```
 1: (* $Id: absyn.mli,v 1.2 2019-01-24 17:42:06-08 - - $ *)
 2:
 3: (*
 4: * Abstract syntax definitions for SB.
 5: *)
 6:
 7: type linenr    = int
 8: type ident     = string
 9: type label     = string
10: type number    = float
11: type oper      = string
12:
13: type printable = Printexpr of expr
14:                | String of string
15:
16: and  memref    = Arrayref of ident * expr
17:                | Variable of ident
18:
19: and  expr      = Number of number
20:                | Memref of memref
21:                | Unary of oper * expr
22:                | Binary of oper * expr * expr
23:
24: type stmt      = Dim of ident * expr
25:                | Let of memref * expr
26:                | Goto of label
27:                | If of expr * label
28:                | Print of printable list
29:                | Input of memref list
30:
31: type progline  = linenr * label option * stmt option
32:
33: type program   = progline list
34:
```

```
 1: (* $Id: etc.mli,v 1.1 2019-01-24 15:47:38-08 - - $ *)
 2:
 3: (*
 4: * Main program and system access.
 5: *)
 6:
 7: val warn : string list -> unit
 8:
 9: val die : string list -> unit
10:
11: val syntax_error : Lexing.position -> string list -> unit
12:
13: val usage_exit : string list -> unit
14:
15: val read_number : unit -> float
16:
```

```
 1: (* $Id: etc.ml,v 1.1 2019-01-24 15:47:38-08 - - $ *)
 2:
 3: let execname = Filename.basename Sys.argv.(0)
 4:
 5: let exit_status_ref = ref 0
 6:
 7: let quit () =
 8:     if !Sys.interactive
 9:     then Printf.printf "Quit: exit %d\n%!" !exit_status_ref
10:     else exit !exit_status_ref
11:
12: let eprint_list message =
13:     (exit_status_ref := 1;
14:      flush_all ();
15:      List.iter prerr_string message;
16:      prerr_newline ();
17:      flush_all ())
18:
19: let warn message = eprint_list (execname :: ": " :: message)
20:
21: let die message = (warn message; quit ())
22:
23: let syntax_error position message =
24:     warn (position.Lexing.pos_fname :: ": "
25:             :: string_of_int position.Lexing.pos_lnum :: ": "
26:             :: message)
27:
28: let usage_exit message =
29:     (eprint_list ("Usage: " :: execname :: " " :: message); quit ())
30:
31: let buffer : string list ref = ref []
32:
33: let rec read_number () = match !buffer with
34:     | head::tail -> (buffer := tail;
35:                      try float_of_string head
36:                      with Failure _ -> nan)
37:     | [] -> let line = input_line stdin
38:             in (buffer := Str.split (Str.regexp "[ \\t]+") line;
39:                 read_number ())
40:
```

```
 1: (* Created Thu Jan 24 19:14:14 PST 2019 *)
 2:
 3: type variable_table_t = (string, float) Hashtbl.t
 4: val variable_table : variable_table_t
 5:
 6: type array_table_t = (string, float array) Hashtbl.t
 7: val array_table : array_table_t
 8:
 9: type unary_fn_table_t = (string, float -> float) Hashtbl.t
10: val unary_fn_table : unary_fn_table_t
11:
12: type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t
13: val binary_fn_table : binary_fn_table_t
14:
15: type label_table_t = (string, Absyn.program) Hashtbl.t
16: val label_table : label_table_t
17: val init_label_table : Absyn.program -> unit
18: val dump_label_table : unit -> unit
```

```
 1: (* $Id: tables.ml,v 1.3 2019-01-24 17:08:37-08 - - $ *)
 2:
 3: type variable_table_t = (string, float) Hashtbl.t
 4: let variable_table : variable_table_t = Hashtbl.create 16
 5: let _ = List.map (fun (label, value) ->
 6:                   Hashtbl.add variable_table label value)
 7:                  ["e"  , exp 1.0;
 8:                   "eof", 0.0;
 9:                   "pi" , acos ~-.1.0;
10:                   "nan", nan]
11:
12: type array_table_t = (string, float array) Hashtbl.t
13: let array_table : array_table_t = Hashtbl.create 16
14:
