```
 1: (* $Id: absyn.mli,v 1.3 2020-01-22 16:07:24-08 - - $ *)
 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: and  memref    = Arrayref of ident * expr
14:                | Variable of ident
15:
16: and  expr      = Number of number
17:                | Memref of memref
18:                | Unary of oper * expr
19:                | Binary of oper * expr * expr
20:
21: type printable = Printexpr of expr
22:                | String of string
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 1.3 2020-01-22 16:07:24-08 - - $ *)
 2:
 3: let execname = Filename.basename Sys.argv.(0)
 4:
 5: let exit_status_ref : int 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: (* Generated: Fri Jan 24 14:55:58 PST 2020 *)
 2: type variable_table_t = (string, float) Hashtbl.t
 3: type array_table_t = (string, float array) Hashtbl.t
 4: type unary_fn_table_t = (string, float -> float) Hashtbl.t
 5: type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t
 6: type label_table_t = (string, Absyn.program) Hashtbl.t
 7: val variable_table : variable_table_t
 8: val array_table : array_table_t
 9: val unary_fn_table : unary_fn_table_t
10: val binary_fn_table : binary_fn_table_t
11: val label_table : label_table_t
12: val init_label_table : Absyn.program -> unit
13: val dump_label_table : unit -> unit
```

```
 1: (* $Id: tables.ml,v 1.5 2019-01-29 17:26:15-08 - - $ *)
 2:
 3: type variable_table_t = (string, float) Hashtbl.t
 4: type array_table_t = (string, float array) Hashtbl.t
 5: type unary_fn_table_t = (string, float -> float) Hashtbl.t
 6: type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t
 7: type label_table_t = (string, Absyn.program) Hashtbl.t
 8:
 9: let variable_table : variable_table_t = Hashtbl.create 16
10: let _ = List.iter (fun (label, value) ->
11:                    Hashtbl.add variable_table label value)
12:                   ["e"  , exp 1.0;
13:                    "eof", 0.0;
14:                    "pi" , acos ~-.1.0;
15:                    "nan", nan]
16:
17: let array_table : array_table_t = Hashtbl.create 16
18:
19: let unary_fn_table : unary_fn_table_t = Hashtbl.create 16
20: let _ = List.iter (fun (label, value) ->
21:                    Hashtbl.add unary_fn_table label value)
22:                   ["+"    , (~+.);
23:                    "-"    , (~-.);
24:                    "abs"  , abs_float;
25:                    "acos" , acos;
26:                    "asin" , asin;
27:                    "atan" , atan;
28:                    "ceil" , ceil;
29:                    "cos"  , cos;
30:                    "exp"  , exp;
31:                    "floor", floor;
32:                    "log"  , log;
33:                    "log10", log10;
34:                    "log2" , (fun x -> log x /. log 2.0);
35:                    "round", (fun x -> floor (x +. 0.5));
36:                    "sin"  , sin;
37:                    "sqrt" , sqrt;
38:                    "tan"  , tan]
39:
40: let binary_fn_table : binary_fn_table_t = Hashtbl.create 16
41: let _ = List.iter (fun (label, value) ->
42:                    Hashtbl.add binary_fn_table label value)
43:                   ["+", (+.);
44:                    "-", (-.);
45:                    "*", ( *.);
46:                    "/", (/.);
47:                    "%", mod_float;
48:                    "^", ( ** )]
49:
```

```
50:
51: let label_table : label_table_t = Hashtbl.create 16
52:
53: let rec init_label_table program =
54:     let rec init program =  match program with
55:         | [] -> ()
56:         | (_, Some label, _)::rest ->
57:             (Hashtbl.add label_table label program; init rest)
58:         | _::rest -> init rest
59:     in (Hashtbl.reset label_table; init program)
60:
61: let dump_label_table () =
62:     let dump key value = match value with
63:         | [] -> ()
64:         | (line, _, _)::_ ->
65:           Printf.fprintf stderr
66:               "label_table: \"%s\" -> line %d\n%!" key line
67:     in Hashtbl.iter dump label_table
68:
```

```
 1: (* Generated: Fri Jan 24 14:55:58 PST 2020 *)
 2: val quote : string -> string
 3: val join : string -> string -> string -> string list -> string
 4: val string_of_option : ('a -> string) -> 'a option -> string
 5: val string_of_ctor : string -> string list -> string
 6: val string_of_list : ('a -> string) -> 'a list -> string
 7: val string_of_printable : Absyn.printable -> string
 8: val string_of_memref : Absyn.memref -> string
 9: val string_of_expr : Absyn.expr -> string
10: val string_of_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 1.16 2020-01-24 14:55:58-08 - - $ *)
 2:
 3: let quote string =
 4:     let regex = Str.regexp "\""
 5:     and subst _ = "\\\""
 6:     in  "\"" ^ Str.global_substitute regex subst string ^ "\""
 7:
 8: let join start sep stop list =
 9:     let rec join' list' = match list' with
10:         | [] -> stop
11:         | [unit] -> unit ^ stop
12:         | head::tail -> head ^ sep ^ " " ^ join' tail
13:     in match list with
14:         | [] -> start ^ stop
15:         | _::_ -> start ^ join' list
16:
17: let string_of_option str_fn item = match item with
18:     | None -> "None"
19:     | Some thing -> "Some (" ^ str_fn thing ^ ")"
20:
21: let string_of_ctor ctor args =
22:     join (ctor ^ " (") "," ")" args
23:
24: let string_of_list str_fn list =
