1. This is yamlpp, a very simple HTML preprocesseur inspired by Nicolas Thiery's htmlpp (see `htmlpp.sourceforge.net`). This is a simpler version with only languages, `#def` and `#ifdef` features.

```
{
open Printf
open Lexing
```

2. Command line options: selected language and output channel. Output is set to standard output by default. It may be set to a file with *set_output_to_file*. *close_output* closes the output channel if different from the standard output. *output_string* prints a string on the output channel.

```
let lang = ref "fr"
let out_channel = ref stdout

let set_output_to_file f =
  out_channel := open_out f

let close_output () =
  if !out_channel ≢ stdout then close_out !out_channel

let output_string s = Pervasives.output_string !out_channel s
```

3. Macros are stored in the hash table *macros*.

```
let macros = (Hashtbl.create 97 : (string,string) Hashtbl.t)

let add_macro = Hashtbl.add macros

let find_macro m =
  try Hashtbl.find macros m
  with Not_found → eprintf "***␣warning:␣undefined␣macro␣%s\n" m; ""

let is_macro = Hashtbl.mem macros
```

4. Predefined macros.

```
let _ =
  add_macro "yamlpp"
  "<a␣href=\"http://www.lri.fr/˜filliatr/yamlpp.<#language>.html\">yamlpp</a>"

let _ =
  let tm = Unix.localtime (Unix.time ()) in
  let d = tm.Unix.tm_mday
  and m = succ tm.Unix.tm_mon
  and y = 1900 + tm.Unix.tm_year in
  add_macro "date" (sprintf "%d/%d/%d" d m y)
```

5. Buffer for macros' definitions.

```
let mbuf = Buffer.create 1024
}
```

6. Regular expressions shortcuts.

```
let space = [' ' '\t' '\n']
let ident = ['0'-'9' 'a'-'z' 'A'-'Z' '_' '-']+
let lang = "fr" | "en" | "it" | "de"
```

7. The entry point of the filter is *process*. If we encounter a `#def` then we read the body in buffer *mbuf* with entry *def_body* and we store the new macro with *add_macro*, possibly hiding a previous macro with same name. If we encounter a `#ifdef` then we skip everything until `#/ifdef`.

```
rule process = parse
  | "<#def" space+ (ident as m) space* ">"
      { Buffer.clear mbuf;
        def_body lexbuf;
        add_macro m (Buffer.contents mbuf);
        process lexbuf }
  | "<#ifdef" space+ (ident as m) space* ">"
      { if ¬ (is_macro m) then skip_until "ifdef" 0 lexbuf;
        process lexbuf }
  | "<#" lang ">"
      { let s = lexeme lexbuf in
        let l = String.sub s 2 (String.length s − 3) in
        if l ≠ !lang then skip_until l 0 lexbuf;
        process lexbuf }
  | "</#" ident ">"
      { process lexbuf }
  | "<#" ident ">"
      { let s = lexeme lexbuf in
        let m = String.sub s 2 (String.length s − 3) in
        let body = find_macro m in
        let lb = from_string body in
        process lb;
        process lexbuf }
  | eof
      { () }
  | _
      { output_string (lexeme lexbuf); process lexbuf }
```

8. Reads a macro body and stores it in buffer *mbuf*.

```
and def_body = parse
  | "</#def>"
      { () }
  | eof
      { eprintf "Error:␣Unterminated␣macro␣definition\n"; exit 1}
  | _
      { Buffer.add_string mbuf (lexeme lexbuf); def_body lexbuf }
```

9. Skips input until a closing tag with name !*skip_tag*.

```
and skip_until skip_tag level = parse
  | "<#" (ident as tag) [^'>']* ">"
      { let level = if tag = skip_tag then level + 1 else level in
        skip_until skip_tag level lexbuf
      }
  | "</#" (ident as tag) ">"
      { if tag = skip_tag then
            if level = 0 then () else skip_until skip_tag (level − 1) lexbuf
        else
            skip_until skip_tag level lexbuf
      }
  | eof
      { eprintf "***␣warning:␣couldn't␣find␣end␣of␣flag␣%s\n" skip_tag;
        flush stderr }
  | _
      { skip_until skip_tag level lexbuf }

{
```

10. Generated file message.

```
let yamlpp_message () =
  output_string "
   <!--␣This␣document␣was␣automatically␣generated␣with␣yamlpp
   ␣␣␣␣␣(see␣http://www.lri.fr/~filliatr/yamlpp.en.html),
   ␣␣␣␣␣with␣the␣following␣command:\n";
  Array.iter (fun s → output_string (s ^ "␣")) Sys.argv;
  output_string "␣-->\n\n"
```

11. Processing a file is just calling *process*.

```
let process_file f =
  if ¬ (Sys.file_exists f) then begin
    eprintf "Error:␣%s:␣no␣such␣file\n" f; exit 1
  end;
  let c = open_in f in
  let lb = from_channel c in
  process lb;
  close_in c
```

12. Usage with minimal documentation.

```
let usage () =
  prerr_endline "usage:␣yamlpp␣[-l␣lang]␣[-o␣file]␣files";
  prerr_endline "";
  prerr_endline "Languages:␣␣␣␣␣␣␣␣␣<#fr>␣...␣</#fr>␣<#en>␣...␣</#en>";
  prerr_endline "Macro␣definition:␣␣<#def␣m>␣...␣</#def>";
  prerr_endline "Macro␣use:␣␣␣␣␣␣␣␣␣<#m>";
  prerr_endline "Macro␣test:␣␣␣␣␣␣␣␣<#ifdef␣m>␣...␣</#ifdef>";
  prerr_endline "Predefined␣macros:␣<#language>␣<#date>␣<#yamlpp>";
  prerr_endline "";
  prerr_endline "Copyright␣(C)␣2001␣Jean-Christophe␣FILLIÂTRE";
  exit 1
```

13. Parsing of the command line.

```
let rec parse = function
  | [] → []
  | ("-h" | "-?" | "-help" | "--help") :: _ → usage ()
  | ("-l" | "--language") :: l :: args → lang := l; parse args
  | ("-l" | "--language") :: [] → usage ()
  | ("-o" | "--output") :: f :: args → set_output_to_file f; parse args
  | ("-o" | "--output") :: [] → usage ()
  | f :: args → f :: (parse args)
```

14. Main program.

```
let main () =
  let files = parse (List.tl (Array.to_list Sys.argv)) in
  add_macro "language" !lang;
  yamlpp_message ();
  List.iter process_file files;
  close_output ()
```

$\mathsf{let}\ \_ \ = \ Printexc.catch\ main\ ()$

}

Index