ch) !offset);
     seek_in =
       (fun x → seek_in ch (posadd x !offset));
     input_char =
       (fun () →
          try Some (input_char ch) with End_of_file → dpr "3A"; None);
     input_byte =
       (fun () →
          try input_byte ch with End_of_file →
            dpr "3B"; no_more);
     in_channel_length =
       (fun () → in_channel_length ch);
     set_offset =
       (fun o → offset := o)
    }
```

▷ Create input functions from a Utility.*stream*.

```
let input_of_stream s =
  let input_int () =
    if s.pos > stream_size s.data − 1
      then
        begin
          s.pos ← s.pos + 1;
          no_more
        end
      else
        begin
          s.pos ← s.pos + 1;
          sget s.data (s.pos − 1)
        end
  in
    {pos_in =
       (fun () → posofi s.pos);
     seek_in =
       (fun p →
          s.pos ← postoi p);
     input_char =
       (fun () →
         match input_int () with x when x = no_more → None | s →
```

```
Some (char_of_int s));
      input_byte =
        input_int;
      in_channel_length =
        (fun () → posofi (stream_size s.data));
      set_offset =
        (* FIXME. *)
        (fun _ → ())
    }
```

▷ Create input functions from a Utility.*bytestream*.

```
let input_of_bytestream b =
  input_of_stream {pos = 0; data = b}

let input_of_string s =
  input_of_bytestream (bytestream_of_string s)
```

▷ Output functions over channels

```
let output_of_channel ch =
  {pos_out = (fun () → pos_out ch);
   seek_out = seek_out ch;
   output_char = (fun c → output_byte ch (int_of_char c));
   output_byte = output_byte ch;
   out_channel_length = (fun () → out_channel_length ch)}
```

▷ Output functions over streams. If data is written past the end of a stream, we extend the stream to that point plus one-third of that (new) size. Note that this has an implication upon mixing reading and writing: the stream will have junk in the extended section and will be longer than that which has been written.

```
let output_of_stream s =
  let highest_written = ref (posofi 0) in
    let output_int i =
      if s.pos > stream_size s.data − 1
        then
          let newstream = mkstream (s.pos × 2 − s.pos / 2) in
            for x = 0 to stream_size s.data − 1 do
              sset newstream x (sget s.data x)
            done;
            sset newstream s.pos i;
            highest_written := pos_max !highest_written (posofi s.pos);
            s.pos ← s.pos + 1;
            s.data ← newstream
        else
          begin
            highest_written := pos_max !highest_written (posofi s.pos);
            sset s.data s.pos i;
            s.pos ← s.pos + 1
          end
    in
        {pos_out =
          (fun () → posofi s.pos);
```

```
seek_out =
  (fun p → s.pos ← postoi p);
output_char =
  (fun c → output_int (int_of_char c));
output_byte =
  output_int;
out_channel_length =
  (fun () → pos_succ !highest_written)}
```

2.2 Utility functions

▷ Nudge forward one character.

```
let nudge i =
  ignore (i.input_byte ())
```

▷ Read one character behind the current position, and reposition ourselves on that character.

```
let read_char_back i =
  let pos = i.pos_in () in
    i.seek_in (pos_pred pos);
    let chr = i.input_char () in
      i.seek_in (pos_pred pos);
      chr
```

▷ Go back one character in a file.

```
let rewind i =
  i.seek_in (pos_pred (i.pos_in ()))

let rewind2 i =
  i.seek_in (possub (i.pos_in ()) (posofi 2))

let rewind3 i =
  i.seek_in (possub (i.pos_in ()) (posofi 3))
```

▷ Read a character, leaving the position unchanged.

```
let peek_char i =
  let r = i.input_char () in
    rewind i; r
```

▷ Read a byte, leaving the position unchanged.

```
let peek_byte i =
  let r = i.input_byte () in
    rewind i; r
```

▷ Output a string.

```
let output_string o s =
  String.iter o.output_char s
```

▷ Make a bytestream of an input channel.

```
let bytestream_of_input_channel ch =
  let fi = input_of_channel ch in
    let size = postoi (fi.in_channel_length ()) in
      let s = mkstream size in
        for x = 1 to size do
          match fi.input_byte () with
          | b when b = no_more → failwith "channel length inconsistent"
          | b → sset s (x − 1) b
        done;
        s
```

▷ Save a bytestream to a channel.

```
let bytestream_to_output_channel ch data =
  for x = 1 to stream_size data do
    output_byte ch (sget data (x − 1))
  done
```

Like Pervasives.*read_line*

```
let read_line i =
  (∗ Raise EndOfInput if at end ∗)
  begin match i.input_byte () with
  | x when x = no_more → dpr "O"; raise End_of_file;
  | _ → ()
  end;
  rewind i;
  (∗ Read characters whilst ¡¿ newline or until end of input ∗)
  let rec read_chars prev =
    match i.input_byte () with
    | x when x = no_more → rev prev
    | x when char_of_int x = '\n' → rev ('\n' :: prev)
    | x → read_chars (char_of_int x :: prev)
  in
    implode (read_chars [])
```

2.3 Reading MSB-first Bit streams

▷ The type of bit (MSB first) streams.

```
type bitstream =
  {input : input; (∗ The input from which bits are taken. It is advanced a byte
at a time ∗)
   mutable currbyte : int; (∗ Current byte value from input ∗)
   mutable bit : int; (∗ Mask for getting the next bit (128, 64,... 2, 1 or 0 =
none left) ∗)
   mutable bitsread : int (∗ A count of the number of bits read since inception.
Debug use only ∗)}
```

▷ Make a *bitstream* from an *input*.

```
let bitstream_of_input i =
  {currbyte = 0;
   bit = 0;
   bitsread = 0;
   input = i}
```

For debug only....

```
let input_in_bitstream b =
  b.input
```

▷ Get a single bit.

```
let rec getbit b =
  if b.bit = 0 then
    begin
      b.currbyte ←
        begin match b.input.input_byte () with
        | x when x = no_more → dpr "P"; raise End_of_file
        | x → x
        end;
      b.bit ← 128;
      getbit b
    end
  else
    let r = b.currbyte land b.bit > 0 in
      b.bitsread ← b.bitsread + 1;
      b.bit ← b.bit / 2;
      r
```

▷ Get a bit as an integer, set = 1, unset = 0

```
let getbitint i =
  if getbit i then 1 else 0
```

▷ Align on a byte boundary.

```
let align b =
  if b.bit > 0 then b.bitsread ← (b.bitsread / 8 + 1) × 8;
  b.bit ← 0
```

Get n (up to 32) bits from b, returned as an *int32*, taken highest bit first. Getting 0 bits gets the value 0..

SPEED: *Far too slow*

```
let char_of_bool = function true → '1' | false → '0'

let getval_32 b n =
  if n < 0 then raise (Invalid_argument "Io.getval_32") else
    if n = 0 then 0l else
      let bits = manyunique (mkunit getbit b) n in
        Int32.of_string ("0b" ^ implode (map char_of_bool bits))
```

2.4 Writing MSB-first bit streams

The type: A current byte, the position in the byte (0 = nothing in it, 7 = almost full), and the list (in reverse order) of full bytes so far

```
type bitstream_write =
  {mutable wcurrbyte : int;
   mutable wbit : int;
   mutable bytes : int list}

let make_write_bitstream () =
  {wcurrbyte = 0;
   wbit = 0;
   bytes = []}

let copy_write_bitstream b =
  let b' = make_write_bitstream () in
    b'.wcurrbyte <- b.wcurrbyte;
    b'.wbit <- b.wbit;
    b'.bytes <- b.bytes;
    b'

let print_bitstream b =
  Printf.printf "wcurrbyte = %i, wbit = %i, %i bytes output\n"
  b.wcurrbyte b.wbit (length b.bytes)
```

Put a single bit into bitstream b

```
let putbit b bit =
  assert (bit = 0 ∨ bit = 1);
  match b.wbit with
  | 7 →
      b.bytes <- (b.wcurrbyte lor bit) :: b.bytes;
      b.wbit <- 0;
      b.wcurrbyte <- 0
  | _ →
      b.wbit <- b.wbit + 1;
      b.wcurrbyte <- b.wcurrbyte lor (bit lsl (8 - b.wbit))

let putbool b bit =
  putbit b ((function false → 0 | true → 1) bit)
```

Put a multi-bit value n of bits bs (given as an *int32*) into bitstream b.

```
let rec putval b bs n =
  if bs < 0 ∨ bs > 32 then raise (Invalid_argument "putval");
  match bs with
  | 0 → ()
  | _ →
      let bit =
        if land32 n (i32ofi (1 lsl (bs - 1))) > 0l then 1 else 0
      in
        putbit b bit;
        putval b (bs - 1) n
```

Align on a byte boundary, writing zeroes.

```
let align_write b =
  if b.wbit > 0 then
    for x = 1 to 8 - b.wbit do
      putbit b 0
    done
```

Get the output out.

```
let bytestream_of_write_bitstream b =
  align_write b;
  bytestream_of_list (rev b.bytes)
```

Return a list of booleans, representing (in order) the bits

```
let bits_of_write_bitstream b =
  let numbits = length b.bytes × 8 + b.wbit
  and bytestream = bytestream_of_write_bitstream b
  and bits = ref [] in
    let bitstream = bitstream_of_input (input_of_bytestream bytestream) in
      for x = 1 to numbits do
        bits =| getbit bitstream
      done;
      rev !bits
```

Same, but from a list

```
let join_write_bitstreams ss =
  let c = make_write_bitstream () in
    iter
      (putbool c)
      (flatten (map bits_of_write_bitstream ss));
    c
```

Append b to a. Inputs unaltered.

```
let write_bitstream_append a b =
  join_write_bitstreams [a; b]
```

Same, but align at the join.

```
let write_bitstream_append_aligned a b =
  let c = copy_write_bitstream a in
    align_write c;
    write_bitstream_append c b
```

3 Module Io
IO Support for zlib


```
open Utility

type input =
  {mutable in_read : unit → char;
   mutable in_input : Istring.t → int → int → int;
   mutable in_close : unit → unit}

exception No_more_input
exception Input_closed
```

API

```
let create_in read input close =
  {in_read = read;
   in_input = input;
   in_close = close}

let nread i n =
        if n < 0 then invalid_arg "IO.nread";
        if n = 0 then
                Istring.create 0
        else
        let s = Istring.create n in
        let l = ref n in
        let p = ref 0 in
        try
```

```
            while !l > 0 do
                let r = i.in_input s !p !l in
                if r = 0 then raise No_more_input;
                p := !p + r;
                l := !l - r;
            done;
            s
        with
            No_more_input as e →
                dpr "2A";
                if !p = 0 then raise e;
                Istring.sub s 0 !p

let close_in i =
    let f _ = raise Input_closed in
    i.in_close();
    i.in_read ← f;
    i.in_input ← f;
    i.in_close ← f

let read_all i =
    let maxlen = 1024 in
    let str = ref [] in
    let pos = ref 0 in
    let rec loop() =
        let s = nread i maxlen in
        str := (s,!pos) :: !str;
        pos := !pos + Istring.length s;
        loop()
    in
    try
        loop()
    with
        No_more_input →
            dpr "2B";
            let buf = Istring.create !pos in
            List.iter (fun (s, p) →
                Istring.blit s 0 buf p (Istring.length s)
            ) !str;
            buf
```

BINARY APIs

```
exception Overflow of string

let read_byte i = int_of_char (i.in_read())

let read_ui16 i =
    let ch1 = read_byte i in
    let ch2 = read_byte i in
    ch1 lor (ch2 lsl 8)
```

Part II

CamlPDF

4 Module PDF
Representing PDF files

This module declares a data type which represents an Adobe PDF document, and defines various simple operations on it.

```
open Utility
open Pdfio
```

4.1 Data Type for Representing PDF Documents

Predicate on characters delimiting entities.

```
let is_delimiter = function
  | '(' | ')' | '<' | '>' | '[' | ']' | '{' | '}' | '%' | '/' → true
  | _ → false
```

▷ Streams of binary data, byte-addressable, can either be in memory (Got) or still in an input channel (ToGet).

```
type stream =
  | Got of bytestream
  | ToGet of input × int64 × int64          (* input, position, length *)
```

▷ Type for individual PDF objects. A Name includes the initial '/'. A Stream consists of a reference to a pair of the stream dictionary (another *pdfobject*) and a *stream*. Thus a *pdfobject* is technically mutable. However, at the user level, it is intended to be immutable: changes should be limited to encoding and decoding of the stream.

Note that pdfobjects are not always amenable to polymorphic equality testing, since the Io.*input* in the ToGet part of a *stream* contains functional values.

```
type pdfobject =
  | Null
  | Boolean of bool
  | Integer of int
  | Real of float
  | String of string
  | Name of string
  | Array of pdfobject list
```

```
  | Dictionary of (string × pdfobject) list
  | Stream of (pdfobject × stream) ref
  | Indirect of int
```

IF-OCAML
Pdf objects are stored in an efficient map structure.

```
module PdfObjMap =
  Map.Make
    (struct
       type t = int
       let compare = compare
     end)

let pdfobjmap_find = PdfObjMap.find
let pdfobjmap_mapi = PdfObjMap.mapi
let pdfobjmap_iter = PdfObjMap.iter
let pdfobjmap_remove = PdfObjMap.remove
let pdfobjmap_add = PdfObjMap.add
let pdfobjmap_empty = PdfObjMap.empty
```

An object is either lexed, or needs to be lexed from a position in the input.

```
type objectdata =
  | Parsed of pdfobject
  | ToParse
```

We hold the maximum object number in use, $maxobjnum$ to allow easy production of new keys for the map.

```
type pdfobjects =
  {mutable maxobjnum : int;
   mutable parse : (PdfObjMap.key → pdfobject) option;
   mutable pdfobjects : (objectdata ref × int) PdfObjMap.t}          (* int is
generation *)
(*ENDIF-OCAML*)
```

▷ PDF Document. The major and minor version numbers, the root object number, the list of objects and the trailer dictionary.

This represents the contents of a PDF file's user objects (object streams and other mechanisms involved only in reading and writing are abstracted away).

```
type pdfdoc =
  {mutable major : int;
   mutable minor : int;
   mutable root : int;
   mutable objects : pdfobjects;
   mutable trailerdict : pdfobject}
```

▷ The null PDF document.

```
let empty () =
  {major = 1;
   minor = 0;
   root = 0;
   objects = {maxobjnum = 0; parse = None; pdfobjects = pdfobjmap_empty};
   trailerdict = Dictionary []}
```

▷ General exception for low-level errors.

```
exception PDFError of string
```

4.2 Utility functions

▷ Predicate on those characters considered whitespace in PDF files.

```
let is_whitespace = function
  | '\000' | '\009' | '\010' | '\012' | ' ' | '\013' → true
  | _ → false
```

▷ Get a stream from disk if it hasn't already been got.

```
let getstream = function
  | Stream ({contents = (d, ToGet (i, o, l))} as stream) →
      if l = 0L then stream := (d, Got (mkstream 0)) else
        let s = mkstream (i64toi l) in
          begin try
            (*IF-OCAML*)i.seek_in o; (*ENDIF-OCAML*)
                          for c = 0 to i64toi l − 1 do
              match i.input_byte () with
              | b when b = Pdfio.no_more → dpr "H"; raise End_of_file
              | b → sset s c b
            done;
            stream := (d, Got s)
          with
            End_of_file →
              raise (PDFError "Pdf.getstream:  can't read stream.")
          end
  | Stream _ → ()
  | _ → raise (PDFError "Pdf.getstream:  not a stream")

let recurse_array (f : pdfobject → pdfobject) elts =
  Array (map f elts)
```

▷ Similarly for dictionaries.

```
let recurse_dict (f : pdfobject → pdfobject) elts =
  let names, objects = split elts in
    let objects' = map f objects in
      Dictionary (combine names objects')
```

▷ Return a float from a PDF number.

```
let getnum = function
  | Real a → a
  | Integer a → float a
  | _ → raise (PDFError "Pdf.getnum:  not a number")
```

▷ Parse a PDF rectangle data structure. Returns min x, min y, max x, max y.

```
let parse_rectangle = function
  | Array [a; b; c; d] →
      begin try
        let x, y, x', y' =
          getnum a, getnum b, getnum c, getnum d
        in
          fmin x x', fmin y y', fmax x x', fmax y y'
      with
        PDFError _ → raise (PDFError "Pdf.parse_rectangle: bad rectangle")
      end
  | _ → raise (PDFError "Pdf.parse_rectangle: not a rectangle")

let change_obj doc i obj =
  match fst (pdfobjmap_find i doc.objects.pdfobjects) with
  | {contents = Parsed _} → assert false
  | {contents = ToParse} as r → r := Parsed obj
```

Parse an object n in document *pdf*, updating the object in the document so it is ready-parsed should it be required again.

```
let parse_lazy pdf n =
  match pdf.objects.parse with
  | None → raise (Assert_failure ("Pdf.parse_lazy", 0, 0))
  | Some f →
      let obj = f n in
        change_obj pdf n obj;
        obj
```

▷ Look up an object. On an error return Pdf.Null

```
let lookup_obj doc i =
  try
    match fst (pdfobjmap_find i doc.objects.pdfobjects) with
    | {contents = Parsed obj} → obj
    | {contents = ToParse} → parse_lazy doc i
  with
    Not_found → dpr "2H"; Null

let catalog_of_pdf pdf =
  try lookup_obj pdf pdf.root with
    Not_found → raise (PDFError "No catalog")
```

▷ Given any pdf document and object, follow indirections to yield a direct object. A hanging indirect is defined as Null.

```
let rec direct pdf = function
  | Indirect i →
      begin try
        match fst (pdfobjmap_find i pdf.objects.pdfobjects) with
        | {contents = Parsed pdfobject} → direct pdf pdfobject
        | {contents = ToParse} → parse_lazy pdf i
      with
        Not_found → dpr "2I"; Null
      end
  | obj → obj
```

▷ Apply a function on Stream objects to all streams in a PDF document. We assume stream dictionaries don't have indirect references to an object which itself contains a stream.

```
let map_stream f pdf =
  let rec map_stream_inner f i = function
    | {contents = Parsed (Stream _ as stream)}, g → ref (Parsed (f stream)), g
    | {contents = Parsed obj}, g → ref (Parsed (obj)), g
    | {contents = ToParse}, g → map_stream_inner f i (ref (Parsed (parse_lazy pdf i)), g)
  in
    let objects' =
      {pdf.objects with
         pdfobjects = pdfobjmap_mapi (map_stream_inner f) pdf.objects.pdfobjects}
    in
      {pdf with objects = objects'}
```

▷ Iterate over a stream.

```
let iter_stream f pdf =
  let rec iter_stream_inner f i = function
    | {contents = Parsed (Stream _ as stream)}, g → f stream
    | {contents = ToParse} as r, g →
        r := Parsed (parse_lazy pdf i);
        iter_stream_inner f i (r, g)
    | _ → ()
  in
    pdfobjmap_iter (iter_stream_inner f) pdf.objects.pdfobjects
```

▷ Lookup a key in a dictionary, following indirect references, returning None on any failure. This works on both plain dictionaries and streams.

```
let lookup_direct pdf key dict =
  match direct pdf dict with
  | Dictionary d | Stream {contents = (Dictionary d, _)} →
      begin match lookup key d with
      | None → None
      | Some o → Some (direct pdf o)
      end
  | _ → None
```

▷ Look up under a key and its alternate. Return the value associated with the key that worked, or None if neither did.

```
let lookup_direct_orelse pdf k k' d =
  match lookup_direct pdf k d with
  | None → lookup_direct pdf k' d
  | result → result
```

▷ Look something up in a dictionary, failing with given exception if not found. We make direct both the dictionary and the result of the lookup. This also allows us to look things up in a stream dictionary transparently.

```
let lookup_exception (exp : exn) pdf key dict =
  let dict' =
    match direct pdf dict with
    | Dictionary d | Stream {contents = Dictionary d, _} → d
    | o → raise (PDFError "not a dictionary")
  in
    match lookup key dict' with
    | None → dpr "G"; raise exp
    | Some v → direct pdf v
```

▷ A specialised one raising PDFError.

```
let lookup_fail text =
  lookup_exception (PDFError text)
```

▷ Parse a matrix.

```
let parse_matrix pdf name dict =
  match lookup_direct pdf name dict with
  | None → Transform.i_matrix
  | Some (Array [a; b; c; d; e; f]) →
      let a = getnum a and b = getnum b and c = getnum c
      and d = getnum d and e = getnum e and f = getnum f in
        {Transform.a = a; Transform.b = b; Transform.c = c;
         Transform.d = d; Transform.e = e; Transform.f = f}
  | _ → raise (PDFError "Malformed matrix")
```

▷ Make a matrix

```
let make_matrix tr =
  Array
    [Real tr.Transform.a; Real tr.Transform.b; Real tr.Transform.c;
     Real tr.Transform.d; Real tr.Transform.e; Real tr.Transform.f]
```

▷ Iterate over the objects in a document, in order of increasing object number.

```
let objiter f doc =
  let f' k v =
    match v with
    | {contents = Parsed obj}, _ → f k obj
    | {contents = ToParse}, _ → f k (parse_lazy doc k)
  in
    pdfobjmap_iter f' doc.objects.pdfobjects
```

▷ Same, but also pass generation number.

```
let objiter_gen f doc =
  let f' k v =
    match v with
    | {contents = Parsed obj}, g → f k g obj
    | {contents = ToParse}, g → f k g (parse_lazy doc k)
  in
    pdfobjmap_iter f' doc.objects.pdfobjects
```

▷ Map on objects.

```
let objmap f doc =
  let f' i = function
    | {contents = Parsed obj}, g → ref (Parsed (f obj)), g
    | {contents = ToParse}, g → ref (Parsed (parse_lazy doc i)), g
  in
    {doc with objects =
       {doc.objects with
          pdfobjects = pdfobjmap_mapi f' doc.objects.pdfobjects}}

let maxobjnum pdf =
  pdf.objects.maxobjnum
```

Return a list of object numbers.

```
let objnumbers pdf =
  let keys = ref [] in
    objiter (fun k _ → keys =| k) pdf;
    rev !keys
```

▷ Cardinality of object set. O(n).

```
let objcard pdf =
  let card = ref 0 in
    objiter (fun _ _ → incr card) pdf;
    !card
```

Remove an object.

```
let removeobj doc o =
  {doc with objects =
    {doc.objects with pdfobjects = pdfobjmap_remove o doc.objects.pdfobjects}}
```

Return a list of (k, v) pairs.

```
let list_of_objs doc =
  let objs = ref [] in
    objiter (fun k v → objs =| (k, Parsed v)) doc;
    !objs
```

▷ Add an object, given an object number.

```
let addobj_given_num doc (num, obj) =
  doc.objects.maxobjnum ← max doc.objects.maxobjnum num;
  doc.objects.pdfobjects ← pdfobjmap_add num (ref (Parsed obj), 0) doc.objects.pdfobjects
```

▷ Add an object. We use the first number larger than the maxobjnum, and update that.

```
let addobj doc obj =
  let num = doc.objects.maxobjnum + 1 in
    addobj_given_num doc (num, obj);
    num
```

Make a objects entry from a list of (number, object) pairs.

```
let objects_of_list parse l =
  let maxobj = ref 0
  and map = ref pdfobjmap_empty in
    iter
      (fun (k, v) ->
                  maxobj := max !maxobj k;
         map := pdfobjmap_add k v !map)
      l;
    {parse = parse; pdfobjects = !map; maxobjnum = !maxobj}
```

Renumber an object given a change table (A hash table mapping old to new numbers).

```
let rec renumber_object_parsed (pdf : pdfdoc) changes obj =
  match obj with
  | Indirect i ->
      let i' =
        match tryfind changes i with
        | Some x -> x
        | None -> i                        (* A dangling indirect is valid. *)
      in
        Indirect i'
  | Array a ->
      recurse_array (renumber_object_parsed pdf changes) a
  | Dictionary d ->
      recurse_dict (renumber_object_parsed pdf changes) d
  | Stream {contents = (p, s)} ->
      Stream {contents = renumber_object_parsed pdf changes p, s}
  | pdfobject -> pdfobject
```

```
let renumber_object pdf changes objnum = function
  | ToParse ->
      renumber_object_parsed pdf changes (parse_lazy pdf objnum)
  | Parsed obj ->
      renumber_object_parsed pdf changes obj
```

Perform all renumberings given by a change table.

```
let renumber change_table pdf =
  let root' =
    match tryfind change_table pdf.root with Some x -> x | None ->
 pdf.root
  and trailerdict' =
    renumber_object pdf change_table 0 (Parsed pdf.trailerdict)
  and objects' =
    let nums, objs = split (list_of_objs pdf) in
      let objs' =
        map2 (renumber_object pdf change_table) nums objs
      and nums' =
        map (function k -> match tryfind change_table k with Some x ->
```

```
x | None → k) nums
      in
        objects_of_list
          pdf.objects.parse
          (combine nums' (map (fun x → ref (Parsed x), 0) objs'))
  in
    {pdf with
     root = root';
     objects = objects';
     trailerdict = trailerdict'}
```

▷ Renumber the objects (including root and trailer dictionary) in a list of pdfs so they are mutually exclusive. We iterate over the key lists to build a list of change tables which are applied to the input PDFs. NOTE: This can't be used on PDFs where the generation numbers still matter (i.e before decryption).

```
let renumber_pdfs pdfs =
  let keylists = map objnumbers pdfs
  and bse = ref 1
  and tables = ref [] in
    iter
      (fun k →
         let length = length k in
           let table = Hashtbl.create length in
             List.iter2 (Hashtbl.add table) k (ilist !bse (!bse + length − 1));
             tables =| table;
             bse += length)
      keylists;
    map2 renumber (rev !tables) pdfs
```

Used for sets of object numbers.
IF-OCAML

```
module RefSet =
  Set.Make
    (struct
       type t = int
       let compare = compare
     end)

let refset_add = RefSet.add
let refset_empty = RefSet.empty
let refset_elements = RefSet.elements
(*ENDIF-OCAML*)
```

Give a list of object numbers referenced in a given *pdfobject*

```
let rec referenced no_follow_entries no_follow_contains pdf found i = function
  | Parsed (Indirect i) →
      if ¬ (RefSet.mem i !found) then
        begin
          let obj =
            try lookup_obj pdf i with
              Not_found → dpr "2M"; Null
```

```
          in
            match obj with
            | Dictionary d →
                if ¬ (mem true (map (mem' no_follow_contains) d)) then
                  begin
                  found := RefSet.add i !found;
                  referenced no_follow_entries no_follow_contains pdf found i (Parsed obj)
                  end
            | _ →
                found := RefSet.add i !found;
                referenced no_follow_entries no_follow_contains pdf found i (Parsed obj)
        end
  | Parsed (Array a) →
      iter
        (referenced no_follow_entries no_follow_contains pdf found i)
        (map (fun x → Parsed x) a)
  | Parsed (Dictionary d) →
      iter
        (referenced no_follow_entries no_follow_contains pdf found i)
        (map
          (fun x → Parsed (snd x))
          (lose (fun (k, _) → mem k no_follow_entries) d))
  | Parsed (Stream s) →
      referenced no_follow_entries no_follow_contains pdf found i (Parsed (fst !s))
  | Parsed _ →
      ()
  | ToParse →
      referenced no_follow_entries no_follow_contains pdf found i (Parsed (parse_lazy pdf i))
```

▷ Remove any unreferenced objects.

```
let remove_unreferenced pdf =
  let found = ref RefSet.empty in
    referenced [] [] pdf found pdf.root (Parsed (lookup_obj pdf pdf.root));
    referenced [] [] pdf found 0 (Parsed pdf.trailerdict);
    found := RefSet.add pdf.root !found;
    let eltnumbers = RefSet.elements !found in
      (* If not found, just ignore. *)
      let elements =
        map
          (fun n → try lookup_obj pdf n with Not_found → dpr "2N"; Null)
          eltnumbers
      in
        pdf.objects ←
          {maxobjnum = 0;
           parse = pdf.objects.parse;
           pdfobjects = pdfobjmap_empty};
        iter (addobj_given_num pdf) (combine eltnumbers elements)
```

▷ Objects referenced from a given one.

```
let objects_referenced no_follow_entries no_follow_contains pdf pdfobject =
  let set = ref RefSet.empty in
    referenced no_follow_entries no_follow_contains pdf set 0 (Parsed pdfobject);
    RefSet.elements !set
```

▷ The same, but return the objects too.

```
let objects_referenced_and_objects no_follow_entries no_follow_contains pdf pdfobject =
  let nums =
    objects_referenced no_follow_entries no_follow_contains pdf pdfobject
  in
    combine nums (map (lookup_obj pdf) nums)
```

▷ Remove a dictionary entry. Also works for streams.

```
let rec remove_dict_entry dict key =
  match dict with
  | Dictionary d → Dictionary (remove key d)
  | Stream ({contents = (dict', stream)} as s) →
      s := (remove_dict_entry dict' key, stream);
      Stream s
  | _ → raise (PDFError "remove_dict_entry:  not a dictionary")
```

▷ Replace dict entry, raising Not_found if it's not there. Also works for streams.

```
let rec replace_dict_entry dict key value =
  match dict with
  | Dictionary d → Dictionary (replace key value d)
  | Stream ({contents = (dict', stream)} as s) →
      s := (replace_dict_entry dict' key value, stream);
      Stream s
  | _ → raise (PDFError "replace_dict_entry:  not a dictionary.")
```

▷ Add a dict entry, replacing if there. Also works for streams.

```
let rec add_dict_entry dict key value =
  match dict with
  | Dictionary d → Dictionary (add key value d)
  | Stream ({contents = (dict', stream)} as s) →
      s := (add_dict_entry dict' key value, stream);
      Stream s
  | _ → raise (PDFError "add_dict_entry:  not a dictionary.")
```

Find the contents of a stream as a bytestream.

```
let rec bigarray_of_stream s =
  getstream s;
  match s with
  | Stream {contents = _, Got bytestream} → bytestream
  | _ → failwith "couldn't extract raw stream"
```

▷ Given a dictionary and a prefix (e.g gs), return a name, starting with the prefix, which is not already in the dictionary (e.g /gs0).

```
let unique_key prefix obj =
  let elts = match obj with
    | Dictionary es
    | Stream {contents = Dictionary es, _} → es
    | _ → raise (PDFError "unique_key:  Not a dictionary or stream")
  in
    let names = fst (split elts)
    and name_of_num n = "/" ^ prefix ^ string_of_int n
    and num = ref 0 in
      while mem (name_of_num !num) names do incr num done;
      name_of_num !num
```

▷ Given a PDF and potential filename, calculate an MD5 string and build a suitable /ID entry from it.

```
let generate_id (pdf : pdfdoc) (path : string) =
  (*IF-OCAML*)
  let gettime () = Unix.gettimeofday () in
  (*ENDIF-OCAML*)
  (* let gettime () = Sys.time () in *)
  let d =
    digest (path ^ string_of_float (gettime ()))
  in
    Array [String d; String d]
```

Find the page reference numbers, given the top level node of the page tree

```
let rec page_reference_numbers_inner pdf pages_node node_number =
  match lookup_direct pdf "/Type" pages_node with
  | Some (Name "/Pages") →
      begin match lookup_direct pdf "/Kids" pages_node with
      | Some (Array elts) →
          flatten
            (map
              (function
               | Indirect i →
                   page_reference_numbers_inner
                     pdf (direct pdf (Indirect i)) i
               | _ → raise (PDFError "badly formed page tree"))
              elts)
      | _ → raise (PDFError "badly formed page tree")
      end
  | Some (Name "/Page") → [node_number]
  | _ → raise (PDFError "badly formed page tree")

let page_reference_numbers pdf =
  let root = lookup_obj pdf pdf.root in
    let pages_node =
        match lookup_direct pdf "/Pages" root with
        | Some p → p
        | None → raise (PDFError "badly formed page tree")
    in
      page_reference_numbers_inner pdf pages_node −1
```

Find all the indirect numbers reachable from an entry in a dictionary, including the indirect of that dictionary entry, if it's an indirect.

```
let reference_numbers_of_dict_entry pdf dict entry =
  match dict with
  | Dictionary d →
      begin match lookup entry d with
      | Some x → objects_referenced [] [] pdf x
      | None →
          raise (PDFError "reference_numbers_of_dict_entry: no entry")
      end
  | _ →
      raise (PDFError "reference_numbers_of_dict_entry: not a dictionary")
```

Find the indirect reference given by the value associated with a key in a dictionary.

```
let find_indirect key dict =
  match dict with
  | Dictionary d →
      begin match lookup key d with
      | Some (Indirect i) → Some i
      | _ → None
      end
  | _ → raise (PDFError "find_indirect: not a dictionary")
```

5 Module PDFCrypt *Encryption and Decryption*

```
open Utility
```

5.1 Hashes, passwords and keys

Given an object number, generation number, input key and key length in bits, apply Algorithm 3.1 from the PDF Reference manual to obtain the hash to be used by the encryption function.

```
let find_hash r obj gen key keylength =
  let from_obj =
    [| i32toi (land32 obj 0x000000ffl);
       i32toi (lsr32 (land32 obj 0x0000ff00l) 8);
       i32toi (lsr32 (land32 obj 0x00ff0000l) 16) |]
  and from_gen =
    [| i32toi (land32 gen 0x000000ffl);
       i32toi (lsr32 (land32 gen 0x0000ff00l) 8) |]
  and extra =
    if r = 4 then [| 0x73; 0x41; 0x6C; 0x54 |] else [| |]
  in
    let digest_input = string_of_int_arrays [key; from_obj; from_gen; extra] in
      int_array_of_string
        (String.sub (digest digest_input) 0 (min 16 (keylength / 8 + 5)))
```

Find a key, given a password, O entry, P entry, id entry, and key length in bits.

```
let padding =
  [| 0x28; 0xbf; 0x4e; 0x5e; 0x4e; 0x75; 0x8a; 0x41;
     0x64; 0x00; 0x4e; 0x56; 0xff; 0xfa; 0x01; 0x08;
     0x2e; 0x2e; 0x00; 0xb6; 0xd0; 0x68; 0x3e; 0x80;
     0x2f; 0x0c; 0xa9; 0xfe; 0x64; 0x53; 0x69; 0x7a |]

let pad_password password =
  let pw = Array.make 32 0 in
    Array.iteri (fun i v -> if i < 32 then pw.(i) <- v) password;
    let n = Array.length password in
      if n < 32 then
        for x = n to 31 do
```

```
        pw.(x) ← padding.(x − n)
      done;
  pw

let find_key no_encrypt_metadata password r o p id keylength =
  let password = int_array_of_string password
  and o = int_array_of_string o
  and id = int_array_of_string id in
    let pw = pad_password password in
      let from_p =
        [| i32toi (land32 p 000000ff16l);
           i32toi (lsr32 (land32 p 0000ff0016l) 8);
           i32toi (lsr32 (land32 p 00ff000016l) 16);
           i32toi (lsr32 (land32 p ff00000016l) 24) |]
      and rev4_no_metadata =
        if r ≥ 4 ∧ no_encrypt_metadata then [|255; 255; 255; 255|] else [||]
      in
        let todigest = [pw; o; from_p; id; rev4_no_metadata] in
          let hash_input = string_of_int_arrays todigest in
            let hashed = digest hash_input in
              let hashed′ =
                if r ≥ 3 then
                  let h = ref hashed in
                    for x = 1 to 50 do
                      let hashed = digest !h in
                        h :=
                          string_of_int_array
                            (Array.sub (int_array_of_string hashed) 0 (keylength / 8))
                    done;
                    !h
                else
                  hashed
              in
                Array.sub (int_array_of_string hashed′) 0 (keylength / 8)
```

5.2 40bit / 128bit Encryption/Decryption Primitives

Encryption / Decryption given a key.

```
let ksa s key =
  let keylength = Array.length key in
    for i = 0 to 255 do s.(i) ← i done;
    let j = ref 0 in
      for i = 0 to 255 do
        j := (!j + s.(i) + key.(i mod keylength)) mod 256;
        swap s i !j
      done
```

```
let prga s pi pj =
  pi := (!pi + 1) mod 256;
  pj := (!pj + s.(!pi)) mod 256;
  swap s !pi !pj;
  s.((s.(!pi) + s.(!pj)) mod 256)

let crypt key data =
  let s = Array.make 256 0
  and pi = ref 0
  and pj = ref 0
  and out = mkstream (stream_size data) in
    ksa s key;
    for x = 0 to stream_size data - 1 do
      sset out x (sget data x lxor prga s pi pj)
    done;
    out
```

5.3 AES Encryption and Decryption Primitives

The state, an array of four length 4 arrays. state.(row).(column)

```
let st =
  Array.create_matrix 4 4 0
```

Finite field addition

```
let ( ++ ) = ( lxor )
```

Finite field multiplication modulo the irreducible polynomial.

```
let ( ** ) a b =
  let aa = ref a
  and bb = ref b
  and r = ref 0
  and t = ref 0 in
    while !aa ≠ 0 do
      if !aa land 1 ≠ 0 then r := !r lxor !bb;
      t := !bb land 80₁₆;
      bb := !bb lsl 1;
      if !t ≠ 0 then bb := !bb lxor 1b₁₆;
      aa := !aa lsr 1
    done;
    !r land ff₁₆
```

Multiplication of a finite field by x

```
let xtime f =
  (f lsl 1) lxor 1b₁₆
```

let $input\_to\_state\ d\ =$
$st.(0).(0)\ \leftarrow\ d.(0);\ st.(1).(0)\ \leftarrow\ d.(1);$
$st.(2).(0)\ \leftarrow\ d.(2);\ st.(3).(0)\ \leftarrow\ d.(3);$
$st.(0).(1)\ \leftarrow\ d.(4);\ st.(1).(1)\ \leftarrow\ d.(5);$
$st.(2).(1)\ \leftarrow\ d.(6);\ st.(3).(1)\ \leftarrow\ d.(7);$
$st.(0).(2)\ \leftarrow\ d.(8);\ st.(1).(2)\ \leftarrow\ d.(9);$
$st.(2).(2)\ \leftarrow\ d.(10);\ st.(3).(2)\ \leftarrow\ d.(11);$
$st.(0).(3)\ \leftarrow\ d.(12);\ st.(1).(3)\ \leftarrow\ d.(13);$
$st.(2).(3)\ \leftarrow\ d.(14);\ st.(3).(3)\ \leftarrow\ d.(15)$

let $sbox\ =$
[|
$63_{16}; 7c_{16}; 77_{16}; 7b_{16}; f2_{16}; 6b_{16}; 6f_{16}; c5_{16}; 30_{16}; 01_{16}; 67_{16}; 2b_{16}; fe_{16}; d7_{16}; ab_{16}; 76_{16};$
$ca_{16}; 82_{16}; c9_{16}; 7d_{16}; fa_{16}; 59_{16}; 47_{16}; f0_{16}; ad_{16}; d4_{16}; a2_{16}; af_{16}; 9c_{16}; a4_{16}; 72_{16}; c0_{16};$
$b7_{16}; fd_{16}; 93_{16}; 26_{16}; 36_{16}; 3f_{16}; f7_{16}; cc_{16}; 34_{16}; a5_{16}; e5_{16}; f1_{16}; 71_{16}; d8_{16}; 31_{16}; 15_{16};$
$04_{16}; c7_{16}; 23_{16}; c3_{16}; 18_{16}; 96_{16}; 05_{16}; 9a_{16}; 07_{16}; 12_{16}; 80_{16}; e2_{16}; eb_{16}; 27_{16}; b2_{16}; 75_{16};$
$09_{16}; 83_{16}; 2c_{16}; 1a_{16}; 1b_{16}; 6e_{16}; 5a_{16}; a0_{16}; 52_{16}; 3b_{16}; d6_{16}; b3_{16}; 29_{16}; e3_{16}; 2f_{16}; 84_{16};$
$53_{16}; d1_{16}; 00_{16}; ed_{16}; 20_{16}; fc_{16}; b1_{16}; 5b_{16}; 6a_{16}; cb_{16}; be_{16}; 39_{16}; 4a_{16}; 4c_{16}; 58_{16}; cf_{16};$
$d0_{16}; ef_{16}; aa_{16}; fb_{16}; 43_{16}; 4d_{16}; 33_{16}; 85_{16}; 45_{16}; f9_{16}; 02_{16}; 7f_{16}; 50_{16}; 3c_{16}; 9f_{16}; a8_{16};$
$51_{16}; a3_{16}; 40_{16}; 8f_{16}; 92_{16}; 9d_{16}; 38_{16}; f5_{16}; bc_{16}; b6_{16}; da_{16}; 21_{16}; 10_{16}; ff_{16}; f3_{16}; d2_{16};$
$cd_{16}; 0c_{16}; 13_{16}; ec_{16}; 5f_{16}; 97_{16}; 44_{16}; 17_{16}; c4_{16}; a7_{16}; 7e_{16}; 3d_{16}; 64_{16}; 5d_{16}; 19_{16}; 73_{16};$
$60_{16}; 81_{16}; 4f_{16}; dc_{16}; 22_{16}; 2a_{16}; 90_{16}; 88_{16}; 46_{16}; ee_{16}; b8_{16}; 14_{16}; de_{16}; 5e_{16}; 0b_{16}; db_{16};$
$e0_{16}; 32_{16}; 3a_{16}; 0a_{16}; 49_{16}; 06_{16}; 24_{16}; 5c_{16}; c2_{16}; d3_{16}; ac_{16}; 62_{16}; 91_{16}; 95_{16}; e4_{16}; 79_{16};$
$e7_{16}; c8_{16}; 37_{16}; 6d_{16}; 8d_{16}; d5_{16}; 4e_{16}; a9_{16}; 6c_{16}; 56_{16}; f4_{16}; ea_{16}; 65_{16}; 7a_{16}; ae_{16}; 08_{16};$
$ba_{16}; 78_{16}; 25_{16}; 2e_{16}; 1c_{16}; a6_{16}; b4_{16}; c6_{16}; e8_{16}; dd_{16}; 74_{16}; 1f_{16}; 4b_{16}; bd_{16}; 8b_{16}; 8a_{16};$
$70_{16}; 3e_{16}; b5_{16}; 66_{16}; 48_{16}; 03_{16}; f6_{16}; 0e_{16}; 61_{16}; 35_{16}; 57_{16}; b9_{16}; 86_{16}; c1_{16}; 1d_{16}; 9e_{16};$
$e1_{16}; f8_{16}; 98_{16}; 11_{16}; 69_{16}; d9_{16}; 8e_{16}; 94_{16}; 9b_{16}; 1e_{16}; 87_{16}; e9_{16}; ce_{16}; 55_{16}; 28_{16}; df_{16};$
$8c_{16}; a1_{16}; 89_{16}; 0d_{16}; bf_{16}; e6_{16}; 42_{16}; 68_{16}; 41_{16}; 99_{16}; 2d_{16}; 0f_{16}; b0_{16}; 54_{16}; bb_{16}; 16_{16}$
|]

let $inv\_sbox\ =$
[|
$52_{16}; 09_{16}; 6a_{16}; d5_{16}; 30_{16}; 36_{16}; a5_{16}; 38_{16}; bf_{16}; 40_{16}; a3_{16}; 9e_{16}; 81_{16}; f3_{16}; d7_{16}; fb_{16};$
$7c_{16}; e3_{16}; 39_{16}; 82_{16}; 9b_{16}; 2f_{16}; ff_{16}; 87_{16}; 34_{16}; 8e_{16}; 43_{16}; 44_{16}; c4_{16}; de_{16}; e9_{16}; cb_{16};$
$54_{16}; 7b_{16}; 94_{16}; 32_{16}; a6_{16}; c2_{16}; 23_{16}; 3d_{16}; ee_{16}; 4c_{16}; 95_{16}; 0b_{16}; 42_{16}; fa_{16}; c3_{16}; 4e_{16};$
$08_{16}; 2e_{16}; a1_{16}; 66_{16}; 28_{16}; d9_{16}; 24_{16}; b2_{16}; 76_{16}; 5b_{16}; a2_{16}; 49_{16}; 6d_{16}; 8b_{16}; d1_{16}; 25_{16};$
$72_{16}; f8_{16}; f6_{16}; 64_{16}; 86_{16}; 68_{16}; 98_{16}; 16_{16}; d4_{16}; a4_{16}; 5c_{16}; cc_{16}; 5d_{16}; 65_{16}; b6_{16}; 92_{16};$
$6c_{16}; 70_{16}; 48_{16}; 50_{16}; fd_{16}; ed_{16}; b9_{16}; da_{16}; 5e_{16}; 15_{16}; 46_{16}; 57_{16}; a7_{16}; 8d_{16}; 9d_{16}; 84_{16};$
$90_{16}; d8_{16}; ab_{16}; 00_{16}; 8c_{16}; bc_{16}; d3_{16}; 0a_{16}; f7_{16}; e4_{16}; 58_{16}; 05_{16}; b8_{16}; b3_{16}; 45_{16}; 06_{16};$
$d0_{16}; 2c_{16}; 1e_{16}; 8f_{16}; ca_{16}; 3f_{16}; 0f_{16}; 02_{16}; c1_{16}; af_{16}; bd_{16}; 03_{16}; 01_{16}; 13_{16}; 8a_{16}; 6b_{16};$
$3a_{16}; 91_{16}; 11_{16}; 41_{16}; 4f_{16}; 67_{16}; dc_{16}; ea_{16}; 97_{16}; f2_{16}; cf_{16}; ce_{16}; f0_{16}; b4_{16}; e6_{16}; 73_{16};$
$96_{16}; ac_{16}; 74_{16}; 22_{16}; e7_{16}; ad_{16}; 35_{16}; 85_{16}; e2_{16}; f9_{16}; 37_{16}; e8_{16}; 1c_{16}; 75_{16}; df_{16}; 6e_{16};$
$47_{16}; f1_{16}; 1a_{16}; 71_{16}; 1d_{16}; 29_{16}; c5_{16}; 89_{16}; 6f_{16}; b7_{16}; 62_{16}; 0e_{16}; aa_{16}; 18_{16}; be_{16}; 1b_{16};$
$fc_{16}; 56_{16}; 3e_{16}; 4b_{16}; c6_{16}; d2_{16}; 79_{16}; 20_{16}; 9a_{16}; db_{16}; c0_{16}; fe_{16}; 78_{16}; cd_{16}; 5a_{16}; f4_{16};$
$1f_{16}; dd_{16}; a8_{16}; 33_{16}; 88_{16}; 07_{16}; c7_{16}; 31_{16}; b1_{16}; 12_{16}; 10_{16}; 59_{16}; 27_{16}; 80_{16}; ec_{16}; 5f_{16};$
$60_{16}; 51_{16}; 7f_{16}; a9_{16}; 19_{16}; b5_{16}; 4a_{16}; 0d_{16}; 2d_{16}; e5_{16}; 7a_{16}; 9f_{16}; 93_{16}; c9_{16}; 9c_{16}; ef_{16};$
$a0_{16}; e0_{16}; 3b_{16}; 4d_{16}; ae_{16}; 2a_{16}; f5_{16}; b0_{16}; c8_{16}; eb_{16}; bb_{16}; 3c_{16}; 83_{16}; 53_{16}; 99_{16}; 61_{16};$
$17_{16}; 2b_{16}; 04_{16}; 7e_{16}; ba_{16}; 77_{16}; d6_{16}; 26_{16}; e1_{16}; 69_{16}; 14_{16}; 63_{16}; 55_{16}; 21_{16}; 0c_{16}; 7d_{16}$
|]

let $subbyte\ b\ =$
$sbox.(b)$

```
let sub_bytes () =
  for r = 0 to 3 do
    for c = 0 to 3 do
      st.(r).(c) <- sbox.(st.(r).(c))
    done
  done

let inv_sub_bytes () =
  for r = 0 to 3 do
    for c = 0 to 3 do
      st.(r).(c) <- inv_sbox.(st.(r).(c))
    done
  done
```

Key schedule

```
let keys =
  Array.create 44 0l

let word_of_bytes a b c d =
  let a, b, c, d =
    (lsl32 (i32ofi a) 24),
    (lsl32 (i32ofi b) 16),
    (lsl32 (i32ofi c) 8),
    (i32ofi d)
  in
    lor32 (lor32 a b) (lor32 c d)

let bytes_of_word w =
  i32toi (lsr32 w 24),
  i32toi (land32 (lsr32 w 16) 0xFFl),
  i32toi (land32 (lsr32 w 8) 0xFFl),
  i32toi (land32 w 0xFFl)

let subword w =
  let a, b, c, d = bytes_of_word w in
    word_of_bytes (subbyte a) (subbyte b) (subbyte c) (subbyte d)

let rotword w =
  let a, b, c, d = bytes_of_word w in
    word_of_bytes b c d a
```

Round Constants (0..10)

```
let rcon =
  [| 0l;
     lsl32 0x01l 24; lsl32 0x02l 24; lsl32 0x04l 24; lsl32 0x08l 24;
     lsl32 0x10l 24; lsl32 0x20l 24; lsl32 0x40l 24; lsl32 0x80l 24;
     lsl32 0x1bl 24; lsl32 0x36l 24; |]
```

Key expansion

```
let key_expansion key =
  let temp = ref 0l
  and i = ref 0 in
    while (!i < 4) do
      keys.(!i) ←
        word_of_bytes
          key.(4 × !i) key.(4 × !i + 1) key.(4 × !i + 2) key.(4 × !i + 3);
      incr i
    done;
    i := 4;
    while (!i < 44) do
      temp := keys.(!i − 1);
      if !i mod 4 = 0 then
        temp := lxor32 (subword (rotword !temp)) rcon.(!i / 4);
      keys.(!i) ← lxor32 keys.(!i − 4) !temp;
      incr i
    done

let shift_rows () =
  let a, b, c, d =
    st.(1).(0), st.(1).(1), st.(1).(2), st.(1).(3)
  in
    st.(1).(0) ← b; st.(1).(1) ← c;
    st.(1).(2) ← d; st.(1).(3) ← a;
  let a, b, c, d =
    st.(2).(0), st.(2).(1), st.(2).(2), st.(2).(3)
  in
    st.(2).(0) ← c; st.(2).(1) ← d;
    st.(2).(2) ← a; st.(2).(3) ← b;
  let a, b, c, d =
    st.(3).(0), st.(3).(1), st.(3).(2), st.(3).(3)
  in
    st.(3).(0) ← d; st.(3).(1) ← a;
    st.(3).(2) ← b; st.(3).(3) ← c

let inv_shift_rows () =
  let a, b, c, d =
    st.(1).(0), st.(1).(1), st.(1).(2), st.(1).(3)
  in
    st.(1).(0) ← d; st.(1).(1) ← a;
    st.(1).(2) ← b; st.(1).(3) ← c;
  let a, b, c, d =
    st.(2).(0), st.(2).(1), st.(2).(2), st.(2).(3)
  in
    st.(2).(0) ← c; st.(2).(1) ← d;
    st.(2).(2) ← a; st.(2).(3) ← b;
  let a, b, c, d =
    st.(3).(0), st.(3).(1), st.(3).(2), st.(3).(3)
  in
    st.(3).(0) ← b; st.(3).(1) ← c;
    st.(3).(2) ← d; st.(3).(3) ← a
```

let $mix\_columns$ () $=$
  for $c = 0$ **to** 3 **do**
    let $s'0 =$
      $(\mathtt{02}_{16} ** st.(0).(c)) ++ (\mathtt{03}_{16} ** st.(1).(c)) ++ st.(2).(c) ++ st.(3).(c)$
    and $s'1 =$
      $st.(0).(c) ++ (\mathtt{02}_{16} ** st.(1).(c)) ++ (\mathtt{03}_{16} ** st.(2).(c)) ++ st.(3).(c)$
    and $s'2 =$
      $st.(0).(c) ++ st.(1).(c) ++ (\mathtt{02}_{16} ** st.(2).(c)) ++ (\mathtt{03}_{16} ** st.(3).(c))$
    and $s'3 =$
      $(\mathtt{03}_{16} ** st.(0).(c)) ++ st.(1).(c) ++ st.(2).(c) ++ (\mathtt{02}_{16} ** st.(3).(c))$
    in
      $st.(0).(c) \leftarrow s'0;$
      $st.(1).(c) \leftarrow s'1;$
      $st.(2).(c) \leftarrow s'2;$
      $st.(3).(c) \leftarrow s'3$
  done

let $inv\_mix\_columns$ () $=$
  for $c = 0$ **to** 3 **do**
    let $s'0 =$
      $(\mathtt{0e}_{16} ** st.(0).(c)) ++ (\mathtt{0b}_{16} ** st.(1).(c)) ++$
      $(\mathtt{0d}_{16} ** st.(2).(c)) ++ (\mathtt{09}_{16} ** st.(3).(c))$
    and $s'1 =$
      $(\mathtt{09}_{16} ** st.(0).(c)) ++ (\mathtt{0e}_{16} ** st.(1).(c)) ++$
      $(\mathtt{0b}_{16} ** st.(2).(c)) ++ (\mathtt{0d}_{16} ** st.(3).(c))$
    and $s'2 =$
      $(\mathtt{0d}_{16} ** st.(0).(c)) ++ (\mathtt{09}_{16} ** st.(1).(c)) ++$
      $(\mathtt{0e}_{16} ** st.(2).(c)) ++ (\mathtt{0b}_{16} ** st.(3).(c))$
    and $s'3 =$
      $(\mathtt{0b}_{16} ** st.(0).(c)) ++ (\mathtt{0d}_{16} ** st.(1).(c)) ++$
      $(\mathtt{09}_{16} ** st.(2).(c)) ++ (\mathtt{0e}_{16} ** st.(3).(c))$
    in
      $st.(0).(c) \leftarrow s'0;$
      $st.(1).(c) \leftarrow s'1;$
      $st.(2).(c) \leftarrow s'2;$
      $st.(3).(c) \leftarrow s'3$
  done

Add a round key to the state.

let $add\_round\_key\ keypos =$
  let $a1, a2, a3, a4 = bytes\_of\_word\ keys.(keypos)$
  and $b1, b2, b3, b4 = bytes\_of\_word\ keys.(keypos + 1)$
  and $c1, c2, c3, c4 = bytes\_of\_word\ keys.(keypos + 2)$
  and $d1, d2, d3, d4 = bytes\_of\_word\ keys.(keypos + 3)$ **in**
    $st.(0).(0) \leftarrow st.(0).(0) ++ a1;\ st.(1).(0) \leftarrow st.(1).(0) ++ a2;$
    $st.(2).(0) \leftarrow st.(2).(0) ++ a3;\ st.(3).(0) \leftarrow st.(3).(0) ++ a4;$
    $st.(0).(1) \leftarrow st.(0).(1) ++ b1;\ st.(1).(1) \leftarrow st.(1).(1) ++ b2;$
    $st.(2).(1) \leftarrow st.(2).(1) ++ b3;\ st.(3).(1) \leftarrow st.(3).(1) ++ b4;$
    $st.(0).(2) \leftarrow st.(0).(2) ++ c1;\ st.(1).(2) \leftarrow st.(1).(2) ++ c2;$
    $st.(2).(2) \leftarrow st.(2).(2) ++ c3;\ st.(3).(2) \leftarrow st.(3).(2) ++ c4;$
    $st.(0).(3) \leftarrow st.(0).(3) ++ d1;\ st.(1).(3) \leftarrow st.(1).(3) ++ d2;$

```
    st.(2).(3) ← st.(2).(3) ++ d3; st.(3).(3) ← st.(3).(3) ++ d4

let output_from_state () =
  [| st.(0).(0); st.(1).(0); st.(2).(0); st.(3).(0);
     st.(0).(1); st.(1).(1); st.(2).(1); st.(3).(1);
     st.(0).(2); st.(1).(2); st.(2).(2); st.(3).(2);
     st.(0).(3); st.(1).(3); st.(2).(3); st.(3).(3) |]
```

Encryption cipher. Assumes key already expanded.

```
let cipher data_in =
  input_to_state data_in;
  add_round_key 0;
  for round = 1 to 9 do
    sub_bytes ();
    shift_rows ();
    mix_columns ();
    add_round_key (round × 4)
  done;
  sub_bytes ();
  shift_rows ();
  add_round_key 40;
  output_from_state ()
```

Decryption cipher. Assumes key already expanded.

```
let inv_cipher data_in =
  input_to_state data_in;
  add_round_key 40;
  for round = 9 downto 1 do
    inv_shift_rows ();
    inv_sub_bytes ();
    add_round_key (round × 4);
    inv_mix_columns ();
  done;
  inv_shift_rows ();
  inv_sub_bytes ();
  add_round_key 0;
  output_from_state ()
```

Pad the input data (RFC2898, PKCS #5), then encrypt using a 16 byte AES cipher in cipher block chaining mode, with a random initialisation vector, which is stored as the first 16 bytes of the result.

```
let ran255 () =
  (*IF-OCAML*)
  Random.self_init ();
  Random.int 255
  (*ENDIF-OCAML*)

let mkiv () =
  let r = ran255 in
    [| r (); r (); r (); r ();
       r (); r (); r (); r ();
```

```
    r (); r (); r (); r ();
    r (); r (); r (); r () |]
```

Debug function to print a block as characters.

```
let print_block arr =
  Array.iter (fun i → Printf.printf "%c" (char_of_int i)) arr;
  flprint "\n\n"
```

Build blocks for encryption, including padding.

```
let get_blocks data =
  let l = stream_size data in
    let fullblocks =
      if l < 16 then [] else
        let blocks = ref [] in
          for x = 0 to l / 16 − 1 do
            blocks =|
              Array.of_list
                (map (sget data) (ilist (x × 16) (x × 16 + 15)))
          done;
          rev !blocks
    and lastblock =
      let getlast n =
        if n = 0 then [] else
          let bytes = ref [] in
            for x = 0 to n − 1 do
              bytes =| sget data (l − 1 − x)
            done;
            !bytes
      and pad n =
        many n n
      in
        let overflow = l mod 16 in
          Array.of_list (getlast overflow @ pad (16 − overflow))
    in
      fullblocks @ [lastblock]
```

Flatten a list of blocks into a bytestream

```
let stream_of_blocks blocks =
  let len = 16 × length blocks in
    let s = mkstream len
    and p = ref 0 in
      iter
        (fun a →
          Array.iter (fun v → sset s !p v; incr p) a)
        blocks;
      s
```

These two functions strip the padding from a stream once it's been decoded.

```
let get_padding s =
  let l = stream_size s in
    assert (l ≥ 16);
    let potential = sget s (l − 1) in
      if potential > 10₁₆ ∨ potential < 01₁₆ then None else
        let rec elts_equal p f t =
          if f = t then p = sget s t else
            p = sget s f ∧ elts_equal p (f + 1) t
        in
          if elts_equal potential (l − potential − 1) (l − 1)
            then Some potential
            else None
```

```
let cutshort s =
  if stream_size s = 0 then mkstream 0 else
    if stream_size s ≤ 16 then s else
      match get_padding s with
      | None → s
      | Some padding →
          let s′ = mkstream (stream_size s − padding) in
            for x = 0 to stream_size s′ − 1 do
              sset s′ x (sget s x)
            done;
            s′
```

Get blocks for decryption

```
let get_plain_blocks d =
  if stream_size d mod 16 ≠ 0 then raise (Pdf.PDFError "Bad AES data") else
    map
      (fun n → Array.init 16 (fun x → sget d (16 × n + x)))
      (ilist 0 (stream_size d / 16 − 1))
```

Decrypt data

```
let aes_decrypt_data key data =
  key_expansion key;
  match get_plain_blocks data with
  | [] | [_] → mkstream 0
  | iv :: codeblocks →
      let prev_ciphertext = ref iv
      and outblocks = ref [] in
        iter
          (fun block →
             let plaintext =
               (array_map2 (lxor)) (inv_cipher block) !prev_ciphertext
             in
               prev_ciphertext := block;
               outblocks = | plaintext)
          codeblocks;
          cutshort (stream_of_blocks (rev !outblocks))
```

Encrypt data

```
let aes_encrypt_data key data =
  key_expansion key;
  let outblocks = ref []
  and firstblock = mkiv () in
    let prev_ciphertext = ref firstblock in
      iter
        (fun block →
          let ciphertext =
            cipher ((array_map2 (lxor)) block !prev_ciphertext)
          in
            prev_ciphertext := ciphertext;
            outblocks =| ciphertext)
        (get_blocks data);
        stream_of_blocks (firstblock :: rev !outblocks)
```

5.4 Encryption and decryption of PDF files

Authenticate the user password, given the password string and U, O, P, id and key length entry.

```
let authenticate_user no_encrypt_metadata password r u o p id keylength =
  let u = int_array_of_string u in
    let key = find_key no_encrypt_metadata password r o p id keylength in
      if r ≥ 3 then
        let id = int_array_of_string id in
          let todigest = [padding; id] in
            let hash_input = string_of_int_arrays todigest in
              let hashed = digest hash_input in
                let encrypted_hashed =
                  int_array_of_stream (crypt key (bytestream_of_string hashed))
                in
                  let u′ = ref [||] in
                    u′ := encrypted_hashed;
                    for x = 1 to 19 do
                      let key′ = Array.make (keylength / 8) 0 in
                        for k = 0 to (keylength / 8) − 1 do
                          key′.(k) ← key.(k) lxor x
                        done;
                        u′ :=
                          int_array_of_stream
                            (crypt key′ (stream_of_int_array !u′))
                    done;
                    Array.sub u 0 16 = !u′
      else
        u = int_array_of_stream (crypt key (stream_of_int_array padding))
```

Decrypt a PDF file, given the user password.

```
let rec decrypt pdf no_encrypt_metadata encrypt obj gen key keylength r = function
  | Pdf.String s →
      let f =
        (if r = 4 then
          (if encrypt then aes_encrypt_data else aes_decrypt_data)
         else
          crypt)
      in
        let s_ints = bytestream_of_string s in
          let hash = find_hash r (i32ofi obj) (i32ofi gen) key keylength in
            Pdf.String (string_of_bytestream (f hash s_ints))
  | (Pdf.Stream _) as stream →
      decrypt_stream pdf no_encrypt_metadata encrypt obj gen key keylength r stream
  | Pdf.Array a →
      Pdf.recurse_array (decrypt pdf no_encrypt_metadata encrypt obj gen key keylength r) a
  | Pdf.Dictionary d →
      Pdf.recurse_dict (decrypt pdf no_encrypt_metadata encrypt obj gen key keylength r) d
  | x → x

and decrypt_stream pdf no_encrypt_metadata encrypt obj gen key keylength r stream =
  Pdf.getstream stream;
  begin match stream with
  | (Pdf.Stream {contents = (Pdf.Dictionary dict as d, Pdf.Got data)}) as stream →

      if
        begin let identity_crypt_filter_present =
          match Pdf.lookup_direct pdf "/Filter" d with
          | Some (Pdf.Name "/Crypt")
          | Some (Pdf.Array (Pdf.Name "/Crypt"::_)) →
              begin match Pdf.lookup_direct pdf "/DecodeParms" d with
              | Some (Pdf.Dictionary decodeparmsdict)
              | Some (Pdf.Array (Pdf.Dictionary decodeparmsdict :: _)) →
                  begin match
                    Pdf.lookup_direct pdf "/Name" (Pdf.Dictionary decodeparmsdict)
                  with
                  | Some (Pdf.Name "/Identity") | None → true
                  | _ → false
                  end
              | _ → true
              end
          | _ → false
        in
          (no_encrypt_metadata ∧
             Pdf.lookup_direct pdf "/Type" d = Some (Pdf.Name "/Metadata"))
          ∨ identity_crypt_filter_present
        end
      then
        stream
      else
        let data' =
```

```
        let f =
          (if r = 4 then
            (if encrypt then aes_encrypt_data else aes_decrypt_data)
          else
            crypt)
        in
          let hash = find_hash r (i32ofi obj) (i32ofi gen) key keylength in
            f hash data
      and dict' =
        Pdf.recurse_dict
          (decrypt pdf no_encrypt_metadata encrypt obj gen key keylength r) dict
      in
        let dict'' =
          if stream_size data ≠ stream_size data' then
            Pdf.replace_dict_entry
              dict' "/Length" (Pdf.Integer (stream_size data'))
          else
            dict'
        in
          Pdf.Stream {contents = (dict'', Pdf.Got data')}
  | _ → raise (Assert_failure ("decrypt_stream", 0, 0))
  end

let process_cryption no_encrypt_metadata encrypt pdf crypt_type user_pw r u o p id keylength =
  if authenticate_user no_encrypt_metadata user_pw r u o p id keylength then
    begin
      let key = find_key no_encrypt_metadata user_pw r o p id keylength in
        Pdf.objiter_gen
          (fun objnum gennum obj →
             ignore
               (Pdf.addobj_given_num
               pdf
               (objnum,
                  decrypt pdf no_encrypt_metadata encrypt objnum gennum key keylength r obj)))
          pdf;
        let trailerdict' = Pdf.remove_dict_entry pdf.Pdf.trailerdict "/Encrypt" in
          pdf.Pdf.trailerdict ← trailerdict';
          Some pdf
    end
  else
    None
```

ARC4 = old style or crypt filter with V2. AESV2 = Crypt filter with AESV2. We don't need to distinguish between old and new ARC4 since support for different crypts for different filter works anyway.

```
type encryption =
  | ARC4 of int × int (∗ keylength, r (= 2 or 3 or 4) ∗)
  | AESV2 (∗ v = 4, r = 4 ∗)
```

```
let get_encryption_values pdf =
  match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
  | None → raise (Assert_failure ("get_encryption_values", 0, 0)) (* Never
called on an unencrypted PDF *)
  | Some encryptdict →
      let crypt_type =
        match
          Pdf.lookup_direct pdf "/Filter" encryptdict,
          Pdf.lookup_direct pdf "/V" encryptdict,
          Pdf.lookup_direct pdf "/Length" encryptdict,
          Pdf.lookup_direct pdf "/R" encryptdict
        with
        | Some (Pdf.Name "/Standard"), Some (Pdf.Integer 1), _, Some (Pdf.Integer r)
        | Some (Pdf.Name "/Standard"), Some (Pdf.Integer 2), None, Some (Pdf.Integer r) →

            Some (ARC4 (40, r))
        | Some (Pdf.Name "/Standard"), Some (Pdf.Integer 2), Some (Pdf.Integer n), _
            when n mod 8 = 0 ∧ n ≥ 40 ∧ n ≤ 128 →
              Some (ARC4 (n, 3))
        | Some (Pdf.Name "/Standard"), Some (Pdf.Integer 4), length, _ →
            begin match Pdf.lookup_direct pdf "/CF" encryptdict with
            | Some cfdict →
                begin match Pdf.lookup_direct pdf "/StdCF" cfdict with
                | Some stdcfdict →
                    begin match Pdf.lookup_direct pdf "/CFM" stdcfdict with
                    | Some (Pdf.Name "/V2") →
                        begin match length with
                        | Some (Pdf.Integer i) → Some (ARC4 (i, 4))
                        | _ →
                            begin match Pdf.lookup_direct pdf "/Length" cfdict with
                            | Some (Pdf.Integer i) → Some (ARC4 (i, 4))
                            | _ → None
                            end
                        end
                    | Some (Pdf.Name "/AESV2") → Some AESV2
                    | _ → None
                    end
                | _ → None
                end
            | _ → None
            end
        | _ → None
      in
        match crypt_type with
        | None → raise (Pdf.PDFError "No encryption method")
        | Some crypt_type →
            let o =
              match Pdf.lookup_direct pdf "/O" encryptdict with
              | Some (Pdf.String o) → o
              | _ → raise (Pdf.PDFError "Bad or missing /O entry")
```

```
        and u =
          match Pdf.lookup_direct pdf "/U" encryptdict with
          | Some (Pdf.String u) → u
          | _ → raise (Pdf.PDFError "Bad or missing /U entry")
        and p =
          match Pdf.lookup_direct pdf "/P" encryptdict with
          | Some (Pdf.Integer flags) → i32ofi flags
          | _ → raise (Pdf.PDFError "Bad or missing /P entry")
        and id =
          match Pdf.lookup_direct pdf "/ID" pdf.Pdf.trailerdict with
          | Some (Pdf.Array [Pdf.String s; _]) → s
          | _ → raise (Pdf.PDFError "Bad or missing /ID element")
        in
          crypt_type, u, o, p, id
```

Permissions

```
type permission =
  | NoEdit                                    (* R2, Bit 4 *)
  | NoPrint                                   (* R2, Bit 3 *)
  | NoCopy                                    (* R2, Bit 5 *)
  | NoAnnot                                   (* R2, Bit 6 *)
  | NoForms                                   (* R3 only, Bit 9 *)
  | NoExtract                                 (* R3 only, Bit 10 *)
  | NoAssemble                                (* R3 only, Bit 11 *)
  | NoHqPrint                                 (* R3 only, Bit 12 *)

let string_of_permission = function
  | NoEdit → "NoEdit"
  | NoPrint → "NoPrint"
  | NoCopy → "NoCopy"
  | NoAnnot → "NoAnnot"
  | NoForms → "NoForms"
  | NoExtract → "NoExtract"
  | NoAssemble → "NoAssemble"
  | NoHqPrint → "NoHqPrint"

let string_of_bans bans =
  fold_left ( ^ ) "" (interleave " " (map string_of_permission bans))

let p_of_banlist toban =
  let p = ref 0l in
    let setbit n b =
      if b then p := Int32.logor !p (Int32.shift_left 1l (n - 1))
    and notin =
      notpred (mem' toban)
    in
      setbit 3 (notin NoPrint);
      setbit 4 (notin NoEdit);
      setbit 5 (notin NoCopy);
      setbit 6 (notin NoAnnot);
      setbit 7 true;
      setbit 8 true;
```

```
      setbit 9 (notin NoForms);
      setbit 10 (notin NoExtract);
      setbit 11 (notin NoAssemble);
      setbit 12 (notin NoHqPrint);
      iter (fun x → setbit x true) (ilist 13 32);
      !p

let banlist_of_p p =
  let l = ref []
  and bitset n =
    Int32.logand (Int32.shift_right p (n − 1)) 1l = 0l
  in
    if bitset 3 then l =| NoPrint;
    if bitset 4 then l =| NoEdit;
    if bitset 5 then l =| NoCopy;
    if bitset 6 then l =| NoAnnot;
    if bitset 9 then l =| NoForms;
    if bitset 10 then l =| NoExtract;
    if bitset 11 then l =| NoAssemble;
    if bitset 12 then l =| NoHqPrint;
    !l
```

Main function for decryption.

```
let decrypt_pdf user_pw pdf =
  match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
  | None → Some pdf, []
  | Some encrypt_dict →
      let crypt_type, u, o, p, id = get_encryption_values pdf in
        let r, keylength =
          match crypt_type with
          | AESV2 → 4, 128
          | ARC4 (k, r) → r, k
        and encrypt_metadata =
          match Pdf.lookup_direct pdf "/EncryptMetadata" encrypt_dict with
          | Some (Pdf.Boolean false) → false
          | _ → true
        in
          process_cryption (¬ encrypt_metadata) false
          pdf crypt_type user_pw r u o p id keylength,
          banlist_of_p p
```

Just decrypt a single stream, given the user password, and pdf. This is used to decrypt cross-reference streams during the reading of a file – the PDF is only partially formed at this stage.

```
let decrypt_single_stream user_pw pdf obj gen stream =
  match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
  | None → stream
  | Some encrypt_dict →
      let crypt_type, u, o, p, id = get_encryption_values pdf in
        let r, keylength =
          match crypt_type with
```

```
      | AESV2 → 4, 128
      | ARC4 (k, r) → r, k
    and no_encrypt_metadata =
      match Pdf.lookup_direct pdf "/EncryptMetadata" encrypt_dict with
      | Some (Pdf.Boolean false) → true
      | _ → false
    in
      if
        authenticate_user no_encrypt_metadata user_pw r u o p id keylength
      then
        let key = find_key no_encrypt_metadata user_pw r o p id keylength in
          decrypt_stream pdf no_encrypt_metadata false obj gen key keylength r stream
      else
        raise (Pdf.PDFError "Bad password when decrypting stream")
```

Calculate the owner key from the padded owner password (as calculated by *pad_password*

```
let owner_key padded_owner keylength r =
  let digest1 = digest (string_of_int_array padded_owner) in
    let digest2 =
      if r ≥ 3 then
        let d = ref digest1 in
          for x = 1 to 50 do
            d := digest !d
          done;
          !d
        else
          digest1
    in
      int_array_of_string (String.sub digest2 0 (keylength / 8))
```

Calculate XOR keys

```
let mkkey key x =
  let key' = Array.copy key in
    for k = 0 to Array.length key − 1 do
      key'.(k) ← key.(k) lxor x
    done;
    key'
```

Decrypt with the owner password.

```
let decrypt_pdf_owner owner_pw pdf =
  match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
  | None → Some pdf
  | _ →
    let padded_owner = pad_password (int_array_of_string owner_pw) in
      let crypt_type, _, o, _, _ = get_encryption_values pdf in
        let r, keylength =
          match crypt_type with
          | AESV2 → 4, 128
          | ARC4 (k, r) → r, k
```

```
      in
        let user_pw =
          let key = owner_key padded_owner keylength r in
            if r = 2 then
              string_of_bytestream (crypt key (bytestream_of_string o))
            else (* r ¿= 3 *)
              begin
                let acc = ref (bytestream_of_string o) in
                  for x = 19 downto 0 do
                    acc := crypt (mkkey key x) !acc
                  done;
                  string_of_bytestream !acc
              end
        in
          fst (decrypt_pdf user_pw pdf)
```

Make an owner password

```
let mk_owner r owner_pw user_pw keylength =
  let padded_owner =
    let source =
      if owner_pw = "" then user_pw else owner_pw
    in
      pad_password (int_array_of_string source)
  in
    let key = owner_key padded_owner keylength r in
      let padded_user = pad_password (int_array_of_string user_pw) in
        if r = 2 then
          string_of_bytestream (crypt key (stream_of_int_array padded_user))
        else (* r ¿= 3 *)
          let acc = ref (crypt key (stream_of_int_array padded_user)) in
            for x = 1 to 19 do
              acc := crypt (mkkey key x) !acc
            done;
            string_of_bytestream !acc
```

Make a user password

```
let mk_user no_encrypt_metadata user_pw o p id r keylength =
  let key = find_key no_encrypt_metadata user_pw r o p id keylength in
    if r = 2 then
      string_of_bytestream (crypt key (stream_of_int_array padding))
    else (* r ¿= 3 *)
      let digest_input = [padding; int_array_of_string id] in
        let d = digest (string_of_int_arrays digest_input) in
          let acc = ref (crypt key (bytestream_of_string d)) in
            for x = 1 to 19 do
              acc := crypt (mkkey key x) !acc
            done;
            string_of_bytestream !acc ^ (implode (many '\000' 16))
```

Get the ID, or add one if there's not one there. Return the updated pdf and the ID

```
let get_or_add_id pdf =
  match Pdf.lookup_direct pdf "/ID" pdf.Pdf.trailerdict with
  | Some (Pdf.Array [Pdf.String s; _]) →
      s, pdf
  | _ →
      let idobj = Pdf.generate_id pdf "" in
        let pdf' =
          {pdf with
            Pdf.trailerdict =
              Pdf.add_dict_entry pdf.Pdf.trailerdict "/ID" idobj}
        in
          match idobj with
          | Pdf.Array [Pdf.String s; _] → s, pdf'
          | _ → raise (Assert_failure ("get_or_add_id", 0, 0))
```

40bit encryption

```
let encrypt_pdf_40bit user_pw owner_pw banlist pdf =
  let p = p_of_banlist banlist
  and owner = mk_owner 2 owner_pw user_pw 40
  and id, pdf = get_or_add_id pdf in
    let user = mk_user false user_pw owner p id 2 40 in
      let crypt_dict =
        Pdf.Dictionary
          ["/Filter", Pdf.Name "/Standard";
           "/V", Pdf.Integer 1;
           "/R", Pdf.Integer 2;
           "/O", Pdf.String owner;
           "/U", Pdf.String user;
           "/P", Pdf.Integer (i32toi p)]
      in
        match process_cryption false false pdf (ARC4 (40, 2)) user_pw 2 user owner p id 40 with
        | Some pdf →
            {pdf with
              Pdf.trailerdict =
                Pdf.add_dict_entry
                  pdf.Pdf.trailerdict "/Encrypt" (Pdf.Indirect (Pdf.addobj pdf crypt_dict))}
        | None → raise (Pdf.PDFError "Encryption failed")
```

128bit encryption

```
let encrypt_pdf_128bit user_pw owner_pw banlist pdf =
  let p = p_of_banlist banlist
  and owner = mk_owner 3 owner_pw user_pw 128
  and id, pdf = get_or_add_id pdf in
    let user = mk_user false user_pw owner p id 3 128 in
      let crypt_dict =
        Pdf.Dictionary
          ["/Filter", Pdf.Name "/Standard";
           "/V", Pdf.Integer 2;
           "/R", Pdf.Integer 3;
           "/O", Pdf.String owner;
```

```
              "/U", Pdf.String user;
              "/Length", Pdf.Integer 128;
              "/P", Pdf.Integer (i32toi p)]
      in
        match process_cryption false false pdf (ARC4 (128, 3)) user_pw 3 user owner p id 128 with
        | Some pdf →
            {pdf with
              Pdf.trailerdict =
                Pdf.add_dict_entry pdf.Pdf.trailerdict "/Encrypt" crypt_dict}
        | None → raise (Pdf.PDFError "Encryption failed")
```

AES Encryption.

```
let encrypt_pdf_AES encrypt_metadata user_pw owner_pw banlist pdf =
  let p = p_of_banlist banlist
  and owner = mk_owner 4 owner_pw user_pw 128
  and id, pdf = get_or_add_id pdf in
    let user = mk_user (¬ encrypt_metadata) user_pw owner p id 4 128 in
      let crypt_dict =
        Pdf.Dictionary
          ["/Filter", Pdf.Name "/Standard";
           "/V", Pdf.Integer 4;
           "/CF",
              Pdf.Dictionary
                ["/StdCF",
                   Pdf.Dictionary
                     ["/Length", Pdf.Integer 16;
                      "/AuthEvent", Pdf.Name "/DocOpen";
                      "/CFM", Pdf.Name "/AESV2"]];
           "/EncryptMetadata", Pdf.Boolean encrypt_metadata;
           "/Length", Pdf.Integer 128;
           "/R", Pdf.Integer 4;
           "/O", Pdf.String owner;
           "/U", Pdf.String user;
           "/P", Pdf.Integer (i32toi p);
           "/StrF", Pdf.Name "/StdCF";
           "/StmF", Pdf.Name "/StdCF"]
      in
        match
          process_cryption
            (¬ encrypt_metadata) true pdf AESV2 user_pw 4 user owner p id 128
        with
        | Some pdf →
            {pdf with
              Pdf.trailerdict =
                Pdf.add_dict_entry pdf.Pdf.trailerdict "/Encrypt" crypt_dict}
        | None → raise (Pdf.PDFError "Encryption failed")
```

5.5 Utility functions

Is a file encrypted?

```
let is_encrypted pdf =
  match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
  | Some _ → true
  | None → false
```

6 Module PdfDoc
Document-level functions

```
open Utility
open Pdf
```

6.1 Types

▷ The type of the four rotations of pages. This defines how a viewing application (e.g Acrobat) displays the page.

```
type rotation =
  Rotate0 | Rotate90 | Rotate180 | Rotate270
```

▷ A type representing a page. *content* is the list of objects containing the graphical content stream (see the Pdfpages module), *mediabox* the page size, *resources* the page's resource dictionary, *rotate* its rotation and *rest* any other entries to reside in the page dictionary.

```
type page =
  {content : pdfobject list;
   mediabox : pdfobject;
   resources : pdfobject;
   rotate : rotation;
   rest : pdfobject}          (* A dictionary of the other records in the page. *)
```

Make a PDF rectangle from a Paper.*papersize*.

```
let rectangle_of_paper paper =
  let u = Paper.unit paper
  and w = Paper.width paper
  and h = Paper.height paper
  in
  let w', h' =
    let f = Units.convert 100. u Units.PdfPoint in
      f w, f h
  in
    Array [Real 0.; Real 0.; Real w'; Real h']
```

▷ Create a page with empty content, media box from the given paper size, empty resources, zero rotation and no extra dictionary entries.

```
let custompage rectangle =
  {content = [];
   mediabox = rectangle;
   resources = Dictionary [];
   rotate = Rotate0;
   rest = Dictionary []}

let blankpage papersize =
  custompage (rectangle_of_paper papersize)
```

6.2 Utilities

▷ Utility function to convert from rotation to integers.

```
let int_of_rotation = function
  Rotate0 → 0 | Rotate90 → 90 | Rotate180 → 180 | Rotate270 → 270
```

▷ The reverse. raises Pdf.PDFError if its input modulo 360 is not 0, 90, 180, 270, -90, -180 or -270.

```
let rotation_of_int i =
  match i mod 360 with
  | 0 → Rotate0
  | 90 | − 270 → Rotate90
  | 180 | − 180 → Rotate180
  | 270 | − 90 → Rotate270
  | _ → raise (PDFError "Bad /Rotate")
```

6.3 Extracting the page tree

▷ Given a page tree, find the first page resources, contents and mediabox. The resources and mediabox may be inherited from any node above in the page tree.

```
let rec find_pages pages pdf resources mediabox rotate =
  match lookup_fail "/Type not in page dict" pdf "/Type" pages with
  | Name "/Pages" →
      begin match
        lookup_fail "No /Kids in page tree" pdf "/Kids" pages
      with
      | Array kids →
          let kids =
            map
              (function
               | Indirect k →
                   (try Pdf.lookup_obj pdf k with
                     Not_found → raise (PDFError "missing kid\n"))
               | _ → raise (PDFError "malformed kid\n"))
              kids
```

```
        in
          let resources =
            match lookup_direct pdf "/Resources" pages with
            | Some x → Some x
            | None → resources
          and mediabox =
            match lookup_direct pdf "/MediaBox" pages with
            | Some x → Some x
            | None → mediabox
          and rotate =
            match lookup_direct pdf "/Rotate" pages with
            | Some (Integer r) → rotation_of_int r
            | _ → rotate
          in
            flatten
              (map
                (fun k → find_pages k pdf resources mediabox rotate)
                kids)
    | _ → raise (PDFError "Malformed /Kids in page tree node")
    end
| Name "/Page" →
    let resources =
      match lookup_direct pdf "/Resources" pages with
      | Some x → Some x
      | None → resources
    and mediabox =
      match lookup_direct pdf "/MediaBox" pages with
      | Some x → Some x
      | None → mediabox
    and contents =
      lookup_direct pdf "/Contents" pages
    and rotate =
      match lookup_direct pdf "/Rotate" pages with
      | Some (Integer r) → rotation_of_int r
      | _ → rotate
    in
      [{resources =
          (match resources with
          | Some r → r
          | None → raise (PDFError "Missing /Resources"));
       content =
          (match contents with
          | None → []
          | Some (Array cs) → map (direct pdf) cs;
          | Some pdfobject →
              begin match direct pdf pdfobject with
              | Stream _ as stream → [stream]
              | _ → raise (PDFError "Bad /Contents")
              end);
       mediabox =
```

```
              (match mediabox with
              | Some m → m
              | None → raise (PDFError "Bad /MediaBox"));
            rotate = rotate;
            rest =
              fold_left remove_dict_entry pages
              ["/Resources"; "/Contents"; "/MediaBox"; "/Rotate"; "/Parent"; "/Type"]
          }]
  | _ → raise (PDFError "find_pages:  Not a page tree node or page
object")
```

▷ Given a pdf, return a list of (resources, contents, mediabox) triples.

```
let pages_of_pagetree pdf =
  let document_catalog =
    try Pdf.lookup_obj pdf pdf.root with
      Not_found → raise (PDFError "/Root entry is incorrect")
  in
    let pages =
      lookup_fail "No or malformed /Pages" pdf "/Pages" document_catalog
    in
      find_pages pages pdf None None Rotate0
```

▷ Make a collection of pages capable of being merged – in other words rename their resources so as not to clash.

```
let source k =
  let k = ref k in (fun () → incr k; !k)

let freshname source =
  "/r" ^ string_of_int (source ())

let resource_keys =
  ["/Font"; "/ExtGState"; "/ColorSpace";
   "/Pattern"; "/Shading"; "/XObject"; "/Properties"]

let make_changes pdf pages =
  let src = source 0 in
    let entries_of_page entry pageseq page =
      let entries =
        match Pdf.lookup_direct pdf entry page.resources with
        | Some (Pdf.Dictionary es) → es
        | _ → []
      in
        map (fun (k, v) → entry, pageseq, k, freshname src) entries
    in
      let pagenums = ilist 1 (length pages) in
        let entries name =
          map2 (entries_of_page name) pagenums pages
        in
          let entries = flatten <| flatten (map entries resource_keys) in
            let table = Hashtbl.create 10000 in
              iter
                (fun (entry, pageseq, k, name) →
```

```
                 Hashtbl.add table (entry, pageseq, k) name)
              entries;
            table

let change_operator lookup lookup_option seqnum = function
  | Pdfpages.Op_Tf (f, s) →
      Pdfpages.Op_Tf (lookup "/Font" seqnum f, s)
  | Pdfpages.Op_gs n →
      Pdfpages.Op_gs (lookup "/ExtGState" seqnum n)
  | Pdfpages.Op_CS n →
      begin match lookup_option "/ColorSpace" seqnum n with
      | Some x → Pdfpages.Op_CS x
      | None → Pdfpages.Op_CS n
      end
        | Pdfpages.Op_cs n →
      begin match lookup_option "/ColorSpace" seqnum n with
      | Some x → Pdfpages.Op_cs x
      | None → Pdfpages.Op_cs n
      end
        | Pdfpages.Op_SCNName (s, ns) →
      Pdfpages.Op_SCNName (lookup "/Pattern" seqnum s, ns)
  | Pdfpages.Op_scnName (s, ns) →
      Pdfpages.Op_scnName (lookup "/Pattern" seqnum s, ns)
  | Pdfpages.Op_sh s →
      Pdfpages.Op_sh (lookup "/Shading" seqnum s)
  | Pdfpages.Op_Do x →
      Pdfpages.Op_Do (lookup "/XObject" seqnum x)
  | Pdfpages.Op_DP (n, Name p) →
      Pdfpages.Op_DP (n, Name (lookup "/Properties" seqnum p))
  | Pdfpages.Op_BDC (n, Name p) →
      Pdfpages.Op_BDC (n, Name (lookup "/Properties" seqnum p))
  | x → x

let renumber_pages pdf pages =
  match pages with
  | [] → []
  | pages →
      let changes = make_changes pdf pages in
        let rec lookup_option dictname page oldkey =
          tryfind changes (dictname, page, oldkey)
        and lookup dictname page oldkey =
          try
            Hashtbl.find changes (dictname, page, oldkey)
          with
            Not_found → raise (Failure "Pdfdoc:  Bad key")
        in
        let change_content seqnum resources content =
          let operators = Pdfpages.parse_operators pdf resources content in
            let operators' =
              map (change_operator lookup lookup_option seqnum) operators
            in
```

```
        [Pdfpages.stream_of_ops operators']
and change_resources seqnum resources =
  let newdict name =
    match Pdf.lookup_direct pdf name resources with
    | Some (Pdf.Dictionary fonts) →
        Pdf.Dictionary
          (map (fun (k, v) → lookup name seqnum k, v) fonts)
    | _ → Pdf.Dictionary []
  in
    let newdicts = map newdict resource_keys in
      let resources = ref resources in
        iter2
          (fun k v →
            resources := Pdf.add_dict_entry !resources k v)
          resource_keys
          newdicts;
        !resources
in
  let process_page seqnum page =
    {page with
       content = change_content seqnum page.resources page.content;
       resources = change_resources seqnum page.resources}
  in
    map2 process_page (indx pages) pages
```

6.4 Adding a page tree

New code for better page trees
Each branch contains a list of pages to go at that branch, and pointers to two more page tree nodes. Each leaf contains just a page list. Page lists must be non-null.

Leaves and branches also hold a parent pointer, and the object number of that leaf or branch.

```
type ptree =
  | Lf of page list × int × int
  | Br of page list × ptree × ptree × int × int
```

Split a list into three equal-ish sized parts

```
let split3 l =
  match splitinto ((length l + 2) / 3) l with
  | [a; b; c] → a, b, c
  | _ → raise (Invalid_argument "split3")
```

Build the pages

```
let rec pagetree objnumsource pages parent =
  if length pages < 10 then Lf (pages, parent, objnumsource ()) else
    let left, this, right = split3 pages in
      let this_num = objnumsource () in
        let left_tree = pagetree objnumsource left this_num
        and right_tree = pagetree objnumsource right this_num in
          Br (this, left_tree, right_tree, parent, this_num)
```

Make a page. Returns, objectnumber, page pdfobject, extra objects to be added.

```
let mkpage getobjnum parent page =
  let content, extras =
    match page.content with
    | [] → [], []                          (* Null Contents not allowed. *)
    | cs →
       let indirects, objects =
          split
            (map
              (fun c →
                 let i = getobjnum () in Indirect i, (i, c))
              cs)
         in
           [("/Contents", Array indirects)], objects
  in
    let page =
      Dictionary
        ([("/Type", Name "/Page");
          ("/Parent", Indirect parent);
          ("/Resources", page.resources);
          ("/MediaBox", page.mediabox);
          ("/Rotate", Integer (int_of_rotation page.rotate))]
      @
        (match page.rest with
         | Dictionary d → d
         | _ → raise (PDFError "mkpage"))
      @
        content)
    in
      getobjnum (), page, extras
```

Build a list of objnum, pdfobject pairs from the ptree. The pages in the ptree are just missing their parent entries, so we add those.

```
let rec objects_of_ptree getobjnum extras = function
  | Lf (pages, parent, this) →
      let page_objects =
        map
         (fun (o, p, x) → extras =@ x; (o, p))
         (map (mkpage getobjnum this) pages)
      in
        let page_tree_node =
          let pdfobject =
```

```
          let parent_entry =
            if parent = 0 then [] else ["/Parent", Indirect parent]
          in
            Dictionary
              (["/Type", Name "/Pages";
                "/Kids",
                   Array (
                     map (fun x → Pdf.Indirect x) (fst <| split page_objects));
                "/Count", Integer (length pages)]
               @ parent_entry)
        in
         this, pdfobject
      in
        page_tree_node :: page_objects
  | Br (pages, left, right, parent, this) →
    let objs_left = objects_of_ptree getobjnum extras left
    and objs_right = objects_of_ptree getobjnum extras right in
      let left_num =
        match objs_left with
        | (n, _) :: _ → n
        | [] → raise (Assert_failure ("", 0, 0))
      and right_num =
        match objs_right with
        | (n, _) :: _ → n
        | [] → raise (Assert_failure ("", 0, 0))
      and count_left =
        match objs_left with
        | (_, Dictionary d) :: _ →
            begin match lookup "/Count" d with
            | Some (Integer i) → i
            | _ → raise (Assert_failure ("", 0, 0))
            end
        | _ → raise (Assert_failure ("", 0, 0))
      and count_right =
        match objs_right with
        | (_, Dictionary d) :: _ →
            begin match lookup "/Count" d with
            | Some (Integer i) → i
            | _ → raise (Assert_failure ("", 0, 0))
            end
        | _ → raise (Assert_failure ("", 0, 0))
      in
        let this_objects =
          let page_objects =
            map
             (fun (o, p, x) → extras =@ x; (o, p))
             (map (mkpage getobjnum this) pages)
          in
            let page_tree_node =
              let pdfobject =
```

```
            let parent_entry =
              if parent = 0 then [] else ["/Parent", Indirect parent]
            in
              let kids = fst <| split page_objects in
                Dictionary
                  (["/Type", Name "/Pages";
                    "/Kids",
                      Array
                        (map
                          (fun x → Pdf.Indirect x)
                          ([left_num] @ kids @ [right_num]));
                    "/Count", Integer (count_left + count_right + length kids)]
                   @ parent_entry)
          in
            this, pdfobject
        in
          page_tree_node :: page_objects
      in
        this_objects @ objs_left @ objs_right
```

▷ Take a list of pages and a PDF. Build a page tree in the PDF, returning the new pdf and the object number assigned to the top page node. All references to objects not forming part of the tree nodes themselves are left unchanged.

```
let add_pagetree pages pdf =
  let extras = ref [] in
    let getobjnum = source (Pdf.maxobjnum pdf) in
      let ptree = pagetree getobjnum pages 0 in
        let objects = objects_of_ptree getobjnum extras ptree in
          let topnode = match hd objects with (n, _) → n in
            iter (fun x → ignore (addobj_given_num pdf x)) (objects @ !extras);
            pdf, topnode
```

▷ Add a root entry, replacing the Type and Pages entry, and any entries in *extras*. Preserves any entries in any existing root (e.g Metadata pointer).

```
let add_root pageroot extras pdf =
  let existing_entries =
    try
      match Pdf.lookup_obj pdf pdf.root with
      | Dictionary d → d
      | _ → []
    with
      _ → dpr "2V"; []
  in
    let root =
      Pdf.Dictionary
        (fold_right (* Right so that /Type, /Pages overwrite *)
          (fun (k, v) d → add k v d)
            ([("/Type", Pdf.Name "/Catalog"); ("/Pages", Pdf.Indirect pageroot)] @ existing_entries)
            extras)
    in
```

```
let rootnum = Pdf.addobj pdf root in
  let trailerdict' =
    match pdf.Pdf.trailerdict with
    | Dictionary d → Dictionary (add "/Root" (Pdf.Indirect rootnum) d)
    | _ → raise (PDFError "add_root:  bad trailer dictionary")
  in
    {pdf with
       Pdf.root = rootnum;
       Pdf.trailerdict = trailerdict'}
```

Return a new PDF containing everything the old one does, but with new pages.

Other objects (e.g destinations in the document outline) may point to the individual page objects, so we must renumber these. We can only do this if the number of pages are the same. We do this **if** *replace_numbers is* **true**.

```
let change_pages change_references basepdf pages' =
  let pdf = Pdf.empty () in
    Pdf.objiter (fun k v → ignore (Pdf.addobj_given_num pdf (k, v))) basepdf;
    let old_page_numbers = Pdf.page_reference_numbers basepdf in
    let pdf, pagetree_num = add_pagetree pages' pdf in
      let pdf =
        {pdf with
           Pdf.major = basepdf.Pdf.major;
           Pdf.minor = basepdf.Pdf.minor;
           Pdf.trailerdict = basepdf.Pdf.trailerdict}
      in
        let existing_root_entries =
          try
            match Pdf.lookup_obj basepdf basepdf.root with
            | Dictionary d → d
            | _ → []
          with
          _ → dpr "2W"; []
        in
          let pdf = add_root pagetree_num existing_root_entries pdf in
            let new_page_numbers = Pdf.page_reference_numbers pdf in
              if change_references ∧ length old_page_numbers = length new_page_numbers
                then
                  let changes = combine old_page_numbers new_page_numbers in
                    Pdf.objmap
                      (Pdf.renumber_object_parsed pdf (hashtable_of_dictionary changes))
                      pdf
                else
                  pdf
```

Ensure that there are no inherited attributes in the page tree — in other words they are all explicit. This is required before writing a file with linearization

```
let pagetree_make_explicit pdf =
  let pages = pages_of_pagetree pdf in
    change_pages true pdf pages
```

let _ $=$
  Pdfwrite.$pagetree\_make\_explicit$ $:=$ $pagetree\_make\_explicit$

7 Module PDFCodec

PDF compression and decompression

```
open Utility
open Pdfio
open Pdf
```

7.1 Preliminaries

Get the next non-whitespace character in a stream.

```
let rec get_streamchar skipped s =
  match s.input_byte () with
  | x when x = Pdfio.no_more → dpr "I"; raise End_of_file
  | x →
    let chr = char_of_int x in
      if is_whitespace chr then
        begin
          incr skipped;
          get_streamchar skipped s
        end
      else chr
```

Same, but return an option type instead of raising an exception at end of input.

```
let get_streamchar_option skipped s =
  try Some (get_streamchar skipped s) with
    End_of_file → dpr "20"; None
```

▷ Raised if there was bad data.

```
exception Couldn'tDecodeStream of string
```

▷ Raised if the codec was not supported.

```
exception DecodeNotSupported
```

7.2 ASCIIHex

We build a list of decoded characters from the input stream, and then convert this to the output stream.

```
let encode_ASCIIHex stream =
  let size = stream_size stream in
    let stream′ = mkstream (size × 2 + 1) in
      sset stream′ (size × 2) (int_of_char '>');        (* '>' is end-of-data *)
      for p = 0 to size − 1 do
        let chars = explode (Printf.sprintf "%02X" (sget stream p)) in
          sset stream′ (p × 2) (int_of_char (hd chars));
          sset stream′ (p × 2 + 1) (int_of_char (hd (tl chars)))
      done;
      stream′
```

Decode ASCIIHex
Calulate a character from two hex digits a and b

```
let char_of_hex a b =
  char_of_int (int_of_string ("0x" ^ string_of_char a ^ string_of_char b))
```

```
let decode_ASCIIHex i =
  let output = ref []
  and enddata = ref false in
    try
      while ¬ !enddata do
        let b = get_streamchar (ref 0) i in
          let b′ = get_streamchar (ref 0) i in
            match b, b′ with
            | '>', _ → set enddata
            | ('0'..'9' | 'a'..'f' | 'A'..'F') as c, '>' →
                output =| char_of_hex c '0';
                set enddata
            | ('0'..'9' | 'a'..'f' | 'A'..'F' as c),
              ('0'..'9' | 'a'..'f' | 'A'..'F' as c′) →
                output =| char_of_hex c c′
            | _ → raise Not_found                        (* Bad data. *)
      done;
      bytestream_of_charlist (rev !output)
    with
      | End_of_file →
          (* We ran out of data. This is a normal exit. *)
          dpr "J";
          bytestream_of_charlist (rev !output)
      | Not_found →
          raise (Couldn'tDecodeStream "ASCIIHex")
```

7.3 ASCII85

Decode five characters.

```
let decode_5bytes (c1, c2, c3, c4, c5) =
  let d x p =
    i32mul (i32ofi (int_of_char x - 33)) (i32ofi (pow p 85))
  in
    let total =
      fold_left i32add 0l [d c1 4; d c2 3; d c3 2; d c4 1; d c5 0]
    in
      let extract t =
        char_of_int (i32toi (lsr32 (lsl32 total (24 - t)) 24))
      in
        extract 24, extract 16, extract 8, extract 0
```

Main function

```
let decode_ASCII85 i =
  let output = ref []
  and enddata = ref false
  and skipped = ref 0 in
    try
      while ¬ !enddata do
        let c1 = get_streamchar_option skipped i in
```

(∗ Ignore any whitespace skipped before getting to the first char of interest. This prevents us sliding too much back and picking up z characters twice. ∗)

```
        skipped := 0;
        let c2 = get_streamchar_option skipped i in
        let c3 = get_streamchar_option skipped i in
        let c4 = get_streamchar_option skipped i in
        let c5 = get_streamchar_option skipped i in
          let ischar c = c ≥ '!' ∧ c ≤ 'u' in
            match c1, c2, c3, c4, c5 with
            | Some 'z', _, _, _, _ →
                i.seek_in (possub (i.pos_in ()) (posadd (posofi 4) (posofi !skipped)));
                output := '\000' :: '\000' :: '\000' :: '\000' ::!output
            | Some c1, Some c2, Some c3, Some c4, Some c5 when
                ischar c1 ∧ ischar c2 ∧ ischar c3 ∧ ischar c4 ∧ ischar c5 →

                  let b1, b2, b3, b4 = decode_5bytes (c1, c2, c3, c4, c5) in
                    output := b4 :: b3 :: b2 :: b1 ::!output
            | Some '~', Some '>', _, _, _ →
                set enddata
            | Some c1, Some c2, Some '~', Some '>', _
                when ischar c1 ∧ ischar c2 →
                  let b1, b2, b3, b4 = decode_5bytes (c1, c2, '~', '>', '!') in
                    set enddata; output := b1 ::!output
            | Some c1, Some c2, Some c3, Some '~', Some '>'
                when ischar c1 ∧ ischar c2 ∧ ischar c3 →
                  let b1, b2, _, _ = decode_5bytes (c1, c2, c3, '~', '>') in
                    set enddata; output := b2 :: b1 ::!output
            | Some c1, Some c2, Some c3, Some c4, Some '~'
                when ischar c1 ∧ ischar c2 ∧ ischar c3 ∧ ischar c4 →
```

```
                  let b1, b2, b3, _ = decode_5bytes (c1, c2, c3, c4, '~') in
                    set enddata; output := b3 :: b2 :: b1 :: !output
              | _ → raise End_of_file
      done;
      bytestream_of_charlist (rev !output)
    with
      End_of_file → raise (Couldn'tDecodeStream "ASCII85")
```

Encode a single symbol set.

