```
bash-9$ rlwrap ocaml
        OCaml version 4.09.0

(* $Id: .ocamlinit,v 1.6 2019-01-24 18:40:26-08 - - $ *)
\033[K# \033[K# \033[K\033[?1034h# #use "using";;# \033[K#use "using";;
val rcs : string = "(* $Id: using,v 1.3 2019-01-24 17:15:07-08 - - $ *)"
(* $Id: using,v 1.3 2019-01-24 17:15:07-08 - - $ *)
- : unit = ()
module Absyn :
  sig
    type linenr = int
    type ident = string
    type label = string
    type number = float
    type oper = string
    and memref = Arrayref of ident * expr | Variable of ident
    and expr =
        Number of number
      | Memref of memref
      | Unary of oper * expr
      | Binary of oper * expr * expr
    type printable = Printexpr of expr | String of string
    type stmt =
        Dim of ident * expr
      | Let of memref * expr
      | Goto of label
      | If of expr * label
      | Print of printable list
      | Input of memref list
    type progline = linenr * label option * stmt option
    type program = progline list
  end
module Etc :
  sig
    val execname : string
    val exit_status_ref : int ref
    val quit : unit -> unit
    val eprint_list : string list -> unit
    val warn : string list -> unit
    val die : string list -> unit
    val syntax_error : Lexing.position -> string list -> unit
    val usage_exit : string list -> unit
    val buffer : string list ref
    val read_number : unit -> float
  end
\033[Kmodule Parser :
  sig
    type token =
        RELOP of string
      | EQUAL of string
      | ADDOP of string
      | MULOP of string
      | POWOP of string
      | IDENT of string
      | NUMBER of string
      | STRING of string
      | COLON
      | COMMA
```

```
          | LPAR
          | RPAR
          | LSUB
          | RSUB
          | EOL
          | EOF
          | DIM
          | LET
          | GOTO
          | IF
          | PRINT
          | INPUT
      val linenr : unit -> int
      val syntax : unit -> unit
      val yytransl_const : int array
      val yytransl_block : int array
      val yylhs : string
      val yylen : string
      val yydefred : string
      val yydgoto : string
      val yysindex : string
      val yyrindex : string
      val yygindex : string
      val yytablesize : int
      val yytable : string
      val yycheck : string
      val yynames_const : string
      val yynames_block : string
      val yyact : (Parsing.parser_env -> Obj.t) array
      val yytables : Parsing.parse_tables
      val program : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Absyn.pro
gram
    end
module Scanner :
  sig
    val lexerror : Lexing.lexbuf -> unit
    val newline : Lexing.lexbuf -> unit
    val lexeme : Lexing.lexbuf -> string
    val __ocaml_lex_tables : Lexing.lex_tables
    val token : Lexing.lexbuf -> Parser.token
    val __ocaml_lex_token_rec : Lexing.lexbuf -> int -> Parser.token
  end
module Tables :
  sig
    type variable_table_t = (string, float) Hashtbl.t
    type array_table_t = (string, float array) Hashtbl.t
    type unary_fn_table_t = (string, float -> float) Hashtbl.t
    type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t
    type label_table_t = (string, Absyn.program) Hashtbl.t
    val variable_table : variable_table_t
    val array_table : array_table_t
    val unary_fn_table : unary_fn_table_t
    val binary_fn_table : binary_fn_table_t
    val label_table : label_table_t
    val init_label_table : Absyn.program -> unit
    val dump_label_table : unit -> unit
  end
module Dumper :
  sig
    val quote : string -> string
    val join : string -> string -> string list -> string
    val string_of_option : ('a -> string) -> 'a option -> string
    val string_of_ctor : string -> string list -> string
    val string_of_list : ('a -> string) -> 'a list -> string
    val string_of_printable : Absyn.printable -> string
    val string_of_memref : Absyn.memref -> string
    val string_of_expr : Absyn.expr -> string
    val string_of_stmt : Absyn.stmt -> string
    val dump_progline : int * string option * Absyn.stmt option -> unit
    val dump_program : Absyn.program -> unit
  end
module Interp :
  sig
    exception Unimplemented of string
    val no_expr : string -> 'a
    val no_stmt : string -> 'a -> 'b
    val want_dump : bool ref
    val eval_expr : Absyn.expr -> float
    val interpret : Absyn.program -> unit
    val interp_stmt : Absyn.stmt -> Absyn.program -> unit
    val interp_print : Absyn.printable list -> Absyn.program -> unit
    val interp_input : Absyn.memref list -> Absyn.program -> unit
    val interpret_program : Absyn.program -> unit
  end
module Main : sig val interpret_source : string -> unit end
- : unit = ()
# \033[K# \033[K# interpret_source ".score/00-hello-world.sb";;# \033[Kinterpret_source ".score/00-hello-world.sb";;
program: 1 None: None
program: 2 None: None
program: 3 None: None
program: 4 None: None
program: 5 None: Some (Print ([String ("\"Hello, World!\"")]))
 Hello, World!
- : unit = ()
# \033[K# \033[K# interpret_source ".score/41-eratosthenes.sb";;# \033[Kinterpret_source ".score/41-eratosthenes.sb";;
label_table: "punch" -> line 22
label_table: "primes" -> line 16
label_table: "init" -> line 9
label_table: "next" -> line 24
label_table: "loop" -> line 20
program: 1 None: None
program: 2 None: None
program: 3 None: Some (Let (Variable ("n"), Number (100.)))
program: 4 None: Some (Dim ("sieve", Memref (Variable ("n"))))
program: 5 None: None
program: 6 None: None
program: 7 None: None
program: 8 None: Some (Let (Variable ("i"), Number (2.)))
program: 9 Some ("init"): Some (Let (Arrayref ("sieve", Memref (Variable ("i"))), Number (1.)))
program: 10 None: Some (Let (Variable ("i"), Binary ("+", Memref (Variable ("i")), Number (1.))))
program: 11 None: Some (If (Binary ("<", Memref (Variable ("i")), Memref (Variable ("n"))), "init"))
```

```
program: 12 None: None
program: 13 None: None
program: 14 None: None
program: 15 None: Some (Let (Variable ("prime"), Number (2.)))
program: 16 Some ("primes"): Some (If (Binary ("=", Memref (Arrayref ("sieve", Memref (Variable ("prime")))), Number (0.)), "next"))
program: 17 None: Some (Print ([Printexpr (Memref (Variable ("prime")))]))
program: 18 None: Some (Let (Variable ("i"), Binary ("*", Memref (Variable ("prime")), Number (2.))))
program: 19 None: Some (Goto ("punch"))
program: 20 Some ("loop"): Some (Let (Arrayref ("sieve", Memref (Variable ("i"))), Number (0.)))
program: 21 None: Some (Let (Variable ("i"), Binary ("+", Memref (Variable ("i")), Memref (Variable ("prime")))))
program: 22 Some ("punch"): Some (If (Binary ("<", Memref (Variable ("i")), Memref (Variable ("n"))), "loop"))
program: 23 None: None
program: 24 Some ("next"): Some (Let (Variable ("prime"), Binary ("+", Memref (Variable ("prime")), Number (1.))))
program: 25 None: Some (If (Binary ("<=", Memref (Variable ("prime")), Memref (Variable ("n"))), "primes"))
Exception: Interp.Unimplemented "Let (memref, expr)".
# \033[K# \033[K#
\033[Kbash-10$ exit
exit

```