15: type unary_fn_table_t = (string, float -> float) Hashtbl.t
16: let unary_fn_table : unary_fn_table_t = Hashtbl.create 16
17: let _ = List.map (fun (label, value) ->
18:                   Hashtbl.add unary_fn_table label value)
19:                  ["+"    , (~+.);
20:                   "-"    , (~-.);
21:                   "abs"  , abs_float;
22:                   "acos" , acos;
23:                   "asin" , asin;
24:                   "atan" , atan;
25:                   "ceil" , ceil;
26:                   "cos"  , cos;
27:                   "exp"  , exp;
28:                   "floor", floor;
29:                   "log"  , log;
30:                   "log10", log10;
31:                   "log2" , (fun x -> log x /. log 2.0);
32:                   "round", (fun x -> floor (x +. 0.5));
33:                   "sin"  , sin;
34:                   "sqrt" , sqrt;
35:                   "tan"  , tan]
36:
37: type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t
38: let binary_fn_table : binary_fn_table_t = Hashtbl.create 16
39: let _ = List.map (fun (label, value) ->
40:                   Hashtbl.add binary_fn_table label value)
41:                  ["+", (+.);
42:                   "-", (-.);
43:                   "*", ( *.);
44:                   "/", (/.);
45:                   "%", mod_float;
46:                   "^", ( ** )]
47:
```

```
48:
49: type label_table_t = (string, Absyn.program) Hashtbl.t
50: let label_table : label_table_t = Hashtbl.create 16
51:
52: let rec init_label_table program =
53:     let rec init program =  match program with
54:         | [] -> ()
55:         | (_, Some label, _)::rest ->
56:               (Hashtbl.add label_table label program; init rest)
57:         | _::rest -> init rest
58:     in (Hashtbl.reset label_table; init program)
59:
60: let dump_label_table () =
61:     let dump key value = match value with
62:         | [] -> ()
63:         | (line, _, _)::_ ->
64:           Printf.fprintf stderr
65:               "label_table: \"%s\" -> line %d\n%!" key line
66:     in Hashtbl.iter dump label_table
67:
```

```
 1: (* Created Thu Jan 24 19:14:14 PST 2019 *)
 2: val quote : string -> string
 3: val join : string -> string -> string -> string list -> string
 4: val str_opt : ('a -> string) -> 'a option -> string
 5: val str_ctor : string -> string list -> string
 6: val str_list : ('a -> string) -> 'a list -> string
 7: val str_printable : Absyn.printable -> string
 8: val str_memref : Absyn.memref -> string
 9: val str_expr : Absyn.expr -> string
10: val str_stmt : Absyn.stmt -> string
11: val dump_progline : int * string option * Absyn.stmt option -> unit
12: val dump_program : Absyn.program -> unit
```

```
 1: (* $Id: dumper.ml,v 1.10 2019-01-24 19:14:14-08 - - $ *)
 2:
 3: open Absyn
 4:
 5: let quote string =
 6:     let regex = Str.regexp "\""
 7:     and subst _ = "\\\""
 8:     in  "\"" ^ Str.global_substitute regex subst string ^ "\""
 9:
10: let join start sep stop list =
11:     let rec join' list' = match list' with
12:         | [] -> stop
13:         | [unit] -> unit ^ stop
14:         | head::tail -> head ^ sep ^ " " ^ join' tail
15:      in match list with
16:         | [] -> start ^ stop
17:         | _::_ -> start ^ join' list
18:
19: let str_opt str_fn item = match item with
20:     | None -> "None"
21:     | Some thing -> "Some (" ^ str_fn thing ^ ")"
22:
23: let str_ctor ctor args = join (ctor ^ " (") "," ")" args
24:
25: let str_list str_fn list = join "[" ";" "]" (List.map str_fn list)
26:
```

```
27:
28: let rec str_printable printable = match printable with
29:     | Printexpr expr -> str_ctor "Printexpr" [str_expr expr]
30:     | String string -> str_ctor "String" [quote string]
31:
32: and str_memref memref = match memref with
33:     | Arrayref (ident, expr) ->
34:           str_ctor "Arrayref" [quote ident; str_expr expr]