```
let encode_4bytes = function
  | [b1; b2; b3; b4] →
      let ( × ) = Int64.mul
      and ( − ) = Int64.sub
      and ( / ) = Int64.div
      and rem = Int64.rem in
        let numbers =
          [i64ofi (int_of_char b1) × i64ofi (pow 3 256);
           i64ofi (int_of_char b2) × i64ofi (pow 2 256);
           i64ofi (int_of_char b3) × i64ofi (pow 1 256);
           i64ofi (int_of_char b4) × i64ofi (pow 0 256)]
        in
          let t = fold_left Int64.add Int64.zero numbers
          and one85 = i64ofi (pow 1 85) and two85 = i64ofi (pow 2 85)
          and three85 = i64ofi (pow 3 85) and zero85 = i64ofi (pow 0 85)
          and four85 = i64ofi (pow 4 85) in
            let t, c5 = t − rem t one85, rem t one85 / zero85 in
              let t, c4 = t − rem t two85, rem t two85 / one85 in
                let t, c3 = t − rem t three85, rem t three85 / two85 in
                  let t, c2 = t − rem t four85, rem t four85 / three85 in
                    i64toi (t / four85), i64toi c2, i64toi c3, i64toi c4, i64toi c5
  | _ → raise (Assert_failure ("encode_4bytes", 0, 0))
```

Encode a stream.

```
let encode_ASCII85 stream =
  let output = ref []
  and enddata = ref false
  and istream = input_of_bytestream stream in
    while ¬ !enddata do
      let b1 = istream.input_char () in
      let b2 = istream.input_char () in
      let b3 = istream.input_char () in
      let b4 = istream.input_char () in
        match b1, b2, b3, b4 with
        | Some b1, Some b2, Some b3, Some b4 →
            output := [b1; b2; b3; b4] :: !output
        | Some b1, Some b2, Some b3, None →
            set enddata; output := [b1; b2; b3] :: !output
        | Some b1, Some b2, None, None →
            set enddata; output := [b1; b2] :: !output
        | Some b1, None, None, None →
```

```
        set enddata; output := [b1] ::!output
    | None, _, _, _ → set enddata
    | _ → assert false
done;
let fix k = char_of_int (k + 33) in
  let charlists' =
    rev_map
      (fun l →
         let len = length l in
           if len < 4
           then
             let l' = l @ (many '\000' (4 − len)) in
               let c1, c2, c3, c4, c5 = encode_4bytes l' in
                 take [fix c1; fix c2; fix c3; fix c4; fix c5] (len + 1)
           else
               let c1, c2, c3, c4, c5 = encode_4bytes l in
                 if c1 + c2 + c3 + c4 + c5 = 0
                   then ['z']
                   else [fix c1; fix c2; fix c3; fix c4; fix c5])
      !output
  in
    bytestream_of_charlist (flatten charlists' @ ['~'; '>'])
```

7.4 Flate

Make a bytestream from a list of strings by taking the contents, in order from the items, in order.

```
let bytestream_of_strings strings =
  let total_length =
    fold_left ( + ) 0 (map String.length strings)
  in
    let s = mkstream total_length
    and pos = ref 0 in
      iter
        (fun str →
           for x = 0 to String.length str − 1 do
             sset s !pos (int_of_char str.[x]); incr pos
           done)
        strings;
      s
```

IF-OCAML

```
let flate_process f data =
  let strings = ref []
  and pos = ref 0
  and inlength = stream_size data in
    let input =
      (fun buf →
```

```
          let s = String.length buf in
            let towrite = min (inlength - !pos) s in
              for x = 0 to towrite - 1 do
                buf.[x] <- char_of_int (sget data !pos); incr pos
              done;
              towrite)
    and output =
      (fun buf length ->
        if length > 0 then strings =| String.sub buf 0 length)
    in
      f input output;
      bytestream_of_strings (rev !strings)
```

```
let decode_flate_input i =
  let strings = ref [] in
    let input =
      (fun buf ->
        let s = String.length buf in
          if s > 0 then
            begin
              match i.input_byte () with
              | x when x = Pdfio.no_more -> raise End_of_file
              | x ->
                  buf.[0] <- char_of_int x; 1
            end
          else 0)
    and output =
      (fun buf length ->
        if length > 0 then strings =| String.sub buf 0 length)
    in
      Zlib.uncompress input output;
      bytestream_of_strings (rev !strings)
```

```
let encode_flate stream =
  flate_process Zlib.compress stream
```

```
let decode_flate stream =
  try flate_process Zlib.uncompress stream with
    Zlib.Error (a, b) -> raise (Couldn'tDecodeStream "Flate")
(*ENDIF-OCAML*)
```

7.5 LZW

Decode LZW.

```
let decode_lzw early i =
  let prefix_code = Array.make 4096 0
  and append_character = Array.make 4096 0
  and bit_count = ref 0
  and bit_buffer = ref 0l
```

```
and endflush = ref 4
and code_length = ref 9
and next_code = ref 258
and new_code = ref 0
and old_code = ref 256
and character = ref 0 in
  let rec decode_string code str =
    if code > 255 then
      decode_string prefix_code.(code) (append_character.(code) :: str)
    else
      code :: str
  and input_code stream =
    while !bit_count ≤ 24 do
      let streambyte =
        match stream.input_byte () with
        | b when b = Pdfio.no_more →
            if !endflush = 0 then raise End_of_file else (decr endflush; 0)
        | b → b
      in
        bit_buffer := lor32 !bit_buffer (lsl32 (i32ofi streambyte) (24 − !bit_count));
        bit_count += 8
    done;
    let result = Int32.to_int (lsr32 !bit_buffer (32 − !code_length)) in
      bit_buffer := lsl32 !bit_buffer !code_length;
      bit_count −= !code_length;
      result
  and strip_cleartable_codes stream =
    while !old_code = 256 do
      old_code := input_code stream
    done
  and reset_table () =
    next_code := 258;
    code_length := 9;
    old_code := 256
  in
    (∗ FIXME: How could it ever be 257? It's a byte... ∗)
    match peek_byte i with 257 → mkstream 0 | _ →
      bit_count := 0; bit_buffer := 0l;
      endflush := 4; reset_table ();
      let outstream_data = {pos = 0; data = mkstream 16034} in
        let outstream = output_of_stream outstream_data
        and finished = ref false in
          strip_cleartable_codes i;
          match !old_code with
          | 257 → mkstream 0
          | _ →
              character := !old_code;
              outstream.output_byte !old_code;
              while ¬ !finished do
                new_code := input_code i;
```

```
match !new_code with
| 257 → set finished
| 256 →
    reset_table ();
    set_array prefix_code 0;
    set_array append_character 0;
    strip_cleartable_codes i;
    character := !old_code;
    outstream.output_byte !old_code
| _ →
  let chars =
    if !new_code ≥ !next_code
      then (decode_string !old_code []) @ [!character]
      else decode_string !new_code []
  in
    character := hd chars;
    iter outstream.output_byte chars;
    prefix_code.(!next_code) ← !old_code;
    append_character.(!next_code) ← !character;
    incr next_code;
    old_code := !new_code;
    match !next_code + early with
    | 512 | 1024 | 2048 → incr code_length
    | _ → ()
done;
let out = mkstream outstream_data.pos in
  for x = 0 to stream_size out − 1 do
    sset out x (sget outstream_data.data x);
  done;
  out
```

7.6 CCITT

Decode a CCITT-encoded stream. Parameter names:

- *eol* – /EndOfLine
- *eba* – /EncodedByteAlign
- *eob* – /EndOfBlock
- *bone* – /BlackIs1
- *dra* – /DamagedRowsBeforeError
- *c* – /Columns
- *r* – /Rows

```
let rec read_white_code i =
  let a = getbitint i in
  let b = getbitint i in
  let c = getbitint i in
  let d = getbitint i in
    match a, b, c, d with
    | 0, 1, 1, 1 → 2
    | 1, 0, 0, 0 → 3
    | 1, 0, 1, 1 → 4
    | 1, 1, 0, 0 → 5
    | 1, 1, 1, 0 → 6
    | 1, 1, 1, 1 → 7
    | _ →
  let e = getbitint i in
    match a, b, c, d, e with
    | 1, 0, 0, 1, 1 → 8
    | 1, 0, 1, 0, 0 → 9
    | 0, 0, 1, 1, 1 → 10
    | 0, 1, 0, 0, 0 → 11
    | 1, 1, 0, 1, 1 → 64 + read_white_code i
    | 1, 0, 0, 1, 0 → 128 + read_white_code i
    | _ →
  let f = getbitint i in
    match a, b, c, d, e, f with
    | 0, 0, 0, 1, 1, 1 → 1
    | 0, 0, 1, 0, 0, 0 → 12
    | 0, 0, 0, 0, 1, 1 → 13
    | 1, 1, 0, 1, 0, 0 → 14
    | 1, 1, 0, 1, 0, 1 → 15
    | 1, 0, 1, 0, 1, 0 → 16
    | 1, 0, 1, 0, 1, 1 → 17
    | 0, 1, 0, 1, 1, 1 → 192 + read_white_code i
    | 0, 1, 1, 0, 0, 0 → 1664 + read_white_code i
    | _ →
  let g = getbitint i in
    match a, b, c, d, e, f, g with
    | 0, 1, 0, 0, 1, 1, 1 → 18
    | 0, 0, 0, 1, 1, 0, 0 → 19
    | 0, 0, 0, 1, 0, 0, 0 → 20
    | 0, 0, 1, 0, 1, 1, 1 → 21
    | 0, 0, 0, 0, 0, 1, 1 → 22
    | 0, 0, 0, 0, 1, 0, 0 → 23
    | 0, 1, 0, 1, 0, 0, 0 → 24
    | 0, 1, 0, 1, 0, 1, 1 → 25
    | 0, 0, 1, 0, 0, 1, 1 → 26
    | 0, 1, 0, 0, 1, 0, 0 → 27
    | 0, 0, 1, 1, 0, 0, 0 → 28
    | 0, 1, 1, 0, 1, 1, 1 → 256 + read_white_code i
    | _ →
  let h = getbitint i in
```

```
match a, b, c, d, e, f, g, h with
| 0, 0, 1, 1, 0, 1, 0, 1 → 0
| 0, 0, 0, 0, 0, 0, 1, 0 → 29
| 0, 0, 0, 0, 0, 0, 1, 1 → 30
| 0, 0, 0, 1, 1, 0, 1, 0 → 31
| 0, 0, 0, 1, 1, 0, 1, 1 → 32
| 0, 0, 0, 1, 0, 0, 1, 0 → 33
| 0, 0, 0, 1, 0, 0, 1, 1 → 34
| 0, 0, 0, 1, 0, 1, 0, 0 → 35
| 0, 0, 0, 1, 0, 1, 0, 1 → 36
| 0, 0, 0, 1, 0, 1, 1, 0 → 37
| 0, 0, 0, 1, 0, 1, 1, 1 → 38
| 0, 0, 1, 0, 1, 0, 0, 0 → 39
| 0, 0, 1, 0, 1, 0, 0, 1 → 40
| 0, 0, 1, 0, 1, 0, 1, 0 → 41
| 0, 0, 1, 0, 1, 0, 1, 1 → 42
| 0, 0, 1, 0, 1, 1, 0, 0 → 43
| 0, 0, 1, 0, 1, 1, 0, 1 → 44
| 0, 0, 0, 0, 0, 1, 0, 0 → 45
| 0, 0, 0, 0, 0, 1, 0, 1 → 46
| 0, 0, 0, 0, 1, 0, 1, 0 → 47
| 0, 0, 0, 0, 1, 0, 1, 1 → 48
| 0, 1, 0, 1, 0, 0, 1, 0 → 49
| 0, 1, 0, 1, 0, 0, 1, 1 → 50
| 0, 1, 0, 1, 0, 1, 0, 0 → 51
| 0, 1, 0, 1, 0, 1, 0, 1 → 52
| 0, 0, 1, 0, 0, 1, 0, 0 → 53
| 0, 0, 1, 0, 0, 1, 0, 1 → 54
| 0, 1, 0, 1, 1, 0, 0, 0 → 55
| 0, 1, 0, 1, 1, 0, 0, 1 → 56
| 0, 1, 0, 1, 1, 0, 1, 0 → 57
| 0, 1, 0, 1, 1, 0, 1, 1 → 58
| 0, 1, 0, 0, 1, 0, 1, 0 → 59
| 0, 1, 0, 0, 1, 0, 1, 1 → 60
| 0, 0, 1, 1, 0, 0, 1, 0 → 61
| 0, 0, 1, 1, 0, 0, 1, 1 → 62
| 0, 0, 1, 1, 0, 1, 0, 0 → 63
| 0, 0, 1, 1, 0, 1, 1, 0 → 320 + read_white_code i
| 0, 0, 1, 1, 0, 1, 1, 1 → 384 + read_white_code i
| 0, 1, 1, 0, 0, 1, 0, 0 → 448 + read_white_code i
| 0, 1, 1, 0, 0, 1, 0, 1 → 512 + read_white_code i
| 0, 1, 1, 0, 1, 0, 0, 0 → 576 + read_white_code i
| 0, 1, 1, 0, 0, 1, 1, 1 → 640 + read_white_code i
| _ →
let j = getbitint i in
  match a, b, c, d, e, f, g, h, j with
  | 0, 1, 1, 0, 0, 1, 1, 0, 0 → 704 + read_white_code i
  | 0, 1, 1, 0, 0, 1, 1, 0, 1 → 768 + read_white_code i
  | 0, 1, 1, 0, 1, 0, 0, 1, 0 → 832 + read_white_code i
  | 0, 1, 1, 0, 1, 0, 0, 1, 1 → 896 + read_white_code i
```

```
    | 0, 1, 1, 0, 1, 0, 1, 0, 0 → 960 + read_white_code i
    | 0, 1, 1, 0, 1, 0, 1, 0, 1 → 1024 + read_white_code i
    | 0, 1, 1, 0, 1, 0, 1, 1, 0 → 1088 + read_white_code i
    | 0, 1, 1, 0, 1, 0, 1, 1, 1 → 1152 + read_white_code i
    | 0, 1, 1, 0, 1, 1, 0, 0, 0 → 1216 + read_white_code i
    | 0, 1, 1, 0, 1, 1, 0, 0, 1 → 1280 + read_white_code i
    | 0, 1, 1, 0, 1, 1, 0, 1, 0 → 1344 + read_white_code i
    | 0, 1, 1, 0, 1, 1, 0, 1, 1 → 1408 + read_white_code i
    | 0, 1, 0, 0, 1, 1, 0, 0, 0 → 1472 + read_white_code i
    | 0, 1, 0, 0, 1, 1, 0, 0, 1 → 1536 + read_white_code i
    | 0, 1, 0, 0, 1, 1, 0, 1, 0 → 1600 + read_white_code i
    | 0, 1, 0, 0, 1, 1, 0, 1, 1 → 1728 + read_white_code i
    | _ →
  let k = getbitint i in
  let l = getbitint i in
    match a, b, c, d, e, f, g, h, j, k, l with
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 → 1792 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0 → 1856 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1 → 1920 + read_white_code i
    | _ →
  let m = getbitint i in
    match a, b, c, d, e, f, g, h, j, k, l, m with
    | 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 → −1
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0 → 1984 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1 → 2048 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0 → 2112 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1 → 2176 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0 → 2240 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 → 2304 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0 → 2368 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1 → 2432 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0 → 2496 + read_white_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 → 2560 + read_white_code i
    | _ → raise (Failure "bad white code")

let rec read_black_code i =
  let a = getbitint i in
  let b = getbitint i in
    match a, b with
    | 1, 1 → 2
    | 1, 0 → 3
    | _ →
  let c = getbitint i in
    match a, b, c with
    | 0, 1, 0 → 1
    | 0, 1, 1 → 4
    | _ →
  let d = getbitint i in
    match a, b, c, d with
    | 0, 0, 1, 1 → 5
```

```
    | 0, 0, 1, 0 → 6
    | _ →
let e = getbitint i in
    match a, b, c, d, e with
    | 0, 0, 0, 1, 1 → 7
    | _ →
let f = getbitint i in
    match a, b, c, d, e, f with
    | 0, 0, 0, 1, 0, 1 → 8
    | 0, 0, 0, 1, 0, 0 → 9
    | _ →
let g = getbitint i in
    match a, b, c, d, e, f, g with
    | 0, 0, 0, 0, 1, 0, 0 → 10
    | 0, 0, 0, 0, 1, 0, 1 → 11
    | 0, 0, 0, 0, 1, 1, 1 → 12
    | _ →
let h = getbitint i in
    match a, b, c, d, e, f, g, h with
    | 0, 0, 0, 0, 0, 1, 0, 0 → 13
    | 0, 0, 0, 0, 0, 1, 1, 1 → 14
    | _ →
let j = getbitint i in
    match a, b, c, d, e, f, g, h, j with
    | 0, 0, 0, 0, 1, 1, 0, 0, 0 → 15
    | _ →
let k = getbitint i in
    match a, b, c, d, e, f, g, h, j, k with
    | 0, 0, 0, 0, 1, 1, 0, 1, 1, 1 → 0
    | 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 → 16
    | 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 → 17
    | 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 → 18
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 1 → 64 + read_black_code i
    | _ →
let l = getbitint i in
    match a, b, c, d, e, f, g, h, j, k, l with
    | 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1 → 19
    | 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0 → 20
    | 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0 → 21
    | 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1 → 22
    | 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0 → 23
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 → 24
    | 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 → 25
    | 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 → 1792 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0 → 1856 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1 → 1920 + read_black_code i
    | _ →
let m = getbitint i in
    match a, b, c, d, e, f, g, h, j, k, l, m with
    | 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0 → 26
```

$$
\begin{array}{l}
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1\ \rightarrow\ 27\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0\ \rightarrow\ 28\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1\ \rightarrow\ 29\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0\ \rightarrow\ 30\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1\ \rightarrow\ 31\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0\ \rightarrow\ 32\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1\ \rightarrow\ 33\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0\ \rightarrow\ 34\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1\ \rightarrow\ 35\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 36\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1\ \rightarrow\ 37\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0\ \rightarrow\ 38\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 39\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 0,\ 0\ \rightarrow\ 40\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1\ \rightarrow\ 41\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0\ \rightarrow\ 42\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1\ \rightarrow\ 43\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 44\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1\ \rightarrow\ 45\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0\ \rightarrow\ 46\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 47\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 48\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 1\ \rightarrow\ 49\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0\ \rightarrow\ 50\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1\ \rightarrow\ 51\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 52\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 53\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0\ \rightarrow\ 54\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 55\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0\ \rightarrow\ 56\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ 0,\ 0\ \rightarrow\ 57\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1\ \rightarrow\ 58\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1,\ 1\ \rightarrow\ 59\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ 0\ \rightarrow\ 60\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0\ \rightarrow\ 61\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 0\ \rightarrow\ 62\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 63\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0\ \rightarrow\ 128\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1\ \rightarrow\ 192\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1\ \rightarrow\ 256\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0,\ 1,\ 1\ \rightarrow\ 320\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 384\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1\ \rightarrow\ 448\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1,\ 0\ \rightarrow\ 1984\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 1,\ 1\ \rightarrow\ 2048\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0\ \rightarrow\ 2112\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 0,\ 1\ \rightarrow\ 2176\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 0\ \rightarrow\ 2240\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 1,\ 1,\ 1\ \rightarrow\ 2304\ +\ read\_black\_code\ i\\
\mid 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 1,\ 0,\ 0\ \rightarrow\ 2368\ +\ read\_black\_code\ i
\end{array}
$$

```
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1 → 2432 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0 → 2496 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 → 2560 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 → −1
    | _ →
  let n = getbitint i in
    match a, b, c, d, e, f, g, h, j, k, l, m, n with
    | 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0 → 512 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1 → 576 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0 → 640 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1 → 704 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0 → 768 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1 → 832 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0 → 896 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1 → 960 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0 → 1024 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1 → 1088 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0 → 1152 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1 → 1216 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0 → 1280 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1 → 1344 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0 → 1408 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1 → 1472 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0 → 1536 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1 → 1600 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0 → 1664 + read_black_code i
    | 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1 → 1728 + read_black_code i
    | _ → raise (Failure "bad black code")
```

Group 4 Fax decoder.

```
type modes =
  | Pass
  | Horizontal
  | Vertical of int
  | Uncompressed
  | EOFB

let read_mode i =
  let a = getbitint i in
    match a with
    | 1 → Vertical 0
    | _ →
  let b = getbitint i in
  let c = getbitint i in
    match a, b, c with
    | 0, 1, 1 → Vertical −1
    | 0, 1, 0 → Vertical 1
    | 0, 0, 1 → Horizontal
    | _ →
  let d = getbitint i in
    match a, b, c, d with
```

```
      | 0, 0, 0, 1 → Pass
      | _ →
   let e = getbitint i in
   let f = getbitint i in
      match a, b, c, d, e, f with
      | 0, 0, 0, 0, 1, 1 → Vertical −2
      | 0, 0, 0, 0, 1, 0 → Vertical 2
      | _ →
   let g = getbitint i in
      match a, b, c, d, e, f, g with
      | 0, 0, 0, 0, 0, 1, 1 → Vertical −3
      | 0, 0, 0, 0, 0, 1, 0 → Vertical 3
      | _ →
   let h = getbitint i in
   let j = getbitint i in
   let k = getbitint i in
   let l = getbitint i in
   let m = getbitint i in
      match a, b, c, d, e, f, g, h, j, k, l, m with
      | 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1 → Uncompressed
      | 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 →
         let a = getbitint i in
         let b = getbitint i in
         let c = getbitint i in
         let d = getbitint i in
         let e = getbitint i in
         let f = getbitint i in
         let g = getbitint i in
         let h = getbitint i in
         let j = getbitint i in
         let k = getbitint i in
         let l = getbitint i in
         let m = getbitint i in
            begin match a, b, c, d, e, f, g, h, j, k, l, m with
            | 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 → EOFB
            | _ → raise (Failure "Not a valid code on EOFB")
            end
   | _ → raise (Failure "Not a valid code")

let decode_CCITTFax k eol eba c r eob bone dra input =
   if k > 0 then raise DecodeNotSupported else
      let whiteval, blackval = if bone then 0, 1 else 1, 0
      and output = make_write_bitstream () in
         let b = bitstream_of_input input
         and column = ref 0
         and row = ref 0
         and refline = ref (Array.make c whiteval)
         and currline = ref (Array.make c 0)
         and white = ref true
         and output_line line =
```

```
        Array.iter (putbit output) line;
        align_write output
      in
        let output_span l v =
          if l < 0 then raise (Failure "Bad CCITT stream") else
            begin
              for x = !column to !column + l - 1 do
                let r = !currline in r.(x) ← v
              done;
              column += l
            end
        and find_b1 () =
          let pos = ref !column
          and curr, opp = if !white then whiteval, blackval else blackval, whiteval in
            (* Altered to get rid of exception - test *)
            let find v =
              while
                let r = !refline in
                  if !pos ≥ 0 ∧ !pos < Array.length r then
                    r.(!pos) ≠ v
                  else
                    false
              do
                incr pos
              done; !pos
            in
              try
                (* Careful to skip imaginary black at beginning *)
                ignore (if !column = 0 ∧ !white then 0 else find curr);
                find opp
              with
                _ → dpr "2Q"; c
        and find_b2 () =
          let pos = ref !column
          and curr, opp = if !white then whiteval, blackval else blackval, whiteval in
            (* Altered to get rid of exception - test *)
            let find v =
              while
                let r = !refline in
                  if !pos ≥ 0 ∧ !pos < Array.length r then
                    r.(!pos) ≠ v
                  else
                    false
              do
                incr pos
              done; !pos
            in
              try
                (* Careful to skip imaginary black at beginning *)
                ignore (if !column = 0 ∧ !white then 0 else find curr);
```

```
          ignore (find opp);
          find curr
        with
          _ → dpr "2R"; c
  in
    try
      while true do
        if !column ≥ c then
          begin
            output_line !currline;
            refline := !currline;
            column := 0;
            set white;
            if eba then align b;
            incr row;
            if !row ≥ r ∧ r > 0 then raise End_of_file
          end
        else
         begin
            if k < 0 then
              (* Group 4 *)
              match read_mode b with
              | Pass →
                  output_span (find_b2 () − !column) (if !white then whiteval else blackval)
              | Horizontal →
                  if !white then
                    begin
                      output_span (read_white_code b) whiteval;
                      output_span (read_black_code b) blackval;
                    end
                  else
                    begin
                      output_span (read_black_code b) blackval;
                      output_span (read_white_code b) whiteval;
                    end
              | Vertical n →
                  output_span (find_b1 () − !column − n) (if !white then whiteval else blackval);
                  flip white
              | EOFB → raise End_of_file
              | Uncompressed → raise DecodeNotSupported
            else if k = 0 then
              (* Group 3 *)
              begin match (if !white then read_white_code else read_black_code) b with
              | −1 →
                 (* Pad it out *)
                 if !column > 0 then output_span (c − !column) whiteval
              | l →
                begin
                  output_span l (if !white then whiteval else blackval);
                  flip white
```

```
                end
              end
            else
              raise DecodeNotSupported
          end
      done;
      mkstream 0
    with
      | End_of_file → dpr "K"; bytestream_of_write_bitstream output
      | _ → raise (Failure "Bad CCITT Stream")
```

7.7 PNG and TIFF Predictors

Get the value at index i from an int array a, giving zero if the index is too low. Fails in the usual manner if the index is too high.

```
let get0 a i =
  if i < 0 then 0 else a.(i)
```

TIFF prediction. 8bpp only for now.

```
let decode_tiff_predictor colors bpc columns stream =
  match bpc with
  | 8 →
      let scanline_width = (colors × bpc × columns + 7) / 8 in
        for line = 0 to stream_size stream / scanline_width − 1 do
          let linestart = line × scanline_width in
            for p = 1 to scanline_width − 1 do
              sset stream (linestart + p)
                ((sget stream (linestart + p − 1) + sget stream (linestart + p)) mod 256)
            done
        done;
        stream
  | _ →
      raise DecodeNotSupported
```

Given two scanlines, the previous and current, and the predictor function p, calculate the output scanline as a list of bytes.

```
let decode_scanline_pair prior_encoded prior_decoded current pred bpc cols =
  let output = Array.copy current in
    begin match pred with
    | 0 → (* None *)
        ()
    | 1 → (* Sub *)
        for x = 0 to Array.length output − 1 do
          output.(x) ← (get0 current x + get0 output (x − cols)) mod 256
        done
    | 2 → (* Up *)
        for x = 0 to Array.length output − 1 do
          output.(x) ← (get0 current x + get0 prior_decoded x) mod 256
```

```
      done
  | 3 → (* Average – No test case yet found. *)
      for x = 0 to Array.length output − 1 do
        output.(x) ←
          (get0 current x +
            (get0 output (x − cols) + get0 prior_decoded x) / 2) mod 256
      done
  | 4 → (* Paeth *)
      let paeth a b c =
        let p = a + b − c in
          let pa = abs (p − a) and pb = abs (p − b) and pc = abs (p − c) in
            if pa ≤ pb ∧ pa ≤ pc then a
            else if pb ≤ pc then b
            else c
        in
          for x = 0 to Array.length output − 1 do
            output.(x) ←
              let curr = get0 current x
              and currback = get0 output (x − cols)
              and decoded = get0 prior_decoded x
              and decodedback = get0 prior_decoded (x − cols) in
                (curr + paeth currback decoded decodedback) mod 256
          done
  | _ → raise DecodeNotSupported
  end;
  output
```

Main function. Given predictor, number of channels, bits-per-channel, columns and the stream data, perform the decoding.

```
let decode_predictor pred colors bpc columns stream =
  if pred = 2 then decode_tiff_predictor colors bpc columns stream else
    let i = input_of_bytestream stream
    and scanline_width = (colors × bpc × columns + 7) / 8 in
      let blank () = ref (Array.make scanline_width 0) in
        let prev, curr, prior_decoded = blank (), blank (), blank ()
        and outputlines = ref []
        and finished = ref false
        and pred = ref 0
        and got_predictor = ref false in
          while ¬ !finished do
            clear got_predictor;
            begin match i.input_byte () with
            | x when x = Pdfio.no_more → set finished
            | x → pred := x
            end;
            if !finished then () else
              begin
                set got_predictor;
                prev := !curr;
                for x = 0 to scanline_width − 1 do
```

```
                match i.input_byte () with
                | x when x = Pdfio.no_more → set finished
                | b → (!curr).(x) ← b
              done
            end;
          (* We allow an unfinished final line only if we managed to get a
predictor byte *)
          if !got_predictor then
            begin
              prior_decoded :=
              decode_scanline_pair
                !prev !prior_decoded !curr !pred bpc ((bpc × colors + 7) / 8);
              outputlines =| !prior_decoded
            end
        done;
        bytestream_of_arraylist (rev !outputlines)
```

7.8 Run Length Encoding

```
let encode_runlength stream =
  let i = input_of_bytestream stream in
    let data_in = ref [] in
      begin try
        while true do
          data_in =|
            begin match i.input_byte () with
            | x when x = Pdfio.no_more → raise End_of_file
            | x → x
            end
        done
      with
        End_of_file → dpr "M"; data_in := rev !data_in
      end;
      let rec runs_of_data prev = function
        | [] → rev prev
        | h :: t →
            let same, rest = cleavewhile (eq h) (h :: t) in
              runs_of_data ((length same, hd same) :: prev) rest
      in
        let runs = ref (runs_of_data [] !data_in)
        and outbytes = ref []
        and chunksize = ref 0
        and chunkdata = ref [] in
          let writechunk () =
            if !chunksize > 0 then
              begin
                outbytes =| !chunksize − 1;
                iter (( =| ) outbytes) (rev !chunkdata);
```

```
                chunkdata := [];
                chunksize := 0;
              end
          in
              while !runs ≠ [] do
                begin match hd !runs with
                | (l, x) when l < 1 →
                    assert false
                | (l, x) when l < 3 →
                    if l + !chunksize > 128 then writechunk ();
                    chunkdata =@ many x l;
                    chunksize += l
                | (l, x) →
                    writechunk ();
                    let l = ref l in
                      while !l > 0 do
                        outbytes =| 257 − min !l 128;
                        outbytes =| x;
                        l −= 128
                      done
                end;
                runs := tl !runs
              done;
              writechunk ();
              outbytes =| 128;                          (* End-of-data *)
              bytestream_of_list (rev !outbytes)

let decode_runlength i =
  let s = {pos = 0; data = mkstream 4096} in
    let o = output_of_stream s in
      let eod = ref false in
        try
          while ¬ !eod do
            let l =
              match i.input_byte () with
              | x when x = Pdfio.no_more → raise End_of_file
              | x → x
            in
              if l < 128 then
                for x = 1 to l + 1 do
                  o.output_byte
                    begin match i.input_byte () with
                      | x when x = Pdfio.no_more → raise End_of_file
                      | x → x
                    end
                done
              else if l > 128 then
                let towrite =
                  begin match i.input_byte () with
                  | x when x = Pdfio.no_more → raise End_of_file
```

```
                | x → x
                end;
              in
                for x = 1 to 257 - l do
                  o.output_byte towrite
                done
            else
              set eod
        done;
        let osize = postoi (o.out_channel_length ()) in
          let output = mkstream osize in
            for x = 0 to osize − 1 do
              sset output x (sget s.data x)
            done;
            output
      with
        End_of_file → raise (Couldn'tDecodeStream "RunLength")
```

7.9 Decoding PDF streams

```
type source =
  | StreamSource of bytestream
  | InputSource of input

let decoder pdf dict source name =
  let input_of_source = function
    | InputSource i → i
    | StreamSource s → input_of_bytestream s
  in
    let i = input_of_source source in
      match name with
      | "/ASCIIHexDecode" | "/AHx" → decode_ASCIIHex i
      | "/ASCII85Decode" | "/A85" → decode_ASCII85 i
      | "/FlateDecode" | "/Fl" →
          begin match source with
          | StreamSource s → decode_flate s
          | InputSource i → decode_flate_input i
          end
      | "/RunLengthDecode" | "/RL" → decode_runlength i
      | "/LZWDecode" | "/LZW" →
          let early =
            match lookup_direct_orelse pdf "/DecodeParms" "/DP" dict with
            | None → 1
            | Some d →
                match lookup_direct pdf "/EarlyChange" d with
                | Some (Integer n) → n
                | None → 1
                | _ → raise (PDFError "malformed /EarlyChange")
```

```
      in
        decode_lzw early i
  | "/CCITTFaxDecode" | "/CCF" ->
      begin match lookup_direct_orelse pdf "/DecodeParms" "/DP" dict with
      | None -> decode_CCITTFax 0 false false 1728 0 true false 0 i
      | Some (Pdf.Dictionary _ as dparms)
      | Some (Array (dparms :: _)) ->
          let dparms = direct pdf dparms in
          let k =
            match lookup_direct pdf "/K" dparms with
            | Some (Integer i) -> i
            | _ -> 0
          and eol =
            match lookup_direct pdf "/EndOfLine" dparms with
            | Some (Boolean b) -> b
            | _ -> false
          and eba =
            match lookup_direct pdf "/EncodedByteAlign" dparms with
            | Some (Boolean b) -> b
            | _ -> false
          and c =
            match lookup_direct pdf "/Columns" dparms with
            | Some (Integer i) -> i
            | _ -> 1728
          and r =
            match lookup_direct pdf "/Rows" dparms with
            | Some (Integer i) -> i
            | _ -> 0
          and eob =
            match lookup_direct pdf "/EndOfBlock" dparms with
            | Some (Boolean b) -> b
            | _ -> true
          and bone =
            match lookup_direct pdf "/BlackIs1" dparms with
            | Some (Boolean b) -> b
            | _ -> false
          and dra =
            match lookup_direct pdf "/DamagedRowsBeforeError" dparms with
            | Some (Integer i) -> i
            | _ -> 0
          in
            decode_CCITTFax k eol eba c r eob bone dra i
        | _ -> raise (Pdf.PDFError "bad Decodeparms")
      end
  | _ -> raise DecodeNotSupported
```

Decode at most one stage.

```
let decode_one pdf dict source =
  match lookup_direct_orelse pdf "/Filter" "/F" dict with
  | None | Some (Array []) ->
```

```
        begin match source with
        | StreamSource s → s
        | InputSource i → raise DecodeNotSupported
        end
    | Some (Name n) | Some (Array (Name n :: _)) →
        let decoded = decoder pdf dict source n in
          let decodeparms =
            match lookup_direct_orelse pdf "/DecodeParms" "/DP" dict with
            | Some (Dictionary d)
            | Some (Array (Dictionary d :: _)) → Dictionary d
            | _ → Dictionary []
          in
            begin match lookup_direct pdf "/Predictor" decodeparms with
            | None | Some (Integer 1) → decoded
            | Some (Integer pred) →
                let colors =
                  match lookup_direct pdf "/Colors" decodeparms with
                  | Some (Integer n) → n
                  | None → 1
                  | _ → raise (PDFError "malformed /Colors")
                and bits_per_component =
                  match lookup_direct pdf "/BitsPerComponent" decodeparms with
                  | Some (Integer n) → n
                  | None → 8
                  | _ → raise (PDFError "malformed /BitsPerComponent")
                and columns =
                  match lookup_direct pdf "/Columns" decodeparms with
                  | Some (Integer n) → n
                  | None → 1
                  | _ → raise (PDFError "malformed /Columns")
                in
                  begin try
                    decode_predictor pred colors bits_per_component columns decoded
                  with
                    _ → raise (Couldn'tDecodeStream "Predictor")
                  end
            | _ → raise (PDFError "Malformed /Predictor")
            end
    | _ →
      raise (PDFError "PDF.decode:  Bad filter specification")
```

Remove a single decoder from a filter list. Also remove the first entry of a DecodeParms array

```
let remove_decoder d =
  let d' =
    match lookup "/Filter" d, lookup "/F" d with
    | None, None → d
    | Some (Name _ | Array [_]), None → lose (fun (n, _) → n = "/Filter") d
    | None, Some (Name _ | Array [_]) → lose (fun (n, _) → n = "/F") d
    | Some (Array (_ :: t)), _ → replace "/Filter" (Array t) d
```

```
    | _, Some (Array (_ :: t)) → replace "/F" (Array t) d
    | _ → raise (PDFError "PDF.remove_decoder: malformed /Filter")
  in
    match lookup "/DecodeParms" d′, lookup "/DP" d′ with
    | None, None → d′
    | Some (Dictionary _ | Array []), _ → remove "/DecodeParms" d′
    | _, Some (Dictionary _ | Array []) → remove "/DP" d′
    | Some (Array (_ :: t)), _ → replace "/DecodeParms" (Array t) d′
    | _, Some (Array (_ :: t)) → replace "/DP" (Array t) d′
    | _ → raise (PDFError "PDF.remove_decoder: malformed /DecodeParms")
```

▷ Decode at most one stage.

```
let rec decode_pdfstream_onestage pdf stream =
  getstream stream;
  match stream with
  | Stream ({contents = (Dictionary d as dict, Got s)} as stream_contents) →

      begin match direct pdf (lookup_fail "no /Length" pdf "/Length" dict) with
      | Integer l → () | _ → raise (PDFError "No /Length")
      end;
      let stream′ = decode_one pdf dict (StreamSource s) in
        let d′ =
          replace "/Length" (Integer (stream_size stream′)) (remove_decoder d)
        in
          stream_contents := Dictionary d′, Got stream′
  | _ → raise (PDFError "Pdf.decode_pdfstream: not a valid Stream")
```

▷ Decode until there's nothing left to do.

```
let rec decode_pdfstream pdf = function
  | Stream {contents = d, _} as stream →
      getstream stream;
      begin match lookup_direct_orelse pdf "/Filter" "/F" d with
      | None → ()
      | Some (Name _ | Array _) →
            begin
              decode_pdfstream_onestage pdf stream;
              match stream with
              | Stream {contents = d′, _} →
                  if d = d′ then () else
                  decode_pdfstream pdf stream
              | _ → assert false
            end
      | _ → raise (PDFError "Pdf.remove_decoder: malformed /Filter")
      end
  | Indirect i →
      decode_pdfstream pdf (Pdf.direct pdf (Indirect i))
  | _ → raise (PDFError "Pdf.decode_pdfstream: malformed Stream")
```

▷ Decode a stream until a decoding isn't supported.

```
let decode_pdfstream_until_unknown pdf s =
  try decode_pdfstream pdf s with
    DecodeNotSupported → dpr "2T"; ()
```

Decode the first decoder from an input. Any further ones can be done in the usual fashion. Fails if no decoder (you should have dealt with this already).

```
let decode_from_input i dict =
  match lookup_direct_orelse (Pdf.empty ()) "/F" "/Filter" dict with
  | Some (Name n) →
      Some (decode_one (Pdf.empty ()) dict (InputSource i))
  | Some (Array (h :: t)) →
      let stream = decode_one (Pdf.empty ()) dict (InputSource i) in
        let rec decode_rest stream = function
          | [] → stream
          | Name n :: more →
              let dict′ = remove_dict_entry dict "/Filter" in
                let dict″ = remove_dict_entry dict′ "/F" in
                  let stream′ =
                    decode_one (Pdf.empty ()) dict″ (StreamSource stream)
                  in
                    decode_rest stream′ more
          | _ → raise (PDFError "Malformed filter array")
        in
          Some (decode_rest stream t)
  | _ → raise (Couldn'tDecodeStream "No or bad filter")
```

7.10 Encoding streams

▷ Supported encodings.

```
type encoding =
  | ASCIIHex
  | ASCII85
  | RunLength
  | Flate
```

The name of an encoding.

```
let name_of_encoding = function
  | ASCIIHex → "/ASCIIHexDecode"
  | ASCII85 → "/ASCII85Decode"
  | RunLength → "/RunLengthDecode"
  | Flate → "/FlateDecode"
```

Add an encoding to the dictionary d.

```
let add_encoding length pdf encoding d =
  let filter' =
    match lookup_direct pdf "/Filter" d with
    | None →
        Name (name_of_encoding encoding)
    | Some (Name n) →
        Array (Name (name_of_encoding encoding) :: [Name n])
    | Some (Array a) →
        Array (Name (name_of_encoding encoding) :: a)
    | _ → raise (PDFError "Malformed /Filter")
  in
    replace_dict_entry (add_dict_entry d "/Filter" filter') "/Length" (Integer length)
```

Find the encoding function.

```
let encoder_of_encoding = function
  | ASCIIHex → encode_ASCIIHex
  | ASCII85 → encode_ASCII85
  | RunLength → encode_runlength
  | Flate → encode_flate
```

▷ Encode a PDF stream with an encoding.

```
let encode_pdfstream pdf encoding stream =
  getstream stream;
  match stream with
  | Stream ({contents = d, Got s} as stream) →
      let data = encoder_of_encoding encoding s in
        let d' = add_encoding (stream_size data) pdf encoding d in
          stream := d', Got data
  | _ → raise (PDFError "Pdf.encode_pdfstream:  malformed Stream")
```

8 Module PDFWrite
Flattening PDF

```
open Utility
open Pdfio

let print_ints is =
  iter (fun x → print_int x; print_string " ") is;
  print_newline ()
```

Flatten a PDF data structure to an output. The specification suggests restricting lines to 255 characters for compatibility with very old PDF software; we don't currently do this.

8.1 Utilities

Renumber a PDF's objects to $1 \ldots n$.
Calculate the substitutions required to renumber the document.

```
let changes pdf =
  let card = Pdf.objcard pdf in
    let order = ilist_fail_null 1 card
    and change_table = Hashtbl.create card in
      List.iter2 (Hashtbl.add change_table) (Pdf.objnumbers pdf) order;
      change_table
```

8.2 Header and Cross-reference table.

The file header. We include four larger-than-127 bytes as requested by the standard to help FTP programs distinguish binary/text transfer modes.

```
let header pdf =
  "%PDF-" ^
  string_of_int pdf.Pdf.major ^
  "." ^
  string_of_int pdf.Pdf.minor ^
  "\n%\128\129\130\131\n"
```

Build an cross-reference table string.

```
let pad_to_ten ch s =
  let l = String.length s in
    if l > 10 then
      (* int64 values could be too big *)
      raise (Pdf.PDFError "xref too big")
    else
      (fold_left ( ^ ) "" (many ch (10 - l))) ^ s

let string_of_xref n =
  pad_to_ten "0" ((*IF-OCAML*)Int64.to_string(*ENDIF-OCAML*)n) ^ " 00000
n \n"
```

Write the cross-reference table to a channel. xrefs is a list of positions, -1L meaning a free entry.

```
let write_xrefs xrefs i =
  let os = output_string i in
    os "xref\n";
    os ("0 " ^ string_of_int (length xrefs + 1) ^ " \n");
    os "0000000000 65535 f \n";
    iter os (map string_of_xref xrefs)
```

8.3 PDF Strings

Convert a string to one suitable for output. The function *escape* escapes parentheses and backslashes.

```
let make_pdf_string s =
  let rec escape = function
    | [] → []
    | ('(' | ')' | '\\') as c :: cs → '\\' :: c :: escape cs
    | '\n' :: cs → '\\' :: 'n' :: escape cs
    | '\r' :: cs → '\\' :: 'r' :: escape cs
    | '\t' :: cs → '\\' :: 't' :: escape cs
    | '\b' :: cs → '\\' :: 'b' :: escape cs
    | '\012' :: cs → '\\' :: 'f' :: escape cs
    | c :: cs → c :: escape cs
  and enclose s = "(" ^ s ^ ")" in
    enclose (implode (escape (explode s)))
```

8.4 Flattening PDF to strings

We have two kinds of flat data to write: Strings and streams (we cannot represent streams as strings, since there is a langauge limit on the length of strings.

```
type writeout =
  | WString of string
  | WStream of Pdf.stream
```

We want real numbers with no exponents (format compliance), and no trailing zeroes (compactness).

```
let format_real = Printf.sprintf "%f"
```

```
(* Character codes in a name < 33 or > 126 are replaced with hashed combinations (e.g #20 for space). If the name contains the null character, an exception is raised. *)
let rec make_pdf_name_inner prev = function
  | [] → rev prev
  | '\000' :: _ →
      raise (Pdf.PDFError "Name cannot contain the null character")
  | h :: t when
      h < '\033' ∨ h > '\126' ∨ Pdf.is_delimiter h ∨ h = '#'
    →
      let chars =
        '#'::explode (Printf.sprintf "%X" (int_of_char h))
      in
        make_pdf_name_inner (rev chars @ prev) t
  | h :: t → make_pdf_name_inner (h :: prev) t
```

See if a name needs altering by *make_pdf_name_inner*.

```
let needs_processing s =
  let result = ref false in
    String.iter
      (function
       | '\000' → raise (Pdf.PDFError "Name cannot contain the null
character")
       | x when x < '\033' ∨ x > '\126' ∨ Pdf.is_delimiter x ∨
 x = '#' → set result
       | _ → ())
      s;
    !result

let make_pdf_name n =
  if needs_processing n then
    match explode n with
    | '/' :: more → "/" ^ (implode <| make_pdf_name_inner [] more)
    | _ → raise (Pdf.PDFError "bad name")
  else
    n
```

Calculate a strings and streams representing the given pdf datatype instance, assuming it has no unresolved indirect references.

```
let rec strings_of_pdf = function
  | Pdf.Null → [WString "null"]
  | Pdf.Boolean b → [WString (string_of_bool b)]
  | Pdf.Integer n → [WString (string_of_int n)]
  | Pdf.Real r → [WString (format_real r)]
  | Pdf.String s → [WString (make_pdf_string s)]
  | Pdf.Name n → [WString (make_pdf_name n)]
```

```
  | Pdf.Array elts →
      let strings =
        map
          (function
            | WString x → WString (x ^ " ")
            | _ → raise (Pdf.PDFError "direct stream object"))
          (flatten (map strings_of_pdf elts))
      in
        [WString "[ "] @ strings @ [WString "]"]
  | Pdf.Dictionary entries →
      let strings =
        map
          (fun (k, v) →
              [WString (make_pdf_name k ^ " ")] @
              strings_of_pdf v
              @ [WString " "])
          entries
      in
        [WString "<< "] @ flatten strings @ [WString ">>"]
  | Pdf.Stream {contents = (dict, data)} →
      strings_of_pdf dict @
        [(WString "\010stream\010"); (WStream data); (WString "\010endstream")]
  | Pdf.Indirect n →
      [WString (string_of_int n ^ " 0 R")]
```

▷ Produce a single string from a PDF object. Only use for things which will always fall under the string size limit.

```
let string_of_pdf s =
  let strings =
    map (function (WString x) → x | _ → "") (strings_of_pdf s)
  in
    fold_left ( ^ ) "" (interleave " " strings)

let string_of_pdf_obj pdf o =
  Printf.sprintf "OBJECT %i\n" o ^
  string_of_pdf (Pdf.lookup_obj pdf o)
```

Calculate strings, one for each indirect object in the body.

```
let strings_of_object (n, pdfobject) =
  [WString (string_of_int n ^ " 0 obj\n")] @
  strings_of_pdf pdfobject @
  [WString "\nendobj\n"]
```

8.5 Stream output

Output a stream.

```
let output_stream o s =
  Pdf.getstream s;
  match s with
  | Pdf.Stream {contents = _, Pdf.Got arr} ->
      if stream_size arr > 0 then
        for i = 0 to stream_size arr - 1 do
          o.output_byte (sget arr i)
        done
  | _ -> raise (Pdf.PDFError "output_stream")
```

8.6 Encrypting a PDF while writing

```
type encryption_method =
  | PDF40bit
  | PDF128bit
  | AES128bit of bool          (* true = encrypt metadata, false = don't. *)

type encryption =
  {encryption_method : encryption_method;
   owner_password : string;
   user_password : string;
   permissions : Pdfcrypt.permission list}

let crypt_if_necessary pdf = function
  | None -> pdf
  | Some enc ->
      let f =
        match enc.encryption_method with
        | PDF40bit -> Pdfcrypt.encrypt_pdf_40bit
        | PDF128bit -> Pdfcrypt.encrypt_pdf_128bit
        | AES128bit em -> Pdfcrypt.encrypt_pdf_AES em
      in
        f enc.user_password enc.owner_password enc.permissions pdf
```

8.7 Linearized (Fast Web View) writing

The Part 6 (First Page Section) object numbers. (1) Page object for first page (2) Outline hierarchy, if PageMode is UseOutlines (3) All objects the page object refers to, except page nodes or other page objects

```
let part6_parts_of_pdf pdf =
  let catalog = Pdf.catalog_of_pdf pdf in
    let first_page_objnum =
      match Pdf.page_reference_numbers pdf with
      | [] -> raise (Pdf.PDFError "No pages in document")
      | i :: _ -> i
    in
```

```
let outline_objnums =
  match Pdf.lookup_direct pdf "/PageMode" catalog with
  | Some (Pdf.Name "/UseOutlines") →
      Pdf.reference_numbers_of_dict_entry pdf catalog "/Outlines"
  | _ → []
in
  let referenced_from_page =
    Pdf.objects_referenced
      ["/Thumb"] [("/Type", Pdf.Name "/Page"); ("/Type", Pdf.Name "/Pages")]
      pdf (Pdf.lookup_obj pdf first_page_objnum)
  in
    setify_preserving_order
      (first_page_objnum :: outline_objnums @ referenced_from_page)
```

The Part 4 (Catalog and Document-Level Objects) object numbers.

```
let part4_parts_of_pdf pdf =
  let catalog_num =
    match pdf.Pdf.trailerdict with
    | Pdf.Dictionary d →
        begin match lookup "/Root" d with
        | Some (Pdf.Indirect i) → i
        | _ → raise (Pdf.PDFError "Bad catalog")
        end
    | _ → raise (Pdf.PDFError "Bad catalog")
  in
    let catalog = Pdf.catalog_of_pdf pdf in
      let indirects_from_catalog no_follow_entries no_follow_contains entry =
        match catalog with
        | Pdf.Dictionary d →
            begin match lookup entry d with
            | Some v →
                Pdf.objects_referenced no_follow_entries no_follow_contains pdf v
            | _ → []
            end
        | _ → raise (Pdf.PDFError "bad catalog")
      in
        let sources_follow =
          ["/ViewerPreferences"; "/PageMode"; "/Threads"; "/OpenAction"; "/Encrypt"]
        in
          let objnum_of_acroform =
            match catalog with
            | Pdf.Dictionary d →
                begin match lookup "/AcroForm" d with
                | Some (Pdf.Indirect i) → [i]
                | _ → []
                end
            | _ → []
          in
            (* Catalog number is the head. *)
            setify_preserving_order
```

```
          (catalog_num ::
            flatten
              (map (indirects_from_catalog
              ["/Parent"]
              ["/Type", Pdf.Name "/Page"; "/Type", Pdf.Name "/Pages"]) sources_follow) @
              objnum_of_acroform)
```

Part 7: For each page, objects reachable from this page which are reachable from no others; Part 8: Objects referenced from ¿ 1 page; Part 9: Anything else

```
let print_nums ls =
  iter (Printf.printf "%i ") ls;
  flprint "\n"

let get_main_parts p3nums pdf =
  let objects_left = setminus (Pdf.objnumbers pdf) p3nums
  and pagenums =
    match Pdf.page_reference_numbers pdf with
    | [] → raise (Pdf.PDFError "This PDF has no pages")
    | _ :: t → t
  in
    let pages = map (Pdf.lookup_obj pdf) pagenums in
      let objects_from_each_page =
        map
          (Pdf.objects_referenced
            ["/Thumb"; "/Parent"]
            [("/Type", Pdf.Name "/Page"); ("/Type", Pdf.Name "/Pages")]
            pdf)
          pages
      in
              let histogram =
          collate compare (sort compare (flatten objects_from_each_page))
        in
                  let shared_objects =
            flatten
              (map (function x → [hd x])
                (keep (function [] | [_] → false | _ → true) histogram))
          in
            let shared_objects = setminus shared_objects p3nums in
                          let unshared_lists =
                map (lose (mem' shared_objects)) objects_from_each_page
              in
                (* Put them in order (page object first) and flatten *)
                let part7_pages =
                  map2 (fun p l → p :: lose (eq p) l) pagenums unshared_lists
                in
                  let unshared_objects = flatten part7_pages in
                    let unshared_objects = setminus unshared_objects p3nums in
                      let part9 =
                        setminus (setminus objects_left shared_objects) unshared_objects
                      in
                        part7_pages, unshared_objects, shared_objects, part9
```

We output 10-character blanks XXXXXXXXXX, overwriting them when we know the values, at the end of the process.
Return all trailerdict entries except for size, root and prev, as a partial dictionary entry list represented as a string. Number changes will need to have occured for everything in the trailerdict by now, since we're creating X O R references to them...

```
let rest_of_trailerdict_entries pdf =
  let str =
    string_of_pdf
      (fold_left Pdf.remove_dict_entry pdf.Pdf.trailerdict ["/Prev"; "/Size"; "/Root"])
  in
    implode (rev (tl (tl (rev (tl (tl (explode str)))))))

let flatten_W o = function
 | WString s → output_string o s
 | WStream data → output_stream o (Pdf.Stream {contents = Pdf.Null, data})
```

Renumber old numbers to new ones, renumbering any other objects in the PDF which clash. Returns the new PDF.

```
let lin_changes old_nums new_nums pdf =
  assert (length old_nums = length new_nums);
  if old_nums = [] then hashtable_of_dictionary [] else
    let clash_changes =
      let maxnum = Pdf.maxobjnum pdf + 1 in
        let new_objnums = ilist maxnum (maxnum + length new_nums − 1) in
          combine new_nums new_objnums
    in
      let changes = clash_changes @ combine old_nums new_nums in
        hashtable_of_dictionary changes

let lin_renumber old_nums new_nums pdf =
  assert (length old_nums = length new_nums);
  match new_nums with
  | [] → pdf
  | _ → Pdf.renumber (lin_changes old_nums new_nums pdf) pdf
```

Rember the items in l according to the (parital) changes given.

```
let list_renumber old_nums new_nums pdf l =
  let changes = lin_changes old_nums new_nums pdf in
    map (fun x → match tryfind changes x with Some y → y | None →
 x) l
```

List of (object number, final position of object in file) pairs

```
type xrefblank =
  | PDFObj of int
  | LinearizationDictionaryPosition
  | PrimaryHintStreamPosition
  | FileLength
  | HintOffset
  | HintLength
  | EndOfFirstPage
```

```
  | MainXRefTableFirstEntry
  | Prev
```

Replace the markers with the (now calculated) contents

```
let replace_xs o object_positions x_positions specials =
  iter
    (function
     | PDFObj i, xpos →
         begin match lookup i !object_positions with
         | Some pos →
             o.seek_out xpos;
             output_string o (pad_to_ten "0"
             ((*IF-OCAML*)Int64.to_string(*ENDIF-OCAML*)pos))
         | None → raise (Pdf.PDFError "Linearization inconsistency")
         end
     | other, xpos →
         let pad =
           match other with
           | LinearizationDictionaryPosition
           | PrimaryHintStreamPosition → "0"
           | _ → " "
         in
           match lookup other !specials with
           | Some pos →
               o.seek_out xpos;
               output_string o (pad_to_ten pad
               ((*IF-OCAML*)Int64.to_string(*ENDIF-OCAML*)pos))
           | _ → ())
    !x_positions
```

Outputting specials markers

```
let output_special_xref_line o xrefblank x_positions =
  x_positions =| (xrefblank, o.pos_out ());
  output_string o "XXXXXXXXXX 00000 n \n"

let output_xref_line o x_positions objnum =
  output_special_xref_line o (PDFObj objnum) x_positions

let output_special o xrefblank x_positions =
  x_positions =| (xrefblank, o.pos_out ());
  output_string o "XXXXXXXXXX"
```

The minimum number of bits needed to represent the number given.

```
let bits_needed n =
  if n = 0 then 0 else log2of (pow2lt n × 2)
```

The number of bytes which an object will use up in the file.

```
let object_bytes pdf objnum =
  let strings = strings_of_object (objnum, Pdf.lookup_obj pdf objnum)
  and length_of_string = function
    | WString s -> String.length s
    | WStream (Pdf.Got data) -> stream_size data
    | WStream (Pdf.ToGet (_, _, length)) -> i64toi length
  in
    fold_left ( + ) 0 (map length_of_string strings)
```

Same for list of objects

```
let objects_bytes pdf objs =
  fold_left ( + ) 0 (map (object_bytes pdf) objs)
```

Calculates a bitstream representing the page offset hint table.

```
let page_offset_hint_table pdf pages first_page_objects shared_objects object_positions =
  assert (length pages > 0);
  let objects_reachable_from_each_page =
    let referenced page_objnum =
      Pdf.objects_referenced
        [] [("/Type", Pdf.Name "/Page"); ("/Type", Pdf.Name "/Pages")]
        pdf (Pdf.lookup_obj pdf page_objnum)
    in
      map
        (function p -> keep (mem' shared_objects) (setify (referenced <
| hd p)))
        pages
  in
  let page_lengths = map length pages
  and page_byte_lengths = map (objects_bytes pdf) pages in
  let least_in_page = hd (sort compare page_lengths)
  and most_in_page = hd (sort rev_compare page_lengths)
  and least_bytes_in_page = hd (sort compare page_byte_lengths)
  and most_bytes_in_page = hd (sort rev_compare page_byte_lengths) in
  (* Least number of objects in a page *)
  let item1 = least_in_page
  (* Location of first page's page object *)
  and item2 = (*IF-OCAML*)i64toi(*ENDIF-OCAML*) (lookup_failnull (hd (hd pages)) !object_positio
  (* Number of bits needed to represent the difference between the greatest
and least number of objects in a page *)
  and item3 = bits_needed (most_in_page - least_in_page)
  (* Least length of a page in the file in bytes *)
  and item4 = least_bytes_in_page
  (* Number of bits needed to represent the difference between the greatest
and least length of a page in the file in bytes *)
  and item5 = bits_needed (most_bytes_in_page - least_bytes_in_page)
  (* Number of bits needed to represent the greatest number of shared object
references. (in other words, in part 8) *)
  and item10 =
    bits_needed (hd (sort rev_compare
    (length (hd pages) :: map length objects_reachable_from_each_page)))
```

```
  (* Number of bits needed to represent the numerically greatest shared object
identifier used by the pages *)
  and item11 = bits_needed (max 0 (length shared_objects + length first_page_objects − 1))
  (* Number of bits needed to represent the numerator of the fractional position
for each shared object reference. *)
  and item12 = 1
  (* The denominator of the fractional position for each shared object reference.
*)
  and item13 = 1
  and b = make_write_bitstream () in
        (* Write the header *)
    putval b 32 (i32ofi item1);
    putval b 32 (i32ofi item2);
    putval b 16 (i32ofi item3);
    putval b 32 (i32ofi item4);
    putval b 16 (i32ofi item5);
    putval b 32 0l;
    putval b 16 0l;
    putval b 32 0l;
    putval b 16 (i32ofi item5);
    putval b 16 (i32ofi item10);
    putval b 16 (i32ofi item11);
    putval b 16 (i32ofi item12);
    putval b 16 (i32ofi item13);
    (* Now the per-page entries *)
    (* Items 1 *)
    for x = 1 to length pages do
      putval b item3 (i32ofi (length (select x pages) − item1))
    done;
    (* Item 2 *)
    for x = 1 to length pages do
      putval b item5 (i32ofi (select x page_byte_lengths − item4))
    done;
    (* Item 3 *)
    for x = 1 to length pages do
      if x = 1 then
        if length pages > 1
          then putval b item10 0l
          else putval b item10 (i32ofi (length (hd pages)))
      else
        putval b item10 (i32ofi (length (select x objects_reachable_from_each_page)))
    done;
    (* Item 4 *)
    for x = 1 to length pages do
      if x = 1 ∧ length pages > 1 then () else
        let shared_objects_reachable =
          select x objects_reachable_from_each_page
        in
          let table =
            let all_objs = first_page_objects @ shared_objects in
```

```
            hashtable_of_dictionary (combine all_objs (indx all_objs))
        in
          iter
            (fun s →
              putval b item11 (i32ofi (Hashtbl.find table s)))
            shared_objects_reachable
  done;
  (* Item 5 *)
  for x = 1 to length pages do
    if x = 1 ∧ length pages > 1 then () else
      let shared_objects_reachable =
        select x objects_reachable_from_each_page
      in
        for y = 1 to length shared_objects_reachable do
          putval b item12 0l (* Always use 0 / 1 fraction *)
        done
  done;
  (* Item 7 (No item 6) *)
  for x = 1 to length pages do
    putval b item5 0l (* Ignored *)
  done;
  b
```

Shared object hint table

```
let shared_object_hint_table
  pdf first_page_objects shared_objects shared_object_positions
=
  assert (length shared_objects = length shared_object_positions);
      let lengths_of_shared_objects =
    map (object_bytes pdf) (shared_objects @ first_page_objects)
  in
    let least =
      match sort compare lengths_of_shared_objects with
      | [] → 0
      | h :: _ → h
    and greatest =
      match sort rev_compare lengths_of_shared_objects with
      | [] → 0
      | h :: _ → h
    in
      let b = make_write_bitstream () in
        (* Object number of first object in shared objects section *)
        let item1 = match shared_objects with [] → 0 | h :: _ → h
        (* Location of the first object in the shared objects section *)
        and item2 = 0 (* The number of shared object entries for the first
page (including unshared objects *)
        and item3 = length first_page_objects
        and item4 = length first_page_objects + length shared_objects
        (* The least length of a shared object group in bytes (= least length of
```

```
an object in bytes) *)
      and item6 = least
      (* Number of bits required to encode the difference between the
greatest and smallest length of an shared object group (=object) in bytes *)
      and item7 = bits_needed (greatest - least)
      in
        putval b 32 (i32ofi item1);
        putval b 32 (i32ofi item2);
        putval b 32 (i32ofi item3);
        putval b 32 (i32ofi item4);
        putval b 16 0l;
        putval b 32 (i32ofi item6);
        putval b 16 (i32ofi item7);
                  (* Main Section, Sequence One (First Page Objects) *)
        (* Item 1s (byte lengths) *)
        iter
          (fun x ->
            let len = object_bytes pdf x - item6 in
              putval b item7 (i32ofi len))
          first_page_objects;
        (* Item 2s *)
        iter (function _ -> putval b 1 0l) first_page_objects;
        (* Item 4s *)
        iter (function _ -> putval b 0 0l) first_page_objects;
        (* Main Section, Sequence Two (Shared Objects (Part 8)) *)
        (* Item 1s *)
        iter
          (fun x ->
            let len = object_bytes pdf x - item6 in
              putval b item7 (i32ofi len))
          shared_objects;
        (* Item 2s *)
        iter (function _ -> putval b 1 0l) shared_objects;
        (* Item 4s *)
        iter (function _ -> putval b 0 0l) shared_objects;
        b
```

This is filled in by the Pdfdoc module at code-loading time. It remains static thereafter.

```
let pagetree_make_explicit = ref ident
```

OBJECT NUMBERS: 1..n Objects not related to the first page n+1 Linearization dictionary n+2 Catalog n+3 First page's page object n+4..m Rest of first page and related content m + 1 Primary hint stream.

```
let pdf_to_output_linearized encrypt pdf o =
  let specials = ref []
  and object_positions = ref []
  and x_positions = ref [] in
  let pdf = !pagetree_make_explicit pdf in
  Pdf.remove_unreferenced pdf;
```

```
let writeobj pdf p =
  let obj =
    try Pdf.lookup_obj pdf p with
      | Not_found -> dpr "3N"; Pdf.Null
  in
    object_positions =| (p, o.pos_out ());
    iter
      (flatten_W o)
      (strings_of_object (p, obj))
in
let p4objs = part4_parts_of_pdf pdf
  (* First object is catalog *)
and p6objs = part6_parts_of_pdf pdf in
  (* First object is first page's page object number *)
assert (length p4objs > 0 && length p6objs > 0);
let objects_in_rest_of_file =
  Pdf.objcard pdf - length p4objs - length p6objs
in
(* Part 1: Header *)
output_string o (header pdf);
(* Part 2: Linearization parameter dictionary *)
let lin_dict_obj_number = objects_in_rest_of_file + 1 in
specials =| (LinearizationDictionaryPosition, o.pos_out ());
output_string o
(string_of_int lin_dict_obj_number ^ " 0 obj\n<< /Linearized 1.0\n/L
");
output_special o FileLength x_positions;
output_string o "\n/H [ ";
output_special o HintOffset x_positions;
output_string o " ";
output_special o HintLength x_positions;
output_string o "]\n";
output_string o ("/O " ^ string_of_int (objects_in_rest_of_file + 3) ^ "\n");
output_string o "/E ";
output_special o EndOfFirstPage x_positions;
output_string o
("\n/N " ^ (string_of_int (length (Pdf.page_reference_numbers pdf))) ^ "\n/T");
output_special o MainXRefTableFirstEntry x_positions;
output_string o "\n>>\nendobj\n";
(* Part 3: First page cross-reference table and trailer *)
let p3length = length p4objs + length p6objs + 2 in
let p3nums =
  if p3length = 0 then [] else
    ilist_null
      (objects_in_rest_of_file + 2)
      (objects_in_rest_of_file + 2 + length p4objs + length p6objs - 1)
in
let order = (hd p4objs :: hd p6objs :: tl p4objs @ tl p6objs) in
let new_p6objs = list_renumber order p3nums pdf p6objs in
let pdf = lin_renumber order p3nums pdf in
```

```
  let p7_pages, p7nums, p8nums, p9nums = get_main_parts p3nums pdf in
  let p7length = objects_bytes pdf p7nums in
  let p8lengths = map (object_bytes pdf) p8nums in
  let main_nums = p7nums @ p8nums @ p9nums in
  let new_main_nums =
    if length main_nums > 0 then ilist 1 (length main_nums) else []
  in
  let list_renumber = list_renumber main_nums new_main_nums pdf in
  let p7_pages = map list_renumber p7_pages in
  let new_p6objs = list_renumber new_p6objs
  and new_p8nums = list_renumber p8nums in
  let pdf = lin_renumber main_nums new_main_nums pdf in
  let pdf = crypt_if_necessary pdf encrypt in
  let position_of_first_page_xref_table = o.pos_out () in
  output_string o
("xref\n" ^ string_of_int (objects_in_rest_of_file + 1) ^
" " ^ string_of_int p3length ^ " \n");
  output_special_xref_line o LinearizationDictionaryPosition x_positions;
  iter (output_xref_line o x_positions) p3nums;
  output_special_xref_line o PrimaryHintStreamPosition x_positions;
  output_string o
("trailer\n << /Size " ^ string_of_int (Pdf.objcard pdf + 3) ^ " /Prev
");
  output_special o Prev x_positions;
  output_string o
(" /Root " ^ string_of_int (objects_in_rest_of_file + 2) ^
" 0 R " ^ rest_of_trailerdict_entries pdf ^ ">>\n" ^ "startxref\n0\n%%EOF\n");
  (* Part 4 and Part 6: Document-level and first page *)
  iter (writeobj pdf) p3nums;
  specials =| (EndOfFirstPage, o.pos_out ());
  (* Part 5: Primary hint stream *)
  let all_pages = tl p3nums :: p7_pages in
  let p8positions = cumulative_sum (p7length + (*IF-OCAML*)i64toi(*ENDIF-
OCAML*) (o.pos_out ())) p8lengths in
  let offset_table = page_offset_hint_table pdf all_pages new_p6objs new_p8nums object_positions in
  let shared_table = shared_object_hint_table pdf new_p6objs new_p8nums p8positions in
  let stream_content =
    bytestream_of_write_bitstream <|
    write_bitstream_append_aligned offset_table shared_table
  in
  let hintstream_dict =
    Pdf.Dictionary
      [("/Length", Pdf.Integer (stream_size stream_content));
       ("/S", Pdf.Integer (stream_size (bytestream_of_write_bitstream offset_table)))]
  in
  let stream_wstrings =
    strings_of_pdf
      (Pdf.Stream (ref (hintstream_dict, Pdf.Got (stream_content))))
  and hint_num = Pdf.objcard pdf + 2 in
  let hs_offset = o.pos_out () in
```

```
specials =| (PrimaryHintStreamPosition, hs_offset);
specials =| (HintOffset, hs_offset);
output_string o ((string_of_int hint_num) ^ " 0 obj\n");
iter (flatten_W o) stream_wstrings;
output_string o "\nendobj\n";
let hs_length = (*IF-OCAML*)i64sub(*ENDIF-OCAML*)(o.pos_out ()) hs_offset in
specials =| (HintLength, hs_length);
(* Parts 7, 8 and 9: Remaining pages and other objects. *)
iter (writeobj pdf) new_main_nums;
(* Part 11: Main cross-reference table and trailer *)
specials =| (Prev, o.pos_out ());
let main_size = length p7nums + length p8nums + length p9nums + 1 in
output_string o ("xref\n0 " ^ string_of_int main_size ^ "\n");
specials =| (MainXRefTableFirstEntry, o.pos_out ());
output_string o ("0000000000 65536 f \n");
iter (output_xref_line o x_positions) new_main_nums;
output_string o ("trailer\n<< /Size " ^ string_of_int main_size ^ " >>\nstartxref\n");
output_string o ((*IF-OCAML*)Int64.to_string(*ENDIF-OCAML*)position_of_first_page_xref_table);
output_string o "\n%%EOF\n";
specials =| (FileLength, o.pos_out ());
replace_xs o object_positions x_positions specials
```

8.8 Main functions

▷ Flatten a PDF document to an Io.*output*.

```
let pdf_to_output linearize encrypt pdf o =
  if linearize then pdf_to_output_linearized encrypt pdf o else
    let pdf = Pdf.renumber (changes pdf) pdf in
      let pdf = crypt_if_necessary pdf encrypt in
        output_string o (header pdf);
        let xrefs = ref [] in
          Pdf.objiter
            (fun ob p →
                let strings = strings_of_object (ob, p) in
                  xrefs =| o.pos_out ();
                  iter (flatten_W o) strings)
            pdf;
          let xrefstart = o.pos_out () in
            write_xrefs (rev !xrefs) o;
            output_string o "trailer\n";
            let trailerdict' =
              match pdf.Pdf.trailerdict with
              | Pdf.Dictionary trailerdict →
                  Pdf.Dictionary
                    (add "/Size" (Pdf.Integer (length !xrefs + 1))
                      (add "/Root" (Pdf.Indirect pdf.Pdf.root) trailerdict))
              | _ →
                  raise
```

```
        (Pdf.PDFError "Pdf.pdf_to_channel:  Bad trailer dictionary")
    in
      iter (flatten_W o) (strings_of_pdf trailerdict');
      output_string o
        ("\nstartxref\n" ^
        (*IF-OCAML*)Int64.to_string(*ENDIF-OCAML*)xrefstart ^ "\n%%EOF\n")

let change_id pdf f =
  match pdf.Pdf.trailerdict with
  | Pdf.Dictionary d ->
      {pdf with
         Pdf.trailerdict = Pdf.Dictionary (add "/ID" (Pdf.generate_id pdf f) d)}
  | _ -> raise (Pdf.PDFError "Bad trailer dictionary")
```

▷ Write a PDF to a channel. Don't use mk_id when the file is encrypted.

```
let pdf_to_channel linearize encrypt mk_id pdf ch =
  let pdf =
    if mk_id then change_id pdf "" else pdf
  in
    pdf_to_output linearize encrypt pdf (output_of_channel ch)
```

▷ Similarly to a named file. If mk_id is set, the /ID entry in the document's trailer dictionary is updated using the current date and time and the filename. Don't use mk_id when the file is encrypted.

```
let pdf_to_file_options linearize encrypt mk_id pdf f =
  let pdf' =
    if mk_id then change_id pdf f else pdf
  in
    let ch = open_out_bin f in
      pdf_to_channel linearize encrypt false pdf' ch;
      close_out ch

let pdf_to_file pdf f =
  pdf_to_file_options false None true pdf f
```

9 Module PDFRead
Reading PDF from File

This module can read PDF files into the format given by the Pdf module. It supports PDF versions 1.0–1.7. The commentary is not in itself sufficient for full understanding: you must read this together with the Adobe PDF Reference Manual. Section numbers are from the Fifth Edition.

```
open Utility
open Pdfio
```

Bring Pdf data constructors and functions up to top level.

```
open Pdf
```

▷ Errors in low-level functions, errors in lexing, errors in parsing.

```
exception PDFReadError of string
exception PDFLexError of string
exception PDFParseError of string
```

▷ Errors in the structure of the PDF (i.e not in its basic syntax.)

```
exception PDFSemanticError of string
```

Predicate on newline characters (carriage return and linefeed).

```
let is_newline = function
  | '\010' | '\013' → true
  | _ → false

let input_line i =
  let goteol = ref false
  and chars = ref []
  and finished = ref false in
    while ¬ !finished do
      match i.input_byte () with
      | x when x = Pdfio.no_more → set finished
      | x →
          let c = char_of_int x in
            if is_newline c then set goteol else
              if !goteol
                then (rewind i; set finished)
                else chars =| c
   done;
   implode (rev !chars)
```

Read back until a predicate is fulfilled.

```
let rec read_back_until p i =
  if (notpred p) (match read_char_back i with Some x → x | None →
 dpr "Q"; raise End_of_file)
    then read_back_until p i
```

Go back one line. In other words, find the second EOL character group seeking back in the file, and seek to the character after it. A blank line after a line with a single EOL character will be treated as being part of that EOL.

```
let backline i =
  read_back_until is_newline i;
  read_back_until (notpred is_newline) i;
  read_back_until is_newline i;
  nudge i
```

Read the major and minor version numbers from a PDF 1.x file. Fail if header invalid or major version number is not 1.

```
let rec read_header_inner pos i =
  try
    if pos > 1024 then (dpr "R"; raise End_of_file) else
      i.seek_in (posofi pos);
      match explode (input_line i) with
      | '%' :: 'P' :: 'D' :: 'F' :: '-' :: _ :: '.' :: minor →
          let minorchars = takewhile isdigit minor in
            if minorchars = []
              then
                raise (PDFReadError "Malformed PDF header")
              else
                begin
                  i.set_offset (posofi pos);
                  1, int_of_string (implode minorchars)
                end
      | _ →
          read_header_inner (pos + 1) i
  with
    End_of_file | Failure "int_of_string" →
      raise (PDFReadError "Could not read PDF header")

let read_header =
  read_header_inner 0
```

Find the EOF marker, and move position to its first character. We allow 1024 bytes from end-of-file for compatibility with Acrobat.

```
let find_eof i =
  let fail () = raise (PDFReadError "Could not find EOF marker")
  and pos = ref (possub (i.in_channel_length ()) (posofi 4)) in
    try
      let notfound = ref true
      and tries = ref 1024 in
        while !notfound do
```

```
        pos := pos_pred !pos;
        i.seek_in !pos;
        if !tries < 0 then fail () else decr tries;
        let l = input_line i in
          if l = "%%EOF" then clear notfound;
      done;
      i.seek_in !pos;
  with
    _ → fail ()
```

Lexemes.

```
type lexeme =
  | LexNull
  | LexBool of bool
  | LexInt of int
  | LexReal of float
  | LexString of string
  | LexName of string
  | LexLeftSquare
  | LexRightSquare
  | LexLeftDict
  | LexRightDict
  | LexStream of stream
  | LexEndStream
  | LexObj
  | LexEndObj
  | LexR
  | LexComment
  | StopLexing
  | LexNone
```

String of lexeme.

```
let string_of_lexeme = function
  | LexNull → "null"
  | LexBool b → Pdfwrite.string_of_pdf (Boolean b)
  | LexInt i → Pdfwrite.string_of_pdf (Integer i)
  | LexReal f → Pdfwrite.string_of_pdf (Real f)
  | LexString s → Pdfwrite.string_of_pdf (String s)
  | LexName s → s
  | LexLeftSquare → "["
  | LexRightSquare → "]"
  | LexLeftDict → "<<"
  | LexRightDict → ">>"
  | LexStream _ → "LexStream"
  | LexEndStream → "EndStream"
  | LexObj → "obj"
  | LexEndObj → "endobj"
  | LexR → "R"
  | LexComment → "Comment"
  | StopLexing → "StopLexing"
```

```
  | LexNone → "LexNone"
let print_lexeme l =
  Printf.printf "%s " (string_of_lexeme l)
```

Predicate on whitespace and delimiters.

```
let is_whitespace_or_delimiter c =
  is_whitespace c ∨ is_delimiter c
```

Return the list of characters between and including the current position and before the next character satisfying a given predicate, leaving the position at the character following the last one returned. Can raise EndOfInput. If *eoi* is true, end of input is considered a delimiter, and the characters up to it are returned if it is reached.

```
let getuntil eoi f i =
  let rec getuntil_inner r eoi f i =
    match i.input_byte () with
    | x when x = Pdfio.no_more →
        if eoi then rev r else (dpr "T"; raise End_of_file)
    | x →
        let chr = char_of_int x in
          if f chr
            then (rewind i; rev r)
            else getuntil_inner (chr :: r) eoi f i
  in
    getuntil_inner [] eoi f i
```

The same, but don't return anything.

```
let rec ignoreuntil eoi f i =
  match i.input_byte () with
  | x when x = Pdfio.no_more → if eoi then () else (dpr "V"; raise End_of_file)
  | x → if f (char_of_int x) then rewind i else ignoreuntil eoi f i
```

Ignore until the next whitespace

```
let ignoreuntilwhite =
  ignoreuntil true is_whitespace
```

Position on the next non-whitespace character.

```
let dropwhite i =
  ignoreuntil true (notpred is_whitespace) i
```

The same, but stop at array, dictionary endings etc.

```
let getuntil_white_or_delimiter =
  getuntil true is_whitespace_or_delimiter
```

9.1 Lexing

Each of the following functions lexes a particular object, leaving the channel position at the character after the end of the lexeme. Upon entry, the file position is on the first character of the potential lexeme.

Lex a bool.

```
let lex_bool i =
  match implode (getuntil_white_or_delimiter i) with
  | "true" → LexBool true
  | "false" → LexBool false
  | _ → LexNone
```

Lex an int or float. See PDF manual for details of policy.

```
let lex_number i =
  let number = implode (getuntil_white_or_delimiter i) in
    try
      match hd (Cgenlex.lex (input_of_bytestream (bytestream_of_string number))) with
      | Cgenlex.Int i → LexInt i
      | Cgenlex.Float f → LexReal f
      | _ → LexNone
    with
    | Failure "hd" → dpr "3F"; LexNone
    | PDFError _ (* can't cope with floats where number has leading point. *)
    | Failure "int_of_string" →
        dpr "3G";
        LexReal (float_of_string number)     (* float_of_string never fails. *)
```

Lex a name. We need to nudge past the slash and then add it manually since it is also a delimiter. Note that this correctly lexes the name consisting of just the slash, which is valid.

```
let rec substitute_hex prev = function
  | [] → rev prev
  | '#' :: a :: b :: more →
      let chr =
        char_of_int (int_of_string ("0x" ^ implode [a; b]))
          in
        substitute_hex (chr :: prev) more
  | chr :: more →
      substitute_hex (chr :: prev) more

let lex_name i =
  nudge i;
  let rawchars = "/" ^ (implode (getuntil_white_or_delimiter i)) in
    let substituted = implode (substitute_hex [] (explode rawchars)) in
      LexName substituted
```

Lex a comment. We throw away everything from here until a new line. In the case of a CRLF, only the CR is consumed, but the LF will be consumed before the next token is read anyway, so this is fine.

```
let lex_comment i =
  ignoreuntil false is_newline i;
  LexComment
```

Lex a string. A string is between parenthesis. Unbalanced parenthesis in the string must be escaped, but balanced ones need not be. We convert escaped characters to the characters themselves. A newline sequence following a backslash represents a newline. The string is returned without its enclosing parameters.

PDF strings can contain characters as a backslash followed by up to three octal characters. If there are fewer than three, the next character in the file cannot be a digit (The format is ambiguous as to whether this means an *octal* digit — we play safe and allow non-octal digits). This replaces these sequences of characters by a single character as used by OCaml in its native strings.

Beware malformed strings. For instance, Reader accepts ((
(ISA))

Build a character from a list of octal digits.

```
let mkchar l =
  try
    char_of_int (int_of_string ("0o" ^ implode l))
      with
    _ → raise (PDFError ("mkchar"))
```

Main function.

```
let lex_string i =
  try
    let str = Buffer.create 16 in
    let addchar = Buffer.add_char str
    and paren = ref 1
    and c = char_of_int (i.input_byte ()) in
    assert (c = '(');
    while !paren > 0 do
      let c = char_of_int (i.input_byte ()) in
      match c with
        | '(' →
            incr paren; addchar c;
        | ')' →
            decr paren; if !paren > 0 then addchar c;
        | '\\' →
            let c' = char_of_int (i.input_byte ()) in
            (match c' with
              | 'n' → addchar '\n'
              | 'r' → addchar '\r'
              | 't' → addchar '\t'
              | 'b' → addchar '\b'
              | 'f' → addchar '\012'
              | '\r' →
                  if char_of_int (i.input_byte ()) ≠ '\n' then
                    rewind i
              | '\n' → ()
              | '0'..'7' →
                (* Replace octal character sequences with the real character.
*)
                  let o2 = char_of_int (i.input_byte ()) in
                  (match o2 with
                    | '0'..'7' →
                        let o3 = char_of_int (i.input_byte ()) in
                        (match o3 with
```

```
                    | '0'..'7' →
                        addchar (mkchar [c'; o2; o3])
                    | _ →
                        rewind i;
                        addchar (mkchar [c'; o2]))
                | _ →
                    rewind i;
                    addchar (mkchar [c']))
            | _ → (* including '(', ')', '\\', and all the others *)
                addchar c' )
        | _ →
            addchar c
    done;
    LexString (Buffer.contents str)
  with
    | Failure "unopt" → raise (PDFReadError "lex_string")
```

Lex a hexadecimal string.

```
let lex_hexstring i =
  let mkchar a b =
    try
      char_of_int (int_of_string ("0x" ^ implode [a; b]))
    with
      _ → raise (PDFError ("Lexing Hexstring:  "))
  in
    try
      let _ = i.input_byte ()                    (* skip start marker *)
      and str = Buffer.create 16
      and finished = ref false in
        let addchar = Buffer.add_char str in
          let rec input_next_char () =
            let c = char_of_int (i.input_byte ()) in
              if is_whitespace c then input_next_char () else c
          in
            while ¬ !finished do
              let c = input_next_char () in
              let c' = input_next_char () in
                match c, c' with
                | '>', _ → rewind i; set finished
                | a, '>' → addchar (mkchar a '0')
                | a, b → addchar (mkchar a b)
            done;
            LexString (Buffer.contents str)
     with
       | Failure "unopt" → raise (PDFReadError "lex_hexstring")
```

Lex a keyword.

```
let lex_keyword i =
  match implode (getuntil_white_or_delimiter i) with
  | "obj" → LexObj
```

```
| "endobj" → LexEndObj
| "R" → LexR
| "null" → LexNull
| "endstream" → LexEndStream
| _ → LexNone
```

Lex a stream, given its length (previously extracted by parsing the stream dictionary). If *opt* is **true** the stream is actually read, if **false** a ToGet tuple is created. The channel is positioned on the first character of the stream keyword.

```
let lex_stream_data i l opt =
  try
    ignoreuntilwhite i;
    (* Skip either CRLF or LF. (See PDF specification for why) *)
    begin match char_of_int (i.input_byte ()) with
    | '\013' →
        begin match char_of_int (i.input_byte ()) with
        | '\010' → () (* It was CRLF *)
        | _ → rewind i (* No padding, happens to be CR *)
        end
    | '\010' → () (* Just LF *)
    | _ → rewind i (* No padding. *)
    end;
    if opt then
      let arr = mkstream l in
        if l > 0 then
          for k = 0 to l − 1 do
            sset arr k (i.input_byte ())
          done;
        LexStream (Got arr)
    else
      (* Advance past the stream data. *)
      let pos = i.pos_in ()
      and l = posofi l in
        i.seek_in (posadd pos l);
        LexStream (ToGet (i, postoi64 pos, postoi64 l))
  with
    _ → raise (PDFError "lex_stream_data")
```

Lex a stream. This involves *parsing* the stream dictionary to get the length. i is at the start of the stream data, suitable for input to *lex_stream_data*. We extract the dictionary by going through *previous_lexemes*, the reverse-order list of the lexemes already read.

```
let lex_stream i p previous_lexemes lexobj opt =
  let fail () = raise (PDFLexError "Failure lexing stream dict.") in
    let dictlexemes =
      [LexInt 0; LexInt 0; LexObj] @
      rev
        (takewhile (fun x → x ≠ LexObj) previous_lexemes) @
      [LexEndObj]
    in
```

```
match p dictlexemes with
  | _, Dictionary a →
    let rec findlength = function
      | Integer l → Some l
      | Indirect k → findlength (snd (p (lexobj k)))
      | _ → None
    in
      begin match lookup "/Length" a with
        | None → fail ()
        | Some v →
            match findlength v with
            | None → fail ()
            | Some l → lex_stream_data i l opt
      end
  | _ → fail ()
```

Find the next lexeme in the channel and return it. The latest-first lexeme list *previous_lexemes* contains all things thus-far lexed. *dictlevel* is a number representing the dictionary and/or array nesting level. If *endonstream* is true, lexing ends upon encountering a LexStream lexeme.

```
let lex_next dictlevel arraylevel endonstream i previous_lexemes p opt lexobj =
  try
    dropwhite i;
    (* To avoid problems with lexing at the end of the input, produce whites-
pace when input ends. *)
    let chr1 = char_of_int (i.input_byte ()) in
      rewind i;
      match chr1 with
      | '%' → lex_comment i
      | 't' | 'f' → lex_bool i
      | '/' → lex_name i
      | '0'..'9' | '+' | '-' | '.' → lex_number i
      | '[' → nudge i; incr arraylevel; LexLeftSquare
      | ']' → nudge i; decr arraylevel; LexRightSquare
      | '(' → lex_string i
      | '<' →
        let _ = char_of_int (i.input_byte ()) in
          let chr2 = char_of_int (i.input_byte ()) in
            rewind2 i;
            begin match chr2 with
            | '<' → nudge i; nudge i; incr dictlevel; LexLeftDict
            | _ → lex_hexstring i
            end
      | '>' →
        let _ = i.input_byte () in
          let chr2 = char_of_int (i.input_byte ()) in
            rewind2 i;
            begin match chr2 with
            | '>' → nudge i; nudge i; decr dictlevel; LexRightDict
            | _ → LexNone
```

```
              end
      | 'R' → nudge i; LexR
      | 's' →
          (* Disambiguate "startxref" and "stream" on the third character. *)
          let _ = i.input_byte () in
            let _ = i.input_byte () in
              let chr3 = char_of_int (i.input_byte ()) in
                rewind3 i;
                begin match chr3 with
                | 'a' → StopLexing                          (* startxref *)
                | _ →                                          (* stream *)
                    if endonstream
                      then StopLexing
                      else lex_stream i p previous_lexemes lexobj opt
                end
      | 'a'..'z' → lex_keyword i
      | 'I' → StopLexing      (* We've hit an ID marker in an inline image *)
      | _ → LexNone
  with
    _ → dpr "3D"; StopLexing
```

Lex just a dictionary, consuming only the tokens to the end of it. This is used in the PDFPages module to read dictionaries in graphics streams.

```
let lex_dictionary i =
  let rec lex_dictionary_getlexemes i lexemes dictlevel arraylevel =
    let lex_dictionary_next i dictlevel arraylevel =
      let dummyparse = fun _ → 0, Null
      and dummylexobj = fun _ → [] in
        lex_next dictlevel arraylevel false i [] dummyparse false dummylexobj
    in
      match lex_dictionary_next i dictlevel arraylevel with
      | LexRightDict when !dictlevel = 0 →
          rev (LexRightDict :: lexemes)
      | StopLexing →
          rev lexemes
      | LexNone →
          raise (PDFReadError "Could not read dictionary")
      | a →
          lex_dictionary_getlexemes i (a :: lexemes) dictlevel arraylevel
  in
    lex_dictionary_getlexemes i [] (ref 0) (ref 0)
```

Calculate a list of lexemes from input i, using parser p to lex streams. Can raise PDFReadError.

```
let lex_object_at oneonly i opt p lexobj =
  let dictlevel = ref 0
  and arraylevel = ref 0 in
    let rec lex_object_at i lexemes =
      let lexeme = lex_next dictlevel arraylevel false i lexemes p opt lexobj in
                match lexeme with
```

```
| LexEndObj → rev (lexeme :: lexemes)
| StopLexing → rev lexemes
| LexComment → lex_object_at i (lexeme :: lexemes)
| LexRightSquare | LexRightDict →
                     if oneonly ∧ !dictlevel = 0 ∧ !arraylevel = 0
      then
        (* 02/12/08 - We need to peek ahead to see if there's a stream
here. *)
        begin
          let pos = i.pos_in () in
            match lex_next dictlevel arraylevel false i (lexeme ::
lexemes) p opt lexobj with
            | LexStream s →
                begin match lex_next dictlevel arraylevel false i (LexStream s ::
lexeme :: lexemes) p opt lexobj with
                | LexEndStream →
                    begin match lex_next dictlevel arraylevel false i (LexEndStream ::
LexStream s :: lexeme :: lexemes) p opt lexobj with
                    | LexEndObj → rev (LexEndObj :: LexEndStream ::
LexStream s :: lexeme :: lexemes)
                    | _ →
                        Printf.eprintf "\nStopped at %Li\n" (i.pos_in ());
                        raise (PDFReadError "Could not read object
(oneonly - 2)")
                    end
                | _ →
                    Printf.eprintf "\nStopped at %Li\n" (i.pos_in ());
                    raise (PDFReadError "Could not read object
(oneonly)")
                end
            | _ → i.seek_in pos; rev (lexeme :: lexemes)
        end
      else lex_object_at i (lexeme :: lexemes)
| LexNone →
    Printf.eprintf "\nStopped at %Li\n" (i.pos_in ());
    raise (PDFReadError "Could not read object")
| LexInt i1 →
    (* Check for the case of "x y obj", which in the case of oneonly
should be returned as the one object. If i is followed by something other than
an integer and 'obj', we must rewind and just return the integer *)
    if oneonly ∧ !dictlevel = 0 ∧ !arraylevel = 0 then
      let pos = i.pos_in () in
        begin match lex_next dictlevel arraylevel false i lexemes p opt lexobj with
        | LexInt i2 →
            begin match lex_next dictlevel arraylevel false i lexemes p opt lexobj with
            | LexObj →
                lex_object_at i (LexObj :: LexInt i2 :: LexInt i1 ::
lexemes)
            | _ →
              i.seek_in pos;
```

```
                    rev (LexInt i1 :: lexemes)
                  end
              | _ →
                  i.seek_in pos;
                  rev (LexInt i1 :: lexemes)
              end
        else
          lex_object_at i (LexInt i1 :: lexemes)
    | a →
        (* If oneonly, then can return if not in an array or dictionary and if
this lexeme was an atom. *)
        (* FIXME: This wouldn't cope with just an indirect reference 0 1 R
- but this would be very odd. *)
        let isatom = function
          | LexBool _ | LexReal _ | LexString _ | LexName _ → true
          | _ → false
        in
          if oneonly ∧ isatom a ∧ !dictlevel = 0 ∧ !arraylevel = 0
            then rev (a :: lexemes)
            else lex_object_at i (a :: lexemes)
  in
    lex_object_at i []
```

Type of sanitized cross-reference entries. They are either plain offsets, or an object stream an index into it.

```
type xref =
  | XRefPlain of pos × int                         (* offset, generation. *)
  | XRefStream of int × int        (* object number of stream, index. *)
```

```
let string_of_xref = function
  | XRefPlain (p, i) → Printf.sprintf "XRefPlain (%Li, %i)" p i
  | XRefStream (o, i) → Printf.sprintf "XrefStream %i, index %i" o i
```

```
let xrefs_table_create () = Hashtbl.create 1001
```

IF-OCAML

```
let xrefs_table_add_if_not_present table k v =
  try ignore (Hashtbl.find table k)with
    Not_found → Hashtbl.add table k v
```

```
let xrefs_table_find table k =
  try Some (Hashtbl.find table k) with
    Not_found → None
```

ENDIF-OCAML

```
let xrefs_table_iter = Hashtbl.iter
```

p is the parser. Since this will be called from within functions it also calls, we must store and retrieve the current file position on entry and exit.

```
let rec lex_object i xrefs p opt n =
  let current_pos = i.pos_in () in
   let xref =
     match xrefs_table_find xrefs n with
     | Some x → x
     | None → raise (PDFReadError "Object not in xref table")
   in
     match xref with
     | XRefStream (objstm, index) →
         raise (Assert_failure ("lex_object", 0, 0)) (* lex object only used on XRefPlain entries *)
     | XRefPlain (o, _) →
         i.seek_in o;
         let result = lex_object_at false i opt p (lex_object i xrefs p opt) in
           i.seek_in current_pos;
           result
```

Given an object stream pdfobject and a list of object indexes to extract, return an *int* × *lexeme list list* representing those object number, lexeme pairs.

```
let lex_stream_object i xrefs parse opt obj indexes user_pw partial_pdf gen =
  let _, stmobj = parse (lex_object i xrefs parse opt obj) in
  match stmobj with
  | Stream {contents = Dictionary d, stream} →
      (* We assume that these are direct entries. *)
      let n =
        match lookup "/N" d with
        | Some (Integer n) → n
        | _ → raise (PDFSemanticError "missing/malformed /N")
      and first =
        match lookup "/First" d with
        | Some (Integer n) → n
        | _ → raise (PDFSemanticError "missing/malformed /First")
      in
        (* Decrypt if necessary *)
        let stmobj =
          Pdfcrypt.decrypt_single_stream user_pw partial_pdf obj gen stmobj
        in
        Pdfcodec.decode_pdfstream (Pdf.empty ()) stmobj;
        begin match stmobj with
        | Stream {contents = _, Got raw} →
          let i = input_of_bytestream raw in
            begin try
              (* Read index. *)
              let rawnums = ref [] in
                for x = 1 to n × 2 do
                  dropwhite i;
                  rawnums =|
                    match lex_number i with
                    | LexInt i → i
                    | k → raise (PDFSemanticError "objstm offset")
```

```
              done;
              rawnums := rev !rawnums;
              (* Read each object *)
              let pairs = pairs_of_list !rawnums
              and objects = ref []
              and index = ref 0 in
                iter
                  (fun (objnum, offset) →
                                                    if mem !index indexes then
                     begin
                                                             i.seek_in (posofi (offset + first));
                       let lexemes =
                         lex_object_at true i opt parse (lex_object i xrefs parse opt)
                       in
                                                                  objects =
| (objnum, lexemes);
                     end;
                     incr index)
                  pairs;
                rev !objects
          with
            End_of_file →
              raise (PDFSemanticError "unexpected objstream end")
          end
        | _ → raise (PDFSemanticError "couldn't decode objstream")
        end
  | _ → raise (PDFSemanticError "lex_stream_object:  not a stream")
```

9.2 Parsing

Parsing proceeds as a series of operations over lists of lexemes or parsed objects. Parsing ends when the list is a singleton and its element is an well-formed object.

```
type partial_parse_element =
  | Lexeme of lexeme
  | Parsed of pdfobject
```

Parse stage one — parse basic lexemes.

```
let parse_initial =
  map
    (function
     | Lexeme LexNull → Parsed Null
     | Lexeme (LexBool b) → Parsed (Boolean b)
     | Lexeme (LexInt i) → Parsed (Integer i)
     | Lexeme (LexReal r) → Parsed (Real r)
     | Lexeme (LexString s) → Parsed (String s)
     | Lexeme (LexName n) →
         Parsed (Name n)
```

```
    | l → l)

let print_partial = function
  | Lexeme l → print_lexeme l
  | Parsed p → Printf.printf "PARSED: %s\n" (Pdfwrite.string_of_pdf p)
```

Parse stage two. Parse indirect references. Also remove any dummy LexComment tokens.

```
let parse_R ts =
  let rec parse_R_inner r = function
    | [] → rev r
    | Parsed (Integer o) :: Parsed (Integer _) :: Lexeme LexR :: rest →
        parse_R_inner (Parsed (Indirect o) :: r) rest
    | Lexeme LexComment :: t → parse_R_inner r t
    | h :: t → parse_R_inner (h :: r) t
  in
    parse_R_inner [] ts
```

Parse stage three. Repeatedly parse dictionaries and arrays, bottom up. This should leave everything parsed other than the object itself.

```
let rec get_lexemes_to_symbol l s = function
  | [] → None
  | Lexeme s' :: t when s = s' → Some (rev l, t)
  | Lexeme (LexLeftDict | LexLeftSquare) :: _ → None
  | Parsed _ as h :: t → get_lexemes_to_symbol (h :: l) s t
  | Lexeme h :: t →
      raise (PDFParseError "get_lexemes_to_symbol:  Bad dict or array?")

let rec replace_dictarray prev = function
  | [] → rev prev
  | Lexeme LexLeftDict :: t →
      begin match get_lexemes_to_symbol [] LexRightDict t with
      | None → replace_dictarray (Lexeme LexLeftDict :: prev) t
      | Some (lexemes, rest) →
          if odd (length lexemes)
            then
              ( raise (PDFParseError "replace_dictarray 1"))
            else
              let pairs =
                map
                  (function
                   | Parsed (Name k), Parsed v → k, v
                   | _ → raise (PDFParseError "replace_dictarray 2"))
                  (pairs_of_list lexemes)
            in
              replace_dictarray (Parsed (Dictionary pairs) :: prev) rest
      end
  | Lexeme LexLeftSquare :: t →
      begin match get_lexemes_to_symbol [] LexRightSquare t with
      | None → replace_dictarray (Lexeme LexLeftSquare :: prev )t
      | Some (lexemes, rest) →
```

```
        let arry =
          map
            (function
             | Parsed x → x
             | _ → raise (PDFParseError "replace_dictarray 3"))
            lexemes
        in
          replace_dictarray (Parsed (Array arry) :: prev) rest
      end
  | h :: t → replace_dictarray (h :: prev) t
```

Debug printing of parsemes.

```
let print_parseme = function
  | Parsed p → flprint "PARSED:"; print_string (Pdfwrite.string_of_pdf p); flprint "\n"
  | Lexeme l → flprint "LEXEME:"; print_lexeme l; flprint "\n"
```

Call *replace_dictarray* repeatedly until no arrays or dictionaries to do, then extract the object. Possible correct forms: (1) Normal object (2) Stream object (3) Trailer dictionary. This can be non-terminating on bad input, so bail out after 5000 recursions.

```
let rec parse_reduce recs l =
  if recs = 5000 then raise (PDFReadError "Parse error") else
    let rec parse_finished = function
      | [] → true
      | Lexeme (LexLeftSquare | LexLeftDict) :: _ → false
      | _ :: t → parse_finished t
    in
      if parse_finished l then
        match l with
        | [Parsed (Integer o); Parsed (Integer g);
          Lexeme LexObj; Parsed obj; Lexeme LexEndObj] →
            o, obj
        | [Parsed (Integer o); Parsed (Integer g);
          Lexeme LexObj; Parsed obj; Lexeme (LexStream s);
          Lexeme LexEndStream; Lexeme LexEndObj] →
            o, Stream {contents = obj, s}
        | [Parsed d] →
            0, d
        | l →
            flprint "PARSEMES:\n";
            iter print_parseme l;
            flprint "END OF PARSEMES\n";
            raise (PDFReadError "Could not extract object")
      else
        parse_reduce (recs + 1) (replace_dictarray [] l)
```

Parse some lexemes

```
let parse lexemes =
  parse_reduce 0 (parse_R (parse_initial (map (fun x → Lexeme x) lexemes)))
```

```
let parse_objnum objnum' lexemes =
  (objnum', snd (parse lexemes))
```

Advance to the first thing after the current pointer which is not a comment.

