1. ocamlwc. Counts the lines of code and comments in an ocaml source. It assumes the files to be lexically well-formed.

2. Command-line options.

```
let code_only = ref false
let percentage = ref false
let skip_header = ref true
let all_files = ref false
```

3. Counters. *clines* counts the number of code lines of the current file, and *dlines* the number of comment lines. *tclines* and *tdlines* are the corresponding totals.

```
let clines = ref 0
let dlines = ref 0

let tclines = ref 0
let tdlines = ref 0

let comment_depth = ref 0

let reset_counters () =
  clines := 0; dlines := 0; comment_depth := 0

let update_totals () =
  tclines := !tclines + !clines; tdlines := !tdlines + !dlines
```

4. Print results.

```
let print_line cl dl fo =
  Printf.printf "%7d" cl;
  if not !code_only then Printf.printf " %7d" dl;
  (match fo with Some f -> Printf.printf " %s" f | _ -> ());
  if !percentage then begin
    let s = cl + dl in
    let p = if s > 0 then 100 * dl / s else 0 in
    Printf.printf " (%d%%)" p
  end;
  print_newline ()

let print_file fo = print_line !clines !dlines fo

let print_totals () = print_line !tclines !tdlines (Some "total")
```

5. Shortcuts for regular expressions. The *rcs* regular expression is used to skip the CVS infos possibly contained in some comments, in order not to consider it as documentation.

```
let space = [' ' '\t' '\r']
let character =
  "'" ([^ '\\' '\''] |
      '\\' (['\\' '\'' 'n' 't' 'b' 'r'] | ['0'-'9'] ['0'-'9'] ['0'-'9'])) "'"
let rcs_keyword =
  "Author" | "Date" | "Header" | "Id" | "Name" | "Locker" | "Log" |
  "RCSfile" | "Revision" | "Source" | "State"
let rcs = "\036" rcs_keyword [^ '$']* "\036"
let stars = "(*" '*'* "*)"
```

6. Lexer. The lexer is a line-driven automaton, with eight main states. The states are named according to the following rules: the last character is either i or o and tells wether we are inside (i) or outside (o) a comment; the previous characters indicate what has been seen so far on the current line: nothing (n), some code (c) and/or some documentation (d).

```
rule s_no = parse
  | "(*" { comment_depth := 1; s_ni lexbuf }
  | '"' { let n = string lexbuf in clines := !clines + n; s_co lexbuf }
  | space+ | stars | '\n' { s_no lexbuf }
  | character | _ { s_co lexbuf }
  | eof { () }

and s_co = parse
  | "(*" { comment_depth := 1; s_ci lexbuf }
  | '"' { let n = string lexbuf in clines := !clines + n; s_co lexbuf }
  | '\n' { incr clines; s_no lexbuf }
  | stars | character | _ { s_co lexbuf }
  | eof { incr clines }

and s_do = parse
  | "(*" { comment_depth := 1; s_di lexbuf }
  | '"' { let n = string lexbuf in clines := !clines + n; s_cdo lexbuf }
  | '\n' { incr dlines; s_no lexbuf }
  | character | _ { s_cdo lexbuf }
  | eof { incr dlines }

and s_cdo = parse
  | "(*" { comment_depth := 1; s_cdi lexbuf }
  | '"' { let n = string lexbuf in clines := !clines + n; s_cdo lexbuf }
  | '\n' { incr clines; incr dlines; s_no lexbuf }
  | character | _ { s_cdo lexbuf }
  | eof { incr clines; incr dlines }

and s_cdi = parse
  | "(*" { incr comment_depth; s_cdi lexbuf }
  | "*)" { decr comment_depth;
             if !comment_depth > 0 then s_cdi lexbuf else s_cdo lexbuf }
  | '\n' { incr clines; incr dlines; s_ni lexbuf }
```

```
  | '"' { let n = string lexbuf in dlines := !dlines + n; s_cdi lexbuf }
  | character | _ { s_cdi lexbuf }
  | eof { incr clines; incr dlines }

and s_ci = parse
  | "(*" { incr comment_depth; s_ci lexbuf }
  | "*)" { decr comment_depth;
            if !comment_depth > 0 then s_ci lexbuf else s_co lexbuf }
  | '\n' { incr clines; s_ni lexbuf }
  | '"' { let n = string lexbuf in dlines := !dlines + n; s_cdi lexbuf }
  | space+ | rcs { s_ci lexbuf }
  | character | _ { s_cdi lexbuf }
  | eof { incr clines }

and s_ni = parse
  | "(*" { incr comment_depth; s_ni lexbuf }
  | "*)" { decr comment_depth;
            if !comment_depth > 0 then s_ni lexbuf else s_no lexbuf }
  | '\n' { s_ni lexbuf }
  | '"' { let n = string lexbuf in dlines := !dlines + n; s_di lexbuf }
  | space+ | rcs { s_ni lexbuf }
  | character | _ { s_di lexbuf }
  | eof { () }

and s_di = parse
  | "(*" { incr comment_depth; s_di lexbuf }
  | "*)" { decr comment_depth;
            if !comment_depth > 0 then s_di lexbuf else s_do lexbuf }
  | '\n' { incr dlines; s_ni lexbuf }
  | '"' { let n = string lexbuf in dlines := !dlines + n; s_di lexbuf }
  | character | _ { s_di lexbuf }
  | eof { incr dlines }
```