35:     | Variable ident -> str_ctor "Variable" [quote ident]
36:
37: and str_expr expr = match expr with
38:     | Number number -> str_ctor "Number" [string_of_float number]
39:     | Memref memref -> str_ctor "Memref" [str_memref memref]
40:     | Unary (oper, expr) -> str_ctor "Unary" [quote oper; str_expr expr]
41:     | Binary (oper, expr1, expr2) ->
42:           str_ctor "Binary" [quote oper; str_expr expr1; str_expr expr2]
43:
44: let str_stmt (stmt: stmt) = match stmt with
45:     | Dim (ident, expr) ->
46:           str_ctor "Dim" [quote ident ^ ", " ^ str_expr expr]
47:     | Let (memref, expr) ->
48:           str_ctor "Let" [str_memref memref; str_expr expr]
49:     | Goto label -> str_ctor "Goto" [quote label]
50:     | If (expr, label) -> str_ctor "If" [str_expr expr; quote label]
51:     | Print printable'list ->
52:           str_ctor "Print" [str_list str_printable printable'list]
53:     | Input memref'list ->
54:           str_ctor "Input" [str_list str_memref memref'list]
55:
56: let dump_progline (linenr, label'opt, stmt'opt) =
57:     Printf.fprintf stderr "%3d %s: %s\n%!" linenr
58:         (str_opt (fun x -> x) label'opt)
59:         (str_opt str_stmt stmt'opt)
60:
61: let dump_program (program : program) =
62:     List.iter dump_progline program
63:
```

```
 1: (* $Id: interp.mli,v 1.5 2019-01-24 17:08:37-08 - - $ *)
 2:
 3: (*
 4: * Interpreter for Silly Basic
 5: *)
 6:
 7: val want_dump : bool ref
 8:
 9: val interpret_program : Absyn.program -> unit
10:
```

```
 1: (* $Id: interp.ml,v 1.6 2019-01-24 19:14:14-08 - - $ *)
 2:
 3: open Absyn
 4:
 5: exception Unimplemented of string
 6: let unimpl reason = raise (Unimplemented reason)
 7:
 8: let want_dump = ref false
 9:
10: let rec eval_expr (expr : Absyn.expr) : float = match expr with
11:     | Number number -> number
12:     | Memref memref -> unimpl "eval_expr Memref"
13:     | Unary (oper, expr) -> unimpl "eval_expr Unary"
14:     | Binary (oper, expr1, expr2) -> unimpl "eval_expr Binary"
15:
16: let interp_print (print_list : Absyn.printable list) =
17:     let print_item item =
18:         (print_string " ";
19:          match item with
20:          | String string ->
21:            let regex = Str.regexp "\"\\(.*\\)\""
22:            in print_string (Str.replace_first regex "\\1" string)
23:          | Printexpr expr ->
24:            print_float (eval_expr expr))
25:     in (List.iter print_item print_list; print_newline ())
26:
27: let interp_input (memref_list : Absyn.memref list) =
28:     let input_number memref =
29:         try  let number = Etc.read_number ()
30:              in (print_float number; print_newline ())
31:         with End_of_file ->
32:              (print_string "End_of_file"; print_newline ())
33:     in List.iter input_number memref_list
34:
35: let interp_stmt (stmt : Absyn.stmt) = match stmt with
36:     | Dim (ident, expr) -> unimpl "Dim (ident, expr)"
37:     | Let (memref, expr) -> unimpl "Let (memref, expr)"
38:     | Goto labsl -> unimpl "Goto labsl"
39:     | If (expr, label) -> unimpl "If (expr, label)"
40:     | Print print_list -> interp_print print_list
41:     | Input memref_list -> interp_input memref_list
42:
43: let rec interpret (program : Absyn.program) = match program with
44:     | [] -> ()
45:     | firstline::otherlines -> match firstline with
46:       | _, _, None -> interpret otherlines
47:       | _, _, Some stmt -> (interp_stmt stmt; interpret otherlines)
48:
49: let interpret_program program =
50:     (Tables.init_label_table program;
51:      if !want_dump then Tables.dump_label_table ();
52:      if !want_dump then Dumper.dump_program program;
53:      interpret program)
54:
```

```
 1: (* $Id: main.ml,v 1.1 2019-01-24 15:47:38-08 - - $ *)
 2:
 3: (*
 4: * Main program reads a file and prints to stdout.