```
let rec ignore_comments i =
  let pos = i.pos_in () in
    match i.input_char () with
    | Some '%' →
      (ignore (input_line i); ignore_comments i)
    | Some _ → i.seek_in pos
    | None → dpr "W"; raise End_of_file
```

9.3 Cross-reference tables

Read the cross-reference table. Supports the multiple sections created when a PDF file is incrementally modified.

```
type xref_line =
  | Invalid
  | Section of int × int                          (* Start, length. *)
  | Valid of pos × int                            (* byte offset, gen. *)
  | Free of pos × int                             (* free entry. *)
  | InObjectStream of int × int                   (* Stream number, index. *)
  | StreamFree of pos × int                       (* free entry in an object stream. *)
  | XRefNull                                      (* is the null object. *)
  | Finished                                      (* end of a table. *)
```

Read and parse a single line of a cross-reference table. We use a long-winded match pattern on the characters of cross-reference lines because a byte offset can exceed the range for Genlex.Int.

```
let rec read_xref_line i =
  let pos = i.pos_in () in
    let line = input_line i in
      if line = "xref" then read_xref_line i else
        let is09 x =
          x ≥ '0' ∧ x ≤ '9'
        in
        match explode line with
        | 't' :: 'r' :: 'a' :: 'i' :: 'l' :: 'e' :: 'r' :: more →
            (* Bad files may not put newlines after the trailer, so input_line
may have taken too much, preventing us from reading the trailer dictionary, so
we rewind. *)
            i.seek_in (posadd pos (posofi 7));
            Finished
        | a :: b :: c :: d :: e :: f :: g :: h :: i :: j :: ' ' :: k :: l :: m :: n :: o ::
```

```
' ' :: r
          when is09 a ∧ is09 b ∧ is09 c
            ∧ is09 d ∧ is09 e ∧ is09 f
            ∧ is09 g ∧ is09 h ∧ is09 i
            ∧ is09 j ∧ is09 k ∧ is09 l
            ∧ is09 m ∧ is09 n ∧ is09 o →
          let p, i =
            Int64.of_string (implode [a; b; c; d; e; f; g; h; i; j]),
            int_of_string (implode [k; l; m; n; o])
          in
            begin
              match r with
              | 'n' :: _ → Valid (posofi64 p, i)
              | 'f' :: _ → Free (posofi64 p, i)
              | _ → Invalid
            end
      | _ →
        (* Artworks produces bad PDF with lines like xref 1 5 *)
        match Cgenlex.lex (input_of_bytestream (bytestream_of_string line)) with
        | [Cgenlex.Ident "xref"; Cgenlex.Int s; Cgenlex.Int l]
        | [Cgenlex.Int s; Cgenlex.Int l] → Section (s, l)
        | _ → Invalid
```

Read the cross-reference table in i at the current position. Leaves i at the first character of the trailer dictionary.

```
let read_xref i =
  let fail () = raise (PDFReadError "Could not read x-ref table")
  and xrefs = ref [] in
    begin try
      let finished = ref false
      and objnumber = ref 1 in
        while ¬ !finished do
          match read_xref_line i with
          | Invalid → fail ()
          | Valid (offset, gen) →
              xrefs =| (!objnumber, XRefPlain (offset, gen));
              incr objnumber
          | Finished → set finished
          | Section (s, _) → objnumber := s
          | Free _ → incr objnumber
          | _ → () (* Xref stream types won't have been generated. *)
        done
      with
        End_of_file | (*IF-OCAML*)Sys_error _ | (*ENDIF-OCAML*)Failure "int_of_string"→
fail ()
    end;
    !xrefs
```

PDF 1.5 cross-reference stream support. i is the input. The tuple describes the lengths in bytes of each of the three fields.

```
let read_xref_line_stream i (w1, w2, w3) =
  assert (w1 ≥ 0 ∧ w2 ≥ 0 ∧ w3 ≥ 0);
  try
    let rec mknum mul = function
      | [] → 0L
      | h :: t → i64add (i64mul (i64ofi h) mul) (mknum (i64mul mul 256L) t)
    in
      let rec read_field bytes = function
        | 0 → mknum 1L bytes (* Lower order byte first. *)
        | n →
          match i.input_byte () with
          | x when x = Pdfio.no_more → raise (PDFError "")
          | b → read_field (b :: bytes) (n − 1)
      in
        let f1 = read_field [] w1 in
          let f2 = read_field [] w2 in
            let f3 = read_field [] w3 in
              match f1 with
              | 0L → StreamFree (posofi64 f2, i64toi f3)
              | 1L → Valid (posofi64 f2, i64toi f3)
              | 2L → InObjectStream (i64toi f2, i64toi f3)
              | n → XRefNull
  with
    _ → raise (PDFReadError "read_xref_line_stream")
```

The function to read a whole cross-reference stream, and return an *xreflist*. Leaves i at the first character of the stream dictionary, which containes the trailer dictionary entries.

```
let read_xref_stream i =
  let original_pos = i.pos_in ()
  and err = PDFReadError "Bad xref stream" in
    let rec lex_untilstream i ls =
      let lexobj = lex_object i (null_hash ()) parse false in
        match lex_next (ref 0) (ref 0) true i [] parse false lexobj with
        | StopLexing → rev ls
        | l → lex_untilstream i (l :: ls)
    in
      let stream, obj, gen =
        match
          let lexobj = lex_object i (null_hash ()) parse true in
            let dictlex = lex_untilstream i [] in
              let obj =
                match hd dictlex with
                | LexInt i → i
                | _ → raise Not_found
              and gen =
                match (hd (tl dictlex)) with
                | LexInt i → i
                | _ → raise Not_found
              in
```

```
              match lex_stream i parse (rev dictlex) lexobj true with
              | LexNone → raise err
              | stream →
                  snd (parse (dictlex @ [stream] @ [LexEndStream; LexEndObj])),
                  obj,
                  gen
      with
      | Stream _ as stream, obj, gen → stream, obj, gen
      | _ → raise err
    in
      Pdfcodec.decode_pdfstream (Pdf.empty ()) stream;
      let ws =
        match lookup_direct (Pdf.empty ()) "/W" stream with
        | Some (Array [Integer w1; Integer w2; Integer w3]) → w1, w2, w3
        | _ → raise err
      and i' =
        match stream with
        | Stream {contents = _, Got s} → input_of_bytestream s
        | _ → raise err
      and xrefs = ref [] in
        begin try
          while true do
            xrefs =| read_xref_line_stream i' ws
          done
        with
          _ → dpr "3H"; ()
        end;
        xrefs := rev !xrefs;
        let starts_and_lens =
          match lookup_direct (Pdf.empty ()) "/Index" stream with
          | Some (Array elts) →
              if odd (length elts) then raise (PDFReadError "Bad /Index");
              map
                (function
                  | (Pdf.Integer s, Pdf.Integer l) → s, l
                  | _ → raise (PDFReadError "Bad /Index entry"))
                (pairs_of_list elts)
          | Some _ → raise (PDFSemanticError "Unknown /Index")
          | None →
              let size =
                match lookup_direct (Pdf.empty ()) "/Size" stream with
                | Some (Integer s) → s
                | _ →
                    raise (PDFSemanticError "Missing /Size in xref dict")
              in
                [0, size]
        in
          let xrefs' = ref [] in
          iter
            (fun (start, len) →
```

```
          let these_xrefs =
            try take !xrefs len with
              _ → raise (PDFReadError "Bad xref stream\n")
          in
            xrefs := drop !xrefs len;
            let objnumber = ref start in
              iter
                (function
                 | Valid (offset, gen) →
                     xrefs' =| (!objnumber, XRefPlain (offset, gen));
                     incr objnumber
                 | InObjectStream (stream, index) →
                     xrefs' =| (!objnumber, XRefStream (stream, index));
                     incr objnumber
                 | _ → incr objnumber)
                these_xrefs)
        starts_and_lens;
      i.seek_in original_pos;
      rev !xrefs'
```

A suitable function for the Pdf module to use to lex and parse an object. Assumes i has been set to the correct position. n is the object number.

```
let get_object i xrefs n =
  let lexemes = lex_object i xrefs parse false n in
    snd (parse_objnum n lexemes)
```

9.4 Main functions

Read a PDF from a channel. If *opt*, streams are read immediately into memory.

```
let read_pdf user_pw opt i =
  let xrefs = xrefs_table_create () in
  let major, minor = read_header i
  and objects, root, trailerdict =
    let addref (n, x) = xrefs_table_add_if_not_present xrefs n x
    and got_all_xref_sections = ref false
    and trailerdict = ref []
    and xref = ref 0L
    and first = ref true in
      (* This function builds a partial pdf of the plain objects whose references
have currently been seen. *)
      let mkpartial trailerdict =
        let objpairs = ref [] in
          (* 1. Build object number, offset pairs *)
          xrefs_table_iter
            (fun n x →
               match x with
               | XRefPlain (offset, gen) → objpairs =| (n, (ref ToParse, gen))
               | _ → ())
```

```
        xrefs;
        (* 2. Build the object map *)
        let objects =
          Pdf.objects_of_list (Some (get_object i xrefs)) !objpairs
        in
          (* 3. Build the Pdf putting the trailerdict in *)
          {(Pdf.empty ()) with
             Pdf.objects = objects;
             Pdf.trailerdict = trailerdict}
    in
    (* Move to the first xref section. *)
    find_eof i;
    backline i;
    (* Drop any initial contents which is not a digit - may occur if there is
legitimate whitespace of if the PDF is malformed such that it has the startxref
keyword and the byte offset on the same line. *)
    ignoreuntil false isdigit i;
    begin match takewhile isdigit (getuntil_white_or_delimiter i) with
    | [] → raise (PDFReadError "Could not find xref pointer")
    | xrefchars → xref := Int64.of_string (implode xrefchars);
    end;
    while ¬ !got_all_xref_sections do
      i.seek_in (posofi64 !xref);
      (* Distinguish between xref table and xref stream. *)
      dropwhite i;
      let f_read_xref =
        if peek_char i = Some 'x'
          then read_xref
          else read_xref_stream
      in
        (* Read cross-reference table *)
        iter addref (f_read_xref i);
        (* It is now assumed that i is at the start of the trailer dictionary. *)
        let trailerdict_current =
                      let lexemes =
            lex_object_at true i opt parse (lex_object i xrefs parse opt)
          in
                          match parse lexemes with
            | (_, Dictionary d)
            | (_, Stream {contents = Dictionary d, _}) → d
            | _ → raise (PDFReadError "Malformed trailer")
        in
          begin
                        if !first then
              begin
                trailerdict := mergedict trailerdict_current !trailerdict;
                clear first
              end;
            (* Do we have a /XRefStm to follow? *)
            begin match lookup "/XRefStm" trailerdict_current with
```

```
          | Some (Integer n) →
              i.seek_in (posofi n);
              iter addref (read_xref_stream i);
          | _ → ()
          end;
          (* Is there another to do? *)
          match lookup "/Prev" trailerdict_current with
          | None → set got_all_xref_sections
          | Some (Integer n) → xref := i64ofi n
          | _ → raise (PDFReadError "Malformed trailer")
        end;
done;
let root =
  match lookup "/Root" !trailerdict with
  | Some (Indirect i) → i
  | None → raise (PDFReadError "No /Root entry")
  | _ → raise (PDFReadError "Malformed /Root entry")
in
  (* Print out the contents of the xref tables *)
          let getgen n =
    match xrefs_table_find xrefs n with
    | Some (XRefPlain (_, g)) → g
    | Some (XRefStream _) → 0
    | None → raise Not_found
  in
  let objects_nonstream =
    let objnumbers = ref [] in
      xrefs_table_iter
        (fun n x →
           match x with
           | XRefPlain (offset, gen) → objnumbers =| n
           | _ → ())
        xrefs;
        map
          (if opt then
             fun o →
               let num, parsed =
                 parse (lex_object i xrefs parse opt o)
               in
                 num, (ref (Pdf.Parsed parsed), getgen o)
           else
             fun o → o, (ref Pdf.ToParse, getgen o))
          !objnumbers
   and objects_stream =
     let streamones =
       map
         (function
            | (n, XRefStream (s, i)) → (n, s, i)
            | _ → raise (Assert_failure ("objects_stream", 0, 0)))
         (keep
```

```
                    (function (n, XRefStream _) → true | _ → false)
                    (list_of_hashtbl xrefs))
              in
                          let cmp_objs (_, s, _) (_, s′, _) = compare s s′ in
                  let sorted = List.sort cmp_objs streamones in
                    let collated = collate cmp_objs sorted in
                      let inputs_to_lex_stream_object =
                        map
                          (fun l →
                            match hd l with (_, s, _) →
                              s, map (fun (_, _, i) → i) l)
                          collated
                      in
                        let outputs_from_lex_stream_object =
                          map
                            (function (s, is) →
                                lex_stream_object
                                  i xrefs parse opt s is user_pw
                                  (mkpartial (Pdf.Dictionary !trailerdict)) (getgen s))
                            inputs_to_lex_stream_object
                        in
                          let object_lexemes_and_numbers =
                            flatten outputs_from_lex_stream_object
                          in
                            map
                              (fun (objnum, lexemes) →
                                  objnum,
                                  (* Generation number of object in stream is
always zero. *)
                                  (ref (Pdf.Parsed (snd (parse_objnum objnum lexemes))), 0))
                              object_lexemes_and_numbers
             in
                        objects_stream @ objects_nonstream, root, trailerdict
        in
```

Fix Size entry and remove Prev and XRefStm

```
          let trailerdict′ =
            Dictionary
              (add "/Size" (Integer (length objects))
                (remove "/Prev" (remove "/XRefStm" !trailerdict)))
          in
            {major = major;
             minor = minor;
             objects = Pdf.objects_of_list (Some (get_object i xrefs)) objects;
             root = root;
             trailerdict = trailerdict′}

let default_upw = function
  | None → ""
  | Some p → p
```

▷ Read a PDF into memory, including its streams.

```
let pdf_of_channel upw ch =
  let upw = default_upw upw in
    read_pdf upw true (input_of_channel ch)
```

▷ Same, but delay reading of streams.

```
let pdf_of_channel_lazy upw ch =
  let upw = default_upw upw in
  read_pdf upw false (input_of_channel ch)
```

▷ Similarly for inputs.

```
let pdf_of_input upw i =
  let upw = default_upw upw in
  read_pdf upw true i
```

▷ And lazy on inputs.

```
let pdf_of_input_lazy upw i =
  let upw = default_upw upw in
  read_pdf upw false i
```

▷ Read a whole PDF file into memory. Closes file.

```
let pdf_of_file upw f =
  try
    let fh = open_in_bin f in
      let pdf = pdf_of_channel upw fh in
        close_in fh;
        pdf
  with
    | (PDFError _ | PDFSemanticError _ | PDFReadError _) as e → raise e
    (*IF-OCAML*)| Sys_error str → raise (PDFError str) (*ENDIF-OCAML*)

let what_encryption pdf =
  if Pdfcrypt.is_encrypted pdf then
    let crypt, _, _, _, _ = Pdfcrypt.get_encryption_values pdf in
      match crypt with
      | Pdfcrypt.ARC4 (40, _) → Some (Pdfwrite.PDF40bit)
      | Pdfcrypt.ARC4 (128, _) → Some (Pdfwrite.PDF128bit)
      | Pdfcrypt.AESV2 →
          let metadata =
            match Pdf.lookup_direct pdf "/Encrypt" pdf.Pdf.trailerdict with
            | Some encrypt_dict →
                begin match Pdf.lookup_direct pdf "/EncryptMetadata" encrypt_dict with
                | Some (Pdf.Boolean false) → false
                | _ → true
                end
            | _ → raise (Assert_failure ("what_encryption", 0, 0))
          in
            Some (Pdfwrite.AES128bit metadata)
      | _ → None
  else
    None
```

```
let permissions pdf =
  if Pdfcrypt.is_encrypted pdf then
    let _, _, _, p, _ = Pdfcrypt.get_encryption_values pdf in
      Pdfcrypt.banlist_of_p p
  else
    []
```

10 Module PDFPages
High level PDF operations

```
open Utility
open Pdfio
open Pdf
open Pdfread
```

10.1 Types

↪ *PDF Manual, Table 4.1*

Graphics operators. Exported in interface.

```
type operator =
  | Op_w of float                                  (* Set line width *)
  | Op_J of int                                    (* Set line cap *)
  | Op_j of int                                    (* Set line join *)
  | Op_M of float                                  (* Set mitre limit *)
  | Op_d of float list × float                     (* Set dash pattern (dash, phase) *)
  | Op_ri of string                                (* Set rendering intent. *)
  | Op_i of int                                    (* Set flatness. *)
  | Op_gs of string                                (* Set graphics state from dictionary *)
  | Op_q                                           (* Save graphics state to stack *)
  | Op_Q                                           (* Restore graphics state from stack *)
  | Op_cm of Transform.transform_matrix            (* Modify CTM by concatenation *)
  | Op_m of float × float                          (* Begin a new subpath *)
  | Op_l of float × float                          (* Append a straight segment *)
  | Op_c of float × float × float × float × float × float   (* Cubic bezier *)
  | Op_v of float × float × float × float          (* Similar. *)
  | Op_y of float × float × float × float          (* Similar. *)
  | Op_h                                           (* Close subpath *)
  | Op_re of float × float × float × float         (* Append rectangle *)
  | Op_S                                           (* Stroke a path *)
  | Op_s                                           (* Close and stroke path *)
  | Op_f                                           (* Fill path, non-zero *)
  | Op_F                                           (* Same. *)
  | Op_f'                                          (* f* operator. Fill path, even-odd. *)
  | Op_B                                           (* Fill and stroke path, non-zero *)
  | Op_B'                                          (* B* operator. Fill and stroke path, even-odd *)
```

Margin notes: *General graphics state* (beside Op_w); *Special graphics state* (beside Op_q); *Path construction* (beside Op_m); *Path painting* (beside Op_S).

```
| Op_b                                      (* Close fill and stroke, non-zero *)
| Op_b'                     (* b* operator. Close fill and stroke, even-odd *)
| Op_n                                                        (* Path no-op *)
| Op_W                                       (* Clipping path, even-odd *)   Clipping paths
| Op_W'                                      (* Clipping path, non-zero *)
| Op_BT                                          (* Begin a text object *)   Text objects
| Op_ET                                            (* End a text object *)
| Op_Tc of float                             (* Set character spacing *)   Text state
| Op_Tw of float                                  (* Set word spacing *)
| Op_Tz of float                            (* Set horizontal scaling *)
| Op_TL of float                                       (* Set leading *)
| Op_Tf of string × float                            (* Set font size *)
| Op_Tr of int                             (* Set text rendering mode *)
| Op_Ts of float                                     (* Set text rise *)
| Op_Td of float × float                         (* Move to next line *)   Text positioning
| Op_TD of float × float                  (* Ditto, but set leading *)
| Op_Tm of Transform.transform_matrix  (* Set text and line matrices *)
| Op_T'                    (* T* operator. Move text to the next line *)
| Op_Tj of string                                (* Show text string *)   Text showing
| Op_TJ of pdfobject                       (* Show many text strings *)
| Op_' of string                     (* Move to next line and show text *)
| Op_" of float × float × string          (* Ditto, extra parameters *)
| Op_d0 of float × float                   (* Set glpyh width info *)   Type 3 fonts
| Op_d1 of float × float × float × float × float × float     (* Similar *)
| Op_CS of string                               (* Set colour space. *)   Colour
| Op_cs of string                 (* Same for nonstroking operations *)
| Op_SC of float list          (* Set colour in current colour space. *)
| Op_sc of float list             (* Same for nonstroking operations *)
| Op_SCN of float list         (* Set colour in current colour space. *)
| Op_scn of float list            (* Same for nonstroking operations *)
| Op_SCNName of string × float list              (* A named Op_SCN *)
| Op_scnName of string × float list              (* Same for Op_scn *)
| Op_G of float                                          (* set gray *)
| Op_g of float                             (* set gray nonstroking *)
| Op_RG of float × float × float            (* Set stroking colour *)
| Op_rg of float × float × float            (* Set painting colour *)
| Op_K of float × float × float × float       (* Set CMYK stroking *)
| Op_k of float × float × float × float    (* Set CMYK nonstroking *)
| Op_sh of string                               (* Shading pattern *)   Shading patterns
| InlineImage of (pdfobject × bytestream) (* Inline image dictionary/data *)
| Op_Do of string                          (* Introduce an XObject *)   XObjects
| Op_MP of string                        (* Marked content point *)   Marked Content
| Op_DP of string × pdfobject           (* same with property list *)
| Op_BMC of string               (* begin marked content sequence *)
| Op_BDC of string × pdfobject          (* same with property list *)
| Op_EMC                           (* end of marked content sequence *)
| Op_BX                               (* Start compatibility mode *)   Compatibility
| Op_EX                                 (* End compatibility mode *)   markers
| Op_Unknown of string        (* Unknown operand / operator sequence *)
```

```
type lexeme =
  | Op of string
  | Obj of Pdfread.lexeme
  | PdfObj of pdfobject
  | LexInlineImage of (pdfobject × bytestream)
  | LexComment
```

10.2 Lexing

```
let lexemes_of_op = function
  | Op_w w → [Obj (LexReal w); Op "w"]
  | Op_J j → [Obj (LexInt j); Op "J"]
  | Op_j j → [Obj (LexInt j); Op "j"]
  | Op_M m → [Obj (LexReal m); Op "M"]
  | Op_d (fl, f) →
      [Obj LexLeftSquare] @
      (map (fun x → Obj (LexReal x)) fl) @
      [Obj LexRightSquare; Obj (LexReal f); Op "d"]
  | Op_ri s → [Obj (LexName s); Op "ri"]
  | Op_i i → [Obj (LexInt i); Op "i"]
  | Op_gs s → [Obj (LexName s); Op "gs"]
  | Op_q → [Op "q"]
  | Op_Q → [Op "Q"]
  | Op_cm t →
      [Obj (LexReal t.Transform.a); Obj (LexReal t.Transform.b);
       Obj (LexReal t.Transform.c); Obj (LexReal t.Transform.d);
       Obj (LexReal t.Transform.e); Obj (LexReal t.Transform.f);
       Op "cm"]
  | Op_m (a, b) →
      [Obj (LexReal a); Obj (LexReal b); Op "m"]
  | Op_l (a, b) →
      [Obj (LexReal a); Obj (LexReal b); Op "l"]
  | Op_c (a, b, c, d, e, f) →
      [Obj (LexReal a); Obj (LexReal b);
       Obj (LexReal c); Obj (LexReal d);
       Obj (LexReal e); Obj (LexReal f); Op "c"]
  | Op_v (a, b, c, d) →
      [Obj (LexReal a); Obj (LexReal b);
       Obj (LexReal c); Obj (LexReal d); Op "v"]
  | Op_y (a, b, c, d) →
      [Obj (LexReal a); Obj (LexReal b);
       Obj (LexReal c); Obj (LexReal d); Op "y"]
  | Op_h → [Op "h"]
  | Op_re (a, b, c, d) →
      [Obj (LexReal a); Obj (LexReal b);
       Obj (LexReal c); Obj (LexReal d); Op "re"]
  | Op_S → [Op "S"]
  | Op_s → [Op "s"]
```

```
| Op_f → [Op "f"]
| Op_F → [Op "F"]
| Op_f' → [Op "f*"]
| Op_B → [Op "B"]
| Op_B' → [Op "B*"]
| Op_b → [Op "b"]
| Op_b' → [Op "b*"]
| Op_n → [Op "n"]
| Op_W → [Op "W"]
| Op_W' → [Op "W*"]
| Op_BT → [Op "BT"]
| Op_ET → [Op "ET"]
| Op_Tc c → [Obj (LexReal c); Op "Tc"]
| Op_Tw w → [Obj (LexReal w); Op "Tw"]
| Op_Tz z → [Obj (LexReal z); Op "Tz"]
| Op_TL l → [Obj (LexReal l); Op "TL"]
| Op_Tf (f, s) → [Obj (LexName f); Obj (LexReal s); Op "Tf"]
| Op_Tr i → [Obj (LexInt i); Op "Tr"]
| Op_Ts f → [Obj (LexReal f); Op "Ts"]
| Op_Td (f, f') → [Obj (LexReal f); Obj (LexReal f'); Op "Td"]
| Op_TD (f, f') → [Obj (LexReal f); Obj (LexReal f'); Op "TD"]
| Op_Tm t →
    [Obj (LexReal t.Transform.a); Obj (LexReal t.Transform.b);
     Obj (LexReal t.Transform.c); Obj (LexReal t.Transform.d);
     Obj (LexReal t.Transform.e); Obj (LexReal t.Transform.f);
     Op "Tm"]
| Op_T' → [Op "T*"]
| Op_Tj s → [Obj (LexString s); Op "Tj"]
| Op_TJ pdfobject → [PdfObj pdfobject; Op "TJ"]
| Op_' s → [Obj (LexString s); Op "'"]
| Op_'' (f, f', s) →
    [Obj (LexReal f); Obj (LexReal f'); Obj (LexString s); Op "\""]
| Op_d0 (f, f') → [Obj (LexReal f); Obj (LexReal f'); Op "d0"]
| Op_d1 (a, b, c, d, e, f) →
    [Obj (LexReal a); Obj (LexReal b);
     Obj (LexReal c); Obj (LexReal d);
     Obj (LexReal e); Obj (LexReal f); Op "d1"]
| Op_CS s → [Obj (LexName s); Op "CS"]
| Op_cs s → [Obj (LexName s); Op "cs"]
| Op_SC fs → map (fun f → Obj (LexReal f)) fs @ [Op "SC"]
| Op_sc fs → map (fun f → Obj (LexReal f)) fs @ [Op "sc"]
| Op_SCN fs → map (fun f → Obj (LexReal f)) fs @ [Op "SCN"]
| Op_scn fs → map (fun f → Obj (LexReal f)) fs @ [Op "scn"]
| Op_SCNName (s, fs) →
    map (fun x → Obj (LexReal x)) fs @ [Obj (LexName s); Op "SCN"]
| Op_scnName (s, fs) →
    map (fun x → Obj (LexReal x)) fs @ [Obj (LexName s); Op "scn"]
| Op_G f → [Obj (LexReal f); Op "G"]
| Op_g f → [Obj (LexReal f); Op "g"]
| Op_RG (r, g, b) →
```

```
        [Obj (LexReal r); Obj (LexReal g); Obj (LexReal b); Op "RG"]
  | Op_rg (r, g, b) →
        [Obj (LexReal r); Obj (LexReal g); Obj (LexReal b); Op "rg"]
  | Op_K (c, m, y, k) →
        [Obj (LexReal c); Obj (LexReal m); Obj (LexReal y); Obj (LexReal k); Op "K"]
  | Op_k (c, m, y, k) →
        [Obj (LexReal c); Obj (LexReal m); Obj (LexReal y); Obj (LexReal k); Op "k"]
  | Op_sh s → [Obj (LexName s); Op "sh"]
  | InlineImage (dict, data) → [LexInlineImage (dict, data)]
  | Op_Do s → [Obj (LexName s); Op "Do"]
  | Op_MP s → [Obj (LexName s); Op "MP"]
  | Op_DP (s, obj) → [Obj (LexName s); PdfObj obj; Op "DP"]
  | Op_BMC s → [Obj (LexName s); Op "BMC"]
  | Op_BDC (s, obj) → [Obj (LexName s); PdfObj obj; Op "BDC"]
  | Op_EMC → [Op "EMC"]
  | Op_BX → [Op "BX"]
  | Op_EX → [Op "EX"]
  | Op_Unknown _ → []
```

Find a string representing some lexemes

```
let string_of_lexemes lexemes =
  let string_of_lexeme = function
    | LexComment → ""
    | Obj o → Pdfread.string_of_lexeme o
    | Op op → op
    | PdfObj obj → Pdfwrite.string_of_pdf obj
    | LexInlineImage (dict, data) →
        let dict_string = Pdfwrite.string_of_pdf dict in
          let dict_string' =
            (* Remove the dictionary markers. *)
            implode (rev (drop' 2 (rev (drop' 2 (explode dict_string)))))
          and data_string =
            string_of_bytestream data
          and space =
            let filters =
              match lookup_direct_orelse (Pdf.empty ()) "/F" "/Filter" dict with
              | Some (Array filters) → filters
              | Some (Name f) → [Name f]
              | _ → []
            in
              if
                (mem (Name "/ASCIIHexDecode") filters
                ∨ mem (Name "/ASCII85Decode") filters
                ∨ mem (Name "/AHx") filters
                ∨ mem (Name "/A85") filters)
              then ""
              else " "
          in
          "BI\n" ^ dict_string' ^ "ID" ^ space ^ data_string ^ "\nEI\n"
  in
```

```
    let strings = map string_of_lexeme lexemes in
      fold_left ( ^ ) "" (interleave " " strings)
```

▷ Make a string of an operation, for debug purposes only.

```
let string_of_op = function
  | Op_Unknown s → "UNKNOWN: " ^ s
  | op → string_of_lexemes (lexemes_of_op op)

let string_of_ops ops =
  fold_left ( ^ ) "" (interleave " " (map string_of_op ops))
```

Lex a name.

```
let lex_name i =
  nudge i;
  Some (Obj (LexName (implode ('/' :: getuntil_white_or_delimiter i))))
```

Lex a number

```
let lex_number i =
  match Pdfread.lex_number i with
  | LexReal r → Some (Obj (LexReal r))
  | LexInt i → Some (Obj (LexInt i))
  | _ → None
```

Lex and parse a dictionary to a Pdf.*pdfobject*. This constitutes a single lexeme in terms of this module.

```
let get_dictionary i =
  Some (PdfObj (snd (Pdfread.parse (Pdfread.lex_dictionary i))))
```

This is raised when something which is a legitimate part of the PDF standard but which we don't understand is found. For now, this is just inline images. Eventually, it will be nothing.

```
exception Couldn'tHandleContent
```

Given a colourspace and the number of bits per component, give the number of bits per pixel in the stored image data.

```
let rec components pdf resources t =
  match t with
  | Name ("/CalGray" | "/DeviceGray" | "/G") → 1
  | Name ("/CalRGB" | "/DeviceRGB" | "/RGB") → 3
  | Name ("/CalCMYK" | "/DeviceCMYK" | "/CMYK") → 4
  | Name "/Pattern" →
      raise (PDFSemanticError "Can't use /Pattern here")
  | Name space →
      begin match lookup_direct pdf "/ColorSpace" resources with
      | Some csdict →
          begin match lookup_direct pdf space csdict with
          | Some space' → components pdf resources space'
          | None → raise (PDFSemanticError "ColorSpace not found")
          end
      | None → raise (PDFSemanticError "ColorSpace dict not found")
      end
```

```
  | Array [Name "/ICCBased"; iccstream] →
      begin match lookup_direct pdf "/N" iccstream with
      | Some (Integer n) → n
      | _ → raise (PDFSemanticError "Bad iccstream")
      end
  | Array (Name "/DeviceN"::_ :: alternate :: _) →
      components pdf resources (direct pdf alternate)
  | Array [Name "/Separation"; _; _; _]
  | Array (Name ("/Indexed" | "/I") :: _ :: _) → 1
  | Array [Name "/CalRGB"; _] → 3
  | Array [Name "/CalCMYK"; _] → 4
  | Array [Name "/CalGray"; _] → 1
  | Array [Name "/Pattern"; alternate] →
      components pdf resources (direct pdf alternate)
  | cs →
     Printf.eprintf "%s\n" (Pdfwrite.string_of_pdf cs);
     raise (PDFSemanticError "Unknown colourspace")
```

Lex an inline image. We read the dictionary, and then the stream.

```
let lex_inline_image pdf resources i =
    try
  let fail () =
    raise Couldn'tHandleContent
  and dict =
    let lexemes = Pdfread.lex_dictionary i in
      snd (Pdfread.parse ([Pdfread.LexLeftDict] @ lexemes @ [Pdfread.LexRightDict]))
  in
    (* Read ID token *)
    dropwhite i;
    let c = char_of_int (i.input_byte ()) in
      let c' = char_of_int (i.input_byte ()) in
        match c, c' with
        | 'I', 'D' →
          (* Skip a byte if not ASCII85 / ASCIIHex as one of the filters. *)
          let toskip =
            let filters =
              match lookup_direct_orelse pdf "/F" "/Filter" dict with
              | Some (Array filters) → filters
              | Some (Name f) → [Name f]
              | _ → []
            in
              ¬
                (mem (Name "/ASCIIHexDecode") filters
                ∨ mem (Name "/ASCII85Decode") filters
                ∨ mem (Name "/AHx") filters
                ∨ mem (Name "/A85") filters)
          in
            if toskip then ignore (i.input_byte ());
            let bytes =
              let bpc =
```

```
        match lookup_direct_orelse pdf "/BPC" "/BitsPerComponent" dict with
        | Some (Integer bpc) → bpc
        | _ → fail ()
      in
      let cspace =
        match lookup_direct_orelse pdf "/CS" "/ColorSpace" dict with
        | Some (Name ("/DeviceGray" | "/DeviceRGB" | "/DeviceCMYK") as n) →

            n
        | Some (Name ("/G" | "/RGB" | "/CMYK") as n) → n
        | Some ((Array _) as n) → n
        | Some (Name cspace) →
            begin match lookup_direct pdf "/ColorSpace" resources with
            | Some (Dictionary _ as d) →
                begin match lookup_direct pdf cspace d with
                | Some c → c
                | _ → fail ()
                end
            | _ → fail ()
            end
        | None →
            (* Could it be an image mask? *)
            begin match lookup_direct_orelse pdf "/IM" "/ImageMask" dict with
            | Some (Pdf.Boolean true) → Name "/DeviceGray"
            | _ → fail ()
            end
        | _ → fail ()
      and width =
        match lookup_direct_orelse pdf "/W" "/Width" dict with
        | Some (Integer w) → w
        | _ → fail ()
      and height =
        match lookup_direct_orelse pdf "/H" "/Height" dict with
        | Some (Integer h) → h
        | _ → fail ()
      in
        let bitwidth =
          components pdf resources cspace × bpc × width
        in
          let bytewidth =
            if bitwidth mod 8 = 0 then bitwidth / 8 else bitwidth / 8 + 1
          in
            bytewidth × height
    in
      let data =
        match lookup_direct_orelse (Pdf.empty ()) "/F" "/Filter" dict with
        | None | Some (Array []) →
            begin try let data = mkstream bytes in
              if bytes > 0 then
                for x = 0 to stream_size data − 1 do
```

```
                    sset data x (i.input_byte ());
                  done;
                data
              with
              | e → print_string (Printexc.to_string e); raise e
              end
          | Some _ →
              try
                match Pdfcodec.decode_from_input i dict with
                | None → raise Couldn'tHandleContent
                | Some data → data
              with
                | Pdfcodec.DecodeNotSupported →
                    raise Couldn'tHandleContent
                | Pdfcodec.Couldn'tDecodeStream r →
                    raise (PDFError ("Inline image, bad data: " ^ r))
                | e → raise e
        in
          (* Read EI token *)
          dropwhite i;
          let c = char_of_int (i.input_byte ()) in
            let c' = char_of_int (i.input_byte ()) in
              begin match c, c' with
              | 'E', 'I' →
                  (* Remove filter, predictor. *)
                  let dict' =
                    fold_left
                      remove_dict_entry
                      dict
                      ["/Filter"; "/F"; "/DecodeParms"; "/DP"]
                  in
                                              dict', data
              | _ →
                  fail ()
              end
    | _ → fail ()
  with
    _ → raise Couldn'tHandleContent
```

Lex a keyword.

```
let lex_keyword pdf resources i =
  match implode (getuntil_white_or_delimiter i) with
  | "true" → Some (Obj (LexBool true))
  | "false" → Some (Obj (LexBool false))
  | "BI" → Some (LexInlineImage (lex_inline_image pdf resources i))
  | "ID" | "EI" → None         (* lex_inline_image should consume these *)
  | ( "w" | "J" | "j" | "M" | "d" | "ri" | "i" | "gs"
    | "q" | "Q" | "cm" | "m" | "l" | "c" | "v" | "y"
    | "h" | "re" | "S" | "s" | "f" | "F" | "f*" | "B"
    | "B*" | "b" | "b*" | "n" | "W" | "W*" | "BT" | "ET"
```

```
    | "Tc" | "Tw" | "Tz" | "TL" | "Tf" | "Tr" | "Ts"
    | "Td" | "TD" | "Tm" | "T*" | "Tj" | "TJ" | "\'"
    | "\"" | "d0" | "d1" | "CS" | "cs" | "SC" | "SCN"
    | "sc" | "scn" | "G" | "g" | "RG" | "rg" | "K" | "k"
    | "sh" | "Do" | "MP" | "DP" | "BMC"
    | "BDC" | "EMC" | "BX" | "EX" ) as opstring →
        Some (Op opstring)
  | _ → None
```

Lex a string.

```
let lex_string i =
  match Pdfread.lex_string i with
  | LexString str → Some (Obj (LexString str))
  | _ → None
```

Lex a hexadecimal string.

```
let lex_hexstring i =
  match Pdfread.lex_hexstring i with
  | LexString str → Some (Obj (LexString str))
  | _ → None
```

Lex one token

```
let lex_next pdf resources i =
  try
    dropwhite i;
    match peek_byte i with
    | x when x = Pdfio.no_more → None
    | chr →
    match char_of_int chr with
    | '/' → lex_name i
    | '0'..'9' | '+' | '-' | '.' → lex_number i
    | 'A'..'Z' | 'a'..'z' | '\'' | '\"' -> lex_keyword pdf resources
i
       | '(' -> lex_string i
       | '[' -> nudge i; Some (Obj (LexLeftSquare))
       | ']' -> nudge i; Some (Obj (LexRightSquare))
       | '<' ->
           begin match nudge i; let c = unopt (peek_char i) in rewind
i; c with
           | '<' -> get_dictionary i
           | _ -> lex_hexstring i
           end
       | '%' -> ignore (Pdfread.lex_comment i); Some (LexComment)
       | _ ->
           for x = 1 to 10 do rewind i done;
           Printf.eprintf "“nLexing failure : chars before : ";
           for x = 1 to 10 do
             print_char (unopt (i.input_char ()))
           done;
           Printf.eprintf "“nChars after\n";
```

```
          for x = 1 to 40 do
            print_char (unopt (i.input_char ()))
          done;
          raise (PDFSemanticError "Lexing failure in content stream")
    with
      | PDFReadError r ->
          raise (PDFReadError ("Pdfpages.lex_next => " ^ r))
      | Failure "unopt" | End_of_file -> dpr "3C"; None
      | Couldn'tHandleContent -> dpr "3E"; None
  let print_lexeme = function
    | Obj p -> print_lexeme p
    | Op s -> print_string s; print_newline ()
    | PdfObj p -> print_string "PDF OBJECT\n"
    | LexInlineImage _ -> print_string "INLINE IMAGE\n"
    | LexComment -> print_string "COMMENT\n"
  (* Lex a graphics stream *)
  let rec lex_stream pdf resources i lexemes =
    match lex_next pdf resources i with
    | None -> rev lexemes
    | Some LexComment ->
        lex_stream pdf resources i lexemes
    | Some lexeme ->
        lex_stream pdf resources i (lexeme::lexemes)
  (* \section{Parsing} *)

  (* Parse a single operator and its operands, provided as a lexeme
list.  The
  string from which these lexemes were extracted is provided so that
[Op_Unknown]
  instances can be generated.  The compatibility level is also provided,
and may be
  updated.  *)
  let parse_operator compatibility string = function
    | [Obj (LexReal tx); Obj (LexReal ty); Op "Td"] -> Op_Td (tx,
ty)
    | [Obj (LexReal tx); Obj (LexReal ty); Op "TD"] -> Op_TD (tx,
ty)
    | [Obj (LexReal width); Op "w"] -> Op_w width
    | [Obj (LexReal cap); Op "J"] -> Op_J (int_of_float cap)
    | [Obj (LexReal join); Op "j"] -> Op_j (int_of_float join)
    | [Op "W"] -> Op_W
    | [Op "W*"] -> Op_W'
    | [Op "q"] -> Op_q
    | [Op "Q"] -> Op_Q
    | [Op "h"] -> Op_h
    | [Op "n"] -> Op_n
    | [Obj (LexReal x); Obj (LexReal y); Op "m"] -> Op_m (x, y)
    | [Obj (LexReal x); Obj (LexReal y); Op "l"] -> Op_l (x, y)
    | [Op "f*"] -> Op_f'
    | [Op "f"] -> Op_f
```

```
| [Op "F"] -> Op_F
| [Op "BT"] -> Op_BT
| [Op "ET"] -> Op_ET
| [Obj (LexReal leading); Op "TL"] -> Op_TL leading
| [Obj (LexName n); Obj (LexReal s); Op "Tf"] -> Op_Tf (n, s)
| [Op "T×"] -> Op_T'
| [Obj (LexString s); Op "Tj"] -> Op_Tj s
| [Obj (LexReal r); Obj (LexReal g); Obj (LexReal b); Op "RG"]
->
    Op_RG (r, g, b)
| [Obj (LexReal r); Obj (LexReal g); Obj (LexReal b); Op "rg"]
->
    Op_rg (r, g, b)
| [Obj (LexReal g); Op "G"] -> Op_G g
| [Obj (LexReal g); Op "g"] -> Op_g g
| [Obj (LexReal c); Obj (LexReal m);
   Obj (LexReal y); Obj (LexReal k); Op "k"] -> Op_k (c, m, y,
k)
| [Obj (LexReal c); Obj (LexReal m);
   Obj (LexReal y); Obj (LexReal k); Op "K"] -> Op_K (c, m, y,
k)
| [Obj (LexReal a); Obj (LexReal b); Obj (LexReal c);
   Obj (LexReal d); Obj (LexReal e); Obj (LexReal f); Op "cm"]
->
     Op_cm
       {Transform.a = a; Transform.b = b; Transform.c = c;
        Transform.d = d; Transform.e = e; Transform.f = f}
| [Obj (LexReal a); Obj (LexReal b); Obj (LexReal c);
   Obj (LexReal d); Obj (LexReal e); Obj (LexReal f); Op "Tm"]
->
     Op_Tm
       {Transform.a = a; Transform.b = b; Transform.c = c;
        Transform.d = d; Transform.e = e; Transform.f = f}
| [Obj (LexName n); Op "MP"] -> Op_MP n
| [Obj (LexName n); PdfObj p; Op "DP"] -> Op_DP (n, p)
| [Obj (LexName n); Obj o; Op "DP"] ->
    let p = snd (Pdfread.parse [o]) in Op_DP (n, p)
| [Obj (LexName n); Op "BMC"] -> Op_BMC n
| [Obj (LexName n); PdfObj p; Op "BDC"] -> Op_BDC (n, p)
| [Obj (LexName n); Obj o; Op "BDC"] ->
    let p = snd (Pdfread.parse [o]) in Op_BDC (n, p)
| [Op "EMC"] -> Op_EMC
| [Obj (LexName n); Op "gs"] -> Op_gs n
| [Obj (LexName n); Op "Do"] -> Op_Do n
| [Obj (LexName n); Op "CS"] -> Op_CS n
| [Obj (LexName n); Op "cs"] -> Op_cs n
| [Obj (LexReal x1); Obj (LexReal y1); Obj (LexReal x2);
   Obj (LexReal y2); Obj (LexReal x3); Obj (LexReal y3);
   Op "c"] -> Op_c (x1, y1, x2, y2, x3, y3)
| [Obj (LexReal x2); Obj (LexReal y2);
```

```
        Obj (LexReal x3); Obj (LexReal y3);
        Op "v"] -> Op_v (x2, y2, x3, y3)
    | [Obj (LexReal x1); Obj (LexReal y1);
        Obj (LexReal x3); Obj (LexReal y3);
        Op "y"] -> Op_y (x1, y1, x3, y3)
    | [Op "B"] -> Op_B
    | [Op "B × "] -> Op_B'
    | [Op "b"] -> Op_b
    | [Op "b × "] -> Op_b'
    | [Op "S"] -> Op_S
    | [Op "s"] -> Op_s
    | [Obj (LexReal x); Obj (LexReal y);
        Obj (LexReal w); Obj (LexReal h);
        Op "re"] -> Op_re (x, y, w, h)
    | [Obj (LexName n); Op "ri"] -> Op_ri n
    | [Obj (LexReal i); Op "i"] -> Op_i (int_of_float i)
    | [Op "BX"] -> incr compatibility; Op_BX
    | [Op "EX"] -> decr compatibility; Op_EX
    | [Obj (LexReal m); Op "M"] -> Op_M m
    | [Obj (LexString s); Op "“’"] -> Op_' s
    | [Obj (LexReal aw); Obj (LexReal ac); Obj (LexString s); Op "“""] →

      Op_" (aw, ac, s)
  | [Obj (LexReal wx); Obj (LexReal wy); Op "d0"] →
      Op_d0 (wx, wy)
  | [Obj (LexReal wx); Obj (LexReal wy);
     Obj (LexReal llx); Obj (LexReal lly);
     Obj (LexReal urx); Obj (LexReal ury); Op "d1"] →
      Op_d1 (wx, wy, llx, lly, urx, ury)
  | [Obj (LexName n); Op "sh"] → Op_sh n
  | [Obj (LexReal tc); Op "Tc"] → Op_Tc tc
  | [Obj (LexReal tw); Op "Tw"] → Op_Tw tw
  | [Obj (LexReal tz); Op "Tz"] → Op_Tz tz
  | [Obj (LexReal tr); Op "Tr"] → Op_Tr (toint tr)
  | [Obj (LexReal ts); Op "Ts"] → Op_Ts ts
  | [LexInlineImage d] → InlineImage d
  | ls →
      (* More complicated things are parsed by reversing the lexemes so we
may inspect the operator. *)
      let reals_of_real_lexemes errtext lexemes =
        let real_of_real_lexeme errtext = function
          | Obj (LexReal n) → n
          | _ → raise (PDFSemanticError errtext)
        in
          (* Adobe Distiller 5.0.5 produces bad Op_scn *)
          try
            map (real_of_real_lexeme errtext) lexemes
          with
            _ → dpr "3E"; [0.; 0.; 0.; ]
      in
```

```
match rev ls with
| Op "sc"::nums →
    Op_sc (rev (reals_of_real_lexemes "Malformed 'sc'" nums))
| Op "SC"::nums →
    Op_SC (rev (reals_of_real_lexemes "Malformed 'SC'" nums))
| Op "scn"::Obj (LexName n) :: rest →
    Op_scnName (n, rev (reals_of_real_lexemes "scn" rest))
| Op "SCN"::Obj (LexName n) :: rest →
    Op_SCNName (n, rev (reals_of_real_lexemes "SCN" rest))
| Op "scn"::nums →
    Op_scn (rev (reals_of_real_lexemes "Malformed 'scn'" nums))
| Op "SCN"::nums →
    Op_SCN (rev (reals_of_real_lexemes "Malformed 'SCN'" nums))
| Op "d"::Obj (LexReal phase) :: Obj LexRightSquare :: r →
    begin match rev r with
    | Obj LexLeftSquare :: t →
        let reals =
          map
            (function
             | (Obj (LexReal i)) → i
             | _ →
               raise (PDFSemanticError "malformed 'd' op"))
            t
        in
          Op_d (reals, phase)
    | _ → raise (PDFSemanticError "malformed 'd' op")
    end
| Op "TJ"::Obj LexRightSquare :: r →
    begin match rev r with
    | Obj LexLeftSquare :: t →
        let elements =
          map
            (function
             | (Obj (LexReal i)) → Real i
             | (Obj (LexString s)) → String s
             | _ → raise (PDFSemanticError "malformed TJ elt"))
            t
        in
          Op_TJ (Array elements)
    | _ → raise (PDFSemanticError "malformed TJ op")
    end
| Op _ :: _ → Op_Unknown string
| _ →
   Printf.eprintf "Empty or malformed graphics operation.";
   Op_Unknown string
```

Split the lexemes into sections (zero or more operands followed by an operator) and parse each.

```
let split s =
  let h, t =
```

```
    cleavewhile (function Op _ | LexInlineImage _ -> false | _ -> true) s
  in
    match t with
    | [] -> raise (PDFSemanticError "premature graphics stream end")
    | f :: l -> append h [f], l

let rec parse_lexemes compatibility ls ops =
  let make_real = function
    | Obj (LexInt i) -> Obj (LexReal (float_of_int i))
    | x -> x
  in
    match ls with
    | [] -> rev ops
    | _ ->
      let section, remaining = split ls in
        let op =
          parse_operator
            compatibility
            (string_of_lexemes section)
            (map make_real section)
        in
          parse_lexemes compatibility remaining (op :: ops)
```

▷ Parse, given a list of streams. The contents of a single PDF page can be split over several streams, which must be concatenated at the lexeme level. Concatenate bytestreams, padding with whitespace

```
let concat_bytestreams ss =
  let total_length = fold_left ( + ) 0 (map stream_size ss) in
    let s' = mkstream (total_length + length ss) in
      let p = ref 0 in
        iter
          (fun s ->
             for x = 0 to stream_size s - 1 do
               sset s' !p (sget s x);
               incr p
             done;
             sset s' !p (int_of_char ' ');
             incr p)
          ss;
        s'

let parse_stream pdf resources streams =
  let stream = match streams with [s] -> s | _ -> concat_bytestreams streams in
  let i = input_of_bytestream stream in
    let lexemes = lex_stream pdf resources i [] in
      parse_lexemes (ref 0) lexemes []
```

▷ Parse the operators in a list of streams.

```
let parse_operators pdf resources streams =
  let rawstreams =
    map
```

```
      (fun c →
        Pdfcodec.decode_pdfstream pdf c;
        bigarray_of_stream c)
      streams
  in
    parse_stream pdf resources rawstreams
```

10.3 Flattening

Give a bigarray representing a list of graphics operators.

```
let stream_of_lexemes ls =
  let oplists = ref []
  and ls = ref ls in
    while !ls ≠ [] do
      let l, l' = split !ls in
        ls := l';
        oplists =| l
    done;
    let strings =
      rev_map
        (fun ls →
           let s = string_of_lexemes ls in
             if lastchar s ≠ Some ' ' then s ^ " " else s)
        !oplists
    in
      let total_length =
        fold_left ( + ) 0 (map String.length strings)
      in
        let s = mkstream total_length
        and strings = ref strings
        and pos = ref 0 in
          while !strings ≠ [] do
            let str = hd !strings in
              let l = String.length str in
                if l > 0 then
                  for n = 0 to l − 1 do
                    sset s !pos (int_of_char str.[n]);
                    incr pos
                  done;
                strings := tl !strings
          done;
          s

let print_stream s =
  if stream_size s > 0 then
    for x = 0 to stream_size s − 1 do
      Printf.printf "%c" (char_of_int (sget s x))
    done;
```

```
  print_newline ()
```

▷ Make a stream from a list of operators.

```
let stream_of_ops ops =
  let data = stream_of_lexemes (flatten (map lexemes_of_op ops)) in
    Pdf.Stream
      (ref (Pdf.Dictionary [("/Length", Pdf.Integer (stream_size data))], Pdf.Got data))
```

11 Module PDFFun
PDF Functions

open Utility
open Pdf

11.1 Types

▷ Postscript calculator functions.

type *calculator* =
| If **of** *calculator list*
| IfElse **of** *calculator list* × *calculator list*
| Bool **of** *bool*
| Float **of** *float*
| Int **of** *int32*
| Abs | Add | Atan | Ceiling | Cos | Cvi | Cvr
| Div | Exp | Floor | Idiv | Ln | Log | Mod
| Mul | Neg | Round | Sin | Sqrt | Sub | Truncate
| And | Bitshift | Eq | Ge | Gt | Le | Lt | Ne | Not
| Or | Xor | Copy | Exch | Pop | Dup | Index | Roll

▷ Sampled functions.

type *sampled* =
{*size* : *int list*;
order : *int*;
encode : *float list*;
decode : *float list*;
bps : *int*;
samples : *int32 array*}

▷ Interpolation functions.

and *interpolation* =
{*c0* : *float list*;
c1 : *float list*;
n : *float*}

▷ Stitching functions.

```
and stitching =
  {functions : pdf_fun list;
   bounds : float list;
   stitch_encode : float list}
```

▷ Collect the above types into a single type.

```
and pdf_fun_kind =
  | Interpolation of interpolation
  | Stitching of stitching
  | Sampled of sampled
  | Calculator of calculator list
```

▷ Main type.

```
and pdf_fun =
  {func : pdf_fun_kind;
   domain : float list;
   range : float list option}
```

11.2 Printing functions

Build a string of a calculator function. For debug only, since could exceed string length limit.

```
let rec string_of_calculator_inner = function
  | If l →
      string_of_calculator l ^ " if"
  | IfElse (l, l′) →
      string_of_calculator l ^ " " ^ string_of_calculator l′ ^ " ifelse"
  | Bool true → "true"
  | Bool false → "false"
  | Float f → string_of_float f
  | Int i → Int32.to_string i
  | Abs → "abs" | Add → "add" | Atan → "atan" | Ceiling → "ceiling"
  | Cos → "cos" | Cvi → "cvi" | Cvr → "cvr" | Div → "div"
  | Exp → "exp" | Floor → "floor" | Idiv → "idiv" | Ln → "ln"
  | Log → "log" | Mod → "mod" | Mul → "mul" | Neg → "neg"
  | Round → "round" | Sin → "sin" | Sqrt → "sqrt" | Sub → "sub"
  | Truncate → "truncate" | And → "and" | Bitshift → "bitshift"
  | Eq → "eq" | Ge → "ge" | Gt → "gt" | Le → "le" | Lt → "lt"
  | Ne → "ne" | Not → "not" | Or → "or" | Xor → "xor"
  | Copy → "copy" | Exch → "exch" | Pop → "pop" | Dup → "dup"
  | Index → "index" | Roll → "roll"

and string_of_calculator cs =
  let ops =
    fold_right ( ^ ) (interleave " " (map string_of_calculator_inner cs)) ""
  in
    "{" ^ ops ^ "}"
```

Print a function out for debug.

```
let rec print_function f =
  print_string "Domain...\n";
  print_floats f.domain;
  begin match f.range with
  | None → print_string "null range\n"
  | Some values → print_floats values
  end;
  match f.func with
  | Sampled s →
      print_string "Sampled\n";
      print_string "size:  ";
      print_ints s.size;
      print_string "order:  ";
      print_int s.order;
      print_string "\nencode:\n";
      print_floats s.encode;
      print_string "decode:\n";
      print_floats s.decode;
      print_string "original bits per sample..\n";
      print_int s.bps;
      print_string "\ndata:\n";
      print_int32s (Array.to_list s.samples)
  | Interpolation i →
      print_string "Interpolation\n";
      print_string "C0:\n";
      print_floats i.c0;
      print_string "C1:\n";
      print_floats i.c1;
      Printf.printf "n = %f\n" i.n;
  | Stitching s →
      print_string "Stitching\n";
      iter print_function s.functions;
      print_string "Bounds:\n";
      print_floats s.bounds;
      print_string "Encode:\n";
      print_floats s.stitch_encode;
  | Calculator c →
      print_string "Calculator:\n";
      print_string (string_of_calculator c)
```

11.3 Parsing Calculator Functions

```
let keyword_of_string = function
  | "abs" → Abs | "add" → Add | "atan" → Atan | "ceiling" → Ceiling
  | "cos" → Cos | "cvr" → Cvr | "div" → Div | "exp" → Exp
  | "floor" → Floor | "idiv" → Idiv | "ln" → Ln | "log" → Log
  | "mod" → Mod | "mul" → Mul | "neg" → Neg | "round" → Round
  | "sin" → Sin | "sqrt" → Sqrt | "sub" → Sub
```

```
    | "truncate" → Truncate | "and" → And | "bitshift" → Bitshift
    | "eq" → Eq | "ge" → Ge | "gt" → Gt | "le" → Le | "lt" → Lt
    | "ne" → Ne | "not" → Not | "or" → Or | "xor" → Xor
    | "copy" → Copy | "exch" → Exch | "pop" → Pop
    | "dup" → Dup | "index" → Index | "roll" → Roll
    | s →
        flprint ("Bad keyword " ^ s); raise (Assert_failure ("", 0, 0))

let parse_calculator s =
  let lexemes =
    Cgenlex.lex (Pdfio.input_of_bytestream (bytestream_of_string s))
  in
    let rec strip_outer_braces = function
      | Cgenlex.Ident ("{" | "}") :: t →
          rev (strip_outer_braces (rev t))
      | x → x
    and group_operators = function
      | [] → []
      | Cgenlex.Ident "{"::t →
          let ops, rest = cleavewhile (neq (Cgenlex.Ident "}")) t in
            ops :: group_operators (tl rest)
      | h :: t → [h] :: group_operators t
    and parse = function
      | [] → []
      | l :: l' :: [Cgenlex.Ident "ifelse"] :: t →
          IfElse (procss l, procss l') :: parse t
      | l :: [Cgenlex.Ident "if"] :: t → If (procss l) :: parse t
      | [Cgenlex.Ident "true"] :: t → Bool true::parse t
      | [Cgenlex.Ident "false"] :: t → Bool false::parse t
      | [Cgenlex.Float f] :: t → Float f :: parse t
      | [Cgenlex.Int i] :: t → Int (i32ofi i) :: parse t (* FIXME: range *)
      | [Cgenlex.Ident x] :: t → keyword_of_string x :: parse t
      | h :: _ → raise (Failure "Bad lexeme")
    and procss lexemes =
      try
        parse (group_operators (strip_outer_braces lexemes))
      with
      _ → raise (Pdf.PDFError "Cannot parse Type 4 function")
    in
      procss lexemes
```

11.4 Parsing functions

```
let rec parse_function pdf f =
  let f = direct pdf f in
    let getnum_direct o = getnum (direct pdf o) in
      let domain =
        match lookup_fail "No /Domain" pdf "/Domain" f with
```

```
    | Array ns → map getnum_direct ns
    | _ → raise (PDFError "Bad /Domain")
and range =
  match lookup_direct pdf "/Range" f with
  | Some (Array ns) → Some (map getnum_direct ns)
  | _ → None
in
let func =
  match lookup_fail "no /FunctionType" pdf "/FunctionType" f with
  | Integer 0 →
      let size =
        match lookup_fail "no /Size (sampled fun)" pdf "/Size" f with
        | Array ns →
            map
              (function
               | Integer n → n
               | _ → raise (PDFError "bad /Size (sampled fun)"))
              ns
        | _ → raise (PDFError "Bad /Size (sampled fun)")
      in
        let order =
          match lookup_direct pdf "/Order" f with
          | Some (Integer n) → n
          | _ → 1
        and encode =
          match lookup_direct pdf "/Encode" f with
          | Some (Array ns) when length ns = 2 × length size →
              map getnum ns
          | _ →
              interleave_lists
                (many 0. (length size))
                (map (fun x → float (x − 1)) size)
        and decode =
          match lookup_direct pdf "/Decode" f with
          | Some (Array ns) → map getnum ns
          | _ →
              match range with
              | Some r → r
              | None → raise (PDFError "No /Range")
        in
          let bitspersample =
            match
              lookup_fail "no /BitsPerSample" pdf "/BitsPerSample" f
            with
            | Integer i → i
            | _ → raise (PDFError "Bad /BitsPerSample")
          in
            let data =
              Pdfcodec.decode_pdfstream pdf f;
              let samples =
```

```
              fold_left ( × ) 1 size × (length decode / 2)
            and bitstream =
              match f with
              | Stream {contents = _, Got data} →
                  Pdfio.bitstream_of_input (Pdfio.input_of_bytestream data)
              | _ → raise (Assert_failure ("", 0, 0))

            in
              let data = Array.make samples 1l in
                for i = 0 to Array.length data − 1 do
                  data.(i) ← Pdfio.getval_32 bitstream bitspersample
                done;
                data
          in
            Sampled
             {size = size;
              order = order;
              encode = encode;
              decode = decode;
              bps = bitspersample;
              samples = data}
  | Integer 2 →
      let c0 =
        match lookup_direct pdf "/C0" f with
        | Some (Array ns) → map getnum_direct ns
        | _ → [0.]
      and c1 =
        match lookup_direct pdf "/C1" f with
        | Some (Array ns) → map getnum_direct ns
        | _ → [1.]
      and n =
        getnum (lookup_fail "No /N in Type 2 fun" pdf "/N" f)
      in
        Interpolation {c0 = c0; c1 = c1; n = n}
  | Integer 3 →
      let functions =
        match lookup_fail "no /Functions" pdf "/Functions" f with
        | Array fs → fs
        | _ → raise (PDFError "Bad /Functions")
      and bounds =
        match lookup_fail "no /Bounds" pdf "/Bounds" f with
        | Array fs → fs
        | _ → raise (PDFError "Bad /Bounds")
      and encode =
        match lookup_fail "no /Encode" pdf "/Encode" f with
        | Array fs → fs
        | _ → raise (PDFError "Bad /Bounds")
      in
        Stitching
          {functions = map (parse_function pdf) functions;
```

```
            bounds = map getnum_direct bounds;
            stitch_encode = map getnum_direct encode}
    | Integer 4 →
        (* Read contents of stream, build string, parse. *)
        Pdfcodec.decode_pdfstream pdf f;
        begin match f with
        | Stream {contents = _, Got data} →
            Calculator (parse_calculator (string_of_bytestream data))
        | _ → raise (PDFError "This is not a function")
        end
    | _ → raise (PDFError "Unknown function type")
  in
    {domain = domain; range = range; func = func}
```

11.5 Evaluating Sampled Functions

▷ Inappropriate inputs have been given to a function.

```
exception BadFunctionEvaluation of string

let interpolate x xmin xmax ymin ymax =
  ymin +. ((x -. xmin) *. ((ymax -. ymin) /. (xmax -. xmin)))
```

Evaluate a sampled function. We support only linear interpolation and then only sensibly for one-dimensional functions. Although results will be produced for higher dimensions, the results will not be fully accurate.

```
let eval_function_sampled f s clamped_inputs =
  (* 1. Encode the input values *)
  let range =
    match f.range with
    | None → raise (BadFunctionEvaluation "No Range")
    | Some r → r
  in
    let d, d' = split (pairs_of_list f.domain)
    and e, e' = split (pairs_of_list s.encode)
    and dec, dec' = split (pairs_of_list s.decode)
    and r, r' = split (pairs_of_list range) in
      let encoded_inputs =
        map5 interpolate clamped_inputs d d' e e'
      in
      (* 2. Clip to the size of the table dimension *)
      let clamped_encoded_inputs =
        map2
          (fun i s → fmin (fmax i 0.) (float s -. 1.))
          encoded_inputs
          s.size
      in
        let read_table inputs =
          let vals_to_read = length range / 2 in
```

```
        let size = s.size in
        if length size ≠ length inputs then
          raise (BadFunctionEvaluation "Incompatible /Size with inputs");
        let pos =
          let multipliers =
            1::
            map
              (function x → fold_left ( × ) 1 (take size x))
              (ilist_fail_null 1 (length inputs − 1))
          in
            fold_left ( + ) 0 (map2 ( × ) inputs multipliers) × vals_to_read
        in
          Array.to_list (Array.sub s.samples pos vals_to_read)
    in
      (* 3. Read values from table. For now, just linear iterpolation. *)
      let ceilings =
        map (fun x → toint (ceil x)) clamped_encoded_inputs
      and floors =
        map (fun x → toint (floor x)) clamped_encoded_inputs
      in
        let outputs =
          let ceiling_results = read_table ceilings
          and floor_results = read_table floors in
            map2
              (fun x y →
                 Int32.to_float x /. 2. +. Int32.to_float y /. 2.)
              ceiling_results
              floor_results
        in
          (* 4. Decode output values *)
          let outputs_decoded =
            map5
              interpolate
              outputs
              (many 0. (length outputs))
              (many (2. ** float s.bps −. 1.) (length outputs))
              dec dec′
          in
            map3 (fun x r r′ → fmin (fmax x r) r′) outputs_decoded r r′
```

11.6 Evaluating Calculator Functions

```
let eval_function_calculator clamped_inputs ops =
  let s =
    ref (map (fun i → Float i) (rev clamped_inputs))
  and typecheck () =
    raise (BadFunctionEvaluation "Type error")
  in
```

```
let rec getfloat () =
  match !s with
  | Int i :: r → s := r; i32tof i
  | Float f :: r → s := r; f
  | _ → typecheck ()
and getint () =
  match !s with
  | Int i :: r → s := r; i
  | _ → typecheck ()
and getfloats () =
  let x = getfloat () in x, getfloat ()
and getints () =
  let x = getint () in x, getint ()
in
  let rec eval k =
        match k with
  | If l →
      begin match !s with
      | Bool b :: r → s := r; if b then iter eval l
      | _ → typecheck ()
      end
  | IfElse (l, l′) →
      begin match !s with
      | Bool b :: r → s := r; iter eval (if b then l else l′)
      | _ → typecheck ()
      end
  | (Bool _ | Float _ | Int _) as immediate →
      s =| immediate
  | Abs →
      begin match !s with
      | Float f :: r → s := Float (fabs f) :: r
      | Int i :: r →
          let out =
            if i = Int32.min_int
              then (Float (i32tof Int32.max_int))
              else (Int (Int32.abs i))
          in
            s := out :: r
      | _ → typecheck ()
      end
  | Add →
      begin match !s with
      | Int i :: Int i′ :: r →
          s := Int (i32add i i′) :: r          (* FIXME: Overflow to float *)
      | Int i :: Float f :: r
      | Float f :: Int i :: r →
          s := Float (i32tof i +. f) :: r
      | Float f :: Float f′ :: r →
          s := Float (f +. f′) :: r
      | _ → typecheck ()
```

```
      end
  | Atan →
      let num, den = getfloats () in
        let result = atan2 num den in
          s := Float result :: tl (tl !s)
  | Ceiling →
      begin match !s with
      | Float f :: r → s := Float (ceil f) :: r
      | Int _ :: _ → ()
      | _ → typecheck ()
      end
  | Cos →
      let f = getfloat () in
        s := Float (cos (rad_of_deg f)) ::!s
  | Cvi →
      begin match !s with
      | Int _ :: r → ()
      | Float f :: r → s := Int (Int32.of_float (floor f)) :: r
      | _ → typecheck ()
      end
  | Cvr →
      begin match !s with
      | Int i :: r → s := Float (i32tof i) :: r
      | Float f :: r → ()
      | _ → typecheck ()
      end
  | Div →
      let n, n' = getfloats () in
        s := Float (n /. n') ::!s
  | Exp →
      let bse, exponent = getfloats () in
        s := Float (bse ** exponent) ::!s
  | Floor →
      begin match !s with
      | Int i :: r → ()
      | Float f :: r → s := Int (Int32.of_float (floor f)) :: r
      | _ → typecheck ()
      end
  | Idiv →
      let i, i' = getints () in
        s := Int (i32div i i') ::!s
  | Ln →
      let f = getfloat () in
        s := Float (log f) ::!s
  | Log →
      let f = getfloat () in
        s := Float (log10 f) ::!s
  | Mod →
      let i, i' = getints () in
        s := Int (Int32.rem i i') ::!s
```

```
| Mul →
    begin match !s with
    | Int i :: Int i' :: r →
        s := Int (i32mul i i') :: r          (* FIXME: Overflow to float *)
    | Int i :: Float f :: r
    | Float f :: Int i :: r →
        s := Float (i32tof i *. f) :: r
    | Float f :: Float f' :: r →
        s := Float (f *. f') :: r
    | _ → typecheck ()
    end
| Neg →
    begin match !s with
    | Float f :: r → s := Float (~-.f) :: r
    | Int i :: r →
        let out =
          if i = Int32.min_int
            then (Float (i32tof Int32.max_int))
            else (Int (Int32.neg i))
        in
          s := out :: r
    | _ → typecheck ()
    end
| Round →
    begin match !s with
    | Int _ :: _ → ()
    | Float f :: r → s := Int (Int32.of_float (round f)) :: r
    | _ → typecheck ()
    end
| Sin →
    let f = getfloat () in
      s := Float (sin (rad_of_deg f)) ::!s
| Sqrt →
    let f = getfloat () in
      s := Float (sqrt f) ::!s
| Sub →
    begin match !s with
    | Int i :: Int i' :: r →
        s := Int (i32sub i' i) :: r                              (* *)
    | Int i :: Float f :: r →
        s := Float (f -. i32tof i) :: r
    | Float f :: Int i :: r →
        s := Float (i32tof i -. f) :: r
    | Float f :: Float f' :: r →
        s := Float (f' -. f) :: r
    | _ → typecheck ()
    end
| Truncate →
    begin match !s with
    | Int _ :: _ → ()
```

TO FIX:
Overflow to float

```
      | Float f :: r → s := Int (i32ofi (toint f)) :: r
      | _ → typecheck ()
      end
  | And →
      begin match !s with
      | Int i :: Int i′ :: r →
          s := Int (Int32.logand i i′) :: r
      | Bool b :: Bool b′ :: r →
          s := Bool (b ∧ b′) :: r
      | _ → typecheck ()
      end
  | Bitshift →
      let i = getint () in
        let shift = i32toi (getint ()) in
          let r =
            if i < 0l
              then Int32.shift_left i shift
              else Int32.shift_right_logical i (abs shift)
          in
            s := Int r ::!s
  | Eq →
      begin match !s with
      | a :: b :: r → s := Bool (a = b) :: r
      | _ → typecheck ()
      end
  | Ge →
      begin match !s with
      | a :: b :: r → s := Bool (a ≥ b) :: r
      | _ → typecheck ()
      end
  | Gt →
      begin match !s with
      | a :: b :: r → s := Bool (a > b) :: r
      | _ → typecheck ()
      end
  | Le →
      begin match !s with
      | a :: b :: r → s := Bool (a ≤ b) :: r
      | _ → typecheck ()
      end
  | Lt →
      begin match !s with
      | a :: b :: r → s := Bool (a < b) :: r
      | _ → typecheck ()
      end
  | Ne →
      begin match !s with
      | a :: b :: r → s := Bool (a ≠ b) :: r
      | _ → typecheck ()
      end
```

```
| Not →
    begin match !s with
    | Int i :: r →
        s := Int (Int32.lognot i) :: r
    | Bool b :: Bool b′ :: r →
        s := Bool (¬ b) :: r
    | _ → typecheck ()
    end
| Or →
    begin match !s with
    | Int i :: Int i′ :: r →
        s := Int (Int32.logor i i′) :: r
    | Bool b :: Bool b′ :: r →
        s := Bool (b ∨ b′) :: r
    | _ → typecheck ()
    end
| Xor →
    begin match !s with
    | Int i :: Int i′ :: r →
        s := Int (Int32.logxor i i′) :: r
    | Bool b :: Bool b′ :: r →
        s := Bool (b |&| b′) :: r
    | _ → typecheck ()
    end
| Copy →
    begin match !s with
    | Int i :: r when i ≥ 0l →
        s := take r (i32toi i) @ r
    | _ → typecheck ()
    end
| Exch →
    begin match !s with
    | a :: b :: r → s := b :: a :: r
    | _ → typecheck ()
    end
| Pop →
    begin match !s with
    | a :: r → s := r
    | _ → typecheck ()
    end
| Dup →
    begin match !s with
    | a :: r → s := a :: a :: r
    | _ → typecheck ()
    end
| Index →
    begin match !s with
    | Int i :: r when i ≥ 0l →
        let v = select (i32toi i + 1) r in
          s := v :: r
```

```
          | _ → typecheck ()
          end
      | Roll →
          let rec rotate j l =
            if j = 0 then l else
            match l with
            | [] → []
            | h :: t → rotate (j − 1) (t @ [h])
          and rotate_o j l =
            if j = 0 then l else
              match rev l with
              | [] → []
              | h :: t → rotate (j − 1) ([h] @ rev t)
          in
            begin match !s with
            | Int j :: Int n :: r when n ≥ 1l →
                let j = i32toi j and n = i32toi n in
                  let vals, rest = cleave r n in
                    let newvals =
                      if j > 0
                        then rotate j vals
                        else rotate_o −j vals
                    in
                    s := newvals @ rest
            | _ → typecheck ()
            end
  in
    try
      iter eval ops;
      rev_map
        (function
         | Float x → x
         | _ → raise (BadFunctionEvaluation "Type 4"))
        !s
    with
      _ → raise (BadFunctionEvaluation "Type 4")
```

11.7 Evaluating functions

▷ Evaluate a function on some inputs.

```
let rec eval_function f inputs =
  let rec clampvals vals domain =
    let clampval v d d′ =
      if v < d then d else if v > d′ then d′ else v
    in
      match vals, domain with
      | [], [] → []
      | v :: vs, d :: d′ :: ds →
```

```
          clampval v d d' :: clampvals vs ds
      | _ → raise (BadFunctionEvaluation "Domain wrong")
in
  let clamped_inputs = clampvals inputs f.domain in
    let outputs =
      match f.func with
      | Calculator ops →
          eval_function_calculator clamped_inputs ops
      | Sampled s →
          eval_function_sampled f s clamped_inputs
      | Interpolation f →
          let interp n c0 c1 i =
            try
              map2
                (fun c0 c1 → c0 +. i ** n *. (c1 -. c0)) c0 c1
            with
              Invalid_argument _ →
                raise (BadFunctionEvaluation "Interpolation")
          in
            flatten (map (interp f.n f.c0 f.c1) clamped_inputs)
      | Stitching s →
          match clamped_inputs, f.domain with
          | [i], [d0; d1] →
              let points = [d0] @ s.bounds @ [d1] in
                let rec extract_subfunction points funs n =
                  match points, funs with
                  | p :: p' :: _, f :: _ when i ≥ p ∧ i < p' → p, p', f, n
                  | p :: p' :: _, f :: _ when i = p' → p, p', f, n
                  | _ :: ps, _ :: fs → extract_subfunction ps fs (n + 1)
                  | _ →
                      raise (BadFunctionEvaluation "stitching:  funs")
                in
                  let d0', d1', f, n = extract_subfunction points s.functions 0 in
                   let encode =
                     try
                       let a, b, c, d =
                         select (n + 1) points, select (n + 2) points,
                         select (n + 1) s.stitch_encode, select (n + 2) s.stitch_encode
                       in
                         fun x → interpolate x a b c d
                     with
                       Invalid_argument "select" →
                         raise (BadFunctionEvaluation "stitching:  encode/domain")
                   in
                     eval_function f [encode i]
          | _ → raise (BadFunctionEvaluation "stitching:  Bad arity")
    in
      match f.range with
      | None → outputs
      | Some range → clampvals outputs range
```

```
let funtype_of_function f =
  match f.func with
  | Interpolation _ → 2
  | Stitching _ → 3
  | Sampled _ → 0
  | Calculator _ → 4

let mkreal x = Pdf.Real x

let entries_of_interpolation i =
  ["/C0", Pdf.Array (map mkreal i.c0);
   "/C1", Pdf.Array (map mkreal i.c1);
   "/N", Pdf.Real i.n]

let rec entries_of_stitching pdf i =
  (* Add the functions as objects. *)
  let numbers =
    map (fun f → Pdf.Indirect (Pdf.addobj pdf (pdfobject_of_function pdf f))) i.functions
  in
    ["/Functions", Pdf.Array numbers;
     "/Bounds", Pdf.Array (map mkreal i.bounds);
     "/Encode", Pdf.Array (map mkreal i.stitch_encode)]

and extra_entries_of_function pdf f =
  match f.func with
  | Interpolation i → entries_of_interpolation i
  | Stitching s → entries_of_stitching pdf s
  | Sampled s → []
  | Calculator c → []

and pdfobject_of_function pdf f =
  let domain =
    Pdf.Array (map (function x → Pdf.Real x) f.domain)
  and range =
    match f.range with
    | None → []
    | Some fs → ["/Range", Pdf.Array (map (function x → Pdf.Real x) fs)]
  in
    Pdf.Dictionary
      (["/FunctionType", Pdf.Integer (funtype_of_function f); "/Domain", domain]
      @ range @ extra_entries_of_function pdf f)
```

12 Module PDFImage
PDF Images

open Utility

What's supported and needs supporting
Unsupported CODECs: DCTDecode, JBIG2Decode, JPXDecode

RGB, 8bpc
CMYK, 8bpc
Gray, 8bpc
Black and white, 1bpp. The only one that /Decode works for
Indexed, RGB and CMYK, 8bpp
Indexed, RGB and CMYK, 4bpp
Separation, CMYK
ICCBased, knows how to find alternate colorspace

```
type pixel_layout =
  | BPP1
  | BPP8
  | BPP24
  | BPP48
```

FIXME: We need to deal with decode and other things for JPEG, if we're not going to decode them.

```
type image =
  | JPEG of bytestream
  | JPEG2000 of bytestream
  | JBIG2 of bytestream
  | Raw of int × int × pixel_layout × bytestream
```

```
let string_of_layout = function
  | BPP1 → "BPP1"
  | BPP8 → "BPP8"
  | BPP24 → "BPP24"
  | BPP48 → "BPP48"
```

```
let string_of_image = function
  | JPEG _ → "JPEG"
  | JPEG2000 _ → "JPEG2000"
  | JBIG2 _ → "JBIG2"
  | Raw (w, h, layout, data) →
      "RAW: " ^ string_of_int w ^ " " ^ string_of_int h
      ^ " " ^ string_of_layout layout ^ " bytes of data = "
      ^ string_of_int (stream_size data)
```

FIXME: Only copes with 1 0 for now, and only 8BPP

```
let decode entry image =
  match entry, image with
  | Some (Pdf.Array [Pdf.Integer 1; Pdf.Integer 0]), Raw (w, h, BPP24, s) →
      for x = 0 to (stream_size s / 3) − 1 do
        sset s (x × 3) (255 − sget s (x × 3));
        sset s (x × 3 + 1) (255 − sget s (x × 3 + 1));
        sset s (x × 3 + 2) (255 − sget s (x × 3 + 2))
      done
  | _ → ()
```

Decode until it is either plain or a type of decoding we can't deal with natively.

```
let rec decode_to_image pdf = function
  | Pdf.Stream {contents = Pdf.Dictionary d, s} as stream →
      begin match lookup "/Filter" d with
      | None
      | Some (Pdf.Array [])
      | Some (Pdf.Name ("/DCTDecode" | "/DCT" | "/JBIG2Decode" | "/JPXDecode"))
      | Some (Pdf.Array [Pdf.Name ("/DCTDecode" | "/DCT" | "/JBIG2Decode" |
 "/JPXDecode")]) → ()
      | _ →
          Pdfcodec.decode_pdfstream_onestage pdf stream;
          decode_to_image pdf stream
      end
  | _ → raise (Pdf.PDFError "decode_to_image:  bad stream")
```

Basic CMYK to RGB conversion

```
let rgb_of_cmyk c m y k =
  let c = float c and m = float m and y = float y and k = float k in
  let r = 255. −. fmin 255. ((c /. 255.) *. (255. −. k) +. k)
  and g = 255. −. fmin 255. ((m /. 255.) *. (255. −. k) +. k)
  and b = 255. −. fmin 255. ((y /. 255.) *. (255. −. k) +. k) in
    toint r, toint g, toint b

let read_cmyk_8bpp_as_rgb24 width height data =
  let data' = mkstream (width × height × 3) in
    for p = 0 to width × height − 1 do
      let c = sget data (p × 4)
      and m = sget data (p × 4 + 1)
      and y = sget data (p × 4 + 2)
      and k = sget data (p × 4 + 3) in
        let r, g, b = rgb_of_cmyk c m y k in
```

```
        sset data' (p × 3) r;
        sset data' (p × 3 + 1) g;
        sset data' (p × 3 + 2) b
    done;
    data'

let read_gray_8bpp_as_rgb24 width height data =
  let data' = mkstream (width × height × 3) in
    for pout = 0 to width × height − 1 do
      sset data' (pout × 3) (sget data pout);
      sset data' (pout × 3 + 1) (sget data pout);
      sset data' (pout × 3 + 2) (sget data pout);
    done;
    data'
```

Input is 1bpp, rows padded to bytes.

```
let read_1bpp_as_rgb24 width height s =
  let s' = mkstream (width × height × 3)
  and s_bits = Pdfio.bitstream_of_input (Pdfio.input_of_bytestream s) in
    let pout = ref 0 in
      for row = 0 to height − 1 do
        let bits_to_do = ref width in
          while !bits_to_do > 0 do
            let bit = if Pdfio.getbit s_bits then 255 else 0 in
              sset s' !pout bit;
              sset s' (!pout + 1) bit;
              sset s' (!pout + 2) bit;
              decr bits_to_do;
              pout += 3
          done;
          Pdfio.align s_bits
      done;
      s'
```

4bpp, rows padded to bytes.

```
let read_4bpp_gray_as_rgb24 width height s =
  let s' = mkstream (width × height × 3)
  and s_bits = Pdfio.bitstream_of_input (Pdfio.input_of_bytestream s) in
    let pout = ref 0 in
      for row = 0 to height − 1 do
        let pix_to_do = ref width in
          while !pix_to_do > 0 do
            let a = if Pdfio.getbit s_bits then 1 else 0 in
            let b = if Pdfio.getbit s_bits then 1 else 0 in
            let c = if Pdfio.getbit s_bits then 1 else 0 in
            let d = if Pdfio.getbit s_bits then 1 else 0 in
              let col = (a × 8 + b × 4 + c × 2 + d) × (16 + 1) in
                sset s' !pout col;
                sset s' (!pout + 1) col;
                sset s' (!pout + 2) col;
                decr pix_to_do;
```

```
            pout += 3
          done;
          Pdfio.align s_bits
        done;
        s'

let read_8bpp_indexed_as_rgb24 table width height s =
  let s' = mkstream (width × height × 3) in
    for x = 0 to width × height − 1 do
      match Hashtbl.find table (sget s x) with
      | [r; g; b] →
          sset s' (x × 3) r;
          sset s' (x × 3 + 1) g;
          sset s' (x × 3 + 2) b
      | _ → failwith "read_8bpp_indexed_as_rgb24"
    done;
    s'

let read_8bpp_cmyk_indexed_as_rgb24 table width height s =
  let s' = mkstream (width × height × 3) in
    for x = 0 to width × height − 1 do
      match Hashtbl.find table (sget s x) with
      | [c; m; y; k] →
          let r, g, b = rgb_of_cmyk c m y k in
            sset s' (x × 3) r;
            sset s' (x × 3 + 1) g;
            sset s' (x × 3 + 2) b
      | _ → failwith "read_8bpp_indexed_as_rgb24"
    done;
    s'

let read_4bpp_indexed_as_rgb24 table width height s =
  let s' = mkstream (width × height × 3) in
    let posin = ref 0
    and posout = ref 0 in
      for row = 0 to height − 1 do
        for byte = 0 to (width + 1) / 2 − 1 do
          let p1 = sget s !posin lsr 4
          and p2 = sget s !posin land 15 in
            begin match Hashtbl.find table p1 with
            | [r1; g1; b1] →
                sset s' !posout r1; incr posout;
                sset s' !posout g1; incr posout;
                sset s' !posout b1; incr posout;
            | _ → failwith "read_4bpp_indexed_as_rgb24"
            end;
            begin
              if ¬ (odd width ∧ byte = (width + 1) / 2 − 1) then
              match Hashtbl.find table p2 with
              | [r2; g2; b2] →
                  sset s' !posout r2; incr posout;
                  sset s' !posout g2; incr posout;
```

```
            sset s' !posout b2; incr posout;
          | _ → failwith "read_4bpp_indexed_as_rgb24"
        end;
        incr posin
      done
    done;
    s'
```

```
let read_4bpp_cmyk_indexed_as_rgb24 table width height s =
  let s' = mkstream (width × height × 3) in
    let posin = ref 0
    and posout = ref 0 in
      for row = 0 to height − 1 do
        for byte = 0 to (width + 1) / 2 − 1 do
          let p1 = sget s !posin lsr 4
          and p2 = sget s !posin land 15 in
            begin match Hashtbl.find table p1 with
            | [c; m; y; k] →
                let r1, g1, b1 = rgb_of_cmyk c m y k in
                  sset s' !posout r1; incr posout;
                  sset s' !posout g1; incr posout;
                  sset s' !posout b1; incr posout;
            | _ → failwith "read_4bpp_cmyk_indexed_as_rgb24"
            end;
            begin
              if ¬ (odd width ∧ byte = (width + 1) / 2 − 1) then
              match Hashtbl.find table p2 with
              | [c; m; y; k] →
                  let r2, g2, b2 = rgb_of_cmyk c m y k in
                    sset s' !posout r2; incr posout;
                    sset s' !posout g2; incr posout;
                    sset s' !posout b2; incr posout;
              | _ → failwith "read_4bpp_cmyk_indexed_as_rgb24"
            end;
            incr posin
        done
      done;
      s'
```

Separation, CMYK alternate, tint transform function.

```
let read_separation_cmyk_as_rgb24 f width height s =
  let s' = mkstream (width × height × 3) in
    for p = 0 to width × height − 1 do
      let v = sget s p in
        match Pdffun.eval_function f [float v /. 255.] with
        | [c; y; m; k] →
            let c = toint (c *. 255.)
            and m = toint (m *. 255.)
            and y = toint (y *. 255.)
            and k = toint (k *. 255.) in
              let r, g, b = rgb_of_cmyk c m y k in
```

```
              sset s' (p × 3) r;
              sset s' (p × 3 + 1) g;
              sset s' (p × 3 + 2) b;
        | _ →
            raise (Pdf.PDFError "Bad tint transform function")
    done;
    s'

let rec read_raw_image size colspace bpc pdf resources width height dict data =
  match size, colspace, bpc with
  | size, (Pdfspace.DeviceRGB | Pdfspace.CalRGB _), Some (Pdf.Integer 8)
      when size ≥ width × height × 3 →
        Raw (width, height, BPP24, data)
  | size, Pdfspace.DeviceCMYK, Some (Pdf.Integer 8)
      when size ≥ width × height × 4 →
        Raw (width, height, BPP24, read_cmyk_8bpp_as_rgb24 width height data)
  | size, (Pdfspace.DeviceGray | Pdfspace.CalGray _), Some (Pdf.Integer 8)
      when size ≥ width × height →
        Raw (width, height, BPP24, read_gray_8bpp_as_rgb24 width height data)
  | size, _, Some (Pdf.Integer 1)
      when size ≥ width × height / 8 →
        Raw (width, height, BPP24, read_1bpp_as_rgb24 width height data)
  | size, Pdfspace.DeviceGray, Some (Pdf.Integer 4)
      when size ≥ width × height / 2 →
        Raw (width, height, BPP24, read_4bpp_gray_as_rgb24 width height data)
  | size, Pdfspace.Indexed ((Pdfspace.DeviceRGB | Pdfspace.CalRGB _), table), Some (Pdf.Integer 8)
  | size,
    Pdfspace.Indexed
      ((Pdfspace.DeviceN (_, (Pdfspace.DeviceRGB | Pdfspace.CalRGB _), _, _) |

       Pdfspace.ICCBased {Pdfspace.icc_alternate = (Pdfspace.DeviceRGB |
       Pdfspace.CalRGB _)}) , table),
    Some (Pdf.Integer 8)
      when size ≥ width × height →
        Raw (width, height, BPP24, read_8bpp_indexed_as_rgb24 table width height data)
  | size, Pdfspace.Indexed (Pdfspace.DeviceCMYK, table), Some (Pdf.Integer 8)
      when size ≥ width × height →
        Raw (width, height, BPP24, read_8bpp_cmyk_indexed_as_rgb24 table width height data)
  | size, Pdfspace.Indexed ((Pdfspace.DeviceRGB | Pdfspace.CalRGB _), table), Some (Pdf.Integer 4)
  | size, Pdfspace.Indexed (Pdfspace.ICCBased {Pdfspace.icc_alternate = (Pdfspace.DeviceRGB |
Pdfspace.CalRGB _)}, table), Some (Pdf.Integer 4)
      when size ≥ width × height / 2 →
        Raw (width, height, BPP24, read_4bpp_indexed_as_rgb24 table width height data)
  | size, Pdfspace.Indexed ((Pdfspace.DeviceCMYK), table), Some (Pdf.Integer 4)
  | size, Pdfspace.Indexed (Pdfspace.ICCBased {Pdfspace.icc_alternate = (Pdfspace.DeviceCMYK)}, ta
      when size ≥ width × height / 2 →
        Raw (width, height, BPP24, read_4bpp_cmyk_indexed_as_rgb24 table width height data)
  | size, Pdfspace.Separation (_, Pdfspace.DeviceCMYK, fn), Some (Pdf.Integer 8)
      when size ≥ width × height →
          Raw (width, height, BPP24, read_separation_cmyk_as_rgb24 fn width height data)
```

```
  | size, Pdfspace.ICCBased {Pdfspace.icc_alternate = cs}, _ →
      read_raw_image size cs bpc pdf resources width height dict data
  | size, cs, bpc →
     Printf.eprintf "NO IMAGE:\n size:%i\n cspace\n%s\n bpc\n%s\n width
%i\n
         height %i\n" size
     (Pdfspace.string_of_colourspace cs)
     (match bpc with None → "NONE" | Some bpc → Pdfwrite.string_of_pdf bpc)
     width
     height;
     raise (Pdf.PDFError "No image\n")

let rec get_raw_image pdf resources width height dict data =
  try
  let size =
    stream_size data
  and colspace =
    (* If an image mask, it's /DeviceGray, effectively *)
    match Pdf.lookup_direct_orelse pdf "/ImageMask" "/IM" dict with
    | Some (Pdf.Boolean true) → Pdfspace.DeviceGray
    | _ →
      let colspace =
        Pdf.lookup_direct_orelse pdf "/ColorSpace" "/CS" dict
      in
        let space =
          match Pdf.lookup_direct pdf "/ColorSpace" resources, colspace with
          | Some (Pdf.Dictionary _ as d), Some (Pdf.Name c) →
              begin match Pdf.lookup_direct pdf c d with
              | Some colspace → colspace
              | _ → (Pdf.Name c)
              end
          | _ →
              match colspace with
              | Some c → c
              | _ → failwith "PDf image:  no cololurspace"
        in
          Pdfspace.read_colourspace pdf resources space
  and bpc =
    Pdf.lookup_direct_orelse pdf "/BitsPerComponent" "/BPC" dict
  in
        read_raw_image size colspace bpc pdf resources width height dict data
  with
    e →
             raise e

let get_image_24bpp pdf resources stream =
  let streamdict, data =
    Pdf.getstream stream;
    match stream with
    | Pdf.Stream {contents = (s, Pdf.Got d)} →
        s, d
```

```
    | _ → raise (Assert_failure ("", 0, 0)) (* Pdf.getstream would have failed
*)
  in
    let width =
      match (Pdf.lookup_direct_orelse pdf "/Width" "/W" streamdict) with
      | Some (Pdf.Integer x) → x
      | _ → raise (Pdfread.PDFSemanticError "Malformed /Image width")
    and height =
      match (Pdf.lookup_direct_orelse pdf "/Height" "/H" streamdict) with
      | Some (Pdf.Integer x) → x
      | _ → raise (Pdfread.PDFSemanticError "Malformed /Image height")
    in
      decode_to_image pdf stream;
      match stream with
      | Pdf.Stream {contents = (Pdf.Dictionary d) as dict, Pdf.Got s} →
          begin match Pdf.lookup_direct_orelse pdf "/Filter" "/F" dict with
          | None | Some (Pdf.Array []) →
              let raw = get_raw_image pdf resources width height dict s
              and decode_entry = Pdf.lookup_direct_orelse pdf "/Decode" "/D" dict in
                decode decode_entry raw;
                raw
          | Some (Pdf.Name ("/DCTDecode" | "/DCT"))
          | Some (Pdf.Array [Pdf.Name ("/DCTDecode" | "/DCT")]) → JPEG s
          | Some (Pdf.Name "/JBIG2Decode")
          | Some (Pdf.Array [Pdf.Name "/JBIG2Decode"]) → JBIG2 s
          | Some (Pdf.Name "/JPXDecode")
          | Some (Pdf.Array [Pdf.Name "/JPXDecode"]) → JPEG2000 s
          | _ → raise (Pdf.PDFError "decode_to_image")
          end
      | _ → raise (Assert_failure ("", 0, 0))
```

13 Module PDFText
Reading and writing text

open Utility

13.1 Data type for fonts

Type 3 Specific Glyph Data

type $type3\_glpyhs$ =
  {$fontbbox$: *float* × *float* × *float* × *float*;
  $fontmatrix$: Transform.$transform\_matrix$;
  $charprocs$: (*string* × Pdf.$pdfobject$) *list*;
  $type3\_resources$: Pdf.$pdfobject$}

A font is either one of the standard 14 fonts, a simple font, or..

type $simple\_fonttype$ =
  | Type1
  | MMType1
  | Type3 **of** $type3\_glpyhs$
  | Truetype

type $fontmetrics$ = *float array* (∗ widths of glyphs 0..255 ∗)

type $fontfile$ =
  | FontFile **of** *int*
  | FontFile2 **of** *int*
  | FontFile3 **of** *int*

The fontfile is an indirect reference into the document, rather than a PDFobject itself. This preserves polymorphic equality (a pdfobject can contain functional values

type $fontdescriptor$ =
  {$ascent$: *float*;
  $descent$: *float*;
  $leading$: *float*;
  $avgwidth$: *float*;
  $maxwidth$: *float*;
  $fontfile$: $fontfile$ *option*}

```
type differences = (string × int) list

type encoding =
  | ImplicitInFontFile
  | StandardEncoding
  | MacRomanEncoding
  | WinAnsiEncoding
  | MacExpertEncoding
  | CustomEncoding of encoding × differences
  | FillUndefinedWithStandard of encoding

type simple_font =
  {fonttype : simple_fonttype;
   basefont : string;
   fontmetrics : fontmetrics option;
   fontdescriptor : fontdescriptor option;
   encoding : encoding}

type standard_font =
  | TimesRoman
  | TimesBold
  | TimesItalic
  | TimesBoldItalic
  | Helvetica
  | HelveticaBold
  | HelveticaOblique
  | HelveticaBoldOblique
  | Courier
  | CourierBold
  | CourierOblique
  | CourierBoldOblique
  | Symbol
  | ZapfDingbats

let string_of_standard_font = function
  | TimesRoman → "Times-Roman"
  | TimesBold → "Times-Bold"
  | TimesItalic → "Times-Italic"
  | TimesBoldItalic → "Times-BoldItalic"
  | Helvetica → "Helvetica"
  | HelveticaBold → "Helvetica-Bold"
  | HelveticaOblique → "Helvetica-Oblique"
  | HelveticaBoldOblique → "Helvetica-BoldOblique"
  | Courier → "Courier"
  | CourierBold → "Courier-Bold"
  | CourierOblique → "Courier-Oblique"
  | CourierBoldOblique → "Courier-BoldOblique"
  | Symbol → "Symbol"
  | ZapfDingbats → "ZapfDingbats"
```

```
type cid_system_info =
  {registry : string;
   ordering : string;
   supplement : int}

type composite_CIDfont =
  {cid_system_info : cid_system_info;
   cid_basefont : string;
   cid_fontdescriptor : fontdescriptor;
   cid_widths : (int × float) list;
   cid_default_width : int}

type cmap_encoding =
  | Predefined of string
  | CMap of int (* indirect reference to CMap stream *)

type font =
  | StandardFont of standard_font × encoding
  | SimpleFont of simple_font
  | CIDKeyedFont of string × composite_CIDfont × cmap_encoding (* string
is top-level basefont *)

let read_type3_data pdf font =
  {fontbbox =
     (let obj = Pdf.lookup_fail "No fontbbox" pdf "/FontBBox" font in
        Pdf.parse_rectangle obj);
   fontmatrix =
     Pdf.parse_matrix pdf "/FontMatrix" font;
   charprocs =
     (match Pdf.lookup_fail "Bad Charprocs" pdf "/CharProcs" font with
      | Pdf.Dictionary l → l
      | _ → raise (Pdfread.PDFSemanticError "Bad charprocs"));
   type3_resources =
     (match Pdf.lookup_direct pdf "/Resources" font with
      | None → Pdf.Dictionary []
      | Some d → d)}

let simple_fonttype_of_string pdf font = function
  | "/Type1" → Some Type1
  | "/MMType1" → Some MMType1
  | "/Type3" →
      Some (Type3 (read_type3_data pdf font))
  | "/TrueType" → Some Truetype
  | _ → None

let read_basefont pdf font =
  match Pdf.lookup_direct pdf "/BaseFont" font with
  | Some (Pdf.Name n) → n
  | _ → ""
```

```
let read_fontdescriptor pdf font =
  match Pdf.lookup_direct pdf "/FontDescriptor" font with
  | None → None
  | Some fontdescriptor →
      let ascent =
        match Pdf.lookup_direct pdf "/Ascent" fontdescriptor with
        | Some x → Pdf.getnum x
        | None → 0.
      and descent =
        match Pdf.lookup_direct pdf "/Descent" fontdescriptor with
        | Some x → Pdf.getnum x
        | None → 0.
      and leading =
        match Pdf.lookup_direct pdf "/Leading" fontdescriptor with
        | Some x → Pdf.getnum x
        | None → 0.
      and avgwidth =
        match Pdf.lookup_direct pdf "/AvgWidth" fontdescriptor with
        | Some x → Pdf.getnum x
        | None → 0.
      and maxwidth =
        match Pdf.lookup_direct pdf "/MaxWidth" fontdescriptor with
        | Some x → Pdf.getnum x
        | None → 0.
      and fontfile =
        match Pdf.find_indirect "/FontFile" fontdescriptor with
        | Some i → Some (FontFile i)
        | None →
            match Pdf.find_indirect "/FontFile2" fontdescriptor with
            | Some i → Some (FontFile2 i)
            | None →
                match Pdf.find_indirect "/FontFile3" fontdescriptor with
                | Some i → Some (FontFile3 i)
                | None → None
      in
        Some
          {ascent = ascent;
           descent = descent;
           leading = leading;
           avgwidth = avgwidth;
           maxwidth = maxwidth;
           fontfile = fontfile}
```

Read the widths from a font. Normally in the font descriptor, but in Type3 fonts at the top level.