7. The entry *string* reads a string until its end and returns the number of newlines it contains.

```
and string = parse
  | '"' { 0 }
  | '\\' ('\\' | 'n' | '"') { string lexbuf }
  | '\n' { succ (string lexbuf) }
  | _ { string lexbuf }
  | eof { 0 }
```

8. The following entry *read_header* is used to skip the possible header at the beggining of files (unless option `-e` is specified). It stops whenever it encounters an empty line or any character outside a comment. In this last case, it correctly resets the lexer position on that character (decreasing *lex_curr_pos* by 1).

```
and read_header = parse
  | "(*" { skip_header_comment lexbuf; skip_until_nl lexbuf;
            read_header lexbuf }
  | "\n" { () }
  | space+ { read_header lexbuf }
  | _ { lexbuf.lex_curr_pos <- lexbuf.lex_curr_pos - 1 }
  | eof { () }

and skip_header_comment = parse
  | "*)" { () }
  | _ { skip_header_comment lexbuf }
  | eof { () }

and skip_until_nl = parse
  | '\n' { () }
  | _ { skip_until_nl lexbuf }
  | eof { () }
```

9. The following hash table *given_files* stores all the files given on the command line. Then *generated_file* tests whether a file is generated by looking for a possible source file in this table (e.g. a file with suffix `.ml` is considered to be generated if there is another file with the same path and name but suffix `.mly`).

```
let given_files = Hashtbl.create 97

let generated_file f =
  let test actual exists =
    (check_suffix f actual) &&
    (Hashtbl.mem given_files ((chop_suffix f actual) ^ exists))
  in
  (test ".ml" ".mll") || (test ".mli" ".mly") ||
  (test ".ml" ".mly") || (test ".ml" ".ml4")
```

10. Processing files and channels.

```
let process_channel ch =
  let lb = Lexing.from_channel ch in
  reset_counters ();
  if !skip_header then read_header lb;
  s_no lb

let process_file f =
  if (not !all_files) && (generated_file f) then begin
    flush stdout; eprintf "        skipped %s\n" f; flush stderr
  end else
    try
      let ch = open_in f in
      process_channel ch;
```

```
      close_in ch;
      print_file (Some f);
      update_totals ()
    with
      | Sys_error "Is␣a␣directory" ->
          flush stdout; eprintf "ocamlwc:␣%s:␣Is␣a␣directory\n" f; flush stderr
      | Sys_error s ->
          flush stdout; eprintf "ocamlwc:␣%s\n" s; flush stderr
```

11. Parsing of the command line.

```
let usage () =
  prerr_endline "usage:␣ocamlwc␣[options]␣[files]";
  prerr_endline "Options␣are:";
  prerr_endline "␣␣-p␣␣␣print␣percentage␣of␣documentation";
  prerr_endline "␣␣-c␣␣␣print␣only␣the␣code␣size";
  prerr_endline "␣␣-e␣␣␣(everything)␣do␣not␣skip␣headers";
  prerr_endline "␣␣-a␣␣␣(all)␣do␣not␣skip␣generated␣files";
  exit 1

let rec parse = function
  | [] -> []
  | ("-h" | "-?" | "-help" | "--help") :: _ -> usage ()
  | ("-c" | "--code-only") :: args -> code_only := true; parse args
  | ("-p" | "--percentage") :: args -> percentage := true; parse args
  | ("-e" | "--header") :: args -> skip_header := false; parse args
  | ("-a" | "--all") :: args -> all_files := true; parse args
  | f :: args -> f :: (parse args)
```

12. Main program.

```
let main () =
  let files = parse (List.tl (Array.to_list Sys.argv)) in
  List.iter (fun f -> Hashtbl.add given_files f ()) files;
  match files with
    | [] -> process_channel stdin; print_file None
    | [f] -> process_file f
    | _ -> List.iter process_file files; print_totals ()

let _ = Printexc.catch main ()
```