 5: *)
 6:
 7: let interpret_source filename =
 8:     try (let sourcefile =
 9:              if filename = "-"
10:              then stdin
11:              else open_in filename in
12:          let lexbuf = Lexing.from_channel sourcefile in
13:          let abstract_syntax = Parser.program Scanner.token lexbuf in
14:          Interp.interpret_program abstract_syntax)
15:     with Sys_error (string) -> Etc.die [string]
16:
17: let _ = if !Sys.interactive
18:         then ()
19:         else match Array.length Sys.argv with
20:              | 1 -> interpret_source "-"
21:              | 2 -> interpret_source Sys.argv.(1)
22:              | _ -> Etc.usage_exit ["[filename.sb]"]
23:
```

```
 1: /* $Id: parser.mly,v 1.1 2019-01-24 15:47:38-08 - - $ */
 2:
 3: %{
 4:
 5: let linenr () = (symbol_start_pos ()).Lexing.pos_lnum
 6:
 7: let syntax () = Etc.syntax_error (symbol_start_pos ()) ["syntax error"]
 8:
 9: %}
10:
11: %token <string> RELOP EQUAL ADDOP MULOP POWOP
12: %token <string> IDENT NUMBER STRING
13: %token COLON COMMA LPAR RPAR LSUB RSUB EOL EOF
14: %token DIM LET GOTO IF PRINT INPUT
15:
16: %type <Absyn.program> program
17:
18: %start program
19:
20: %%
21:
22: program    : stmt_list EOF             {List.rev $1}
23:
24: stmt_list  : stmt_list stmt EOL        {$2::$1}
25:            | stmt_list error EOL       {syntax (); $1}
26:            |                           {[]}
27:
28: stmt       : label action              {(linenr (), Some $1, Some $2)}
29:            | action                    {(linenr (), None, Some $1)}
30:            | label                     {(linenr (), Some $1, None)}
31:            |                           {(linenr (), None, None)}
32:
33: label      : IDENT COLON               {$1}
34:
35: action     : DIM IDENT LSUB expr RSUB {Absyn.Dim ($2, $4)}
36:            | LET memref EQUAL expr     {Absyn.Let ($2, $4)}
37:            | GOTO IDENT                {Absyn.Goto $2}
38:            | IF relexpr GOTO IDENT     {Absyn.If ($2, $4)}
39:            | PRINT print_list          {Absyn.Print $2}
40:            | PRINT                     {Absyn.Print ([])}
41:            | INPUT input_list          {Absyn.Input $2}
42:
43: print_list : print COMMA print_list    {$1::$3}
44:            | print                     {[$1]}
45:
46: print      : expr                      {Absyn.Printexpr $1}
47:            | STRING                    {Absyn.String $1}
48:
49: input_list : memref COMMA input_list   {$1::$3}
50:            | memref                    {[$1]}
51:
```

```
   52:
   53: memref     : IDENT                    {Absyn.Variable $1}
   54:            | IDENT LSUB expr RSUB     {Absyn.Arrayref ($1, $3)}
   55:
   56: relexpr    : expr RELOP expr          {Absyn.Binary ($2, $1, $3)}
   57:            | expr EQUAL expr          {Absyn.Binary ($2, $1, $3)}
   58:
   59: expr       : expr ADDOP term          {Absyn.Binary ($2, $1, $3)}
   60:            | term                     {$1}
   61:
   62: term       : term MULOP factor        {Absyn.Binary ($2, $1, $3)}
   63:            | factor                   {$1}
   64:
   65: factor     : primary POWOP factor     {Absyn.Binary ($2, $1, $3)}
   66:            | primary                  {$1}
   67:
   68: primary    : LPAR expr RPAR           {$2}
   69:            | ADDOP primary            {Absyn.Unary ($1, $2)}
   70:            | NUMBER                   {Absyn.Number (float_of_string $1)
}
   71:            | memref                   {Absyn.Memref $1}
   72:            | IDENT LPAR expr RPAR     {Absyn.Unary ($1, $3)}
   73:
```

```
 1: (* $Id: scanner.mll,v 1.1 2019-01-24 15:47:38-08 - - $ *)
 2:
 3: {
 4:
 5: let lexerror lexbuf =
 6:     Etc.syntax_error (Lexing.lexeme_start_p lexbuf)
 7:             ["invalid character `" ^ (Lexing.lexeme lexbuf) ^ "'"]
 8:
 9: let newline lexbuf =
10:     let incr pos =
11:         {pos with Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
12:                   Lexing.pos_bol = pos.Lexing.pos_cnum}
13:     in  (lexbuf.Lexing.lex_start_p <- incr lexbuf.Lexing.lex_start_p;
14:          lexbuf.Lexing.lex_curr_p <- incr lexbuf.Lexing.lex_curr_p)
15:
16: let lexeme = Lexing.lexeme
17:
18: }
19:
20: let letter          = ['a'-'z' 'A'-'Z' '_']
21: let digit           = ['0'-'9']
22: let fraction        = (digit+ '.'? digit* | '.' digit+)
23: let exponent        = (['E' 'e'] ['+' '-']? digit+)
24:
25: let comment         = ('#' [^'\n']*)
26: let ident           = (letter (letter | digit)*)
27: let number          = (fraction exponent?)