```
let read_metrics pdf font =
  let fontdescriptor =
    match Pdf.lookup_direct pdf "/Subtype" font with
    | Some (Pdf.Name "/Type3") → Some font
    | _ → Pdf.lookup_direct pdf "/FontDescriptor" font
```

```
in
  match fontdescriptor with
  | None → None
  | Some fontdescriptor →
      let firstchar =
        match Pdf.lookup_direct pdf "/FirstChar" font with
        | Some (Pdf.Integer i) →
            if i ≤ 255 ∧ i ≥ 0 then i else
              raise (Pdf.PDFError "Bad /Firstchar")
        | _ → raise (Pdf.PDFError "No /FirstChar")
      and lastchar =
        match Pdf.lookup_direct pdf "/LastChar" font with
        | Some (Pdf.Integer i) →
            if i ≤ 255 ∧ i ≥ 0 then i else
              raise (Pdf.PDFError "Bad /Lastchar")
        | _ → raise (Pdf.PDFError "No /LastChar")
      and missingwidth =
        match Pdf.lookup_direct pdf "/MissingWidth" fontdescriptor with
        | Some (Pdf.Integer w) → float w
        | Some (Pdf.Real w) → w
        | _ → 0.
      in
        let elts =
          match Pdf.lookup_direct pdf "/Widths" font with
          | Some (Pdf.Array elts) → elts
          | _ → raise (Pdf.PDFError "No /Widths")
        in
          if length elts ≠ lastchar − firstchar + 1
            then raise (Pdf.PDFError "Bad /Widths")
            else
              let before =
                many missingwidth firstchar
              and given =
                map
                  (fun elt →
                     match Pdf.direct pdf elt with
                     | Pdf.Integer i → float i
                     | Pdf.Real f → f
                     | _ → raise (Pdf.PDFError "Bad /Width entry"))
                  elts
              and after =
                many missingwidth (255 − lastchar)
              in
                Some (Array.of_list (before @ given @ after))
```

Parse a /Differences entry to get a list of (name, number) pairs

```
let pairs_of_differences pdf differences =
  let rec groups_of_differences prev elts =
    match elts with
    | [] → prev
```

```
      | Pdf.Integer n :: rest →
          let stripname = function Pdf.Name n → n | _ → raise (Assert_failure ("", 0, 0)) in
            let names, more =
              cleavewhile (function Pdf.Name _ → true | _ → false) rest
            in
              groups_of_differences ((n, map stripname names) :: prev) more
      | _ → raise (Pdf.PDFError "Malformed /Differences")
    and mappings_of_group (x, es) =
      let additions = ilist 0 (length es − 1) in
        map2 (fun e a → (x + a, e)) es additions
    in
      match differences with
      | Pdf.Array elts →
          let direct_elements = map (Pdf.direct pdf) elts in
            let groups = groups_of_differences [] direct_elements in
              map
                (fun (k, v) → (v, k))
                (flatten (map mappings_of_group groups))
      | _ → raise (Pdf.PDFError "Bad /Differences")

let standard_font_of_name = function
  | "/Times-Roman" | "/TimesNewRoman" →
      Some TimesRoman
  | "/Times-Bold" | "/TimesNewRoman,Bold" →
      Some TimesBold
  | "/Times-Italic" | "/TimesNewRoman,Italic" →
      Some TimesItalic
  | "/Times-BoldItalic" | "/TimesNewRoman,BoldItalic" →
      Some TimesBoldItalic
  | "/Helvetica" | "/Arial" →
      Some Helvetica
  | "/Helvetica-Bold" | "/Arial,Bold" →
      Some HelveticaBold
  | "/Helvetica-Oblique" | "/Arial,Italic" →
      Some HelveticaOblique
  | "/Helvetica-BoldOblique" | "/Arial,BoldItalic" →
      Some HelveticaBoldOblique
  | "/Courier" | "/CourierNew" →
      Some Courier
  | "/CourierBold" | "/CourierNew,Bold" →
      Some CourierBold
  | "/Courier-Oblique" | "/CourierNew,Italic" →
      Some CourierOblique
  | "/Courier-BoldOblique" | "/CourierNew,BoldItalic" →
      Some CourierBoldOblique
  | "/Symbol" →
      Some Symbol
  | "/ZapfDingbats" →
      Some ZapfDingbats
  | _ →
```

```
    None
```

Predicate: is it a standard 14 font? If it's been overriden (contains widths etc, we treat it as a simple font.

```
let is_standard14font pdf font =
  match Pdf.lookup_direct pdf "/Subtype" font with
  | Some (Pdf.Name "/Type1") →
      begin match Pdf.lookup_direct pdf "/BaseFont" font with
      | Some (Pdf.Name name) →
          begin match standard_font_of_name name with
          | None → false
          | Some _ →
              (* Check to see if it's been overriden *)
              match Pdf.lookup_direct pdf "/Widths" font with
              | None → true
              | _ → false
          end
      | _ → false
      end
  | _ → false
```

Is a font embedded in the document?

```
let is_embedded pdf font =
  match Pdf.lookup_direct pdf "/FontDescriptor" font with
  | None → false
  | Some fontdescriptor →
      match
        Pdf.lookup_direct_orelse pdf "/FontFile" "/FontFile2" fontdescriptor
      with
      | Some _ → true
      | None →
          match Pdf.lookup_direct pdf "/FontFile3" fontdescriptor with
          | Some _ → true
          | None → false
```

Is a font symbolic? (Doesn't deal with standard 14 Zapf and Symbol)

```
let is_symbolic pdf font =
  match Pdf.lookup_direct pdf "/FontDescriptor" font with
  | None → false
  | Some fontdescriptor →
      match Pdf.lookup_direct pdf "/Flags" fontdescriptor with
      | Some (Pdf.Integer flags) → flags land (1 lsl 3) > 0
      | _ → raise (Pdf.PDFError "No /Flags in font descriptor")
```

For now, not for truetype fonts: add pg 399-401 later. Need to clarify what happens if a standard-14 font is overriden.

```
let read_encoding pdf font =
  match Pdf.lookup_direct pdf "/Encoding" font with
  | Some (Pdf.Name "/MacRomanEncoding") → MacRomanEncoding
  | Some (Pdf.Name "/MacExpertEncoding") → MacExpertEncoding
```

```
    | Some (Pdf.Name "/WinAnsiEncoding") → WinAnsiEncoding
    | Some (Pdf.Dictionary _ as encdict) →
        begin match Pdf.lookup_direct pdf "/Subtype" font with
        | Some
            (Pdf.Name (("/Type1" | "/MMType1" | "/Type3" | "/TrueType") as fonttype))
          →
            let encoding =
              let base_encoding =
                match Pdf.lookup_direct pdf "/BaseEncoding" encdict with
                | Some (Pdf.Name "/MacRomanEncoding") → MacRomanEncoding
                | Some (Pdf.Name "/MacExpertEncoding") → MacExpertEncoding
                | Some (Pdf.Name "/WinAnsiEncoding") → WinAnsiEncoding
                | None →
                    if is_embedded pdf font
                    then ImplicitInFontFile
                      else if is_symbolic pdf font
                        then ImplicitInFontFile
                        else StandardEncoding
                | _ → raise (Pdf.PDFError "unknown /BaseEncoding")
              in
                begin match Pdf.lookup_direct pdf "/Differences" encdict with
                | Some differences →
                    CustomEncoding
                      (base_encoding, pairs_of_differences pdf differences)
                | _ → base_encoding
                end
            in
              if fonttype = "/Truetype"
                then FillUndefinedWithStandard encoding
                else encoding
        | _ → raise (Pdf.PDFError "Bad font /Subtype")
        end
    | _ → ImplicitInFontFile

let read_simple_font pdf font =
  match Pdf.lookup_direct pdf "/Subtype" font with
  | Some (Pdf.Name n) →
      begin match simple_fonttype_of_string pdf font n with
      | Some fonttype →
          let fontdescriptor = read_fontdescriptor pdf font in
            SimpleFont
              {fonttype = fonttype;
               basefont = read_basefont pdf font;
               fontmetrics = read_metrics pdf font;
               fontdescriptor = fontdescriptor;
               encoding = read_encoding pdf font}
      | None → raise (Pdf.PDFError "Not a simple font")
      end
  | _ → raise (Pdf.PDFError "No font /Subtype")
```

Read a base 14 font

```
let read_standard14font pdf font =
  match Pdf.lookup_direct pdf "/BaseFont" font with
  | Some (Pdf.Name name) →
      begin match standard_font_of_name name with
      | None → raise (Pdf.PDFError "Not a base 14 font")
      | Some f → StandardFont (f, read_encoding pdf font)
      end
  | _ → raise (Pdf.PDFError "Bad base 14 font")
```

Predicate: is it a simple font, assuming it's not a standard 14 font.

```
let is_simple_font pdf font =
  match Pdf.lookup_direct pdf "/Subtype" font with
  | Some (Pdf.Name ("/Type1" | "/MMType1" | "/Type3" | "/TrueType")) →
 true
  | _ → false
```

Predicate: is it a CIDKeyed font?

```
let is_cidkeyed_font pdf font =
  match Pdf.lookup_direct pdf "/Subtype" font with
  | Some (Pdf.Name "/Type0") → true
  | _ → false
```

Read a CID system info dictionary

```
let read_cid_system_info pdf dict =
  {registry =
     begin match Pdf.lookup_direct pdf "/Registry" dict with
     | Some (Pdf.String s) → s
     | _ → raise (Pdf.PDFError "No /Registry")
     end;
   ordering =
     begin match Pdf.lookup_direct pdf "/Ordering" dict with
     | Some (Pdf.String s) → s
     | _ → raise (Pdf.PDFError "No /Ordering")
     end;
   supplement =
     begin match Pdf.lookup_direct pdf "/Supplement" dict with
     | Some (Pdf.Integer i) → i
     | _ → raise (Pdf.PDFError "No /Supplement")
     end}
```

This returns the explicit pairs, which need to be combined with the default value to look a width up.

```
let rec read_cid_widths = function
  | Pdf.Integer c :: Pdf.Array ws :: more →
      let nums =
        map
          (function
           | Pdf.Integer i → float i
           | Pdf.Real r → r
           | _ → raise (Pdf.PDFError "Bad /W array"))
```

```
        ws
      in
        combine (indxn c nums) nums @ read_cid_widths more
  | Pdf.Integer c_first :: Pdf.Integer c_last :: w :: more →
      let w =
        match w with
        | Pdf.Integer i → float i
        | Pdf.Real r → r
        | _ → raise (Pdf.PDFError "Bad /W array")
      in
        if c_last ≤ c_first
          then raise (Pdf.PDFError "Bad /W array")
          else
            let pairs =
              combine
                (ilist c_first c_last)
                (many w (c_last − c_first + 1))
            in
              pairs @ read_cid_widths more
  | [] → []
  | _ → raise (Pdf.PDFError "Malformed /W in CIDfont")
```

Read a composite CID font
FIXME: Doesn't support vertical modes (DW2 / W2)

```
let read_descendant pdf dict =
  let cid_system_info =
    match Pdf.lookup_direct pdf "/CIDSystemInfo" dict with
    | Some cid_dict → read_cid_system_info pdf cid_dict
    | None → raise (Pdf.PDFError "No CIDSystemInfo")
  and cid_basefont =
    match Pdf.lookup_direct pdf "/BaseFont" dict with
    | Some (Pdf.Name n) → n
    | _ → raise (Pdf.PDFError "No /BaseFont")
  and cid_fontdescriptor =
    match read_fontdescriptor pdf dict with
    | Some f → f
    | None → raise (Pdf.PDFError "No FontDescriptor in CIDkeyed font")
  and cid_widths =
    match Pdf.lookup_direct pdf "/W" dict with
    | Some (Pdf.Array ws) → read_cid_widths ws
    | _ → []
  and default_width =
    match Pdf.lookup_direct pdf "/DW" dict with
    | Some (Pdf.Integer d) → d
    | _ → 1000
  in
    {cid_system_info = cid_system_info;
     cid_basefont = cid_basefont;
     cid_fontdescriptor = cid_fontdescriptor;
     cid_widths = cid_widths;
```

```
    cid_default_width = default_width}
```

Read a CIDKeyed (Type 0) font

```
let read_cidkeyed_font pdf font =
  let basefont =
    match Pdf.lookup_direct pdf "/BaseFont" font with
    | Some (Pdf.Name b) → b
    | _ → raise (Pdf.PDFError "Bad /BaseFont")
  and composite_CIDfont =
    match Pdf.lookup_direct pdf "/DescendantFonts" font with
    | Some (Pdf.Array [e]) →
        read_descendant pdf (Pdf.direct pdf e)
    | _ → raise (Pdf.PDFError "Bad descendant font")
  and encoding =
    match Pdf.lookup_direct pdf "/Encoding" font with
    | Some (Pdf.Name e) → Predefined e
    | Some (Pdf.Stream _) →
        begin match Pdf.find_indirect "/Encoding" font with
        | Some n → CMap n
        | None → raise (Pdf.PDFError "malformed /Encoding")
        end
    | _ → raise (Pdf.PDFError "malformed or missing /Encoding")
  in
    CIDKeyedFont (basefont, composite_CIDfont, encoding)
```

Reads a font

```
let read_font pdf font =
    if is_standard14font pdf font
    then read_standard14font pdf font
    else if is_simple_font pdf font
      then read_simple_font pdf font
      else if is_cidkeyed_font pdf font
        then read_cidkeyed_font pdf font
        else raise (Pdf.PDFError "Unknown font type")
```

13.2 Font encodings

Standard encoding

```
let name_to_standard =
  ["/A", 101₈; "/AE", 341₈; "/B", 102₈; "/C", 103₈; "/D", 104₈; "/E",
  105₈; "/F", 106₈; "/G", 107₈; "/H", 110₈; "/I", 111₈; "/J", 112₈; "/K",
  113₈; "/L", 114₈; "/Lslash", 350₈; "/M", 115₈; "/N", 116₈; "/O", 117₈;
  "/OE", 352₈; "/Oslash", 351₈; "/P", 120₈; "/Q", 121₈; "/R", 122₈; "/S",
  123₈; "/T", 124₈; "/U", 125₈; "/V", 126₈; "/W", 127₈; "/X", 130₈; "/Y",
  131₈; "/Z", 132₈; "/a", 141₈; "/acute", 302₈; "/ae", 361₈; "/ampersand",
  046₈; "/asciicircum", 136₈; "/asciitilde", 176₈; "/asterisk", 052₈; "/at",
  100₈; "/b", 142₈; "/backslash", 134₈; "/bar", 174₈; "/braceleft", 173₈;
  "/braceright", 175₈; "/bracketleft", 133₈; "/bracketright", 135₈; "/breve",
```

306_8; "/bullet", 267_8; "/c", 143_8; "/caron", 317_8; "/cedilla", 313_8; "/cent", 242_8; "/circumflex", 303_8; "/colon", 072_8; "/comma", 054_8; "/currency", 250_8; "/d", 144_8; "/dagger", 262_8; "/daggerdbl", 263_8; "/dieresis", 310_8; "/dollar", 044_8; "/dotaccent", 307_8; "/dottlessi", 365_8; "/e", 145_8; "/eight", 070_8; "/ellipsis", 274_8; "/emdash", 320_8; "/endash", 261_8; "/equal", 075_8; "/exclam", 041_8; "/exclamdown", 241_8; "/f", 146_8; "/fi", 256_8; "/five", 065_8; "/fl", 257_8; "/florin", 246_8; "/four", 064_8; "/fraction", 244_8; "/g", 147_8; "/germandbls", 373_8; "/grave", 301_8; "/greater", 076_8; "/guillemotleft", 253_8; "/guillemotright", 273_8; "/guilsinglleft", 254_8; "/guilsinglright", 255_8; "/h", 150_8; "/hungarumlaut", 315_8; "/hyphen", 055_8; "/i", 151_8; "/j", 152_8; "/k", 153_8; "/l", 154_8; "/less", 074_8; "/lslash", 370_8; "/m", 155_8; "/macron", 305_8; "/n", 156_8; "/nine", 071_8; "/numbersign", 043_8; "/o", 157_8; "/oe", 372_8; "/ogonek", 316_8; "/one", 061_8; "/ordfeminine", 343_8; "/ordmasculine", 353_8; "/oslash", 361_8; "/p", 160_8; "/paragraph", 266_8; "/parenleft", 050_8; "/parenright", 051_8; "/percent", 045_8; "/period", 056_8; "/periodcentered", 264_8; "/perthousand", 275_8; "/plus", 053_8; "/q", 161_8; "/question", 077_8; "/questiondown", 277_8; "/quotedbl", 042_8; "/quotedblbase", 271_8; "/quotedblleft", 252_8; "/quotedblright", 272_8; "/quoteleft", 140_8; "/quoteright", 047_8; "/quotesinglbase", 270_8; "/quotesingle", 251_8; "/r", 162_8; "/ring", 312_8; "/s", 163_8; "/section", 247_8; "/semicolon", 073_8; "/seven", 067_8; "/six", 066_8; "/slash", 057_8; "/space", 040_8; "/sterling", 243_8; "/t", 164_8; "/three", 063_8; "/tilde", 304_8; "/two", 062_8; "/u", 165_8; "/underscore", 137_8; "/v", 166_8; "/w", 167_8; "/x", 170_8; "/y", 171_8; "/yen", 245_8; "/z", 172_8; "/zero", 060_8]

Mac Roman Encoding

let *name_to_macroman* =
["/A", 101_8; "/AE", 256_8; "/Aacute", 347_8; "/Acircumflex", 345_8; "/Adieresis", 200_8; "/Agrave", 313_8; "/Aring", 201_8; "/Atilde", 314_8; "/B", 102_8; "/C", 103_8; "/Ccedilla", 202_8; "/D", 104_8; "/E", 105_8; "/Eacute", 203_8; "/Ecircumflex", 346_8; "/Edieresis", 350_8; "/Egrave", 351_8; "/F", 106_8; "/G", 107_8; "/H", 110_8; "/I", 111_8; "/Iacute", 352_8; "/Icircumflex", 353_8; "/Idieresis", 354_8; "/Igrave", 355_8; "/J", 112_8; "/K", 113_8; "/L", 114_8; "/M", 115_8; "/N", 116_8; "/Ntilde", 204_8; "/O", 117_8; "/OE", 316_8; "/Oacute", 356_8; "/Ocircumflex", 357_8; "/Odieresis", 205_8; "/Ograve", 361_8; "/Oslash", 257_8; "/Otilde", 315_8; "/P", 120_8; "/Q", 121_8; "/R", 122_8; "/S", 123_8; "/T", 124_8; "/U", 125_8; "/Uacute", 362_8; "/Ucircumflex", 363_8; "/Udieresis", 206_8; "/Ugrave", 364_8; "/V", 126_8; "/W", 127_8; "/X", 130_8; "/Y", 131_8; "/Ydieresis", 331_8; "/Z", 132_8; "/a", 141_8; "/aacute", 207_8; "/acircumflex", 211_8; "/acute", 253_8; "/adieresis", 212_8; "/ae", 276_8; "/agrave", 210_8; "/ampersand", 046_8; "/aring", 214_8; "/asciicircum", 136_8; "/asciitilde", 176_8; "/asterisk", 052_8; "/at", 100_8; "/atilde", 213_8; "/b", 142_8; "/backslash", 134_8; "/bar", 174_8; "/braceleft", 173_8; "/braceright", 175_8; "/bracketleft", 133_8; "/bracketright", 135_8; "/breve", 371_8; "/bullet", 245_8; "/c", 143_8; "/caron", 377_8; "/ccedilla", 215_8; "/cedilla", 374_8; "/cent", 242_8; "/circumflex", 366_8; "/colon", 072_8; "/comma", 054_8; "/copyright", 251_8; "/currency", 333_8; "/d", 144_8; "/dagger", 240_8; "/daggerdbl", 340_8; "/degree", 241_8; "/dieresis", 254_8; "/divide", 326_8; "/dollar", 044_8;

"/dotaccent", 372_8; "/dotlessi", 365_8; "/e", 145_8; "/eacute", 216_8; "/ecircumflex", 220_8; "/edieresis", 221_8; "/egrave", 217_8; "/eight", 070_8; "/ellipsis", 311_8; "/emdash", 321_8; "/endash", 320_8; "/equal", 075_8; "/exclam", 041_8; "/exclamdown", 301_8; "/f", 146_8; "/fi", 336_8; "/five", 065_8; "/fl", 337_8; "/florin", 304_8; "/four", 064_8; "/fraction", 332_8; "/g", 147_8; "/germandbls", 247_8; "/grave", 140_8; "/greater", 076_8; "/guillemotleft", 307_8; "/guillemotright", 310_8; "/guilsinglleft", 334_8; "/guilsinglright", 335_8; "/h", 150_8; "/hungrumlaut", 375_8; "/hyphen", 055_8; "/i", 151_8; "/iacute", 222_8; "/icircumflex", 224_8; "/idieresis", 225_8; "/igrave", 223_8; "/j", 152_8; "/k", 153_8; "/l", 154_8; "/less", 074_8; "/logicalnot", 302_8; "/m", 155_8; "/macron", 370_8; "/mu", 265_8; "/n", 156_8; "/nine", 071_8; "/ntilde", 226_8; "/numbersign", 043_8; "/o", 157_8; "/oacute", 227_8; "/ocircumflex", 231_8; "/odieresis", 232_8; "/oe", 317_8; "/ogonek", 376_8; "/one", 061_8; "/ordfeminine", 273_8; "/ordmasculine", 274_8; "/oslash", 277_8; "/otilde", 233_8; "/p", 160_8; "/paragraph", 246_8; "/parenleft", 050_8; "/parenright", 051_8; "/percent", 045_8; "/period", 056_8; "/periodcentered", 341_8; "/perthousand", 344_8; "/plus", 053_8; "/plusminus", 261_8; "/q", 161_8; "/question", 077_8; "/questiondown", 300_8; "/quotedbl", 042_8; "/quotedblbase", 343_8; "/quotedblleft", 322_8; "/quotedblright", 323_8; "/quoteleft", 324_8; "/quoteright", 325_8; "/quotesinglbase", 342_8; "/quotesingle", 047_8; "/r", 162_8; "/registered", 250_8; "/ring", 373_8; "/s", 163_8; "/section", 244_8; "/semicolon", 073_8; "/seven", 067_8; "/six", 066_8; "/slash", 057_8; "/space", 040_8; "/sterling", 243_8; "/t", 164_8; "/three", 063_8; "/tilde", 367_8; "/trademark", 252_8; "/two", 062_8; "/u", 165_8; "/uacute", 234_8; "/ucircumflex", 236_8; "/udieresis", 237_8; "/ugrave", 235_8; "/underscore", 137_8; "/v", 166_8; "/w", 167_8; "/x", 170_8; "/y", 171_8; "/ydieresis", 330_8; "/yen", 264_8; "/z", 172_8; "/zero", 060_8; "/space", 312_8]

Win Ansi Encoding

let $name\_to\_win$ $=$
["/A", 101_8; "/AE", 306_8; "/Aacute", 301_8; "/Acircumflex", 302_8; "/Adieresis", 304_8; "/Agrave", 300_8; "/Aring", 305_8; "/Atilde", 303_8; "/B", 102_8; "/C", 103_8; "/Ccedilla", 307_8; "/D", 104_8; "/E", 105_8; "/Eacute", 311_8; "/Ecircumflex", 312_8; "/Edieresis", 313_8; "/Egrave", 310_8; "/Eth", 320_8; "/Euro", 200_8; "/F", 106_8; "/G", 107_8; "/H", 110_8; "/I", 111_8; "/Iacute", 315_8; "/Icircumflex", 316_8; "/Idieresis", 317_8; "/Igrave", 314_8; "/J", 112_8; "/K", 113_8; "/L", 114_8; "/M", 115_8; "/N", 116_8; "/Ntilde", 321_8; "/O", 117_8; "/OE", 214_8; "/Oacute", 323_8; "/Ocircumflex", 324_8; "/Odieresis", 326_8; "/Ograve", 322_8; "/Oslash", 330_8; "/Otilde", 325_8; "/P", 120_8; "/Q", 121_8; "/R", 122_8; "/S", 123_8; "/Scaron", 212_8; "/T", 124_8; "/Thorn", 336_8; "/U", 125_8; "/Uacute", 332_8; "/Ucircumflex", 333_8; "/Udieresis", 334_8; "/Ugrave", 331_8; "/V", 126_8; "/W", 127_8; "/X", 130_8; "/Y", 131_8; "/Yacute", 335_8; "/Ydieresis", 237_8; "/Z", 132_8; "/Zcaron", 216_8; "/a", 141_8; "/aacute", 341_8; "/acircumflex", 342_8; "/acute", 264_8; "/adieresis", 344_8; "/ae", 346_8; "/agrave", 340_8; "/ampersand", 046_8; "/aring", 345_8; "/asciicircum", 136_8; "/asciitilde", 176_8; "/asterisk", 052_8; "/at", 100_8; "/atilde", 343_8; "/b", 142_8; "/backslash", 134_8; "/bar", 174_8; "/braceleft", 173_8; "/braceright", 175_8; "/bracketleft", 133_8; "/bracketright", 135_8; "/brokenbar", 246_8; "/bullet",

225_8; "/c", 143_8; "/ccedilla", 347_8; "/cedilla", 270_8; "/cent", 242_8;
"/circumflex", 210_8; "/colon", 072_8; "/comma", 054_8; "/copyright", 251_8;
"/currency", 244_8; "/d", 144_8; "/dagger", 206_8; "/daggerdbl", 207_8;
"/degree", 260_8; "/dieresis", 250_8; "/divide", 367_8; "/dollar",044_8; "/e",
145_8; "/eacute", 351_8; "/ecircumflex", 352_8; "/edieresis", 353_8;
"/egrave", 350_8; "/eight", 070_8; "/ellipsis", 205_8; "/emdash", 227_8;
"/endash", 226_8; "/equal", 075_8; "/eth", 360_8; "/exclam", 041_8;
"/exclamdown", 241_8; "/f", 146_8; "/five", 065_8; "/florin", 203_8; "/four",
064_8; "/g", 147_8; "/germandbls", 337_8; "/grave", 140_8; "/greater", 076_8;
"/guillemotleft", 253_8; "/guillemotright", 273_8; "/guilsinglleft", 213_8;
"/guilsinglright", 233_8; "/h", 150_8; "/hyphen", 055_8; "/i", 151_8;
"/iacute", 355_8; "/icircumflex", 356_8; "/idieresis", 357_8; "/igrave",
354_8; "/j", 152_8; "/k", 153_8; "/l", 154_8; "/less", 074_8; "/logicalnot",
254_8; "/m", 155_8; "/macron", 257_8; "/mu", 265_8; "/multiply", 327_8; "/n",
156_8; "/nine", 071_8; "/ntilde", 361_8; "/numbersign", 043_8; "/o", 157_8;
"/oacute", 363_8; "/ocircumflex", 364_8; "/odieresis", 366_8; "/oe", 234_8;
"/ograve", 362_8; "/one", 061_8; "/onehalf", 275_8; "/onequarter", 274_8;
"/onesuperior", 271_8; "/ordfeminine", 252_8; "/ordmasculine", 272_8;
"/oslash", 370_8; "/otilde", 365_8; "/p", 160_8; "/paragraph", 266_8;
"/parenleft", 050_8; "/parenright", 051_8; "/percent", 045_8; "/period",
056_8; "/periodcentered", 267_8; "/perthousand", 211_8; "/plus", 053_8;
"/plusminus", 261_8; "/q", 161_8; "/question", 077_8; "/questiondown", 277_8;
"/quotedbl", 042_8; "/quotedblbase", 204_8; "/quotedblleft", 223_8;
"/quotedblright", 224_8; "/quoteleft", 221_8; "/quoteright", 222_8;
"/quotesinglbase", 202_8; "/quotesingle", 047_8; "/r", 162_8; "/registered",
256_8; "/s", 163_8; "/scaron", 232_8; "/section", 247_8; "/semicolon", 073_8;
"/seven", 067_8; "/six", 066_8; "/slash", 057_8; "/space", 040_8; "/sterling",
243_8; "/t", 164_8; "/thorn", 376_8; "/three", 063_8; "/threequarters", 276_8;
"/threesuperior", 263_8; "/tilde", 230_8; "/trademark", 231_8; "/two", 062_8;
"/twosuperior", 262_8; "/u", 165_8; "/uacute", 372_8; "/ucircumflex", 373_8;
"/udieresis", 374_8; "/ugrave", 371_8; "/underscore", 137_8; "/v", 166_8;
"/w", 167_8; "/x", 170_8; "/y", 171_8; "/yacute", 375_8; "/ydieresis", 377_8;
"/yen", 245_8; "/z", 172_8; "/zcaron", 236_8; "/zero", 060_8; "/space", 240_8;
"/hyphen", 255_8]

Mac Expert Encoding

let $name\_to\_macexpert$ =
["/AEsmall", 276_8; "/Aacutesmall", 207_8; "/Acircumflexsmall", 211_8;
"/Acutesmall", 047_8; "/Adieresissmall", 212_8; "/Agravesmall", 210_8;
"/Aringsmall", 214_8; "/Asmall", 141_8; "/Atildesmall", 213_8; "/Brevesmall",
363_8; "/Bsmall", 142_8; "/Caronsmall", 256_8; "/Ccedillasmall", 215_8;
"/Cedillasmall", 311_8; "/Circumflexsmall", 136_8; "/Csmall", 143_8;
"/Dieresissmall", 254_8; "/Dotaccentsmall", 372_8; "/Dsmall", 144_8;
"/Eacutesmall", 216_8; "/Ecircumflexsmall", 220_8; "/Edieresissmall", 221_8;
"/Egravesmall", 217_8; "/Esmall", 145_8; "/Ethsmall", 104_8; "/Fsmall", 146_8;
"/Gravesmall", 140_8; "/Gsmall", 147_8; "/Hsmall", 150_8;
"/Hungarumlautsmall", 042_8; "/Iacutesmall", 222_8; "/Icircumflexsmall",
224_8; "/Idieresissmall", 225_8; "/Igravesmall", 223_8; "/Ismall", 151_8;
"/Jsmall", 152_8; "/Ksmall", 153_8; "/Lslashsmall", 302_8; "/Lsmall", 154_8;
"/Macronsmall", 364_8; "/Msmall", 155_8; "/Nsmall", 156_8; "/Ntildesmall",

226_8; "/OEsmall", 317_8; "/Oacutesmall", 227_8; "/Ocircumflexsmall", 231_8; "/Odieresissmall", 232_8; "/Ogoneksmall", 362_8; "/Ogravesmall", 230_8; "/Oslashsmall", 277_8; "/Osmall", 157_8; "/Otildesmall", 233_8; "/Psmall", 160_8; "/Qsmall", 161_8; "/Ringsmall", 373_8; "/Rsmall", 162_8; "/Scaronsmall", 247_8; "/Ssmall", 163_8; "/Thornsmall", 271_8; "/Tildesmall", 176_8; "/Tsmall", 164_8; "/Uacutesmall", 234_8; "/Ucircumflexsmall", 236_8; "/Udieresissmall", 237_8; "/Ugravesmall", 235_8; "/Usmall", 165_8; "/Vsmall", 166_8; "/Wsmall", 167_8; "/Xsmall", 170_8; "/Yacutesmall", 264_8; "/Ydieresissmall", 330_8; "/Ysmall", 171_8; "/Zcaronsmall", 275_8; "/Zsmall", 172_8; "/ampersandsmall", 046_8; "/asuperior", 201_8; "/bsuperior", 365_8; "/centinferior", 251_8; "/centoldstyle", 043_8; "/centsuperior", 202_8; "/colon", 072_8; "/colonmonetary", 173_8; "/comma", 054_8; "/commainferior", 262_8; "/commasuperior", 370_8; "/dollarinferior", 266_8; "/dollaroldstyle", 044_8; "/dsuperior", 353_8; "/eightinferior", 245_8; "/eightoldstyle", 070_8; "/eightsuperior", 241_8; "/esuperior", 344_8; "/exclamdownsmall", 326_8; "/exclamsmall", 041_8; "/ff", 126_8; "/ffi", 131_8; "/ffl", 132_8; "/fi", 127_8; "/figuredash", 320_8; "/fiveeighths", 114_8; "/fiveinferior", 260_8; "/fiveoldstyle", 065_8; "/fivesuperior", 336_8; "/fl", 130_8; "/fourinferior", 242_8; "/fouroldstyle", 064_8; "/foursuperior", 335_8; "/fraction", 057_8; "/hyphen", 055_8; "/hypheninferior", 137_8; "/hyphensuperior", 137_8; "/isuperior", 351_8; "/lsuperior", 361_8; "/msuperior", 367_8; "/nineinferior", 273_8; "/nineoldstyle", 071_8; "/ninesuperior", 341_8; "/nsuperior", 366_8; "/onedotenleader", 053_8; "/oneeighth", 112_8; "/onefitted", 174_8; "/onehalf", 110_8; "/oneinferior", 301_8; "/oneoldstyle", 061_8; "/onequarter", 107_8; "/onesuperior", 332_8; "/onethird", 116_8; "/osuperior", 257_8; "/parenleftinferior", 133_8; "/parenleftsuperior", 050_8; "/parenrightinferior", 135_8; "/parenrightsuperior", 051_8; "/period", 056_8; "/periodinferior", 263_8; "/periodsuperior", 371_8; "/questiondownsmall", 300_8; "/questionsmall", 077_8; "/rsuperior", 345_8; "/rupiah", 175_8; "/semicolon", 073_8; "/seveneighths", 115_8; "/seveninferior", 246_8; "/sevenoldstyle", 067_8; "/sevensuperior", 340_8; "/sixinferior", 244_8; "/sixoldstyle", 066_8; "/sixsuperior", 337_8; "/space", 040_8; "/ssuperior", 352_8; "/threeeighths", 113_8; "/threeinferior", 243_8; "/threeoldstyle", 063_8; "/threequarters", 111_8; "/threequartersemdash", 075_8; "/threesuperior", 334_8; "/tsuperior", 346_8; "/twodotenleader", 052_8; "/twoinferior", 252_8; "/twooldstyle", 062_8; "/twosuperior", 333_8; "/twothirds", 117_8; "/zeroinferior", 274_8; "/zerooldstyle", 060_8; "/zerosuperior", 342_8]

Symbol Encoding

let $name\_to\_symbol$ =
["/Alpha", 101_8; "/Beta", 102_8; "/Chi", 103_8; "/Delta", 104_8; "/Epsilon", 105_8; "/Eta", 110_8; "/Euro", 240_8; "/Gamma", 107_8; "/Ifraktur", 301_8; "/Iota", 111_8; "/Kappa", 113_8; "/Lambda", 114_8; "/Mu", 115_8; "/Nu", 116_8; "/Omega", 127_8; "/Omicron", 117_8; "/Phi", 106_8; "/Pi", 120_8; "/Psi", 131_8; "/Rfraktur", 302_8; "/Rho", 122_8; "/Sigma", 123_8; "/Tau", 124_8; "/Theta", 121_8; "/Upsilon", 125_8; "/Upsilon1", 241_8; "/Xi", 130_8; "/Zeta", 132_8; "/aleph", 300_8; "/alpha", 141_8; "/ampersand", 046_8; "/angle", 320_8; "/angleleft", 341_8; "/angleright", 361_8; "/approxequal", 273_8; "/arrowboth", 253_8; "/arrowdblboth", 333_8; "/arrowdbldown", 337_8;

"/arrowdblleft", 334_8; "/arrowdblright", 336_8; "/arrowhorizex", 276_8; "/arrowleft", 254_8; "/arrowright", 256_8; "/arrowup", 255_8; "/arrowvertex", 275_8; "/asteriskmath", 052_8; "/bar", 174_8; "/beta", 142_8; "/braceleft", 173_8; "/braceright", 175_8; "/bracelefttp", 354_8; "/braceleftmid", 355_8; "/braceleftbt", 376_8; "/bracerighttp", 374_8; "/bracerightmid", 375_8; "/bracerightbt", 376_8; "/braceex", 357_8; "/bracketleft", 133_8; "/bracketright", 135_8; "/bracketlefttp", 351_8; "/bracketleftex", 352_8; "/bracketleftbt", 353_8; "/bracketrighttp", 371_8; "/brackerrightex", 372_8; "/bracketrightbt", 373_8; "/bullet", 267_8; "/carriagereturn", 277_8; "/chi", 143_8; "/circlemultiply", 304_8; "/circleplus", 305_8; "/club", 247_8; "/colon", 072_8; "/comma", 054_8; "/congruent", 100_8; "/copyrightsans", 343_8; "/copyrightserif", 323_8; "/degree", 260_8; "/delta", 144_8; "/diamond", 250_8; "/divide", 270_8; "/dotmath", 327_8; "/eight", 070_8; "/element", 316_8; "/ellipsis", 274_8; "/emptyset", 306_8; "/epsilon", 145_8; "/equal", 075_8; "/equivalence", 272_8; "/eta", 150_8; "/exclam", 041_8; "/existential", 044_8; "/five", 065_8; "/florin", 246_8; "/four", 064_8; "/fraction", 244_8; "/gamma", 147_8; "/gradient", 321_8; "/greater", 076_8; "/greaterequal", 263_8; "/heart", 251_8; "/infinity", 245_8; "/integral", 362_8; "/integraltp", 363_8; "/integralex", 364_8; "/integralbt", 365_8; "/intersection", 307_8; "/iota", 151_8; "/kappa", 153_8; "/lambda", 154_8; "/less", 074_8; "/lessequal", 243_8; "/logicaland", 331_8; "/logicalnot", 330_8; "/logicalor", 332_8; "/lozenge", 340_8; "/minus", 055_8; "/minute", 242_8; "/mu", 155_8; "/multiply", 264_8; "/nine", 071_8; "/notelement", 317_8; "/notequal", 271_8; "/notsubset", 313_8; "/nu", 156_8; "/numbersign", 043_8; "/omega", 167_8; "/omega1", 166_8; "/omicron", 157_8; "/one", 061_8; "/parenleft", 050_8; "/parenright", 051_8; "/parenlefttp", 346_8; "/parenleftex", 347_8; "/parenleftbt", 350_8; "/parenrighttp", 366_8; "/parenrightex", 367_8; "/parenrightbt", 370_8; "/partialdiff", 266_8; "/percent", 045_8; "/period", 056_8; "/perpendicular", 136_8; "/phi", 146_8; "/phi1", 152_8; "/pi", 160_8; "/plus", 153_8; "/plusminus", 261_8; "/product", 325_8; "/propersubset", 314_8; "/propersuperset", 311_8; "/proportional", 265_8; "/psi", 171_8; "/question", 077_8; "/radical", 326_8; "/radicalex", 140_8; "/reflexsubset", 315_8; "/reflexsuperset", 312_8; "/registersans", 342_8; "/registerserif", 322_8; "/rho", 162_8; "/second", 262_8; "/semicolon", 073_8; "/seven", 067_8; "/sigma", 163_8; "/sigma1", 126_8; "/similar", 176_8; "/six", 066_8; "/slash", 157_8; "/space", 040_8; "/spade", 252_8; "/suchthat", 047_8; "/summation", 345_8; "/tau", 164_8; "/therefore", 134_8; "/theta", 161_8; "/theta1", 112_8; "/three", 063_8; "/trademarksans", 344_8; "/trademarkserif", 324_8; "/two", 062_8; "/underscore", 137_8; "/union", 310_8; "/universal", 042_8; "/upsilon", 165_8; "/weierstrass", 303_8; "/xi", 303_8; "/zero", 060_8; "/zeta", 172_8]

6. Dingbats encoding

let $name\_to\_dingbats$ =
["/space", 040_8; "/a1", 041_8; "/a2", 042_8; "/a202", 043_8; "/a3", 044_8; "/a4", 045_8; "/a5", 046_8; "/a119", 047_8; "/a118", 050_8; "/a117", 051_8; "/a11", 052_8; "/a12", 053_8; "/a13", 054_8; "/a14", 055_8; "/a15", 056_8; "/a16", 057_8; "/a105", 060_8; "/a17", 061_8; "/a18", 062_8; "/a19", 063_8; "/a20", 064_8; "/a21", 065_8; "/a22", 066_8; "/a23", 067_8; "/a24", 070_8; "/a25", 071_8; "/a26", 072_8; "/a27", 073_8; "/a28", 074_8; "/a6", 075_8;

"/a7", 076_8; "/a8", 077_8; "/a9", 100_8; "/a10", 101_8; "/a29", 102_8; "/a30", 103_8; "/a31", 104_8; "/a32", 105_8; "/a33", 106_8; "/a34", 107_8; "/a35", 110_8; "/a36", 111_8; "/a37", 112_8; "/a38", 113_8; "/a39", 114_8; "/a40", 115_8; "/a41", 116_8; "/a42", 117_8; "/a43", 120_8; "/a44", 121_8; "/a45", 122_8; "/a46", 123_8; "/a47", 124_8; "/a48", 125_8; "/a49", 126_8; "/a50", 127_8; "/a51", 130_8; "/a52", 131_8; "/a53", 132_8; "/a54", 133_8; "/a55", 134_8; "/a56", 135_8; "/a57", 136_8; "/a58", 137_8; "/a59", 140_8; "/a60", 141_8; "/a61", 142_8; "/a62", 143_8; "/a63", 144_8; "/a64", 145_8; "/a65", 146_8; "/a66", 147_8; "/a67", 150_8; "/a68", 151_8; "/a69", 152_8; "/a70", 153_8; "/a71", 154_8; "/a72", 155_8; "/a73", 156_8; "/a74", 157_8; "/a203", 160_8; "/a75", 161_8; "/a204", 162_8; "/a76", 163_8; "/a77", 164_8; "/a78", 165_8; "/a79", 166_8; "/a81", 167_8; "/a82", 170_8; "/a83", 171_8; "/a84", 172_8; "/a97", 173_8; "/a98", 174_8; "/a99", 175_8; "/a100", 176_8; "/a101", 241_8; "/a102", 242_8; "/a103", 243_8; "/a104", 244_8; "/a106", 245_8; "/a107", 246_8; "/a108", 247_8; "/a112", 250_8; "/a111", 251_8; "/a110", 252_8; "/a109", 253_8; "/a120", 254_8; "/a121", 255_8; "/a122", 256_8; "/a123", 257_8; "/a124", 260_8; "/a125", 261_8; "/a126", 262_8; "/a127", 263_8; "/a128", 264_8; "/a129", 265_8; "/a130", 266_8; "/a131", 267_8; "/a132", 270_8; "/a133", 271_8; "/a134", 272_8; "/a135", 273_8; "/a136", 274_8; "/a137", 275_8; "/a138", 276_8; "/a139", 277_8; "/a140", 300_8; "/a141", 301_8; "/a142", 302_8; "/a143", 303_8; "/a144", 304_8; "/a145", 305_8; "/a146", 306_8; "/a147", 307_8; "/a148", 310_8; "/a149", 311_8; "/a150", 312_8; "/a151", 313_8; "/a152", 314_8; "/a153", 315_8; "/a154", 316_8; "/a155", 317_8; "/a156", 320_8; "/a157", 321_8; "/a158", 322_8; "/a159", 323_8; "/a160", 324_8; "/a161", 325_8; "/a163", 326_8; "/a164", 327_8; "/a196", 330_8; "/a165", 331_8; "/a192", 332_8; "/a166", 333_8; "/a167", 334_8; "/a168", 335_8; "/a169", 336_8; "/a170", 337_8; "/a171", 340_8; "/a172", 341_8; "/a173", 342_8; "/a162", 343_8; "/a174", 344_8; "/a175", 345_8; "/a176", 346_8; "/a177", 347_8; "/a178", 350_8; "/a179", 351_8; "/a193", 352_8; "/a180", 353_8; "/a199", 354_8; "/a181", 355_8; "/a200", 356_8; "/a182", 357_8; "/a201", 361_8; "/a183", 362_8; "/a184", 363_8; "/a197", 364_8; "/a185", 365_8; "/a194", 366_8; "/a198", 367_8; "/a186", 370_8; "/a195", 371_8; "/a187", 372_8; "/a188", 373_8; "/a189", 374_8; "/a190", 375_8; "/a191", 376_8]

Parse a /ToUnicode CMap to extract font mapping.

```
type section =
  | BfChar of char list
  | BfRange of char list

let rec getuntilend prev = function
  | [] → rev prev, []
  | 'e' :: 'n' :: 'd' :: 'b' :: 'f' :: 'c' :: 'h' :: 'a' :: 'r' :: more →
rev prev, more
  | h :: t → getuntilend (h :: prev) t

let rec getuntilend_range prev = function
  | [] → rev prev, []
  | 'e' :: 'n' :: 'd' :: 'b' :: 'f' :: 'r' :: 'a' :: 'n' :: 'g' :: 'e' :: more →
rev prev, more
  | h :: t → getuntilend_range (h :: prev) t
```

```
let rec get_section = function
  | [] → None
  | 'b' :: 'e' :: 'g' :: 'i' :: 'n' :: 'b' :: 'f' :: 'c' :: 'h' :: 'a' :: 'r' ::
more →
      let numbers, rest = getuntilend [] more in
        Some (BfChar numbers, rest)
  | 'b' :: 'e' :: 'g' :: 'i' :: 'n' :: 'b' :: 'f' :: 'r' :: 'a' :: 'n' :: 'g' :: 'e' ::
more →
      let numbers, rest = getuntilend_range [] more in
        Some (BfRange numbers, rest)
  | _ :: t → get_section t
```

Read a character code.

```
let rec read_number = function
  | x :: more when Pdf.is_whitespace x → read_number more
  | '<' :: a :: '>' :: more →
      int_of_string (implode ['0'; 'x'; a]), more
  | '<' :: a :: b :: '>' :: more →
      int_of_string (implode ['0'; 'x'; a; b]), more
  | '<' :: a :: b :: c :: '>' :: more →
      int_of_string (implode ['0'; 'x'; a; b; c]), more
  | '<' :: a :: b :: c :: d :: '>' :: more →
      int_of_string (implode ['0'; 'x'; a; b; c; d]), more
  | [] → dpr "Z"; raise Not_found
  | _ → raise (Pdf.PDFError "Unknown number in /ToUnicode")
```

Read the bytes of the UTF-16BE unicode sequence as a string.

```
let fail () =
  raise (Pdf.PDFError "Bad unicode value")
```

```
let rec read_unicode = function
  | x :: rest when Pdf.is_whitespace x → read_unicode rest
  | '<' :: rest →
      let chars, rest = cleavewhile (neq '>') rest in
        let is_hex_digit = function
          | '0'..'9' | 'a'..'f' | 'A'..'F' → true
          | _ → false
        in
          iter
            (fun x → if ¬ (is_hex_digit x) then fail ())
            chars;
          if length chars > 0 ∧ even (length chars) then
            let bytes =
              map
                (function
                  | [x; y] → char_of_int (int_of_string (implode ['0'; 'x'; x; y]))
                  | _ → raise (Assert_failure ("", 0, 0)))
                (splitinto 2 chars)
            in
              let rest' =
                match rest with
```

```
              | [] → []
              | _ → tl rest
            in
              implode bytes, rest'
          else
            fail ()
  | _ → fail ()

let print_bytestream s =
  flprint "\n";
  for x = 0 to stream_size s − 1 do
    print_char (char_of_int (sget s x))
  done;
  flprint "\n"

let rec get_sections chars =
  match get_section chars with
  | None → []
  | Some (sec, restchars) →
      sec :: get_sections restchars

let pairs_of_section = function
  | BfChar numbers →
      let results = ref []
      and numbers = ref numbers in
        begin try
          while true do
            let number, rest = read_number !numbers in
              let str, rest = read_unicode rest in
                numbers := rest;
                results =| (number, str)
          done;
          []
        with
          Not_found → dpr "3J"; rev !results
        end
  | BfRange numbers →
      let results = ref []
      and numbers = ref numbers in
        begin try
          while true do
            let src1, rest = read_number !numbers in
              let src2, rest = read_number rest in
                if src1 > src2 then raise (Pdf.PDFError "Bad /ToUnicode") else
                  match rest with
                  | '<' :: _ →
                      (∗ It's a single unicode string ∗)
                      let increment_final code d =
                        match code with
                        | "" → ""
                        | s →
                            let chars = rev (explode s) in
```

```
                          implode ((rev (tl chars)) @ [char_of_int (int_of_char (hd chars) + c
                                          in
                    let code, rest = read_unicode rest in
                      results =@
                        rev
                          (combine
                            (ilist src1 src2)
                            (map (increment_final code) (ilist 0 (src2 - src1))));
                      numbers := rest
              | '[' :: rest →
                  (* It's several. *)
                  let rest = ref rest in
                    results =@
                      combine
                        (ilist src1 src2)
                        (map
                          (fun _ →
                             let num, rest' = read_unicode !rest in
                               rest := rest';
                               num)
                          (ilist 0 (src2 - src1)));
                  rest := (match !rest with [] → [] | x → tl x);
                  numbers := !rest
              | _ → raise (Pdf.PDFError "Bad BfRange")
       done;
       []
     with
       Not_found → dpr "3I";rev !results
     end

let rec parse_tounicode pdf tounicode =
  match tounicode with
  | Pdf.Stream {contents = (dict, Pdf.Got data)} →
      Pdfcodec.decode_pdfstream pdf tounicode;
      begin match tounicode with
      | Pdf.Stream {contents = (dict, Pdf.Got data)} →
          begin try
            flatten
              (map pairs_of_section
                (get_sections
                   (lose Pdf.is_whitespace (charlist_of_bytestream data))))
          with
            e → Printf.eprintf "/ToUnicode Parse Error : %s\n" (Printexc.to_string e); []
          end
      | _ → raise (Assert_failure ("", 0, 0))
      end
  | Pdf.Stream {contents = (_, Pdf.ToGet (_, _, _))} →
      Pdf.getstream tounicode;
      parse_tounicode pdf tounicode
  | e → raise (Pdf.PDFError "Bad /ToUnicode")
```

Extracting of Text
A text extractor takes a character and returns a list of unicode codepoints. This may have to be extended when we deal with composite fonts.

```
type text_extractor =
  {convert : int → int list;
   font : font}
```

Encode utf16be

```
let utf16be_of_codepoint u =
  if u < 0 ∨ u > 10FFFF₁₆ then
    raise (Invalid_argument "utf16be_of_codepoints")
  else
    if u < 10000₁₆ then [u] else
      let u' = u − 10000₁₆
      and w1 = D800₁₆
      and w2 = DC00₁₆ in
        let w1 = w1 lor (u' lsr 10)
        and w2 = w2 lor (u' land 1111111111₂) in
          [w1; w2]

let utf16be_of_codepoints l =
  implode (map char_of_int (flatten (map utf16be_of_codepoint l)))
```

Return a list of codepoints from a UTF-16BE string. See RFC2871

```
let fail2 () =
  raise (Invalid_argument "codepoints_of_utf16be")

let rec codepoints_of_utf16be_inner prev = function
  | [] → rev prev
  | [w1] → fail2 ()
  | [w1a; w1b] →
      let w1 = (w1a lsl 8) lor w1b in
        if w1 < D800₁₆ ∨ w1 > DFFF₁₆ then
          codepoints_of_utf16be_inner (w1 :: prev) []
        else
          fail2 ()
  | [_; _; _] → fail2 ()
  | w1a :: w1b :: w2a :: w2b :: more →
      let w1 = (w1a lsl 8) lor w1b in
        if w1 < D800₁₆ ∨ w1 > DFFF₁₆ then
          codepoints_of_utf16be_inner (w1 :: prev) (w2a :: w2b :: more)
        else
          if w1 ≥ D800₁₆ ∧ w1 ≤ DBFF₁₆ then
            let w2 = (w2a lsl 8) lor w2b in
              if w2 ≥ DC00₁₆ ∧ w2 ≤ DFFF₁₆ then
                let ho = w1 land 1111111111₂
                and lo = w2 lsr 6 in
                  codepoints_of_utf16be_inner
                    ((((ho lsl 10) lor lo) + 10000₁₆) :: prev) more
              else
                fail2 ()
```

```
        else
          fail2 ()

let codepoints_of_utf16be str =
  codepoints_of_utf16be_inner [] (map int_of_char (explode str))

let glyph_hashes =
  hashtable_of_dictionary
    (Glyphlist.glyphmap @ Glyphlist.dingbatmap @ Glyphlist.truetypemap)
```

Build a hashtable for lookups based on an encoding

```
let table_of_encoding pdf font encoding =
    let table = Hashtbl.create 203
  and swp = map (fun (k, v) → (v, k)) in
    let addvals = iter (fun (k, v) → Hashtbl.add table k v) in
      let rec add_encoding = function
        | ImplicitInFontFile →
                          ()
        | StandardEncoding →
                          addvals (swp name_to_standard)
        | MacRomanEncoding →
                          addvals (swp name_to_macroman)
        | WinAnsiEncoding →
                          addvals (swp name_to_win)
        | MacExpertEncoding →
                          addvals (swp name_to_macexpert)
        | CustomEncoding (e, ds) →
                          add_encoding e;
            addvals (swp ds)
        | FillUndefinedWithStandard e →
                          addvals (swp name_to_standard);
            add_encoding e
      in
        add_encoding encoding;
        (* Print the table out *)
                table
```

Method: 1. If there's a /ToUnicode CMap, use it. 2. If it is a standard 14 or simple font, use the encoding to get a glyph name, then look up the character in the glyph list. 3. If it's a CID font, which we don't understand, just return. The font here is the PDF font structure, not our font data type. If we need to parse it, we do.

```
let text_extractor_of_font pdf font =
  match Pdf.lookup_direct pdf "/ToUnicode" font with
  | Some tounicode →
      let convert =
        begin try
          let table =
            hashtable_of_dictionary <| parse_tounicode pdf tounicode
          in
            begin function i →
```

```
              try
                codepoints_of_utf16be (Hashtbl.find table i)
              with
              Not_found → [i]
            end
        with
          Pdf.PDFError ("Bad /ToUnicode") → dpr "3K"; (function i →
[i])
        end
      in
        {convert = convert; font = read_font pdf font}
  | None →
      let convert =
        if is_simple_font pdf font ∨ is_standard14font pdf font then
          let encoding =
            match read_font pdf font with
            | StandardFont (_, e) → e
            | SimpleFont {encoding = e} → e
            | _ → raise (Assert_failure ("", 0, 0))
          in
           let table = table_of_encoding pdf font encoding in
          begin function i →
            try
                          let decoded = Hashtbl.find table i in
                            let r = Hashtbl.find glyph_hashes decoded in
                              r
            with
              Not_found → dpr "3L"; [i]
          end
        else
          (function i → [i])
    in
      {convert = convert; font = read_font pdf font}
```

For now, the only composite font encoding scheme we understand is /Identity-H

```
let is_identity_h = function
  | CIDKeyedFont (_, _, Predefined "/Identity-H") → true
  | _ → false

let codepoints_of_text extractor text =
  if text = "" then [] else
    if is_identity_h extractor.font then
      let chars = map int_of_char <| explode text in
        if odd (length chars)
          then raise (Pdf.PDFError "Bad Text")
          else
            let pairs = pairs_of_list chars in
              let cs = ref [] in
              iter
                (fun (h, l) →
                  let codepoints = extractor.convert ((h lsl 8) lor l) in
```

```
                    cs =@ rev codepoints)
                  pairs;
                rev !cs
    else
      begin
        let cs = ref [] in
        for x = 0 to String.length text - 1 do
          cs =@ rev (extractor.convert (int_of_char text.[x]))
        done;
        rev !cs
      end
```

Send each byte to the text extractor, and concatenate the unicode codepoint lists which result.

```
let utf16be_of_text extractor text =
  utf16be_of_codepoints (codepoints_of_text extractor text)
```

Convert UTF16BE to Latin1. Chars ¿ U+0255 are dropped silently.

```
let latin1_of_utf16be str =
  implode
  (map char_of_int
  (option_map
      (fun x →
         if x < 0 then raise (Assert_failure ("", 0, 0)) else
         if x < 256 then Some x else None)
  (codepoints_of_utf16be str)))
```

Lossily convert to Latin1 string

```
let latin1_string_of_text extractor text =
  latin1_of_utf16be <| utf16be_of_text extractor text
```

Decode a character according to an encoding

```
let decode_char encoding chr =
  try
    (* FIXME: Addded Pdf.empty and Pdf.Null here - do it properly? *)
    let table = table_of_encoding (Pdf.empty ()) Pdf.Null encoding in
      let name = Hashtbl.find table (int_of_char chr) in
        let number = Hashtbl.find glyph_hashes name in
          match number with
          | [number] →
            if number < 0 then raise (Assert_failure ("", 0, 0)) else
            if number > 255 then chr else
            char_of_int number
          | _ → dpr "a"; raise Not_found
  with
    Not_found → chr
```

Return the glyph name from a char in a type3 font. Raises Not_found if not found.

```
let decode_type3_char encoding chr =
  let table = table_of_encoding (Pdf.empty ()) Pdf.Null encoding in
      let r = Hashtbl.find table (int_of_char chr) in
        r
```

Is a PDF string unicode (does it have a byte order marker at the beginning).

```
let is_unicode s =
  (String.length s ≥ 2) ∧ s.[0] = '\254' ∧ s.[1] = '\255'
```

Convert a codepoint in PDFDocEncoding to a Unicode Codepoint - just drop ¿ 127, ¡ 32.

```
let codepoint_of_pdfdocencoding_character i =
  if i > 127 ∨ i < 32 then None else Some i
```

Look at a PDF string, determine if it's PDFDocEncoding or UTF16BE, and return the unicode codepoints or the byte values as a list of integers in either case.

```
let codepoints_of_textstring s =
  if is_unicode s then
    codepoints_of_utf16be (String.sub s 2 (String.length s − 2))
  else
    option_map codepoint_of_pdfdocencoding_character (map int_of_char (explode s))
```

PDF Colour space parsing

```
open Utility

type point = float × float × float

type iccbased =
 {icc_n : int;
  icc_alternate : colourspace;
  icc_range : float array;
  icc_metadata : Pdf.pdfobject option;
  icc_stream : Pdf.pdfobject}

and colourspace =
  | DeviceGray
  | DeviceRGB
  | DeviceCMYK
  | CalGray of point × point × float (* White, Black, Gamma *)
  | CalRGB of point × point × float array × float array (* White, Black, Gamma, Matrix *)
  | Lab of point × point × float array (* White, Black, Range *)
  | ICCBased of iccbased
  | Indexed of colourspace × (int, int list) Hashtbl.t (* Base colourspace, values *)
  | Pattern
  | Separation of string × colourspace × Pdffun.pdf_fun
  | DeviceN of string array × colourspace × Pdffun.pdf_fun × Pdf.pdfobject
```

```
let rec string_of_colourspace = function
  | DeviceGray → "/DeviceGray"
  | DeviceRGB → "/DeviceRGB"
  | DeviceCMYK → "/DeviceCMYK"
  | CalGray (_, _, _) → "/CalGray"
  | CalRGB (_, _, _, _) → "/CalRGB"
  | Lab (_, _, _) → "/Lab"
  | ICCBased {icc_alternate = a} →
      "ICC Based - alternate is " ^ string_of_colourspace a
  | Indexed (a, _) →
      "Indexed - base is " ^ string_of_colourspace a
  | Pattern → "/Pattern"
  | Separation (_, a, _) →
      "Separation - base is " ^ string_of_colourspace a
  | DeviceN (_, a, _, _) →
      "DeviceN - base is " ^ string_of_colourspace a
```

Read a tristimulus point.

```
let read_point pdf d n =
  match Pdf.lookup_direct pdf n d with
  | Some (Pdf.Array [a; b; c]) →
      Pdf.getnum a, Pdf.getnum b, Pdf.getnum c
  | _ →
      0., 0., 0.
```

```
let rec get_basic_table_colourspace c =
  match c with
  | Indexed (alt, _)
  (* FIXME Not actually checked the following two are correct *)
  | DeviceN (_, alt, _, _)
  | Separation (_, alt, _)
  | ICCBased {icc_alternate = alt} → get_basic_table_colourspace alt
  | x → x
```

Read a colour space. Raises Not_found on error.

```
let rec read_colourspace_inner pdf resources = function
  | Pdf.Indirect i →
      read_colourspace_inner pdf resources (Pdf.direct pdf (Pdf.Indirect i))
  | Pdf.Name ("/DeviceGray" | "/G") → DeviceGray
  | Pdf.Name ("/DeviceRGB" | "/RGB") → DeviceRGB
  | Pdf.Name ("/DeviceCMYK" | "/CMYK") → DeviceCMYK
  | Pdf.Name "/Pattern" → Pattern
  | Pdf.Array [Pdf.Name "/Pattern"; base_colspace] → Pattern (* FIXME *)
  | Pdf.Array [onething] → read_colourspace_inner pdf resources onething (*
illus_effects.pdf [/Pattern] *)
  | Pdf.Name space →
          begin match Pdf.lookup_direct pdf "/ColorSpace" resources with
      | Some csdict →
          begin match Pdf.lookup_direct pdf space csdict with
          | Some space' →
                      read_colourspace_inner pdf resources space'
```

```
        | None → dpr "X"; raise Not_found
        end
    | None → dpr "Y"; raise Not_found
    end
| Pdf.Array [Pdf.Name "/CalGray"; dict] →
    let whitepoint = read_point pdf dict "/WhitePoint"
    and blackpoint = read_point pdf dict "/BlackPoint"
    and gamma =
      match Pdf.lookup_direct pdf "/Gamma" dict with
      | Some n → Pdf.getnum n
      | None → 1.
    in
      CalGray (whitepoint, blackpoint, gamma)
| Pdf.Array [Pdf.Name "/CalRGB"; dict] →
    let whitepoint = read_point pdf dict "/WhitePoint"
    and blackpoint = read_point pdf dict "/BlackPoint"
    and gamma =
      match Pdf.lookup_direct pdf "/Gamma" dict with
      | Some (Pdf.Array [a; b; c]) →
          [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c|]
      | _ →
          [|1.; 1.; 1.|]
    and matrix =
      match Pdf.lookup_direct pdf "/Matrix" dict with
      | Some (Pdf.Array [a; b; c; d; e; f; g; h; i]) →
          [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c;
            Pdf.getnum d; Pdf.getnum e; Pdf.getnum f;
            Pdf.getnum g; Pdf.getnum h; Pdf.getnum i|]
      | _ →
          [|1.; 0.; 0.; 0.; 1.; 0.; 0.; 0.; 1.|]
    in
      CalRGB (whitepoint, blackpoint, gamma, matrix)
| Pdf.Array [Pdf.Name "/Lab"; dict] →
    let whitepoint = read_point pdf dict "/WhitePoint"
    and blackpoint = read_point pdf dict "/BlackPoint"
    and range =
      match Pdf.lookup_direct pdf "/Range" dict with
      | Some (Pdf.Array [a; b; c; d]) →
          [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c; Pdf.getnum d|]
      | _ →
          [|~-.100.; 100.; ~-.100.; 100.|]
    in
      Lab (whitepoint, blackpoint, range)
| Pdf.Array [Pdf.Name "/ICCBased"; stream] →
    begin match Pdf.direct pdf stream with
    | Pdf.Stream {contents = (dict, _)} →
        let n =
          match Pdf.lookup_direct pdf "/N" dict with
          | Some (Pdf.Integer n) →
              if n = 1 ∨ n = 3 ∨ n = 4 then n else raise Not_found
```

```
            | _ → raise Not_found
          in
            let alternate =
              match Pdf.lookup_direct pdf "/Alternate" dict with
              | Some cs → read_colourspace_inner pdf resources cs
              | _ →
                  match n with
                  | 1 → DeviceGray
                  | 3 → DeviceRGB
                  | 4 → DeviceCMYK
                  | _ → raise (Assert_failure ("", 0, 0))
            and range =
              match Pdf.lookup_direct pdf "/Range" dict with
              | Some (Pdf.Array elts) when length elts = 2 × n →
                  Array.of_list (map Pdf.getnum elts)
              | _ →
                  Array.of_list (flatten (many [0.; 1.] n))
            and metadata =
              Pdf.lookup_direct pdf "/Metadata" dict
            in
              ICCBased
                {icc_n = n;
                 icc_alternate = alternate;
                 icc_range = range;
                 icc_metadata = metadata;
                 icc_stream = stream}
      | _ → raise Not_found
      end
  | Pdf.Array [Pdf.Name ("/Indexed" | "/I"); bse; hival; lookup_data] →
      let hival =
        match hival with
        | Pdf.Integer h → h
        | _ → raise (Pdf.PDFError "Bad /Hival")
      and bse =
        read_colourspace_inner pdf resources bse
      in
        let mktable_rgb data =
          try
            let table = Hashtbl.create (hival + 1)
            and i = Pdfio.input_of_bytestream data in
              for x = 0 to hival do
                let r = i.Pdfio.input_byte () in
                let g = i.Pdfio.input_byte () in
                let b = i.Pdfio.input_byte () in
                  Hashtbl.add table x [r; g; b]
              done;
              table
          with _ → failwith "bad table"
        and mktable_cmyk data =
          try
```

```
            let table = Hashtbl.create (hival + 1)
            and i = Pdfio.input_of_bytestream data in
              for x = 0 to hival do
                let c = i.Pdfio.input_byte () in
                let m = i.Pdfio.input_byte () in
                let y = i.Pdfio.input_byte () in
                let k = i.Pdfio.input_byte () in
                  Hashtbl.add table x [c; m; y; k]
              done;
              table
          with _ → failwith "bad table"
        in
          let table =
            begin match Pdf.direct pdf lookup_data with
            | (Pdf.Stream _) as stream →
                Pdfcodec.decode_pdfstream pdf stream;
                begin match stream with
                | (Pdf.Stream {contents = (_, Pdf.Got data)}) →
                    begin match get_basic_table_colourspace bse with
                    | DeviceRGB | CalRGB _ → mktable_rgb data
                    | DeviceCMYK → mktable_cmyk data
                    | _ → failwith "Unknown base colourspace in index
colourspace"
                    end
                | _ → raise (Pdf.PDFError "Indexed/Inconsistent")
                end
            | Pdf.String s →
                let data = mkstream (String.length s) in
                  for x = 0 to stream_size data − 1 do
                    sset data x (int_of_char s.[x])
                  done;
                  begin match get_basic_table_colourspace bse with
                  | DeviceRGB | CalRGB _ → mktable_rgb data
                  | DeviceCMYK → mktable_cmyk data
                  | _ → failwith "Unknown base colourspace in index
colourspace"
                  end
            | _ → failwith "unknown indexed colourspace"
            end
          in
            Indexed (bse, table)
  | Pdf.Array [Pdf.Name "/Separation"; Pdf.Name name; alternate; tint] →
      let alt_space =
        read_colourspace_inner pdf resources alternate
      and tint_transform =
        Pdffun.parse_function pdf tint
      in
        Separation (name, alt_space, tint_transform)
  | Pdf.Array [Pdf.Name "/DeviceN"; Pdf.Array names; alternate; tint] →
      let names =
```

```
          Array.of_list (map (function Pdf.Name s → s | _ → raise Not_found) names)
        and alternate =
          read_colourspace_inner pdf resources alternate
        and tint =
          Pdffun.parse_function pdf tint
        in
          DeviceN (names, alternate, tint, Pdf.Dictionary [])
  | Pdf.Array [Pdf.Name "/DeviceN"; Pdf.Array names; alternate; tint; attributes] →

        let names =
          Array.of_list (map (function Pdf.Name s → s | _ → raise Not_found) names)
        and alternate =
          read_colourspace_inner pdf resources alternate
        and tint =
          Pdffun.parse_function pdf tint
        in
          DeviceN (names, alternate, tint, attributes)
  | _ → raise Not_found

let read_colourspace pdf resources space =
  try
    read_colourspace_inner pdf resources space
  with
    e →
        raise e
```

PDF Bookmarks

```
open Utility

type target = int (* Just page number for now *)

type bookmark =
  {level : int;
   text : string;
   target : target;
   isopen : bool}

let remove_bookmarks pdf =
  match Pdf.lookup_direct pdf "/Root" pdf.Pdf.trailerdict with
  | None → raise (Pdf.PDFError "remove_boomarks:  Bad PDF: no root")
  | Some catalog →
      let catalog' = Pdf.remove_dict_entry catalog "/Outlines" in
        let newcatalognum = Pdf.addobj pdf catalog' in
          {pdf with
            Pdf.root = newcatalognum;
            Pdf.trailerdict =
              Pdf.add_dict_entry
                pdf.Pdf.trailerdict "/Root" (Pdf.Indirect newcatalognum)}

type ntree =
  Br of int × Pdf.pdfobject × ntree list
```

```
let rec print_ntree (Br (i, _, l)) =
  Printf.printf "%i (" i;
  iter print_ntree l;
  flprint ")"
```

```
let fresh source pdf =
  incr source; Pdf.maxobjnum pdf + !source
```

Flatten a tree and produce a root object for it. Return a list of (num, pdfobject) pairs with the root first.

```
let flatten_tree source pdf = function
  | [] →
      let n = fresh source pdf in
        [(n, Pdf.Dictionary [])], n
  | tree →
      let root_objnum = fresh source pdf in
      (* Add /Parent links to root *)
      let tree =
        let add_root_parent (Br (i, dict, children)) =
          Br
            (i,
             Pdf.add_dict_entry dict "/Parent" (Pdf.Indirect root_objnum),
             children)
        in
          map add_root_parent tree
      in
        let rec really_flatten = function
          Br (i, pdfobject, children) →
            (i, pdfobject) :: flatten (map really_flatten children)
        in
          let all_but_top = flatten (map really_flatten tree)
          and top, topnum =
            (* Make top level from objects at first level of tree *)
            match extremes tree with
              Br (first, _, _), Br (last, _, _) →
                (root_objnum, Pdf.Dictionary
                  [("/First", Pdf.Indirect first); ("/Last", Pdf.Indirect last)]),
                root_objnum
          in
            top :: all_but_top, topnum
```

Add /Count entries to an ntree

```
let add_counts tree = tree
```

Add /Parent entries to an ntree

```
let rec add_parent parent (Br (i, obj, children)) =
  let obj' =
    match parent with
    | None → obj
    | Some parent_num →
        Pdf.add_dict_entry obj "/Parent" (Pdf.Indirect parent_num)
```

```
    in
      Br (i, obj′, map (add_parent (Some i)) children)
```

Add /First and /Last entries to an ntree

```
let rec add_firstlast (Br (i, obj, children)) =
  match children with
  | [] → (Br (i, obj, children))
  | c →
      match extremes c with
        Br (i′, _, _), Br (i″, _, _) →
          let obj = Pdf.add_dict_entry obj "/First" (Pdf.Indirect i′) in
            let obj = Pdf.add_dict_entry obj "/Last" (Pdf.Indirect i″) in
              (Br (i, obj, map add_firstlast children))
```

Add /Next and /Prev entries to an ntree

```
let rec add_next (Br (i, obj, children)) =
  match children with
  | [] → Br (i, obj, children)
  | [_] → Br (i, obj, map add_next children)
  | c :: cs →
      let numbers = map (fun (Br (i, _, _)) → i) cs in
        let children′ =
          (map2
             (fun (Br (i, obj, children)) nextnum →
                Br (i,
                    Pdf.add_dict_entry obj "/Next" (Pdf.Indirect nextnum),
                    children))
             (all_but_last (c :: cs))
             numbers)
          @ [last cs]
        in
          Br (i, obj, map add_next children′)

let rec add_prev (Br (i, obj, children)) =
  match children with
  | [] → Br (i, obj, children)
  | [_] → Br (i, obj, map add_prev children)
  | c :: cs →
      let numbers = map (fun (Br (i, _, _)) → i) (all_but_last (c :: cs)) in
        let children′ =
          c ::
            map2
              (fun (Br (i, obj, children)) prevnum →
                 Br (i,
                     Pdf.add_dict_entry obj "/Prev" (Pdf.Indirect prevnum),
                     children))
              cs
              numbers
        in
          Br (i, obj, map add_prev children′)
```

Find a page indirect from the page tree of a document, given a page number.

```
let page_object_number pdf destpage =
  try
    Pdf.Indirect (select destpage (Pdf.page_reference_numbers pdf))
  with
    (* The page might not exist in the output *)
    Invalid_argument "select" → dpr "3b"; Pdf.Null
```

Make a node from a given title, destination page number in a given PDF ond open flag.

```
let node_of_line pdf title destpage isopen =
  if destpage > 0 then (* destpage = 0 means no destination. *)
    Pdf.Dictionary
      [("/Title", Pdf.String title);
       ("/Dest", Pdf.Array
          [page_object_number pdf destpage; Pdf.Name "/Fit"])]
  else
    Pdf.Dictionary [("/Title", Pdf.String title)]
```

Make an ntree list from a list of parsed bookmark lines.

```
let rec make_outline_ntree source pdf = function
  | [] → []
  | (n, title, destpage, isopen) :: t →
      let lower, rest = cleavewhile (fun (n', _, _, _) → n' > n) t in
        let node = node_of_line pdf title destpage isopen in
          Br (fresh source pdf, node, make_outline_ntree source pdf lower)
            ::make_outline_ntree source pdf rest
```

Temporary routine

```
let tuple_of_record r =
  r.level, r.text, r.target, r.isopen
```

Add bookmarks.

```
let add_bookmarks parsed pdf =
  let parsed = map tuple_of_record parsed in
  if parsed = [] then remove_bookmarks pdf else
  begin
    let source = ref 0 in
    let tree = make_outline_ntree source pdf parsed in
      (* Build the (object number, bookmark tree object) pairs. *)
      let pairs, tree_root_num =
        let tree =
          map add_firstlast tree
        in
          let tree =
            match add_next (add_prev (Br (0, Pdf.Null, tree))) with
              Br (_, _, children) → children
          in
            flatten_tree source pdf (add_counts (map (add_parent None) tree))
      in
```

```
        (* Add the objects to the pdf *)
        iter
          (function x → ignore (Pdf.addobj_given_num pdf x))
          pairs;
          (* Replace the /Outlines entry in the document catalog. *)
          match Pdf.lookup_direct pdf "/Root" pdf.Pdf.trailerdict with
          | None → raise (Pdf.PDFError "Bad PDF: no root")
          | Some catalog →
              let catalog' =
                Pdf.add_dict_entry catalog "/Outlines" (Pdf.Indirect tree_root_num)
              in
                let newcatalognum = Pdf.addobj pdf catalog' in
                  {pdf with
                    Pdf.root = newcatalognum;
                    Pdf.trailerdict =
                      Pdf.add_dict_entry
                        pdf.Pdf.trailerdict "/Root" (Pdf.Indirect newcatalognum)}
  end
```

The 14 Standard PDF Fonts (Widths and Kerns).

```
open Utility
```

The raw kern data is a list of integer triples. We need the first two as a pair, to form a key for the lookup hashtable.

```
let hashtable_of_kerns kerns =
  hashtable_of_dictionary
    (map (fun (c, c', k) → (c, c'), k) kerns)
```

TimesRoman

```
let times_roman_widths =
  hashtable_of_dictionary
  [32, 250; 33, 333; 34, 408; 35, 500; 36, 500; 37, 833; 38, 778; 39, 333;
   40, 333; 41, 333; 42, 500; 43, 564; 44, 250; 45, 333; 46, 250; 47, 278;
   48, 500; 49, 500; 50, 500; 51, 500; 52, 500; 53, 500; 54, 500; 55, 500;
   56, 500; 57, 500; 58, 278; 59, 278; 60, 564; 61, 564; 62, 564; 63, 444;
   64, 921; 65, 722; 66, 667; 67, 667; 68, 722; 69, 611; 70, 556; 71, 722;
   72, 722; 73, 333; 74, 389; 75, 722; 76, 611; 77, 889; 78, 722; 79, 722;
   80, 556; 81, 722; 82, 667; 83, 556; 84, 611; 85, 722; 86, 722; 87, 944;
   88, 722; 89, 722; 90, 611; 91, 333; 92, 278; 93, 333; 94, 469; 95, 500;
   96, 333; 97, 444; 98, 500; 99, 444; 100, 500; 101, 444; 102, 333; 103, 500;
   104, 500; 105, 278; 106, 278; 107, 500; 108, 278; 109, 778; 110, 500;
   111, 500; 112, 500; 113, 500; 114, 333; 115, 389; 116, 278; 117, 500;
   118, 500; 119, 722; 120, 500; 121, 500; 122, 444; 123, 480; 124, 200;
   125, 480; 126, 541; 161, 333; 162, 500; 163, 500; 164, 167; 165, 500;
   166, 500; 167, 500; 168, 500; 169, 180; 170, 444; 171, 500; 172, 333;
   173, 333; 174, 556; 175, 556; 177, 500; 178, 500; 179, 500; 180, 250;
   182, 453; 183, 350; 184, 333; 185, 444; 186, 444; 187, 500; 188, 1000;
   189, 1000; 191, 444; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
   198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
   207, 333; 208, 1000; 225, 889; 227, 276; 232, 611; 233, 722; 234, 889;
   235, 310; 241, 667; 245, 278; 248, 278; 249, 500; 250, 722; 251, 500]
```

let $times\_roman\_kerns$ $=$
$hashtable\_of\_kerns$
$[65,\ 121,\ -92;\ 65,\ 119,\ -92;\ 65,\ 118,\ -74;\ 65,\ 117,\ 0;\ 65,\ 39,\ -111;$
65, 186, 0; 65, 112, 0; 65, 89, -105; 65, 87, -90; 65, 86, -135;
65, 85, -55; 65, 84, -111; 65, 81, -55; 65, 79, -55; 65, 71, -40;
65, 67, -40; 66, 46, 0; 66, 44, 0; 66, 85, -10; 66, 65, -35; 68, 46, 0;
68, 44, 0; 68, 89, -55; 68, 87, -30; 68, 86, -40; 68, 65, -40; 70, 114, 0;
70, 46, -80; 70, 111, -15; 70, 105, 0; 70, 101, 0; 70, 44, -80; 70, 97, -15;
70, 65, -74; 71, 46, 0; 71, 44, 0; 74, 117, 0; 74, 46, 0; 74, 111, 0;
74, 101, 0; 74, 44, 0; 74, 97, 0; 74, 65, -60; 75, 121, -25; 75, 117, -15;
75, 111, -35; 75, 101, -25; 75, 79, -30; 76, 121, -55; 76, 39, -92;
76, 186, 0; 76, 89, -100; 76, 87, -74; 76, 86, -100; 76, 84, -92; 78, 46, 0;
78, 44, 0; 78, 65, -35; 79, 46, 0; 79, 44, 0; 79, 89, -50; 79, 88, -40;
79, 87, -35; 79, 86, -50; 79, 84, -40; 79, 65, -35; 80, 46, -111; 80, 111, 0;
80, 101, 0; 80, 44, -111; 80, 97, -15; 80, 65, -92; 81, 46, 0; 81, 44, 0;
81, 85, -10; 82, 89, -65; 82, 87, -55; 82, 86, -80; 82, 85, -40; 82, 84, -60;
82, 79, -40; 83, 46, 0; 83, 44, 0; 84, 121, -80; 84, 119, -80; 84, 117, -45;
84, 59, -55; 84, 114, -35; 84, 46, -74; 84, 111, -80; 84, 105, -35;
84, 45, -92; 84, 104, 0; 84, 101, -70; 84, 44, -74; 84, 58, -50; 84, 97, -80;
84, 79, -18; 84, 65, -93; 85, 46, 0; 85, 44, 0; 85, 65, -40; 86, 117, -75;
86, 59, -74; 86, 46, -129; 86, 111, -129; 86, 105, -60; 86, 45, -100;
86, 101, -111; 86, 44, -129; 86, 58, -74; 86, 97, -111; 86, 79, -40;
86, 71, -15; 86, 65, -135; 87, 121, -73; 87, 117, -50; 87, 59, -37;
87, 46, -92; 87, 111, -80; 87, 105, -40; 87, 45, -65; 87, 104, 0;
87, 101, -80; 87, 44, -92; 87, 58, -37; 87, 97, -80; 87, 79, -10;
87, 65, -120; 89, 117, -111; 89, 59, -92; 89, 46, -129; 89, 111, -110;
89, 105, -55; 89, 45, -111; 89, 101, -100; 89, 44, -129; 89, 58, -92;
89, 97, -100; 89, 79, -30; 89, 65, -120; 97, 121, 0; 97, 119, -15;
97, 118, -20; 97, 116, 0; 97, 112, 0; 97, 103, 0; 97, 98, 0; 98, 121, 0;
98, 118, -15; 98, 117, -20; 98, 46, -40; 98, 108, 0; 98, 44, 0; 98, 98, 0;
99, 121, -15; 99, 46, 0; 99, 108, 0; 99, 107, 0; 99, 104, 0; 99, 44, 0;
58, 32, 0; 44, 32, 0; 44, 39, -70; 44, 186, -70; 100, 121, 0; 100, 119, 0;
100, 118, 0; 100, 46, 0; 100, 100, 0; 100, 44, 0; 101, 121, -15;
101, 120, -15; 101, 119, -25; 101, 118, -25; 101, 46, 0; 101, 112, 0;
101, 103, -15; 101, 44, 0; 101, 98, 0; 102, 39, 55; 102, 186, 0; 102, 46, 0;
102, 111, 0; 102, 108, 0; 102, 105, -20; 102, 102, -25; 102, 101, 0;
102, 245, -50; 102, 44, 0; 102, 97, -10; 103, 121, 0; 103, 114, 0;
103, 46, 0; 103, 111, 0; 103, 105, 0; 103, 103, 0; 103, 101, 0; 103, 44, 0;
103, 97, -5; 104, 121, -5; 105, 118, -25; 107, 121, -15; 107, 111, -10;
107, 101, -10; 108, 121, 0; 108, 119, -10; 109, 121, 0; 109, 117, 0;
110, 121, -15; 110, 118, -40; 110, 117, 0; 111, 121, -10; 111, 120, 0;
111, 119, -25; 111, 118, -15; 111, 103, 0; 112, 121, -10; 46, 39, -70;
46, 186, -70; 170, 96, 0; 170, 65, -80; 186, 32, 0; 96, 96, -74; 96, 65, -80;
39, 118, -50; 39, 116, -18; 39, 32, -74; 39, 115, -55; 39, 114, -50;
39, 39, -74; 39, 186, 0; 39, 108, -10; 39, 100, -50; 114, 121, 0; 114, 118, 0;
114, 117, 0; 114, 116, 0; 114, 115, 0; 114, 114, 0; 114, 113, 0; 114, 46, -55;
114, 112, 0; 114, 111, 0; 114, 110, 0; 114, 109, 0; 114, 108, 0; 114, 107, 0;
114, 105, 0; 114, 45, -20; 114, 103, -18; 114, 101, 0; 114, 100, 0;
114, 44, -40; 114, 99, 0; 114, 97, 0; 115, 119, 0; 32, 96, 0; 32, 170, 0;
32, 89, -90; 32, 87, -30; 32, 86, -50; 32, 84, -18; 32, 65, -55; 118, 46, -65;

118, 111, -20; 118, 101, -15; 118, 44, -65; 118, 97, -25; 119, 46, -65;
119, 111, -10; 119, 104, 0; 119, 101, 0; 119, 44, -65; 119, 97, -10;
120, 101, -15; 121, 46, -65; 121, 111, 0; 121, 101, 0; 121, 44, -65;
121, 97, 0; 122, 111, 0; 122, 101, 0]

TimesBold

let *times_bold_widths* =
hashtable_of_dictionary
[32, 250; 33, 333; 34, 555; 35, 500; 36, 500; 37, 1000; 38, 833; 39, 333;
40, 333; 41, 333; 42, 500; 43, 570; 44, 250; 45, 333; 46, 250; 47, 278;
48, 500; 49, 500; 50, 500; 51, 500; 52, 500; 53, 500; 54, 500; 55, 500;
56, 500; 57, 500; 58, 333; 59, 333; 60, 570; 61, 570; 62, 570; 63, 500;
64, 930; 65, 722; 66, 667; 67, 722; 68, 722; 69, 667; 70, 611; 71, 778;
72, 778; 73, 389; 74, 500; 75, 778; 76, 667; 77, 944; 78, 722; 79, 778;
80, 611; 81, 778; 82, 722; 83, 556; 84, 667; 85, 722; 86, 722; 87, 1000;
88, 722; 89, 722; 90, 667; 91, 333; 92, 278; 93, 333; 94, 581; 95, 500;
96, 333; 97, 500; 98, 556; 99, 444; 100, 556; 101, 444; 102, 333; 103, 500;
104, 556; 105, 278; 106, 333; 107, 556; 108, 278; 109, 833; 110, 556;
111, 500; 112, 556; 113, 556; 114, 444; 115, 389; 116, 333; 117, 556;
118, 500; 119, 722; 120, 500; 121, 500; 122, 444; 123, 394; 124, 220;
125, 394; 126, 520; 161, 333; 162, 500; 163, 500; 164, 167; 165, 500;
166, 500; 167, 500; 168, 500; 169, 278; 170, 500; 171, 500; 172, 333;
173, 333; 174, 556; 175, 556; 177, 500; 178, 500; 179, 500; 180, 250;
182, 540; 183, 350; 184, 333; 185, 500; 186, 500; 187, 500; 188, 1000;
189, 1000; 191, 500; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 1000; 225, 1000; 227, 300; 232, 667; 233, 778; 234, 1000;
235, 330; 241, 722; 245, 278; 248, 278; 249, 500; 250, 722; 251, 556]

let *times_bold_kerns* =
hashtable_of_kerns
[65, 121, −74; 65, 119, −90; 65, 118, −100; 65, 117, −50; 65, 39, −74;
65, 186, 0; 65, 112, -25; 65, 89, -100; 65, 87, -130; 65, 86, -145;
65, 85, -50; 65, 84, -95; 65, 81, -45; 65, 79, -45; 65, 71, -55;
65, 67, -55; 66, 46, 0; 66, 44, 0; 66, 85, -10; 66, 65, -30; 68, 46, -20;
68, 44, 0; 68, 89, -40; 68, 87, -40; 68, 86, -40; 68, 65, -35; 70, 114, 0;
70, 46, -110; 70, 111, -25; 70, 105, 0; 70, 101, -25; 70, 44, -92;
70, 97, -25; 70, 65, -90; 71, 46, 0; 71, 44, 0; 74, 117, -15; 74, 46, -20;
74, 111, -15; 74, 101, -15; 74, 44, 0; 74, 97, -15; 74, 65, -30;
75, 121, -45; 75, 117, -15; 75, 111, -25; 75, 101, -25; 75, 79, -30;
76, 121, -55; 76, 39, -110; 76, 186, -20; 76, 89, -92; 76, 87, -92;
76, 86, -92; 76, 84, -92; 78, 46, 0; 78, 44, 0; 78, 65, -20; 79, 46, 0;
79, 44, 0; 79, 89, -50; 79, 88, -40; 79, 87, -50; 79, 86, -50; 79, 84, -40;
79, 65, -40; 80, 46, -110; 80, 111, -20; 80, 101, -20; 80, 44, -92;
80, 97, -10; 80, 65, -74; 81, 46, -20; 81, 44, 0; 81, 85, -10; 82, 89, -35;
82, 87, -35; 82, 86, -55; 82, 85, -30; 82, 84, -40; 82, 79, -30; 83, 46, 0;
83, 44, 0; 84, 121, -74; 84, 119, -74; 84, 117, -92; 84, 59, -74;
84, 114, -74; 84, 46, -90; 84, 111, -92; 84, 105, -18; 84, 45, -92;
84, 104, 0; 84, 101, -92; 84, 44, -74; 84, 58, -74; 84, 97, -92;
84, 79, -18; 84, 65, -90; 85, 46, -50; 85, 44, -50; 85, 65, -60;
86, 117, -92; 86, 59, -92; 86, 46, -145; 86, 111, -100; 86, 105, -37;

86, 45, -74; 86, 101, -100; 86, 44, -129; 86, 58, -92; 86, 97, -92;
86, 79, -45; 86, 71, -30; 86, 65, -135; 87, 121, -60; 87, 117, -50;
87, 59, -55; 87, 46, -92; 87, 111, -75; 87, 105, -18; 87, 45, -37;
87, 104, 0; 87, 101, -65; 87, 44, -92; 87, 58, -55; 87, 97, -65;
87, 79, -10; 87, 65, -120; 89, 117, -92; 89, 59, -92; 89, 46, -92;
89, 111, -111; 89, 105, -37; 89, 45, -92; 89, 101, -111; 89, 44, -92;
89, 58, -92; 89, 97, -85; 89, 79, -35; 89, 65, -110; 97, 121, 0; 97, 119, 0;
97, 118, -25; 97, 116, 0; 97, 112, 0; 97, 103, 0; 97, 98, 0; 98, 121, 0;
98, 118, -15; 98, 117, -20; 98, 46, -40; 98, 108, 0; 98, 44, 0; 98, 98, -10;
99, 121, 0; 99, 46, 0; 99, 108, 0; 99, 107, 0; 99, 104, 0; 99, 44, 0;
58, 32, 0; 44, 32, 0; 44, 39, -55; 44, 186, -45; 100, 121, 0; 100, 119, -15;
100, 118, 0; 100, 46, 0; 100, 100, 0; 100, 44, 0; 101, 121, 0; 101, 120, 0;
101, 119, 0; 101, 118, -15; 101, 46, 0; 101, 112, 0; 101, 103, 0; 101, 44, 0;
101, 98, 0; 102, 39, 55; 102, 186, 50; 102, 46, -15; 102, 111, -25;
102, 108, 0; 102, 105, -25; 102, 102, 0; 102, 101, 0; 102, 245, -35;
102, 44, -15; 102, 97, 0; 103, 121, 0; 103, 114, 0; 103, 46, -15; 103, 111, 0;
103, 105, 0; 103, 103, 0; 103, 101, 0; 103, 44, 0; 103, 97, 0; 104, 121, -15;
105, 118, -10; 107, 121, -15; 107, 111, -15; 107, 101, -10; 108, 121, 0;
108, 119, 0; 109, 121, 0; 109, 117, 0; 110, 121, 0; 110, 118, -40;
110, 117, 0; 111, 121, 0; 111, 120, 0; 111, 119, -10; 111, 118, -10;
111, 103, 0; 112, 121, 0; 46, 39, -55; 46, 186, -55; 170, 96, 0;
170, 65, -10; 186, 32, 0; 96, 96, -63; 96, 65, -10; 39, 118, -20;
39, 116, 0; 39, 32, -74; 39, 115, -37; 39, 114, -20; 39, 39, -63;
39, 186, 0; 39, 108, 0; 39, 100, -20; 114, 121, 0; 114, 118, -10;
114, 117, 0; 114, 116, 0; 114, 115, 0; 114, 114, 0; 114, 113, -18;
114, 46, -100; 114, 112, -10; 114, 111, -18; 114, 110, -15; 114, 109, 0;
114, 108, 0; 114, 107, 0; 114, 105, 0; 114, 45, -37; 114, 103, -10;
114, 101, -18; 114, 100, 0; 114, 44, -92; 114, 99, -18; 114, 97, 0;
115, 119, 0; 32, 96, 0; 32, 170, 0; 32, 89, -55; 32, 87, -30; 32, 86, -45;
32, 84, -30; 32, 65, -55; 118, 46, -70; 118, 111, -10; 118, 101, -10;
118, 44, -55; 118, 97, -10; 119, 46, -70; 119, 111, -10; 119, 104, 0;
119, 101, 0; 119, 44, -55; 119, 97, 0; 120, 101, 0; 121, 46, -70;
121, 111, -25; 121, 101, -10; 121, 44, -55; 121, 97, 0; 122, 111, 0;
122, 101, 0]

TimesItalic

let *times_italic_widths* =
hashtable_of_dictionary
[32, 250; 33, 333; 34, 420; 35, 500; 36, 500; 37, 833; 38, 778; 39, 333;
40, 333; 41, 333; 42, 500; 43, 675; 44, 250; 45, 333; 46, 250; 47, 278;
48, 500; 49, 500; 50, 500; 51, 500; 52, 500; 53, 500; 54, 500; 55, 500;
56, 500; 57, 500; 58, 333; 59, 333; 60, 675; 61, 675; 62, 675; 63, 500;
64, 920; 65, 611; 66, 611; 67, 667; 68, 722; 69, 611; 70, 611; 71, 722;
72, 722; 73, 333; 74, 444; 75, 667; 76, 556; 77, 833; 78, 667; 79, 722;
80, 611; 81, 722; 82, 611; 83, 500; 84, 556; 85, 722; 86, 611; 87, 833;
88, 611; 89, 556; 90, 556; 91, 389; 92, 278; 93, 389; 94, 422; 95, 500;
96, 333; 97, 500; 98, 500; 99, 444; 100, 500; 101, 444; 102, 278; 103, 500;
104, 500; 105, 278; 106, 278; 107, 444; 108, 278; 109, 722; 110, 500;
111, 500; 112, 500; 113, 500; 114, 389; 115, 389; 116, 278; 117, 500;
118, 444; 119, 667; 120, 444; 121, 444; 122, 389; 123, 400; 124, 275;

125, 400; 126, 541; 161, 389; 162, 500; 163, 500; 164, 167; 165, 500;
166, 500; 167, 500; 168, 500; 169, 214; 170, 556; 171, 500; 172, 333;
173, 333; 174, 500; 175, 500; 177, 500; 178, 500; 179, 500; 180, 250;
182, 523; 183, 350; 184, 333; 185, 556; 186, 556; 187, 500; 188, 889;
189, 1000; 191, 500; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 889; 225, 889; 227, 276; 232, 556; 233, 722; 234, 944;
235, 310; 241, 667; 245, 278; 248, 278; 249, 500; 250, 667; 251, 500]

let $times\_italic\_kerns$ =
$hashtable\_of\_kerns$
[65, 121, -55; 65, 119, -55; 65, 118, -55; 65, 117, -20; 65, 39, -37;
65, 186, 0; 65, 112, 0; 65, 89, -55; 65, 87, -95; 65, 86, -105; 65, 85, -50;
65, 84, -37; 65, 81, -40; 65, 79, -40; 65, 71, -35; 65, 67, -30; 66, 46, 0;
66, 44, 0; 66, 85, -10; 66, 65, -25; 68, 46, 0; 68, 44, 0; 68, 89, -40;
68, 87, -40; 68, 86, -40; 68, 65, -35; 70, 114, -55; 70, 46, -135;
70, 111, -105; 70, 105, -45; 70, 101, -75; 70, 44, -135; 70, 97, -75;
70, 65, -115; 71, 46, 0; 71, 44, 0; 74, 117, -35; 74, 46, -25; 74, 111, -25;
74, 101, -25; 74, 44, -25; 74, 97, -35; 74, 65, -40; 75, 121, -40;
75, 117, -40; 75, 111, -40; 75, 101, -35; 75, 79, -50; 76, 121, -30;
76, 39, -37; 76, 186, 0; 76, 89, -20; 76, 87, -55; 76, 86, -55; 76, 84, -20;
78, 46, 0; 78, 44, 0; 78, 65, -27; 79, 46, 0; 79, 44, 0; 79, 89, -50;
79, 88, -40; 79, 87, -50; 79, 86, -50; 79, 84, -40; 79, 65, -55;
80, 46, -135; 80, 111, -80; 80, 101, -80; 80, 44, -135; 80, 97, -80;
80, 65, -90; 81, 46, 0; 81, 44, 0; 81, 85, -10; 82, 89, -18; 82, 87, -18;
82, 86, -18; 82, 85, -40; 82, 84, 0; 82, 79, -40; 83, 46, 0; 83, 44, 0;
84, 121, -74; 84, 119, -74; 84, 117, -55; 84, 59, -65; 84, 114, -55;
84, 46, -74; 84, 111, -92; 84, 105, -55; 84, 45, -74; 84, 104, 0;
84, 101, -92; 84, 44, -74; 84, 58, -55; 84, 97, -92; 84, 79, -18;
84, 65, -50; 85, 46, -25; 85, 44, -25; 85, 65, -40; 86, 117, -74;
86, 59, -74; 86, 46, -129; 86, 111, -111; 86, 105, -74; 86, 45, -55;
86, 101, -111; 86, 44, -129; 86, 58, -65; 86, 97, -111; 86, 79, -30;
86, 71, 0; 86, 65, -60; 87, 121, -70; 87, 117, -55; 87, 59, -65;
87, 46, -92; 87, 111, -92; 87, 105, -55; 87, 45, -37; 87, 104, 0;
87, 101, -92; 87, 44, -92; 87, 58, -65; 87, 97, -92; 87, 79, -25;
87, 65, -60; 89, 117, -92; 89, 59, -65; 89, 46, -92; 89, 111, -92;
89, 105, -74; 89, 45, -74; 89, 101, -92; 89, 44, -92; 89, 58, -65;
89, 97, -92; 89, 79, -15; 89, 65, -50; 97, 121, 0; 97, 119, 0; 97, 118, 0;
97, 116, 0; 97, 112, 0; 97, 103, -10; 97, 98, 0; 98, 121, 0; 98, 118, 0;
98, 117, -20; 98, 46, -40; 98, 108, 0; 98, 44, 0; 98, 98, 0; 99, 121, 0;
99, 46, 0; 99, 108, 0; 99, 107, -20; 99, 104, -15; 99, 44, 0; 58, 32, 0;
44, 32, 0; 44, 39, -140; 44, 186, -140; 100, 121, 0; 100, 119, 0;
100, 118, 0; 100, 46, 0; 100, 100, 0; 100, 44, 0; 101, 121, -30;
101, 120, -20; 101, 119, -15; 101, 118, -15; 101, 46, -15; 101, 112, 0;
101, 103, -40; 101, 44, -10; 101, 98, 0; 102, 39, 92; 102, 186, 0;
102, 46, -15; 102, 111, 0; 102, 108, 0; 102, 105, -20; 102, 102, -18;
102, 101, 0; 102, 245, -60; 102, 44, -10; 102, 97, 0; 103, 121, 0;
103, 114, 0; 103, 46, -15; 103, 111, 0; 103, 105, 0; 103, 103, -10;
103, 101, -10; 103, 44, -10; 103, 97, 0; 104, 121, 0; 105, 118, 0;
107, 121, -10; 107, 111, -10; 107, 101, -10; 108, 121, 0; 108, 119, 0;

109, 121, 0; 109, 117, 0; 110, 121, 0; 110, 118, -40; 110, 117, 0;
111, 121, 0; 111, 120, 0; 111, 119, 0; 111, 118, -10; 111, 103, -10;
112, 121, 0; 46, 39, -140; 46, 186, -140; 170, 96, 0; 170, 65, 0;
186, 32, 0; 96, 96, -111; 96, 65, 0; 39, 118, -10; 39, 116, -30;
39, 32, -111; 39, 115, -40; 39, 114, -25; 39, 39, -111; 39, 186, 0;
39, 108, 0; 39, 100, -25; 114, 121, 0; 114, 118, 0; 114, 117, 0;
114, 116, 0; 114, 115, -10; 114, 114, 0; 114, 113, -37; 114, 46, -111;
114, 112, 0; 114, 111, -45; 114, 110, 0; 114, 109, 0; 114, 108, 0;
114, 107, 0; 114, 105, 0; 114, 45, -20; 114, 103, -37; 114, 101, -37;
114, 100, -37; 114, 44, -111; 114, 99, -37; 114, 97, -15; 115, 119, 0;
32, 96, 0; 32, 170, 0; 32, 89, -75; 32, 87, -40; 32, 86, -35; 32, 84, -18;
32, 65, -18; 118, 46, -74; 118, 111, 0; 118, 101, 0; 118, 44, -74;
118, 97, 0; 119, 46, -74; 119, 111, 0; 119, 104, 0; 119, 101, 0;
119, 44, -74; 119, 97, 0; 120, 101, 0; 121, 46, -55; 121, 111, 0;
121, 101, 0; 121, 44, -55; 121, 97, 0; 122, 111, 0; 122, 101, 0]

TimesBoldItalic

let $times\_bold\_italic\_widths$ =
$hashtable\_of\_dictionary$
[32, 250; 33, 389; 34, 555; 35, 500; 36, 500; 37, 833; 38, 778; 39, 333;
40, 333; 41, 333; 42, 500; 43, 570; 44, 250; 45, 333; 46, 250; 47, 278;
48, 500; 49, 500; 50, 500; 51, 500; 52, 500; 53, 500; 54, 500; 55, 500;
56, 500; 57, 500; 58, 333; 59, 333; 60, 570; 61, 570; 62, 570; 63, 500;
64, 832; 65, 667; 66, 667; 67, 667; 68, 722; 69, 667; 70, 667; 71, 722;
72, 778; 73, 389; 74, 500; 75, 667; 76, 611; 77, 889; 78, 722; 79, 722;
80, 611; 81, 722; 82, 667; 83, 556; 84, 611; 85, 722; 86, 667; 87, 889;
88, 667; 89, 611; 90, 611; 91, 333; 92, 278; 93, 333; 94, 570; 95, 500;
96, 333; 97, 500; 98, 500; 99, 444; 100, 500; 101, 444; 102, 333; 103, 500;
104, 556; 105, 278; 106, 278; 107, 500; 108, 278; 109, 778; 110, 556;
111, 500; 112, 500; 113, 500; 114, 389; 115, 389; 116, 278; 117, 556;
118, 444; 119, 667; 120, 500; 121, 444; 122, 389; 123, 348; 124, 220;
125, 348; 126, 570; 161, 389; 162, 500; 163, 500; 164, 167; 165, 500;
166, 500; 167, 500; 168, 500; 169, 278; 170, 500; 171, 500; 172, 333;
173, 333; 174, 556; 175, 556; 177, 500; 178, 500; 179, 500; 180, 250;
182, 500; 183, 350; 184, 333; 185, 500; 186, 500; 187, 500; 188, 1000;
189, 1000; 191, 500; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 1000; 225, 944; 227, 266; 232, 611; 233, 722; 234, 944;
235, 300; 241, 722; 245, 278; 248, 278; 249, 500; 250, 722; 251, 500]

let $times\_bold\_italic\_kerns$ =
$hashtable\_of\_kerns$
[65, 121, −74; 65, 119, −74; 65, 118, −74; 65, 117, −30; 65, 39, −74;
65, 186, 0; 65, 112, 0; 65, 89, -70; 65, 87, -100; 65, 86, -95; 65, 85, -50;
65, 84, -55; 65, 81, -55; 65, 79, -50; 65, 71, -60; 65, 67, -65; 66, 46, 0;
66, 44, 0; 66, 85, -10; 66, 65, -25; 68, 46, 0; 68, 44, 0; 68, 89, -50;
68, 87, -40; 68, 86, -50; 68, 65, -25; 70, 114, -50; 70, 46, -129;
70, 111, -70; 70, 105, -40; 70, 101, -100; 70, 44, -129; 70, 97, -95;
70, 65, -100; 71, 46, 0; 71, 44, 0; 74, 117, -40; 74, 46, -10; 74, 111, -40;
74, 101, -40; 74, 44, -10; 74, 97, -40; 74, 65, -25; 75, 121, -20;
75, 117, -20; 75, 111, -25; 75, 101, -25; 75, 79, -30; 76, 121, -37;

76, 39, -55; 76, 186, 0; 76, 89, -37; 76, 87, -37; 76, 86, -37; 76, 84, -18;
78, 46, 0; 78, 44, 0; 78, 65, -30; 79, 46, 0; 79, 44, 0; 79, 89, -50;
79, 88, -40; 79, 87, -50; 79, 86, -50; 79, 84, -40; 79, 65, -40;
80, 46, -129; 80, 111, -55; 80, 101, -50; 80, 44, -129; 80, 97, -40;
80, 65, -85; 81, 46, 0; 81, 44, 0; 81, 85, -10; 82, 89, -18; 82, 87, -18;
82, 86, -18; 82, 85, -40; 82, 84, -30; 82, 79, -40; 83, 46, 0; 83, 44, 0;
84, 121, -37; 84, 119, -37; 84, 117, -37; 84, 59, -74; 84, 114, -37;
84, 46, -92; 84, 111, -95; 84, 105, -37; 84, 45, -92; 84, 104, 0;
84, 101, -92; 84, 44, -92; 84, 58, -74; 84, 97, -92; 84, 79, -18;
84, 65, -55; 85, 46, 0; 85, 44, 0; 85, 65, -45; 86, 117, -55; 86, 59, -74;
86, 46, -129; 86, 111, -111; 86, 105, -55; 86, 45, -70; 86, 101, -111;
86, 44, -129; 86, 58, -74; 86, 97, -111; 86, 79, -30; 86, 71, -10;
86, 65, -85; 87, 121, -55; 87, 117, -55; 87, 59, -55; 87, 46, -74;
87, 111, -80; 87, 105, -37; 87, 45, -50; 87, 104, 0; 87, 101, -90;
87, 44, -74; 87, 58, -55; 87, 97, -85; 87, 79, -15; 87, 65, -74;
89, 117, -92; 89, 59, -92; 89, 46, -74; 89, 111, -111; 89, 105, -55;
89, 45, -92; 89, 101, -111; 89, 44, -92; 89, 58, -92; 89, 97, -92;
89, 79, -25; 89, 65, -74; 97, 121, 0; 97, 119, 0; 97, 118, 0; 97, 116, 0;
97, 112, 0; 97, 103, 0; 97, 98, 0; 98, 121, 0; 98, 118, 0; 98, 117, -20;
98, 46, -40; 98, 108, 0; 98, 44, 0; 98, 98, -10; 99, 121, 0; 99, 46, 0;
99, 108, 0; 99, 107, -10; 99, 104, -10; 99, 44, 0; 58, 32, 0; 44, 32, 0;
44, 39, -95; 44, 186, -95; 100, 121, 0; 100, 119, 0; 100, 118, 0;
100, 46, 0; 100, 100, 0; 100, 44, 0; 101, 121, 0; 101, 120, 0; 101, 119, 0;
101, 118, 0; 101, 46, 0; 101, 112, 0; 101, 103, 0; 101, 44, 0; 101, 98, -10;
102, 39, 55; 102, 186, 0; 102, 46, -10; 102, 111, -10; 102, 108, 0;
102, 105, 0; 102, 102, -18; 102, 101, -10; 102, 245, -30; 102, 44, -10;
102, 97, 0; 103, 121, 0; 103, 114, 0; 103, 46, 0; 103, 111, 0; 103, 105, 0;
103, 103, 0; 103, 101, 0; 103, 44, 0; 103, 97, 0; 104, 121, 0; 105, 118, 0;
107, 121, 0; 107, 111, -10; 107, 101, -30; 108, 121, 0; 108, 119, 0;
109, 121, 0; 109, 117, 0; 110, 121, 0; 110, 118, -40; 110, 117, 0;
111, 121, -10; 111, 120, -10; 111, 119, -25; 111, 118, -15; 111, 103, 0;
112, 121, 0; 46, 39, -95; 46, 186, -95; 170, 96, 0; 170, 65, 0; 186, 32, 0;
96, 96, -74; 96, 65, 0; 39, 118, -15; 39, 116, -37; 39, 32, -74;
39, 115, -74; 39, 114, -15; 39, 39, -74; 39, 186, 0; 39, 108, 0;
39, 100, -15; 114, 121, 0; 114, 118, 0; 114, 117, 0; 114, 116, 0;
114, 115, 0; 114, 114, 0; 114, 113, 0; 114, 46, -65; 114, 112, 0;
114, 111, 0; 114, 110, 0; 114, 109, 0; 114, 108, 0; 114, 107, 0;
114, 105, 0; 114, 45, 0; 114, 103, 0; 114, 101, 0; 114, 100, 0; 114, 44, -65;
114, 99, 0; 114, 97, 0; 115, 119, 0; 32, 96, 0; 32, 170, 0; 32, 89, -70;
32, 87, -70; 32, 86, -70; 32, 84, 0; 32, 65, -37; 118, 46, -37;
118, 111, -15; 118, 101, -15; 118, 44, -37; 118, 97, 0; 119, 46, -37;
119, 111, -15; 119, 104, 0; 119, 101, -10; 119, 44, -37; 119, 97, -10;
120, 101, -10; 121, 46, -37; 121, 111, 0; 121, 101, 0; 121, 44, -37;
121, 97, 0; 122, 111, 0; 122, 101, 0]

Helvetica

let *helvetica_widths* =
hashtable_of_dictionary
[32, 278; 33, 278; 34, 355; 35, 556; 36, 556; 37, 889; 38, 667; 39, 222;
40, 333; 41, 333; 42, 389; 43, 584; 44, 278; 45, 333; 46, 278; 47, 278;

```
  48, 556; 49, 556; 50, 556; 51, 556; 52, 556; 53, 556; 54, 556; 55, 556;
  56, 556; 57, 556; 58, 278; 59, 278; 60, 584; 61, 584; 62, 584; 63, 556;
  64, 1015; 65, 667; 66, 667; 67, 722; 68, 722; 69, 667; 70, 611; 71, 778;
  72, 722; 73, 278; 74, 500; 75, 667; 76, 556; 77, 833; 78, 722; 79, 778;
  80, 667; 81, 778; 82, 722; 83, 667; 84, 611; 85, 722; 86, 667; 87, 944;
  88, 667; 89, 667; 90, 611; 91, 278; 92, 278; 93, 278; 94, 469; 95, 556;
  96, 222; 97, 556; 98, 556; 99, 500; 100, 556; 101, 556; 102, 278; 103, 556;
  104, 556; 105, 222; 106, 222; 107, 500; 108, 222; 109, 833; 110, 556;
  111, 556; 112, 556; 113, 556; 114, 333; 115, 500; 116, 278; 117, 556;
  118, 500; 119, 722; 120, 500; 121, 500; 122, 500; 123, 334; 124, 260;
  125, 334; 126, 584; 161, 333; 162, 556; 163, 556; 164, 167; 165, 556;
  166, 556; 167, 556; 168, 556; 169, 191; 170, 333; 171, 556; 172, 333;
  173, 333; 174, 500; 175, 500; 177, 556; 178, 556; 179, 556; 180, 278;
  182, 537; 183, 350; 184, 222; 185, 333; 186, 333; 187, 556; 188, 1000;
  189, 1000; 191, 611; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
  198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
  207, 333; 208, 1000; 225, 1000; 227, 370; 232, 556; 233, 778; 234, 1000;
  235, 365; 241, 889; 245, 278; 248, 222; 249, 611; 250, 944; 251, 611]

let helvetica_kerns =
 hashtable_of_kerns
 [65, 121, -40; 65, 119, -40; 65, 118, -40; 65, 117, -30; 65, 89, -100;
  65, 87, -50; 65, 86, -70; 65, 85, -50; 65, 84, -120; 65, 81, -30;
  65, 79, -30; 65, 71, -30; 65, 67, -30; 66, 46, -20; 66, 44, -20;
  66, 85, -10; 67, 46, -30; 67, 44, -30; 68, 46, -70; 68, 44, -70;
  68, 89, -90; 68, 87, -40; 68, 86, -70; 68, 65, -40; 70, 114, -45;
  70, 46, -150; 70, 111, -30; 70, 101, -30; 70, 44, -150; 70, 97, -50;
  70, 65, -80; 74, 117, -20; 74, 46, -30; 74, 44, -30; 74, 97, -20;
  74, 65, -20; 75, 121, -50; 75, 117, -30; 75, 111, -40; 75, 101, -40;
  75, 79, -50; 76, 121, -30; 76, 39, -160; 76, 186, -140; 76, 89, -140;
  76, 87, -70; 76, 86, -110; 76, 84, -110; 79, 46, -40; 79, 44, -40;
  79, 89, -70; 79, 88, -60; 79, 87, -30; 79, 86, -50; 79, 84, -40;
  79, 65, -20; 80, 46, -180; 80, 111, -50; 80, 101, -50; 80, 44, -180;
  80, 97, -40; 80, 65, -120; 81, 85, -10; 82, 89, -50; 82, 87, -30;
  82, 86, -50; 82, 85, -40; 82, 84, -30; 82, 79, -20; 83, 46, -20;
  83, 44, -20; 84, 121, -120; 84, 119, -120; 84, 117, -120; 84, 59, -20;
  84, 114, -120; 84, 46, -120; 84, 111, -120; 84, 45, -140; 84, 101, -120;
  84, 44, -120; 84, 58, -20; 84, 97, -120; 84, 79, -40; 84, 65, -120;
  85, 46, -40; 85, 44, -40; 85, 65, -40; 86, 117, -70; 86, 59, -40;
  86, 46, -125; 86, 111, -80; 86, 45, -80; 86, 101, -80; 86, 44, -125;
  86, 58, -40; 86, 97, -70; 86, 79, -40; 86, 71, -40; 86, 65, -80;
  87, 121, -20; 87, 117, -30; 87, 46, -80; 87, 111, -30; 87, 45, -40;
  87, 101, -30; 87, 44, -80; 87, 97, -40; 87, 79, -20; 87, 65, -50;
  89, 117, -110; 89, 59, -60; 89, 46, -140; 89, 111, -140; 89, 105, -20;
  89, 45, -140; 89, 101, -140; 89, 44, -140; 89, 58, -60; 89, 97, -140;
  89, 79, -85; 89, 65, -110; 97, 121, -30; 97, 119, -20; 97, 118, -20;
  98, 121, -20; 98, 118, -20; 98, 117, -20; 98, 46, -40; 98, 108, -20;
  98, 44, -40; 98, 98, -10; 99, 107, -20; 99, 44, -15; 58, 32, -50;
  44, 39, -100; 44, 186, -100; 101, 121, -20; 101, 120, -30; 101, 119, -20;
  101, 118, -30; 101, 46, -15; 101, 44, -15; 102, 39, 50; 102, 186, 60;
```

102, 46, -30; 102, 111, -30; 102, 101, -30; 102, 245, -28; 102, 44, -30;
102, 97, -30; 103, 114, -10; 104, 121, -30; 107, 111, -20; 107, 101, -20;
109, 121, -15; 109, 117, -10; 110, 121, -15; 110, 118, -20; 110, 117, -10;
111, 121, -30; 111, 120, -30; 111, 119, -15; 111, 118, -15; 111, 46, -40;
111, 44, -40; 249, 122, -55; 249, 121, -70; 249, 120, -85; 249, 119, -70;
249, 118, -70; 249, 117, -55; 249, 116, -55; 249, 115, -55; 249, 114, -55;
249, 113, -55; 249, 46, -95; 249, 112, -55; 249, 111, -55; 249, 110, -55;
249, 109, -55; 249, 108, -55; 249, 107, -55; 249, 106, -55; 249, 105, -55;
249, 104, -55; 249, 103, -55; 249, 102, -55; 249, 101, -55; 249, 100, -55;
249, 44, -95; 249, 99, -55; 249, 98, -55; 249, 97, -55; 112, 121, -30;
112, 46, -35; 112, 44, -35; 46, 32, -60; 46, 39, -100; 46, 186, -100;
186, 32, -40; 96, 96, -57; 39, 32, -70; 39, 115, -50; 39, 114, -50;
39, 39, -57; 39, 100, -50; 114, 121, 30; 114, 118, 30; 114, 117, 15;
114, 116, 40; 114, 59, 30; 114, 46, -50; 114, 112, 30; 114, 110, 25;
114, 109, 25; 114, 108, 15; 114, 107, 15; 114, 105, 15; 114, 44, -50;
114, 58, 30; 114, 97, -10; 115, 119, -30; 115, 46, -15; 115, 44, -15;
59, 32, -50; 32, 96, -60; 32, 170, -30; 32, 89, -90; 32, 87, -40;
32, 86, -50; 32, 84, -50; 118, 46, -80; 118, 111, -25; 118, 101, -25;
118, 44, -80; 118, 97, -25; 119, 46, -60; 119, 111, -10; 119, 101, -10;
119, 44, -60; 119, 97, -15; 120, 101, -30; 121, 46, -100; 121, 111, -20;
121, 101, -20; 121, 44, -100; 121, 97, -20; 122, 111, -15; 122, 101, -15]

HelveticaBold

let $helvetica\_bold\_widths$ =
$hashtable\_of\_dictionary$
[32, 278; 33, 333; 34, 474; 35, 556; 36, 556; 37, 889; 38, 722; 39, 278;
40, 333; 41, 333; 42, 389; 43, 584; 44, 278; 45, 333; 46, 278; 47, 278;
48, 556; 49, 556; 50, 556; 51, 556; 52, 556; 53, 556; 54, 556; 55, 556;
56, 556; 57, 556; 58, 333; 59, 333; 60, 584; 61, 584; 62, 584; 63, 611;
64, 975; 65, 722; 66, 722; 67, 722; 68, 722; 69, 667; 70, 611; 71, 778;
72, 722; 73, 278; 74, 556; 75, 722; 76, 611; 77, 833; 78, 722; 79, 778;
80, 667; 81, 778; 82, 722; 83, 667; 84, 611; 85, 722; 86, 667; 87, 944;
88, 667; 89, 667; 90, 611; 91, 333; 92, 278; 93, 333; 94, 584; 95, 556;
96, 278; 97, 556; 98, 611; 99, 556; 100, 611; 101, 556; 102, 333; 103, 611;
104, 611; 105, 278; 106, 278; 107, 556; 108, 278; 109, 889; 110, 611;
111, 611; 112, 611; 113, 611; 114, 389; 115, 556; 116, 333; 117, 611;
118, 556; 119, 778; 120, 556; 121, 556; 122, 500; 123, 389; 124, 280;
125, 389; 126, 584; 161, 333; 162, 556; 163, 556; 164, 167; 165, 556;
166, 556; 167, 556; 168, 556; 169, 238; 170, 500; 171, 556; 172, 333;
173, 333; 174, 611; 175, 611; 177, 556; 178, 556; 179, 556; 180, 278;
182, 556; 183, 350; 184, 278; 185, 500; 186, 500; 187, 556; 188, 1000;
189, 1000; 191, 611; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 1000; 225, 1000; 227, 370; 232, 611; 233, 778; 234, 1000;
235, 365; 241, 889; 245, 278; 248, 278; 249, 611; 250, 944; 251, 611]

let $helvetica\_bold\_kerns$ =
$hashtable\_of\_kerns$
[65, 121, −30; 65, 119, −30; 65, 118, −40; 65, 117, −30; 65, 89, −110;
65, 87, -60; 65, 86, -80; 65, 85, -50; 65, 84, -90; 65, 81, -40;
65, 79, -40; 65, 71, -50; 65, 67, -40; 66, 85, -10; 66, 65, -30;

```
 68, 46, -30; 68, 44, -30; 68, 89, -70; 68, 87, -40; 68, 86, -40;
 68, 65, -40; 70, 46, -100; 70, 44, -100; 70, 97, -20; 70, 65, -80;
 74, 117, -20; 74, 46, -20; 74, 44, -20; 74, 65, -20; 75, 121, -40;
 75, 117, -30; 75, 111, -35; 75, 101, -15; 75, 79, -30; 76, 121, -30;
 76, 39, -140; 76, 186, -140; 76, 89, -120; 76, 87, -80; 76, 86, -110;
 76, 84, -90; 79, 46, -40; 79, 44, -40; 79, 89, -70; 79, 88, -50;
 79, 87, -50; 79, 86, -50; 79, 84, -40; 79, 65, -50; 80, 46, -120;
 80, 111, -40; 80, 101, -30; 80, 44, -120; 80, 97, -30; 80, 65, -100;
 81, 46, 20; 81, 44, 20; 81, 85, -10; 82, 89, -50; 82, 87, -40; 82, 86, -50;
 82, 85, -20; 82, 84, -20; 82, 79, -20; 84, 121, -60; 84, 119, -60;
 84, 117, -90; 84, 59, -40; 84, 114, -80; 84, 46, -80; 84, 111, -80;
 84, 45, -120; 84, 101, -60; 84, 44, -80; 84, 58, -40; 84, 97, -80;
 84, 79, -40; 84, 65, -90; 85, 46, -30; 85, 44, -30; 85, 65, -50;
 86, 117, -60; 86, 59, -40; 86, 46, -120; 86, 111, -90; 86, 45, -80;
 86, 101, -50; 86, 44, -120; 86, 58, -40; 86, 97, -60; 86, 79, -50;
 86, 71, -50; 86, 65, -80; 87, 121, -20; 87, 117, -45; 87, 59, -10;
 87, 46, -80; 87, 111, -60; 87, 45, -40; 87, 101, -35; 87, 44, -80;
 87, 58, -10; 87, 97, -40; 87, 79, -20; 87, 65, -60; 89, 117, -100;
 89, 59, -50; 89, 46, -100; 89, 111, -100; 89, 101, -80; 89, 44, -100;
 89, 58, -50; 89, 97, -90; 89, 79, -70; 89, 65, -110; 97, 121, -20;
 97, 119, -15; 97, 118, -15; 97, 103, -10; 98, 121, -20; 98, 118, -20;
 98, 117, -20; 98, 108, -10; 99, 121, -10; 99, 108, -20; 99, 107, -20;
 99, 104, -10; 58, 32, -40; 44, 32, -40; 44, 39, -120; 44, 186, -120;
 100, 121, -15; 100, 119, -15; 100, 118, -15; 100, 100, -10; 101, 121, -15;
 101, 120, -15; 101, 119, -15; 101, 118, -15; 101, 46, 20; 101, 44, 10;
 102, 39, 30; 102, 186, 30; 102, 46, -10; 102, 111, -20; 102, 101, -10;
 102, 44, -10; 103, 103, -10; 103, 101, 10; 104, 121, -20; 107, 111, -15;
 108, 121, -15; 108, 119, -15; 109, 121, -30; 109, 117, -20; 110, 121, -20;
 110, 118, -40; 110, 117, -10; 111, 121, -20; 111, 120, -30; 111, 119, -15;
 111, 118, -20; 112, 121, -15; 46, 32, -40; 46, 39, -120; 46, 186, -120;
 186, 32, -80; 96, 96, -46; 39, 118, -20; 39, 32, -80; 39, 115, -60;
 39, 114, -40; 39, 39, -46; 39, 108, -20; 39, 100, -80; 114, 121, 10;
 114, 118, 10; 114, 116, 20; 114, 115, -15; 114, 113, -20; 114, 46, -60;
 114, 111, -20; 114, 45, -20; 114, 103, -15; 114, 100, -20; 114, 44, -60;
 114, 99, -20; 115, 119, -15; 59, 32, -40; 32, 96, -60; 32, 170, -80;
 32, 89, -120; 32, 87, -80; 32, 86, -80; 32, 84, -100; 118, 46, -80;
 118, 111, -30; 118, 44, -80; 118, 97, -20; 119, 46, -40; 119, 111, -20;
 119, 44, -40; 120, 101, -10; 121, 46, -80; 121, 111, -25; 121, 101, -10;
 121, 44, -80; 121, 97, -30; 122, 101, 10]
```

HelveticaOblique