28: let string          = '"' [^'\n' '"']* '"'
29:
```

```
30:
31: rule token     = parse
32:      | eof            { Parser.EOF }
33:      | [' ' '\t']     { token lexbuf }
34:      | comment        { token lexbuf }
35:      | "\n"           { newline lexbuf; Parser.EOL }
36:      | ":"            { Parser.COLON }
37:      | ","            { Parser.COMMA }
38:      | "("            { Parser.LPAR }
39:      | ")"            { Parser.RPAR }
40:      | "["            { Parser.LSUB }
41:      | "]"            { Parser.RSUB }
42:      | "="            { Parser.EQUAL (lexeme lexbuf) }
43:      | "<>"           { Parser.RELOP (lexeme lexbuf) }
44:      | "<"            { Parser.RELOP (lexeme lexbuf) }
45:      | "<="           { Parser.RELOP (lexeme lexbuf) }
46:      | ">"            { Parser.RELOP (lexeme lexbuf) }
47:      | ">="           { Parser.RELOP (lexeme lexbuf) }
48:      | "+"            { Parser.ADDOP (lexeme lexbuf) }
49:      | "-"            { Parser.ADDOP (lexeme lexbuf) }
50:      | "*"            { Parser.MULOP (lexeme lexbuf) }
51:      | "/"            { Parser.MULOP (lexeme lexbuf) }
52:      | "%"            { Parser.MULOP (lexeme lexbuf) }
53:      | "^"            { Parser.POWOP (lexeme lexbuf) }
54:      | "dim"          { Parser.DIM }
55:      | "goto"         { Parser.GOTO }
56:      | "if"           { Parser.IF }
57:      | "input"        { Parser.INPUT }
58:      | "let"          { Parser.LET }
59:      | "print"        { Parser.PRINT }
60:      | number         { Parser.NUMBER (lexeme lexbuf) }
61:      | string         { Parser.STRING (lexeme lexbuf) }
62:      | ident          { Parser.IDENT (lexeme lexbuf) }
63:      | _              { lexerror lexbuf; token lexbuf }
64:
```

```
# $Id: Makefile,v 1.5 2019-01-24 17:51:18-08 - - $

#
# General useful macros
#

MKFILE     = Makefile
MAKEFLAGS += --no-builtin-rules
DEPSFILE   = ${MKFILE}.deps
NOINCLUDE  = ci clean spotless
NEEDINCL   = ${filter ${NOINCLUDE}, ${MAKECMDGOALS}}
GMAKE      = ${MAKE} --no-print-directory

#
# File list macros
#

EXECBIN    = sbinterp
OBJCMO     = etc.cmo parser.cmo scanner.cmo tables.cmo \
             dumper.cmo interp.cmo main.cmo
OBJCMI     = ${OBJCMO:.cmo=.cmi} absyn.cmi
OBJBIN     = ${OBJCMO:.cmo=.o}
MLSOURCE   = absyn.mli etc.mli etc.ml tables.mli tables.ml \
             dumper.mli dumper.ml interp.mli interp.ml main.ml
GENSOURCE  = dumper.mli tables.mli parser.mli parser.ml scanner.ml
GENFILES   = ${GENSOURCE} parser.output ${DEPSFILE}
OTHERFILES = ${MKFILE} ${DEPSFILE} using .ocamlinit
ALLSOURCES = ${MLSOURCE} parser.mly scanner.mll ${OTHERFILES}
LISTING    = Listing.ps

#
# General targets
#

all : ${EXECBIN}

${EXECBIN} : ${OBJCMO}
        ocamlc str.cma ${OBJCMO} -o ${EXECBIN}

%.cmi : %.mli
        ocamlc -c $<

%.cmo : %.ml
        ocamlc -c $<

%.ml : %.mll
        ocamllex $<

%.mli %.ml : %.mly