```
let helvetica_oblique_widths =
  hashtable_of_dictionary
  [32, 278; 33, 278; 34, 355; 35, 556; 36, 556; 37, 889; 38, 667; 39, 222;
   40, 333; 41, 333; 42, 389; 43, 584; 44, 278; 45, 333; 46, 278; 47, 278;
   48, 556; 49, 556; 50, 556; 51, 556; 52, 556; 53, 556; 54, 556; 55, 556;
   56, 556; 57, 556; 58, 278; 59, 278; 60, 584; 61, 584; 62, 584; 63, 556;
   64, 1015; 65, 667; 66, 667; 67, 722; 68, 722; 69, 667; 70, 611; 71, 778;
   72, 722; 73, 278; 74, 500; 75, 667; 76, 556; 77, 833; 78, 722; 79, 778;
   80, 667; 81, 778; 82, 722; 83, 667; 84, 611; 85, 722; 86, 667; 87, 944;
```

88, 667; 89, 667; 90, 611; 91, 278; 92, 278; 93, 278; 94, 469; 95, 556;
96, 222; 97, 556; 98, 556; 99, 500; 100, 556; 101, 556; 102, 278; 103, 556;
104, 556; 105, 222; 106, 222; 107, 500; 108, 222; 109, 833; 110, 556;
111, 556; 112, 556; 113, 556; 114, 333; 115, 500; 116, 278; 117, 556;
118, 500; 119, 722; 120, 500; 121, 500; 122, 500; 123, 334; 124, 260;
125, 334; 126, 584; 161, 333; 162, 556; 163, 556; 164, 167; 165, 556;
166, 556; 167, 556; 168, 556; 169, 191; 170, 333; 171, 556; 172, 333;
173, 333; 174, 500; 175, 500; 177, 556; 178, 556; 179, 556; 180, 278;
182, 537; 183, 350; 184, 222; 185, 333; 186, 333; 187, 556; 188, 1000;
189, 1000; 191, 611; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 1000; 225, 1000; 227, 370; 232, 556; 233, 778; 234, 1000;
235, 365; 241, 889; 245, 278; 248, 222; 249, 611; 250, 944; 251, 611]

let $helvetica\_oblique\_kerns\ =$
$hashtable\_of\_kerns$
$[65,\ 121,\ -40;\ 65,\ 119,\ -40;\ 65,\ 118,\ -40;\ 65,\ 117,\ -30;\ 65,\ 89,\ -100;$
65, 87, -50; 65, 86, -70; 65, 85, -50; 65, 84, -120; 65, 81, -30;
65, 79, -30; 65, 71, -30; 65, 67, -30; 66, 46, -20; 66, 44, -20;
66, 85, -10; 67, 46, -30; 67, 44, -30; 68, 46, -70; 68, 44, -70;
68, 89, -90; 68, 87, -40; 68, 86, -70; 68, 65, -40; 70, 114, -45;
70, 46, -150; 70, 111, -30; 70, 101, -30; 70, 44, -150; 70, 97, -50;
70, 65, -80; 74, 117, -20; 74, 46, -30; 74, 44, -30; 74, 97, -20;
74, 65, -20; 75, 121, -50; 75, 117, -30; 75, 111, -40; 75, 101, -40;
75, 79, -50; 76, 121, -30; 76, 39, -160; 76, 186, -140; 76, 89, -140;
76, 87, -70; 76, 86, -110; 76, 84, -110; 79, 46, -40; 79, 44, -40;
79, 89, -70; 79, 88, -60; 79, 87, -30; 79, 86, -50; 79, 84, -40;
79, 65, -20; 80, 46, -180; 80, 111, -50; 80, 101, -50; 80, 44, -180;
80, 97, -40; 80, 65, -120; 81, 85, -10; 82, 89, -50; 82, 87, -30;
82, 86, -50; 82, 85, -40; 82, 84, -30; 82, 79, -20; 83, 46, -20;
83, 44, -20; 84, 121, -120; 84, 119, -120; 84, 117, -120; 84, 59, -20;
84, 114, -120; 84, 46, -120; 84, 111, -120; 84, 45, -140; 84, 101, -120;
84, 44, -120; 84, 58, -20; 84, 97, -120; 84, 79, -40; 84, 65, -120;
85, 46, -40; 85, 44, -40; 85, 65, -40; 86, 117, -70; 86, 59, -40;
86, 46, -125; 86, 111, -80; 86, 45, -80; 86, 101, -80; 86, 44, -125;
86, 58, -40; 86, 97, -70; 86, 79, -40; 86, 71, -40; 86, 65, -80;
87, 121, -20; 87, 117, -30; 87, 46, -80; 87, 111, -30; 87, 45, -40;
87, 101, -30; 87, 44, -80; 87, 97, -40; 87, 79, -20; 87, 65, -50;
89, 117, -110; 89, 59, -60; 89, 46, -140; 89, 111, -140; 89, 105, -20;
89, 45, -140; 89, 101, -140; 89, 44, -140; 89, 58, -60; 89, 97, -140;
89, 79, -85; 89, 65, -110; 97, 121, -30; 97, 119, -20; 97, 118, -20;
98, 121, -20; 98, 118, -20; 98, 117, -20; 98, 46, -40; 98, 108, -20;
98, 44, -40; 98, 98, -10; 99, 107, -20; 99, 44, -15; 58, 32, -50;
44, 39, -100; 44, 186, -100; 101, 121, -20; 101, 120, -30; 101, 119, -20;
101, 118, -30; 101, 46, -15; 101, 44, -15; 102, 39, 50; 102, 186, 60;
102, 46, -30; 102, 111, -30; 102, 101, -30; 102, 245, -28; 102, 44, -30;
102, 97, -30; 103, 114, -10; 104, 121, -30; 107, 111, -20; 107, 101, -20;
109, 121, -15; 109, 117, -10; 110, 121, -15; 110, 118, -20; 110, 117, -10;
111, 121, -30; 111, 120, -30; 111, 119, -15; 111, 118, -15; 111, 46, -40;
111, 44, -40; 249, 122, -55; 249, 121, -70; 249, 120, -85; 249, 119, -70;

249, 118, -70; 249, 117, -55; 249, 116, -55; 249, 115, -55; 249, 114, -55;
249, 113, -55; 249, 46, -95; 249, 112, -55; 249, 111, -55; 249, 110, -55;
249, 109, -55; 249, 108, -55; 249, 107, -55; 249, 106, -55; 249, 105, -55;
249, 104, -55; 249, 103, -55; 249, 102, -55; 249, 101, -55; 249, 100, -55;
249, 44, -95; 249, 99, -55; 249, 98, -55; 249, 97, -55; 112, 121, -30;
112, 46, -35; 112, 44, -35; 46, 32, -60; 46, 39, -100; 46, 186, -100;
186, 32, -40; 96, 96, -57; 39, 32, -70; 39, 115, -50; 39, 114, -50;
39, 39, -57; 39, 100, -50; 114, 121, 30; 114, 118, 30; 114, 117, 15;
114, 116, 40; 114, 59, 30; 114, 46, -50; 114, 112, 30; 114, 110, 25;
114, 109, 25; 114, 108, 15; 114, 107, 15; 114, 105, 15; 114, 44, -50;
114, 58, 30; 114, 97, -10; 115, 119, -30; 115, 46, -15; 115, 44, -15;
59, 32, -50; 32, 96, -60; 32, 170, -30; 32, 89, -90; 32, 87, -40;
32, 86, -50; 32, 84, -50; 118, 46, -80; 118, 111, -25; 118, 101, -25;
118, 44, -80; 118, 97, -25; 119, 46, -60; 119, 111, -10; 119, 101, -10;
119, 44, -60; 119, 97, -15; 120, 101, -30; 121, 46, -100; 121, 111, -20;
121, 101, -20; 121, 44, -100; 121, 97, -20; 122, 111, -15; 122, 101, -15]

HelveticaBoldOblique

let $helvetica\_bold\_oblique\_widths$ $=$
$hashtable\_of\_dictionary$
[32, 278; 33, 333; 34, 474; 35, 556; 36, 556; 37, 889; 38, 722; 39, 278;
40, 333; 41, 333; 42, 389; 43, 584; 44, 278; 45, 333; 46, 278; 47, 278;
48, 556; 49, 556; 50, 556; 51, 556; 52, 556; 53, 556; 54, 556; 55, 556;
56, 556; 57, 556; 58, 333; 59, 333; 60, 584; 61, 584; 62, 584; 63, 611;
64, 975; 65, 722; 66, 722; 67, 722; 68, 722; 69, 667; 70, 611; 71, 778;
72, 722; 73, 278; 74, 556; 75, 722; 76, 611; 77, 833; 78, 722; 79, 778;
80, 667; 81, 778; 82, 722; 83, 667; 84, 611; 85, 722; 86, 667; 87, 944;
88, 667; 89, 667; 90, 611; 91, 333; 92, 278; 93, 333; 94, 584; 95, 556;
96, 278; 97, 556; 98, 611; 99, 556; 100, 611; 101, 556; 102, 333; 103, 611;
104, 611; 105, 278; 106, 278; 107, 556; 108, 278; 109, 889; 110, 611;
111, 611; 112, 611; 113, 611; 114, 389; 115, 556; 116, 333; 117, 611;
118, 556; 119, 778; 120, 556; 121, 556; 122, 500; 123, 389; 124, 280;
125, 389; 126, 584; 161, 333; 162, 556; 163, 556; 164, 167; 165, 556;
166, 556; 167, 556; 168, 556; 169, 238; 170, 500; 171, 556; 172, 333;
173, 333; 174, 611; 175, 611; 177, 556; 178, 556; 179, 556; 180, 278;
182, 556; 183, 350; 184, 278; 185, 500; 186, 500; 187, 556; 188, 1000;
189, 1000; 191, 611; 193, 333; 194, 333; 195, 333; 196, 333; 197, 333;
198, 333; 199, 333; 200, 333; 202, 333; 203, 333; 205, 333; 206, 333;
207, 333; 208, 1000; 225, 1000; 227, 370; 232, 611; 233, 778; 234, 1000;
235, 365; 241, 889; 245, 278; 248, 278; 249, 611; 250, 944; 251, 611]

let $helvetica\_bold\_oblique\_kerns$ $=$
$hashtable\_of\_kerns$
[65, 121, -30; 65, 119, -30; 65, 118, -40; 65, 117, -30; 65, 89, -110;
65, 87, -60; 65, 86, -80; 65, 85, -50; 65, 84, -90; 65, 81, -40;
65, 79, -40; 65, 71, -50; 65, 67, -40; 66, 85, -10; 66, 65, -30;
68, 46, -30; 68, 44, -30; 68, 89, -70; 68, 87, -40; 68, 86, -40;
68, 65, -40; 70, 46, -100; 70, 44, -100; 70, 97, -20; 70, 65, -80;
74, 117, -20; 74, 46, -20; 74, 44, -20; 74, 65, -20; 75, 121, -40;
75, 117, -30; 75, 111, -35; 75, 101, -15; 75, 79, -30; 76, 121, -30;
76, 39, -140; 76, 186, -140; 76, 89, -120; 76, 87, -80; 76, 86, -110;

76, 84, -90; 79, 46, -40; 79, 44, -40; 79, 89, -70; 79, 88, -50;
79, 87, -50; 79, 86, -50; 79, 84, -40; 79, 65, -50; 80, 46, -120;
80, 111, -40; 80, 101, -30; 80, 44, -120; 80, 97, -30; 80, 65, -100;
81, 46, 20; 81, 44, 20; 81, 85, -10; 82, 89, -50; 82, 87, -40; 82, 86, -50;
82, 85, -20; 82, 84, -20; 82, 79, -20; 84, 121, -60; 84, 119, -60;
84, 117, -90; 84, 59, -40; 84, 114, -80; 84, 46, -80; 84, 111, -80;
84, 45, -120; 84, 101, -60; 84, 44, -80; 84, 58, -40; 84, 97, -80;
84, 79, -40; 84, 65, -90; 85, 46, -30; 85, 44, -30; 85, 65, -50;
86, 117, -60; 86, 59, -40; 86, 46, -120; 86, 111, -90; 86, 45, -80;
86, 101, -50; 86, 44, -120; 86, 58, -40; 86, 97, -60; 86, 79, -50;
86, 71, -50; 86, 65, -80; 87, 121, -20; 87, 117, -45; 87, 59, -10;
87, 46, -80; 87, 111, -60; 87, 45, -40; 87, 101, -35; 87, 44, -80;
87, 58, -10; 87, 97, -40; 87, 79, -20; 87, 65, -60; 89, 117, -100;
89, 59, -50; 89, 46, -100; 89, 111, -100; 89, 101, -80; 89, 44, -100;
89, 58, -50; 89, 97, -90; 89, 79, -70; 89, 65, -110; 97, 121, -20;
97, 119, -15; 97, 118, -15; 97, 103, -10; 98, 121, -20; 98, 118, -20;
98, 117, -20; 98, 108, -10; 99, 121, -10; 99, 108, -20; 99, 107, -20;
99, 104, -10; 58, 32, -40; 44, 32, -40; 44, 39, -120; 44, 186, -120;
100, 121, -15; 100, 119, -15; 100, 118, -15; 100, 100, -10; 101, 121, -15;
101, 120, -15; 101, 119, -15; 101, 118, -15; 101, 46, 20; 101, 44, 10;
102, 39, 30; 102, 186, 30; 102, 46, -10; 102, 111, -20; 102, 101, -10;
102, 44, -10; 103, 103, -10; 103, 101, 10; 104, 121, -20; 107, 111, -15;
108, 121, -15; 108, 119, -15; 109, 121, -30; 109, 117, -20; 110, 121, -20;
110, 118, -40; 110, 117, -10; 111, 121, -20; 111, 120, -30; 111, 119, -15;
111, 118, -20; 112, 121, -15; 46, 32, -40; 46, 39, -120; 46, 186, -120;
186, 32, -80; 96, 96, -46; 39, 118, -20; 39, 32, -80; 39, 115, -60;
39, 114, -40; 39, 39, -46; 39, 108, -20; 39, 100, -80; 114, 121, 10;
114, 118, 10; 114, 116, 20; 114, 115, -15; 114, 113, -20; 114, 46, -60;
114, 111, -20; 114, 45, -20; 114, 103, -15; 114, 100, -20; 114, 44, -60;
114, 99, -20; 115, 119, -15; 59, 32, -40; 32, 96, -60; 32, 170, -80;
32, 89, -120; 32, 87, -80; 32, 86, -80; 32, 84, -100; 118, 46, -80;
118, 111, -30; 118, 44, -80; 118, 97, -20; 119, 46, -40; 119, 111, -20;
119, 44, -40; 120, 101, -10; 121, 46, -80; 121, 111, -25; 121, 101, -10;
121, 44, -80; 121, 97, -30; 122, 101, 10]

Courier

let *courier_widths* =
hashtable_of_dictionary
[32, 600; 33, 600; 34, 600; 35, 600; 36, 600; 37, 600; 38, 600; 39, 600;
40, 600; 41, 600; 42, 600; 43, 600; 44, 600; 45, 600; 46, 600; 47, 600;
48, 600; 49, 600; 50, 600; 51, 600; 52, 600; 53, 600; 54, 600; 55, 600;
56, 600; 57, 600; 58, 600; 59, 600; 60, 600; 61, 600; 62, 600; 63, 600;
64, 600; 65, 600; 66, 600; 67, 600; 68, 600; 69, 600; 70, 600; 71, 600;
72, 600; 73, 600; 74, 600; 75, 600; 76, 600; 77, 600; 78, 600; 79, 600;
80, 600; 81, 600; 82, 600; 83, 600; 84, 600; 85, 600; 86, 600; 87, 600;
88, 600; 89, 600; 90, 600; 91, 600; 92, 600; 93, 600; 94, 600; 95, 600;
96, 600; 97, 600; 98, 600; 99, 600; 100, 600; 101, 600; 102, 600; 103, 600;
104, 600; 105, 600; 106, 600; 107, 600; 108, 600; 109, 600; 110, 600;
111, 600; 112, 600; 113, 600; 114, 600; 115, 600; 116, 600; 117, 600;
118, 600; 119, 600; 120, 600; 121, 600; 122, 600; 123, 600; 124, 600;

125, 600; 126, 600; 161, 600; 162, 600; 163, 600; 164, 600; 165, 600;
166, 600; 167, 600; 168, 600; 169, 600; 170, 600; 171, 600; 172, 600;
173, 600; 174, 600; 175, 600; 177, 600; 178, 600; 179, 600; 180, 600;
182, 600; 183, 600; 184, 600; 185, 600; 186, 600; 187, 600; 188, 600;
189, 600; 191, 600; 193, 600; 194, 600; 195, 600; 196, 600; 197, 600;
198, 600; 199, 600; 200, 600; 202, 600; 203, 600; 205, 600; 206, 600;
207, 600; 208, 600; 225, 600; 227, 600; 232, 600; 233, 600; 234, 600;
235, 600; 241, 600; 245, 600; 248, 600; 249, 600; 250, 600; 251, 600]

let $courier\_kerns = hashtable\_of\_kerns$ [ ]

CourierBold

let $courier\_bold\_widths =$
$hashtable\_of\_dictionary$
[32, 600; 33, 600; 34, 600; 35, 600; 36, 600; 37, 600; 38, 600; 39, 600;
40, 600; 41, 600; 42, 600; 43, 600; 44, 600; 45, 600; 46, 600; 47, 600;
48, 600; 49, 600; 50, 600; 51, 600; 52, 600; 53, 600; 54, 600; 55, 600;
56, 600; 57, 600; 58, 600; 59, 600; 60, 600; 61, 600; 62, 600; 63, 600;
64, 600; 65, 600; 66, 600; 67, 600; 68, 600; 69, 600; 70, 600; 71, 600;
72, 600; 73, 600; 74, 600; 75, 600; 76, 600; 77, 600; 78, 600; 79, 600;
80, 600; 81, 600; 82, 600; 83, 600; 84, 600; 85, 600; 86, 600; 87, 600;
88, 600; 89, 600; 90, 600; 91, 600; 92, 600; 93, 600; 94, 600; 95, 600;
96, 600; 97, 600; 98, 600; 99, 600; 100, 600; 101, 600; 102, 600; 103, 600;
104, 600; 105, 600; 106, 600; 107, 600; 108, 600; 109, 600; 110, 600;
111, 600; 112, 600; 113, 600; 114, 600; 115, 600; 116, 600; 117, 600;
118, 600; 119, 600; 120, 600; 121, 600; 122, 600; 123, 600; 124, 600;
125, 600; 126, 600; 161, 600; 162, 600; 163, 600; 164, 600; 165, 600;
166, 600; 167, 600; 168, 600; 169, 600; 170, 600; 171, 600; 172, 600;
173, 600; 174, 600; 175, 600; 177, 600; 178, 600; 179, 600; 180, 600;
182, 600; 183, 600; 184, 600; 185, 600; 186, 600; 187, 600; 188, 600;
189, 600; 191, 600; 193, 600; 194, 600; 195, 600; 196, 600; 197, 600;
198, 600; 199, 600; 200, 600; 202, 600; 203, 600; 205, 600; 206, 600;
207, 600; 208, 600; 225, 600; 227, 600; 232, 600; 233, 600; 234, 600;
235, 600; 241, 600; 245, 600; 248, 600; 249, 600; 250, 600; 251, 600]

let $courier\_bold\_kerns = hashtable\_of\_kerns$ [ ]

CourierOblique

let $courier\_oblique\_widths =$
$hashtable\_of\_dictionary$
[32, 600; 33, 600; 34, 600; 35, 600; 36, 600; 37, 600; 38, 600; 39, 600;
40, 600; 41, 600; 42, 600; 43, 600; 44, 600; 45, 600; 46, 600; 47, 600;
48, 600; 49, 600; 50, 600; 51, 600; 52, 600; 53, 600; 54, 600; 55, 600;
56, 600; 57, 600; 58, 600; 59, 600; 60, 600; 61, 600; 62, 600; 63, 600;
64, 600; 65, 600; 66, 600; 67, 600; 68, 600; 69, 600; 70, 600; 71, 600;
72, 600; 73, 600; 74, 600; 75, 600; 76, 600; 77, 600; 78, 600; 79, 600;
80, 600; 81, 600; 82, 600; 83, 600; 84, 600; 85, 600; 86, 600; 87, 600;
88, 600; 89, 600; 90, 600; 91, 600; 92, 600; 93, 600; 94, 600; 95, 600;
96, 600; 97, 600; 98, 600; 99, 600; 100, 600; 101, 600; 102, 600; 103, 600;
104, 600; 105, 600; 106, 600; 107, 600; 108, 600; 109, 600; 110, 600;
111, 600; 112, 600; 113, 600; 114, 600; 115, 600; 116, 600; 117, 600;

118, 600; 119, 600; 120, 600; 121, 600; 122, 600; 123, 600; 124, 600; 125, 600; 126, 600; 161, 600; 162, 600; 163, 600; 164, 600; 165, 600; 166, 600; 167, 600; 168, 600; 169, 600; 170, 600; 171, 600; 172, 600; 173, 600; 174, 600; 175, 600; 177, 600; 178, 600; 179, 600; 180, 600; 182, 600; 183, 600; 184, 600; 185, 600; 186, 600; 187, 600; 188, 600; 189, 600; 191, 600; 193, 600; 194, 600; 195, 600; 196, 600; 197, 600; 198, 600; 199, 600; 200, 600; 202, 600; 203, 600; 205, 600; 206, 600; 207, 600; 208, 600; 225, 600; 227, 600; 232, 600; 233, 600; 234, 600; 235, 600; 241, 600; 245, 600; 248, 600; 249, 600; 250, 600; 251, 600]

let $courier\_oblique\_kerns = hashtable\_of\_kerns$ [ ]

CourierBoldOblique

let $courier\_bold\_oblique\_widths =$
$hashtable\_of\_dictionary$
[32, 600; 33, 600; 34, 600; 35, 600; 36, 600; 37, 600; 38, 600; 39, 600; 40, 600; 41, 600; 42, 600; 43, 600; 44, 600; 45, 600; 46, 600; 47, 600; 48, 600; 49, 600; 50, 600; 51, 600; 52, 600; 53, 600; 54, 600; 55, 600; 56, 600; 57, 600; 58, 600; 59, 600; 60, 600; 61, 600; 62, 600; 63, 600; 64, 600; 65, 600; 66, 600; 67, 600; 68, 600; 69, 600; 70, 600; 71, 600; 72, 600; 73, 600; 74, 600; 75, 600; 76, 600; 77, 600; 78, 600; 79, 600; 80, 600; 81, 600; 82, 600; 83, 600; 84, 600; 85, 600; 86, 600; 87, 600; 88, 600; 89, 600; 90, 600; 91, 600; 92, 600; 93, 600; 94, 600; 95, 600; 96, 600; 97, 600; 98, 600; 99, 600; 100, 600; 101, 600; 102, 600; 103, 600; 104, 600; 105, 600; 106, 600; 107, 600; 108, 600; 109, 600; 110, 600; 111, 600; 112, 600; 113, 600; 114, 600; 115, 600; 116, 600; 117, 600; 118, 600; 119, 600; 120, 600; 121, 600; 122, 600; 123, 600; 124, 600; 125, 600; 126, 600; 161, 600; 162, 600; 163, 600; 164, 600; 165, 600; 166, 600; 167, 600; 168, 600; 169, 600; 170, 600; 171, 600; 172, 600; 173, 600; 174, 600; 175, 600; 177, 600; 178, 600; 179, 600; 180, 600; 182, 600; 183, 600; 184, 600; 185, 600; 186, 600; 187, 600; 188, 600; 189, 600; 191, 600; 193, 600; 194, 600; 195, 600; 196, 600; 197, 600; 198, 600; 199, 600; 200, 600; 202, 600; 203, 600; 205, 600; 206, 600; 207, 600; 208, 600; 225, 600; 227, 600; 232, 600; 233, 600; 234, 600; 235, 600; 241, 600; 245, 600; 248, 600; 249, 600; 250, 600; 251, 600]

let $courier\_bold\_oblique\_kerns = hashtable\_of\_kerns$ [ ]

Symbol

let $symbol\_widths =$
$hashtable\_of\_dictionary$
[32, 250; 33, 333; 34, 713; 35, 500; 36, 549; 37, 833; 38, 778; 39, 439; 40, 333; 41, 333; 42, 500; 43, 549; 44, 250; 45, 549; 46, 250; 47, 278; 48, 500; 49, 500; 50, 500; 51, 500; 52, 500; 53, 500; 54, 500; 55, 500; 56, 500; 57, 500; 58, 278; 59, 278; 60, 549; 61, 549; 62, 549; 63, 444; 64, 549; 65, 722; 66, 667; 67, 722; 68, 612; 69, 611; 70, 763; 71, 603; 72, 722; 73, 333; 74, 631; 75, 722; 76, 686; 77, 889; 78, 722; 79, 722; 80, 768; 81, 741; 82, 556; 83, 592; 84, 611; 85, 690; 86, 439; 87, 768; 88, 645; 89, 795; 90, 611; 91, 333; 92, 863; 93, 333; 94, 658; 95, 500; 96, 500; 97, 631; 98, 549; 99, 549; 100, 494; 101, 439; 102, 521; 103, 411; 104, 603; 105, 329; 106, 603; 107, 549; 108, 549; 109, 576; 110, 521;

111, 549; 112, 549; 113, 521; 114, 549; 115, 603; 116, 439; 117, 576;
118, 713; 119, 686; 120, 493; 121, 686; 122, 494; 123, 480; 124, 200;
125, 480; 126, 549; 161, 620; 162, 247; 163, 549; 164, 167; 165, 713;
166, 500; 167, 753; 168, 753; 169, 753; 170, 753; 171, 1042; 172, 987;
173, 603; 174, 987; 175, 603; 176, 400; 177, 549; 178, 411; 179, 549;
180, 549; 181, 713; 182, 494; 183, 460; 184, 549; 185, 549; 186, 549;
187, 549; 188, 1000; 189, 603; 190, 1000; 191, 658; 192, 823; 193, 686;
194, 795; 195, 987; 196, 768; 197, 768; 198, 823; 199, 768; 200, 768;
201, 713; 202, 713; 203, 713; 204, 713; 205, 713; 206, 713; 207, 713;
208, 768; 209, 713; 210, 790; 211, 790; 212, 890; 213, 823; 214, 549;
215, 250; 216, 713; 217, 603; 218, 603; 219, 1042; 220, 987; 221, 603;
222, 987; 223, 603; 224, 494; 225, 329; 226, 790; 227, 790; 228, 786;
229, 713; 230, 384; 231, 384; 232, 384; 233, 384; 234, 384; 235, 384;
236, 494; 237, 494; 238, 494; 239, 494; 241, 329; 242, 274; 243, 686;
244, 686; 245, 686; 246, 384; 247, 384; 248, 384; 249, 384; 250, 384;
251, 384; 252, 494; 253, 494; 254, 494]

let $symbol\_kerns\ =\ hashtable\_of\_kerns$ [ ]

ZapfDingbats

let $zapfdingbats\_widths\ =$
$hashtable\_of\_dictionary$
[32, 278; 33, 974; 34, 961; 35, 974; 36, 980; 37, 719; 38, 789; 39, 790;
40, 791; 41, 690; 42, 960; 43, 939; 44, 549; 45, 855; 46, 911; 47, 933;
48, 911; 49, 945; 50, 974; 51, 755; 52, 846; 53, 762; 54, 761; 55, 571;
56, 677; 57, 763; 58, 760; 59, 759; 60, 754; 61, 494; 62, 552; 63, 537;
64, 577; 65, 692; 66, 786; 67, 788; 68, 788; 69, 790; 70, 793; 71, 794;
72, 816; 73, 823; 74, 789; 75, 841; 76, 823; 77, 833; 78, 816; 79, 831;
80, 923; 81, 744; 82, 723; 83, 749; 84, 790; 85, 792; 86, 695; 87, 776;
88, 768; 89, 792; 90, 759; 91, 707; 92, 708; 93, 682; 94, 701; 95, 826;
96, 815; 97, 789; 98, 789; 99, 707; 100, 687; 101, 696; 102, 689; 103, 786;
104, 787; 105, 713; 106, 791; 107, 785; 108, 791; 109, 873; 110, 761;
111, 762; 112, 762; 113, 759; 114, 759; 115, 892; 116, 892; 117, 788;
118, 784; 119, 438; 120, 138; 121, 277; 122, 415; 123, 392; 124, 392;
125, 668; 126, 668; 161, 732; 162, 544; 163, 544; 164, 910; 165, 667;
166, 760; 167, 760; 168, 776; 169, 595; 170, 694; 171, 626; 172, 788;
173, 788; 174, 788; 175, 788; 176, 788; 177, 788; 178, 788; 179, 788;
180, 788; 181, 788; 182, 788; 183, 788; 184, 788; 185, 788; 186, 788;
187, 788; 188, 788; 189, 788; 190, 788; 191, 788; 192, 788; 193, 788;
194, 788; 195, 788; 196, 788; 197, 788; 198, 788; 199, 788; 200, 788;
201, 788; 202, 788; 203, 788; 204, 788; 205, 788; 206, 788; 207, 788;
208, 788; 209, 788; 210, 788; 211, 788; 212, 894; 213, 838; 214, 1016;
215, 458; 216, 748; 217, 924; 218, 748; 219, 918; 220, 927; 221, 928;
222, 928; 223, 834; 224, 873; 225, 828; 226, 924; 227, 924; 228, 917;
229, 930; 230, 931; 231, 463; 232, 883; 233, 836; 234, 836; 235, 867;
236, 867; 237, 696; 238, 696; 239, 874; 241, 874; 242, 760; 243, 946;
244, 771; 245, 865; 246, 771; 247, 888; 248, 967; 249, 888; 250, 831;
251, 873; 252, 927; 253, 970; 254, 918]

let $zapfdingbats\_kerns\ =\ hashtable\_of\_kerns$ [ ]

Main functions

type *widths* = (*int*, *int*) Hashtbl.t

type *kerns* = ((*int* × *int*), *int*) Hashtbl.t

type *table* =
  Pdftext.*standard_font* × *widths* × *kerns*

let *tables* =
  [Pdftext.TimesRoman, (*times_roman_widths*, *times_roman_kerns*);
   Pdftext.TimesBold, (*times_bold_widths*, *times_bold_kerns*);
   Pdftext.TimesItalic, (*times_italic_widths*, *times_italic_kerns*);
   Pdftext.TimesBoldItalic, (*times_bold_italic_widths*, *times_bold_italic_kerns*);
   Pdftext.Helvetica, (*helvetica_widths*, *helvetica_kerns*);
   Pdftext.HelveticaBold, (*helvetica_bold_widths*, *helvetica_bold_kerns*);
   Pdftext.HelveticaOblique, (*helvetica_oblique_widths*, *helvetica_oblique_kerns*);
   Pdftext.HelveticaBoldOblique, (*helvetica_bold_oblique_widths*, *helvetica_bold_oblique_kerns*);
   Pdftext.Courier, (*courier_widths*, *courier_kerns*);
   Pdftext.CourierBold, (*courier_bold_widths*, *courier_bold_kerns*);
   Pdftext.CourierOblique, (*courier_oblique_widths*, *courier_oblique_kerns*);
   Pdftext.CourierBoldOblique, (*courier_bold_oblique_widths*, *courier_bold_oblique_kerns*);
   Pdftext.Symbol, (*symbol_widths*, *symbol_kerns*);
   Pdftext.ZapfDingbats, (*zapfdingbats_widths*, *zapfdingbats_kerns*)]

Calculate the width of a list of characters, taking account of kerning.

let *find_kern kerns key* =
  match *tryfind kerns key* **with** Some x → x | None → 0

let *find_width widths h* =
  match *tryfind widths h* **with** Some x → x | None → 0

let rec *width widths kerns* = **function**
  | [] → 0
  | [h] → *find_width widths h*
  | h :: h' :: t →
      find_width widths h +
      find_kern kerns (h, h') +
      width widths kerns (h' :: t)

▷ The main function. Give a font and the text string.

let *textwidth f s* =
  let *widths*, *kerns* = *lookup_failnull f tables* **in**
    width widths kerns (*map int_of_char* (*explode s*))
  Unicode equivalents for some of the PDF ZapfDingbats Encoding.

let *dingbatmap_arr* =
[|"/a100", [$275E_{16}$]; "/a101", [2761_{16}]; "/a102", [2762_{16}]; "/a103", [2763_{16}];
"/a104", [2764_{16}]; "/a105", [2710_{16}]; "/a106", [2765_{16}]; "/a107", [2766_{16}];
"/a108", [2767_{16}]; "/a109", [2660_{16}]; "/a10", [2721_{16}]; "/a110", [2665_{16}];
"/a111", [2666_{16}]; "/a112", [2663_{16}]; "/a117", [2709_{16}]; "/a118", [2708_{16}];
"/a119", [2707_{16}]; "/a11", [$261B_{16}$]; "/a120", [2460_{16}]; "/a121", [2461_{16}];
"/a122", [2462_{16}]; "/a123", [2463_{16}]; "/a124", [2464_{16}]; "/a125", [2465_{16}];
"/a126", [2466_{16}]; "/a127", [2467_{16}]; "/a128", [2468_{16}]; "/a129", [2469_{16}];
"/a12", [$261E_{16}$]; "/a130", [2776_{16}]; "/a131", [2777_{16}]; "/a132", [2778_{16}];

"/a133", [2779_{16}]; "/a134", [$277A_{16}$]; "/a135", [$277B_{16}$]; "/a136", [$277C_{16}$];
"/a137", [$277D_{16}$]; "/a138", [$277E_{16}$]; "/a139", [$277F_{16}$]; "/a13", [$270C_{16}$];
"/a140", [2780_{16}]; "/a141", [2781_{16}]; "/a142", [2782_{16}]; "/a143", [2783_{16}];
"/a144", [2784_{16}]; "/a145", [2785_{16}]; "/a146", [2786_{16}]; "/a147", [2787_{16}];
"/a148", [2788_{16}]; "/a149", [2789_{16}]; "/a14", [$270D_{16}$]; "/a150", [$278A_{16}$];
"/a151", [$278B_{16}$]; "/a152", [$278C_{16}$]; "/a153", [$278D_{16}$]; "/a154", [$278E_{16}$];
"/a155", [$278F_{16}$]; "/a156", [2790_{16}]; "/a157", [2791_{16}]; "/a158", [2792_{16}];
"/a159", [2793_{16}]; "/a15", [$270E_{16}$]; "/a160", [2794_{16}]; "/a161", [2192_{16}];
"/a162", [$27A3_{16}$]; "/a163", [2194_{16}]; "/a164", [2195_{16}]; "/a165", [2799_{16}];
"/a166", [$279B_{16}$]; "/a167", [$279C_{16}$]; "/a168", [$279D_{16}$]; "/a169", [$279E_{16}$];
"/a16", [$270F_{16}$]; "/a170", [$279F_{16}$]; "/a171", [$27A0_{16}$]; "/a172", [$27A1_{16}$];
"/a173", [$27A2_{16}$]; "/a174", [$27A4_{16}$]; "/a175", [$27A5_{16}$]; "/a176", [$27A6_{16}$];
"/a177", [$27A7_{16}$]; "/a178", [$27A8_{16}$]; "/a179", [$27A9_{16}$]; "/a17", [2711_{16}];
"/a180", [$27AB_{16}$]; "/a181", [$27AD_{16}$]; "/a182", [$27AF_{16}$]; "/a183", [$27B2_{16}$];
"/a184", [$27B3_{16}$]; "/a185", [$27B5_{16}$]; "/a186", [$27B8_{16}$]; "/a187", [$27BA_{16}$];
"/a188", [$27BB_{16}$]; "/a189", [$27BC_{16}$]; "/a18", [2712_{16}]; "/a190", [$27BD_{16}$];
"/a191", [$27BE_{16}$]; "/a192", [$279A_{16}$]; "/a193", [$27AA_{16}$]; "/a194", [$27B6_{16}$];
"/a195", [$27B9_{16}$]; "/a196", [2798_{16}]; "/a197", [$27B4_{16}$]; "/a198", [$27B7_{16}$];
"/a199", [$27AC_{16}$]; "/a19", [2713_{16}]; "/a1", [2701_{16}]; "/a200", [$27AE_{16}$];
"/a201", [$27B1_{16}$]; "/a202", [2703_{16}]; "/a203", [2750_{16}]; "/a204", [2752_{16}];
"/a205", [$276E_{16}$]; "/a206", [2770_{16}]; "/a20", [2714_{16}]; "/a21", [2715_{16}];
"/a22", [2716_{16}]; "/a23", [2717_{16}]; "/a24", [2718_{16}]; "/a25", [2719_{16}];
"/a26", [$271A_{16}$]; "/a27", [$271B_{16}$]; "/a28", [$271C_{16}$]; "/a29", [2722_{16}];
"/a2", [2702_{16}]; "/a30", [2723_{16}]; "/a31", [2724_{16}]; "/a32", [2725_{16}];
"/a33", [2726_{16}]; "/a34", [2727_{16}]; "/a35", [2605_{16}]; "/a36", [2729_{16}];
"/a37", [$272A_{16}$]; "/a38", [$272B_{16}$]; "/a39", [$272C_{16}$]; "/a3", [2704_{16}];
"/a40", [$272D_{16}$]; "/a41", [$272E_{16}$]; "/a42", [$272F_{16}$]; "/a43", [2730_{16}];
"/a44", [2731_{16}]; "/a45", [2732_{16}]; "/a46", [2733_{16}]; "/a47", [2734_{16}];
"/a48", [2735_{16}]; "/a49", [2736_{16}]; "/a4", [$260E_{16}$]; "/a50", [2737_{16}];
"/a51", [2738_{16}]; "/a52", [2739_{16}]; "/a53", [$273A_{16}$]; "/a54", [$273B_{16}$];
"/a55", [$273C_{16}$]; "/a56", [$273D_{16}$]; "/a57", [$273E_{16}$]; "/a58", [$273F_{16}$];
"/a59", [2740_{16}]; "/a5", [2706_{16}]; "/a60", [2741_{16}]; "/a61", [2742_{16}];
"/a62", [2743_{16}]; "/a63", [2744_{16}]; "/a64", [2745_{16}]; "/a65", [2746_{16}];
"/a66", [2747_{16}]; "/a67", [2748_{16}]; "/a68", [2749_{16}]; "/a69", [$274A_{16}$];
"/a6", [$271D_{16}$]; "/a70", [$274B_{16}$]; "/a71", [$25CF_{16}$]; "/a72", [$274D_{16}$];
"/a73", [$25A0_{16}$]; "/a74", [$274F_{16}$]; "/a75", [2751_{16}]; "/a76", [$25B2_{16}$];
"/a77", [$25BC_{16}$]; "/a78", [$25C6_{16}$]; "/a79", [2756_{16}]; "/a7", [$271E_{16}$];
"/a81", [$25D7_{16}$]; "/a82", [2758_{16}]; "/a83", [2759_{16}]; "/a84", [$275A_{16}$];
"/a85", [$276F_{16}$]; "/a86", [2771_{16}]; "/a87", [2772_{16}]; "/a88", [2773_{16}];
"/a89", [2768_{16}]; "/a8", [$271F_{16}$]; "/a90", [2769_{16}]; "/a91", [$276C_{16}$];
"/a92", [$276D_{16}$]; "/a93", [$276A_{16}$]; "/a94", [$276B_{16}$]; "/a95", [2774_{16}];
"/a96", [2775_{16}]; "/a97", [$275B_{16}$]; "/a98", [$275C_{16}$]; "/a99", [$275D_{16}$];
"/a9", [2720_{16}]|]

let *dingbatmap* = Array.*to_list dingbatmap_arr*

let *truetypemap_arr* =
[|"/G20", [0020_{16}]; "/G21", [0021_{16}]; "/G22", [0022_{16}]; "/G23", [0023_{16}];
"/G24", [0024_{16}]; "/G25", [0025_{16}]; "/G26", [0026_{16}]; "/G27", [0027_{16}];
"/G28", [0028_{16}]; "/G29", [0029_{16}]; "/G2a", [$002A_{16}$]; "/G2b", [$002B_{16}$];
"/G2c", [$002C_{16}$]; "/G2d", [$002D_{16}$]; "/G2e", [$002E_{16}$]; "/G2f", [$002F_{16}$];

"/G30", $[0030_{16}]$; "/G31", $[0031_{16}]$; "/G32", $[0032_{16}]$; "/G33", $[0033_{16}]$;
"/G34", $[0034_{16}]$; "/G35", $[0035_{16}]$; "/G36", $[0036_{16}]$; "/G37", $[0037_{16}]$;
"/G38", $[0038_{16}]$; "/G39", $[0039_{16}]$; "/G3a", $[003A_{16}]$; "/G3b", $[003B_{16}]$;
"/G3c", $[003C_{16}]$; "/G3d", $[003D_{16}]$; "/G3e", $[003E_{16}]$; "/G3f", $[003F_{16}]$;
"/G40", $[0040_{16}]$; "/G41", $[0041_{16}]$; "/G42", $[0042_{16}]$; "/G43", $[0043_{16}]$;
"/G44", $[0044_{16}]$; "/G45", $[0045_{16}]$; "/G46", $[0046_{16}]$; "/G47", $[0047_{16}]$;
"/G48", $[0048_{16}]$; "/G49", $[0049_{16}]$; "/G4a", $[004A_{16}]$; "/G4b", $[004B_{16}]$;
"/G4c", $[004C_{16}]$; "/G4d", $[004D_{16}]$; "/G4e", $[004E_{16}]$; "/G4f", $[004F_{16}]$;
"/G50", $[0050_{16}]$; "/G51", $[0051_{16}]$; "/G52", $[0052_{16}]$; "/G53", $[0053_{16}]$;
"/G54", $[0054_{16}]$; "/G55", $[0055_{16}]$; "/G56", $[0056_{16}]$; "/G57", $[0057_{16}]$;
"/G58", $[0058_{16}]$; "/G59", $[0059_{16}]$; "/G5a", $[005A_{16}]$; "/G5b", $[005B_{16}]$;
"/G5c", $[005C_{16}]$; "/G5d", $[005D_{16}]$; "/G5e", $[005E_{16}]$; "/G5f", $[005F_{16}]$;
"/G60", $[0060_{16}]$; "/G61", $[0061_{16}]$; "/G62", $[0062_{16}]$; "/G63", $[0063_{16}]$;
"/G64", $[0064_{16}]$; "/G65", $[0065_{16}]$; "/G66", $[0066_{16}]$; "/G67", $[0067_{16}]$;
"/G68", $[0068_{16}]$; "/G69", $[0069_{16}]$; "/G6a", $[006A_{16}]$; "/G6b", $[006B_{16}]$;
"/G6c", $[006C_{16}]$; "/G6d", $[006D_{16}]$; "/G6e", $[006E_{16}]$; "/G6f", $[006F_{16}]$;
"/G70", $[0070_{16}]$; "/G71", $[0071_{16}]$; "/G72", $[0072_{16}]$; "/G73", $[0073_{16}]$;
"/G74", $[0074_{16}]$; "/G75", $[0075_{16}]$; "/G76", $[0076_{16}]$; "/G77", $[0077_{16}]$;
"/G78", $[0078_{16}]$; "/G79", $[0079_{16}]$; "/G7a", $[007A_{16}]$; "/G7b", $[007B_{16}]$;
"/G7c", $[007C_{16}]$; "/G7d", $[007D_{16}]$; "/G7e", $[007E_{16}]$; "/Ga0", $[00A0_{16}]$;
"/Ga1", $[00A1_{16}]$; "/Ga2", $[00A2_{16}]$; "/Ga3", $[00A3_{16}]$; "/Ga4", $[00A4_{16}]$;
"/Ga5", $[00A5_{16}]$; "/Ga6", $[00A6_{16}]$; "/Ga7", $[00A7_{16}]$; "/Ga8", $[00A8_{16}]$;
"/Ga9", $[00A9_{16}]$; "/Gaa", $[00AA_{16}]$; "/Gab", $[00AB_{16}]$; "/Gac", $[00AC_{16}]$;
"/Gad", $[00AD_{16}]$; "/Gae", $[00AE_{16}]$; "/Gaf", $[00AF_{16}]$; "/Gb0", $[00B0_{16}]$;
"/Gb1", $[00B1_{16}]$; "/Gb2", $[00B2_{16}]$; "/Gb3", $[00B3_{16}]$; "/Gb4", $[00B4_{16}]$;
"/Gb5", $[00B5_{16}]$; "/Gb6", $[00B6_{16}]$; "/Gb7", $[00B7_{16}]$; "/Gb8", $[00B8_{16}]$;
"/Gb9", $[00B9_{16}]$; "/Gba", $[00BA_{16}]$; "/Gbb", $[00BB_{16}]$; "/Gbc", $[00BC_{16}]$;
"/Gbd", $[00BD_{16}]$; "/Gbe", $[00BE_{16}]$; "/Gbf", $[00BF_{16}]$; "/Gc0", $[00C0_{16}]$;
"/Gc1", $[00C1_{16}]$; "/Gc2", $[00C2_{16}]$; "/Gc3", $[00C3_{16}]$; "/Gc4", $[00C4_{16}]$;
"/Gc5", $[00C5_{16}]$; "/Gc6", $[00C6_{16}]$; "/Gc7", $[00C7_{16}]$; "/Gc8", $[00C8_{16}]$;
"/Gc9", $[00C9_{16}]$; "/Gca", $[00CA_{16}]$; "/Gcb", $[00CB_{16}]$; "/Gcc", $[00CC_{16}]$;
"/Gcd", $[00CD_{16}]$; "/Gce", $[00CE_{16}]$; "/Gcf", $[00CF_{16}]$; "/Gd0", $[00D0_{16}]$;
"/Gd1", $[00D1_{16}]$; "/Gd2", $[00D2_{16}]$; "/Gd3", $[00D3_{16}]$; "/Gd4", $[00D4_{16}]$;
"/Gd5", $[00D5_{16}]$; "/Gd6", $[00D6_{16}]$; "/Gd7", $[00D7_{16}]$; "/Gd8", $[00D8_{16}]$;
"/Gd9", $[00D9_{16}]$; "/Gda", $[00DA_{16}]$; "/Gdb", $[00DB_{16}]$; "/Gdc", $[00DC_{16}]$;
"/Gdd", $[00DD_{16}]$; "/Gde", $[00DE_{16}]$; "/Gdf", $[00DF_{16}]$; "/Ge0", $[00E0_{16}]$;
"/Ge1", $[00E1_{16}]$; "/Ge2", $[00E2_{16}]$; "/Ge3", $[00E3_{16}]$; "/Ge4", $[00E4_{16}]$;
"/Ge5", $[00E5_{16}]$; "/Ge6", $[00E6_{16}]$; "/Ge7", $[00E7_{16}]$; "/Ge8", $[00E8_{16}]$;
"/Ge9", $[00E9_{16}]$; "/Gea", $[00EA_{16}]$; "/Geb", $[00EB_{16}]$; "/Gec", $[00EC_{16}]$;
"/Ged", $[00ED_{16}]$; "/Gee", $[00EE_{16}]$; "/Gef", $[00EF_{16}]$; "/Gf0", $[00F0_{16}]$;
"/Gf1", $[00F1_{16}]$; "/Gf2", $[00F2_{16}]$; "/Gf3", $[00F3_{16}]$; "/Gf4", $[00F4_{16}]$;
"/Gf5", $[00F5_{16}]$; "/Gf6", $[00F6_{16}]$; "/Gf7", $[00F7_{16}]$; "/Gf8", $[00F8_{16}]$;
"/Gf9", $[00F9_{16}]$; "/Gfa", $[00FA_{16}]$; "/Gfb", $[00FB_{16}]$; "/Gfc", $[00FC_{16}]$;
"/Gfd", $[00FD_{16}]$; "/Gfe", $[00FE_{16}]$; "/Gff", $[00FF_{16}]$; "/G82", $[201A_{16}]$;
"/G83", $[0192_{16}]$; "/G84", $[201E_{16}]$; "/G85", $[2026_{16}]$; "/G86", $[2020_{16}]$;
"/G87", $[2021_{16}]$; "/G88", $[02C6_{16}]$; "/G89", $[2030_{16}]$; "/G8a", $[0160_{16}]$;
"/G8b", $[2039_{16}]$; "/G8c", $[0152_{16}]$; "/G91", $[2018_{16}]$; "/G92", $[2019_{16}]$;
"/G93", $[201C_{16}]$; "/G94", $[201D_{16}]$; "/G95", $[2022_{16}]$; "/G96", $[2013_{16}]$;
"/G97", $[2014_{16}]$; "/G98", $[02DC_{16}]$; "/G99", $[2122_{16}]$; "/G9a", $[0161_{16}]$;
"/G9b", $[203A_{16}]$; "/G9c", $[0153_{16}]$; "/G9f", $[0178_{16}]$; "/G2A", $[002A_{16}]$;

"/G2B", [002B$_{16}$]; "/G2C", [002C$_{16}$]; "/G2D", [002D$_{16}$]; "/G2E", [002E$_{16}$];
"/G2F", [002F$_{16}$]; "/G3A", [003A$_{16}$]; "/G3B", [003B$_{16}$]; "/G3C", [003C$_{16}$];
"/G3D", [003D$_{16}$]; "/G3E", [003E$_{16}$]; "/G3F", [003F$_{16}$]; "/G4A", [004A$_{16}$];
"/G4B", [004B$_{16}$]; "/G4C", [004C$_{16}$]; "/G4D", [004D$_{16}$]; "/G4E", [004E$_{16}$];
"/G4F", [004F$_{16}$]; "/G5A", [005A$_{16}$]; "/G5B", [005B$_{16}$]; "/G5C", [005C$_{16}$];
"/G5D", [005D$_{16}$]; "/G5E", [005E$_{16}$]; "/G5F", [005F$_{16}$]; "/G6A", [006A$_{16}$];
"/G6B", [006B$_{16}$]; "/G6C", [006C$_{16}$]; "/G6D", [002D$_{16}$]; "/G6E", [006E$_{16}$];
"/G6F", [006F$_{16}$]; "/G7A", [007A$_{16}$]; "/G7B", [007B$_{16}$]; "/G7C", [007C$_{16}$];
"/G7D", [007D$_{16}$]; "/G7E", [007E$_{16}$]; "/G8A", [008A$_{16}$]; "/G8B", [008B$_{16}$];
"/G8C", [008C$_{16}$]; "/G9A", [009A$_{16}$]; "/G9B", [009B$_{16}$]; "/G9C", [009C$_{16}$];
"/GA0", [00A0$_{16}$]; "/GA1", [00A1$_{16}$]; "/GA2", [00A2$_{16}$]; "/GA3", [00A3$_{16}$];
"/GA4", [00A4$_{16}$]; "/GA5", [00A5$_{16}$]; "/GA6", [00A6$_{16}$]; "/GA7", [00A7$_{16}$];
"/GA8", [00A8$_{16}$]; "/GA9", [00A9$_{16}$]; "/GAA", [00AA$_{16}$]; "/GAB", [00AB$_{16}$];
"/GAC", [00AC$_{16}$]; "/GAD", [00AD$_{16}$]; "/GAE", [00AE$_{16}$]; "/GAF", [00AF$_{16}$];
"/GB0", [00B0$_{16}$]; "/GB1", [00B1$_{16}$]; "/GB2", [00B2$_{16}$]; "/GB3", [00B3$_{16}$];
"/GB4", [00B4$_{16}$]; "/GB5", [00B5$_{16}$]; "/GA6", [00B6$_{16}$]; "/GB7", [00B7$_{16}$];
"/GB8", [00B8$_{16}$]; "/GB9", [00B9$_{16}$]; "/GBA", [00BA$_{16}$]; "/GBB", [00BB$_{16}$];
"/GBC", [00BC$_{16}$]; "/GBD", [00BD$_{16}$]; "/GBE", [00BE$_{16}$]; "/GBF", [00BF$_{16}$];
"/GC0", [00C0$_{16}$]; "/GC1", [00C1$_{16}$]; "/GC2", [00C2$_{16}$]; "/GC3", [00C3$_{16}$];
"/GC4", [00C4$_{16}$]; "/GC5", [00C5$_{16}$]; "/GC6", [00C6$_{16}$]; "/GC7", [00C7$_{16}$];
"/GC8", [00C8$_{16}$]; "/GC9", [00C9$_{16}$]; "/GCA", [00CA$_{16}$]; "/GCB", [00CB$_{16}$];
"/GCC", [00CC$_{16}$]; "/GCD", [00CD$_{16}$]; "/GCE", [00CE$_{16}$]; "/GCF", [00CF$_{16}$];
"/GD0", [00D0$_{16}$]; "/GD1", [00D1$_{16}$]; "/GD2", [00D2$_{16}$]; "/GD3", [00D3$_{16}$];
"/GD4", [00D4$_{16}$]; "/GD5", [00D5$_{16}$]; "/GD6", [00D6$_{16}$]; "/GD7", [00D7$_{16}$];
"/GD8", [00D8$_{16}$]; "/GD9", [00D9$_{16}$]; "/GDA", [00DA$_{16}$]; "/GDB", [00DB$_{16}$];
"/GDC", [00DC$_{16}$]; "/GDD", [00DD$_{16}$]; "/GDE", [00DE$_{16}$]; "/GDF", [00DF$_{16}$];
"/GE0", [00E0$_{16}$]; "/GE1", [00E1$_{16}$]; "/GE2", [00E2$_{16}$]; "/GE3", [00E3$_{16}$];
"/GE4", [00E4$_{16}$]; "/GE5", [00E5$_{16}$]; "/GE6", [00E6$_{16}$]; "/GE7", [00E7$_{16}$];
"/GE8", [00E8$_{16}$]; "/GE9", [00E9$_{16}$]; "/GEA", [00EA$_{16}$]; "/GEB", [00EB$_{16}$];
"/GEC", [00EC$_{16}$]; "/GED", [00ED$_{16}$]; "/GEE", [00EE$_{16}$]; "/GEF", [00EF$_{16}$];
"/GF0", [00F0$_{16}$]; "/GF1", [00F1$_{16}$]; "/GF2", [00F2$_{16}$]; "/GF3", [00F3$_{16}$];
"/GF4", [00F4$_{16}$]; "/GF5", [00F5$_{16}$]; "/GF6", [00F6$_{16}$]; "/GF7", [00F7$_{16}$];
"/GF8", [00F8$_{16}$]; "/GF9", [00F9$_{16}$]; "/GFA", [00FA$_{16}$]; "/GFB", [00FB$_{16}$];
"/GFC", [00FC$_{16}$]; "/GFD", [00FD$_{16}$]; "/GFE", [00FE$_{16}$]; "/GFF", [00FF$_{16}$]|]

let *truetypemap* = Array.*to_list truetypemap_arr*

The Adobe Glyph list, 2.0

let *glyphmap_arr* = [|"/A", [0041$_{16}$]; "/AE", [00C6$_{16}$]; "/AEacute", [01FC$_{16}$];
"/AEmacron", [01E2$_{16}$]; "/AEsmall", [F7E6$_{16}$]; "/Aacute", [00C1$_{16}$];
"/Aacutesmall", [F7E1$_{16}$]; "/Abreve", [0102$_{16}$]; "/Abreveacute", [1EAE$_{16}$];
"/Abrevecyrillic", [04D0$_{16}$]; "/Abrevedotbelow", [1EB6$_{16}$]; "/Abrevegrave",
[1EB0$_{16}$]; "/Abrevehookabove", [1EB2$_{16}$]; "/Abrevetilde", [1EB4$_{16}$]; "/Acaron",
[01CD$_{16}$]; "/Acircle", [24B6$_{16}$]; "/Acircumflex", [00C2$_{16}$]; "/Acircumflexacute",
[1EA4$_{16}$]; "/Acircumflexdotbelow", [1EAC$_{16}$]; "/Acircumflexgrave", [1EA6$_{16}$];
"/Acircumflexhookabove", [1EA8$_{16}$]; "/Acircumflexsmall", [F7E2$_{16}$];
"/Acircumflextilde", [1EAA$_{16}$]; "/Acute", [F6C9$_{16}$]; "/Acutesmall", [F7B4$_{16}$];
"/Acyrillic", [0410$_{16}$]; "/Adblgrave", [0200$_{16}$]; "/Adieresis", [00C4$_{16}$];
"/Adieresiscyrillic", [04D2$_{16}$]; "/Adieresismacron", [01DE$_{16}$]; "/Adieresissmall",
[F7E4$_{16}$]; "/Adotbelow", [1EA0$_{16}$]; "/Adotmacron", [01E0$_{16}$]; "/Agrave", [00C0$_{16}$];
"/Agravesmall", [F7E0$_{16}$]; "/Ahookabove", [1EA2$_{16}$]; "/Aiecyrillic", [04D4$_{16}$];

"/Ainvertedbreve", [0202$_{16}$]; "/Alpha", [0391$_{16}$]; "/Alphatonos", [0386$_{16}$]; "/Amacron", [0100$_{16}$]; "/Amonospace", [FF21$_{16}$]; "/Aogonek", [0104$_{16}$]; "/Aring", [00C5$_{16}$]; "/Aringacute", [01FA$_{16}$]; "/Aringbelow", [1E00$_{16}$]; "/Aringsmall", [F7E5$_{16}$]; "/Asmall", [F761$_{16}$]; "/Atilde", [00C3$_{16}$]; "/Atildesmall", [F7E3$_{16}$]; "/Aybarmenian", [0531$_{16}$]; "/B", [0042$_{16}$]; "/Bcircle", [24B7$_{16}$]; "/Bdotaccent", [1E02$_{16}$]; "/Bdotbelow", [1E04$_{16}$]; "/Becyrillic", [0411$_{16}$]; "/Benarmenian", [0532$_{16}$]; "/Beta", [0392$_{16}$]; "/Bhook", [0181$_{16}$]; "/Blinebelow", [1E06$_{16}$]; "/Bmonospace", [FF22$_{16}$]; "/Brevesmall", [F6F4$_{16}$]; "/Bsmall", [F762$_{16}$]; "/Btopbar", [0182$_{16}$]; "/C", [0043$_{16}$]; "/Caarmenian", [053E$_{16}$]; "/Cacute", [0106$_{16}$]; "/Caron", [F6CA$_{16}$]; "/Caronsmall", [F6F5$_{16}$]; "/Ccaron", [010C$_{16}$]; "/Ccedilla", [00C7$_{16}$]; "/Ccedillaacute", [1E08$_{16}$]; "/Ccedillasmall", [F7E7$_{16}$]; "/Ccircle", [24B8$_{16}$]; "/Ccircumflex", [0108$_{16}$]; "/Cdot", [010A$_{16}$]; "/Cdotaccent", [010A$_{16}$]; "/Cedillasmall", [F7B8$_{16}$]; "/Chaarmenian", [0549$_{16}$]; "/Cheabkhasiancyrillic", [04BC$_{16}$]; "/Checyrillic", [0427$_{16}$]; "/Chedescenderabkhasiancyrillic", [04BE$_{16}$]; "/Chedescendercyrillic", [04B6$_{16}$]; "/Chedieresiscyrillic", [04F4$_{16}$]; "/Cheharmenian", [0543$_{16}$]; "/Chekhakassiancyrillic", [04CB$_{16}$]; "/Cheverticalstrokecyrillic", [04B8$_{16}$]; "/Chi", [03A7$_{16}$]; "/Chook", [0187$_{16}$]; "/Circumflexsmall", [F6F6$_{16}$]; "/Cmonospace", [FF23$_{16}$]; "/Coarmenian", [0551$_{16}$]; "/Csmall", [F763$_{16}$]; "/D", [0044$_{16}$]; "/DZ", [01F1$_{16}$]; "/DZcaron", [01C4$_{16}$]; "/Daarmenian", [0534$_{16}$]; "/Dafrican", [0189$_{16}$]; "/Dcaron", [010E$_{16}$]; "/Dcedilla", [1E10$_{16}$]; "/Dcircle", [24B9$_{16}$]; "/Dcircumflexbelow", [1E12$_{16}$]; "/Dcroat", [0110$_{16}$]; "/Ddotaccent", [1E0A$_{16}$]; "/Ddotbelow", [1E0C$_{16}$]; "/Decyrillic", [0414$_{16}$]; "/Deicoptic", [03EE$_{16}$]; "/Delta", [2206$_{16}$]; "/Deltagreek", [0394$_{16}$]; "/Dhook", [018A$_{16}$]; "/Dieresis", [F6CB$_{16}$]; "/DieresisAcute", [F6CC$_{16}$]; "/DieresisGrave", [F6CD$_{16}$]; "/Dieresissmall", [F7A8$_{16}$]; "/Digammagreek", [03DC$_{16}$]; "/Djecyrillic", [0402$_{16}$]; "/Dlinebelow", [1E0E$_{16}$]; "/Dmonospace", [FF24$_{16}$]; "/Dotaccentsmall", [F6F7$_{16}$]; "/Dslash", [0110$_{16}$]; "/Dsmall", [F764$_{16}$]; "/Dtopbar", [018B$_{16}$]; "/Dz", [01F2$_{16}$]; "/Dzcaron", [01C5$_{16}$]; "/Dzeabkhasiancyrillic", [04E0$_{16}$]; "/Dzecyrillic", [0405$_{16}$]; "/Dzhecyrillic", [040F$_{16}$]; "/E", [0045$_{16}$]; "/Eacute", [00C9$_{16}$]; "/Eacutesmall", [F7E9$_{16}$]; "/Ebreve", [0114$_{16}$]; "/Ecaron", [011A$_{16}$]; "/Ecedillabreve", [1E1C$_{16}$]; "/Echarmenian", [0535$_{16}$]; "/Ecircle", [24BA$_{16}$]; "/Ecircumflex", [00CA$_{16}$]; "/Ecircumflexacute", [1EBE$_{16}$]; "/Ecircumflexbelow", [1E18$_{16}$]; "/Ecircumflexdotbelow", [1EC6$_{16}$]; "/Ecircumflexgrave", [1EC0$_{16}$]; "/Ecircumflexhookabove", [1EC2$_{16}$]; "/Ecircumflexsmall", [F7EA$_{16}$]; "/Ecircumflextilde", [1EC4$_{16}$]; "/Ecyrillic", [0404$_{16}$]; "/Edblgrave", [0204$_{16}$]; "/Edieresis", [00CB$_{16}$]; "/Edieresissmall", [F7EB$_{16}$]; "/Edot", [0116$_{16}$]; "/Edotaccent", [0116$_{16}$]; "/Edotbelow", [1EB8$_{16}$]; "/Efcyrillic", [0424$_{16}$]; "/Egrave", [00C8$_{16}$]; "/Egravesmall", [F7E8$_{16}$]; "/Eharmenian", [0537$_{16}$]; "/Ehookabove", [1EBA$_{16}$]; "/Eightroman", [2167$_{16}$]; "/Einvertedbreve", [0206$_{16}$]; "/Eiotifiedcyrillic", [0464$_{16}$]; "/Elcyrillic", [041B$_{16}$]; "/Elevenroman", [216A$_{16}$]; "/Emacron", [0112$_{16}$]; "/Emacronacute", [1E16$_{16}$]; "/Emacrongrave", [1E14$_{16}$]; "/Emcyrillic", [041C$_{16}$]; "/Emonospace", [FF25$_{16}$]; "/Encyrillic", [041D$_{16}$]; "/Endescendercyrillic", [04A2$_{16}$]; "/Eng", [014A$_{16}$]; "/Enghecyrillic", [04A4$_{16}$]; "/Enhookcyrillic", [04C7$_{16}$]; "/Eogonek", [0118$_{16}$]; "/Eopen", [0190$_{16}$]; "/Epsilon", [0395$_{16}$]; "/Epsilontonos", [0388$_{16}$]; "/Ercyrillic", [0420$_{16}$]; "/Ereversed", [018E$_{16}$]; "/Ereversedcyrillic", [042D$_{16}$]; "/Escyrillic", [0421$_{16}$]; "/Esdescendercyrillic", [04AA$_{16}$]; "/Esh", [01A9$_{16}$]; "/Esmall", [F765$_{16}$]; "/Eta", [0397$_{16}$]; "/Etarmenian", [0538$_{16}$]; "/Etatonos", [0389$_{16}$]; "/Eth", [00D0$_{16}$]; "/Ethsmall", [F7F0$_{16}$]; "/Etilde", [1EBC$_{16}$]; "/Etildebelow", [1E1A$_{16}$]; "/Euro", [20AC$_{16}$]; "/Ezh", [01B7$_{16}$]; "/Ezhcaron", [01EE$_{16}$]; "/Ezhreversed", [01B8$_{16}$]; "/F", [0046$_{16}$]; "/Fcircle", [24BB$_{16}$]; "/Fdotaccent", [1E1E$_{16}$]; "/Feharmenian", [0556$_{16}$]; "/Feicoptic", [03E4$_{16}$]; "/Fhook", [0191$_{16}$]; "/Fitacyrillic", [0472$_{16}$]; "/Fiveroman", [2164$_{16}$]; "/Fmonospace", [FF26$_{16}$]; "/Fourroman", [2163$_{16}$]; "/Fsmall", [F766$_{16}$]; "/G", [0047$_{16}$]; "/GBsquare", [3387$_{16}$]; "/Gacute", [01F4$_{16}$]; "/Gamma", [0393$_{16}$]; "/Gammaafrican", [0194$_{16}$]; "/Gangiacoptic", [03EA$_{16}$]; "/Gbreve", [011E$_{16}$]; "/Gcaron", [01E6$_{16}$]; "/Gcedilla", [0122$_{16}$]; "/Gcircle", [24BC$_{16}$]; "/Gcircumflex", [011C$_{16}$]; "/Gcommaaccent", [0122$_{16}$]; "/Gdot", [0120$_{16}$]; "/Gdotaccent", [0120$_{16}$]; "/Gecyrillic", [0413$_{16}$]; "/Ghadarmenian", [0542$_{16}$]; "/Ghemiddlehookcyrillic", [0494$_{16}$]; "/Ghestrokecyrillic", [0492$_{16}$]; "/Gheupturncyrillic", [0490$_{16}$]; "/Ghook", [0193$_{16}$]; "/Gimarmenian", [0533$_{16}$]; "/Gjecyrillic", [0403$_{16}$]; "/Gmacron", [1E20$_{16}$]; "/Gmonospace", [FF27$_{16}$]; "/Grave", [F6CE$_{16}$]; "/Gravesmall", [F760$_{16}$]; "/Gsmall", [F767$_{16}$]; "/Gsmallhook", [029B$_{16}$]; "/Gstroke", [01E4$_{16}$]; "/H", [0048$_{16}$]; "/H18533", [25CF$_{16}$]; "/H18543", [25AA$_{16}$]; "/H18551", [25AB$_{16}$]; "/H22073", [25A1$_{16}$]; "/HPsquare", [33CB$_{16}$]; "/Haabkhasiancyrillic", [04A8$_{16}$]; "/Hadescendercyrillic", [04B2$_{16}$]; "/Hardsigncyrillic", [042A$_{16}$]; "/Hbar", [0126$_{16}$]; "/Hbrevebelow", [1E2A$_{16}$]; "/Hcedilla", [1E28$_{16}$]; "/Hcircle", [24BD$_{16}$]; "/Hcircumflex", [0124$_{16}$]; "/Hdieresis", [1E26$_{16}$]; "/Hdotaccent", [1E22$_{16}$]; "/Hdotbelow", [1E24$_{16}$]; "/Hmonospace", [FF28$_{16}$]; "/Hoarmenian", [0540$_{16}$]; "/Horicoptic", [03E8$_{16}$]; "/Hsmall", [F768$_{16}$]; "/Hungarumlaut", [F6CF$_{16}$]; "/Hungarumlautsmall", [F6F8$_{16}$]; "/Hzsquare", [3390$_{16}$]; "/I", [0049$_{16}$]; "/IAcyrillic", [042F$_{16}$]; "/IJ", [0132$_{16}$]; "/IUcyrillic", [042E$_{16}$]; "/Iacute", [00CD$_{16}$]; "/Iacutesmall", [F7ED$_{16}$]; "/Ibreve", [012C$_{16}$]; "/Icaron", [01CF$_{16}$]; "/Icircle", [24BE$_{16}$]; "/Icircumflex", [00CE$_{16}$]; "/Icircumflexsmall", [F7EE$_{16}$]; "/Icyrillic", [0406$_{16}$]; "/Idblgrave", [0208$_{16}$]; "/Idieresis", [00CF$_{16}$]; "/Idieresisacute", [1E2E$_{16}$]; "/Idieresiscyrillic", [04E4$_{16}$]; "/Idieresissmall", [F7EF$_{16}$]; "/Idot", [0130$_{16}$]; "/Idotaccent", [0130$_{16}$]; "/Idotbelow", [1ECA$_{16}$]; "/Iebrevecyrillic", [04D6$_{16}$]; "/Iecyrillic", [0415$_{16}$]; "/Ifraktur", [2111$_{16}$]; "/Igrave", [00CC$_{16}$]; "/Igravesmall", [F7EC$_{16}$]; "/Ihookabove", [1EC8$_{16}$]; "/Iicyrillic", [0418$_{16}$]; "/Iinvertedbreve", [020A$_{16}$]; "/Iishortcyrillic", [0419$_{16}$]; "/Imacron", [012A$_{16}$]; "/Imacroncyrillic", [04E2$_{16}$]; "/Imonospace", [FF29$_{16}$]; "/Iniarmenian", [053B$_{16}$]; "/Iocyrillic", [0401$_{16}$]; "/Iogonek", [012E$_{16}$]; "/Iota", [0399$_{16}$]; "/Iotaafrican", [0196$_{16}$]; "/Iotadieresis", [03AA$_{16}$]; "/Iotatonos", [038A$_{16}$]; "/Ismall", [F769$_{16}$]; "/Istroke", [0197$_{16}$]; "/Itilde", [0128$_{16}$]; "/Itildebelow", [1E2C$_{16}$]; "/Izhitsacyrillic", [0474$_{16}$]; "/Izhitsadblgravecyrillic", [0476$_{16}$]; "/J", [004A$_{16}$]; "/Jaarmenian", [0541$_{16}$]; "/Jcircle", [24BF$_{16}$]; "/Jcircumflex", [0134$_{16}$]; "/Jecyrillic", [0408$_{16}$]; "/Jheharmenian", [054B$_{16}$]; "/Jmonospace", [FF2A$_{16}$]; "/Jsmall", [F76A$_{16}$]; "/K", [004B$_{16}$]; "/KBsquare", [3385$_{16}$]; "/KKsquare", [33CD$_{16}$]; "/Kabashkircyrillic", [04A0$_{16}$]; "/Kacute", [1E30$_{16}$]; "/Kacyrillic", [041A$_{16}$]; "/Kadescendercyrillic", [049A$_{16}$]; "/Kahookcyrillic", [04C3$_{16}$]; "/Kappa", [039A$_{16}$]; "/Kastrokecyrillic", [049E$_{16}$]; "/Kaverticalstrokecyrillic", [049C$_{16}$]; "/Kcaron", [01E8$_{16}$]; "/Kcedilla", [0136$_{16}$]; "/Kcircle", [24C0$_{16}$]; "/Kcommaaccent", [0136$_{16}$]; "/Kdotbelow", [1E32$_{16}$]; "/Keharmenian", [0554$_{16}$]; "/Kenarmenian", [053F$_{16}$]; "/Khacyrillic", [0425$_{16}$]; "/Kheicoptic", [03E6$_{16}$]; "/Khook", [0198$_{16}$]; "/Kjecyrillic", [040C$_{16}$]; "/Klinebelow", [1E34$_{16}$]; "/Kmonospace", [FF2B$_{16}$]; "/Koppacyrillic", [0480$_{16}$]; "/Koppagreek", [03DE$_{16}$]; "/Ksicyrillic", [046E$_{16}$]; "/Ksmall", [F76B$_{16}$]; "/L", [004C$_{16}$]; "/LJ", [01C7$_{16}$]; "/LL", [F6BF$_{16}$]; "/Lacute", [0139$_{16}$]; "/Lambda", [039B$_{16}$]; "/Lcaron", [013D$_{16}$]; "/Lcedilla", [013B$_{16}$]; "/Lcircle", [24C1$_{16}$];

"/Lcircumflexbelow", [1E3C$_{16}$]; "/Lcommaaccent", [013B$_{16}$]; "/Ldot", [013F$_{16}$]; "/Ldotaccent", [013F$_{16}$]; "/Ldotbelow", [1E36$_{16}$]; "/Ldotbelowmacron", [1E38$_{16}$]; "/Liwnarmenian", [053C$_{16}$]; "/Lj", [01C8$_{16}$]; "/Ljecyrillic", [0409$_{16}$]; "/Llinebelow", [1E3A$_{16}$]; "/Lmonospace", [FF2C$_{16}$]; "/Lslash", [0141$_{16}$]; "/Lslashsmall", [F6F9$_{16}$]; "/Lsmall", [F76C$_{16}$]; "/M", [004D$_{16}$]; "/MBsquare", [3386$_{16}$]; "/Macron", [F6D0$_{16}$]; "/Macronsmall", [F7AF$_{16}$]; "/Macute", [1E3E$_{16}$]; "/Mcircle", [24C2$_{16}$]; "/Mdotaccent", [1E40$_{16}$]; "/Mdotbelow", [1E42$_{16}$]; "/Menarmenian", [0544$_{16}$]; "/Mmonospace", [FF2D$_{16}$]; "/Msmall", [F76D$_{16}$]; "/Mturned", [019C$_{16}$]; "/Mu", [039C$_{16}$]; "/N", [004E$_{16}$]; "/NJ", [01CA$_{16}$]; "/Nacute", [0143$_{16}$]; "/Ncaron", [0147$_{16}$]; "/Ncedilla", [0145$_{16}$]; "/Ncircle", [24C3$_{16}$]; "/Ncircumflexbelow", [1E4A$_{16}$]; "/Ncommaaccent", [0145$_{16}$]; "/Ndotaccent", [1E44$_{16}$]; "/Ndotbelow", [1E46$_{16}$]; "/Nhookleft", [019D$_{16}$]; "/Nineroman", [2168$_{16}$]; "/Nj", [01CB$_{16}$]; "/Njecyrillic", [040A$_{16}$]; "/Nlinebelow", [1E48$_{16}$]; "/Nmonospace", [FF2E$_{16}$]; "/Nowarmenian", [0546$_{16}$]; "/Nsmall", [F76E$_{16}$]; "/Ntilde", [00D1$_{16}$]; "/Ntildesmall", [F7F1$_{16}$]; "/Nu", [039D$_{16}$]; "/O", [004F$_{16}$]; "/OE", [0152$_{16}$]; "/OEsmall", [F6FA$_{16}$]; "/Oacute", [00D3$_{16}$]; "/Oacutesmall", [F7F3$_{16}$]; "/Obarredcyrillic", [04E8$_{16}$]; "/Obarreddieresiscyrillic", [04EA$_{16}$]; "/Obreve", [014E$_{16}$]; "/Ocaron", [01D1$_{16}$]; "/Ocenteredtilde", [019F$_{16}$]; "/Ocircle", [24C4$_{16}$]; "/Ocircumflex", [00D4$_{16}$]; "/Ocircumflexacute", [1ED0$_{16}$]; "/Ocircumflexdotbelow", [1ED8$_{16}$]; "/Ocircumflexgrave", [1ED2$_{16}$]; "/Ocircumflexhookabove", [1ED4$_{16}$]; "/Ocircumflexsmall", [F7F4$_{16}$]; "/Ocircumflextilde", [1ED6$_{16}$]; "/Ocyrillic", [041E$_{16}$]; "/Odblacute", [0150$_{16}$]; "/Odblgrave", [020C$_{16}$]; "/Odieresis", [00D6$_{16}$]; "/Odieresiscyrillic", [04E6$_{16}$]; "/Odieresissmall", [F7F6$_{16}$]; "/Odotbelow", [1ECC$_{16}$]; "/Ogoneksmall", [F6FB$_{16}$]; "/Ograve", [00D2$_{16}$]; "/Ogravesmall", [F7F2$_{16}$]; "/Oharmenian", [0555$_{16}$]; "/Ohm", [2126$_{16}$]; "/Ohookabove", [1ECE$_{16}$]; "/Ohorn", [01A0$_{16}$]; "/Ohornacute", [1EDA$_{16}$]; "/Ohorndotbelow", [1EE2$_{16}$]; "/Ohorngrave", [1EDC$_{16}$]; "/Ohornhookabove", [1EDE$_{16}$]; "/Ohorntilde", [1EE0$_{16}$]; "/Ohungarumlaut", [0150$_{16}$]; "/Oi", [01A2$_{16}$]; "/Oinvertedbreve", [020E$_{16}$]; "/Omacron", [014C$_{16}$]; "/Omacronacute", [1E52$_{16}$]; "/Omacrongrave", [1E50$_{16}$]; "/Omega", [2126$_{16}$]; "/Omegacyrillic", [0460$_{16}$]; "/Omegagreek", [03A9$_{16}$]; "/Omegaroundcyrillic", [047A$_{16}$]; "/Omegatitlocyrillic", [047C$_{16}$]; "/Omegatonos", [038F$_{16}$]; "/Omicron", [039F$_{16}$]; "/Omicrontonos", [038C$_{16}$]; "/Omonospace", [FF2F$_{16}$]; "/Oneroman", [2160$_{16}$]; "/Oogonek", [01EA$_{16}$]; "/Oogonekmacron", [01EC$_{16}$]; "/Oopen", [0186$_{16}$]; "/Oslash", [00D8$_{16}$]; "/Oslashacute", [01FE$_{16}$]; "/Oslashsmall", [F7F8$_{16}$]; "/Osmall", [F76F$_{16}$]; "/Ostrokeacute", [01FE$_{16}$]; "/Otcyrillic", [047E$_{16}$]; "/Otilde", [00D5$_{16}$]; "/Otildeacute", [1E4C$_{16}$]; "/Otildedieresis", [1E4E$_{16}$]; "/Otildesmall", [F7F5$_{16}$]; "/P", [0050$_{16}$]; "/Pacute", [1E54$_{16}$]; "/Pcircle", [24C5$_{16}$]; "/Pdotaccent", [1E56$_{16}$]; "/Pecyrillic", [041F$_{16}$]; "/Peharmenian", [054A$_{16}$]; "/Pemiddlehookcyrillic", [04A6$_{16}$]; "/Phi", [03A6$_{16}$]; "/Phook", [01A4$_{16}$]; "/Pi", [03A0$_{16}$]; "/Piwrarmenian", [0553$_{16}$]; "/Pmonospace", [FF30$_{16}$]; "/Psi", [03A8$_{16}$]; "/Psicyrillic", [0470$_{16}$]; "/Psmall", [F770$_{16}$]; "/Q", [0051$_{16}$]; "/Qcircle", [24C6$_{16}$]; "/Qmonospace", [FF31$_{16}$]; "/Qsmall", [F771$_{16}$]; "/R", [0052$_{16}$]; "/Raarmenian", [054C$_{16}$]; "/Racute", [0154$_{16}$]; "/Rcaron", [0158$_{16}$]; "/Rcedilla", [0156$_{16}$]; "/Rcircle", [24C7$_{16}$]; "/Rcommaaccent", [0156$_{16}$]; "/Rdblgrave", [0210$_{16}$]; "/Rdotaccent", [1E58$_{16}$]; "/Rdotbelow", [1E5A$_{16}$]; "/Rdotbelowmacron", [1E5C$_{16}$]; "/Reharmenian", [0550$_{16}$]; "/Rfraktur", [211C$_{16}$]; "/Rho", [03A1$_{16}$]; "/Ringsmall", [F6FC$_{16}$]; "/Rinvertedbreve", [0212$_{16}$]; "/Rlinebelow", [1E5E$_{16}$]; "/Rmonospace", [FF32$_{16}$]; "/Rsmall", [F772$_{16}$]; "/Rsmallinverted", [0281$_{16}$];

"/Rsmallinvertedsuperior", [$02B6_{16}$]; "/S", [0053_{16}]; "/SF010000", [$250C_{16}$]; "/SF020000", [2514_{16}]; "/SF030000", [2510_{16}]; "/SF040000", [2518_{16}]; "/SF050000", [$253C_{16}$]; "/SF060000", [$252C_{16}$]; "/SF070000", [2534_{16}]; "/SF080000", [$251C_{16}$]; "/SF090000", [2524_{16}]; "/SF100000", [2500_{16}]; "/SF110000", [2502_{16}]; "/SF190000", [2561_{16}]; "/SF200000", [2562_{16}]; "/SF210000", [2556_{16}]; "/SF220000", [2555_{16}]; "/SF230000", [2563_{16}]; "/SF240000", [2551_{16}]; "/SF250000", [2557_{16}]; "/SF260000", [$255D_{16}$]; "/SF270000", [$255C_{16}$]; "/SF280000", [$255B_{16}$]; "/SF360000", [$255E_{16}$]; "/SF370000", [$255F_{16}$]; "/SF380000", [$255A_{16}$]; "/SF390000", [2554_{16}]; "/SF400000", [2569_{16}]; "/SF410000", [2566_{16}]; "/SF420000", [2560_{16}]; "/SF430000", [2550_{16}]; "/SF440000", [$256C_{16}$]; "/SF450000", [2567_{16}]; "/SF460000", [2568_{16}]; "/SF470000", [2564_{16}]; "/SF480000", [2565_{16}]; "/SF490000", [2559_{16}]; "/SF500000", [2558_{16}]; "/SF510000", [2552_{16}]; "/SF520000", [2553_{16}]; "/SF530000", [$256B_{16}$]; "/SF540000", [$256A_{16}$]; "/Sacute", [$015A_{16}$]; "/Sacutedotaccent", [$1E64_{16}$]; "/Sampigreek", [$03E0_{16}$]; "/Scaron", [0160_{16}]; "/Scarondotaccent", [$1E66_{16}$]; "/Scaronsmall", [$F6FD_{16}$]; "/Scedilla", [$015E_{16}$]; "/Schwa", [$018F_{16}$]; "/Schwacyrillic", [$04D8_{16}$]; "/Schwadieresiscyrillic", [$04DA_{16}$]; "/Scircle", [$24C8_{16}$]; "/Scircumflex", [$015C_{16}$]; "/Scommaaccent", [0218_{16}]; "/Sdotaccent", [$1E60_{16}$]; "/Sdotbelow", [$1E62_{16}$]; "/Sdotbelowdotaccent", [$1E68_{16}$]; "/Seharmenian", [$054D_{16}$]; "/Sevenroman", [2166_{16}]; "/Shaarmenian", [0547_{16}]; "/Shacyrillic", [0428_{16}]; "/Shchacyrillic", [0429_{16}]; "/Sheicoptic", [$03E2_{16}$]; "/Shhacyrillic", [$04BA_{16}$]; "/Shimacoptic", [$03EC_{16}$]; "/Sigma", [$03A3_{16}$]; "/Sixroman", [2165_{16}]; "/Smonospace", [$FF33_{16}$]; "/Softsigncyrillic", [$042C_{16}$]; "/Ssmall", [$F773_{16}$]; "/Stigmagreek", [$03DA_{16}$]; "/T", [0054_{16}]; "/Tau", [$03A4_{16}$]; "/Tbar", [0166_{16}]; "/Tcaron", [0164_{16}]; "/Tcedilla", [0162_{16}]; "/Tcircle", [$24C9_{16}$]; "/Tcircumflexbelow", [$1E70_{16}$]; "/Tcommaaccent", [0162_{16}]; "/Tdotaccent", [$1E6A_{16}$]; "/Tdotbelow", [$1E6C_{16}$]; "/Tecyrillic", [0422_{16}]; "/Tedescendercyrillic", [$04AC_{16}$]; "/Tenroman", [2169_{16}]; "/Tetsecyrillic", [$04B4_{16}$]; "/Theta", [0398_{16}]; "/Thook", [$01AC_{16}$]; "/Thorn", [$00DE_{16}$]; "/Thornsmall", [$F7FE_{16}$]; "/Threeroman", [2162_{16}]; "/Tildesmall", [$F6FE_{16}$]; "/Tiwnarmenian", [$054F_{16}$]; "/Tlinebelow", [$1E6E_{16}$]; "/Tmonospace", [$FF34_{16}$]; "/Toarmenian", [0539_{16}]; "/Tonefive", [$01BC_{16}$]; "/Tonesix", [0184_{16}]; "/Tonetwo", [$01A7_{16}$]; "/Tretroflexhook", [$01AE_{16}$]; "/Tsecyrillic", [0426_{16}]; "/Tshecyrillic", [$040B_{16}$]; "/Tsmall", [$F774_{16}$]; "/Twelveroman", [$216B_{16}$]; "/Tworoman", [2161_{16}]; "/U", [0055_{16}]; "/Uacute", [$00DA_{16}$]; "/Uacutesmall", [$F7FA_{16}$]; "/Ubreve", [$016C_{16}$]; "/Ucaron", [$01D3_{16}$]; "/Ucircle", [$24CA_{16}$]; "/Ucircumflex", [$00DB_{16}$]; "/Ucircumflexbelow", [$1E76_{16}$]; "/Ucircumflexsmall", [$F7FB_{16}$]; "/Ucyrillic", [0423_{16}]; "/Udblacute", [0170_{16}]; "/Udblgrave", [0214_{16}]; "/Udieresis", [$00DC_{16}$]; "/Udieresisacute", [$01D7_{16}$]; "/Udieresisbelow", [$1E72_{16}$]; "/Udieresiscaron", [$01D9_{16}$]; "/Udieresiscyrillic", [$04F0_{16}$]; "/Udieresisgrave", [$01DB_{16}$]; "/Udieresismacron", [$01D5_{16}$]; "/Udieresissmall", [$F7FC_{16}$]; "/Udotbelow", [$1EE4_{16}$]; "/Ugrave", [$00D9_{16}$]; "/Ugravesmall", [$F7F9_{16}$]; "/Uhookabove", [$1EE6_{16}$]; "/Uhorn", [$01AF_{16}$]; "/Uhornacute", [$1EE8_{16}$]; "/Uhorndotbelow", [$1EF0_{16}$]; "/Uhorngrave", [$1EEA_{16}$]; "/Uhornhookabove", [$1EEC_{16}$]; "/Uhorntilde", [$1EEE_{16}$]; "/Uhungarumlaut", [0170_{16}]; "/Uhungarumlautcyrillic", [$04F2_{16}$]; "/Uinvertedbreve", [0216_{16}]; "/Ukcyrillic", [0478_{16}]; "/Umacron", [$016A_{16}$]; "/Umacroncyrillic", [$04EE_{16}$]; "/Umacrondieresis", [$1E7A_{16}$]; "/Umonospace", [$FF35_{16}$]; "/Uogonek", [0172_{16}]; "/Upsilon", [$03A5_{16}$]; "/Upsilon1", [$03D2_{16}$]; "/Upsilonacutehooksymbolgreek",

[03D3$_{16}$]; "/Upsilonafrican", [01B1$_{16}$]; "/Upsilondieresis", [03AB$_{16}$];
"/Upsilondieresishooksymbolgreek", [03D4$_{16}$]; "/Upsilonhooksymbol", [03D2$_{16}$];
"/Upsilontonos", [038E$_{16}$]; "/Uring", [016E$_{16}$]; "/Ushortcyrillic", [040E$_{16}$];
"/Usmall", [F775$_{16}$]; "/Ustraightcyrillic", [04AE$_{16}$]; "/Ustraightstrokecyrillic",
[04B0$_{16}$]; "/Utilde", [0168$_{16}$]; "/Utildeacute", [1E78$_{16}$]; "/Utildebelow",
[1E74$_{16}$]; "/V", [0056$_{16}$]; "/Vcircle", [24CB$_{16}$]; "/Vdotbelow", [1E7E$_{16}$];
"/Vecyrillic", [0412$_{16}$]; "/Vewarmenian", [054E$_{16}$]; "/Vhook", [01B2$_{16}$];
"/Vmonospace", [FF36$_{16}$]; "/Voarmenian", [0548$_{16}$]; "/Vsmall", [F776$_{16}$];
"/Vtilde", [1E7C$_{16}$]; "/W", [0057$_{16}$]; "/Wacute", [1E82$_{16}$]; "/Wcircle", [24CC$_{16}$];
"/Wcircumflex", [0174$_{16}$]; "/Wdieresis", [1E84$_{16}$]; "/Wdotaccent", [1E86$_{16}$];
"/Wdotbelow", [1E88$_{16}$]; "/Wgrave", [1E80$_{16}$]; "/Wmonospace", [FF37$_{16}$]; "/Wsmall",
[F777$_{16}$]; "/X", [0058$_{16}$]; "/Xcircle", [24CD$_{16}$]; "/Xdieresis", [1E8C$_{16}$];
"/Xdotaccent", [1E8A$_{16}$]; "/Xeharmenian", [053D$_{16}$]; "/Xi", [039E$_{16}$];
"/Xmonospace", [FF38$_{16}$]; "/Xsmall", [F778$_{16}$]; "/Y", [0059$_{16}$]; "/Yacute",
[00DD$_{16}$]; "/Yacutesmall", [F7FD$_{16}$]; "/Yatcyrillic", [0462$_{16}$]; "/Ycircle",
[24CE$_{16}$]; "/Ycircumflex", [0176$_{16}$]; "/Ydieresis", [0178$_{16}$]; "/Ydieresissmall",
[F7FF$_{16}$]; "/Ydotaccent", [1E8E$_{16}$]; "/Ydotbelow", [1EF4$_{16}$]; "/Yericyrillic",
[042B$_{16}$]; "/Yerudieresiscyrillic", [04F8$_{16}$]; "/Ygrave", [1EF2$_{16}$]; "/Yhook",
[01B3$_{16}$]; "/Yhookabove", [1EF6$_{16}$]; "/Yiarmenian", [0545$_{16}$]; "/Yicyrillic",
[0407$_{16}$]; "/Yiwnarmenian", [0552$_{16}$]; "/Ymonospace", [FF39$_{16}$]; "/Ysmall",
[F779$_{16}$]; "/Ytilde", [1EF8$_{16}$]; "/Yusbigcyrillic", [046A$_{16}$];
"/Yusbigiotifiedcyrillic", [046C$_{16}$]; "/Yuslittlecyrillic", [0466$_{16}$];
"/Yuslittleiotifiedcyrillic", [0468$_{16}$]; "/Z", [005A$_{16}$]; "/Zaarmenian", [0536$_{16}$];
"/Zacute", [0179$_{16}$]; "/Zcaron", [017D$_{16}$]; "/Zcaronsmall", [F6FF$_{16}$]; "/Zcircle",
[24CF$_{16}$]; "/Zcircumflex", [1E90$_{16}$]; "/Zdot", [017B$_{16}$]; "/Zdotaccent", [017B$_{16}$];
"/Zdotbelow", [1E92$_{16}$]; "/Zecyrillic", [0417$_{16}$]; "/Zedescendercyrillic",
[0498$_{16}$]; "/Zedieresiscyrillic", [04DE$_{16}$]; "/Zeta", [0396$_{16}$]; "/Zhearmenian",
[053A$_{16}$]; "/Zhebrevecyrillic", [04C1$_{16}$]; "/Zhecyrillic", [0416$_{16}$];
"/Zhedescendercyrillic", [0496$_{16}$]; "/Zhedieresiscyrillic", [04DC$_{16}$];
"/Zlinebelow", [1E94$_{16}$]; "/Zmonospace", [FF3A$_{16}$]; "/Zsmall", [F77A$_{16}$];
"/Zstroke", [01B5$_{16}$]; "/a", [0061$_{16}$]; "/aabengali", [0986$_{16}$]; "/aacute",
[00E1$_{16}$]; "/aadeva", [0906$_{16}$]; "/aagujarati", [0A86$_{16}$]; "/aagurmukhi", [0A06$_{16}$];
"/aamatragurmukhi", [0A3E$_{16}$]; "/aarusquare", [3303$_{16}$]; "/aavowelsignbengali",
[09BE$_{16}$]; "/aavowelsigndeva", [093E$_{16}$]; "/aavowelsigngujarati", [0ABE$_{16}$];
"/abbreviationmarkarmenian", [055F$_{16}$]; "/abbreviationsigndeva", [0970$_{16}$];
"/abengali", [0985$_{16}$]; "/abopomofo", [311A$_{16}$]; "/abreve", [0103$_{16}$];
"/abreveacute", [1EAF$_{16}$]; "/abrevecyrillic", [04D1$_{16}$]; "/abrevedotbelow",
[1EB7$_{16}$]; "/abrevegrave", [1EB1$_{16}$]; "/abrevehookabove", [1EB3$_{16}$];
"/abrevetilde", [1EB5$_{16}$]; "/acaron", [01CE$_{16}$]; "/acircle", [24D0$_{16}$];
"/acircumflex", [00E2$_{16}$]; "/acircumflexacute", [1EA5$_{16}$]; "/acircumflexdotbelow",
[1EAD$_{16}$]; "/acircumflexgrave", [1EA7$_{16}$]; "/acircumflexhookabove", [1EA9$_{16}$];
"/acircumflextilde", [1EAB$_{16}$]; "/acute", [00B4$_{16}$]; "/acutebelowcmb", [0317$_{16}$];
"/acutecmb", [0301$_{16}$]; "/acutecomb", [0301$_{16}$]; "/acutedeva", [0954$_{16}$];
"/acutelowmod", [02CF$_{16}$]; "/acutetonecmb", [0341$_{16}$]; "/acyrillic", [0430$_{16}$];
"/adblgrave", [0201$_{16}$]; "/addakgurmukhi", [0A71$_{16}$]; "/adeva", [0905$_{16}$];
"/adieresis", [00E4$_{16}$]; "/adieresiscyrillic", [04D3$_{16}$]; "/adieresismacron",
[01DF$_{16}$]; "/adotbelow", [1EA1$_{16}$]; "/adotmacron", [01E1$_{16}$]; "/ae", [00E6$_{16}$];
"/aeacute", [01FD$_{16}$]; "/aekorean", [3150$_{16}$]; "/aemacron", [01E3$_{16}$];
"/afii00208", [2015$_{16}$]; "/afii08941", [20A4$_{16}$]; "/afii10017", [0410$_{16}$];
"/afii10018", [0411$_{16}$]; "/afii10019", [0412$_{16}$]; "/afii10020", [0413$_{16}$];

"/afii10021", $[0414_{16}]$; "/afii10022", $[0415_{16}]$; "/afii10023", $[0401_{16}]$;
"/afii10024", $[0416_{16}]$; "/afii10025", $[0417_{16}]$; "/afii10026", $[0418_{16}]$;
"/afii10027", $[0419_{16}]$; "/afii10028", $[041A_{16}]$; "/afii10029", $[041B_{16}]$;
"/afii10030", $[041C_{16}]$; "/afii10031", $[041D_{16}]$; "/afii10032", $[041E_{16}]$;
"/afii10033", $[041F_{16}]$; "/afii10034", $[0420_{16}]$; "/afii10035", $[0421_{16}]$;
"/afii10036", $[0422_{16}]$; "/afii10037", $[0423_{16}]$; "/afii10038", $[0424_{16}]$;
"/afii10039", $[0425_{16}]$; "/afii10040", $[0426_{16}]$; "/afii10041", $[0427_{16}]$;
"/afii10042", $[0428_{16}]$; "/afii10043", $[0429_{16}]$; "/afii10044", $[042A_{16}]$;
"/afii10045", $[042B_{16}]$; "/afii10046", $[042C_{16}]$; "/afii10047", $[042D_{16}]$;
"/afii10048", $[042E_{16}]$; "/afii10049", $[042F_{16}]$; "/afii10050", $[0490_{16}]$;
"/afii10051", $[0402_{16}]$; "/afii10052", $[0403_{16}]$; "/afii10053", $[0404_{16}]$;
"/afii10054", $[0405_{16}]$; "/afii10055", $[0406_{16}]$; "/afii10056", $[0407_{16}]$;
"/afii10057", $[0408_{16}]$; "/afii10058", $[0409_{16}]$; "/afii10059", $[040A_{16}]$;
"/afii10060", $[040B_{16}]$; "/afii10061", $[040C_{16}]$; "/afii10062", $[040E_{16}]$;
"/afii10063", $[F6C4_{16}]$; "/afii10064", $[F6C5_{16}]$; "/afii10065", $[0430_{16}]$;
"/afii10066", $[0431_{16}]$; "/afii10067", $[0432_{16}]$; "/afii10068", $[0433_{16}]$;
"/afii10069", $[0434_{16}]$; "/afii10070", $[0435_{16}]$; "/afii10071", $[0451_{16}]$;
"/afii10072", $[0436_{16}]$; "/afii10073", $[0437_{16}]$; "/afii10074", $[0438_{16}]$;
"/afii10075", $[0439_{16}]$; "/afii10076", $[043A_{16}]$; "/afii10077", $[043B_{16}]$;
"/afii10078", $[043C_{16}]$; "/afii10079", $[043D_{16}]$; "/afii10080", $[043E_{16}]$;
"/afii10081", $[043F_{16}]$; "/afii10082", $[0440_{16}]$; "/afii10083", $[0441_{16}]$;
"/afii10084", $[0442_{16}]$; "/afii10085", $[0443_{16}]$; "/afii10086", $[0444_{16}]$;
"/afii10087", $[0445_{16}]$; "/afii10088", $[0446_{16}]$; "/afii10089", $[0447_{16}]$;
"/afii10090", $[0448_{16}]$; "/afii10091", $[0449_{16}]$; "/afii10092", $[044A_{16}]$;
"/afii10093", $[044B_{16}]$; "/afii10094", $[044C_{16}]$; "/afii10095", $[044D_{16}]$;
"/afii10096", $[044E_{16}]$; "/afii10097", $[044F_{16}]$; "/afii10098", $[0491_{16}]$;
"/afii10099", $[0452_{16}]$; "/afii10100", $[0453_{16}]$; "/afii10101", $[0454_{16}]$;
"/afii10102", $[0455_{16}]$; "/afii10103", $[0456_{16}]$; "/afii10104", $[0457_{16}]$;
"/afii10105", $[0458_{16}]$; "/afii10106", $[0459_{16}]$; "/afii10107", $[045A_{16}]$;
"/afii10108", $[045B_{16}]$; "/afii10109", $[045C_{16}]$; "/afii10110", $[045E_{16}]$;
"/afii10145", $[040F_{16}]$; "/afii10146", $[0462_{16}]$; "/afii10147", $[0472_{16}]$;
"/afii10148", $[0474_{16}]$; "/afii10192", $[F6C6_{16}]$; "/afii10193", $[045F_{16}]$;
"/afii10194", $[0463_{16}]$; "/afii10195", $[0473_{16}]$; "/afii10196", $[0475_{16}]$;
"/afii10831", $[F6C7_{16}]$; "/afii10832", $[F6C8_{16}]$; "/afii10846", $[04D9_{16}]$;
"/afii299", $[200E_{16}]$; "/afii300", $[200F_{16}]$; "/afii301", $[200D_{16}]$; "/afii57381",
$[066A_{16}]$; "/afii57388", $[060C_{16}]$; "/afii57392", $[0660_{16}]$; "/afii57393",
$[0661_{16}]$; "/afii57394", $[0662_{16}]$; "/afii57395", $[0663_{16}]$; "/afii57396",
$[0664_{16}]$; "/afii57397", $[0665_{16}]$; "/afii57398", $[0666_{16}]$; "/afii57399",
$[0667_{16}]$; "/afii57400", $[0668_{16}]$; "/afii57401", $[0669_{16}]$; "/afii57403",
$[061B_{16}]$; "/afii57407", $[061F_{16}]$; "/afii57409", $[0621_{16}]$; "/afii57410",
$[0622_{16}]$; "/afii57411", $[0623_{16}]$; "/afii57412", $[0624_{16}]$; "/afii57413",
$[0625_{16}]$; "/afii57414", $[0626_{16}]$; "/afii57415", $[0627_{16}]$; "/afii57416",
$[0628_{16}]$; "/afii57417", $[0629_{16}]$; "/afii57418", $[062A_{16}]$; "/afii57419",
$[062B_{16}]$; "/afii57420", $[062C_{16}]$; "/afii57421", $[062D_{16}]$; "/afii57422",
$[062E_{16}]$; "/afii57423", $[062F_{16}]$; "/afii57424", $[0630_{16}]$; "/afii57425",
$[0631_{16}]$; "/afii57426", $[0632_{16}]$; "/afii57427", $[0633_{16}]$; "/afii57428",
$[0634_{16}]$; "/afii57429", $[0635_{16}]$; "/afii57430", $[0636_{16}]$; "/afii57431",
$[0637_{16}]$; "/afii57432", $[0638_{16}]$; "/afii57433", $[0639_{16}]$; "/afii57434",
$[063A_{16}]$; "/afii57440", $[0640_{16}]$; "/afii57441", $[0641_{16}]$; "/afii57442",
$[0642_{16}]$; "/afii57443", $[0643_{16}]$; "/afii57444", $[0644_{16}]$; "/afii57445",

[0645_{16}]; "/afii57446", [0646_{16}]; "/afii57448", [0648_{16}]; "/afii57449", [0649_{16}]; "/afii57450", [$064A_{16}$]; "/afii57451", [$064B_{16}$]; "/afii57452", [$064C_{16}$]; "/afii57453", [$064D_{16}$]; "/afii57454", [$064E_{16}$]; "/afii57455", [$064F_{16}$]; "/afii57456", [0650_{16}]; "/afii57457", [0651_{16}]; "/afii57458", [0652_{16}]; "/afii57470", [0647_{16}]; "/afii57505", [$06A4_{16}$]; "/afii57506", [$067E_{16}$]; "/afii57507", [0686_{16}]; "/afii57508", [0698_{16}]; "/afii57509", [$06AF_{16}$]; "/afii57511", [0679_{16}]; "/afii57512", [0688_{16}]; "/afii57513", [0691_{16}]; "/afii57514", [$06BA_{16}$]; "/afii57519", [$06D2_{16}$]; "/afii57534", [$06D5_{16}$]; "/afii57636", [$20AA_{16}$]; "/afii57645", [$05BE_{16}$]; "/afii57658", [$05C3_{16}$]; "/afii57664", [$05D0_{16}$]; "/afii57665", [$05D1_{16}$]; "/afii57666", [$05D2_{16}$]; "/afii57667", [$05D3_{16}$]; "/afii57668", [$05D4_{16}$]; "/afii57669", [$05D5_{16}$]; "/afii57670", [$05D6_{16}$]; "/afii57671", [$05D7_{16}$]; "/afii57672", [$05D8_{16}$]; "/afii57673", [$05D9_{16}$]; "/afii57674", [$05DA_{16}$]; "/afii57675", [$05DB_{16}$]; "/afii57676", [$05DC_{16}$]; "/afii57677", [$05DD_{16}$]; "/afii57678", [$05DE_{16}$]; "/afii57679", [$05DF_{16}$]; "/afii57680", [$05E0_{16}$]; "/afii57681", [$05E1_{16}$]; "/afii57682", [$05E2_{16}$]; "/afii57683", [$05E3_{16}$]; "/afii57684", [$05E4_{16}$]; "/afii57685", [$05E5_{16}$]; "/afii57686", [$05E6_{16}$]; "/afii57687", [$05E7_{16}$]; "/afii57688", [$05E8_{16}$]; "/afii57689", [$05E9_{16}$]; "/afii57690", [$05EA_{16}$]; "/afii57694", [$FB2A_{16}$]; "/afii57695", [$FB2B_{16}$]; "/afii57700", [$FB4B_{16}$]; "/afii57705", [$FB1F_{16}$]; "/afii57716", [$05F0_{16}$]; "/afii57717", [$05F1_{16}$]; "/afii57718", [$05F2_{16}$]; "/afii57723", [$FB35_{16}$]; "/afii57793", [$05B4_{16}$]; "/afii57794", [$05B5_{16}$]; "/afii57795", [$05B6_{16}$]; "/afii57796", [$05BB_{16}$]; "/afii57797", [$05B8_{16}$]; "/afii57798", [$05B7_{16}$]; "/afii57799", [$05B0_{16}$]; "/afii57800", [$05B2_{16}$]; "/afii57801", [$05B1_{16}$]; "/afii57802", [$05B3_{16}$]; "/afii57803", [$05C2_{16}$]; "/afii57804", [$05C1_{16}$]; "/afii57806", [$05B9_{16}$]; "/afii57807", [$05BC_{16}$]; "/afii57839", [$05BD_{16}$]; "/afii57841", [$05BF_{16}$]; "/afii57842", [$05C0_{16}$]; "/afii57929", [$02BC_{16}$]; "/afii61248", [2105_{16}]; "/afii61289", [2113_{16}]; "/afii61352", [2116_{16}]; "/afii61573", [$202C_{16}$]; "/afii61574", [$202D_{16}$]; "/afii61575", [$202E_{16}$]; "/afii61664", [$200C_{16}$]; "/afii63167", [$066D_{16}$]; "/afii64937", [$02BD_{16}$]; "/agrave", [$00E0_{16}$]; "/agujarati", [$0A85_{16}$]; "/agurmukhi", [$0A05_{16}$]; "/ahiragana", [3042_{16}]; "/ahookabove", [$1EA3_{16}$]; "/aibengali", [0990_{16}]; "/aibopomofo", [$311E_{16}$]; "/aideva", [0910_{16}]; "/aiecyrillic", [$04D5_{16}$]; "/aigujarati", [$0A90_{16}$]; "/aigurmukhi", [$0A10_{16}$]; "/aimatragurmukhi", [$0A48_{16}$]; "/ainarabic", [0639_{16}]; "/ainfinalarabic", [$FECA_{16}$]; "/aininitialarabic", [$FECB_{16}$]; "/ainmedialarabic", [$FECC_{16}$]; "/ainvertedbreve", [0203_{16}]; "/aivowelsignbengali", [$09C8_{16}$]; "/aivowelsigndeva", [0948_{16}]; "/aivowelsigngujarati", [$0AC8_{16}$]; "/akatakana", [$30A2_{16}$]; "/akatakanahalfwidth", [$FF71_{16}$]; "/akorean", [$314F_{16}$]; "/alef", [$05D0_{16}$]; "/alefarabic", [0627_{16}]; "/alefdageshhebrew", [$FB30_{16}$]; "/aleffinalarabic", [$FE8E_{16}$]; "/alefhamzaabovearabic", [0623_{16}]; "/alefhamzaabovefinalarabic", [$FE84_{16}$]; "/alefhamzabelowarabic", [0625_{16}]; "/alefhamzabelowfinalarabic", [$FE88_{16}$]; "/alefhebrew", [$05D0_{16}$]; "/aleflamedhebrew", [$FB4F_{16}$]; "/alefmaddaabovearabic", [0622_{16}]; "/alefmaddaabovefinalarabic", [$FE82_{16}$]; "/alefmaksuraarabic", [0649_{16}]; "/alefmaksurafinalarabic", [$FEF0_{16}$]; "/alefmaksurainitialarabic", [$FEF3_{16}$]; "/alefmaksuramedialarabic", [$FEF4_{16}$]; "/alefpatahhebrew", [$FB2E_{16}$]; "/alefqamatshebrew", [$FB2F_{16}$]; "/aleph", [2135_{16}]; "/allequal", [$224C_{16}$]; "/alpha", [$03B1_{16}$]; "/alphatonos", [$03AC_{16}$]; "/amacron", [0101_{16}]; "/amonospace", [$FF41_{16}$]; "/ampersand", [0026_{16}]; "/ampersandmonospace", [$FF06_{16}$]; "/ampersandsmall", [$F726_{16}$]; "/amsquare", [$33C2_{16}$]; "/anbopomofo", [3122$_{16}$]; "/angbopomofo", [3124$_{16}$]; "/angkhankhuthai", [0E5A$_{16}$]; "/angle", [2220$_{16}$]; "/anglebracketleft", [3008$_{16}$]; "/anglebracketleftvertical", [FE3F$_{16}$]; "/anglebracketright", [3009$_{16}$]; "/anglebracketrightvertical", [FE40$_{16}$]; "/angleleft", [2329$_{16}$]; "/angleright", [232A$_{16}$]; "/angstrom", [212B$_{16}$]; "/anoteleia", [0387$_{16}$]; "/anudattadeva", [0952$_{16}$]; "/anusvarabengali", [0982$_{16}$]; "/anusvaradeva", [0902$_{16}$]; "/anusvaragujarati", [0A82$_{16}$]; "/aogonek", [0105$_{16}$]; "/apaatosquare", [3300$_{16}$]; "/aparen", [249C$_{16}$]; "/apostrophearmenian", [055A$_{16}$]; "/apostrophemod", [02BC$_{16}$]; "/apple", [F8FF$_{16}$]; "/approaches", [2250$_{16}$]; "/approxequal", [2248$_{16}$]; "/approxequalorimage", [2252$_{16}$]; "/approximatelyequal", [2245$_{16}$]; "/araeaekorean", [318E$_{16}$]; "/araeakorean", [318D$_{16}$]; "/arc", [2312$_{16}$]; "/arighthalfring", [1E9A$_{16}$]; "/aring", [00E5$_{16}$]; "/aringacute", [01FB$_{16}$]; "/aringbelow", [1E01$_{16}$]; "/arrowboth", [2194$_{16}$]; "/arrowdashdown", [21E3$_{16}$]; "/arrowdashleft", [21E0$_{16}$]; "/arrowdashright", [21E2$_{16}$]; "/arrowdashup", [21E1$_{16}$]; "/arrowdblboth", [21D4$_{16}$]; "/arrowdbldown", [21D3$_{16}$]; "/arrowdblleft", [21D0$_{16}$]; "/arrowdblright", [21D2$_{16}$]; "/arrowdblup", [21D1$_{16}$]; "/arrowdown", [2193$_{16}$]; "/arrowdownleft", [2199$_{16}$]; "/arrowdownright", [2198$_{16}$]; "/arrowdownwhite", [21E9$_{16}$]; "/arrowheaddownmod", [02C5$_{16}$]; "/arrowheadleftmod", [02C2$_{16}$]; "/arrowheadrightmod", [02C3$_{16}$]; "/arrowheadupmod", [02C4$_{16}$]; "/arrowhorizex", [F8E7$_{16}$]; "/arrowleft", [2190$_{16}$]; "/arrowleftdbl", [21D0$_{16}$]; "/arrowleftdblstroke", [21CD$_{16}$]; "/arrowleftoverright", [21C6$_{16}$]; "/arrowleftwhite", [21E6$_{16}$]; "/arrowright", [2192$_{16}$]; "/arrowrightdblstroke", [21CF$_{16}$]; "/arrowrightheavy", [279E$_{16}$]; "/arrowrightoverleft", [21C4$_{16}$]; "/arrowrightwhite", [21E8$_{16}$]; "/arrowtableft", [21E4$_{16}$]; "/arrowtabright", [21E5$_{16}$]; "/arrowup", [2191$_{16}$]; "/arrowupdn", [2195$_{16}$]; "/arrowupdnbse", [21A8$_{16}$]; "/arrowupdownbase", [21A8$_{16}$]; "/arrowupleft", [2196$_{16}$]; "/arrowupleftofdown", [21C5$_{16}$]; "/arrowupright", [2197$_{16}$]; "/arrowupwhite", [21E7$_{16}$]; "/arrowvertex", [F8E6$_{16}$]; "/asciicircum", [005E$_{16}$]; "/asciicircummonospace", [FF3E$_{16}$]; "/asciitilde", [007E$_{16}$]; "/asciitildemonospace", [FF5E$_{16}$]; "/ascript", [0251$_{16}$]; "/ascriptturned", [0252$_{16}$]; "/asmallhiragana", [3041$_{16}$]; "/asmallkatakana", [30A1$_{16}$]; "/asmallkatakanahalfwidth", [FF67$_{16}$]; "/asterisk", [002A$_{16}$]; "/asteriskaltonearabic", [066D$_{16}$]; "/asteriskarabic", [066D$_{16}$]; "/asteriskmath", [2217$_{16}$]; "/asteriskmonospace", [FF0A$_{16}$]; "/asterisksmall", [FE61$_{16}$]; "/asterism", [2042$_{16}$]; "/asuperior", [F6E9$_{16}$]; "/asymptoticallyequal", [2243$_{16}$]; "/at", [0040$_{16}$]; "/atilde", [00E3$_{16}$]; "/atmonospace", [FF20$_{16}$]; "/atsmall", [FE6B$_{16}$]; "/aturned", [0250$_{16}$]; "/aubengali", [0994$_{16}$]; "/aubopomofo", [3120$_{16}$]; "/audeva", [0914$_{16}$]; "/augujarati", [0A94$_{16}$]; "/augurmukhi", [0A14$_{16}$]; "/aulengthmarkbengali", [09D7$_{16}$]; "/aumatragurmukhi", [0A4C$_{16}$]; "/auvowelsignbengali", [09CC$_{16}$]; "/auvowelsigndeva", [094C$_{16}$]; "/auvowelsigngujarati", [0ACC$_{16}$]; "/avagrahadeva", [093D$_{16}$]; "/aybarmenian", [0561$_{16}$]; "/ayin", [05E2$_{16}$]; "/ayinaltonehebrew", [FB20$_{16}$]; "/ayinhebrew", [05E2$_{16}$]; "/b", [0062$_{16}$]; "/babengali", [09AC$_{16}$]; "/backslash", [005C$_{16}$]; "/backslashmonospace", [FF3C$_{16}$]; "/badeva", [092C$_{16}$]; "/bagujarati", [0AAC$_{16}$]; "/bagurmukhi", [0A2C$_{16}$]; "/bahiragana", [3070$_{16}$]; "/bahtthai", [0E3F$_{16}$]; "/bakatakana", [30D0$_{16}$]; "/bar", [007C$_{16}$]; "/barmonospace", [FF5C$_{16}$]; "/bbopomofo", [3105$_{16}$]; "/bcircle", [24D1$_{16}$]; "/bdotaccent", [1E03$_{16}$]; "/bdotbelow", [1E05$_{16}$]; "/beamedsixteenthnotes", [266C$_{16}$]; "/because", [2235$_{16}$]; "/becyrillic", [0431$_{16}$]; "/beharabic", [0628$_{16}$]; "/behfinalarabic", [FE90$_{16}$]; "/behinitialarabic", [FE91$_{16}$]; "/behiragana", [3079$_{16}$]; "/behmedialarabic", [FE92$_{16}$]; "/behmeeminitialarabic", [FC9F$_{16}$]; "/behmeemisolatedarabic", [FC08$_{16}$];

"/behnoonfinalarabic", [$FC6D_{16}$]; "/bekatakana", [$30D9_{16}$]; "/benarmenian", [0562_{16}]; "/bet", [$05D1_{16}$]; "/beta", [$03B2_{16}$]; "/betasymbolgreek", [$03D0_{16}$]; "/betdagesh", [$FB31_{16}$]; "/betdageshhebrew", [$FB31_{16}$]; "/bethebrew", [$05D1_{16}$]; "/betrafehebrew", [$FB4C_{16}$]; "/bhabengali", [$09AD_{16}$]; "/bhadeva", [$092D_{16}$]; "/bhagujarati", [$0AAD_{16}$]; "/bhagurmukhi", [$0A2D_{16}$]; "/bhook", [0253_{16}]; "/bihiragana", [3073_{16}]; "/bikatakana", [$30D3_{16}$]; "/bilabialclick", [0298_{16}]; "/bindigurmukhi", [$0A02_{16}$]; "/birusquare", [3331_{16}]; "/blackcircle", [$25CF_{16}$]; "/blackdiamond", [$25C6_{16}$]; "/blackdownpointingtriangle", [$25BC_{16}$]; "/blackleftpointingpointer", [$25C4_{16}$]; "/blackleftpointingtriangle", [$25C0_{16}$]; "/blacklenticularbracketleft", [3010_{16}]; "/blacklenticularbracketleftvertical", [$FE3B_{16}$]; "/blacklenticularbracketright", [3011_{16}]; "/blacklenticularbracketrightvertical", [$FE3C_{16}$]; "/blacklowerlefttriangle", [$25E3_{16}$]; "/blacklowerrighttriangle", [$25E2_{16}$]; "/blackrectangle", [$25AC_{16}$]; "/blackrightpointingpointer", [$25BA_{16}$]; "/blackrightpointingtriangle", [$25B6_{16}$]; "/blacksmallsquare", [$25AA_{16}$]; "/blacksmilingface", [$263B_{16}$]; "/blacksquare", [$25A0_{16}$]; "/blackstar", [2605_{16}]; "/blackupperlefttriangle", [$25E4_{16}$]; "/blackupperrighttriangle", [$25E5_{16}$]; "/blackuppointingsmalltriangle", [$25B4_{16}$]; "/blackuppointingtriangle", [$25B2_{16}$]; "/blank", [2423_{16}]; "/blinebelow", [$1E07_{16}$]; "/block", [2588_{16}]; "/bmonospace", [$FF42_{16}$]; "/bobaimaithai", [$0E1A_{16}$]; "/bohiragana", [$307C_{16}$]; "/bokatakana", [$30DC_{16}$]; "/bparen", [$249D_{16}$]; "/bqsquare", [$33C3_{16}$]; "/braceex", [$F8F4_{16}$]; "/braceleft", [$007B_{16}$]; "/braceleftbt", [$F8F3_{16}$]; "/braceleftmid", [$F8F2_{16}$]; "/braceleftmonospace", [$FF5B_{16}$]; "/braceleftsmall", [$FE5B_{16}$]; "/bracelefttp", [$F8F1_{16}$]; "/braceleftvertical", [$FE37_{16}$]; "/braceright", [$007D_{16}$]; "/bracerightbt", [$F8FE_{16}$]; "/bracerightmid", [$F8FD_{16}$]; "/bracerightmonospace", [$FF5D_{16}$]; "/bracerightsmall", [$FE5C_{16}$]; "/bracerighttp", [$F8FC_{16}$]; "/bracerightvertical", [$FE38_{16}$]; "/bracketleft", [$005B_{16}$]; "/bracketleftbt", [$F8F0_{16}$]; "/bracketleftex", [$F8EF_{16}$]; "/bracketleftmonospace", [$FF3B_{16}$]; "/bracketlefttp", [$F8EE_{16}$]; "/bracketright", [$005D_{16}$]; "/bracketrightbt", [$F8FB_{16}$]; "/bracketrightex", [$F8FA_{16}$]; "/bracketrightmonospace", [$FF3D_{16}$]; "/bracketrighttp", [$F8F9_{16}$]; "/breve", [$02D8_{16}$]; "/brevebelowcmb", [$032E_{16}$]; "/brevecmb", [0306_{16}]; "/breveinvertedbelowcmb", [$032F_{16}$]; "/breveinvertedcmb", [0311_{16}]; "/breveinverteddoublecmb", [0361_{16}]; "/bridgebelowcmb", [$032A_{16}$]; "/bridgeinvertedbelowcmb", [$033A_{16}$]; "/brokenbar", [$00A6_{16}$]; "/bstroke", [0180_{16}]; "/bsuperior", [$F6EA_{16}$]; "/btopbar", [0183_{16}]; "/buhiragana", [3076_{16}]; "/bukatakana", [$30D6_{16}$]; "/bullet", [2022_{16}]; "/bulletinverse", [$25D8_{16}$]; "/bulletoperator", [2219_{16}]; "/bullseye", [$25CE_{16}$]; "/c", [0063_{16}]; "/caarmenian", [$056E_{16}$]; "/cabengali", [$099A_{16}$]; "/cacute", [0107_{16}]; "/cadeva", [$091A_{16}$]; "/cagujarati", [$0A9A_{16}$]; "/cagurmukhi", [$0A1A_{16}$]; "/calsquare", [3388_{16}]; "/candrabindubengali", [0981_{16}]; "/candrabinducmb", [0310_{16}]; "/candrabindudeva", [0901_{16}]; "/candrabindugujarati", [$0A81_{16}$]; "/capslock", [$21EA_{16}$]; "/careof", [2105_{16}]; "/caron", [$02C7_{16}$]; "/caronbelowcmb", [$032C_{16}$]; "/caroncmb", [$030C_{16}$]; "/carriagereturn", [$21B5_{16}$]; "/cbopomofo", [3118_{16}]; "/ccaron", [$010D_{16}$]; "/ccedilla", [$00E7_{16}$]; "/ccedillaacute", [$1E09_{16}$]; "/ccircle", [$24D2_{16}$]; "/ccircumflex", [0109_{16}]; "/ccurl", [0255_{16}]; "/cdot", [$010B_{16}$]; "/cdotaccent", [$010B_{16}$]; "/cdsquare", [$33C5_{16}$]; "/cedilla", [$00B8_{16}$]; "/cedillacmb", [0327_{16}]; "/cent", [$00A2_{16}$]; "/centigrade", [2103_{16}]; "/centinferior", [$F6DF_{16}$]; "/centmonospace", [$FFE0_{16}$]; "/centoldstyle", [$F7A2_{16}$]; "/centsuperior", [$F6E0_{16}$]; "/chaarmenian", [0579_{16}]; "/chabengali", [$099B_{16}$]; "/chadeva", [$091B_{16}$]; "/chagujarati", [$0A9B_{16}$]; "/chagurmukhi", [0A1B$_{16}$]; "/chbopomofo", [3114$_{16}$]; "/cheabkhasiancyrillic", [04BD$_{16}$]; "/checkmark", [2713$_{16}$]; "/checyrillic", [0447$_{16}$]; "/chedescenderabkhasiancyrillic", [04BF$_{16}$]; "/chedescendercyrillic", [04B7$_{16}$]; "/chedieresiscyrillic", [04F5$_{16}$]; "/cheharmenian", [0573$_{16}$]; "/chekhakassiancyrillic", [04CC$_{16}$]; "/cheverticalstrokecyrillic", [04B9$_{16}$]; "/chi", [03C7$_{16}$]; "/chieuchacirclekorean", [3277$_{16}$]; "/chieuchaparenkorean", [3217$_{16}$]; "/chieuchcirclekorean", [3269$_{16}$]; "/chieuchkorean", [314A$_{16}$]; "/chieuchparenkorean", [3209$_{16}$]; "/chochangthai", [0E0A$_{16}$]; "/chochanthai", [0E08$_{16}$]; "/chochingthai", [0E09$_{16}$]; "/chochoethai", [0E0C$_{16}$]; "/chook", [0188$_{16}$]; "/cieucacirclekorean", [3276$_{16}$]; "/cieucaparenkorean", [3216$_{16}$]; "/cieuccirclekorean", [3268$_{16}$]; "/cieuckorean", [3148$_{16}$]; "/cieucparenkorean", [3208$_{16}$]; "/cieucuparenkorean", [321C$_{16}$]; "/circle", [25CB$_{16}$]; "/circlemultiply", [2297$_{16}$]; "/circleot", [2299$_{16}$]; "/circleplus", [2295$_{16}$]; "/circlepostalmark", [3036$_{16}$]; "/circlewithlefthalfblack", [25D0$_{16}$]; "/circlewithrighthalfblack", [25D1$_{16}$]; "/circumflex", [02C6$_{16}$]; "/circumflexbelowcmb", [032D$_{16}$]; "/circumflexcmb", [0302$_{16}$]; "/clear", [2327$_{16}$]; "/clickalveolar", [01C2$_{16}$]; "/clickdental", [01C0$_{16}$]; "/clicklateral", [01C1$_{16}$]; "/clickretroflex", [01C3$_{16}$]; "/club", [2663$_{16}$]; "/clubsuitblack", [2663$_{16}$]; "/clubsuitwhite", [2667$_{16}$]; "/cmcubedsquare", [33A4$_{16}$]; "/cmonospace", [FF43$_{16}$]; "/cmsquaredsquare", [33A0$_{16}$]; "/coarmenian", [0581$_{16}$]; "/colon", [003A$_{16}$]; "/colonmonetary", [20A1$_{16}$]; "/colonmonospace", [FF1A$_{16}$]; "/colonsign", [20A1$_{16}$]; "/colonsmall", [FE55$_{16}$]; "/colontriangularhalfmod", [02D1$_{16}$]; "/colontriangularmod", [02D0$_{16}$]; "/comma", [002C$_{16}$]; "/commaabovecmb", [0313$_{16}$]; "/commaaboverightcmb", [0315$_{16}$]; "/commaaccent", [F6C3$_{16}$]; "/commaarabic", [060C$_{16}$]; "/commaarmenian", [055D$_{16}$]; "/commainferior", [F6E1$_{16}$]; "/commamonospace", [FF0C$_{16}$]; "/commareversedabovecmb", [0314$_{16}$]; "/commareversedmod", [02BD$_{16}$]; "/commasmall", [FE50$_{16}$]; "/commasuperior", [F6E2$_{16}$]; "/commaturnedabovecmb", [0312$_{16}$]; "/commaturnedmod", [02BB$_{16}$]; "/compass", [263C$_{16}$]; "/congruent", [2245$_{16}$]; "/contourintegral", [222E$_{16}$]; "/control", [2303$_{16}$]; "/controlACK", [0006$_{16}$]; "/controlBEL", [0007$_{16}$]; "/controlBS", [0008$_{16}$]; "/controlCAN", [0018$_{16}$]; "/controlCR", [000D$_{16}$]; "/controlDC1", [0011$_{16}$]; "/controlDC2", [0012$_{16}$]; "/controlDC3", [0013$_{16}$]; "/controlDC4", [0014$_{16}$]; "/controlDEL", [007F$_{16}$]; "/controlDLE", [0010$_{16}$]; "/controlEM", [0019$_{16}$]; "/controlENQ", [0005$_{16}$]; "/controlEOT", [0004$_{16}$]; "/controlESC", [001B$_{16}$]; "/controlETB", [0017$_{16}$]; "/controlETX", [0003$_{16}$]; "/controlFF", [000C$_{16}$]; "/controlFS", [001C$_{16}$]; "/controlGS", [001D$_{16}$]; "/controlHT", [0009$_{16}$]; "/controlLF", [000A$_{16}$]; "/controlNAK", [0015$_{16}$]; "/controlRS", [001E$_{16}$]; "/controlSI", [000F$_{16}$]; "/controlSO", [000E$_{16}$]; "/controlSOT", [0002$_{16}$]; "/controlSTX", [0001$_{16}$]; "/controlSUB", [001A$_{16}$]; "/controlSYN", [0016$_{16}$]; "/controlUS", [001F$_{16}$]; "/controlVT", [000B$_{16}$]; "/copyright", [00A9$_{16}$]; "/copyrightsans", [F8E9$_{16}$]; "/copyrightserif", [F6D9$_{16}$]; "/cornerbracketleft", [300C$_{16}$]; "/cornerbracketlefthalfwidth", [FF62$_{16}$]; "/cornerbracketleftvertical", [FE41$_{16}$]; "/cornerbracketright", [300D$_{16}$]; "/cornerbracketrighthalfwidth", [FF63$_{16}$]; "/cornerbracketrightvertical", [FE42$_{16}$]; "/corporationsquare", [337F$_{16}$]; "/cosquare", [33C7$_{16}$]; "/coverkgsquare", [33C6$_{16}$]; "/cparen", [249E$_{16}$]; "/cruzeiro", [20A2$_{16}$]; "/cstretched", [0297$_{16}$]; "/curlyand", [22CF$_{16}$]; "/curlyor", [22CE$_{16}$]; "/currency", [00A4$_{16}$]; "/cyrBreve", [F6D1$_{16}$]; "/cyrFlex", [F6D2$_{16}$]; "/cyrbreve", [F6D4$_{16}$]; "/cyrflex", [F6D5$_{16}$]; "/d", [0064$_{16}$]; "/daarmenian", [0564$_{16}$]; "/dabengali", [09A6$_{16}$]; "/dadarabic", [0636$_{16}$]; "/dadeva", [0926$_{16}$];

"/dadfinalarabic", [FEBE$_{16}$]; "/dadinitialarabic", [FEBF$_{16}$]; "/dadmedialarabic", [FEC0$_{16}$]; "/dagesh", [05BC$_{16}$]; "/dageshhebrew", [05BC$_{16}$]; "/dagger", [2020$_{16}$]; "/daggerdbl", [2021$_{16}$]; "/dagujarati", [0AA6$_{16}$]; "/dagurmukhi", [0A26$_{16}$]; "/dahiragana", [3060$_{16}$]; "/dakatakana", [30C0$_{16}$]; "/dalarabic", [062F$_{16}$]; "/dalet", [05D3$_{16}$]; "/daletdagesh", [FB33$_{16}$]; "/daletdageshhebrew", [FB33$_{16}$]; "/dalethatafpatah", [05D3$_{16}$; 05B2$_{16}$]; "/dalethatafpatahhebrew", [05D3$_{16}$; 05B2$_{16}$]; "/dalethatafsegol", [05D3$_{16}$; 05B1$_{16}$]; "/dalethatafsegolhebrew", [05D3$_{16}$; 05B1$_{16}$]; "/dalethebrew", [05D3$_{16}$]; "/dalethiriq", [05D3$_{16}$; 05B4$_{16}$]; "/dalethiriqhebrew", [05D3$_{16}$; 05B4$_{16}$]; "/daletholam", [05D3$_{16}$; 05B9$_{16}$]; "/daletholamhebrew", [05D3$_{16}$; 05B9$_{16}$]; "/daletpatah", [05D3$_{16}$; 05B7$_{16}$]; "/daletpatahhebrew", [05D3$_{16}$; 05B7$_{16}$]; "/daletqamats", [05D3$_{16}$; 05B8$_{16}$]; "/daletqamatshebrew", [05D3$_{16}$; 05B8$_{16}$]; "/daletqubuts", [05D3$_{16}$; 05BB$_{16}$]; "/daletqubutshebrew", [05D3$_{16}$; 05BB$_{16}$]; "/daletsegol", [05D3$_{16}$; 05B6$_{16}$]; "/daletsegolhebrew", [05D3$_{16}$; 05B6$_{16}$]; "/daletsheva", [05D3$_{16}$; 05B0$_{16}$]; "/daletshevahebrew", [05D3$_{16}$; 05B0$_{16}$]; "/dalettsere", [05D3$_{16}$; 05B5$_{16}$]; "/dalettserehebrew", [05D3$_{16}$; 05B5$_{16}$]; "/dalfinalarabic", [FEAA$_{16}$]; "/dammaarabic", [064F$_{16}$]; "/dammalowarabic", [064F$_{16}$]; "/dammatanaltonearabic", [064C$_{16}$]; "/dammatanarabic", [064C$_{16}$]; "/danda", [0964$_{16}$]; "/dargahebrew", [05A7$_{16}$]; "/dargalefthebrew", [05A7$_{16}$]; "/dasiapneumatacyrilliccmb", [0485$_{16}$]; "/dblGrave", [F6D3$_{16}$]; "/dblanglebracketleft", [300A$_{16}$]; "/dblanglebracketleftvertical", [FE3D$_{16}$]; "/dblanglebracketright", [300B$_{16}$]; "/dblanglebracketrightvertical", [FE3E$_{16}$]; "/dblarchinvertedbelowcmb", [032B$_{16}$]; "/dblarrowleft", [21D4$_{16}$]; "/dblarrowright", [21D2$_{16}$]; "/dbldanda", [0965$_{16}$]; "/dblgrave", [F6D6$_{16}$]; "/dblgravecmb", [030F$_{16}$]; "/dblintegral", [222C$_{16}$]; "/dbllowline", [2017$_{16}$]; "/dbllowlinecmb", [0333$_{16}$]; "/dbloverlinecmb", [033F$_{16}$]; "/dblprimemod", [02BA$_{16}$]; "/dblverticalbar", [2016$_{16}$]; "/dblverticallineabovecmb", [030E$_{16}$]; "/dbopomofo", [3109$_{16}$]; "/dbsquare", [33C8$_{16}$]; "/dcaron", [010F$_{16}$]; "/dcedilla", [1E11$_{16}$]; "/dcircle", [24D3$_{16}$]; "/dcircumflexbelow", [1E13$_{16}$]; "/dcroat", [0111$_{16}$]; "/ddabengali", [09A1$_{16}$]; "/ddadeva", [0921$_{16}$]; "/ddagujarati", [0AA1$_{16}$]; "/ddagurmukhi", [0A21$_{16}$]; "/ddalarabic", [0688$_{16}$]; "/ddalfinalarabic", [FB89$_{16}$]; "/dddhadeva", [095C$_{16}$]; "/ddhabengali", [09A2$_{16}$]; "/ddhadeva", [0922$_{16}$]; "/ddhagujarati", [0AA2$_{16}$]; "/ddhagurmukhi", [0A22$_{16}$]; "/ddotaccent", [1E0B$_{16}$]; "/ddotbelow", [1E0D$_{16}$]; "/decimalseparatorarabic", [066B$_{16}$]; "/decimalseparatorpersian", [066B$_{16}$]; "/decyrillic", [0434$_{16}$]; "/degree", [00B0$_{16}$]; "/dehihebrew", [05AD$_{16}$]; "/dehiragana", [3067$_{16}$]; "/deicoptic", [03EF$_{16}$]; "/dekatakana", [30C7$_{16}$]; "/deleteleft", [232B$_{16}$]; "/deleteright", [2326$_{16}$]; "/delta", [03B4$_{16}$]; "/deltaturned", [018D$_{16}$]; "/denominatorminusonenumeratorbengali", [09F8$_{16}$]; "/dezh", [02A4$_{16}$]; "/dhabengali", [09A7$_{16}$]; "/dhadeva", [0927$_{16}$]; "/dhagujarati", [0AA7$_{16}$]; "/dhagurmukhi", [0A27$_{16}$]; "/dhook", [0257$_{16}$]; "/dialytikatonos", [0385$_{16}$]; "/dialytikatonoscmb", [0344$_{16}$]; "/diamond", [2666$_{16}$]; "/diamondsuitwhite", [2662$_{16}$]; "/dieresis", [00A8$_{16}$]; "/dieresisacute", [F6D7$_{16}$]; "/dieresisbelowcmb", [0324$_{16}$]; "/dieresiscmb", [0308$_{16}$]; "/dieresisgrave", [F6D8$_{16}$]; "/dieresistonos", [0385$_{16}$]; "/dihiragana", [3062$_{16}$]; "/dikatakana", [30C2$_{16}$]; "/dittomark", [3003$_{16}$]; "/divide", [00F7$_{16}$]; "/divides", [2223$_{16}$]; "/divisionslash", [2215$_{16}$]; "/djecyrillic", [0452$_{16}$]; "/dkshade", [2593$_{16}$]; "/dlinebelow", [1E0F$_{16}$]; "/dlsquare", [3397$_{16}$]; "/dmacron", [0111$_{16}$]; "/dmonospace", [FF44$_{16}$]; "/dnblock", [2584$_{16}$]; "/dochadathai", [0E0E$_{16}$]; "/dodekthai", [0E14$_{16}$]; "/dohiragana", [3069$_{16}$]; "/dokatakana", [30C9$_{16}$]; "/dollar", [0024$_{16}$]; "/dollarinferior", [F6E3$_{16}$];

"/dollarmonospace", [FF04$_{16}$]; "/dollaroldstyle", [F724$_{16}$]; "/dollarsmall", [FE69$_{16}$]; "/dollarsuperior", [F6E4$_{16}$]; "/dong", [20AB$_{16}$]; "/dorusquare", [3326$_{16}$]; "/dotaccent", [02D9$_{16}$]; "/dotaccentcmb", [0307$_{16}$]; "/dotbelowcmb", [0323$_{16}$]; "/dotbelowcomb", [0323$_{16}$]; "/dotkatakana", [30FB$_{16}$]; "/dotlessi", [0131$_{16}$]; "/dotlessj", [F6BE$_{16}$]; "/dotlessjstrokehook", [0284$_{16}$]; "/dotmath", [22C5$_{16}$]; "/dottedcircle", [25CC$_{16}$]; "/doubleyodpatah", [FB1F$_{16}$]; "/doubleyodpatahhebrew", [FB1F$_{16}$]; "/downtackbelowcmb", [031E$_{16}$]; "/downtackmod", [02D5$_{16}$]; "/dparen", [249F$_{16}$]; "/dsuperior", [F6EB$_{16}$]; "/dtail", [0256$_{16}$]; "/dtopbar", [018C$_{16}$]; "/duhiragana", [3065$_{16}$]; "/dukatakana", [30C5$_{16}$]; "/dz", [01F3$_{16}$]; "/dzaltone", [02A3$_{16}$]; "/dzcaron", [01C6$_{16}$]; "/dzcurl", [02A5$_{16}$]; "/dzeabkhasiancyrillic", [04E1$_{16}$]; "/dzecyrillic", [0455$_{16}$]; "/dzhecyrillic", [045F$_{16}$]; "/e", [0065$_{16}$]; "/eacute", [00E9$_{16}$]; "/earth", [2641$_{16}$]; "/ebengali", [098F$_{16}$]; "/ebopomofo", [311C$_{16}$]; "/ebreve", [0115$_{16}$]; "/ecandradeva", [090D$_{16}$]; "/ecandragujarati", [0A8D$_{16}$]; "/ecandravowelsigndeva", [0945$_{16}$]; "/ecandravowelsigngujarati", [0AC5$_{16}$]; "/ecaron", [011B$_{16}$]; "/ecedillabreve", [1E1D$_{16}$]; "/echarmenian", [0565$_{16}$]; "/echyiwnarmenian", [0587$_{16}$]; "/ecircle", [24D4$_{16}$]; "/ecircumflex", [00EA$_{16}$]; "/ecircumflexacute", [1EBF$_{16}$]; "/ecircumflexbelow", [1E19$_{16}$]; "/ecircumflexdotbelow", [1EC7$_{16}$]; "/ecircumflexgrave", [1EC1$_{16}$]; "/ecircumflexhookabove", [1EC3$_{16}$]; "/ecircumflextilde", [1EC5$_{16}$]; "/ecyrillic", [0454$_{16}$]; "/edblgrave", [0205$_{16}$]; "/edeva", [090F$_{16}$]; "/edieresis", [00EB$_{16}$]; "/edot", [0117$_{16}$]; "/edotaccent", [0117$_{16}$]; "/edotbelow", [1EB9$_{16}$]; "/eegurmukhi", [0A0F$_{16}$]; "/eematragurmukhi", [0A47$_{16}$]; "/efcyrillic", [0444$_{16}$]; "/egrave", [00E8$_{16}$]; "/egujarati", [0A8F$_{16}$]; "/eharmenian", [0567$_{16}$]; "/ehbopomofo", [311D$_{16}$]; "/ehiragana", [3048$_{16}$]; "/ehookabove", [1EBB$_{16}$]; "/eibopomofo", [311F$_{16}$]; "/eight", [0038$_{16}$]; "/eightarabic", [0668$_{16}$]; "/eightbengali", [09EE$_{16}$]; "/eightcircle", [2467$_{16}$]; "/eightcircleinversesansserif", [2791$_{16}$]; "/eightdeva", [096E$_{16}$]; "/eighteencircle", [2471$_{16}$]; "/eighteenparen", [2485$_{16}$]; "/eighteenperiod", [2499$_{16}$]; "/eightgujarati", [0AEE$_{16}$]; "/eightgurmukhi", [0A6E$_{16}$]; "/eighthackarabic", [0668$_{16}$]; "/eighthangzhou", [3028$_{16}$]; "/eighthnotebeamed", [266B$_{16}$]; "/eightideographicparen", [3227$_{16}$]; "/eightinferior", [2088$_{16}$]; "/eightmonospace", [FF18$_{16}$]; "/eightoldstyle", [F738$_{16}$]; "/eightparen", [247B$_{16}$]; "/eightperiod", [248F$_{16}$]; "/eightpersian", [06F8$_{16}$]; "/eightroman", [2177$_{16}$]; "/eightsuperior", [2078$_{16}$]; "/eightthai", [0E58$_{16}$]; "/einvertedbreve", [0207$_{16}$]; "/eiotifiedcyrillic", [0465$_{16}$]; "/ekatakana", [30A8$_{16}$]; "/ekatakanahalfwidth", [FF74$_{16}$]; "/ekonkargurmukhi", [0A74$_{16}$]; "/ekorean", [3154$_{16}$]; "/elcyrillic", [043B$_{16}$]; "/element", [2208$_{16}$]; "/elevencircle", [246A$_{16}$]; "/elevenparen", [247E$_{16}$]; "/elevenperiod", [2492$_{16}$]; "/elevenroman", [217A$_{16}$]; "/ellipsis", [2026$_{16}$]; "/ellipsisvertical", [22EE$_{16}$]; "/emacron", [0113$_{16}$]; "/emacronacute", [1E17$_{16}$]; "/emacrongrave", [1E15$_{16}$]; "/emcyrillic", [043C$_{16}$]; "/emdash", [2014$_{16}$]; "/emdashvertical", [FE31$_{16}$]; "/emonospace", [FF45$_{16}$]; "/emphasismarkarmenian", [055B$_{16}$]; "/emptyset", [2205$_{16}$]; "/enbopomofo", [3123$_{16}$]; "/encyrillic", [043D$_{16}$]; "/endash", [2013$_{16}$]; "/endashvertical", [FE32$_{16}$]; "/endescendercyrillic", [04A3$_{16}$]; "/eng", [014B$_{16}$]; "/engbopomofo", [3125$_{16}$]; "/enghecyrillic", [04A5$_{16}$]; "/enhookcyrillic", [04C8$_{16}$]; "/enspace", [2002$_{16}$]; "/eogonek", [0119$_{16}$]; "/eokorean", [3153$_{16}$]; "/eopen", [025B$_{16}$]; "/eopenclosed", [029A$_{16}$]; "/eopenreversed", [025C$_{16}$]; "/eopenreversedclosed", [025E$_{16}$]; "/eopenreversedhook", [025D$_{16}$]; "/eparen", [24A0$_{16}$]; "/epsilon", [03B5$_{16}$]; "/epsilontonos", [03AD$_{16}$]; "/equal", [003D$_{16}$];

"/equalmonospace", [FF1D$_{16}$]; "/equalsmall", [FE66$_{16}$]; "/equalsuperior", [207C$_{16}$]; "/equivalence", [2261$_{16}$]; "/erbopomofo", [3126$_{16}$]; "/ercyrillic", [0440$_{16}$]; "/ereversed", [0258$_{16}$]; "/ereversedcyrillic", [044D$_{16}$]; "/escyrillic", [0441$_{16}$]; "/esdescendercyrillic", [04AB$_{16}$]; "/esh", [0283$_{16}$]; "/eshcurl", [0286$_{16}$]; "/eshortdeva", [090E$_{16}$]; "/eshortvowelsigndeva", [0946$_{16}$]; "/eshreversedloop", [01AA$_{16}$]; "/eshsquatreversed", [0285$_{16}$]; "/esmallhiragana", [3047$_{16}$]; "/esmallkatakana", [30A7$_{16}$]; "/esmallkatakanahalfwidth", [FF6A$_{16}$]; "/estimated", [212E$_{16}$]; "/esuperior", [F6EC$_{16}$]; "/eta", [03B7$_{16}$]; "/etarmenian", [0568$_{16}$]; "/etatonos", [03AE$_{16}$]; "/eth", [00F0$_{16}$]; "/etilde", [1EBD$_{16}$]; "/etildebelow", [1E1B$_{16}$]; "/etnahtafoukhhebrew", [0591$_{16}$]; "/etnahtafoukhlefthebrew", [0591$_{16}$]; "/etnahtahebrew", [0591$_{16}$]; "/etnahtalefthebrew", [0591$_{16}$]; "/eturned", [01DD$_{16}$]; "/eukorean", [3161$_{16}$]; "/euro", [20AC$_{16}$]; "/evowelsignbengali", [09C7$_{16}$]; "/evowelsigndeva", [0947$_{16}$]; "/evowelsigngujarati", [0AC7$_{16}$]; "/exclam", [0021$_{16}$]; "/exclamarmenian", [055C$_{16}$]; "/exclamdbl", [203C$_{16}$]; "/exclamdown", [00A1$_{16}$]; "/exclamdownsmall", [F7A1$_{16}$]; "/exclammonospace", [FF01$_{16}$]; "/exclamsmall", [F721$_{16}$]; "/existential", [2203$_{16}$]; "/ezh", [0292$_{16}$]; "/ezhcaron", [01EF$_{16}$]; "/ezhcurl", [0293$_{16}$]; "/ezhreversed", [01B9$_{16}$]; "/ezhtail", [01BA$_{16}$]; "/f", [0066$_{16}$]; "/fadeva", [095E$_{16}$]; "/fagurmukhi", [0A5E$_{16}$]; "/fahrenheit", [2109$_{16}$]; "/fathaarabic", [064E$_{16}$]; "/fathalowarabic", [064E$_{16}$]; "/fathatanarabic", [064B$_{16}$]; "/fbopomofo", [3108$_{16}$]; "/fcircle", [24D5$_{16}$]; "/fdotaccent", [1E1F$_{16}$]; "/feharabic", [0641$_{16}$]; "/feharmenian", [0586$_{16}$]; "/fehfinalarabic", [FED2$_{16}$]; "/fehinitialarabic", [FED3$_{16}$]; "/fehmedialarabic", [FED4$_{16}$]; "/feicoptic", [03E5$_{16}$]; "/female", [2640$_{16}$]; "/ff", [FB00$_{16}$]; "/ffi", [FB03$_{16}$]; "/ffl", [FB04$_{16}$]; "/fi", [FB01$_{16}$]; "/fifteencircle", [246E$_{16}$]; "/fifteenparen", [2482$_{16}$]; "/fifteenperiod", [2496$_{16}$]; "/figuredash", [2012$_{16}$]; "/filledbox", [25A0$_{16}$]; "/filledrect", [25AC$_{16}$]; "/finalkaf", [05DA$_{16}$]; "/finalkafdagesh", [FB3A$_{16}$]; "/finalkafdageshhebrew", [FB3A$_{16}$]; "/finalkafhebrew", [05DA$_{16}$]; "/finalkafqamats", [05DA$_{16}$; 05B8$_{16}$]; "/finalkafqamatshebrew", [05DA$_{16}$; 05B8$_{16}$]; "/finalkafsheva", [05DA$_{16}$; 05B0$_{16}$]; "/finalkafshevahebrew", [05DA$_{16}$; 05B0$_{16}$]; "/finalmem", [05DD$_{16}$]; "/finalmemhebrew", [05DD$_{16}$]; "/finalnun", [05DF$_{16}$]; "/finalnunhebrew", [05DF$_{16}$]; "/finalpe", [05E3$_{16}$]; "/finalpehebrew", [05E3$_{16}$]; "/finaltsadi", [05E5$_{16}$]; "/finaltsadihebrew", [05E5$_{16}$]; "/firsttonechinese", [02C9$_{16}$]; "/fisheye", [25C9$_{16}$]; "/fitacyrillic", [0473$_{16}$]; "/five", [0035$_{16}$]; "/fivearabic", [0665$_{16}$]; "/fivebengali", [09EB$_{16}$]; "/fivecircle", [2464$_{16}$]; "/fivecircleinversesansserif", [278E$_{16}$]; "/fivedeva", [096B$_{16}$]; "/fiveeighths", [215D$_{16}$]; "/fivegujarati", [0AEB$_{16}$]; "/fivegurmukhi", [0A6B$_{16}$]; "/fivehackarabic", [0665$_{16}$]; "/fivehangzhou", [3025$_{16}$]; "/fiveideographicparen", [3224$_{16}$]; "/fiveinferior", [2085$_{16}$]; "/fivemonospace", [FF15$_{16}$]; "/fiveoldstyle", [F735$_{16}$]; "/fiveparen", [2478$_{16}$]; "/fiveperiod", [248C$_{16}$]; "/fivepersian", [06F5$_{16}$]; "/fiveroman", [2174$_{16}$]; "/fivesuperior", [2075$_{16}$]; "/fivethai", [0E55$_{16}$]; "/fl", [FB02$_{16}$]; "/florin", [0192$_{16}$]; "/fmonospace", [FF46$_{16}$]; "/fmsquare", [3399$_{16}$]; "/fofanthai", [0E1F$_{16}$]; "/fofathai", [0E1D$_{16}$]; "/fongmanthai", [0E4F$_{16}$]; "/forall", [2200$_{16}$]; "/four", [0034$_{16}$]; "/fourarabic", [0664$_{16}$]; "/fourbengali", [09EA$_{16}$]; "/fourcircle", [2463$_{16}$]; "/fourcircleinversesansserif", [278D$_{16}$]; "/fourdeva", [096A$_{16}$]; "/fourgujarati", [0AEA$_{16}$]; "/fourgurmukhi", [0A6A$_{16}$]; "/fourhackarabic", [0664$_{16}$]; "/fourhangzhou", [3024$_{16}$]; "/fourideographicparen", [3223$_{16}$]; "/fourinferior", [2084$_{16}$]; "/fourmonospace", [FF14$_{16}$]; "/fournumeratorbengali", [09F7$_{16}$]; "/fouroldstyle", [F734$_{16}$]; "/fourparen", [2477$_{16}$]; "/fourperiod", [248B$_{16}$];

"/fourpersian", [06F4$_{16}$]; "/fourroman", [2173$_{16}$]; "/foursuperior", [2074$_{16}$];
"/fourteencircle", [246D$_{16}$]; "/fourteenparen", [2481$_{16}$]; "/fourteenperiod",
[2495$_{16}$]; "/fourthai", [0E54$_{16}$]; "/fourthtonechinese", [02CB$_{16}$]; "/fparen",
[24A1$_{16}$]; "/fraction", [2044$_{16}$]; "/franc", [20A3$_{16}$]; "/g", [0067$_{16}$];
"/gabengali", [0997$_{16}$]; "/gacute", [01F5$_{16}$]; "/gadeva", [0917$_{16}$]; "/gafarabic",
[06AF$_{16}$]; "/gaffinalarabic", [FB93$_{16}$]; "/gafinitialarabic", [FB94$_{16}$];
"/gafmedialarabic", [FB95$_{16}$]; "/gagujarati", [0A97$_{16}$]; "/gagurmukhi", [0A17$_{16}$];
"/gahiragana", [304C$_{16}$]; "/gakatakana", [30AC$_{16}$]; "/gamma", [03B3$_{16}$];
"/gammalatinsmall", [0263$_{16}$]; "/gammasuperior", [02E0$_{16}$]; "/gangiacoptic",
[03EB$_{16}$]; "/gbopomofo", [310D$_{16}$]; "/gbreve", [011F$_{16}$]; "/gcaron", [01E7$_{16}$];
"/gcedilla", [0123$_{16}$]; "/gcircle", [24D6$_{16}$]; "/gcircumflex", [011D$_{16}$];
"/gcommaaccent", [0123$_{16}$]; "/gdot", [0121$_{16}$]; "/gdotaccent", [0121$_{16}$];
"/gecyrillic", [0433$_{16}$]; "/gehiragana", [3052$_{16}$]; "/gekatakana", [30B2$_{16}$];
"/geometricallyequal", [2251$_{16}$]; "/gereshaccenthebrew", [059C$_{16}$];
"/gereshhebrew", [05F3$_{16}$]; "/gereshmuqdamhebrew", [059D$_{16}$]; "/germandbls",
[00DF$_{16}$]; "/gershayimaccenthebrew", [059E$_{16}$]; "/gershayimhebrew", [05F4$_{16}$];
"/getamark", [3013$_{16}$]; "/ghabengali", [0998$_{16}$]; "/ghadarmenian", [0572$_{16}$];
"/ghadeva", [0918$_{16}$]; "/ghagujarati", [0A98$_{16}$]; "/ghagurmukhi", [0A18$_{16}$];
"/ghainarabic", [063A$_{16}$]; "/ghainfinalarabic", [FECE$_{16}$]; "/ghaininitialarabic",
[FECF$_{16}$]; "/ghainmedialarabic", [FED0$_{16}$]; "/ghemiddlehookcyrillic", [0495$_{16}$];
"/ghestrokecyrillic", [0493$_{16}$]; "/gheupturncyrillic", [0491$_{16}$]; "/ghhadeva",
[095A$_{16}$]; "/ghhagurmukhi", [0A5A$_{16}$]; "/ghook", [0260$_{16}$]; "/ghzsquare", [3393$_{16}$];
"/gihiragana", [304E$_{16}$]; "/gikatakana", [30AE$_{16}$]; "/gimarmenian", [0563$_{16}$];
"/gimel", [05D2$_{16}$]; "/gimeldagesh", [FB32$_{16}$]; "/gimeldageshhebrew", [FB32$_{16}$];
"/gimelhebrew", [05D2$_{16}$]; "/gjecyrillic", [0453$_{16}$]; "/glottalinvertedstroke",
[01BE$_{16}$]; "/glottalstop", [0294$_{16}$]; "/glottalstopinverted", [0296$_{16}$];
"/glottalstopmod", [02C0$_{16}$]; "/glottalstopreversed", [0295$_{16}$];
"/glottalstopreversedmod", [02C1$_{16}$]; "/glottalstopreversedsuperior", [02E4$_{16}$];
"/glottalstopstroke", [02A1$_{16}$]; "/glottalstopstrokereversed", [02A2$_{16}$];
"/gmacron", [1E21$_{16}$]; "/gmonospace", [FF47$_{16}$]; "/gohiragana", [3054$_{16}$];
"/gokatakana", [30B4$_{16}$]; "/gparen", [24A2$_{16}$]; "/gpasquare", [33AC$_{16}$];
"/gradient", [2207$_{16}$]; "/grave", [0060$_{16}$]; "/gravebelowcmb", [0316$_{16}$];
"/gravecmb", [0300$_{16}$]; "/gravecomb", [0300$_{16}$]; "/gravedeva", [0953$_{16}$];
"/gravelowmod", [02CE$_{16}$]; "/gravemonospace", [FF40$_{16}$]; "/gravetonecmb",
[0340$_{16}$]; "/greater", [003E$_{16}$]; "/greaterequal", [2265$_{16}$];
"/greaterequalorless", [22DB$_{16}$]; "/greatermonospace", [FF1E$_{16}$];
"/greaterorequivalent", [2273$_{16}$]; "/greaterorless", [2277$_{16}$];
"/greateroverequal", [2267$_{16}$]; "/greatersmall", [FE65$_{16}$]; "/gscript", [0261$_{16}$];
"/gstroke", [01E5$_{16}$]; "/guhiragana", [3050$_{16}$]; "/guillemotleft", [00AB$_{16}$];
"/guillemotright", [00BB$_{16}$]; "/guilsinglleft", [2039$_{16}$]; "/guilsinglright",
[203A$_{16}$]; "/gukatakana", [30B0$_{16}$]; "/guramusquare", [3318$_{16}$]; "/gysquare",
[33C9$_{16}$]; "/h", [0068$_{16}$]; "/haabkhasiancyrillic", [04A9$_{16}$]; "/haaltonearabic",
[06C1$_{16}$]; "/habengali", [09B9$_{16}$]; "/hadescendercyrillic", [04B3$_{16}$]; "/hadeva",
[0939$_{16}$]; "/hagujarati", [0AB9$_{16}$]; "/hagurmukhi", [0A39$_{16}$]; "/haharabic",
[062D$_{16}$]; "/hahfinalarabic", [FEA2$_{16}$]; "/hahinitialarabic", [FEA3$_{16}$];
"/hahiragana", [306F$_{16}$]; "/hahmedialarabic", [FEA4$_{16}$]; "/haitusquare", [332A$_{16}$];
"/hakatakana", [30CF$_{16}$]; "/hakatakanahalfwidth", [FF8A$_{16}$]; "/halantgurmukhi",
[0A4D$_{16}$]; "/hamzaarabic", [0621$_{16}$]; "/hamzadammaarabic", [0621$_{16}$; 064F$_{16}$];
"/hamzadammatanarabic", [0621$_{16}$; 064C$_{16}$]; "/hamzafathaarabic", [0621$_{16}$; 064E$_{16}$];
"/hamzafathatanarabic", [0621$_{16}$; 064B$_{16}$]; "/hamzalowarabic", [0621$_{16}$];

"/hamzalowkasraarabic", [0621$_{16}$; 0650$_{16}$]; "/hamzalowkasratanarabic", [0621$_{16}$; 064D$_{16}$]; "/hamzasukunarabic", [0621$_{16}$; 0652$_{16}$]; "/hangulfiller", [3164$_{16}$]; "/hardsigncyrillic", [044A$_{16}$]; "/harpoonleftbarbup", [21BC$_{16}$]; "/harpoonrightbarbup", [21C0$_{16}$]; "/hasquare", [33CA$_{16}$]; "/hatafpatah", [05B2$_{16}$]; "/hatafpatah16", [05B2$_{16}$]; "/hatafpatah23", [05B2$_{16}$]; "/hatafpatah2f", [05B2$_{16}$]; "/hatafpatahhebrew", [05B2$_{16}$]; "/hatafpatahnarrowhebrew", [05B2$_{16}$]; "/hatafpatahquarterhebrew", [05B2$_{16}$]; "/hatafpatahwidehebrew", [05B2$_{16}$]; "/hatafqamats", [05B3$_{16}$]; "/hatafqamats1b", [05B3$_{16}$]; "/hatafqamats28", [05B3$_{16}$]; "/hatafqamats34", [05B3$_{16}$]; "/hatafqamatshebrew", [05B3$_{16}$]; "/hatafqamatsnarrowhebrew", [05B3$_{16}$]; "/hatafqamatsquarterhebrew", [05B3$_{16}$]; "/hatafqamatswidehebrew", [05B3$_{16}$]; "/hatafsegol", [05B1$_{16}$]; "/hatafsegol17", [05B1$_{16}$]; "/hatafsegol24", [05B1$_{16}$]; "/hatafsegol30", [05B1$_{16}$]; "/hatafsegolhebrew", [05B1$_{16}$]; "/hatafsegolnarrowhebrew", [05B1$_{16}$]; "/hatafsegolquarterhebrew", [05B1$_{16}$]; "/hatafsegolwidehebrew", [05B1$_{16}$]; "/hbar", [0127$_{16}$]; "/hbopomofo", [310F$_{16}$]; "/hbrevebelow", [1E2B$_{16}$]; "/hcedilla", [1E29$_{16}$]; "/hcircle", [24D7$_{16}$]; "/hcircumflex", [0125$_{16}$]; "/hdieresis", [1E27$_{16}$]; "/hdotaccent", [1E23$_{16}$]; "/hdotbelow", [1E25$_{16}$]; "/he", [05D4$_{16}$]; "/heart", [2665$_{16}$]; "/heartsuitblack", [2665$_{16}$]; "/heartsuitwhite", [2661$_{16}$]; "/hedagesh", [FB34$_{16}$]; "/hedageshhebrew", [FB34$_{16}$]; "/hehaltonearabic", [06C1$_{16}$]; "/heharabic", [0647$_{16}$]; "/hehebrew", [05D4$_{16}$]; "/hehfinalaltonearabic", [FBA7$_{16}$]; "/hehfinalalttwoarabic", [FEEA$_{16}$]; "/hehfinalarabic", [FEEA$_{16}$]; "/hehhamzaabovefinalarabic", [FBA5$_{16}$]; "/hehhamzaaboveisolatedarabic", [FBA4$_{16}$]; "/hehinitialaltonearabic", [FBA8$_{16}$]; "/hehinitialarabic", [FEEB$_{16}$]; "/hehiragana", [3078$_{16}$]; "/hehmedialaltonearabic", [FBA9$_{16}$]; "/hehmedialarabic", [FEEC$_{16}$]; "/heiseierasquare", [337B$_{16}$]; "/hekatakana", [30D8$_{16}$]; "/hekatakanahalfwidth", [FF8D$_{16}$]; "/hekutaarusquare", [3336$_{16}$]; "/henghook", [0267$_{16}$]; "/herutusquare", [3339$_{16}$]; "/het", [05D7$_{16}$]; "/hethebrew", [05D7$_{16}$]; "/hhook", [0266$_{16}$]; "/hhooksuperior", [02B1$_{16}$]; "/hieuhacirclekorean", [327B$_{16}$]; "/hieuhaparenkorean", [321B$_{16}$]; "/hieuhcirclekorean", [326D$_{16}$]; "/hieuhkorean", [314E$_{16}$]; "/hieuhparenkorean", [320D$_{16}$]; "/hihiragana", [3072$_{16}$]; "/hikatakana", [30D2$_{16}$]; "/hikatakanahalfwidth", [FF8B$_{16}$]; "/hiriq", [05B4$_{16}$]; "/hiriq14", [05B4$_{16}$]; "/hiriq21", [05B4$_{16}$]; "/hiriq2d", [05B4$_{16}$]; "/hiriqhebrew", [05B4$_{16}$]; "/hiriqnarrowhebrew", [05B4$_{16}$]; "/hiriqquarterhebrew", [05B4$_{16}$]; "/hiriqwidehebrew", [05B4$_{16}$]; "/hlinebelow", [1E96$_{16}$]; "/hmonospace", [FF48$_{16}$]; "/hoarmenian", [0570$_{16}$]; "/hohipthai", [0E2B$_{16}$]; "/hohiragana", [307B$_{16}$]; "/hokatakana", [30DB$_{16}$]; "/hokatakanahalfwidth", [FF8E$_{16}$]; "/holam", [05B9$_{16}$]; "/holam19", [05B9$_{16}$]; "/holam26", [05B9$_{16}$]; "/holam32", [05B9$_{16}$]; "/holamhebrew", [05B9$_{16}$]; "/holamnarrowhebrew", [05B9$_{16}$]; "/holamquarterhebrew", [05B9$_{16}$]; "/holamwidehebrew", [05B9$_{16}$]; "/honokhukthai", [0E2E$_{16}$]; "/hookabovecomb", [0309$_{16}$]; "/hookcmb", [0309$_{16}$]; "/hookpalatalizedbelowcmb", [0321$_{16}$]; "/hookretroflexbelowcmb", [0322$_{16}$]; "/hoonsquare", [3342$_{16}$]; "/horicoptic", [03E9$_{16}$]; "/horizontalbar", [2015$_{16}$]; "/horncmb", [031B$_{16}$]; "/hotsprings", [2668$_{16}$]; "/house", [2302$_{16}$]; "/hparen", [24A3$_{16}$]; "/hsuperior", [02B0$_{16}$]; "/hturned", [0265$_{16}$]; "/huhiragana", [3075$_{16}$]; "/huiitosquare", [3333$_{16}$]; "/hukatakana", [30D5$_{16}$]; "/hukatakanahalfwidth", [FF8C$_{16}$]; "/hungarumlaut", [02DD$_{16}$]; "/hungarumlautcmb", [030B$_{16}$]; "/hv", [0195$_{16}$]; "/hyphen", [002D$_{16}$]; "/hypheninferior", [F6E5$_{16}$]; "/hyphenmonospace", [FF0D$_{16}$]; "/hyphensmall", [FE63$_{16}$]; "/hyphensuperior", [F6E6$_{16}$]; "/hyphentwo", [2010$_{16}$]; "/i", [0069$_{16}$]; "/iacute", [00ED$_{16}$]; "/iacyrillic", [044F$_{16}$]; "/ibengali", [0987$_{16}$]; "/ibopomofo", [3127$_{16}$]; "/ibreve", [012D$_{16}$]; "/icaron", [01D0$_{16}$]; "/icircle", [24D8$_{16}$]; "/icircumflex", [00EE$_{16}$]; "/icyrillic", [0456$_{16}$]; "/idblgrave", [0209$_{16}$]; "/ideographearthcircle", [328F$_{16}$]; "/ideographfirecircle", [328B$_{16}$]; "/ideographicallianceparen", [323F$_{16}$]; "/ideographiccallparen", [323A$_{16}$]; "/ideographiccentrecircle", [32A5$_{16}$]; "/ideographicclose", [3006$_{16}$]; "/ideographiccomma", [3001$_{16}$]; "/ideographiccommaleft", [FF64$_{16}$]; "/ideographiccongratulationparen", [3237$_{16}$]; "/ideographiccorrectcircle", [32A3$_{16}$]; "/ideographicearthparen", [322F$_{16}$]; "/ideographicenterpriseparen", [323D$_{16}$]; "/ideographicexcellentcircle", [329D$_{16}$]; "/ideographicfestivalparen", [3240$_{16}$]; "/ideographicfinancialcircle", [3296$_{16}$]; "/ideographicfinancialparen", [3236$_{16}$]; "/ideographicfireparen", [322B$_{16}$]; "/ideographichaveparen", [3232$_{16}$]; "/ideographichighcircle", [32A4$_{16}$]; "/ideographiciterationmark", [3005$_{16}$]; "/ideographiclaborcircle", [3298$_{16}$]; "/ideographiclaborparen", [3238$_{16}$]; "/ideographicleftcircle", [32A7$_{16}$]; "/ideographiclowcircle", [32A6$_{16}$]; "/ideographicmedicinecircle", [32A9$_{16}$]; "/ideographicmetalparen", [322E$_{16}$]; "/ideographicmoonparen", [322A$_{16}$]; "/ideographicnameparen", [3234$_{16}$]; "/ideographicperiod", [3002$_{16}$]; "/ideographicprintcircle", [329E$_{16}$]; "/ideographicreachparen", [3243$_{16}$]; "/ideographicrepresentparen", [3239$_{16}$]; "/ideographicresourceparen", [323E$_{16}$]; "/ideographicrightcircle", [32A8$_{16}$]; "/ideographicsecretcircle", [3299$_{16}$]; "/ideographicselfparen", [3242$_{16}$]; "/ideographicsocietyparen", [3233$_{16}$]; "/ideographicspace", [3000$_{16}$]; "/ideographicspecialparen", [3235$_{16}$]; "/ideographicstockparen", [3231$_{16}$]; "/ideographicstudyparen", [323B$_{16}$]; "/ideographicsunparen", [3230$_{16}$]; "/ideographicsuperviseparen", [323C$_{16}$]; "/ideographicwaterparen", [322C$_{16}$]; "/ideographicwoodparen", [322D$_{16}$]; "/ideographiczero", [3007$_{16}$]; "/ideographmetalcircle", [328E$_{16}$]; "/ideographmooncircle", [328A$_{16}$]; "/ideographnamecircle", [3294$_{16}$]; "/ideographsuncircle", [3290$_{16}$]; "/ideographwatercircle", [328C$_{16}$]; "/ideographwoodcircle", [328D$_{16}$]; "/ideva", [0907$_{16}$]; "/idieresis", [00EF$_{16}$]; "/idieresisacute", [1E2F$_{16}$]; "/idieresiscyrillic", [04E5$_{16}$]; "/idotbelow", [1ECB$_{16}$]; "/iebrevecyrillic", [04D7$_{16}$]; "/iecyrillic", [0435$_{16}$]; "/ieungacirclekorean", [3275$_{16}$]; "/ieungaparenkorean", [3215$_{16}$]; "/ieungcirclekorean", [3267$_{16}$]; "/ieungkorean", [3147$_{16}$]; "/ieungparenkorean", [3207$_{16}$]; "/igrave", [00EC$_{16}$]; "/igujarati", [0A87$_{16}$]; "/igurmukhi", [0A07$_{16}$]; "/ihiragana", [3044$_{16}$]; "/ihookabove", [1EC9$_{16}$]; "/iibengali", [0988$_{16}$]; "/iicyrillic", [0438$_{16}$]; "/iideva", [0908$_{16}$]; "/iigujarati", [0A88$_{16}$]; "/iigurmukhi", [0A08$_{16}$]; "/iimatragurmukhi", [0A40$_{16}$]; "/iinvertedbreve", [020B$_{16}$]; "/iishortcyrillic", [0439$_{16}$]; "/iivowelsignbengali", [09C0$_{16}$]; "/iivowelsigndeva", [0940$_{16}$]; "/iivowelsigngujarati", [0AC0$_{16}$]; "/ij", [0133$_{16}$]; "/ikatakana", [30A4$_{16}$]; "/ikatakanahalfwidth", [FF72$_{16}$]; "/ikorean", [3163$_{16}$]; "/ilde", [02DC$_{16}$]; "/iluyhebrew", [05AC$_{16}$]; "/imacron", [012B$_{16}$]; "/imacroncyrillic", [04E3$_{16}$]; "/imageorapproximatelyequal", [2253$_{16}$]; "/imatragurmukhi", [0A3F$_{16}$]; "/imonospace", [FF49$_{16}$]; "/increment", [2206$_{16}$]; "/infinity", [221E$_{16}$]; "/iniarmenian", [056B$_{16}$]; "/integral", [222B$_{16}$]; "/integralbottom", [2321$_{16}$]; "/integralbt", [2321$_{16}$]; "/integralex", [F8F5$_{16}$]; "/integraltop", [2320$_{16}$]; "/integraltp", [2320$_{16}$]; "/intersection", [2229$_{16}$]; "/intisquare", [3305$_{16}$]; "/invbullet", [25D8$_{16}$]; "/invcircle", [25D9$_{16}$]; "/invsmileface", [263B$_{16}$]; "/iocyrillic", [0451$_{16}$]; "/iogonek", [012F$_{16}$]; "/iota", [03B9$_{16}$]; "/iotadieresis", [03CA$_{16}$]; "/iotadieresistonos", [0390$_{16}$]; "/iotalatin", [0269$_{16}$]; "/iotatonos", [03AF$_{16}$]; "/iparen", [24A4$_{16}$];

"/irigurmukhi", [0A72$_{16}$]; "/ismallhiragana", [3043$_{16}$]; "/ismallkatakana", [30A3$_{16}$]; "/ismallkatakanahalfwidth", [FF68$_{16}$]; "/issharbengali", [09FA$_{16}$]; "/istroke", [0268$_{16}$]; "/isuperior", [F6ED$_{16}$]; "/iterationhiragana", [309D$_{16}$]; "/iterationkatakana", [30FD$_{16}$]; "/itilde", [0129$_{16}$]; "/itildebelow", [1E2D$_{16}$]; "/iubopomofo", [3129$_{16}$]; "/iucyrillic", [044E$_{16}$]; "/ivowelsignbengali", [09BF$_{16}$]; "/ivowelsigndeva", [093F$_{16}$]; "/ivowelsigngujarati", [0ABF$_{16}$]; "/izhitsacyrillic", [0475$_{16}$]; "/izhitsadblgravecyrillic", [0477$_{16}$]; "/j", [006A$_{16}$]; "/jaarmenian", [0571$_{16}$]; "/jabengali", [099C$_{16}$]; "/jadeva", [091C$_{16}$]; "/jagujarati", [0A9C$_{16}$]; "/jagurmukhi", [0A1C$_{16}$]; "/jbopomofo", [3110$_{16}$]; "/jcaron", [01F0$_{16}$]; "/jcircle", [24D9$_{16}$]; "/jcircumflex", [0135$_{16}$]; "/jcrossedtail", [029D$_{16}$]; "/jdotlessstroke", [025F$_{16}$]; "/jecyrillic", [0458$_{16}$]; "/jeemarabic", [062C$_{16}$]; "/jeemfinalarabic", [FE9E$_{16}$]; "/jeeminitialarabic", [FE9F$_{16}$]; "/jeemmedialarabic", [FEA0$_{16}$]; "/jeharabic", [0698$_{16}$]; "/jehfinalarabic", [FB8B$_{16}$]; "/jhabengali", [099D$_{16}$]; "/jhadeva", [091D$_{16}$]; "/jhagujarati", [0A9D$_{16}$]; "/jhagurmukhi", [0A1D$_{16}$]; "/jheharmenian", [057B$_{16}$]; "/jis", [3004$_{16}$]; "/jmonospace", [FF4A$_{16}$]; "/jparen", [24A5$_{16}$]; "/jsuperior", [02B2$_{16}$]; "/k", [006B$_{16}$]; "/kabashkircyrillic", [04A1$_{16}$]; "/kabengali", [0995$_{16}$]; "/kacute", [1E31$_{16}$]; "/kacyrillic", [043A$_{16}$]; "/kadescendercyrillic", [049B$_{16}$]; "/kadeva", [0915$_{16}$]; "/kaf", [05DB$_{16}$]; "/kafarabic", [0643$_{16}$]; "/kafdagesh", [FB3B$_{16}$]; "/kafdageshhebrew", [FB3B$_{16}$]; "/kaffinalarabic", [FEDA$_{16}$]; "/kafhebrew", [05DB$_{16}$]; "/kafinitialarabic", [FEDB$_{16}$]; "/kafmedialarabic", [FEDC$_{16}$]; "/kafrafehebrew", [FB4D$_{16}$]; "/kagujarati", [0A95$_{16}$]; "/kagurmukhi", [0A15$_{16}$]; "/kahiragana", [304B$_{16}$]; "/kahookcyrillic", [04C4$_{16}$]; "/kakatakana", [30AB$_{16}$]; "/kakatakanahalfwidth", [FF76$_{16}$]; "/kappa", [03BA$_{16}$]; "/kappasymbolgreek", [03F0$_{16}$]; "/kapyeounmieumkorean", [3171$_{16}$]; "/kapyeounphieuphkorean", [3184$_{16}$]; "/kapyeounpieupkorean", [3178$_{16}$]; "/kapyeounssangpieupkorean", [3179$_{16}$]; "/karoriisquare", [330D$_{16}$]; "/kashidaautoarabic", [0640$_{16}$]; "/kashidaautonosidebearingarabic", [0640$_{16}$]; "/kasmallkatakana", [30F5$_{16}$]; "/kasquare", [3384$_{16}$]; "/kasraarabic", [0650$_{16}$]; "/kasratanarabic", [064D$_{16}$]; "/kastrokecyrillic", [049F$_{16}$]; "/katahiraprolongmarkhalfwidth", [FF70$_{16}$]; "/kaverticalstrokecyrillic", [049D$_{16}$]; "/kbopomofo", [310E$_{16}$]; "/kcalsquare", [3389$_{16}$]; "/kcaron", [01E9$_{16}$]; "/kcedilla", [0137$_{16}$]; "/kcircle", [24DA$_{16}$]; "/kcommaaccent", [0137$_{16}$]; "/kdotbelow", [1E33$_{16}$]; "/keharmenian", [0584$_{16}$]; "/kehiragana", [3051$_{16}$]; "/kekatakana", [30B1$_{16}$]; "/kekatakanahalfwidth", [FF79$_{16}$]; "/kenarmenian", [056F$_{16}$]; "/kesmallkatakana", [30F6$_{16}$]; "/kgreenlandic", [0138$_{16}$]; "/khabengali", [0996$_{16}$]; "/khacyrillic", [0445$_{16}$]; "/khadeva", [0916$_{16}$]; "/khagujarati", [0A96$_{16}$]; "/khagurmukhi", [0A16$_{16}$]; "/khaharabic", [062E$_{16}$]; "/khahfinalarabic", [FEA6$_{16}$]; "/khahinitialarabic", [FEA7$_{16}$]; "/khahmedialarabic", [FEA8$_{16}$]; "/kheicoptic", [03E7$_{16}$]; "/khhadeva", [0959$_{16}$]; "/khhagurmukhi", [0A59$_{16}$]; "/khieukhacirclekorean", [3278$_{16}$]; "/khieukhaparenkorean", [3218$_{16}$]; "/khieukhcirclekorean", [326A$_{16}$]; "/khieukhkorean", [314B$_{16}$]; "/khieukhparenkorean", [320A$_{16}$]; "/khokhaithai", [0E02$_{16}$]; "/khokhonthai", [0E05$_{16}$]; "/khokhuatthai", [0E03$_{16}$]; "/khokhwaithai", [0E04$_{16}$]; "/khomutthai", [0E5B$_{16}$]; "/khook", [0199$_{16}$]; "/khorakhangthai", [0E06$_{16}$]; "/khzsquare", [3391$_{16}$]; "/kihiragana", [304D$_{16}$]; "/kikatakana", [30AD$_{16}$]; "/kikatakanahalfwidth", [FF77$_{16}$]; "/kiroguramusquare", [3315$_{16}$]; "/kiromeetorusquare", [3316$_{16}$]; "/kirosquare", [3314$_{16}$]; "/kiyeokacirclekorean", [326E$_{16}$]; "/kiyeokaparenkorean", [320E$_{16}$]; "/kiyeokcirclekorean", [3260$_{16}$]; "/kiyeokkorean", [3131$_{16}$]; "/kiyeokparenkorean", [3200$_{16}$]; "/kiyeoksioskorean",

[3133$_{16}$]; "/kjecyrillic", [045C$_{16}$]; "/klinebelow", [1E35$_{16}$]; "/klsquare", [3398$_{16}$]; "/kmcubedsquare", [33A6$_{16}$]; "/kmonospace", [FF4B$_{16}$]; "/kmsquaredsquare", [33A2$_{16}$]; "/kohiragana", [3053$_{16}$]; "/kohmsquare", [33C0$_{16}$]; "/kokaithai", [0E01$_{16}$]; "/kokatakana", [30B3$_{16}$]; "/kokatakanahalfwidth", [FF7A$_{16}$]; "/kooposquare", [331E$_{16}$]; "/koppacyrillic", [0481$_{16}$]; "/koreanstandardsymbol", [327F$_{16}$]; "/koroniscmb", [0343$_{16}$]; "/kparen", [24A6$_{16}$]; "/kpasquare", [33AA$_{16}$]; "/ksicyrillic", [046F$_{16}$]; "/ktsquare", [33CF$_{16}$]; "/kturned", [029E$_{16}$]; "/kuhiragana", [304F$_{16}$]; "/kukatakana", [30AF$_{16}$]; "/kukatakanahalfwidth", [FF78$_{16}$]; "/kvsquare", [33B8$_{16}$]; "/kwsquare", [33BE$_{16}$]; "/l", [006C$_{16}$]; "/labengali", [09B2$_{16}$]; "/lacute", [013A$_{16}$]; "/ladeva", [0932$_{16}$]; "/lagujarati", [0AB2$_{16}$]; "/lagurmukhi", [0A32$_{16}$]; "/lakkhangyaothai", [0E45$_{16}$]; "/lamaleffinalarabic", [FEFC$_{16}$]; "/lamalefhamzaabovefinalarabic", [FEF8$_{16}$]; "/lamalefhamzaaboveisolatedarabic", [FEF7$_{16}$]; "/lamalefhamzabelowfinalarabic", [FEFA$_{16}$]; "/lamalefhamzabelowisolatedarabic", [FEF9$_{16}$]; "/lamalefisolatedarabic", [FEFB$_{16}$]; "/lamalefmaddaabovefinalarabic", [FEF6$_{16}$]; "/lamalefmaddaaboveisolatedarabic", [FEF5$_{16}$]; "/lamarabic", [0644$_{16}$]; "/lambda", [03BB$_{16}$]; "/lambdastroke", [019B$_{16}$]; "/lamed", [05DC$_{16}$]; "/lameddagesh", [FB3C$_{16}$]; "/lameddageshhebrew", [FB3C$_{16}$]; "/lamedhebrew", [05DC$_{16}$]; "/lamedholam", [05DC$_{16}$; 05B9$_{16}$]; "/lamedholamdagesh", [05DC$_{16}$; 05B9$_{16}$; 05BC$_{16}$];
"/lamedholamdageshhebrew", [05DC$_{16}$; 05B9$_{16}$; 05BC$_{16}$]; "/lamedholamhebrew", [05DC$_{16}$;
05B9$_{16}$]; "/lamfinalarabic", [FEDE$_{16}$]; "/lamhahinitialarabic", [FCCA$_{16}$]; "/laminitialarabic", [FEDF$_{16}$]; "/lamjeeminitialarabic", [FCC9$_{16}$]; "/lamkhahinitialarabic", [FCCB$_{16}$]; "/lamlamhehisolatedarabic", [FDF2$_{16}$]; "/lammedialarabic", [FEE0$_{16}$]; "/lammeemhahinitialarabic", [FD88$_{16}$]; "/lammeeminitialarabic", [FCCC$_{16}$]; "/lammeemjeeminitialarabic", [FEDF$_{16}$; FEE4$_{16}$; FEA0$_{16}$]; "/lammeemkhahinitialarabic", [FEDF$_{16}$; FEE4$_{16}$; FEA8$_{16}$]; "/largecircle", [25EF$_{16}$]; "/lbar", [019A$_{16}$]; "/lbelt", [026C$_{16}$]; "/lbopomofo", [310C$_{16}$]; "/lcaron", [013E$_{16}$]; "/lcedilla", [013C$_{16}$]; "/lcircle", [24DB$_{16}$]; "/lcircumflexbelow", [1E3D$_{16}$]; "/lcommaaccent", [013C$_{16}$]; "/ldot", [0140$_{16}$]; "/ldotaccent", [0140$_{16}$]; "/ldotbelow", [1E37$_{16}$]; "/ldotbelowmacron", [1E39$_{16}$]; "/leftangleabovecmb", [031A$_{16}$]; "/lefttackbelowcmb", [0318$_{16}$]; "/less", [003C$_{16}$]; "/lessequal", [2264$_{16}$]; "/lessequalorgreater", [22DA$_{16}$]; "/lessmonospace", [FF1C$_{16}$]; "/lessorequivalent", [2272$_{16}$]; "/lessorgreater", [2276$_{16}$]; "/lessoverequal", [2266$_{16}$]; "/lesssmall", [FE64$_{16}$]; "/lezh", [026E$_{16}$]; "/lfblock", [258C$_{16}$]; "/lhookretroflex", [026D$_{16}$]; "/lira", [20A4$_{16}$]; "/liwnarmenian", [056C$_{16}$]; "/lj", [01C9$_{16}$]; "/ljecyrillic", [0459$_{16}$]; "/ll", [F6C0$_{16}$]; "/lladeva", [0933$_{16}$]; "/llagujarati", [0AB3$_{16}$]; "/llinebelow", [1E3B$_{16}$]; "/llladeva", [0934$_{16}$]; "/llvocalicbengali", [09E1$_{16}$]; "/llvocalicdeva", [0961$_{16}$]; "/llvocalicvowelsignbengali", [09E3$_{16}$]; "/llvocalicvowelsigndeva", [0963$_{16}$]; "/lmiddletilde", [026B$_{16}$]; "/lmonospace", [FF4C$_{16}$]; "/lmsquare", [33D0$_{16}$]; "/lochulathai", [0E2C$_{16}$]; "/logicaland", [2227$_{16}$]; "/logicalnot", [00AC$_{16}$]; "/logicalnotreversed", [2310$_{16}$]; "/logicalor", [2228$_{16}$]; "/lolingthai", [0E25$_{16}$]; "/longs", [017F$_{16}$]; "/lowlinecenterline", [FE4E$_{16}$]; "/lowlinecmb", [0332$_{16}$]; "/lowlinedashed", [FE4D$_{16}$]; "/lozenge", [25CA$_{16}$]; "/lparen", [24A7$_{16}$]; "/lslash", [0142$_{16}$]; "/lsquare", [2113$_{16}$]; "/lsuperior", [F6EE$_{16}$]; "/ltshade", [2591$_{16}$]; "/luthai", [0E26$_{16}$]; "/lvocalicbengali", [098C$_{16}$]; "/lvocalicdeva", [090C$_{16}$]; "/lvocalicvowelsignbengali", [09E2$_{16}$]; "/lvocalicvowelsigndeva", [0962$_{16}$];

"/lxsquare", [33D3$_{16}$]; "/m", [006D$_{16}$]; "/mabengali", [09AE$_{16}$]; "/macron", [00AF$_{16}$]; "/macronbelowcmb", [0331$_{16}$]; "/macroncmb", [0304$_{16}$]; "/macronlowmod", [02CD$_{16}$]; "/macronmonospace", [FFE3$_{16}$]; "/macute", [1E3F$_{16}$]; "/madeva", [092E$_{16}$]; "/magujarati", [0AAE$_{16}$]; "/magurmukhi", [0A2E$_{16}$]; "/mahapakhhebrew", [05A4$_{16}$]; "/mahapakhlefthebrew", [05A4$_{16}$]; "/mahiragana", [307E$_{16}$]; "/maichattawalowleftthai", [F895$_{16}$]; "/maichattawalowrightthai", [F894$_{16}$]; "/maichattawathai", [0E4B$_{16}$]; "/maichattawaupperleftthai", [F893$_{16}$]; "/maieklowleftthai", [F88C$_{16}$]; "/maieklowrightthai", [F88B$_{16}$]; "/maiekthai", [0E48$_{16}$]; "/maiekupperleftthai", [F88A$_{16}$]; "/maihanakatleftthai", [F884$_{16}$]; "/maihanakatthai", [0E31$_{16}$]; "/maitaikhuleftthai", [F889$_{16}$]; "/maitaikhuthai", [0E47$_{16}$]; "/maitholowleftthai", [F88F$_{16}$]; "/maitholowrightthai", [F88E$_{16}$]; "/maithothai", [0E49$_{16}$]; "/maithoupperleftthai", [F88D$_{16}$]; "/maitrilowleftthai", [F892$_{16}$]; "/maitrilowrightthai", [F891$_{16}$]; "/maitrithai", [0E4A$_{16}$]; "/maitriupperleftthai", [F890$_{16}$]; "/maiyamokthai", [0E46$_{16}$]; "/makatakana", [30DE$_{16}$]; "/makatakanahalfwidth", [FF8F$_{16}$]; "/male", [2642$_{16}$]; "/mansyonsquare", [3347$_{16}$]; "/maqafhebrew", [05BE$_{16}$]; "/mars", [2642$_{16}$]; "/masoracirclehebrew", [05AF$_{16}$]; "/masquare", [3383$_{16}$]; "/mbopomofo", [3107$_{16}$]; "/mbsquare", [33D4$_{16}$]; "/mcircle", [24DC$_{16}$]; "/mcubedsquare", [33A5$_{16}$]; "/mdotaccent", [1E41$_{16}$]; "/mdotbelow", [1E43$_{16}$]; "/meemarabic", [0645$_{16}$]; "/meemfinalarabic", [FEE2$_{16}$]; "/meeminitialarabic", [FEE3$_{16}$]; "/meemmedialarabic", [FEE4$_{16}$]; "/meemmeeminitialarabic", [FCD1$_{16}$]; "/meemmeemisolatedarabic", [FC48$_{16}$]; "/meetorusquare", [334D$_{16}$]; "/mehiragana", [3081$_{16}$]; "/meizierasquare", [337E$_{16}$]; "/mekatakana", [30E1$_{16}$]; "/mekatakanahalfwidth", [FF92$_{16}$]; "/mem", [05DE$_{16}$]; "/memdagesh", [FB3E$_{16}$]; "/memdageshhebrew", [FB3E$_{16}$]; "/memhebrew", [05DE$_{16}$]; "/menarmenian", [0574$_{16}$]; "/merkhahebrew", [05A5$_{16}$]; "/merkhakefulahebrew", [05A6$_{16}$]; "/merkhakefulalefthebrew", [05A6$_{16}$]; "/merkhalefthebrew", [05A5$_{16}$]; "/mhook", [0271$_{16}$]; "/mhzsquare", [3392$_{16}$]; "/middledotkatakanahalfwidth", [FF65$_{16}$]; "/middot", [00B7$_{16}$]; "/mieumacirclekorean", [3272$_{16}$]; "/mieumaparenkorean", [3212$_{16}$]; "/mieumcirclekorean", [3264$_{16}$]; "/mieumkorean", [3141$_{16}$]; "/mieumpansioskorean", [3170$_{16}$]; "/mieumparenkorean", [3204$_{16}$]; "/mieumpieupkorean", [316E$_{16}$]; "/mieumsioskorean", [316F$_{16}$]; "/mihiragana", [307F$_{16}$]; "/mikatakana", [30DF$_{16}$]; "/mikatakanahalfwidth", [FF90$_{16}$]; "/minus", [2212$_{16}$]; "/minusbelowcmb", [0320$_{16}$]; "/minuscircle", [2296$_{16}$]; "/minusmod", [02D7$_{16}$]; "/minusplus", [2213$_{16}$]; "/minute", [2032$_{16}$]; "/miribaarusquare", [334A$_{16}$]; "/mirisquare", [3349$_{16}$]; "/mlonglegturned", [0270$_{16}$]; "/mlsquare", [3396$_{16}$]; "/mmcubedsquare", [33A3$_{16}$]; "/mmonospace", [FF4D$_{16}$]; "/mmsquaredsquare", [339F$_{16}$]; "/mohiragana", [3082$_{16}$]; "/mohmsquare", [33C1$_{16}$]; "/mokatakana", [30E2$_{16}$]; "/mokatakanahalfwidth", [FF93$_{16}$]; "/molsquare", [33D6$_{16}$]; "/momathai", [0E21$_{16}$]; "/moverssquare", [33A7$_{16}$]; "/moverssquaredsquare", [33A8$_{16}$]; "/mparen", [24A8$_{16}$]; "/mpasquare", [33AB$_{16}$]; "/mssquare", [33B3$_{16}$]; "/msuperior", [F6EF$_{16}$]; "/mturned", [026F$_{16}$]; "/mu", [00B5$_{16}$]; "/mu1", [00B5$_{16}$]; "/muasquare", [3382$_{16}$]; "/muchgreater", [226B$_{16}$]; "/muchless", [226A$_{16}$]; "/mufsquare", [338C$_{16}$]; "/mugreek", [03BC$_{16}$]; "/mugsquare", [338D$_{16}$]; "/muhiragana", [3080$_{16}$]; "/mukatakana", [30E0$_{16}$]; "/mukatakanahalfwidth", [FF91$_{16}$]; "/mulsquare", [3395$_{16}$]; "/multiply", [00D7$_{16}$]; "/mumsquare", [339B$_{16}$]; "/munahhebrew", [05A3$_{16}$]; "/munahlefthebrew", [05A3$_{16}$]; "/musicalnote", [266A$_{16}$]; "/musicalnotedbl", [266B$_{16}$]; "/musicflatsign", [266D$_{16}$]; "/musicsharpsign", [266F$_{16}$]; "/mussquare", [33B2$_{16}$]; "/muvsquare", [33B6$_{16}$]; "/muwsquare", [33BC$_{16}$]; "/mvmegasquare", [33B9$_{16}$]; "/mvsquare", [33B7$_{16}$]; "/mwmegasquare", [33BF$_{16}$]; "/mwsquare", [33BD$_{16}$]; "/n", [006E$_{16}$]; "/nabengali", [09A8$_{16}$]; "/nabla", [2207$_{16}$]; "/nacute", [0144$_{16}$]; "/nadeva", [0928$_{16}$]; "/nagujarati", [0AA8$_{16}$]; "/nagurmukhi", [0A28$_{16}$]; "/nahiragana", [306A$_{16}$]; "/nakatakana", [30CA$_{16}$]; "/nakatakanahalfwidth", [FF85$_{16}$]; "/napostrophe", [0149$_{16}$]; "/nasquare", [3381$_{16}$]; "/nbopomofo", [310B$_{16}$]; "/nbspace", [00A0$_{16}$]; "/ncaron", [0148$_{16}$]; "/ncedilla", [0146$_{16}$]; "/ncircle", [24DD$_{16}$]; "/ncircumflexbelow", [1E4B$_{16}$]; "/ncommaaccent", [0146$_{16}$]; "/ndotaccent", [1E45$_{16}$]; "/ndotbelow", [1E47$_{16}$]; "/nehiragana", [306D$_{16}$]; "/nekatakana", [30CD$_{16}$]; "/nekatakanahalfwidth", [FF88$_{16}$]; "/newsheqelsign", [20AA$_{16}$]; "/nfsquare", [338B$_{16}$]; "/ngabengali", [0999$_{16}$]; "/ngadeva", [0919$_{16}$]; "/ngagujarati", [0A99$_{16}$]; "/ngagurmukhi", [0A19$_{16}$]; "/ngonguthai", [0E07$_{16}$]; "/nhiragana", [3093$_{16}$]; "/nhookleft", [0272$_{16}$]; "/nhookretroflex", [0273$_{16}$]; "/nieunacirclekorean", [326F$_{16}$]; "/nieunaparenkorean", [320F$_{16}$]; "/nieuncieuckorean", [3135$_{16}$]; "/nieuncirclekorean", [3261$_{16}$]; "/nieunhieuhkorean", [3136$_{16}$]; "/nieunkorean", [3134$_{16}$]; "/nieunpansioskorean", [3168$_{16}$]; "/nieunparenkorean", [3201$_{16}$]; "/nieunsioskorean", [3167$_{16}$]; "/nieuntikeutkorean", [3166$_{16}$]; "/nihiragana", [306B$_{16}$]; "/nikatakana", [30CB$_{16}$]; "/nikatakanahalfwidth", [FF86$_{16}$]; "/nikhahitleftthai", [F899$_{16}$]; "/nikhahitthai", [0E4D$_{16}$]; "/nine", [0039$_{16}$]; "/ninearabic", [0669$_{16}$]; "/ninebengali", [09EF$_{16}$]; "/ninecircle", [2468$_{16}$]; "/ninecircleinversesansserif", [2792$_{16}$]; "/ninedeva", [096F$_{16}$]; "/ninegujarati", [0AEF$_{16}$]; "/ninegurmukhi", [0A6F$_{16}$]; "/ninehackarabic", [0669$_{16}$]; "/ninehangzhou", [3029$_{16}$]; "/nineideographicparen", [3228$_{16}$]; "/nineinferior", [2089$_{16}$]; "/ninemonospace", [FF19$_{16}$]; "/nineoldstyle", [F739$_{16}$]; "/nineparen", [247C$_{16}$]; "/nineperiod", [2490$_{16}$]; "/ninepersian", [06F9$_{16}$]; "/nineroman", [2178$_{16}$]; "/ninesuperior", [2079$_{16}$]; "/nineteencircle", [2472$_{16}$]; "/nineteenparen", [2486$_{16}$]; "/nineteenperiod", [249A$_{16}$]; "/ninethai", [0E59$_{16}$]; "/nj", [01CC$_{16}$]; "/njecyrillic", [045A$_{16}$]; "/nkatakana", [30F3$_{16}$]; "/nkatakanahalfwidth", [FF9D$_{16}$]; "/nlegrightlong", [019E$_{16}$]; "/nlinebelow", [1E49$_{16}$]; "/nmonospace", [FF4E$_{16}$]; "/nmsquare", [339A$_{16}$]; "/nnabengali", [09A3$_{16}$]; "/nnadeva", [0923$_{16}$]; "/nnagujarati", [0AA3$_{16}$]; "/nnagurmukhi", [0A23$_{16}$]; "/nnnadeva", [0929$_{16}$]; "/nohiragana", [306E$_{16}$]; "/nokatakana", [30CE$_{16}$]; "/nokatakanahalfwidth", [FF89$_{16}$]; "/nonbreakingspace", [00A0$_{16}$]; "/nonenthai", [0E13$_{16}$]; "/nonuthai", [0E19$_{16}$]; "/noonarabic", [0646$_{16}$]; "/noonfinalarabic", [FEE6$_{16}$]; "/noonghunnaarabic", [06BA$_{16}$]; "/noonghunnafinalarabic", [FB9F$_{16}$]; "/noonhehinitialarabic", [FEE7$_{16}$; FEEC$_{16}$]; "/nooninitialarabic", [FEE7$_{16}$]; "/noonjeeminitialarabic", [FCD2$_{16}$]; "/noonjeemisolatedarabic", [FC4B$_{16}$]; "/noonmedialarabic", [FEE8$_{16}$]; "/noonmeeminitialarabic", [FCD5$_{16}$]; "/noonmeemisolatedarabic", [FC4E$_{16}$]; "/noonnoonfinalarabic", [FC8D$_{16}$]; "/notcontains", [220C$_{16}$]; "/notelement", [2209$_{16}$]; "/notelementof", [2209$_{16}$]; "/notequal", [2260$_{16}$]; "/notgreater", [226F$_{16}$]; "/notgreaternorequal", [2271$_{16}$]; "/notgreaternorless", [2279$_{16}$]; "/notidentical", [2262$_{16}$]; "/notless", [226E$_{16}$]; "/notlessnorequal", [2270$_{16}$]; "/notparallel", [2226$_{16}$]; "/notprecedes", [2280$_{16}$]; "/notsubset", [2284$_{16}$]; "/notsucceeds", [2281$_{16}$]; "/notsuperset", [2285$_{16}$]; "/nowarmenian", [0576$_{16}$]; "/nparen", [24A9$_{16}$]; "/nssquare", [33B1$_{16}$]; "/nsuperior", [207F$_{16}$]; "/ntilde", [00F1$_{16}$]; "/nu", [03BD$_{16}$]; "/nuhiragana", [306C$_{16}$]; "/nukatakana", [30CC$_{16}$]; "/nukatakanahalfwidth", [FF87$_{16}$]; "/nuktabengali", [09BC$_{16}$]; "/nuktadeva", [093C$_{16}$]; "/nuktagujarati", [0ABC$_{16}$]; "/nuktagurmukhi", [0A3C$_{16}$]; "/numbersign", [0023$_{16}$]; "/numbersignmonospace", [FF03$_{16}$]; "/numbersignsmall", [FE5F$_{16}$];

"/numeralsigngreek", [0374$_{16}$]; "/numeralsignlowergreek", [0375$_{16}$]; "/numero", [2116$_{16}$]; "/nun", [05E0$_{16}$]; "/nundagesh", [FB40$_{16}$]; "/nundageshhebrew", [FB40$_{16}$]; "/nunhebrew", [05E0$_{16}$]; "/nvsquare", [33B5$_{16}$]; "/nwsquare", [33BB$_{16}$]; "/nyabengali", [099E$_{16}$]; "/nyadeva", [091E$_{16}$]; "/nyagujarati", [0A9E$_{16}$]; "/nyagurmukhi", [0A1E$_{16}$]; "/o", [006F$_{16}$]; "/oacute", [00F3$_{16}$]; "/oangthai", [0E2D$_{16}$]; "/obarred", [0275$_{16}$]; "/obarredcyrillic", [04E9$_{16}$]; "/obarreddieresiscyrillic", [04EB$_{16}$]; "/obengali", [0993$_{16}$]; "/obopomofo", [311B$_{16}$]; "/obreve", [014F$_{16}$]; "/ocandradeva", [0911$_{16}$]; "/ocandragujarati", [0A91$_{16}$]; "/ocandravowelsigndeva", [0949$_{16}$]; "/ocandravowelsigngujarati", [0AC9$_{16}$]; "/ocaron", [01D2$_{16}$]; "/ocircle", [24DE$_{16}$]; "/ocircumflex", [00F4$_{16}$]; "/ocircumflexacute", [1ED1$_{16}$]; "/ocircumflexdotbelow", [1ED9$_{16}$]; "/ocircumflexgrave", [1ED3$_{16}$]; "/ocircumflexhookabove", [1ED5$_{16}$]; "/ocircumflextilde", [1ED7$_{16}$]; "/ocyrillic", [043E$_{16}$]; "/odblacute", [0151$_{16}$]; "/odblgrave", [020D$_{16}$]; "/odeva", [0913$_{16}$]; "/odieresis", [00F6$_{16}$]; "/odieresiscyrillic", [04E7$_{16}$]; "/odotbelow", [1ECD$_{16}$]; "/oe", [0153$_{16}$]; "/oekorean", [315A$_{16}$]; "/ogonek", [02DB$_{16}$]; "/ogonekcmb", [0328$_{16}$]; "/ograve", [00F2$_{16}$]; "/ogujarati", [0A93$_{16}$]; "/oharmenian", [0585$_{16}$]; "/ohiragana", [304A$_{16}$]; "/ohookabove", [1ECF$_{16}$]; "/ohorn", [01A1$_{16}$]; "/ohornacute", [1EDB$_{16}$]; "/ohorndotbelow", [1EE3$_{16}$]; "/ohorngrave", [1EDD$_{16}$]; "/ohornhookabove", [1EDF$_{16}$]; "/ohorntilde", [1EE1$_{16}$]; "/ohungarumlaut", [0151$_{16}$]; "/oi", [01A3$_{16}$]; "/oinvertedbreve", [020F$_{16}$]; "/okatakana", [30AA$_{16}$]; "/okatakanahalfwidth", [FF75$_{16}$]; "/okorean", [3157$_{16}$]; "/olehebrew", [05AB$_{16}$]; "/omacron", [014D$_{16}$]; "/omacronacute", [1E53$_{16}$]; "/omacrongrave", [1E51$_{16}$]; "/omdeva", [0950$_{16}$]; "/omega", [03C9$_{16}$]; "/omega1", [03D6$_{16}$]; "/omegacyrillic", [0461$_{16}$]; "/omegalatinclosed", [0277$_{16}$]; "/omegaroundcyrillic", [047B$_{16}$]; "/omegatitlocyrillic", [047D$_{16}$]; "/omegatonos", [03CE$_{16}$]; "/omgujarati", [0AD0$_{16}$]; "/omicron", [03BF$_{16}$]; "/omicrontonos", [03CC$_{16}$]; "/omonospace", [FF4F$_{16}$]; "/one", [0031$_{16}$]; "/onearabic", [0661$_{16}$]; "/onebengali", [09E7$_{16}$]; "/onecircle", [2460$_{16}$]; "/onecircleinversesansserif", [278A$_{16}$]; "/onedeva", [0967$_{16}$]; "/onedotenleader", [2024$_{16}$]; "/oneeighth", [215B$_{16}$]; "/onefitted", [F6DC$_{16}$]; "/onegujarati", [0AE7$_{16}$]; "/onegurmukhi", [0A67$_{16}$]; "/onehackarabic", [0661$_{16}$]; "/onehalf", [00BD$_{16}$]; "/onehangzhou", [3021$_{16}$]; "/oneideographicparen", [3220$_{16}$]; "/oneinferior", [2081$_{16}$]; "/onemonospace", [FF11$_{16}$]; "/onenumeratorbengali", [09F4$_{16}$]; "/oneoldstyle", [F731$_{16}$]; "/oneparen", [2474$_{16}$]; "/oneperiod", [2488$_{16}$]; "/onepersian", [06F1$_{16}$]; "/onequarter", [00BC$_{16}$]; "/oneroman", [2170$_{16}$]; "/onesuperior", [00B9$_{16}$]; "/onethai", [0E51$_{16}$]; "/onethird", [2153$_{16}$]; "/oogonek", [01EB$_{16}$]; "/oogonekmacron", [01ED$_{16}$]; "/oogurmukhi", [0A13$_{16}$]; "/oomatragurmukhi", [0A4B$_{16}$]; "/oopen", [0254$_{16}$]; "/oparen", [24AA$_{16}$]; "/openbullet", [25E6$_{16}$]; "/option", [2325$_{16}$]; "/ordfeminine", [00AA$_{16}$]; "/ordmasculine", [00BA$_{16}$]; "/orthogonal", [221F$_{16}$]; "/oshortdeva", [0912$_{16}$]; "/oshortvowelsigndeva", [094A$_{16}$]; "/oslash", [00F8$_{16}$]; "/oslashacute", [01FF$_{16}$]; "/osmallhiragana", [3049$_{16}$]; "/osmallkatakana", [30A9$_{16}$]; "/osmallkatakanahalfwidth", [FF6B$_{16}$]; "/ostrokeacute", [01FF$_{16}$]; "/osuperior", [F6F0$_{16}$]; "/otcyrillic", [047F$_{16}$]; "/otilde", [00F5$_{16}$]; "/otildeacute", [1E4D$_{16}$]; "/otildedieresis", [1E4F$_{16}$]; "/oubopomofo", [3121$_{16}$]; "/overline", [203E$_{16}$]; "/overlinecenterline", [FE4A$_{16}$]; "/overlinecmb", [0305$_{16}$]; "/overlinedashed", [FE49$_{16}$]; "/overlinedblwavy", [FE4C$_{16}$]; "/overlinewavy", [FE4B$_{16}$]; "/overscore", [00AF$_{16}$]; "/ovowelsignbengali", [09CB$_{16}$]; "/ovowelsigndeva", [094B$_{16}$]; "/ovowelsigngujarati", [0ACB$_{16}$]; "/p", [0070$_{16}$]; "/paampssquare", [3380$_{16}$];

"/paasentosquare", [$332B_{16}$]; "/pabengali", [$09AA_{16}$]; "/pacute", [$1E55_{16}$]; "/padeva", [$092A_{16}$]; "/pagedown", [$21DF_{16}$]; "/pageup", [$21DE_{16}$]; "/pagujarati", [$0AAA_{16}$]; "/pagurmukhi", [$0A2A_{16}$]; "/pahiragana", [3071_{16}]; "/paiyannoithai", [$0E2F_{16}$]; "/pakatakana", [$30D1_{16}$]; "/palatalizationcyrilliccmb", [0484_{16}]; "/palochkacyrillic", [$04C0_{16}$]; "/pansioskorean", [$317F_{16}$]; "/paragraph", [$00B6_{16}$]; "/parallel", [2225_{16}]; "/parenleft", [0028_{16}]; "/parenleftaltonearabic", [$FD3E_{16}$]; "/parenleftbt", [$F8ED_{16}$]; "/parenleftex", [$F8EC_{16}$]; "/parenleftinferior", [$208D_{16}$]; "/parenleftmonospace", [$FF08_{16}$]; "/parenleftsmall", [$FE59_{16}$]; "/parenleftsuperior", [$207D_{16}$]; "/parenlefttp", [$F8EB_{16}$]; "/parenleftvertical", [$FE35_{16}$]; "/parenright", [0029_{16}]; "/parenrightaltonearabic", [$FD3F_{16}$]; "/parenrightbt", [$F8F8_{16}$]; "/parenrightex", [$F8F7_{16}$]; "/parenrightinferior", [$208E_{16}$]; "/parenrightmonospace", [$FF09_{16}$]; "/parenrightsmall", [$FE5A_{16}$]; "/parenrightsuperior", [$207E_{16}$]; "/parenrighttp", [$F8F6_{16}$]; "/parenrightvertical", [$FE36_{16}$]; "/partialdiff", [2202_{16}]; "/paseqhebrew", [$05C0_{16}$]; "/pashtahebrew", [0599_{16}]; "/pasquare", [$33A9_{16}$]; "/patah", [$05B7_{16}$]; "/patah11", [$05B7_{16}$]; "/patah1d", [$05B7_{16}$]; "/patah2a", [$05B7_{16}$]; "/patahhebrew", [$05B7_{16}$]; "/patahnarrowhebrew", [$05B7_{16}$]; "/patahquarterhebrew", [$05B7_{16}$]; "/patahwidehebrew", [$05B7_{16}$]; "/pazerhebrew", [$05A1_{16}$]; "/pbopomofo", [3106_{16}]; "/pcircle", [$24DF_{16}$]; "/pdotaccent", [$1E57_{16}$]; "/pe", [$05E4_{16}$]; "/pecyrillic", [$043F_{16}$]; "/pedagesh", [$FB44_{16}$]; "/pedageshhebrew", [$FB44_{16}$]; "/peezisquare", [$333B_{16}$]; "/pefinaldageshhebrew", [$FB43_{16}$]; "/peharabic", [$067E_{16}$]; "/peharmenian", [$057A_{16}$]; "/pehebrew", [$05E4_{16}$]; "/pehfinalarabic", [$FB57_{16}$]; "/pehinitialarabic", [$FB58_{16}$]; "/pehiragana", [$307A_{16}$]; "/pehmedialarabic", [$FB59_{16}$]; "/pekatakana", [$30DA_{16}$]; "/pemiddlehookcyrillic", [$04A7_{16}$]; "/perafehebrew", [$FB4E_{16}$]; "/percent", [0025_{16}]; "/percentarabic", [$066A_{16}$]; "/percentmonospace", [$FF05_{16}$]; "/percentsmall", [$FE6A_{16}$]; "/period", [$002E_{16}$]; "/periodarmenian", [0589_{16}]; "/periodcentered", [$00B7_{16}$]; "/periodhalfwidth", [$FF61_{16}$]; "/periodinferior", [$F6E7_{16}$]; "/periodmonospace", [$FF0E_{16}$]; "/periodsmall", [$FE52_{16}$]; "/periodsuperior", [$F6E8_{16}$]; "/perispomenigreekcmb", [0342_{16}]; "/perpendicular", [$22A5_{16}$]; "/perthousand", [2030_{16}]; "/peseta", [$20A7_{16}$]; "/pfsquare", [$338A_{16}$]; "/phabengali", [$09AB_{16}$]; "/phadeva", [$092B_{16}$]; "/phagujarati", [$0AAB_{16}$]; "/phagurmukhi", [$0A2B_{16}$]; "/phi", [$03C6_{16}$]; "/phi1", [$03D5_{16}$]; "/phieuphacirclekorean", [$327A_{16}$]; "/phieuphaparenkorean", [$321A_{16}$]; "/phieuphcirclekorean", [$326C_{16}$]; "/phieuphkorean", [$314D_{16}$]; "/phieuphparenkorean", [$320C_{16}$]; "/philatin", [0278_{16}]; "/phinthuthai", [$0E3A_{16}$]; "/phisymbolgreek", [$03D5_{16}$]; "/phook", [$01A5_{16}$]; "/phophanthai", [$0E1E_{16}$]; "/phophungthai", [$0E1C_{16}$]; "/phosamphaothai", [$0E20_{16}$]; "/pi", [$03C0_{16}$]; "/pieupacirclekorean", [3273_{16}]; "/pieupaparenkorean", [3213_{16}]; "/pieupcieuckorean", [3176_{16}]; "/pieupcirclekorean", [3265_{16}]; "/pieupkiyeokkorean", [3172_{16}]; "/pieupkorean", [3142_{16}]; "/pieupparenkorean", [3205_{16}]; "/pieupsioskiyeokkorean", [3174_{16}]; "/pieupsioskorean", [3144_{16}]; "/pieupsiostikeutkorean", [3175_{16}]; "/pieupthieuthkorean", [3177_{16}]; "/pieuptikeutkorean", [3173_{16}]; "/pihiragana", [3074_{16}]; "/pikatakana", [$30D4_{16}$]; "/pisymbolgreek", [$03D6_{16}$]; "/piwrarmenian", [0583_{16}]; "/plus", [$002B_{16}$]; "/plusbelowcmb", [$031F_{16}$]; "/pluscircle", [2295_{16}]; "/plusminus", [$00B1_{16}$]; "/plusmod", [$02D6_{16}$]; "/plusmonospace", [$FF0B_{16}$]; "/plussmall", [$FE62_{16}$]; "/plussuperior", [$207A_{16}$]; "/pmonospace", [$FF50_{16}$]; "/pmsquare", [$33D8_{16}$]; "/pohiragana", [$307D_{16}$]; "/pointingindexdownwhite", [$261F_{16}$]; "/pointingindexleftwhite", [$261C_{16}$]; "/pointingindexrightwhite", [$261E_{16}$];

"/pointingindexupwhite", [261D$_{16}$]; "/pokatakana", [30DD$_{16}$]; "/poplathai", [0E1B$_{16}$]; "/postalmark", [3012$_{16}$]; "/postalmarkface", [3020$_{16}$]; "/pparen", [24AB$_{16}$]; "/precedes", [227A$_{16}$]; "/prescription", [211E$_{16}$]; "/primemod", [02B9$_{16}$]; "/primereversed", [2035$_{16}$]; "/product", [220F$_{16}$]; "/projective", [2305$_{16}$]; "/prolongedkana", [30FC$_{16}$]; "/propellor", [2318$_{16}$]; "/propersubset", [2282$_{16}$]; "/propersuperset", [2283$_{16}$]; "/proportion", [2237$_{16}$]; "/proportional", [221D$_{16}$]; "/psi", [03C8$_{16}$]; "/psicyrillic", [0471$_{16}$]; "/psilipneumatacyrilliccmb", [0486$_{16}$]; "/pssquare", [33B0$_{16}$]; "/puhiragana", [3077$_{16}$]; "/pukatakana", [30D7$_{16}$]; "/pvsquare", [33B4$_{16}$]; "/pwsquare", [33BA$_{16}$]; "/q", [0071$_{16}$]; "/qadeva", [0958$_{16}$]; "/qadmahebrew", [05A8$_{16}$]; "/qafarabic", [0642$_{16}$]; "/qaffinalarabic", [FED6$_{16}$]; "/qafinitialarabic", [FED7$_{16}$]; "/qafmedialarabic", [FED8$_{16}$]; "/qamats", [05B8$_{16}$]; "/qamats10", [05B8$_{16}$]; "/qamats1a", [05B8$_{16}$]; "/qamats1c", [05B8$_{16}$]; "/qamats27", [05B8$_{16}$]; "/qamats29", [05B8$_{16}$]; "/qamats33", [05B8$_{16}$]; "/qamatsde", [05B8$_{16}$]; "/qamatshebrew", [05B8$_{16}$]; "/qamatsnarrowhebrew", [05B8$_{16}$]; "/qamatsqatanhebrew", [05B8$_{16}$]; "/qamatsqatannarrowhebrew", [05B8$_{16}$]; "/qamatsqatanquarterhebrew", [05B8$_{16}$]; "/qamatsqatanwidehebrew", [05B8$_{16}$]; "/qamatsquarterhebrew", [05B8$_{16}$]; "/qamatswidehebrew", [05B8$_{16}$]; "/qarneyparahebrew", [059F$_{16}$]; "/qbopomofo", [3111$_{16}$]; "/qcircle", [24E0$_{16}$]; "/qhook", [02A0$_{16}$]; "/qmonospace", [FF51$_{16}$]; "/qof", [05E7$_{16}$]; "/qofdagesh", [FB47$_{16}$]; "/qofdageshhebrew", [FB47$_{16}$]; "/qofhatafpatah", [05E7$_{16}$; 05B2$_{16}$]; "/qofhatafpatahhebrew", [05E7$_{16}$; 05B2$_{16}$]; "/qofhatafsegol", [05E7$_{16}$; 05B1$_{16}$]; "/qofhatafsegolhebrew", [05E7$_{16}$; 05B1$_{16}$]; "/qofhebrew", [05E7$_{16}$]; "/qofhiriq", [05E7$_{16}$; 05B4$_{16}$]; "/qofhiriqhebrew", [05E7$_{16}$; 05B4$_{16}$]; "/qofholam", [05E7$_{16}$; 05B9$_{16}$]; "/qofholamhebrew", [05E7$_{16}$; 05B9$_{16}$]; "/qofpatah", [05E7$_{16}$; 05B7$_{16}$]; "/qofpatahhebrew", [05E7$_{16}$; 05B7$_{16}$]; "/qofqamats", [05E7$_{16}$; 05B8$_{16}$]; "/qofqamatshebrew", [05E7$_{16}$; 05B8$_{16}$];
"/qofqubuts", [05E7$_{16}$; 05BB$_{16}$]; "/qofqubutshebrew", [05E7$_{16}$; 05BB$_{16}$]; "/qofsegol", [05E7$_{16}$; 05B6$_{16}$]; "/qofsegolhebrew", [05E7$_{16}$; 05B6$_{16}$]; "/qofsheva", [05E7$_{16}$; 05B0$_{16}$]; "/qofshevahebrew", [05E7$_{16}$; 05B0$_{16}$]; "/qoftsere", [05E7$_{16}$; 05B5$_{16}$]; "/qoftserehebrew", [05E7$_{16}$; 05B5$_{16}$]; "/qparen", [24AC$_{16}$]; "/quarternote", [2669$_{16}$]; "/qubuts", [05BB$_{16}$]; "/qubuts18", [05BB$_{16}$]; "/qubuts25", [05BB$_{16}$]; "/qubuts31", [05BB$_{16}$]; "/qubutshebrew", [05BB$_{16}$]; "/qubutsnarrowhebrew", [05BB$_{16}$]; "/qubutsquarterhebrew", [05BB$_{16}$]; "/qubutswidehebrew", [05BB$_{16}$]; "/question", [003F$_{16}$]; "/questionarabic", [061F$_{16}$]; "/questionarmenian", [055E$_{16}$]; "/questiondown", [00BF$_{16}$]; "/questiondownsmall", [F7BF$_{16}$]; "/questiongreek", [037E$_{16}$]; "/questionmonospace", [FF1F$_{16}$]; "/questionsmall", [F73F$_{16}$]; "/quotedbl", [0022$_{16}$]; "/quotedblbase", [201E$_{16}$]; "/quotedblleft", [201C$_{16}$]; "/quotedblmonospace", [FF02$_{16}$]; "/quotedblprime", [301E$_{16}$]; "/quotedblprimereversed", [301D$_{16}$]; "/quotedblright", [201D$_{16}$]; "/quoteleft", [2018$_{16}$]; "/quoteleftreversed", [201B$_{16}$]; "/quotereversed", [201B$_{16}$]; "/quoteright", [2019$_{16}$]; "/quoterightn", [0149$_{16}$]; "/quotesinglbase", [201A$_{16}$]; "/quotesingle", [0027$_{16}$]; "/quotesinglemonospace", [FF07$_{16}$]; "/r", [0072$_{16}$]; "/raarmenian", [057C$_{16}$]; "/rabengali", [09B0$_{16}$]; "/racute", [0155$_{16}$]; "/radeva", [0930$_{16}$]; "/radical", [221A$_{16}$]; "/radicalex", [F8E5$_{16}$]; "/radoverssquare", [33AE$_{16}$]; "/radoverssquaredsquare", [33AF$_{16}$]; "/radsquare", [33AD$_{16}$]; "/rafe", [05BF$_{16}$]; "/rafehebrew", [05BF$_{16}$]; "/ragujarati", [0AB0$_{16}$]; "/ragurmukhi", [0A30$_{16}$]; "/rahiragana", [3089$_{16}$]; "/rakatakana", [30E9$_{16}$]; "/rakatakanahalfwidth", [FF97$_{16}$]; "/ralowerdiagonalbengali", [09F1$_{16}$]; "/ramiddlediagonalbengali", [09F0$_{16}$]; "/ramshorn", [0264$_{16}$]; "/ratio", [2236$_{16}$];

"/rbopomofo", [3116$_{16}$]; "/rcaron", [0159$_{16}$]; "/rcedilla", [0157$_{16}$]; "/rcircle", [24E1$_{16}$]; "/rcommaaccent", [0157$_{16}$]; "/rdblgrave", [0211$_{16}$]; "/rdotaccent", [1E59$_{16}$]; "/rdotbelow", [1E5B$_{16}$]; "/rdotbelowmacron", [1E5D$_{16}$]; "/referencemark", [203B$_{16}$]; "/reflexsubset", [2286$_{16}$]; "/reflexsuperset", [2287$_{16}$]; "/registered", [00AE$_{16}$]; "/registersans", [F8E8$_{16}$]; "/registerserif", [F6DA$_{16}$]; "/reharabic", [0631$_{16}$]; "/reharmenian", [0580$_{16}$]; "/rehfinalarabic", [FEAE$_{16}$]; "/rehiragana", [308C$_{16}$]; "/rehyehaleflamarabic", [0631$_{16}$; FEF3$_{16}$; FE8E$_{16}$;
0644$_{16}$]; "/rekatakana", [30EC$_{16}$]; "/rekatakanahalfwidth", [FF9A$_{16}$]; "/resh", [05E8$_{16}$]; "/reshdageshhebrew", [FB48$_{16}$]; "/reshhatafpatah", [05E8$_{16}$; 05B2$_{16}$]; "/reshhatafpatahhebrew", [05E8$_{16}$; 05B2$_{16}$]; "/reshhatafsegol", [05E8$_{16}$; 05B1$_{16}$]; "/reshhatafsegolhebrew", [05E8$_{16}$; 05B1$_{16}$]; "/reshhebrew", [05E8$_{16}$]; "/reshhiriq", [05E8$_{16}$; 05B4$_{16}$]; "/reshhiriqhebrew", [05E8$_{16}$; 05B4$_{16}$]; "/reshholam", [05E8$_{16}$; 05B9$_{16}$]; "/reshholamhebrew", [05E8$_{16}$; 05B9$_{16}$]; "/reshpatah", [05E8$_{16}$; 05B7$_{16}$]; "/reshpatahhebrew", [05E8$_{16}$; 05B7$_{16}$]; "/reshqamats", [05E8$_{16}$; 05B8$_{16}$]; "/reshqamatshebrew", [05E8$_{16}$; 05B8$_{16}$]; "/reshqubuts", [05E8$_{16}$; 05BB$_{16}$]; "/reshqubutshebrew", [05E8$_{16}$; 05BB$_{16}$]; "/reshsegol", [05E8$_{16}$; 05B6$_{16}$]; "/reshsegolhebrew", [05E8$_{16}$; 05B6$_{16}$]; "/reshsheva", [05E8$_{16}$; 05B0$_{16}$]; "/reshshevahebrew", [05E8$_{16}$; 05B0$_{16}$]; "/reshtsere", [05E8$_{16}$; 05B5$_{16}$]; "/reshtserehebrew", [05E8$_{16}$; 05B5$_{16}$]; "/reversedtilde", [223D$_{16}$]; "/reviahebrew", [0597$_{16}$]; "/reviamugrashhebrew", [0597$_{16}$]; "/revlogicalnot", [2310$_{16}$]; "/rfishhook", [027E$_{16}$]; "/rfishhookreversed", [027F$_{16}$]; "/rhabengali", [09DD$_{16}$]; "/rhadeva", [095D$_{16}$]; "/rho", [03C1$_{16}$]; "/rhook", [027D$_{16}$]; "/rhookturned", [027B$_{16}$]; "/rhookturnedsuperior", [02B5$_{16}$]; "/rhosymbolgreek", [03F1$_{16}$]; "/rhotichookmod", [02DE$_{16}$]; "/rieulacirclekorean", [3271$_{16}$]; "/rieulaparenkorean", [3211$_{16}$]; "/rieulcirclekorean", [3263$_{16}$]; "/rieulhieuhkorean", [3140$_{16}$]; "/rieulkiyeokkorean", [313A$_{16}$]; "/rieulkiyeoksioskorean", [3169$_{16}$]; "/rieulkorean", [3139$_{16}$]; "/rieulmieumkorean", [313B$_{16}$]; "/rieulpansioskorean", [316C$_{16}$]; "/rieulparenkorean", [3203$_{16}$]; "/rieulphieuphkorean", [313F$_{16}$]; "/rieulpieupkorean", [313C$_{16}$]; "/rieulpieupsioskorean", [316B$_{16}$]; "/rieulsioskorean", [313D$_{16}$]; "/rieulthieuthkorean", [313E$_{16}$]; "/rieultikeutkorean", [316A$_{16}$]; "/rieulyeorinhieuhkorean", [316D$_{16}$]; "/rightangle", [221F$_{16}$]; "/righttackbelowcmb", [0319$_{16}$]; "/righttriangle", [22BF$_{16}$]; "/rihiragana", [308A$_{16}$]; "/rikatakana", [30EA$_{16}$]; "/rikatakanahalfwidth", [FF98$_{16}$]; "/ring", [02DA$_{16}$]; "/ringbelowcmb", [0325$_{16}$]; "/ringcmb", [030A$_{16}$]; "/ringhalfleft", [02BF$_{16}$]; "/ringhalfleftarmenian", [0559$_{16}$]; "/ringhalfleftbelowcmb", [031C$_{16}$]; "/ringhalfleftcentered", [02D3$_{16}$]; "/ringhalfright", [02BE$_{16}$]; "/ringhalfrightbelowcmb", [0339$_{16}$]; "/ringhalfrightcentered", [02D2$_{16}$]; "/rinvertedbreve", [0213$_{16}$]; "/rittorusquare", [3351$_{16}$]; "/rlinebelow", [1E5F$_{16}$]; "/rlongleg", [027C$_{16}$]; "/rlonglegturned", [027A$_{16}$]; "/rmonospace", [FF52$_{16}$]; "/rohiragana", [308D$_{16}$]; "/rokatakana", [30ED$_{16}$]; "/rokatakanahalfwidth", [FF9B$_{16}$]; "/roruathai", [0E23$_{16}$]; "/rparen", [24AD$_{16}$]; "/rrabengali", [09DC$_{16}$]; "/rradeva", [0931$_{16}$]; "/rragurmukhi", [0A5C$_{16}$]; "/rreharabic", [0691$_{16}$]; "/rrehfinalarabic", [FB8D$_{16}$]; "/rrvocalicbengali", [09E0$_{16}$]; "/rrvocalicdeva", [0960$_{16}$]; "/rrvocalicgujarati", [0AE0$_{16}$]; "/rrvocalicvowelsignbengali", [09C4$_{16}$]; "/rrvocalicvowelsigndeva", [0944$_{16}$]; "/rrvocalicvowelsigngujarati", [0AC4$_{16}$]; "/rsuperior", [F6F1$_{16}$]; "/rtblock", [2590$_{16}$]; "/rturned", [0279$_{16}$]; "/rturnedsuperior", [02B4$_{16}$]; "/ruhiragana", [308B$_{16}$]; "/rukatakana", [30EB$_{16}$]; "/rukatakanahalfwidth",

[FF99$_{16}$]; "/rupeemarkbengali", [09F2$_{16}$]; "/rupeesignbengali", [09F3$_{16}$]; "/rupiah", [F6DD$_{16}$]; "/ruthai", [0E24$_{16}$]; "/rvocalicbengali", [098B$_{16}$]; "/rvocalicdeva", [090B$_{16}$]; "/rvocalicgujarati", [0A8B$_{16}$]; "/rvocalicvowelsignbengali", [09C3$_{16}$]; "/rvocalicvowelsigndeva", [0943$_{16}$]; "/rvocalicvowelsigngujarati", [0AC3$_{16}$]; "/s", [0073$_{16}$]; "/sabengali", [09B8$_{16}$]; "/sacute", [015B$_{16}$]; "/sacutedotaccent", [1E65$_{16}$]; "/sadarabic", [0635$_{16}$]; "/sadeva", [0938$_{16}$]; "/sadfinalarabic", [FEBA$_{16}$]; "/sadinitialarabic", [FEBB$_{16}$]; "/sadmedialarabic", [FEBC$_{16}$]; "/sagujarati", [0AB8$_{16}$]; "/sagurmukhi", [0A38$_{16}$]; "/sahiragana", [3055$_{16}$]; "/sakatakana", [30B5$_{16}$]; "/sakatakanahalfwidth", [FF7B$_{16}$]; "/sallallahoualayhewasallamarabic", [FDFA$_{16}$]; "/samekh", [05E1$_{16}$]; "/samekhdagesh", [FB41$_{16}$]; "/samekhdageshhebrew", [FB41$_{16}$]; "/samekhhebrew", [05E1$_{16}$]; "/saraaathai", [0E32$_{16}$]; "/saraaethai", [0E41$_{16}$]; "/saraaimaimalaithai", [0E44$_{16}$]; "/saraaimaimuanthai", [0E43$_{16}$]; "/saraamthai", [0E33$_{16}$]; "/saraathai", [0E30$_{16}$]; "/saraethai", [0E40$_{16}$]; "/saraiileftthai", [F886$_{16}$]; "/saraiithai", [0E35$_{16}$]; "/saraileftthai", [F885$_{16}$]; "/saraithai", [0E34$_{16}$]; "/saraothai", [0E42$_{16}$]; "/saraueeleftthai", [F888$_{16}$]; "/saraueethai", [0E37$_{16}$]; "/saraueleftthai", [F887$_{16}$]; "/sarauethai", [0E36$_{16}$]; "/sarauthai", [0E38$_{16}$]; "/sarauuthai", [0E39$_{16}$]; "/sbopomofo", [3119$_{16}$]; "/scaron", [0161$_{16}$]; "/scarondotaccent", [1E67$_{16}$]; "/scedilla", [015F$_{16}$]; "/schwa", [0259$_{16}$]; "/schwacyrillic", [04D9$_{16}$]; "/schwadieresiscyrillic", [04DB$_{16}$]; "/schwahook", [025A$_{16}$]; "/scircle", [24E2$_{16}$]; "/scircumflex", [015D$_{16}$]; "/scommaaccent", [0219$_{16}$]; "/sdotaccent", [1E61$_{16}$]; "/sdotbelow", [1E63$_{16}$]; "/sdotbelowdotaccent", [1E69$_{16}$]; "/seagullbelowcmb", [033C$_{16}$]; "/second", [2033$_{16}$]; "/secondtonechinese", [02CA$_{16}$]; "/section", [00A7$_{16}$]; "/seenarabic", [0633$_{16}$]; "/seenfinalarabic", [FEB2$_{16}$]; "/seeninitialarabic", [FEB3$_{16}$]; "/seenmedialarabic", [FEB4$_{16}$]; "/segol", [05B6$_{16}$]; "/segol13", [05B6$_{16}$]; "/segol1f", [05B6$_{16}$]; "/segol2c", [05B6$_{16}$]; "/segolhebrew", [05B6$_{16}$]; "/segolnarrowhebrew", [05B6$_{16}$]; "/segolquarterhebrew", [05B6$_{16}$]; "/segoltahebrew", [0592$_{16}$]; "/segolwidehebrew", [05B6$_{16}$]; "/seharmenian", [057D$_{16}$]; "/sehiragana", [305B$_{16}$]; "/sekatakana", [30BB$_{16}$]; "/sekatakanahalfwidth", [FF7E$_{16}$]; "/semicolon", [003B$_{16}$]; "/semicolonarabic", [061B$_{16}$]; "/semicolonmonospace", [FF1B$_{16}$]; "/semicolonsmall", [FE54$_{16}$]; "/semivoicedmarkkana", [309C$_{16}$]; "/semivoicedmarkkanahalfwidth", [FF9F$_{16}$]; "/sentisquare", [3322$_{16}$]; "/sentosquare", [3323$_{16}$]; "/seven", [0037$_{16}$]; "/sevenarabic", [0667$_{16}$]; "/sevenbengali", [09ED$_{16}$]; "/sevencircle", [2466$_{16}$]; "/sevencircleinversesansserif", [2790$_{16}$]; "/sevendeva", [096D$_{16}$]; "/seveneighths", [215E$_{16}$]; "/sevengujarati", [0AED$_{16}$]; "/sevengurmukhi", [0A6D$_{16}$]; "/sevenhackarabic", [0667$_{16}$]; "/sevenhangzhou", [3027$_{16}$]; "/sevenideographicparen", [3226$_{16}$]; "/seveninferior", [2087$_{16}$]; "/sevenmonospace", [FF17$_{16}$]; "/sevenoldstyle", [F737$_{16}$]; "/sevenparen", [247A$_{16}$]; "/sevenperiod", [248E$_{16}$]; "/sevenpersian", [06F7$_{16}$]; "/sevenroman", [2176$_{16}$]; "/sevensuperior", [2077$_{16}$]; "/seventeencircle", [2470$_{16}$]; "/seventeenparen", [2484$_{16}$]; "/seventeenperiod", [2498$_{16}$]; "/seventhai", [0E57$_{16}$]; "/sfthyphen", [00AD$_{16}$]; "/shaarmenian", [0577$_{16}$]; "/shabengali", [09B6$_{16}$]; "/shacyrillic", [0448$_{16}$]; "/shaddaarabic", [0651$_{16}$]; "/shaddadammaarabic", [FC61$_{16}$]; "/shaddadammatanarabic", [FC5E$_{16}$]; "/shaddafathaarabic", [FC60$_{16}$]; "/shaddafathatanarabic", [0651$_{16}$; 064B$_{16}$]; "/shaddakasraarabic", [FC62$_{16}$]; "/shaddakasratanarabic", [FC5F$_{16}$]; "/shade", [2592$_{16}$]; "/shadedark", [2593$_{16}$]; "/shadelight", [2591$_{16}$]; "/shademedium", [2592$_{16}$]; "/shadeva", [0936$_{16}$]; "/shagujarati", [0AB6$_{16}$]; "/shagurmukhi", [0A36$_{16}$]; "/shalshelethebrew", [0593$_{16}$]; "/shbopomofo", [3115$_{16}$]; "/shchacyrillic", [0449$_{16}$]; "/sheenarabic", [0634$_{16}$]; "/sheenfinalarabic", [FEB6$_{16}$]; "/sheeninitialarabic", [FEB7$_{16}$]; "/sheenmedialarabic", [FEB8$_{16}$]; "/sheicoptic", [03E3$_{16}$]; "/sheqel", [20AA$_{16}$]; "/sheqelhebrew", [20AA$_{16}$]; "/sheva", [05B0$_{16}$]; "/sheva115", [05B0$_{16}$]; "/sheva15", [05B0$_{16}$]; "/sheva22", [05B0$_{16}$]; "/sheva2e", [05B0$_{16}$]; "/shevahebrew", [05B0$_{16}$]; "/shevanarrowhebrew", [05B0$_{16}$]; "/shevaquarterhebrew", [05B0$_{16}$]; "/shevawidehebrew", [05B0$_{16}$]; "/shhacyrillic", [04BB$_{16}$]; "/shimacoptic", [03ED$_{16}$]; "/shin", [05E9$_{16}$]; "/shindagesh", [FB49$_{16}$]; "/shindageshhebrew", [FB49$_{16}$]; "/shindageshshindot", [FB2C$_{16}$]; "/shindageshshindothebrew", [FB2C$_{16}$]; "/shindageshsindot", [FB2D$_{16}$]; "/shindageshsindothebrew", [FB2D$_{16}$]; "/shindothebrew", [05C1$_{16}$]; "/shinhebrew", [05E9$_{16}$]; "/shinshindot", [FB2A$_{16}$]; "/shinshindothebrew", [FB2A$_{16}$]; "/shinsindot", [FB2B$_{16}$]; "/shinsindothebrew", [FB2B$_{16}$]; "/shook", [0282$_{16}$]; "/sigma", [03C3$_{16}$]; "/sigma1", [03C2$_{16}$]; "/sigmafinal", [03C2$_{16}$]; "/sigmalunatesymbolgreek", [03F2$_{16}$]; "/sihiragana", [3057$_{16}$]; "/sikatakana", [30B7$_{16}$]; "/sikatakanahalfwidth", [FF7C$_{16}$]; "/siluqhebrew", [05BD$_{16}$]; "/siluqlefthebrew", [05BD$_{16}$]; "/similar", [223C$_{16}$]; "/sindothebrew", [05C2$_{16}$]; "/siosacirclekorean", [3274$_{16}$]; "/siosaparenkorean", [3214$_{16}$]; "/sioscieuckorean", [317E$_{16}$]; "/sioscirclekorean", [3266$_{16}$]; "/sioskiyeokkorean", [317A$_{16}$]; "/sioskorean", [3145$_{16}$]; "/siosnieunkorean", [317B$_{16}$]; "/siosparenkorean", [3206$_{16}$]; "/siospieupkorean", [317D$_{16}$]; "/siostikeutkorean", [317C$_{16}$]; "/six", [0036$_{16}$]; "/sixarabic", [0666$_{16}$]; "/sixbengali", [09EC$_{16}$]; "/sixcircle", [2465$_{16}$]; "/sixcircleinversesansserif", [278F$_{16}$]; "/sixdeva", [096C$_{16}$]; "/sixgujarati", [0AEC$_{16}$]; "/sixgurmukhi", [0A6C$_{16}$]; "/sixhackarabic", [0666$_{16}$]; "/sixhangzhou", [3026$_{16}$]; "/sixideographicparen", [3225$_{16}$]; "/sixinferior", [2086$_{16}$]; "/sixmonospace", [FF16$_{16}$]; "/sixoldstyle", [F736$_{16}$]; "/sixparen", [2479$_{16}$]; "/sixperiod", [248D$_{16}$]; "/sixpersian", [06F6$_{16}$]; "/sixroman", [2175$_{16}$]; "/sixsuperior", [2076$_{16}$]; "/sixteencircle", [246F$_{16}$]; "/sixteencurrencydenominatorbengali", [09F9$_{16}$]; "/sixteenparen", [2483$_{16}$]; "/sixteenperiod", [2497$_{16}$]; "/sixthai", [0E56$_{16}$]; "/slash", [002F$_{16}$]; "/slashmonospace", [FF0F$_{16}$]; "/slong", [017F$_{16}$]; "/slongdotaccent", [1E9B$_{16}$]; "/smileface", [263A$_{16}$]; "/smonospace", [FF53$_{16}$]; "/sofpasuqhebrew", [05C3$_{16}$]; "/softhyphen", [00AD$_{16}$]; "/softsigncyrillic", [044C$_{16}$]; "/sohiragana", [305D$_{16}$]; "/sokatakana", [30BD$_{16}$]; "/sokatakanahalfwidth", [FF7F$_{16}$]; "/soliduslongoverlaycmb", [0338$_{16}$]; "/solidusshortoverlaycmb", [0337$_{16}$]; "/sorusithai", [0E29$_{16}$]; "/sosalathai", [0E28$_{16}$]; "/sosothai", [0E0B$_{16}$]; "/sosuathai", [0E2A$_{16}$]; "/space", [0020$_{16}$]; "/spacehackarabic", [0020$_{16}$]; "/spade", [2660$_{16}$]; "/spadesuitblack", [2660$_{16}$]; "/spadesuitwhite", [2664$_{16}$]; "/sparen", [24AE$_{16}$]; "/squarebelowcmb", [033B$_{16}$]; "/squarecc", [33C4$_{16}$]; "/squarecm", [339D$_{16}$]; "/squarediagonalcrosshatchfill", [25A9$_{16}$]; "/squarehorizontalfill", [25A4$_{16}$]; "/squarekg", [338F$_{16}$]; "/squarekm", [339E$_{16}$]; "/squarekmcapital", [33CE$_{16}$]; "/squareln", [33D1$_{16}$]; "/squarelog", [33D2$_{16}$]; "/squaremg", [338E$_{16}$]; "/squaremil", [33D5$_{16}$]; "/squaremm", [339C$_{16}$]; "/squaremsquared", [33A1$_{16}$]; "/squareorthogonalcrosshatchfill", [25A6$_{16}$]; "/squareupperlefttolowerrightfill", [25A7$_{16}$]; "/squareupperrighttolowerleftfill", [25A8$_{16}$]; "/squareverticalfill", [25A5$_{16}$]; "/squarewhitewithsmallblack", [25A3$_{16}$]; "/srsquare", [33DB$_{16}$]; "/ssabengali", [09B7$_{16}$]; "/ssadeva", [0937$_{16}$]; "/ssagujarati", [0AB7$_{16}$]; "/ssangcieuckorean", [3149$_{16}$]; "/ssanghieuhkorean", [3185$_{16}$]; "/ssangieungkorean", [3180$_{16}$]; "/ssangkiyeokkorean", [3132$_{16}$]; "/ssangnieunkorean", [3165$_{16}$];

"/ssangpieupkorean", [3143_{16}]; "/ssangsioskorean", [3146_{16}]; "/ssangtikeutkorean", [3138_{16}]; "/ssuperior", [$F6F2_{16}$]; "/sterling", [$00A3_{16}$]; "/sterlingmonospace", [$FFE1_{16}$]; "/strokelongoverlaycmb", [0336_{16}]; "/strokeshortoverlaycmb", [0335_{16}]; "/subset", [2282_{16}]; "/subsetnotequal", [$228A_{16}$]; "/subsetorequal", [2286_{16}]; "/succeeds", [$227B_{16}$]; "/suchthat", [$220B_{16}$]; "/suhiragana", [3059_{16}]; "/sukatakana", [$30B9_{16}$]; "/sukatakanahalfwidth", [$FF7D_{16}$]; "/sukunarabic", [0652_{16}]; "/summation", [2211_{16}]; "/sun", [$263C_{16}$]; "/superset", [2283_{16}]; "/supersetnotequal", [$228B_{16}$]; "/supersetorequal", [2287_{16}]; "/svsquare", [$33DC_{16}$]; "/syouwaerasquare", [$337C_{16}$]; "/t", [0074_{16}]; "/tabengali", [$09A4_{16}$]; "/tackdown", [$22A4_{16}$]; "/tackleft", [$22A3_{16}$]; "/tadeva", [0924_{16}]; "/tagujarati", [$0AA4_{16}$]; "/tagurmukhi", [$0A24_{16}$]; "/taharabic", [0637_{16}]; "/tahfinalarabic", [$FEC2_{16}$]; "/tahinitialarabic", [$FEC3_{16}$]; "/tahiragana", [$305F_{16}$]; "/tahmedialarabic", [$FEC4_{16}$]; "/taisyouerasquare", [$337D_{16}$]; "/takatakana", [$30BF_{16}$]; "/takatakanahalfwidth", [$FF80_{16}$]; "/tatweelarabic", [0640_{16}]; "/tau", [$03C4_{16}$]; "/tav", [$05EA_{16}$]; "/tavdages", [$FB4A_{16}$]; "/tavdagesh", [$FB4A_{16}$]; "/tavdageshhebrew", [$FB4A_{16}$]; "/tavhebrew", [$05EA_{16}$]; "/tbar", [0167_{16}]; "/tbopomofo", [$310A_{16}$]; "/tcaron", [0165_{16}]; "/tccurl", [$02A8_{16}$]; "/tcedilla", [0163_{16}]; "/tcheharabic", [0686_{16}]; "/tchehfinalarabic", [$FB7B_{16}$]; "/tchehinitialarabic", [$FB7C_{16}$]; "/tchehmedialarabic", [$FB7D_{16}$]; "/tchehmeeminitialarabic", [$FB7C_{16}$; $FEE4_{16}$]; "/tcircle", [$24E3_{16}$]; "/tcircumflexbelow", [$1E71_{16}$]; "/tcommaaccent", [0163_{16}]; "/tdieresis", [$1E97_{16}$]; "/tdotaccent", [$1E6B_{16}$]; "/tdotbelow", [$1E6D_{16}$]; "/tecyrillic", [0442_{16}]; "/tedescendercyrillic", [$04AD_{16}$]; "/teharabic", [$062A_{16}$]; "/tehfinalarabic", [$FE96_{16}$]; "/tehhahinitialarabic", [$FCA2_{16}$]; "/tehhahisolatedarabic", [$FC0C_{16}$]; "/tehinitialarabic", [$FE97_{16}$]; "/tehiragana", [3066_{16}]; "/tehjeeminitialarabic", [$FCA1_{16}$]; "/tehjeemisolatedarabic", [$FC0B_{16}$]; "/tehmarbutaarabic", [0629_{16}]; "/tehmarbutafinalarabic", [$FE94_{16}$]; "/tehmedialarabic", [$FE98_{16}$]; "/tehmeeminitialarabic", [$FCA4_{16}$]; "/tehmeemisolatedarabic", [$FC0E_{16}$]; "/tehnoonfinalarabic", [$FC73_{16}$]; "/tekatakana", [$30C6_{16}$]; "/tekatakanahalfwidth", [$FF83_{16}$]; "/telephone", [2121_{16}]; "/telephoneblack", [$260E_{16}$]; "/telishagedolahebrew", [$05A0_{16}$]; "/telishaqetanahebrew", [$05A9_{16}$]; "/tencircle", [2469_{16}]; "/tenideographicparen", [3229_{16}]; "/tenparen", [$247D_{16}$]; "/tenperiod", [2491_{16}]; "/tenroman", [2179_{16}]; "/tesh", [$02A7_{16}$]; "/tet", [$05D8_{16}$]; "/tetdagesh", [$FB38_{16}$]; "/tetdageshhebrew", [$FB38_{16}$]; "/tethebrew", [$05D8_{16}$]; "/tetsecyrillic", [$04B5_{16}$]; "/tevirhebrew", [$059B_{16}$]; "/tevirlefthebrew", [$059B_{16}$]; "/thabengali", [$09A5_{16}$]; "/thadeva", [0925_{16}]; "/thagujarati", [$0AA5_{16}$]; "/thagurmukhi", [$0A25_{16}$]; "/thalarabic", [0630_{16}]; "/thalfinalarabic", [$FEAC_{16}$]; "/thanthakhatlowleftthai", [$F898_{16}$]; "/thanthakhatlowrightthai", [$F897_{16}$]; "/thanthakhatthai", [$0E4C_{16}$]; "/thanthakhatupperleftthai", [$F896_{16}$]; "/theharabic", [$062B_{16}$]; "/thehfinalarabic", [$FE9A_{16}$]; "/thehinitialarabic", [$FE9B_{16}$]; "/thehmedialarabic", [$FE9C_{16}$]; "/thereexists", [2203_{16}]; "/therefore", [2234_{16}]; "/theta", [$03B8_{16}$]; "/theta1", [$03D1_{16}$]; "/thetasymbolgreek", [$03D1_{16}$]; "/thieuthacirclekorean", [3279_{16}]; "/thieuthaparenkorean", [3219_{16}]; "/thieuthcirclekorean", [$326B_{16}$]; "/thieuthkorean", [$314C_{16}$]; "/thieuthparenkorean", [$320B_{16}$]; "/thirteencircle", [$246C_{16}$]; "/thirteenparen", [2480_{16}]; "/thirteenperiod", [2494_{16}]; "/thonangmonthothai", [$0E11_{16}$]; "/thook", [$01AD_{16}$]; "/thophuthaothai", [$0E12_{16}$]; "/thorn", [$00FE_{16}$]; "/thothahanthai", [$0E17_{16}$]; "/thothanthai", [$0E10_{16}$]; "/thothongthai", [$0E18_{16}$]; "/thothungthai", [0E16$_{16}$]; "/thousandcyrillic", [0482$_{16}$]; "/thousandsseparatorarabic", [066C$_{16}$]; "/thousandsseparatorpersian", [066C$_{16}$]; "/three", [0033$_{16}$]; "/threearabic", [0663$_{16}$]; "/threebengali", [09E9$_{16}$]; "/threecircle", [2462$_{16}$]; "/threecircleinversesansserif", [278C$_{16}$]; "/threedeva", [0969$_{16}$]; "/threeeighths", [215C$_{16}$]; "/threegujarati", [0AE9$_{16}$]; "/threegurmukhi", [0A69$_{16}$]; "/threehackarabic", [0663$_{16}$]; "/threehangzhou", [3023$_{16}$]; "/threeideographicparen", [3222$_{16}$]; "/threeinferior", [2083$_{16}$]; "/threemonospace", [FF13$_{16}$]; "/threenumeratorbengali", [09F6$_{16}$]; "/threeoldstyle", [F733$_{16}$]; "/threeparen", [2476$_{16}$]; "/threeperiod", [248A$_{16}$]; "/threepersian", [06F3$_{16}$]; "/threequarters", [00BE$_{16}$]; "/threequartersemdash", [F6DE$_{16}$]; "/threeroman", [2172$_{16}$]; "/threesuperior", [00B3$_{16}$]; "/threethai", [0E53$_{16}$]; "/thzsquare", [3394$_{16}$]; "/tihiragana", [3061$_{16}$]; "/tikatakana", [30C1$_{16}$]; "/tikatakanahalfwidth", [FF81$_{16}$]; "/tikeutacirclekorean", [3270$_{16}$]; "/tikeutaparenkorean", [3210$_{16}$]; "/tikeutcirclekorean", [3262$_{16}$]; "/tikeutkorean", [3137$_{16}$]; "/tikeutparenkorean", [3202$_{16}$]; "/tilde", [02DC$_{16}$]; "/tildebelowcmb", [0330$_{16}$]; "/tildecmb", [0303$_{16}$]; "/tildecomb", [0303$_{16}$]; "/tildedoublecmb", [0360$_{16}$]; "/tildeoperator", [223C$_{16}$]; "/tildeoverlaycmb", [0334$_{16}$]; "/tildeverticalcmb", [033E$_{16}$]; "/timescircle", [2297$_{16}$]; "/tipehahebrew", [0596$_{16}$]; "/tipehalefthebrew", [0596$_{16}$]; "/tippigurmukhi", [0A70$_{16}$]; "/titlocyrilliccmb", [0483$_{16}$]; "/tiwnarmenian", [057F$_{16}$]; "/tlinebelow", [1E6F$_{16}$]; "/tmonospace", [FF54$_{16}$]; "/toarmenian", [0569$_{16}$]; "/tohiragana", [3068$_{16}$]; "/tokatakana", [30C8$_{16}$]; "/tokatakanahalfwidth", [FF84$_{16}$]; "/tonebarextrahighmod", [02E5$_{16}$]; "/tonebarextralowmod", [02E9$_{16}$]; "/tonebarhighmod", [02E6$_{16}$]; "/tonebarlowmod", [02E8$_{16}$]; "/tonebarmidmod", [02E7$_{16}$]; "/tonefive", [01BD$_{16}$]; "/tonesix", [0185$_{16}$]; "/tonetwo", [01A8$_{16}$]; "/tonos", [0384$_{16}$]; "/tonsquare", [3327$_{16}$]; "/topatakthai", [0E0F$_{16}$]; "/tortoiseshellbracketleft", [3014$_{16}$]; "/tortoiseshellbracketleftsmall", [FE5D$_{16}$]; "/tortoiseshellbracketleftvertical", [FE39$_{16}$]; "/tortoiseshellbracketright", [3015$_{16}$]; "/tortoiseshellbracketrightsmall", [FE5E$_{16}$]; "/tortoiseshellbracketrightvertical", [FE3A$_{16}$]; "/totaothai", [0E15$_{16}$]; "/tpalatalhook", [01AB$_{16}$]; "/tparen", [24AF$_{16}$]; "/trademark", [2122$_{16}$]; "/trademarksans", [F8EA$_{16}$]; "/trademarkserif", [F6DB$_{16}$]; "/tretroflexhook", [0288$_{16}$]; "/triagdn", [25BC$_{16}$]; "/triaglf", [25C4$_{16}$]; "/triagrt", [25BA$_{16}$]; "/triagup", [25B2$_{16}$]; "/ts", [02A6$_{16}$]; "/tsadi", [05E6$_{16}$]; "/tsadidagesh", [FB46$_{16}$]; "/tsadidageshhebrew", [FB46$_{16}$]; "/tsadihebrew", [05E6$_{16}$]; "/tsecyrillic", [0446$_{16}$]; "/tsere", [05B5$_{16}$]; "/tsere12", [05B5$_{16}$]; "/tsere1e", [05B5$_{16}$]; "/tsere2b", [05B5$_{16}$]; "/tserehebrew", [05B5$_{16}$]; "/tserenarrowhebrew", [05B5$_{16}$]; "/tserequarterhebrew", [05B5$_{16}$]; "/tserewidehebrew", [05B5$_{16}$]; "/tshecyrillic", [045B$_{16}$]; "/tsuperior", [F6F3$_{16}$]; "/ttabengali", [099F$_{16}$]; "/ttadeva", [091F$_{16}$]; "/ttagujarati", [0A9F$_{16}$]; "/ttagurmukhi", [0A1F$_{16}$]; "/tteharabic", [0679$_{16}$]; "/ttehfinalarabic", [FB67$_{16}$]; "/ttehinitialarabic", [FB68$_{16}$]; "/ttehmedialarabic", [FB69$_{16}$]; "/tthabengali", [09A0$_{16}$]; "/tthadeva", [0920$_{16}$]; "/tthagujarati", [0AA0$_{16}$]; "/tthagurmukhi", [0A20$_{16}$]; "/tturned", [0287$_{16}$]; "/tuhiragana", [3064$_{16}$]; "/tukatakana", [30C4$_{16}$]; "/tukatakanahalfwidth", [FF82$_{16}$]; "/tusmallhiragana", [3063$_{16}$]; "/tusmallkatakana", [30C3$_{16}$]; "/tusmallkatakanahalfwidth", [FF6F$_{16}$]; "/twelvecircle", [246B$_{16}$]; "/twelveparen", [247F$_{16}$]; "/twelveperiod", [2493$_{16}$]; "/twelveroman", [217B$_{16}$]; "/twentycircle", [2473$_{16}$]; "/twentyhangzhou", [5344$_{16}$]; "/twentyparen", [2487$_{16}$]; "/twentyperiod", [249B$_{16}$]; "/two", [0032$_{16}$]; "/twoarabic", [0662$_{16}$]; "/twobengali", [09E8$_{16}$]; "/twocircle", [2461$_{16}$];

"/twocircleinversesansserif", [$278B_{16}$]; "/twodeva", [0968_{16}]; "/twodotenleader", [2025_{16}]; "/twodotleader", [2025_{16}]; "/twodotleadervertical", [$FE30_{16}$]; "/twogujarati", [$0AE8_{16}$]; "/twogurmukhi", [$0A68_{16}$]; "/twohackarabic", [0662_{16}]; "/twohangzhou", [3022_{16}]; "/twoideographicparen", [3221_{16}]; "/twoinferior", [2082_{16}]; "/twomonospace", [$FF12_{16}$]; "/twonumeratorbengali", [$09F5_{16}$]; "/twooldstyle", [$F732_{16}$]; "/twoparen", [2475_{16}]; "/twoperiod", [2489_{16}]; "/twopersian", [$06F2_{16}$]; "/tworoman", [2171_{16}]; "/twostroke", [$01BB_{16}$]; "/twosuperior", [$00B2_{16}$]; "/twothai", [$0E52_{16}$]; "/twothirds", [2154_{16}]; "/u", [0075_{16}]; "/uacute", [$00FA_{16}$]; "/ubar", [0289_{16}]; "/ubengali", [0989_{16}]; "/ubopomofo", [3128_{16}]; "/ubreve", [$016D_{16}$]; "/ucaron", [$01D4_{16}$]; "/ucircle", [$24E4_{16}$]; "/ucircumflex", [$00FB_{16}$]; "/ucircumflexbelow", [$1E77_{16}$]; "/ucyrillic", [0443_{16}]; "/udattadeva", [0951_{16}]; "/udblacute", [0171_{16}]; "/udblgrave", [0215_{16}]; "/udeva", [0909_{16}]; "/udieresis", [$00FC_{16}$]; "/udieresisacute", [$01D8_{16}$]; "/udieresisbelow", [$1E73_{16}$]; "/udieresiscaron", [$01DA_{16}$]; "/udieresiscyrillic", [$04F1_{16}$]; "/udieresisgrave", [$01DC_{16}$]; "/udieresismacron", [$01D6_{16}$]; "/udotbelow", [$1EE5_{16}$]; "/ugrave", [$00F9_{16}$]; "/ugujarati", [$0A89_{16}$]; "/ugurmukhi", [$0A09_{16}$]; "/uhiragana", [3046_{16}]; "/uhookabove", [$1EE7_{16}$]; "/uhorn", [$01B0_{16}$]; "/uhornacute", [$1EE9_{16}$]; "/uhorndotbelow", [$1EF1_{16}$]; "/uhorngrave", [$1EEB_{16}$]; "/uhornhookabove", [$1EED_{16}$]; "/uhorntilde", [$1EEF_{16}$]; "/uhungarumlaut", [0171_{16}]; "/uhungarumlautcyrillic", [$04F3_{16}$]; "/uinvertedbreve", [0217_{16}]; "/ukatakana", [$30A6_{16}$]; "/ukatakanahalfwidth", [$FF73_{16}$]; "/ukcyrillic", [0479_{16}]; "/ukorean", [$315C_{16}$]; "/umacron", [$016B_{16}$]; "/umacroncyrillic", [$04EF_{16}$]; "/umacrondieresis", [$1E7B_{16}$]; "/umatragurmukhi", [$0A41_{16}$]; "/umonospace", [$FF55_{16}$]; "/underscore", [$005F_{16}$]; "/underscoredbl", [2017_{16}]; "/underscoremonospace", [$FF3F_{16}$]; "/underscorevertical", [$FE33_{16}$]; "/underscorewavy", [$FE4F_{16}$]; "/union", [$222A_{16}$]; "/universal", [2200_{16}]; "/uogonek", [0173_{16}]; "/uparen", [$24B0_{16}$]; "/upblock", [2580_{16}]; "/upperdothebrew", [$05C4_{16}$]; "/upsilon", [$03C5_{16}$]; "/upsilondieresis", [$03CB_{16}$]; "/upsilondieresistonos", [$03B0_{16}$]; "/upsilonlatin", [$028A_{16}$]; "/upsilontonos", [$03CD_{16}$]; "/uptackbelowcmb", [$031D_{16}$]; "/uptackmod", [$02D4_{16}$]; "/uragurmukhi", [$0A73_{16}$]; "/uring", [$016F_{16}$]; "/ushortcyrillic", [$045E_{16}$]; "/usmallhiragana", [3045_{16}]; "/usmallkatakana", [$30A5_{16}$]; "/usmallkatakanahalfwidth", [$FF69_{16}$]; "/ustraightcyrillic", [$04AF_{16}$]; "/ustraightstrokecyrillic", [$04B1_{16}$]; "/utilde", [0169_{16}]; "/utildeacute", [$1E79_{16}$]; "/utildebelow", [$1E75_{16}$]; "/uubengali", [$098A_{16}$]; "/uudeva", [$090A_{16}$]; "/uugujarati", [$0A8A_{16}$]; "/uugurmukhi", [$0A0A_{16}$]; "/uumatragurmukhi", [$0A42_{16}$]; "/uuvowelsignbengali", [$09C2_{16}$]; "/uuvowelsigndeva", [0942_{16}]; "/uuvowelsigngujarati", [$0AC2_{16}$]; "/uvowelsignbengali", [$09C1_{16}$]; "/uvowelsigndeva", [0941_{16}]; "/uvowelsigngujarati", [$0AC1_{16}$]; "/v", [0076_{16}]; "/vadeva", [0935_{16}]; "/vagujarati", [$0AB5_{16}$]; "/vagurmukhi", [$0A35_{16}$]; "/vakatakana", [$30F7_{16}$]; "/vav", [$05D5_{16}$]; "/vavdagesh", [$FB35_{16}$]; "/vavdagesh65", [$FB35_{16}$]; "/vavdageshhebrew", [$FB35_{16}$]; "/vavhebrew", [$05D5_{16}$]; "/vavholam", [$FB4B_{16}$]; "/vavholamhebrew", [$FB4B_{16}$]; "/vavvavhebrew", [$05F0_{16}$]; "/vavyodhebrew", [$05F1_{16}$]; "/vcircle", [$24E5_{16}$]; "/vdotbelow", [$1E7F_{16}$]; "/vecyrillic", [0432_{16}]; "/veharabic", [$06A4_{16}$]; "/vehfinalarabic", [$FB6B_{16}$]; "/vehinitialarabic", [$FB6C_{16}$]; "/vehmedialarabic", [$FB6D_{16}$]; "/vekatakana", [$30F9_{16}$]; "/venus", [2640_{16}]; "/verticalbar", [$007C_{16}$]; "/verticallineabovecmb", [$030D_{16}$]; "/verticallinebelowcmb", [0329_{16}]; "/verticallinelowmod", [$02CC_{16}$]; "/verticallinemod", [$02C8_{16}$]; "/vewarmenian", [$057E_{16}$]; "/vhook", [$028B_{16}$]; "/vikatakana", [$30F8_{16}$]; "/viramabengali", [$09CD_{16}$]; "/viramadeva", [$094D_{16}$];

"/viramagujarati", [0ACD$_{16}$]; "/visargabengali", [0983$_{16}$]; "/visargadeva", [0903$_{16}$]; "/visargagujarati", [0A83$_{16}$]; "/vmonospace", [FF56$_{16}$]; "/voarmenian", [0578$_{16}$]; "/voicediterationhiragana", [309E$_{16}$]; "/voicediterationkatakana", [30FE$_{16}$]; "/voicedmarkkana", [309B$_{16}$]; "/voicedmarkkanahalfwidth", [FF9E$_{16}$]; "/vokatakana", [30FA$_{16}$]; "/vparen", [24B1$_{16}$]; "/vtilde", [1E7D$_{16}$]; "/vturned", [028C$_{16}$]; "/vuhiragana", [3094$_{16}$]; "/vukatakana", [30F4$_{16}$]; "/w", [0077$_{16}$]; "/wacute", [1E83$_{16}$]; "/waekorean", [3159$_{16}$]; "/wahiragana", [308F$_{16}$]; "/wakatakana", [30EF$_{16}$]; "/wakatakanahalfwidth", [FF9C$_{16}$]; "/wakorean", [3158$_{16}$]; "/wasmallhiragana", [308E$_{16}$]; "/wasmallkatakana", [30EE$_{16}$]; "/wattosquare", [3357$_{16}$]; "/wavedash", [301C$_{16}$]; "/wavyunderscorevertical", [FE34$_{16}$]; "/wawarabic", [0648$_{16}$]; "/wawfinalarabic", [FEEE$_{16}$]; "/wawhamzaabovearabic", [0624$_{16}$]; "/wawhamzaabovefinalarabic", [FE86$_{16}$]; "/wbsquare", [33DD$_{16}$]; "/wcircle", [24E6$_{16}$]; "/wcircumflex", [0175$_{16}$]; "/wdieresis", [1E85$_{16}$]; "/wdotaccent", [1E87$_{16}$]; "/wdotbelow", [1E89$_{16}$]; "/wehiragana", [3091$_{16}$]; "/weierstrass", [2118$_{16}$]; "/wekatakana", [30F1$_{16}$]; "/wekorean", [315E$_{16}$]; "/weokorean", [315D$_{16}$]; "/wgrave", [1E81$_{16}$]; "/whitebullet", [25E6$_{16}$]; "/whitecircle", [25CB$_{16}$]; "/whitecircleinverse", [25D9$_{16}$]; "/whitecornerbracketleft", [300E$_{16}$]; "/whitecornerbracketleftvertical", [FE43$_{16}$]; "/whitecornerbracketright", [300F$_{16}$]; "/whitecornerbracketrightvertical", [FE44$_{16}$]; "/whitediamond", [25C7$_{16}$]; "/whitediamondcontainingblacksmalldiamond", [25C8$_{16}$]; "/whitedownpointingsmalltriangle", [25BF$_{16}$]; "/whitedownpointingtriangle", [25BD$_{16}$]; "/whiteleftpointingsmalltriangle", [25C3$_{16}$]; "/whiteleftpointingtriangle", [25C1$_{16}$]; "/whitelenticularbracketleft", [3016$_{16}$]; "/whitelenticularbracketright", [3017$_{16}$]; "/whiterightpointingsmalltriangle", [25B9$_{16}$]; "/whiterightpointingtriangle", [25B7$_{16}$]; "/whitesmallsquare", [25AB$_{16}$]; "/whitesmilingface", [263A$_{16}$]; "/whitesquare", [25A1$_{16}$]; "/whitestar", [2606$_{16}$]; "/whitetelephone", [260F$_{16}$]; "/whitetortoiseshellbracketleft", [3018$_{16}$]; "/whitetortoiseshellbracketright", [3019$_{16}$]; "/whiteuppointingsmalltriangle", [25B5$_{16}$]; "/whiteuppointingtriangle", [25B3$_{16}$]; "/wihiragana", [3090$_{16}$]; "/wikatakana", [30F0$_{16}$]; "/wikorean", [315F$_{16}$]; "/wmonospace", [FF57$_{16}$]; "/wohiragana", [3092$_{16}$]; "/wokatakana", [30F2$_{16}$]; "/wokatakanahalfwidth", [FF66$_{16}$]; "/won", [20A9$_{16}$]; "/wonmonospace", [FFE6$_{16}$]; "/wowaenthai", [0E27$_{16}$]; "/wparen", [24B2$_{16}$]; "/wring", [1E98$_{16}$]; "/wsuperior", [02B7$_{16}$]; "/wturned", [028D$_{16}$]; "/wynn", [01BF$_{16}$]; "/x", [0078$_{16}$]; "/xabovecmb", [033D$_{16}$]; "/xbopomofo", [3112$_{16}$]; "/xcircle", [24E7$_{16}$]; "/xdieresis", [1E8D$_{16}$]; "/xdotaccent", [1E8B$_{16}$]; "/xeharmenian", [056D$_{16}$]; "/xi", [03BE$_{16}$]; "/xmonospace", [FF58$_{16}$]; "/xparen", [24B3$_{16}$]; "/xsuperior", [02E3$_{16}$]; "/y", [0079$_{16}$]; "/yaadosquare", [334E$_{16}$]; "/yabengali", [09AF$_{16}$]; "/yacute", [00FD$_{16}$]; "/yadeva", [092F$_{16}$]; "/yaekorean", [3152$_{16}$]; "/yagujarati", [0AAF$_{16}$]; "/yagurmukhi", [0A2F$_{16}$]; "/yahiragana", [3084$_{16}$]; "/yakatakana", [30E4$_{16}$]; "/yakatakanahalfwidth", [FF94$_{16}$]; "/yakorean", [3151$_{16}$]; "/yamakkanthai", [0E4E$_{16}$]; "/yasmallhiragana", [3083$_{16}$]; "/yasmallkatakana", [30E3$_{16}$]; "/yasmallkatakanahalfwidth", [FF6C$_{16}$]; "/yatcyrillic", [0463$_{16}$]; "/ycircle", [24E8$_{16}$]; "/ycircumflex", [0177$_{16}$]; "/ydieresis", [00FF$_{16}$]; "/ydotaccent", [1E8F$_{16}$]; "/ydotbelow", [1EF5$_{16}$]; "/yeharabic", [064A$_{16}$]; "/yehbarreearabic", [06D2$_{16}$]; "/yehbarreefinalarabic", [FBAF$_{16}$]; "/yehfinalarabic", [FEF2$_{16}$]; "/yehhamzaabovearabic", [0626$_{16}$]; "/yehhamzaabovefinalarabic", [FE8A$_{16}$]; "/yehhamzaaboveinitialarabic", [FE8B$_{16}$]; "/yehhamzaabovemedialarabic", [FE8C$_{16}$]; "/yehinitialarabic", [FEF3$_{16}$]; "/yehmedialarabic", [FEF4$_{16}$];

"/yehmeeminitialarabic", [$FCDD_{16}$]; "/yehmeemisolatedarabic", [$FC58_{16}$]; "/yehnoonfinalarabic", [$FC94_{16}$]; "/yehthreedotsbelowarabic", [$06D1_{16}$]; "/yekorean", [3156_{16}]; "/yen", [$00A5_{16}$]; "/yenmonospace", [$FFE5_{16}$]; "/yeokorean", [3155_{16}]; "/yeorinhieuhkorean", [3186_{16}]; "/yerahbenyomohebrew", [$05AA_{16}$]; "/yerahbenyomolefthebrew", [$05AA_{16}$]; "/yericyrillic", [$044B_{16}$]; "/yerudieresiscyrillic", [$04F9_{16}$]; "/yesieungkorean", [3181_{16}]; "/yesieungpansioskorean", [3183_{16}]; "/yesieungsioskorean", [3182_{16}]; "/yetivhebrew", [$059A_{16}$]; "/ygrave", [$1EF3_{16}$]; "/yhook", [$01B4_{16}$]; "/yhookabove", [$1EF7_{16}$]; "/yiarmenian", [0575_{16}]; "/yicyrillic", [0457_{16}]; "/yikorean", [3162_{16}]; "/yinyang", [$262F_{16}$]; "/yiwnarmenian", [0582_{16}]; "/ymonospace", [$FF59_{16}$]; "/yod", [$05D9_{16}$]; "/yoddagesh", [$FB39_{16}$]; "/yoddageshhebrew", [$FB39_{16}$]; "/yodhebrew", [$05D9_{16}$]; "/yodyodhebrew", [$05F2_{16}$]; "/yodyodpatahhebrew", [$FB1F_{16}$]; "/yohiragana", [3088_{16}]; "/yoikorean", [3189_{16}]; "/yokatakana", [$30E8_{16}$]; "/yokatakanahalfwidth", [$FF96_{16}$]; "/yokorean", [$315B_{16}$]; "/yosmallhiragana", [3087_{16}]; "/yosmallkatakana", [$30E7_{16}$]; "/yosmallkatakanahalfwidth", [$FF6E_{16}$]; "/yotgreek", [$03F3_{16}$]; "/yoyaekorean", [3188_{16}]; "/yoyakorean", [3187_{16}]; "/yoyakthai", [$0E22_{16}$]; "/yoyingthai", [$0E0D_{16}$]; "/yparen", [$24B4_{16}$]; "/ypogegrammeni", [$037A_{16}$]; "/ypogegrammenigreekcmb", [0345_{16}]; "/yr", [$01A6_{16}$]; "/yring", [$1E99_{16}$]; "/ysuperior", [$02B8_{16}$]; "/ytilde", [$1EF9_{16}$]; "/yturned", [$028E_{16}$]; "/yuhiragana", [3086_{16}]; "/yuikorean", [$318C_{16}$]; "/yukatakana", [$30E6_{16}$]; "/yukatakanahalfwidth", [$FF95_{16}$]; "/yukorean", [3160_{16}]; "/yusbigcyrillic", [$046B_{16}$]; "/yusbigiotifiedcyrillic", [$046D_{16}$]; "/yuslittlecyrillic", [0467_{16}]; "/yuslittleiotifiedcyrillic", [0469_{16}]; "/yusmallhiragana", [3085_{16}]; "/yusmallkatakana", [$30E5_{16}$]; "/yusmallkatakanahalfwidth", [$FF6D_{16}$]; "/yuyekorean", [$318B_{16}$]; "/yuyeokorean", [$318A_{16}$]; "/yyabengali", [$09DF_{16}$]; "/yyadeva", [$095F_{16}$]; "/z", [$007A_{16}$]; "/zaarmenian", [0566_{16}]; "/zacute", [$017A_{16}$]; "/zadeva", [$095B_{16}$]; "/zagurmukhi", [$0A5B_{16}$]; "/zaharabic", [0638_{16}]; "/zahfinalarabic", [$FEC6_{16}$]; "/zahinitialarabic", [$FEC7_{16}$]; "/zahiragana", [3056_{16}]; "/zahmedialarabic", [$FEC8_{16}$]; "/zainarabic", [0632_{16}]; "/zainfinalarabic", [$FEB0_{16}$]; "/zakatakana", [$30B6_{16}$]; "/zaqefgadolhebrew", [0595_{16}]; "/zaqefqatanhebrew", [0594_{16}]; "/zarqahebrew", [0598_{16}]; "/zayin", [$05D6_{16}$]; "/zayindagesh", [$FB36_{16}$]; "/zayindageshhebrew", [$FB36_{16}$]; "/zayinhebrew", [$05D6_{16}$]; "/zbopomofo", [3117_{16}]; "/zcaron", [$017E_{16}$]; "/zcircle", [$24E9_{16}$]; "/zcircumflex", [$1E91_{16}$]; "/zcurl", [0291_{16}]; "/zdot", [$017C_{16}$]; "/zdotaccent", [$017C_{16}$]; "/zdotbelow", [$1E93_{16}$]; "/zecyrillic", [0437_{16}]; "/zedescendercyrillic", [0499_{16}]; "/zedieresiscyrillic", [$04DF_{16}$]; "/zehiragana", [$305C_{16}$]; "/zekatakana", [$30BC_{16}$]; "/zero", [0030_{16}]; "/zeroarabic", [0660_{16}]; "/zerobengali", [$09E6_{16}$]; "/zerodeva", [0966_{16}]; "/zerogujarati", [$0AE6_{16}$]; "/zerogurmukhi", [$0A66_{16}$]; "/zerohackarabic", [0660_{16}]; "/zeroinferior", [2080_{16}]; "/zeromonospace", [$FF10_{16}$]; "/zerooldstyle", [$F730_{16}$]; "/zeropersian", [$06F0_{16}$]; "/zerosuperior", [2070_{16}]; "/zerothai", [$0E50_{16}$]; "/zerowidthjoiner", [$FEFF_{16}$]; "/zerowidthnonjoiner", [$200C_{16}$]; "/zerowidthspace", [$200B_{16}$]; "/zeta", [$03B6_{16}$]; "/zhbopomofo", [3113_{16}]; "/zhearmenian", [$056A_{16}$]; "/zhebrevecyrillic", [$04C2_{16}$]; "/zhecyrillic", [0436_{16}]; "/zhedescendercyrillic", [0497_{16}]; "/zhedieresiscyrillic", [$04DD_{16}$]; "/zihiragana", [3058_{16}]; "/zikatakana", [$30B8_{16}$]; "/zinorhebrew", [$05AE_{16}$]; "/zlinebelow", [$1E95_{16}$]; "/zmonospace", [$FF5A_{16}$]; "/zohiragana", [$305E_{16}$]; "/zokatakana", [$30BE_{16}$]; "/zparen", [$24B5_{16}$]; "/zretroflexhook", [0290_{16}]; "/zstroke", [$01B6_{16}$]; "/zuhiragana", [$305A_{16}$];

"/zukatakana", $[\mathtt{30BA}_{16}]|]$

let $glyphmap\ =\ \mathsf{Array}.to\_list\ glyphmap\_arr$

14 Module Transform
Affine transforms in 2D

This module provides affine transformation on cartesian coordinates, using the standard methods given in [?]. Two patterns of use are supported: building a single matrix from the composition of the desired transformation operations and then using it repeatedly (preferable when one wishes to transform many points); and transforming a point directly from the transformation operations (requires no state at the caller, so simpler).

open Utility

14.1 Types

▷ Individual transformation operations.

type *transform_op* =
  | Scale **of** (*float* × *float*) × *float* × *float* (∗ centre, x scale, y scale. ∗)
  | Rotate **of** (*float* × *float*) × *float* (∗ centre, angle in radians. ∗)
  | Translate **of** *float* × *float* (∗ change in x, change in y. ∗)
  | ShearX **of** (*float* × *float*) × *float* (∗ centre, x shear. ∗)
  | ShearY **of** (*float* × *float*) × *float* (∗ centre, y shear. ∗)

▷ A transform is a list of operations $t_n :: t_{n-1} :: \ldots :: t_2 :: t_1$. which means t_1 followed by t_2 etc.

type *transform* = *transform_op list*

▷ The matrix $\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$ for affine transforms in 2D homogeneous coordinates.

type *transform_matrix* =
  {*a* : *float*; *b* : *float*; *c* : *float*; *d* : *float*; *e* : *float*; *f* : *float*}

14.2 Printers

Debug printers for transformation operations.

```
let string_of_trop = function
  | Scale ((x, y), sx, sy) →
      Printf.sprintf "Scale about (%f, %f) by %f in x and %f in y\n" x y sx sy
  | Rotate ((x, y), a) →
      Printf.sprintf "Rotate by %f about (%f, %f)\n" a x y
  | Translate (dx, dy) →
      Printf.sprintf "Translate by %f, %f\n" dx dy
  | ShearX ((x, y), sx) →
      Printf.sprintf "Shear in X about (%f, %f), proportionality constant
%f\n" x y sx
  | ShearY ((x, y), sy) →
      Printf.sprintf "Shear in Y about (%f, %f), proportionality constant
%f\n" x y sy
```

▷ Same for transforms.

```
let string_of_transform tr =
  fold_left ( ^ ) "" (rev_map string_of_trop tr)
```

14.3 Building and manipulating transforms

▷ The identity transform.

```
let i = ([] : transform)
```

▷ Compose a transformation operation t onto an existing transform ts. We perform a simple optimisation — combining like with like at the head.

```
let compose t = function
  | [] → [t]
  | h :: r →
      match h, t with
      | Translate (dx, dy), Translate (dx', dy') →
          Translate (dx +. dx', dy +. dy') :: r
      | Scale (p, sx, sy), Scale (p', sx', sy') when p = p' →
          Scale (p, sx *. sx', sy *. sy') :: r
      | Rotate (p, a), Rotate (p', a') when p = p' →
          Rotate (p, a +. a') :: r
      | ShearX (p, a), ShearX (p', a') when p = p' →
          ShearX (p, a +. a') :: r
      | ShearY (p, a), ShearY (p', a') when p = p' →
          ShearY (p, a +. a') :: r
      | _ → t :: h :: r
```

▷ Append two transforms. The result is all operations in the second argument followed by all operations in the first.

```
let append = (( @ ) : transform → transform → transform)
```

The identity transformation matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

```
let i_matrix =
  {a = 1.; c = 0.; e = 0.; b = 0.; d = 1.; f = 0.}
```

▷ Compose two matrices. Applying the result is equivalent to applying m then m'.

```
let matrix_compose m' m =
  {a = m'.a *. m.a +. m'.c *. m.b;
   c = m'.a *. m.c +. m'.c *. m.d;
   e = m'.a *. m.e +. m'.c *. m.f +. m'.e;
   b = m'.b *. m.a +. m'.d *. m.b;
   d = m'.b *. m.c +. m'.d *. m.d;
   f = m'.b *. m.e +. m'.d *. m.f +. m'.f}
```

▷ String of matrix

```
let string_of_matrix m =
  Printf.sprintf "%f, %f, %f, %f, %f, %f" m.a m.b m.c m.d m.e m.f
```

▷ Invert a matrix. The exeption:

```
exception NonInvertable
```

▷ And the function.

```
let matrix_invert m =
  let det =
    let divisor = m.a *. m.d -. m.b *. m.c in
      if divisor = 0. then raise NonInvertable else
        match 1. /. divisor with
        | 0. -> raise NonInvertable
        | d -> ( *. ) d
  in
    {a = det m.d;
     b = det ~-.(m.b);
     c = det ~-.(m.c);
     d = det m.a;
     e = det (m.c *. m.f -. m.d *. m.e);
     f = det (m.b *. m.e -. m.a *. m.f)}
```

These functions build matrices for the transformation operations defined above.

▷ Translate by (tx, ty)

```
let mktranslate tx ty =
  {i_matrix with e = tx; f = ty}
```

▷ Scale about an origin (ox, oy) by x factor sx and y factor sy.

```
let mkscale (ox, oy) sx sy =
  let translate = mktranslate ~-.ox ~-.oy
  and translateback = mktranslate ox oy in
    matrix_compose
      translateback
      (matrix_compose {i_matrix with a = sx; d = sy} translate)
```

▷ Rotate about an origin (ox, oy) by angle (in radians) $angle$.

```
let mkrotate (ox, oy) angle =
  let translate = mktranslate ~-.ox ~-.oy
  and translateback = mktranslate ox oy in
    matrix_compose
      translateback
      (matrix_compose
          {i_matrix with a = cos angle; c = ~-. (sin angle);
          b = sin angle; d = cos angle}
          translate)
```

▷ Skew in x about an origin $(ox,\ oy)$ by $factor$.

```
let mkshearx (ox, oy) factor =
  let translate = mktranslate ~-.ox ~-.oy
  and translateback = mktranslate ox oy in
    matrix_compose
      translateback
      (matrix_compose {i_matrix with c = factor} translate)
```

▷ Skew in y about an origin $(ox,\ oy)$ by $factor$.

```
let mksheary (ox, oy) factor =
  let translate = mktranslate ~-.ox ~-.oy
  and translateback = mktranslate ox oy in
    matrix_compose
      translateback
      (matrix_compose {i_matrix with b = factor} translate)
```

▷ Use the preceeding functions to make a matrix from a transformation operation.

```
let matrix_of_op = function
  | Scale (c, sx, sy) → mkscale c sx sy
  | Rotate (c, a) → mkrotate c a
  | Translate (dx, dy) → mktranslate dx dy
  | ShearX (c, a) → mkshearx c a
  | ShearY (c, a) → mksheary c a
```

▷ Transform a point $(x,\ y)$ with a matrix m.

```
let transform_matrix m (x, y) =
  x *. m.a +. y *. m.c +. m.e,
  x *. m.b +. y *. m.d +. m.f
```

▷ Method 1. When transforming many points, it makes sense to calculate the composition of the transformation matrices and then apply this to each of the points.

```
let matrix_of_transform tr =
  let matrices = map matrix_of_op tr in
    fold_left matrix_compose i_matrix matrices
```

▷ Method 2. Transform a point p by a transformation ts. This is faster when we wish to transform a few points. It requires no state at the caller.

```
let transform ts (x, y) =
  let x = ref x and y = ref y in
    iter
      (function
       | Scale ((cx, cy), sx, sy) →
             let x' = !x -. cx and y' = !y -. cy in
               let x'' = x' *. sx and y'' = y' *. sy in
                 x := x'' +. cx;
                 y := y'' +. cy
       | Rotate ((cx, cy), a) →
           let cosine = cos a and sine = sin a in
             let invsine = ~-.sine in
               let x' = !x -. cx and y' = !y -. cy in
                 let x'' = x' *. cosine +. y' *. invsine
                 and y'' = x' *. sine +. y' *. cosine in
                   x := x'' +. cx;
                   y := y'' +. cy
       | Translate (dx, dy) →
           x := !x +. dx; y := !y +. dy
       | ShearX ((cx, cy), a) →
           let x' = !x -. cx and y' = !y -. cy in
             let x'' = x' +. y' *. a and y'' = y' in
               x := x'' +. cx;
               y := y'' +. cy
       | ShearY ((cx, cy), a) →
           let x' = !x -. cx and y' = !y -. cy in
             let x'' = x' and y'' = x' *. a +. y' in
               x := x'' +. cx;
               y := y'' +. cy)
      (rev ts);
    !x, !y
```

14.4 Decomposition and Recomposition

▷ Decompose a matrix to a scale, aspect, rotation, shear and translation.