        ocamlyacc -v $<

```

```
52:
53: tables.mli : tables.ml absyn.cmi
54:         @ echo "(* Created $$(date) *)" >tables.mli
55:         ocamlc -i tables.ml | sed 's/^type/\n&/' >>tables.mli
56:
57: dumper.mli : dumper.ml absyn.cmi
58:         @ echo "(* Created $$(date) *)" >dumper.mli
59:         ocamlc -i dumper.ml >>dumper.mli
60:
61: #
62: # Misc targets
63: #
64:
65: clean :
66:         - rm ${OBJCMI} ${OBJCMO} ${OBJBIN} ${GENSOURCE}
67:
68: spotless : clean
69:         - rm ${EXECBIN} ${GENFILES} ${LISTING} ${LISTING:.ps=.pdf}
70:
71: ci : ${ALLSOURCES}
72:         cid + ${ALLSOURCES}
73:
74: deps : ${MLSOURCE} ${GENSOURCE}
75:         @ echo "# Created $$(date)" >${DEPSFILE}
76:         ocamldep ${MLSOURCE} ${GENSOURCE} >>${DEPSFILE}
77:
78: ${DEPSFILE} : tables.mli
79:         @touch ${DEPSFILE}
80:         ${GMAKE} deps
81:
82: lis : ${ALLSOURCES}
83:         mkpspdf ${LISTING} ${ALLSOURCES}
84:
85: again :
86:         ${GMAKE} spotless
87:         ${GMAKE} deps
88:         ${GMAKE} ci
89:         ${GMAKE} all
90:         ${GMAKE} lis
91:
92: ifeq "${NEEDINCL}" ""
93: include ${DEPSFILE}
94: endif
95:
```

```
 1: # Created Thu Jan 24 19:14:14 PST 2019
 2: absyn.cmi :
 3: etc.cmi :
 4: etc.cmo : etc.cmi
 5: etc.cmx : etc.cmi
 6: tables.cmi : absyn.cmi
 7: tables.cmo : absyn.cmi tables.cmi
 8: tables.cmx : absyn.cmi tables.cmi
 9: dumper.cmi : absyn.cmi
10: dumper.cmo : absyn.cmi dumper.cmi
11: dumper.cmx : absyn.cmi dumper.cmi
12: interp.cmi : absyn.cmi
13: interp.cmo : tables.cmi etc.cmi dumper.cmi absyn.cmi interp.cmi
14: interp.cmx : tables.cmx etc.cmx dumper.cmx absyn.cmi interp.cmi
15: main.cmo : scanner.cmo parser.cmi interp.cmi etc.cmi
16: main.cmx : scanner.cmx parser.cmx interp.cmx etc.cmx
17: dumper.cmi : absyn.cmi
18: tables.cmi : absyn.cmi
19: parser.cmi : absyn.cmi
20: parser.cmo : etc.cmi absyn.cmi parser.cmi
21: parser.cmx : etc.cmx absyn.cmi parser.cmi
22: scanner.cmo : parser.cmi etc.cmi
23: scanner.cmx : parser.cmx etc.cmx
```

```
 1: let rcs = "(* $Id: using,v 1.3 2019-01-24 17:15:07-08 - - $ *)";;
 2:
 3: print_endline rcs;;
 4:
 5: #load "str.cma";;
 6:
 7: #mod_use "absyn.mli";;
 8: #mod_use "etc.ml";;
 9:
10: #mod_use "parser.ml";;
11: #mod_use "scanner.ml";;
12:
13: #mod_use "tables.ml";;
14: #mod_use "dumper.ml";;
15:
16: #mod_use "interp.ml";;
17: #mod_use "main.ml";;
18:
19: open Interp;;
20: open Main;;
21:
22: want_dump := true;;
23:
```

```
 1: let rcs = "(* $Id: .ocamlinit,v 1.6 2019-01-24 18:40:26-08 - - $ *)";;
 2:
 3: print_endline rcs;;
 4:
 5: #load "str.cma";;
 6:
 7: #mod_use "absyn.mli";;
 8: #mod_use "etc.ml";;
 9:
10: #mod_use "parser.ml";;
11: #mod_use "scanner.ml";;
12:
13: #mod_use "tables.ml";;
14: #mod_use "dumper.ml";;
15:
16: #mod_use "interp.ml";;
17: #mod_use "main.ml";;
18:
19: open Interp;;
20: open Main;;
21:
22: want_dump := true;;
23:
```