```
let decompose m =
  let axb = m.a *. m.d -. m.c *. m.b
  and moda = sqrt (m.a *. m.a +. m.b *. m.b)
  and modb = sqrt (m.c *. m.c +. m.d *. m.d)
  and adotb = m.a *. m.c +. m.b *. m.d in
    let scale = axb /. moda in
      let aspect =
        if fabs scale = 0. then 1. else moda /. fabs scale
      and rotation =
        atan2 m.b m.a
      and shear =
        if moda *. modb = 0. then 0. else
          pi /. 2. -. acos (adotb /. (moda *. modb))
```

```
      in
        safe_float scale,
        safe_float aspect,
        safe_float rotation,
        safe_float shear,
        safe_float m.e,
        safe_float m.f
```

▷ Rebuild a matrix from those components.

```
let recompose scale aspect rotation shear tx ty =
  let scale_aspect_shear =
    {i_matrix with
      a = fabs scale *. aspect;
      c = scale *. tan shear;
      d = scale}
  in
    let rotated =
      matrix_compose (mkrotate (0., 0.) rotation) scale_aspect_shear
    in
      matrix_compose (mktranslate tx ty) rotated
```

15 Module Units
Measure and Conversion

open Utility

15.1 Definitions

▷ Units. To add a new unit, extend here and in the graph following.

```
type unit = PdfPoint | Inch | Centimetre | Millimetre | Pixel
```

15.2 Building convertors

Conversions. Must form a connected graph. Each unit is listed at most once as the first of each pair, and at mose once in each associated list.
Create the symmetric closure of the conversions graph, allowing any conversion to be achieved by the following of the appropriate arcs.

```
let conversions dpi =
  let conversions =
    [Millimetre, [Centimetre, 10.];              (* 10mm = 1cm. *)
     PdfPoint, [Inch, 72.];                      (* 72pt = 1in. *)
     Centimetre, [Inch, 2.54];                   (* 2.54cm = 1in. *)
     Pixel, [Inch, dpi]]                         (* dpi pixels = 1in. *)
  in
    let conversions' = ref conversions in
      let insert unit (unit', k) =
        conversions' :=
          match lookup unit !conversions' with
          | None → add unit [unit', k] !conversions'
          | Some cs → replace unit ((unit', k) :: cs) !conversions'
      in
        (* For each item, insert reverse arcs for all in its conversion list. *)
        iter
          (fun (u, cs) →
              iter (fun (u', k) → insert u' (u, 1. /. k)) cs)
          conversions;
      !conversions'
```

To convert, we use a breadth-first search to find the shortest path in the graph, thus minimising the number of conversions. This is not optimal from a floating-point perspective (where certain conversions are worse than others).

Create an index relating types *unit* to index numbers beginning at 0.

```
let index conversions =
  combine (map fst conversions) (ilist 0 (length conversions − 1))
```

Make an array of lists representing the conversions graph, using the index numbers.

```
let conv_array index conversions =
  let adjacency_lists =
    map
      (fun (u, l) →
          lookup_failnull u index,
          map (fun (u, k) → lookup_failnull u index, k) l)
      conversions
  in
    Array.of_list (map snd adjacency_lists)
```

Colours for breadth-first search

```
type colour = White | Grey | Black
```

Perform a breadth-first search starting at u, thus creating a predecessor subgraph *pred*, which is returned.

```
let breadth_first index conv_array u =
  let size = Array.length conv_array in
    let pred = Array.make size −1        (* −1 = null in predecessor array *)
    and colours = Array.make size White                  (* Colour array. *)
    and s = lookup_failnull u index in                        (* Source. *)
      let q = ref (q_enq q_mk s) in      (* Queue for set of grey vertices. *)
        while ¬ (q_null !q) do
          let u = q_hd !q in
            iter
              (fun (i, _) →
                  if colours.(i) = White then
                    begin
                      colours.(i) ← Grey;
                      pred.(i) ← u;
                      q := q_enq !q i
                    end)
              conv_array.(u);
            q := q_deq !q;
            colours.(u) ← Black;
        done;
        pred
```

15.3 Converting

▷ Given source and destination units, we return a conversion function. This follows the appropriate arcs, accumulating the total multiplier. Obviously, the user can provide a third argument to do the computation immediately.

let rec *convert dpi u u′* =
  let *conversions* = *conversions dpi* **in**
    let *index* = *index conversions* **in**
      let *conv_array* = *conv_array index conversions* **in**
        let *pred* = *breadth_first index conv_array u′* **in**
          let *i* = *ref* (*lookup_failnull u index*)
          and *m* = *ref* 1. **in**
            while ¬ (*pred*.(!*i*) = −1) **do**
              let *i′* = *pred*.(!*i*) **in**
                m ∗. = *lookup_failnull* !*i* *conv_array*.(*i′*);
                i := *i′*
            done;
            fun *x* → *x* ∗. !*m*

16 Module Paper
Standard Media Sizes

▷ Type for paper sizes — unit, width, height.

```
type papersize = Paper of (Units.unit × float × float)

let make u w h = Paper (u, w, h)

let unit (Paper (u, _, _)) = u
let width (Paper (_, w, _)) = w
let height (Paper (_, _, h)) = h
```

▷ Make a paper size landscape

```
let landscape (Paper (u, w, h)) = Paper (u, h, w)
```

▷ European 'A' sizes.

```
let a0 = Paper (Units.Millimetre, 841., 1189.)
and a1 = Paper (Units.Millimetre, 594., 841.)
and a2 = Paper (Units.Millimetre, 420., 594.)
and a3 = Paper (Units.Millimetre, 297., 420.)
and a4 = Paper (Units.Millimetre, 210., 297.)
and a5 = Paper (Units.Millimetre, 148., 210.)
and a6 = Paper (Units.Millimetre, 105., 148.)
and a7 = Paper (Units.Millimetre, 74., 105.)
and a8 = Paper (Units.Millimetre, 52., 74.)
```

▷ US Imperial sizes.

```
let usletter = Paper (Units.Inch, 8.5, 11.)
let uslegal = Paper (Units.Inch, 8.5, 14.)
```

PDF Bookmarks

```
open Utility

type target = int (* Just page number for now *)

type bookmark =
  {level : int;
   text : string;
   target : target;
   isopen : bool}
```

```
let remove_bookmarks pdf =
  match Pdf.lookup_direct pdf "/Root" pdf.Pdf.trailerdict with
  | None → raise (Pdf.PDFError "remove_boomarks:  Bad PDF: no root")
  | Some catalog →
      let catalog' = Pdf.remove_dict_entry catalog "/Outlines" in
        let newcatalognum = Pdf.addobj pdf catalog' in
          {pdf with
            Pdf.root = newcatalognum;
            Pdf.trailerdict =
              Pdf.add_dict_entry
                pdf.Pdf.trailerdict "/Root" (Pdf.Indirect newcatalognum)}

type ntree =
  Br of int × Pdf.pdfobject × ntree list

let rec print_ntree (Br (i, _, l)) =
  Printf.printf "%i (" i;
  iter print_ntree l;
  flprint ")"

let fresh source pdf =
  incr source; Pdf.maxobjnum pdf + !source
```

Flatten a tree and produce a root object for it. Return a list of (num, pdfobject) pairs with the root first.

```
let flatten_tree source pdf = function
  | [] →
      let n = fresh source pdf in
        [(n, Pdf.Dictionary [])], n
  | tree →
      let root_objnum = fresh source pdf in
      (* Add /Parent links to root *)
      let tree =
        let add_root_parent (Br (i, dict, children)) =
          Br
            (i,
             Pdf.add_dict_entry dict "/Parent" (Pdf.Indirect root_objnum),
             children)
        in
          map add_root_parent tree
      in
        let rec really_flatten = function
          Br (i, pdfobject, children) →
            (i, pdfobject) :: flatten (map really_flatten children)
        in
          let all_but_top = flatten (map really_flatten tree)
          and top, topnum =
            (* Make top level from objects at first level of tree *)
            match extremes tree with
              Br (first, _, _), Br (last, _, _) →
                (root_objnum, Pdf.Dictionary
                  [("/First", Pdf.Indirect first); ("/Last", Pdf.Indirect last)]),
```

```
        root_objnum
    in
      top :: all_but_top, topnum
```

Add /Count entries to an ntree

```
let add_counts tree = tree
```

Add /Parent entries to an ntree

```
let rec add_parent parent (Br (i, obj, children)) =
  let obj' =
    match parent with
    | None → obj
    | Some parent_num →
        Pdf.add_dict_entry obj "/Parent" (Pdf.Indirect parent_num)
  in
    Br (i, obj', map (add_parent (Some i)) children)
```

Add /First and /Last entries to an ntree

```
let rec add_firstlast (Br (i, obj, children)) =
  match children with
  | [] → (Br (i, obj, children))
  | c →
      match extremes c with
        Br (i', _, _), Br (i'', _, _) →
          let obj = Pdf.add_dict_entry obj "/First" (Pdf.Indirect i') in
            let obj = Pdf.add_dict_entry obj "/Last" (Pdf.Indirect i'') in
              (Br (i, obj, map add_firstlast children))
```

Add /Next and /Prev entries to an ntree

```
let rec add_next (Br (i, obj, children)) =
  match children with
  | [] → Br (i, obj, children)
  | [_] → Br (i, obj, map add_next children)
  | c :: cs →
      let numbers = map (fun (Br (i, _, _)) → i) cs in
        let children' =
          (map2
             (fun (Br (i, obj, children)) nextnum →
                Br (i,
                    Pdf.add_dict_entry obj "/Next" (Pdf.Indirect nextnum),
                    children))
             (all_but_last (c :: cs))
             numbers)
          @ [last cs]
        in
          Br (i, obj, map add_next children')
```

```
let rec add_prev (Br (i, obj, children)) =
  match children with
  | [] → Br (i, obj, children)
  | [_] → Br (i, obj, map add_prev children)
  | c :: cs →
      let numbers = map (fun (Br (i, _, _)) → i) (all_but_last (c :: cs)) in
        let children' =
          c ::
            map2
              (fun (Br (i, obj, children)) prevnum →
                 Br (i,
                      Pdf.add_dict_entry obj "/Prev" (Pdf.Indirect prevnum),
                      children))
              cs
              numbers
        in
          Br (i, obj, map add_prev children')
```

Find a page indirect from the page tree of a document, given a page number.

```
let page_object_number pdf destpage =
  try
    Pdf.Indirect (select destpage (Pdf.page_reference_numbers pdf))
  with
    (* The page might not exist in the output *)
    Invalid_argument "select" → dpr "3b"; Pdf.Null
```

Make a node from a given title, destination page number in a given PDF ond open flag.

```
let node_of_line pdf title destpage isopen =
  if destpage > 0 then (* destpage = 0 means no destination. *)
    Pdf.Dictionary
      [("/Title", Pdf.String title);
       ("/Dest", Pdf.Array
          [page_object_number pdf destpage; Pdf.Name "/Fit"])]
  else
    Pdf.Dictionary [("/Title", Pdf.String title)]
```

Make an ntree list from a list of parsed bookmark lines.

```
let rec make_outline_ntree source pdf = function
  | [] → []
  | (n, title, destpage, isopen) :: t →
      let lower, rest = cleavewhile (fun (n', _, _, _) → n' > n) t in
        let node = node_of_line pdf title destpage isopen in
          Br (fresh source pdf, node, make_outline_ntree source pdf lower)
            ::make_outline_ntree source pdf rest
```

Temporary routine

```
let tuple_of_record r =
  r.level, r.text, r.target, r.isopen
```

Add bookmarks.

```
let add_bookmarks parsed pdf =
  let parsed = map tuple_of_record parsed in
  if parsed = [] then remove_bookmarks pdf else
  begin
    let source = ref 0 in
    let tree = make_outline_ntree source pdf parsed in
      (* Build the (object number, bookmark tree object) pairs. *)
      let pairs, tree_root_num =
        let tree =
          map add_firstlast tree
        in
          let tree =
            match add_next (add_prev (Br (0, Pdf.Null, tree))) with
              Br (_, _, children) -> children
          in
            flatten_tree source pdf (add_counts (map (add_parent None) tree))
      in
        (* Add the objects to the pdf *)
        iter
          (function x -> ignore (Pdf.addobj_given_num pdf x))
          pairs;
          (* Replace the /Outlines entry in the document catalog. *)
          match Pdf.lookup_direct pdf "/Root" pdf.Pdf.trailerdict with
          | None -> raise (Pdf.PDFError "Bad PDF: no root")
          | Some catalog ->
              let catalog' =
                Pdf.add_dict_entry catalog "/Outlines" (Pdf.Indirect tree_root_num)
              in
                let newcatalognum = Pdf.addobj pdf catalog' in
                  {pdf with
                    Pdf.root = newcatalognum;
                    Pdf.trailerdict =
                      Pdf.add_dict_entry
                        pdf.Pdf.trailerdict "/Root" (Pdf.Indirect newcatalognum)}
  end
```

Read and Write Annotations

```
open Utility
```

Annotation Border Styles

```
type style = NoStyle | Solid | Dashed | Beveled | Inset | UnderlineStyle
```

```
type border =
  {width : float;
   vradius : float;
   hradius : float;
   style : style;
   dasharray : int array}
```

```
type subtype =
  | Text
  | Link
  | FreeText
  | Line
  | Square
  | Circle
  | Polygon
  | PolyLine
  | Highlight
  | Underline
  | Squiggly
  | StrikeOut
  | Stamp
  | Caret
  | Ink
  | Popup of annotation
  | FileAttachment
  | Sound
  | Movie
  | Widget
  | Screen
  | PrinterMark
  | TrapNet
  | Watermark
  | ThreeDee
  | Unknown
```

Main type. 'rest' contains the raw annotation dictionary with the exception of the entries corresponding to the other items in the record.

```
and annotation =
  {subtype : subtype;
   contents : string option;
   rectangle : float × float × float × float;
   border : border;
   rest : Pdf.pdfobject}
```

Read a single annotation

```
let rec read_annotation pdf annot =
  flprint (Pdfwrite.string_of_pdf (Pdf.direct pdf annot));
  let subtype =
    match Pdf.lookup_direct pdf "/Subtype" annot with
    | Some (Pdf.Name "/Text") → Text
    | Some (Pdf.Name "/Popup") →
        (* Look up /Parent. If exists, include it *)
        begin match Pdf.direct pdf annot with
        | Pdf.Dictionary d →
            begin match lookup "/Parent" d with
            | Some (Pdf.Indirect i) → Popup (read_annotation pdf (Pdf.Indirect i))
            | _ → Unknown
```

```
          end
      | _ → failwith "read_annotation failed"
      end
  | _ → Unknown
and contents =
  match Pdf.lookup_direct pdf "/Contents" annot with
  | Some (Pdf.String s) → Some s
  | _ → None
and rectangle =
  Pdf.parse_rectangle (Pdf.lookup_fail "No /rect in annot" pdf "/Rect" annot)
and border =
  match Pdf.lookup_direct pdf "/BS" annot with
  | Some bsdict →
      let width =
        match Pdf.lookup_direct pdf "/W" bsdict with
        | Some x → Pdf.getnum x
        | _ → 1.
      and style =
        match Pdf.lookup_direct pdf "/S" bsdict with
        | Some (Pdf.Name "/S") → Solid
        | Some (Pdf.Name "/D") → Dashed
        | Some (Pdf.Name "/B") → Beveled
        | Some (Pdf.Name "/I") → Inset
        | Some (Pdf.Name "/U") → UnderlineStyle
        | _ → NoStyle
      and dasharray =
        match Pdf.lookup_direct pdf "/D" bsdict with
        | Some (Pdf.Array dash) →
            Array.of_list (map int_of_float (map Pdf.getnum (map (Pdf.direct pdf) dash)))
        | _ → [||]
      in
        {width = width;
         vradius = 0.;
         hradius = 0.;
         style = style;
         dasharray = dasharray}
  | None →
      match Pdf.lookup_direct pdf "/Border" annot with
      | Some (Pdf.Array [h; v; w]) →
          {width = Pdf.getnum (Pdf.direct pdf w);
           vradius = Pdf.getnum (Pdf.direct pdf v);
           hradius = Pdf.getnum (Pdf.direct pdf h);
           style = NoStyle;
           dasharray = [||]}
      | Some (Pdf.Array [h; v; w; Pdf.Array dash]) →
          {width = Pdf.getnum (Pdf.direct pdf w);
           vradius = Pdf.getnum (Pdf.direct pdf v);
           hradius = Pdf.getnum (Pdf.direct pdf h);
           style = NoStyle;
           dasharray = Array.of_list (map int_of_float (map Pdf.getnum (map (Pdf.direct pdf) dash)))}
```

```
        | _ →
            {width = 1.;
             vradius = 0.;
             hradius = 0.;
             style = NoStyle;
             dasharray = [||]}
  and rest =
    match Pdf.direct pdf annot with
    | Pdf.Dictionary entries →
        Pdf.Dictionary
          (lose (fun (k, _) → mem k ["/Subtype"; "/Contents"; "/Rect"; "/Border"]) entries)
    | _ → failwith "Bad annotation dictionary"
  in
    {subtype = subtype;
     contents = contents;
     rectangle = rectangle;
     border = border;
     rest = rest}

let get_popup_parent pdf annotation =
  match Pdf.direct pdf annotation with
  | Pdf.Dictionary d →
      begin match lookup "/Parent" d with
      | Some (Pdf.Indirect i) → Some i
      | _ → None
      end
  | _ → failwith "get_popup_parent:  this is not a dictionary"
```

Read the annotations from a page.

```
let annotations_of_page pdf page =
  match Pdf.lookup_direct pdf "/Annots" page.Pdfdoc.rest with
  | Some (Pdf.Array annotations) →
      (* We don't read annotations which are parents of Popup annotations
- they will be caught anyway. This seems to be the right thing to do, but will
need more advice. *)
      let popup_parents =
        option_map (get_popup_parent pdf) annotations
      in
        map
          (read_annotation pdf)
          (lose (function Pdf.Indirect i → mem i popup_parents | _ →
 false) annotations)
  | _ → []
```

Add an annotation to a page

17 Module Pdfgraphics
Structured Graphics

open Utility
open Pdf
open Pdfread
open Pdfpages

FIXME: Replace failwiths with proper error handling

type $fpoint$ = *float* × *float*

type $winding\_rule$ = EvenOdd | NonZero

type $segment$ =
  | Straight **of** $fpoint$ × $fpoint$
  | Bezier **of** $fpoint$ × $fpoint$ × $fpoint$ × $fpoint$

Each segment list may be marked as a hole or not.

type $hole$ = Hole | Not_hole

A $subpath$ is either closed or open.

type $closure$ = Closed | Open

A $subpath$ is the pair of a hole and a list of segments.

type $subpath$ = $hole$ × $closure$ × $segment$ *list*

A path is made from a number of subpaths.

type $path$ = $winding\_rule$ × $subpath$ *list*

type $tiling$ = Tiling

type $function\_shading$ =
  {$funshading\_domain$: *float* × *float* × *float* × *float*;
  $funshading\_matrix$: Transform.$transform\_matrix$;
  $funshading\_function$: Pdffun.pdf_fun}

type $radial\_shading$ =
  {$radialshading\_coords$: *float* × *float* × *float* × *float* × *float* × *float*;
  $radialshading\_domain$: *float* × *float*;
  $radialshading\_function$: Pdffun.pdf_fun *list*;
  $radialshading\_extend$: *bool* × *bool*}

```
type axial_shading =
  {axialshading_coords : float × float × float × float;
   axialshading_domain : float × float;
   axialshading_function : Pdffun.pdf_fun list;
   axialshading_extend : bool × bool}

type shading_kind =
 | FunctionShading of function_shading
 | AxialShading of axial_shading
 | RadialShading of radial_shading
 | FreeFormGouraudShading
 | LatticeFormGouraudShading
 | CoonsPatchMesh
 | TensorProductPatchMesh

type shading =
 {shading_colourspace : Pdf.pdfobject;
  shading_background : Pdf.pdfobject option;
  shading_bbox : Pdf.pdfobject option;
  shading_antialias : bool;
  shading_matrix : Transform.transform_matrix;
  shading_extgstate : Pdf.pdfobject;
  shading : shading_kind}

type pattern =
  | ColouredTilingPattern of tiling
  | UncolouredTilingPattern of tiling
  | ShadingPattern of shading

type colvals =
  | Floats of float list
  | Named of (string × float list)
  | Pattern of pattern

let rec string_of_colvals = function
  | Floats fs →
      "Floats " ^ fold_left ( ^ ) "" (map (function x → string_of_float x ^ "
") fs)
  | Named (n, fs) →
      "Named " ^ n ^ " " ^ string_of_colvals (Floats fs)
  | Pattern p →
      "Pattern"

type objectclass =
  | PathObject
  | TextObject
  | ClippingPathObject
  | PageDescriptionLevel
  | ShadingObject
  | InlineImageObject
  | ExternalObject
```

```
let string_of_objectclass = function
  | PathObject -> "PathObject"
  | TextObject -> "TextObject"
  | ClippingPathObject -> "ClippingPathObject"
  | PageDescriptionLevel -> "PageDescriptionLevel"
  | ShadingObject -> "ShadingObject"
  | InlineImageObject -> "InlineImageObject"
  | ExternalObject -> "ExternalObject"

type transparency_attributes =
  {fill_transparency : float;
   line_transparency : float}

type path_attributes =
  {path_transform : Transform.transform_matrix;
   path_fill : (Pdf.pdfobject × colvals) option;
   path_line : (Pdf.pdfobject × colvals) option;
   path_linewidth : float;
   path_joinstyle : int;
   path_capstyle : int;
   path_dash : float list × float;
   path_mitrelimit : float;
   path_transparency : transparency_attributes;
   path_intent : string}

type text_attributes =
  {textmode : int}

type textblock_attributes =
  {textblock_transform : Transform.transform_matrix}

type textblock =
  text_attributes × Pdfpages.operator

type softmask_subtype =
  Alpha | Luminosity

type transparency_group =
  {tr_group_colourspace : Pdf.pdfobject option;
   isolated : bool;
   knockout : bool;
   tr_graphic : graphic}

and softmask =
  {softmask_subtype : softmask_subtype;
   transparency_group : transparency_group;
   softmask_bbox : float × float × float × float;
   backdrop : float list option;
   softmask_transfer : Pdffun.pdf_fun option}

and image_attributes =
  {image_transform : Transform.transform_matrix;
   image_transparency : float; (* The /ca value *)
   image_softmask : softmask option}
```

```
and fontname = string × Pdf.pdfobject                    (* Name, font *)
```

The main type for a graphic. It must be kept paired with the PDF it comes from, since it will reference objects (fonts, images etc) in that PDF.

```
and graphic_elt =
  | Path of (path × path_attributes)
  | Text of textblock list × textblock_attributes
  | MCPoint of string
  | MCPointProperties of string × Pdf.pdfobject
  | MCSection of string × graphic_elt list
  | MCSectionProperties of string × Pdf.pdfobject × graphic_elt list
  | Image of image_attributes × int (* object number *)
  | GraphicInlineImage of Pdf.pdfobject × bytestream × Transform.transform_matrix
  | Clip of path × graphic_elt list
  | Shading of path option × shading × Transform.transform_matrix

and graphic =
  {elements : graphic_elt list; (* Page content *)
   fonts : fontname list; (* Fonts *)
   resources : Pdf.pdfobject} (* Anything else in /Resources *)
```

Calculate the bounding box (xmin, xmax, ymin, ymax) of a graphic.

```
let bbox_of_segment = function
  | Straight ((x1, y1), (x2, y2)) →
      fmin x1 x2, fmax x1 x2, fmin y1 y2, fmax y1 y2
  | Bezier ((x1, y1), (x2, y2), (x3, y3), (x4, y4)) →
      fmin (fmin x1 x2) (fmin x3 x4), fmax (fmax x1 x2) (fmax x3 x4),
      fmin (fmin y1 y2) (fmin y3 y4), fmax (fmax y1 y2) (fmax y3 y4)

let bbox_of_path (_, subpaths) =
  let segments =
    flatten (map (function (_, _, l) → l) subpaths)
  in
    fold_left
      box_union_float
      (max_float, min_float, max_float, min_float)
      (map bbox_of_segment segments)

let rec bbox_of_graphic_inner (xmin, xmax, ymin, ymax) = function
  | [] → xmin, xmax, ymin, ymax
  | (Path (p, _) | Clip (p, _)) :: t →
      bbox_of_graphic_inner
        (box_union_float (xmin, xmax, ymin, ymax) (bbox_of_path p)) t
  | h :: t → bbox_of_graphic_inner (xmin, xmax, ymin, ymax) t

let bbox_of_graphic graphic =
  bbox_of_graphic_inner
    (max_float, min_float, max_float, min_float)
    graphic.elements
```

For debug purposes, build a string of a graphic.

```
let string_of_segment = function
  | Straight ((ax, ay), (bx, by)) →
      Printf.sprintf "Straight line: (%f, %f) --> (%f, %f)\n" ax ay bx by
  | Bezier ((ax, ay), (bx, by), (cx, cy), (dx, dy)) →
      Printf.sprintf
        "Bezier curve: (%f, %f) --> (%f, %f) --> (%f, %f) --> (%f,
%f)\n"
        ax ay bx by cx cy dx dy

let string_of_subpath (h, o , segments) =
  Printf.sprintf "Hole: %b, Open: %b, segments:%s\n"
    (h = Hole) (o = Open) (fold_left ( ^ ) "" (map string_of_segment segments))

let string_of_path (windingrule, subpaths) =
  Printf.sprintf "%s %s"
    (match windingrule with
    | EvenOdd → "Even-odd\n"
    | NonZero → "Non-zero\n")
    (fold_left ( ^ ) "" (map string_of_subpath subpaths))

let string_of_textblock (st, op) =
  "TEXTPIECE: " ^ Pdfpages.string_of_op op ^ "\n"

let string_of_font (f, i) = f ^ " " ^ Pdfwrite.string_of_pdf i ^ "\n"

let rec string_of_graphic_elt = function
  | MCSection (n, g) →
      Printf.sprintf "Marked content section %s...\n" n ^ "BEGIN\n" ^
      (fold_left ( ^ ) "" (map string_of_graphic_elt g))
      ^ "\nEND Marked content section\n"
  | MCSectionProperties (n, d, g) →
      Printf.sprintf "Marked content section %s with properties %s...\n" n
      (Pdfwrite.string_of_pdf d)
      ^ "BEGIN\n" ^
      (fold_left ( ^ ) "" (map string_of_graphic_elt g))
      ^ "\nEND Marked content section\n"
  | MCPoint n →
      Printf.sprintf "Marked content point %s...\n" n
  | MCPointProperties (n, d) →
      Printf.sprintf "Marked content point %s with properties %s...\n" n (Pdfwrite.string_of_pdf d)
  | Path (p, _) → Printf.sprintf "Path: %s\n" (string_of_path p)
  | Text (ts, attr) →
      "-----BEGIN TEXT - fonts:\n" ^
      fold_left ( ^ ) "" (map string_of_textblock ts) ^
      "-----END TEXT"
  | Image (tr, x) →
      "Image " ^ string_of_int x ^ "\n"
  | GraphicInlineImage _ →
      "Inline image\n"
  | Clip (p, g) →
      "Clipview: path = " ^ string_of_path p ^ "\ngraphic is " ^
      fold_left ( ^ ) "" (map string_of_graphic_elt g)
  | Shading (clip, shading, tr) →
```

```
      "Shading"
and string_of_graphic g =
  "Elements:\n" ^
  fold_left ( ^ ) "" (map string_of_graphic_elt g.elements) ^
  "Fonts:\n" ^
  fold_left ( ^ ) ""
    (map
      (fun (name, obj) → name ^ " " ^ Pdfwrite.string_of_pdf obj)
      g.fonts) ^
  "Resources:\n" ^
  Pdfwrite.string_of_pdf g.resources

type state =
  {mutable objectclass : objectclass;  (* Not strictly part of the state, but fits here. *)
    mutable clip : path option;  (* Ditto - stores a clipping path which is to be invoked on the next path operation. *)
    mutable intent : string;
    mutable fill : colvals;
    mutable linewidth : float;
    mutable line : colvals;
    mutable mitrelimit : float;
    mutable joinstyle : int;
    mutable capstyle : int;
    mutable colourspace_stroke : Pdf.pdfobject;
    mutable colourspace_nonstroke : Pdf.pdfobject;
    mutable dash : float list × float;
    mutable flatness : int;
    mutable transform : Transform.transform_matrix;
    mutable extra_transform : Transform.transform_matrix;
    mutable text_transform : Transform.transform_matrix;
    mutable text_line_transform : Transform.transform_matrix;
    mutable opacity_stroke : float;
    mutable opacity_nonstroke : float;
    mutable character_spacing : float;
    mutable word_spacing : float;
    mutable scale : float;
    mutable leading : float;
    mutable font_and_size : (string × float) option;
    mutable font_render : int;
    mutable font_rise : float;
    mutable blendmode : int;
    mutable softmask : softmask option;
    mutable in_xobject : int;
    mutable opdo_matrix : Transform.transform_matrix}

let default_state () =
  {objectclass = PageDescriptionLevel;
    clip = None;
    intent = "/RelativeColorimetric";
    fill = Floats [1.];
```

```
   linewidth = 1.;
   line = Floats [1.];
   mitrelimit = 10.;
   joinstyle = 0;
   capstyle = 0;
   colourspace_stroke = Name "/DeviceGray";
   colourspace_nonstroke = Name "/DeviceGray";
   dash = [], 0.;
   flatness = 0;
   transform = Transform.i_matrix;
   extra_transform = Transform.i_matrix;
   text_transform = Transform.i_matrix;
   text_line_transform = Transform.i_matrix;
   opacity_stroke = 1.;
   opacity_nonstroke = 1.;
   character_spacing = 0.;
   word_spacing = 0.;
   scale = 100.;
   leading = 0.;
   font_and_size = None;                         (* No initial value. *)
   font_render = 0;
   font_rise = 0.;
   blendmode = 1;
   softmask = None;
   in_xobject = 0;
   opdo_matrix = Transform.i_matrix}

let state = ref (default_state ())

let string_of_state s =
    "Stroke Colourspace:  " ^ Pdfwrite.string_of_pdf s.colourspace_stroke ^ "\n" ^
  "Nonstroke Colourspace:  " ^ Pdfwrite.string_of_pdf s.colourspace_nonstroke ^ "\n" ^
  "Stroke colours:  " ^ string_of_colvals s.line ^ "\n" ^
  "NonStroke colours:  " ^ string_of_colvals s.fill ^ "\n"

let path_attributes_fill_and_stroke () =
  {path_transform = !state.transform;
   path_fill = Some (!state.colourspace_nonstroke, !state.fill);
   path_line = Some (!state.colourspace_stroke, !state.line);
   path_linewidth = !state.linewidth;
   path_joinstyle = !state.joinstyle;
   path_capstyle = !state.capstyle;
   path_dash = !state.dash;
   path_mitrelimit = !state.mitrelimit;
   path_transparency =
     {fill_transparency = !state.opacity_nonstroke;
      line_transparency = !state.opacity_stroke};
   path_intent = !state.intent}

let path_attributes_fill () =
  {path_transform = !state.transform;
   path_fill = Some (!state.colourspace_nonstroke, !state.fill);
```

```
    path_line = None;
    path_linewidth = !state.linewidth;
    path_joinstyle = !state.joinstyle;
    path_capstyle = !state.capstyle;
    path_dash = !state.dash;
    path_mitrelimit = !state.mitrelimit;
    path_transparency =
      {fill_transparency = !state.opacity_nonstroke;
       line_transparency = 1.};
    path_intent = !state.intent}

let path_attributes_stroke () =
  {path_transform = !state.transform;
   path_fill = None;
   path_line = Some (!state.colourspace_stroke, !state.line);
   path_linewidth = !state.linewidth;
   path_joinstyle = !state.joinstyle;
   path_capstyle = !state.capstyle;
   path_dash = !state.dash;
   path_mitrelimit = !state.mitrelimit;
   path_transparency =
     {fill_transparency = 1.;
      line_transparency = !state.opacity_stroke};
   path_intent = !state.intent}

let textstate () =
  {textmode = 0}

let nonzero = EvenOdd

let rec initial_colour pdf resources = function
  | Name "/DeviceGray"
  | Array (Name "/CalGray"::_) →
      Floats [0.]
  | Name "/DeviceRGB"
  | Array (Name "/CalRGB"::_) →
      Floats [0.; 0.; 0.]
  | Name "/DeviceCMYK" →
      Floats [0.; 0.; 0.; 1.]
  | Name "/Pattern" →
      Floats [0.]
  | Array elts →
      begin match elts with
        | [Name "/ICCBased"; iccstream] →
            begin match lookup_direct pdf "/Alternate" iccstream with
            | Some space → initial_colour pdf resources space
            | None →
                begin match lookup_direct pdf "/N" iccstream with
                | Some (Integer 1) → Floats [0.]
                | Some (Integer 3) → Floats [0.; 0.; 0.]
                | Some (Integer 4) → Floats [0.; 0.; 0.; 0.]
                | _ → raise (PDFSemanticError "Bad ICCBased Alternate")
```

```
            end
          end
        | Name "/DeviceN"::_ :: alternate :: _
        | [Name "/Separation"; _; alternate; _] →
            initial_colour pdf resources alternate
        | [Name "/Pattern"; alternate] →
            initial_colour pdf resources alternate
        | _ → raise (PDFSemanticError "Unknown colourspace A")
      end
  | Indirect _ as indirect →
      initial_colour pdf resources (direct pdf indirect)
  | _ → raise (PDFSemanticError "Unknown colourspace B")
```

PartialPath (sp, cp, p, s) is starting point sp, current point cp the partial segment list p, subpath s and graphic g.

```
type partial =
  | NoPartial
  | PartialText of textblock list
  | PartialPath of fpoint × fpoint × segment list × subpath list
```

g is a *group_transparency* xobject

```
let rec read_transparency_group pdf g =
  let group =
    match Pdf.lookup_direct pdf "/Group" g with
    | Some gr → gr
    | None → failwith "no /Group found"
  in
    let colourspace =
      Pdf.lookup_direct pdf "/CS" group
    and isolated =
      match Pdf.lookup_direct pdf "/I" group with
      | Some (Pdf.Boolean b) → b
      | _ → false
    and knockout =
      match Pdf.lookup_direct pdf "/K" group with
      | Some (Pdf.Boolean b) → b
      | _ → false
    and graphic =
      let fakepage =
        let resources =
          match Pdf.lookup_direct pdf "/Resources" g with
          | Some (Pdf.Dictionary d) → Pdf.Dictionary d
          | _ → Pdf.Dictionary []
        and contents =
          [g]
        in
          {Pdfdoc.content = contents;
           Pdfdoc.mediabox = Pdf.Null;
           Pdfdoc.resources = resources;
           Pdfdoc.rotate = Pdfdoc.Rotate0;
```

```
          Pdfdoc.rest = Pdf.Dictionary []}
      in
        graphic_of_page pdf fakepage
    and a, b, c, d =
      Pdf.parse_rectangle (Pdf.lookup_fail "no bbox" pdf "/BBox" g)
    in
      {tr_group_colourspace = colourspace;
       isolated = isolated;
       knockout = knockout;
       tr_graphic = graphic}, a, b, c, d

and read_soft_mask pdf mask =
  match
    match Pdf.lookup_direct pdf "/S" mask with
    | Some (Pdf.Name "/Alpha") → Some Alpha
    | Some (Pdf.Name "/Luminosity") → Some Luminosity
    | _ → None
  with
  | None → None
  | Some subtype →
      let transparency_group, a, b, c, d =
        match Pdf.lookup_direct pdf "/G" mask with
        | Some g → read_transparency_group pdf g
        | None → failwith "transparency group not found in soft mask"
      and backdrop =
        match Pdf.lookup_direct pdf "/BC" mask with
        | Some (Pdf.Array nums) → Some (map getnum nums)
        | _ → None
      and transfer =
        match Pdf.lookup_direct pdf "/TR" mask with
        | Some (Pdf.Dictionary d) →
            Some (Pdffun.parse_function pdf (Pdf.Dictionary d))
        | _ → None
      in
        Some
          {softmask_subtype = subtype;
           transparency_group = transparency_group;
           backdrop = backdrop;
           softmask_transfer = transfer;
           softmask_bbox = (a, b, c, d)}

and update_graphics_state_from_dict pdf resources gdict =
  begin match lookup_direct pdf "/SMask" gdict with
  | Some softmask → !state.softmask ← read_soft_mask pdf softmask
  | None → ()
  end;
  begin match lookup_direct pdf "/CA" gdict with
  | Some (Real o) → !state.opacity_stroke ← o
  | _ → ()
  end;
  begin match lookup_direct pdf "/ca" gdict with
```

```
| Some (Real o) → !state.opacity_nonstroke ← o
| _ → ()
end;
begin match lookup_direct pdf "/BM" gdict with
| Some (Name n)
| Some (Array (Name n :: _)) →
    !state.blendmode ← 0 (* FIXME: Do properly *)
| _ → ()
end;
begin match lookup_direct pdf "/LW" gdict with
  | Some (Integer width) →
      !state.linewidth ← float width
  | Some (Real width) →
      !state.linewidth ← width
  | _ → ()
end;
begin match lookup_direct pdf "/LC" gdict with
  | Some (Integer style) →
      !state.capstyle ← style
  | _ → ()
end;
begin match lookup_direct pdf "/LC" gdict with
  | Some (Integer join) →
      !state.joinstyle ← join
  | _ → ()
end;
begin match lookup_direct pdf "/ML" gdict with
  | Some (Integer limit) →
      !state.mitrelimit ← float limit
  | Some (Real limit) →
      !state.mitrelimit ← limit
  | _ → ()
end;
begin match lookup_direct pdf "/D" gdict with
| Some (Array [Array dashes; phase]) →
    let dashnums, phase =
      map
        (function
         | (Integer n) → float n
         | (Real n) → n
         | _ → raise (PDFSemanticError "Malformed dash."))
        dashes,
      match phase with
      | Integer phase → float phase
      | Real phase → phase
      | _ → raise (PDFSemanticError "Malformed dash phase.")
    in
      !state.dash ← dashnums, phase
| _ → ()
end
```

```
and statestack : state list ref = ref []

and copystate () =
  {!state with fill = !state.fill}

and push_statestack () =
  statestack =| copystate ()

and pop_statestack () =
  match !statestack with
  | [] → raise (PDFSemanticError "Unbalanced q/Q Ops")
  | h :: t → statestack := t; state := h

and read_tiling_pattern _ =
  ColouredTilingPattern Tiling

and read_function_shading pdf shading =
  let domain =
    match Pdf.lookup_direct pdf "/Domain" shading with
    | Some (Pdf.Array [a; b; c; d]) → getnum a, getnum b, getnum c, getnum d
    | _ → 0., 1., 0., 1.
  and matrix =
    Pdf.parse_matrix pdf "/Matrix" shading
  and func =
    Pdf.lookup_fail "No function found" pdf "/Function" shading
  in
    FunctionShading
      {funshading_domain = domain;
       funshading_matrix = matrix;
       funshading_function = Pdffun.parse_function pdf func}

and read_radial_shading pdf shading =
  let coords =
    match Pdf.lookup_direct pdf "/Coords" shading with
    | Some (Pdf.Array [a; b; c; d; e; f]) →
        getnum a, getnum b, getnum c, getnum d, getnum e, getnum f
    | _ → failwith "no coords in radial shading"
  and domain =
    match Pdf.lookup_direct pdf "/Domain" shading with
    | Some (Pdf.Array [a; b]) → getnum a, getnum b
    | _ → 0., 1.
  and func =
    match Pdf.lookup_direct pdf "/Function" shading with
    | Some (Pdf.Array fs) → map (Pdffun.parse_function pdf) fs
    | Some f → [Pdffun.parse_function pdf f]
    | _ → failwith "no function in radial shading"
  and extend =
    match Pdf.lookup_direct pdf "/Extend" shading with
    | Some (Pdf.Array [Pdf.Boolean a; Pdf.Boolean b]) → a, b
    | _ → false, false
  in
    RadialShading
      {radialshading_coords = coords;
```

```
      radialshading_domain = domain;
      radialshading_function = func;
      radialshading_extend = extend}
and read_axial_shading pdf shading =
  let coords =
    match Pdf.lookup_direct pdf "/Coords" shading with
    | Some (Pdf.Array [a; b; c; d]) →
        getnum a, getnum b, getnum c, getnum d
    | _ → failwith "no coords in radial shading"
  and domain =
    match Pdf.lookup_direct pdf "/Domain" shading with
    | Some (Pdf.Array [a; b]) → getnum a, getnum b
    | _ → 0., 1.
  and func =
    match Pdf.lookup_direct pdf "/Function" shading with
    | Some (Pdf.Array fs) → map (Pdffun.parse_function pdf) fs
    | Some f → [Pdffun.parse_function pdf f]
    | _ → failwith "no function in radial shading"
  and extend =
    match Pdf.lookup_direct pdf "/Extend" shading with
    | Some (Pdf.Array [Pdf.Boolean a; Pdf.Boolean b]) → a, b
    | _ → false, false
  in
    AxialShading
      {axialshading_coords = coords;
       axialshading_domain = domain;
       axialshading_function = func;
       axialshading_extend = extend}
```

Read a shading pattern

```
and read_shading pdf matrix extgstate shading =
  let colourspace =
    Pdf.lookup_fail "No colourspace in shading" pdf "/ColorSpace" shading
  and background =
    Pdf.lookup_direct pdf "/Background" shading
  and bbox =
    Pdf.lookup_direct pdf "/BBox" shading
  and antialias =
    match Pdf.lookup_direct pdf "/BBox" shading with
    | Some (Pdf.Boolean true) → true
    | _ → false
  in
    let shading =
      match Pdf.lookup_fail "no /ShadingType" pdf "/ShadingType" shading with
      | Pdf.Integer 1 → read_function_shading pdf shading
      | Pdf.Integer 3 → read_radial_shading pdf shading
      | Pdf.Integer 2 → read_axial_shading pdf shading
            | _ → failwith "unknown shadingtype"
    in
      {shading_colourspace = colourspace;
```

```
      shading_background = background;
      shading_bbox = bbox;
      shading_antialias = antialias;
      shading_matrix = matrix;
      shading_extgstate = extgstate;
      shading = shading}

and read_shading_pattern pdf p =
  let matrix = Pdf.parse_matrix pdf "/Matrix" p
  and extgstate =
    match Pdf.lookup_direct pdf "/ExtGState" p with
    | Some (Pdf.Dictionary _ as d) → d
    | _ → Pdf.Dictionary []
  in
    match Pdf.lookup_direct pdf "/Shading" p with
    | Some shading →
        ShadingPattern (read_shading pdf matrix extgstate shading)
    | _ →
        failwith "No shading dictionary"

and read_pattern pdf page name =
  match lookup_direct pdf "/Pattern" page.Pdfdoc.resources with
  | None → raise (Pdf.PDFError "No pattern dictionary")
  | Some patterndict →
      match lookup_direct pdf name patterndict with
      | None → raise (Pdf.PDFError "Pattern not found")
      | Some pattern →
          match lookup_direct pdf "/PatternType" pattern with
          | Some (Pdf.Integer 1) →
              read_tiling_pattern pattern
          | Some (Pdf.Integer 2) →
              read_shading_pattern pdf pattern
          | _ → failwith "unknown pattern"

and process_op pdf page ((partial, graphic) as return) op =
    match op with
  | Op_W →
      (* Move the current partial path into Clip, and return *)
      begin match partial with
      | PartialPath (_, _, segments, subpaths) →
          if segments = [] ∧ subpaths = [] then return else
            let path =
              if segments ≠ []
                then (Not_hole, Closed, rev segments) :: subpaths
                else subpaths
            in
              !state.clip ← Some (NonZero, path); return
      | _ → return
      end
      (* FIXME: In NextClip needs to support possibly several clips, since we
```

can do W n W n W n f, for instance? *)

```
| Op_W' →
    begin match partial with
    | PartialPath (_, _, segments, subpaths) →
        if segments = [] ∧ subpaths = [] then return else
          let path =
            if segments ≠ []
              then (Not_hole, Closed, rev segments) :: subpaths
              else subpaths
          in
            !state.clip ← Some (EvenOdd, path); return
    | _ → return
    end
| InlineImage (dict, data) →
    (NoPartial, GraphicInlineImage (dict, data, !state.transform) :: graphic)
| Op_MP name →
    begin match !state.objectclass with
    | PageDescriptionLevel → (NoPartial, MCPoint name :: graphic)
    | TextObject → return (* FIXME – Add it to the text partial. *)
    | _ → return (* Invalid, silently drop *)
    end
| Op_DP (name, properties) →
    begin match !state.objectclass with
    | PageDescriptionLevel →
        (NoPartial, MCPointProperties (name, properties) :: graphic)
    | TextObject → return (* FIXME – Add it to the text partial. *)
    | _ → return (* Invalid, silently drop *)
    end
| Op_BX | Op_EX → return
| Op_ri n → !state.intent ← n; return
| Op_j j → !state.joinstyle ← j; return
| Op_J c → !state.capstyle ← c; return
| Op_w w → !state.linewidth ← w; return
| Op_M m → !state.mitrelimit ← m; return
| Op_q →
    push_statestack ();
    return
| Op_Q →
    pop_statestack ();
    return
| Op_SC vals | Op_SCN vals →
    !state.line ← Floats vals;
    return
| Op_sc vals | Op_scn vals →
    !state.fill ← Floats vals;
    return
| Op_scnName (name, vals) when !state.colourspace_nonstroke = Name "/Pattern" →

    !state.fill ← Pattern (read_pattern pdf page name);
    return
```

```
| Op_scnName (name, vals) →
    !state.fill ← Named (name, vals);
    return
| Op_SCNName (name, vals) when !state.colourspace_stroke = Name "/Pattern" →

    !state.line ← Pattern (read_pattern pdf page name);
    return
| Op_SCNName (name, vals) →
    !state.line ← Named (name, vals);
    return
| Op_CS (("/DeviceGray" | "/DeviceRGB" | "/DeviceCMYK" | "/Pattern") as name) →

    !state.colourspace_nonstroke ← Name name;
    !state.line ← initial_colour pdf page.Pdfdoc.resources (Name name);
    return
| Op_CS other →
    (* Look up in colourspace directory *)
    let space =
      match lookup_direct pdf "/ColorSpace" page.Pdfdoc.resources with
      | None →
          raise (PDFSemanticError ("Pdfpages:  Colourspace not found:
" ^ other))
      | Some csdict →
          match lookup_direct pdf other csdict with
          | Some space → space
          | None → raise (PDFSemanticError "Pdfpages:  colourspace
not found")
    in
      !state.colourspace_nonstroke ← space;
      !state.line ← initial_colour pdf page.Pdfdoc.resources space;
      return
| Op_cs (("/DeviceGray" | "/DeviceRGB" | "/DeviceCMYK" | "/Pattern") as name) →

    !state.colourspace_nonstroke ← Name name;
    !state.fill ← initial_colour pdf page.Pdfdoc.resources (Name name);
    return
| Op_cs other →
    (* Look up in colourspace directory *)
    let space =
      match lookup_direct pdf "/ColorSpace" page.Pdfdoc.resources with
      | None →
          raise (PDFSemanticError ("Pdfpages:  Colourspace not found:
" ^ other))
      | Some csdict →
          match lookup_direct pdf other csdict with
          | Some space → space
          | None → raise (PDFSemanticError "Pdfpages:  colourspace
not found")
    in
      !state.colourspace_nonstroke ← space;
```

```
        !state.fill ← initial_colour pdf page.Pdfdoc.resources space;
        return
  | Op_G gv →
      !state.colourspace_stroke ← Name "/DeviceGray";
      !state.line ← Floats [gv];
      return
  | Op_g gv →
      !state.colourspace_nonstroke ← Name "/DeviceGray";
      !state.fill ← Floats [gv];
      return
  | Op_RG (rv, gv, bv) →
      !state.colourspace_stroke ← Name "/DeviceRGB";
      !state.line ← Floats [rv; gv; bv];
      return
  | Op_rg (rv, gv, bv) →
      !state.colourspace_nonstroke ← Name "/DeviceRGB";
      !state.fill ← Floats [rv; gv; bv];
      return
  | Op_K (c, m, y, k) →
      !state.colourspace_stroke ← Name "/DeviceCMYK";
      !state.line ← Floats [c; y; m; k];
      return
  | Op_k (c, m, y, k) →
      !state.colourspace_nonstroke ← Name "/DeviceCMYK";
      !state.fill ← Floats [c; y; m; k];
      return
  | Op_gs name →
      let ext_state_dict = lookup_fail "Bad Op_gs" pdf "/ExtGState" page.Pdfdoc.resources in
        let gdict = lookup_fail "Bad Op_gs" pdf name ext_state_dict in
          update_graphics_state_from_dict pdf page.Pdfdoc.resources gdict;
          return
  | Op_m (x, y) →
      (* Begin a new subpath. Get into path mode if not already there. If the
last op was an Op_m, it should have no effect. *)
      !state.objectclass ← PathObject;
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          if segs = []
            then PartialPath ((x, y), (x, y), [], subpaths), graphic
            else PartialPath ((x, y), (x, y), [], (Not_hole, Open, rev segs) ::
subpaths), graphic
      | _ →
          PartialPath ((x, y), (x, y), [], []), graphic
      end
  | Op_l (x, y) →
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics:  Op_l");
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          PartialPath (sp, (x, y), Straight (cp, (x, y)) :: segs, subpaths), graphic
```

```
      | _ →
          raise (Pdf.PDFError "Pdfgraphics:  Op_l")
      end
  | Op_c (a, b, c, d, e, f) →
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics:  Op_c");
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          let ep = (e, f) in
            let curve = Bezier (cp, (a, b), (c, d), ep) in
              PartialPath (sp, ep, curve :: segs, subpaths), graphic
      | _ →
          raise (Pdf.PDFError "Pdfgraphics:  Op_c")
      end
  | Op_v (a, b, c, d) →
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics:  Op_v");
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          let ep = (c, d) in
            let curve = Bezier (cp, cp, (a, b), ep) in
              PartialPath (sp, ep, curve :: segs, subpaths), graphic
      | _ →
          raise (Pdf.PDFError "Pdfgraphics:  Op_v")
      end
  | Op_y (a, b, c, d) →
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics:  Op_y");
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          let ep = (c, d) in
            let curve = Bezier (cp, (a, b), ep, ep) in
              PartialPath (sp, ep, curve :: segs, subpaths), graphic
      | _ →
          raise (Pdf.PDFError "Pdfgraphics:  Op_y")
      end
  | Op_h →
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics:  Op_h - not in PathObject");
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          PartialPath (sp, cp, [], (Not_hole, Closed, rev segs) :: subpaths), graphic
      | _ →
          raise (Pdf.PDFError "Pdfgraphics:  Op_h - not a partial path")
      end
  | Op_s →
      (* Close and stroke. Equivalent to h S *)
      process_ops pdf page return [Op_h; Op_S]
  | Op_b →
      (* Close, fill, stroke, nonzero. Equivalent to h B *)
```

```
      process_ops pdf page return [Op_h; Op_B]
  | Op_b' →
      (* Close, fill, stroke, evenodd. Equivalent to h B* *)
      process_ops pdf page return [Op_h; Op_B']
  | Op_f | Op_F →
      (* Close and Fill non-zero *)
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics: Op_f");
      let partial, graphic = process_op pdf page (partial, graphic) Op_h in
        !state.objectclass ← PageDescriptionLevel;
        begin match partial with
        | PartialPath (sp, cp, segs, subpaths) →
            (* segs is empty, due to Op_h *)
            PartialPath (sp, cp, [], []),
            Path ((NonZero, rev subpaths), path_attributes_fill ()) :: graphic
        | _ →
           raise (Pdf.PDFError "Pdfgraphics: Op_f")
        end
  | Op_S →
      (* Stroke *)
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics: Op_S");
      !state.objectclass ← PageDescriptionLevel;
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          if segs = [] then
            PartialPath (sp, cp, [], []),
            Path ((EvenOdd, rev subpaths), path_attributes_stroke ()) :: graphic
          else
            PartialPath (sp, cp, [], []),
            Path ((EvenOdd, rev ((Not_hole, Open, rev segs) :: subpaths)), path_attributes_stroke ()) ::
graphic
      | _ →
         raise (Pdf.PDFError "Pdfgraphics: Op_S")
      end
  | Op_B →
      (* Fill and stroke, non-zero. *)
      if !state.objectclass ≠ PathObject then
        raise (Pdf.PDFError "Pdfgraphics: Op_B");
      !state.objectclass ← PageDescriptionLevel;
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          if segs = [] then
            PartialPath (sp, cp, [], []),
            Path ((NonZero, rev subpaths), path_attributes_fill_and_stroke ()) ::
graphic
          else
            PartialPath (sp, cp, [], []),
            Path ((NonZero, rev ((Not_hole, Open, rev segs) :: subpaths)), path_attributes_fill_and_stroke ())
            ::graphic
```

```
    | _ →
      raise (Pdf.PDFError "Pdfgraphics:  Op_B")
    end
| Op_B' →
    (* Fill and stroke, even-odd. *)
    if !state.objectclass ≠ PathObject then
      raise (Pdf.PDFError "Pdfgraphics:  Op_B*");
    let partial, graphic = process_op pdf page (partial, graphic) Op_h in
      !state.objectclass ← PageDescriptionLevel;
      begin match partial with
      | PartialPath (sp, cp, segs, subpaths) →
          if segs = [] then
            PartialPath (sp, cp, [], []),
            Path ((EvenOdd, rev subpaths), path_attributes_fill_and_stroke ()) ::
graphic
          else
            PartialPath (sp, cp, [], []),
            Path ((EvenOdd, rev ((Not_hole, Open, rev segs) :: subpaths)), path_attributes_fill_and
            ::graphic
      | _ →
         raise (Pdf.PDFError "Pdfgraphics:  Op_B*")
      end
| Op_f' →
    (* Fill, even-odd *)
    if !state.objectclass ≠ PathObject then
      raise (Pdf.PDFError "Pdfgraphics:  Op_f*");
    !state.objectclass ← PageDescriptionLevel;
    begin match partial with
    | PartialPath (sp, cp, segs, subpaths) →
        if segs = [] then
          PartialPath (sp, cp, [], []),
          Path ((EvenOdd, rev subpaths), path_attributes_fill ()) :: graphic
        else
          PartialPath (sp, cp, [], []),
          Path ((EvenOdd, rev ((Not_hole, Open, rev segs) :: subpaths)), path_attributes_fill ())
          ::graphic
    | _ →
       raise (Pdf.PDFError "Pdfgraphics:  Op_f*")
    end
| Op_n →
    (* no-op *)
    !state.objectclass ← PageDescriptionLevel;
    (* for now, until we support clipviews, clean up the polygon *)
    (NoPartial, graphic)
| Op_re (x, y, w, h) →
    (* Rectangle. *)
    let ops =
      [Op_m (x, y);
       Op_l (x +. w, y);
       Op_l (x +. w, y +. h);
```

```
        Op_l (x, y +. h);
        Op_h]
    in
      process_ops pdf page (partial, graphic) ops
| Op_Do name →
    begin match Pdf.lookup_direct pdf "/XObject" page.Pdfdoc.resources with
    | Some d →
        begin match Pdf.lookup_direct pdf name d with
        | Some xobj →
            begin match Pdf.lookup_direct pdf "/Subtype" xobj with
            | Some (Pdf.Name "/Image") →
                let objnum =
                  match Pdf.find_indirect name d with
                  | None → failwith "image not found"
                  | Some i → i
                in
                  partial,
                  Image
                     ({image_transform = !state.transform;
                       image_transparency = !state.opacity_nonstroke;
                       image_softmask = !state.softmask}
                     , objnum) :: graphic
            | Some (Pdf.Name "/Form") →
                let elts = read_form_xobject pdf page xobj in
                  partial, rev elts @ graphic
            | _ → failwith "Unknown kind of xobject"
            end
        | _ → failwith "Unknown xobject"
        end
    | None → failwith "xobject not found"
    end
| Op_cm tr →
    !state.transform ← Transform.matrix_compose !state.transform tr;
    return
| (Op_Tc _ | Op_Tw _ | Op_Tz _ | Op_TL _ | Op_Tf _ | Op_Tr _ |
Op_Ts _ | Op_Td _ |
     Op_TD _ | Op_Tm _ | Op_T' | Op_Tj _ | Op_TJ _ | Op_' _ |
Op_'' _ | Op_d0 _ | Op_d1 _) as op →
      begin match partial with
      | PartialText t →
          let st = textstate () in
            PartialText ((st, op) :: t), graphic
      | _ →
        (* If there's no partial text, this is an op affecting the text state but
not in a text section. Such ops are allowed. FIXME: Deal with them properly -
by ops altering the text state so this can be reflected in the initial state at the
start of a text section *)
        return
      end
| Op_sh n →
```

```
      let shading =
        let shadingdict = Pdf.lookup_fail "no /Shading" pdf "/Shading" page.Pdfdoc.resources in
          let shading = Pdf.lookup_fail "named shading not found" pdf n shadingdict in
            read_shading pdf Transform.i_matrix Pdf.Null shading
      and currentclip = !state.clip in
        partial, Shading (currentclip, shading, !state.transform) :: graphic
  | Op_i flatness →
      if flatness ≥ 0 ∧ flatness ≤ 100 then !state.flatness ← flatness;
      return
  | Op_d (spec, phase) →
      !state.dash ← spec, phase;
      return
  | Op_Unknown _ → return
  | _ → failwith "Operator shouldn't appear at this place"
and getuntil_matching_emc level prev = function
  | (Op_BMC _ | Op_BDC (_, _)) as h :: t →
      getuntil_matching_emc (level + 1) (h :: prev) t
  | Op_EMC :: t →
      if level < 0
        then raise (Pdf.PDFError "Too many EMCs\n")
        else if level = 0
          then rev prev, t
          else getuntil_matching_emc (level − 1) (Op_EMC :: prev) t
  | h :: t → getuntil_matching_emc level (h :: prev) t
  | [] → raise (Pdf.PDFError "Missing EMC\n")
and getuntil_matching_Q level prev = function
  | Op_q :: t → getuntil_matching_Q (level + 1) (Op_q :: prev) t
  | Op_Q :: t →
      if level = 0
        then rev prev, Op_Q :: t
        else getuntil_matching_Q (level − 1) (Op_Q :: prev) t
  | [] → rev prev, []
  | h :: t → getuntil_matching_Q level (h :: prev) t
and process_ops pdf page (partial, graphic) = function
  | [] → partial, rev graphic
  | Op_n :: t →
      (* If there's a NextClip, select all operators within the scope of this clip.
That is, all operators until an Op_Q which puts the stack level below the current
level or the end of the stream, whichever comes first.*)
      begin match !state.clip with
      | None →
          process_ops pdf page (partial, graphic) t
      | Some path →
          (* We process the operators concerned, putting them inside a Clip,
and then proceed with the remaining operators (including any Op_Q) *)
          let toq, rest = getuntil_matching_Q 0 [] t in
            let _, elts =
              process_ops pdf page (NoPartial, []) toq
            in
```

```
            process_ops pdf page (NoPartial, Clip (path, elts) :: graphic) rest
      end
  | Op_BMC n :: t →
      let ops, rest = getuntil_matching_emc 0 [] t in
        let partial, graphic' = process_ops pdf page (partial, []) ops in
          process_ops pdf page (partial, MCSection (n, graphic') :: graphic) rest
  | Op_BDC (n, d) :: t →
      let ops, rest = getuntil_matching_emc 0 [] t in
        let partial, graphic' = process_ops pdf page (partial, []) ops in
          process_ops pdf page (partial, MCSectionProperties (n, d, graphic') ::
graphic) rest
  | Op_BT :: t →
     (* Can't nest text sections, so just get to ET *)
     let textops, rest = cleavewhile (neq Op_ET) t in
       begin match rest with
       | Op_ET :: _ | [] →
           (* We allow blank in case of wrongly nested EMC / ET etc *)
           let more = tail_no_fail rest in
           (* We need to process the ops, and capture the text operations, but
state changes inside text sections (e.g colour changes) have global effect, so
need to keep the state *)
           !state.objectclass ← TextObject;
           let partial, _ =
             process_ops pdf page (PartialText [], graphic) textops
           in
             begin match partial with
             | PartialText t →
                 let textblock =
                   Text (rev t, {textblock_transform = !state.transform})
                 in
                   process_ops pdf page (partial, textblock :: graphic) (Op_ET ::
more)
             | _ → failwith "Bad operations in text block"
             end
       | _ →
                   failwith "No Matching Op_ET"
       end
  | Op_ET :: t →
     !state.objectclass ← PageDescriptionLevel;
     process_ops pdf page (partial, graphic) t
  | h :: t → process_ops pdf page (process_op pdf page (partial, graphic) h) t
```

Load the fonts as (name, pdfobject) pairs

```
and fonts_of_page pdf page =
  match Pdf.lookup_direct pdf "/Font" page.Pdfdoc.resources with
  | Some (Pdf.Dictionary fs) → fs
  | _ → []
```

Find the operations of a form xobject.

```
and read_form_xobject pdf page pdfobject =
  let content = [Pdf.direct pdf pdfobject] in
    let pagedict =
      match Pdf.direct pdf page.Pdfdoc.resources with
      | Pdf.Dictionary rs → rs
      | _ → []
    and xobjdict =
      match Pdf.direct pdf pdfobject with
      | Pdf.Stream {contents = (dict, _)} →
          begin match Pdf.lookup_direct pdf "/Resources" dict with
          | Some (Pdf.Dictionary rs) → rs
          | _ → []
          end
      | _ → failwith "bad stream"
    in
      let total_resources =
        Pdf.Dictionary (mergedict pagedict xobjdict)
      in
        let fake_page =
          {Pdfdoc.content = [];
           Pdfdoc.mediabox = Pdf.Null;
           Pdfdoc.resources = total_resources;
           Pdfdoc.rotate = Pdfdoc.Rotate0;
           Pdfdoc.rest = Pdf.Dictionary []}
        in
          let _, graphic_elts =
              (process_ops pdf fake_page (NoPartial, []))
              (Pdfpages.parse_operators pdf total_resources content))
          in
            graphic_elts
```

Main function - build a graphic from a page

```
and graphic_of_page pdf page =
  if Pdfcrypt.is_encrypted pdf then failwith "File is encrypted" else
    begin
      let _, elts =
        let ops =
          Pdfpages.parse_operators pdf page.Pdfdoc.resources page.Pdfdoc.content
        in
          process_ops pdf page (NoPartial, []) ops
      in
        {elements = elts;
         fonts = fonts_of_page pdf page;
         resources = page.Pdfdoc.resources}
    end

let graphic_of_ops ops =
    graphic_of_page
      (Pdf.empty ())
      {(Pdfdoc.blankpage Paper.a4) with
           Pdfdoc.content =
```

```
      [Pdf.Stream {contents =
        (Pdf.Dictionary [], Pdf.Got (bytestream_of_string (Pdfpages.string_of_ops ops)))}]}
```

17.1 Building a page from a graphic

```
let int_of_shading_kind = function
  | FunctionShading _ → 1
  | AxialShading _ → 2
  | RadialShading _ → 3
  | FreeFormGouraudShading → 4
  | LatticeFormGouraudShading → 5
  | CoonsPatchMesh → 6
  | TensorProductPatchMesh → 7

let entries_of_shading pdf s =
  match s.shading with
  | RadialShading r →
      let coords =
        let a, b, c, d, e, f = r.radialshading_coords in
          Pdf.Array
            [Pdf.Real a; Pdf.Real b; Pdf.Real c; Pdf.Real d; Pdf.Real e; Pdf.Real f]
      and domain =
        let a, b = r.radialshading_domain in
          Pdf.Array
            [Pdf.Real a; Pdf.Real b]
      and funcnum =
        match r.radialshading_function with
        | [f] →
           Pdf.addobj pdf (Pdffun.pdfobject_of_function pdf f)
        | funs →
           Pdf.addobj pdf (Pdf.Array (map (Pdffun.pdfobject_of_function pdf) funs))
      and extend =
        Pdf.Array
          [Pdf.Boolean (fst r.radialshading_extend);
           Pdf.Boolean (snd r.radialshading_extend)]
      in
        ["/Coords", coords;
         "/Domain", domain;
         "/Function", Pdf.Indirect funcnum;
         "/Extend", extend]
  | AxialShading a →
      let coords =
        let a, b, c, d = a.axialshading_coords in
          Pdf.Array
            [Pdf.Real a; Pdf.Real b; Pdf.Real c; Pdf.Real d]
      and domain =
        let a, b = a.axialshading_domain in
          Pdf.Array
```

```
            [Pdf.Real a; Pdf.Real b]
      and funcnum =
        match a.axialshading_function with
        | [f] →
            Pdf.addobj pdf (Pdffun.pdfobject_of_function pdf f)
        | funs →
            Pdf.addobj pdf (Pdf.Array (map (Pdffun.pdfobject_of_function pdf) funs))
      and extend =
        Pdf.Array
          [Pdf.Boolean (fst a.axialshading_extend);
           Pdf.Boolean (snd a.axialshading_extend)]
      in
        ["/Coords", coords;
         "/Domain", domain;
         "/Function", Pdf.Indirect funcnum;
         "/Extend", extend]
  | _ → []

let shading_object_of_shading pdf s =
  let background =
    match s.shading_background with
    | None → []
    | Some b → ["/Background", b]
  and bbox =
    match s.shading_bbox with
    | None → []
    | Some b → ["/BBox", b]
  in
    Pdf.Dictionary
      (["/ShadingType", Pdf.Integer (int_of_shading_kind s.shading);
        "/ColorSpace", s.shading_colourspace;
        "/AntiAlias", Pdf.Boolean s.shading_antialias]
      @ background @ bbox @ entries_of_shading pdf s)

let pattern_object_of_pattern xobject_level opdo_matrix pdf = function
  | ShadingPattern s →
      begin try
        let shading_matrix =
          if xobject_level > 0 then
            let inverted = Transform.matrix_invert opdo_matrix in
              Transform.matrix_compose inverted s.shading_matrix
          else
            s.shading_matrix
        in
          Pdf.Dictionary
            ["/Type", Pdf.Name "/Pattern";
             "/PatternType", Pdf.Integer 2;
             "/Shading", shading_object_of_shading pdf s;
             "/Matrix", Pdf.make_matrix shading_matrix]
      with
        Transform.NonInvertable → failwith "Bad pattern"
```

```
      end
  | _ →
      Printf.eprintf "Unknown pattern\n";
      Pdf.Dictionary []
```

Output a move and line/curve ops.

```
let ops_of_segs segs closure =
  let raw_seg_ops =
    map
      (function
       | Straight (_, (x, y)) → Op_l (x, y)
       | Bezier (_, (bx, by), (cx, cy), (dx, dy)) → Op_c (bx, by, cx, cy, dx, dy))
      segs
  and get_move = function
    | Straight ((x, y), _) | Bezier ((x, y), _, _, _) → Op_m (x, y)
  in
    (* Predicate: Do we need to close this subpath? *)
    match segs with
    | [] → []
    | h :: _ → get_move h :: raw_seg_ops @ (if closure = Closed then [Op_h] else [])

let protect ops =
  [Op_q] @ ops @ [Op_Q]

let attribute_ops_of_path (_, a) =
  [Op_w a.path_linewidth;
   Op_J a.path_capstyle;
   Op_j a.path_joinstyle;
   Op_M a.path_mitrelimit;
   Op_ri a.path_intent]

let transform_ops_of_path (_, a) =
  [Op_cm a.path_transform]

let stroke_ops_of_path ((winding, _), a) =
  match winding, a.path_fill, a.path_line with
  | _, None, None → Op_n
  | EvenOdd, Some _, Some _ → Op_B'
  | EvenOdd, Some _, None → Op_f'
  | NonZero, Some _, Some _ → Op_B
  | NonZero, Some _, None → Op_f
  | _, None, Some _ → Op_S

let path_ops_of_path (_, subpaths) =
  flatten (map (fun (_, closure, segs) → ops_of_segs segs closure) subpaths)

let ops_of_path pdf page (((winding, subpaths), a) as p) =
  let resources = page.Pdfdoc.resources in
  let attribute_ops = attribute_ops_of_path p
  and transform = transform_ops_of_path p
  and stroke_op = stroke_ops_of_path p in
  let colours_stroke, resources =
    match a.path_line with
```

```
      | Some (cs, Floats vals) →
          [Op_CS (Pdfwrite.string_of_pdf cs); Op_SCN vals], resources
      | Some (cs, Named (n, vals)) →
          [Op_CS (Pdfwrite.string_of_pdf cs); Op_SCNName (n, vals)], resources
      | _ → [], resources
    in
    let colours_nonstroke, resources =
      match a.path_fill with
      | Some (cs, Floats vals) →
          [Op_cs (Pdfwrite.string_of_pdf cs); Op_scn vals], resources
      | Some (cs, Named (n, vals)) →
          [Op_cs (Pdfwrite.string_of_pdf cs); Op_scnName (n, vals)], resources
      | Some (_, Pattern p) →
          (* Build /Pattern cs and reference to pattern, having built the pattern
  in the pattern dictionary *)
          let pattern = pattern_object_of_pattern !state.in_xobject !state.opdo_matrix pdf p in
           let resources, name =
             let existing_patterndict =
               match lookup_direct pdf "/Pattern" resources with
               | Some ((Dictionary _) as d) → d
               | _ → Dictionary []
             in
               let name = unique_key "pt" existing_patterndict in
                 let newpatterndict = add_dict_entry existing_patterndict name pattern in
                   add_dict_entry page.Pdfdoc.resources "/Pattern" newpatterndict, name
          in
            [Op_cs "/Pattern"; Op_scnName (name, [])], resources
      | _ → [], resources
    in
    let gs, resources =
      if a.path_transparency.fill_transparency < 1. ∨ a.path_transparency.line_transparency < 1.
        then
           let resources, name =
             let existing_extgstate =
               match lookup_direct pdf "/ExtGState" resources with
               | Some ((Dictionary _) as d) → d
               | _ → Dictionary []
             in
               let name = unique_key "gs" existing_extgstate
               and gsdict =
                 Dictionary
                   [("/ca", Real a.path_transparency.fill_transparency);
                    ("/CA", Real a.path_transparency.line_transparency)]
               in
                 let new_extgstate = add_dict_entry existing_extgstate name gsdict in
                   add_dict_entry page.Pdfdoc.resources "/ExtGState" new_extgstate, name
           in
             [Op_gs name], resources
        else
          [], resources
```

```
  in
   let path_ops = path_ops_of_path (winding, subpaths) in
     protect (gs @ transform @ attribute_ops @ colours_stroke @ colours_nonstroke @ path_ops @ [stroke_op]),
     resources

let ops_of_textstate st = []

let ops_of_textpiece (st, op) =
  ops_of_textstate st @ [op]
```

Upon entry to this, the transformation matrix is identity

```
let ops_of_text tr ops =
  protect ([Op_cm tr; Op_BT] @ (flatten <| map ops_of_textpiece ops) @ [Op_ET])
```

Transform a bounding box by a given matrix

```
let extreme_of_4 f a b c d =
  hd <| sort f [a; b; c; d]

let min_of_4 = extreme_of_4 compare

let max_of_4 = extreme_of_4 (fun a b → − (compare a b))

let transform_bbox tr l b r t =
  let (x0, y0) = Transform.transform_matrix tr (l, t)
  and (x1, y1) = Transform.transform_matrix tr (l, b)
  and (x2, y2) = Transform.transform_matrix tr (r, t)
  and (x3, y3) = Transform.transform_matrix tr (r, b) in
    min_of_4 x0 x1 x2 x3,
    min_of_4 y0 y1 y2 y3,
    max_of_4 x0 x1 x2 x3,
    max_of_4 y0 y1 y2 y3
```

Build a transparency group xobject, add it to the pdf and return its object number

```
let rec pdfobject_of_transparency_group (a, b, c, d) pdf t =
  !state.in_xobject ← !state.in_xobject + 1;
      let r =
      let page = page_of_graphic pdf (0., 0., 0., 0.) t.tr_graphic
      and group_attributes =
        let cs =
          match t.tr_group_colourspace with
          | None → []
          | Some pdfobject → ["/CS", pdfobject]
        in
          Pdf.Dictionary
            (["/Type", Pdf.Name "/Group";
              "/S", Pdf.Name "/Transparency";
              "/I", Pdf.Boolean t.isolated;
              "/K", Pdf.Boolean t.knockout] @ cs)
      in
        let extras =
          ["/Type", Pdf.Name "/XObject";
           "/Subtype", Pdf.Name "/Form";
```

```
            "/BBox", Pdf.Array [Pdf.Real a; Pdf.Real b; Pdf.Real c; Pdf.Real d];
            "/Resources", page.Pdfdoc.resources;
            "/Group", group_attributes]
        in
          match page.Pdfdoc.content with
          | Pdf.Stream ({contents = Pdf.Dictionary dict, Got data}) :: _ →
              Pdf.addobj pdf (Pdf.Stream ({contents = Pdf.Dictionary (extras @ dict), Got data}))
          | _ → failwith "Bad page content"
    in
      !state.in_xobject ← !state.in_xobject − 1;
      r

and pdfobject_of_softmask pdf m =
  let bc =
    match m.backdrop with
    | None → []
    | Some fs → ["/BC", Pdf.Array (map (function x → Pdf.Real x) fs)]
  and tr =
    match m.softmask_transfer with
    | None → []
    | Some f → ["/TR", Pdffun.pdfobject_of_function pdf f]
  in
    Pdf.addobj pdf
      (Pdf.Dictionary
        (["/Type", Pdf.Name "/Mask";
          "/S", Pdf.Name (match m.softmask_subtype with Alpha → "/Alpha" |
Luminosity → "/Luminosity");
          "/G", Pdf.Indirect (pdfobject_of_transparency_group m.softmask_bbox pdf m.transparency_g
        @ bc @ tr))

and ops_of_image pdf page (a, i) =
  !state.opdo_matrix ← a.image_transform;
  let resources = page.Pdfdoc.resources in
    let ops, resources =
      let opgs, resources =
        if a.image_transparency < 1. ∨ a.image_softmask ≠ None
          then
            let resources, name =
              let existing_extgstate =
                match lookup_direct pdf "/ExtGState" page.Pdfdoc.resources with
                | Some ((Dictionary _) as d) → d
                | _ → Dictionary []
              in
                let name = unique_key "gs" existing_extgstate
                and gsdict =
                  let softmask =
                    match a.image_softmask with
                    | None → []
                    | Some m → ["/SMask", Pdf.Indirect (pdfobject_of_softmask pdf m)]
                  in
                    Dictionary
```

```
                    ([("/ca", Real a.image_transparency)] @ softmask)
                in
                  let new_extgstate = add_dict_entry existing_extgstate name gsdict in
                    add_dict_entry resources "/ExtGState" new_extgstate, name
              in
                [Op_gs name], resources
          else
            [], resources
      in
        [Op_cm a.image_transform] @ opgs @ [Op_Do ("/Im" ^ string_of_int i)], resources
    in
      protect ops, resources

and ops_of_shading pdf page path shading transform =
  let resources', name =
    (* Add new entry to shading dictionary, return its name, new resources *)
    let existing_shadingdict =
      match lookup_direct pdf "/Shading" page.Pdfdoc.resources with
      | Some ((Dictionary _) as d) → d
      | _ → Dictionary []
    in
      let name = unique_key "sh" existing_shadingdict
      and objnum = Pdf.addobj pdf (shading_object_of_shading pdf shading) in
        let shadingref = Pdf.Indirect objnum in
          let new_shadingdict = add_dict_entry existing_shadingdict name shadingref in
            let r =
              add_dict_entry page.Pdfdoc.resources "/Shading" new_shadingdict
            in
              r, name
  in
    let ops =
      let pathops, clipops =
        match path with
        | None → [], []
        | Some p →
            path_ops_of_path p, [Op_W; Op_n] (* FIXME: Even-odd vs Non-
Zero *)
      in
        pathops @ clipops @ [Op_cm transform; Op_sh name]
    in
      protect ops, resources'

and ops_of_graphic_acc pdf page oplists q =
  match q with
  | [] →
      flatten (rev oplists), page
  | Path p :: t →
      let ops, resources' = ops_of_path pdf page p in
        let page' = {page with Pdfdoc.resources = resources'} in
          ops_of_graphic_acc pdf page' (ops :: oplists) t
  | Image (attr, i) :: t →
```

```
        let ops, resources' = ops_of_image pdf page (attr, i) in
          let page' = {page with Pdfdoc.resources = resources'} in
            ops_of_graphic_acc pdf page' (ops :: oplists) t
  | Text (ts, {textblock_transform = tr}) :: t →
        let ops = ops_of_text tr ts in
          ops_of_graphic_acc pdf page (ops :: oplists) t
  | MCSection (n, graphic) :: t →
        let oplist, page' =
          ops_of_graphic_acc pdf page [] graphic
        in
          ops_of_graphic_acc pdf page' (([Op_BMC n] @ oplist @ [Op_EMC]) ::
oplists) t
  | MCSectionProperties (n, d, graphic) :: t →
        let oplist, page' =
          ops_of_graphic_acc pdf page [] graphic
        in
          ops_of_graphic_acc pdf page' (([Op_BDC (n, d)] @ oplist @ [Op_EMC]) ::
oplists) t
  | MCPoint n :: t →
        ops_of_graphic_acc pdf page ([Op_MP n] :: oplists) t
  | MCPointProperties (n, d) :: t →
        ops_of_graphic_acc pdf page ([Op_DP (n, d)] :: oplists) t
  | GraphicInlineImage (dict, data, tr) :: t →
        ops_of_graphic_acc pdf page (protect [Op_cm tr; InlineImage (dict, data)] ::
oplists) t
  | Clip ((w, _) as p, elts) :: t →
        let ops, page' =
          let path_ops =
            [Op_cm !state.transform] @ path_ops_of_path p
          and clipviewops =
            [if w = NonZero then Op_W else Op_W'; Op_n]
          and insideclipops, page' =
            ops_of_graphic_acc pdf page [] elts
          in
            protect (path_ops @ clipviewops @ insideclipops), page'
        in
          ops_of_graphic_acc pdf page' (ops :: oplists) t
   | Shading (path, shading, transform) :: t →
         let ops, resources' = ops_of_shading pdf page path shading transform in
           let oplists' = protect ops :: oplists
           and page' = {page with Pdfdoc.resources = resources'} in
             ops_of_graphic_acc pdf page' oplists' t
```

Build a page from a graphic in the same PDF.

```
and image_numbers_of_elts prev = function
  | Image (_, i) :: t → image_numbers_of_elts (i :: prev) t
  | MCSection (_, elts) :: t
  | MCSectionProperties (_, _, elts) :: t
  | Clip (_, elts) :: t →
        let these = image_numbers_of_elts [] elts in
```

```
        image_numbers_of_elts (these @ prev) t
  | _ :: t → image_numbers_of_elts prev t
  | [] → prev

and make_xobjects pdf elts =
  let numbers = image_numbers_of_elts [] elts in
    setify <| map (function n → ("/Im" ^ string_of_int n), Pdf.Indirect n) numbers

and make_resources pdf g page' =
  let resources =
    match g.resources with
    | Pdf.Dictionary rs → rs
    | _ → []
  and fontdict =
    Pdf.Dictionary g.fonts
  and xobjdict =
    let objs = make_xobjects pdf g.elements in
      Pdf.Dictionary objs
  and resources_frompage =
    match page'.Pdfdoc.resources with
    | Pdf.Dictionary d → d
    | _ → assert false
  in
    let resources = remove "/Shading" resources in
    let resources = remove "/Pattern" resources in
    let resources = remove "/ExtGState" resources in
      (* fold_right so that entries overwrite *)
      Pdf.Dictionary
        (fold_right
            (fun (k, v) d → add k v d)
            ["/Font", fontdict; "/XObject", xobjdict]
            (resources_frompage @ resources))

and page_of_graphic pdf (xmin, ymin, xmax, ymax) graphic =
  let page =
    Pdfdoc.custompage (Pdf.Array [Pdf.Real xmin; Pdf.Real ymin; Pdf.Real xmax; Pdf.Real ymax])
  in
    let ops, page' = ops_of_graphic_acc pdf page [] graphic.elements in
      (* We're not including the ExtGState because it's in the page', so need to
merge with resources *)
      let resources = make_resources pdf graphic page' in
        {page' with
            Pdfdoc.content = [Pdfpages.stream_of_ops ops];
            Pdfdoc.resources = resources}

let ops_of_simple_graphic graphic =
  fst (ops_of_graphic_acc (Pdf.empty ()) (Pdfdoc.blankpage Paper.a4) [] graphic.elements)
```

FIXME Add in here a function to copy a page/graphic from one document to another

```
let streams_of_simple_graphic g =
  (page_of_graphic (Pdf.empty ()) (0., 0., 600., 400.) g).Pdfdoc.content
```

PDF Colour space parsing

```
open Utility

type point = float × float × float

type iccbased =
 {icc_n : int;
  icc_alternate : colourspace;
  icc_range : float array;
  icc_metadata : Pdf.pdfobject option;
  icc_stream : Pdf.pdfobject}

and colourspace =
  | DeviceGray
  | DeviceRGB
  | DeviceCMYK
  | CalGray of point × point × float (* White, Black, Gamma *)
  | CalRGB of point × point × float array × float array (* White, Black,
Gamma, Matrix *)
  | Lab of point × point × float array (* White, Black, Range *)
  | ICCBased of iccbased
  | Indexed of colourspace × (int, int list) Hashtbl.t (* Base colourspace, values
*)
  | Pattern
  | Separation of string × colourspace × Pdffun.pdf_fun
  | DeviceN of string array × colourspace × Pdffun.pdf_fun × Pdf.pdfobject

let rec string_of_colourspace = function
  | DeviceGray → "/DeviceGray"
  | DeviceRGB → "/DeviceRGB"
  | DeviceCMYK → "/DeviceCMYK"
  | CalGray (_, _, _) → "/CalGray"
  | CalRGB (_, _, _, _) → "/CalRGB"
  | Lab (_, _, _) → "/Lab"
  | ICCBased {icc_alternate = a} →
      "ICC Based - alternate is " ^ string_of_colourspace a
  | Indexed (a, _) →
      "Indexed - base is " ^ string_of_colourspace a
  | Pattern → "/Pattern"
  | Separation (_, a, _) →
      "Separation - base is " ^ string_of_colourspace a
  | DeviceN (_, a, _, _) →
      "DeviceN - base is " ^ string_of_colourspace a
```

Read a tristimulus point.

```
let read_point pdf d n =
  match Pdf.lookup_direct pdf n d with
  | Some (Pdf.Array [a; b; c]) →
      Pdf.getnum a, Pdf.getnum b, Pdf.getnum c
  | _ →
      0., 0., 0.
```

```
let rec get_basic_table_colourspace c =
  match c with
  | Indexed (alt, _)
  (* FIXME Not actually checked the following two are correct *)
  | DeviceN (_, alt, _, _)
  | Separation (_, alt, _)
  | ICCBased {icc_alternate = alt} → get_basic_table_colourspace alt
  | x → x
```

Read a colour space. Raises Not_found on error.

```
let rec read_colourspace_inner pdf resources = function
  | Pdf.Indirect i →
      read_colourspace_inner pdf resources (Pdf.direct pdf (Pdf.Indirect i))
  | Pdf.Name ("/DeviceGray" | "/G") → DeviceGray
  | Pdf.Name ("/DeviceRGB" | "/RGB") → DeviceRGB
  | Pdf.Name ("/DeviceCMYK" | "/CMYK") → DeviceCMYK
  | Pdf.Name "/Pattern" → Pattern
  | Pdf.Array [Pdf.Name "/Pattern"; base_colspace] → Pattern (* FIXME *)
  | Pdf.Array [onething] → read_colourspace_inner pdf resources onething (*
illus_effects.pdf [/Pattern] *)
  | Pdf.Name space →
          begin match Pdf.lookup_direct pdf "/ColorSpace" resources with
      | Some csdict →
          begin match Pdf.lookup_direct pdf space csdict with
          | Some space' →
                  read_colourspace_inner pdf resources space'
          | None → dpr "X"; raise Not_found
          end
      | None → dpr "Y"; raise Not_found
      end
  | Pdf.Array [Pdf.Name "/CalGray"; dict] →
      let whitepoint = read_point pdf dict "/WhitePoint"
      and blackpoint = read_point pdf dict "/BlackPoint"
      and gamma =
        match Pdf.lookup_direct pdf "/Gamma" dict with
        | Some n → Pdf.getnum n
        | None → 1.
      in
        CalGray (whitepoint, blackpoint, gamma)
  | Pdf.Array [Pdf.Name "/CalRGB"; dict] →
      let whitepoint = read_point pdf dict "/WhitePoint"
      and blackpoint = read_point pdf dict "/BlackPoint"
      and gamma =
        match Pdf.lookup_direct pdf "/Gamma" dict with
        | Some (Pdf.Array [a; b; c]) →
            [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c|]
        | _ →
            [|1.; 1.; 1.|]
      and matrix =
        match Pdf.lookup_direct pdf "/Matrix" dict with
```

```
        | Some (Pdf.Array [a; b; c; d; e; f; g; h; i]) →
            [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c;
              Pdf.getnum d; Pdf.getnum e; Pdf.getnum f;
              Pdf.getnum g; Pdf.getnum h; Pdf.getnum i|]
        | _ →
            [|1.; 0.; 0.; 0.; 1.; 0.; 0.; 0.; 1.|]
      in
        CalRGB (whitepoint, blackpoint, gamma, matrix)
  | Pdf.Array [Pdf.Name "/Lab"; dict] →
      let whitepoint = read_point pdf dict "/WhitePoint"
      and blackpoint = read_point pdf dict "/BlackPoint"
      and range =
        match Pdf.lookup_direct pdf "/Range" dict with
        | Some (Pdf.Array [a; b; c; d]) →
            [|Pdf.getnum a; Pdf.getnum b; Pdf.getnum c; Pdf.getnum d|]
        | _ →
            [|~-.100.; 100.; ~-.100.; 100.|]
      in
        Lab (whitepoint, blackpoint, range)
  | Pdf.Array [Pdf.Name "/ICCBased"; stream] →
      begin match Pdf.direct pdf stream with
      | Pdf.Stream {contents = (dict, _)} →
          let n =
            match Pdf.lookup_direct pdf "/N" dict with
            | Some (Pdf.Integer n) →
                if n = 1 ∨ n = 3 ∨ n = 4 then n else raise Not_found
            | _ → raise Not_found
          in
            let alternate =
              match Pdf.lookup_direct pdf "/Alternate" dict with
              | Some cs → read_colourspace_inner pdf resources cs
              | _ →
                  match n with
                  | 1 → DeviceGray
                  | 3 → DeviceRGB
                  | 4 → DeviceCMYK
                  | _ → raise (Assert_failure ("", 0, 0))
            and range =
              match Pdf.lookup_direct pdf "/Range" dict with
              | Some (Pdf.Array elts) when length elts = 2 × n →
                  Array.of_list (map Pdf.getnum elts)
              | _ →
                  Array.of_list (flatten (many [0.; 1.] n))
            and metadata =
              Pdf.lookup_direct pdf "/Metadata" dict
            in
              ICCBased
                {icc_n = n;
                 icc_alternate = alternate;
                 icc_range = range;
```

```
              icc_metadata = metadata;
              icc_stream = stream}
    | _ → raise Not_found
    end
| Pdf.Array [Pdf.Name ("/Indexed" | "/I"); bse; hival; lookup_data] →
    let hival =
      match hival with
      | Pdf.Integer h → h
      | _ → raise (Pdf.PDFError "Bad /Hival")
    and bse =
      read_colourspace_inner pdf resources bse
    in
      let mktable_rgb data =
        try
          let table = Hashtbl.create (hival + 1)
          and i = Pdfio.input_of_bytestream data in
            for x = 0 to hival do
              let r = i.Pdfio.input_byte () in
              let g = i.Pdfio.input_byte () in
              let b = i.Pdfio.input_byte () in
                Hashtbl.add table x [r; g; b]
            done;
            table
        with _ → failwith "bad table"
      and mktable_cmyk data =
        try
          let table = Hashtbl.create (hival + 1)
          and i = Pdfio.input_of_bytestream data in
            for x = 0 to hival do
              let c = i.Pdfio.input_byte () in
              let m = i.Pdfio.input_byte () in
              let y = i.Pdfio.input_byte () in
              let k = i.Pdfio.input_byte () in
                Hashtbl.add table x [c; m; y; k]
            done;
            table
        with _ → failwith "bad table"
      in
        let table =
          begin match Pdf.direct pdf lookup_data with
          | (Pdf.Stream _) as stream →
              Pdfcodec.decode_pdfstream pdf stream;
              begin match stream with
              | (Pdf.Stream {contents = (_, Pdf.Got data)}) →
                  begin match get_basic_table_colourspace bse with
                  | DeviceRGB | CalRGB _ → mktable_rgb data
                  | DeviceCMYK → mktable_cmyk data
                  | _ → failwith "Unknown base colourspace in index
colourspace"
                  end
```

```
                | _ → raise (Pdf.PDFError "Indexed/Inconsistent")
                end
            | Pdf.String s →
                let data = mkstream (String.length s) in
                  for x = 0 to stream_size data − 1 do
                    sset data x (int_of_char s.[x])
                  done;
                  begin match get_basic_table_colourspace bse with
                  | DeviceRGB | CalRGB _ → mktable_rgb data
                  | DeviceCMYK → mktable_cmyk data
                  | _ → failwith "Unknown base colourspace in index
colourspace"
                  end
            | _ → failwith "unknown indexed colourspace"
            end
          in
            Indexed (bse, table)
  | Pdf.Array [Pdf.Name "/Separation"; Pdf.Name name; alternate; tint] →
      let alt_space =
        read_colourspace_inner pdf resources alternate
      and tint_transform =
        Pdffun.parse_function pdf tint
      in
        Separation (name, alt_space, tint_transform)
  | Pdf.Array [Pdf.Name "/DeviceN"; Pdf.Array names; alternate; tint] →
      let names =
        Array.of_list (map (function Pdf.Name s → s | _ → raise Not_found) names)
      and alternate =
        read_colourspace_inner pdf resources alternate
      and tint =
        Pdffun.parse_function pdf tint
      in
        DeviceN (names, alternate, tint, Pdf.Dictionary [])
  | Pdf.Array [Pdf.Name "/DeviceN"; Pdf.Array names; alternate; tint; attributes] →

      let names =
        Array.of_list (map (function Pdf.Name s → s | _ → raise Not_found) names)
      and alternate =
        read_colourspace_inner pdf resources alternate
      and tint =
        Pdffun.parse_function pdf tint
      in
        DeviceN (names, alternate, tint, attributes)
  | _ → raise Not_found

let read_colourspace pdf resources space =
  try
    read_colourspace_inner pdf resources space
  with
    e →
```

raise e

Part III

Examples

18 Module PdfHello
Hello world, in PDF

We build a font dictionary for one of the 14 standard PostScript fonts, (which are supported by all PDF readers), make a graphics stream using the Pdfpages module, build a PDF document in memory and then write it to `hello.pdf`

```
let _ =
  let font =
    Pdf.Dictionary
      [("/Type", Pdf.Name "/Font");
       ("/Subtype", Pdf.Name "/Type1");
       ("/BaseFont", Pdf.Name "/Times-Italic")]
  and ops =
    [Pdfpages.Op_cm (Transform.matrix_of_transform [Transform.Translate (50., 770.)]);
     Pdfpages.Op_BT;
     Pdfpages.Op_Tf ("/F0", 36.);
     Pdfpages.Op_Tj "Hello, World!";
     Pdfpages.Op_ET]
  in
    let page =
      {(Pdfdoc.blankpage Paper.a4) with
          Pdfdoc.content = [Pdfpages.stream_of_ops ops];
          Pdfdoc.resources = Pdf.Dictionary [("/Font", Pdf.Dictionary [("/F0", font)])]}
    in
      let pdf, pageroot = Pdfdoc.add_pagetree [page] (Pdf.empty ()) in
        let pdf = Pdfdoc.add_root pageroot [] pdf in
          Pdfwrite.pdf_to_file pdf "hello.pdf"
```

19 Module Pdfdecomp

Decompress streams

Summary: `pdfdecomp a.pdf b.pdf` decompresses all streams in `a.pdf`, writing the result to `b.pdf`.

```
let decompress_pdf pdf =
  Pdf.map_stream (fun x → Pdfcodec.decode_pdfstream_until_unknown pdf x; x) pdf

let _ =
  match Array.to_list Sys.argv with
  | [_; in_file; out_file] →
      begin try
        let pdf = Pdfread.pdf_of_file None in_file in
          Pdfwrite.pdf_to_file (decompress_pdf pdf) out_file
      with
        err →
          Printf.printf "Failed to decompress file.\n%s\n\n" (Printexc.to_string err);
          exit 1
      end
  | _ →
      print_string "Syntax:  pdfdecomp <input> <output>\n\n"; exit 1
```

20 Module Pdftest
Test on a document

Summary: `pdftest in.pdf out.pdf` reads, lexes, parses a document `in.pdf` and its graphics streams, then writes it to `out.pdf`.

```
open Utility

let _ =
  let in_name, out_name =
    match tl (Array.to_list Sys.argv) with
    | [i; o] → i, o
    | _ → print_string "Syntax:  pdftest <input> <output>\n\n"; exit 1
  in
    try
      let pdf = Pdfread.pdf_of_file None in_name in
        let pages = Pdfdoc.pages_of_pagetree pdf in
          let pages' =
                          pages
          in
            let pdf = Pdfdoc.change_pages true pdf pages' in
              Pdf.remove_unreferenced pdf;
              Pdfwrite.pdf_to_file pdf out_name
    with
      err →
        Printf.printf "Test failed:\n%s\n\n" (Printexc.to_string err);
        exit 1
```

21 Module Pdfmerge
Concatentate documents

Summary: `pdfmerge a.pdf b.pdf c.pdf` appends `b.pdf` to `a.pdf` and writes to `c.pdf`.

```
open Utility
```

We read all the files, read their pages and concatenate them, dealing with clashing object numbers. We then build a new page tree, and build the output PDF document, with a new root and trailer dictionary. We then remove any unreferenced objects, and write to file.

```
let merge_pdfs pdfs out_name =
  let pdfs = Pdf.renumber_pdfs pdfs
  and minor' = fold_left max 0 (map (fun p -> p.Pdf.minor) pdfs) in
    let pages = flatten (map Pdfdoc.pages_of_pagetree pdfs)
    and pdf = ref (Pdf.empty ()) in
      iter (Pdf.objiter (fun k v -> ignore (Pdf.addobj_given_num !pdf (k, v)))) pdfs;
      let pdf, pagetree_num = Pdfdoc.add_pagetree pages !pdf in
        let pdf = Pdfdoc.add_root pagetree_num [] pdf in
          let pdf = {pdf with Pdf.major = 1; Pdf.minor = minor'} in
            Pdf.remove_unreferenced pdf;
            pdf
```

Read command line arguments, read files, call *merge*, write result.

```
let _ =
  let in_names, out_name =
    match rev (tl (Array.to_list Sys.argv)) with
    | h :: t :: t' -> rev (t :: t'), h
    | _ -> print_string "Syntax:  pdfmerge <inputs> <output>\n\n"; exit 1
  in
    try
      let pdfs = map (Pdfread.pdf_of_file None) in_names in
        let result = merge_pdfs pdfs out_name in
          Pdfwrite.pdf_to_file result out_name
    with
      err ->
        Printf.printf "Failed to merge files.\n%s\n\n" (Printexc.to_string err);
        exit 1
```

22 Module Pdfdraft
Make Draft Documents

Make a PDF suitable for draft printing by replacing its images by crossed boxes. Summary: pdfdraft `input.pdf output.pdf`.

```
open Utility
```

Predicate on an xobject: true if an image xobject.

```
let isimage pdf (_, xobj) =
  Pdf.lookup_direct pdf "/Subtype" xobj = Some (Pdf.Name "/Image")
```

Given a set of resources for a page, and the name of a resource, determine if that name refers to an image xobject.

```
let xobject_isimage pdf resources name =
  match resources with
  | Pdf.Dictionary _ →
      begin match Pdf.lookup_direct pdf "/XObject" resources with
      | Some xobjects →
          isimage pdf ("", Pdf.lookup_fail "xobject not there" pdf name xobjects)
      | _ → false
      end
  | _ → failwith "bad resources"
```

Remove any image xobjects from a set of resources.

```
let remove_image_xobjects pdf resources =
  match resources with
  | Pdf.Dictionary res →
      begin match Pdf.lookup_direct pdf "/XObject" resources with
      | Some (Pdf.Dictionary xobjects) →
          Pdf.Dictionary
            (replace "/XObject" (Pdf.Dictionary (lose (isimage pdf) xobjects)) res)
      | _ → resources
      end
  | _ → failwith "bad resources"
```

The subsitute for an image.

```
let substitute =
  rev
    [Pdfpages.Op_q;
     Pdfpages.Op_w 0.;
     Pdfpages.Op_G 0.;
     Pdfpages.Op_re (0., 0., 1., 1.);
     Pdfpages.Op_m (0., 0.);
     Pdfpages.Op_l (1., 1.);
     Pdfpages.Op_m (0., 1.);
     Pdfpages.Op_l (1., 0.);
     Pdfpages.Op_S;
     Pdfpages.Op_Q]
```

Remove references to images from a graphics stream.

```
let rec remove_images_stream pdf resources prev = function
  | [] → rev prev
  | (Pdfpages.Op_Do name) as h :: t →
      if xobject_isimage pdf resources name
        then remove_images_stream pdf resources (substitute @ prev) t
        else remove_images_stream pdf resources (h :: prev) t
  | Pdfpages.InlineImage _ :: t →
      remove_images_stream pdf resources (substitute @ prev) t
  | h :: t →
      remove_images_stream pdf resources (h :: prev) t
```

Remove images from a page.

```
let remove_images_page pdf page =
  let content' =
    remove_images_stream pdf page.Pdfdoc.resources []
      (Pdfpages.parse_operators pdf page.Pdfdoc.resources page.Pdfdoc.content)
  in
    {page with
      Pdfdoc.content =
        (let stream = Pdfpages.stream_of_ops content' in
          Pdfcodec.encode_pdfstream pdf Pdfcodec.Flate stream;
          [stream]);
      Pdfdoc.resources =
        remove_image_xobjects pdf page.Pdfdoc.resources}
```

Remove images from all pages in a document.

```
let remove_images pdf =
  let pages = Pdfdoc.pages_of_pagetree pdf in
    let pages' = map (remove_images_page pdf) pages in
      let pdf, pagetree_num = Pdfdoc.add_pagetree pages' pdf in
        let pdf = Pdfdoc.add_root pagetree_num [] pdf in
          Pdf.remove_unreferenced pdf;
          pdf
```

Read command line arguments and call *remove_images*

```
let _ =
  match Array.to_list Sys.argv with
  | [_; in_file; out_file] →
      begin try
        let ch = open_in_bin in_file in
          let pdf = Pdfread.pdf_of_channel None ch in
            Pdfwrite.pdf_to_file (remove_images pdf) out_file;
            close_in ch
      with
        err →
          Printf.printf "Failed to produce output.\n%s\n\n" (Printexc.to_string err);
          exit 1
      end
  | _ →
      print_string "Syntax:  pdfdraft <input> <output>\n\n"; exit 1
```

Index