25:     join "[" ";" "]" (List.map str_fn list)
26:
27: let rec string_of_printable printable = match printable with
28:     | Absyn.Printexpr expr ->
29:           string_of_ctor "Printexpr" [string_of_expr expr]
30:     | Absyn.String string ->
31:           string_of_ctor "String" [quote string]
32:
33: and string_of_memref memref = match memref with
34:     | Absyn.Arrayref (ident, expr) ->
35:           string_of_ctor "Arrayref" [quote ident; string_of_expr expr]
36:     | Absyn.Variable ident -> string_of_ctor "Variable" [quote ident]
37:
38: and string_of_expr expr = match expr with
39:     | Absyn.Number number ->
40:           string_of_ctor "Number" [string_of_float number]
41:     | Absyn.Memref memref ->
42:           string_of_ctor "Memref" [string_of_memref memref]
43:     | Absyn.Unary (oper, expr) ->
44:           string_of_ctor "Unary" [quote oper; string_of_expr expr]
45:     | Absyn.Binary (oper, expr1, expr2) ->
46:           string_of_ctor "Binary"
47:               [quote oper; string_of_expr expr1; string_of_expr expr2]
48:
```

```
49:
50: let string_of_stmt (stmt: Absyn.stmt) = match stmt with
51:     | Absyn.Dim (ident, expr) ->
52:           string_of_ctor "Dim"
53:               [quote ident ^ ", " ^ string_of_expr expr]
54:     | Absyn.Let (memref, expr) ->
55:           string_of_ctor "Let"
56:               [string_of_memref memref; string_of_expr expr]
57:     | Absyn.Goto label ->
58:           string_of_ctor "Goto" [quote label]
59:     | Absyn.If (expr, label) ->
60:           string_of_ctor "If" [string_of_expr expr; quote label]
61:     | Absyn.Print printable'list ->
62:           string_of_ctor "Print"
63:               [string_of_list string_of_printable printable'list]
64:     | Absyn.Input memref'list ->
65:           string_of_ctor "Input"
66:               [string_of_list string_of_memref memref'list]
67:
68: let dump_progline (linenr, label'option, stmt'option) =
69:     Printf.fprintf stderr "program: %d %s: %s\n%!" linenr
70:         (string_of_option quote label'option)
71:         (string_of_option string_of_stmt stmt'option)
72:
73: let dump_program (program : Absyn.program) =
74:     List.iter dump_progline program
75:
```

```
 1: (* $Id: interp.mli,v 1.7 2020-01-24 12:57:06-08 - - $ *)
 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 1.8 2020-01-24 11:42:24-08 - - $ *)
 2:
 3: open Absyn
 4:
 5: exception Unimplemented of string
 6: let no_expr reason = raise (Unimplemented reason)
 7: let no_stmt reason continuation = raise (Unimplemented reason)
 8:
 9: let want_dump = ref false
10:
11: let rec eval_expr (expr : Absyn.expr) : float = match expr with
12:     | Number number -> number
13:     | Memref memref -> no_expr "eval_expr Memref"
14:     | Unary (oper, expr) -> no_expr "eval_expr Unary"
15:     | Binary (oper, expr1, expr2) -> no_expr "eval_expr Binary"
16:
17: let rec interpret (program : Absyn.program) = match program with
18:     | [] -> ()
19:     | firstline::continuation -> match firstline with
20:       | _, _, None -> interpret continuation
21:       | _, _, Some stmt -> (interp_stmt stmt continuation)
22:
23: and interp_stmt (stmt : Absyn.stmt) (continuation : Absyn.program) =
24:     match stmt with
25:     | Dim (ident, expr) -> no_stmt "Dim (ident, expr)" continuation
26:     | Let (memref, expr) -> no_stmt "Let (memref, expr)" continuation
27:     | Goto label -> no_stmt "Goto label" continuation
28:     | If (expr, label) -> no_stmt "If (expr, label)" continuation
29:     | Print print_list -> interp_print print_list continuation
30:     | Input memref_list -> interp_input memref_list continuation
31:
32: and interp_print (print_list : Absyn.printable list)
33:                  (continuation : Absyn.program) =
34:     let print_item item =
35:         (print_string " ";
36:          match item with
37:          | String string ->
38:            let regex = Str.regexp "\"\\(.*\\)\""
39:            in print_string (Str.replace_first regex "\\1" string)
40:          | Printexpr expr ->
41:            print_float (eval_expr expr))
42:     in (List.iter print_item print_list; print_newline ());
43:     interpret continuation
44:
45: and interp_input (memref_list : Absyn.memref list)
46:                  (continuation : Absyn.program)  =
47:     let input_number memref =
48:         try  let number = Etc.read_number ()
49:              in (print_float number; print_newline ())
50:         with End_of_file ->
51:              (print_string "End_of_file"; print_newline ())
52:     in List.iter input_number memref_list;
53:     interpret continuation
54:
55: let interpret_program program =
56:     (Tables.init_label_table program;
57:      if !want_dump then Tables.dump_label_table ();
58:      if !want_dump then Dumper.dump_program program;
```

```
59:       interpret program)
60:
```

```
 1: (* $Id: main.ml,v 1.3 2020-01-24 12:57:06-08 - - $ *)
 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 1.2 2019-01-25 16:49:38-08 - - $ */
 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 1.2 2019-11-26 14:09:51-08 - - $ *)
 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 1.15 2020-01-24 12:54:40-08 - - $

#
# 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
OCAMLOPT   = ocamlopt -g

#
# File macros
#

EXECBIN    = sbinterp
OBJCMX     = etc.cmx parser.cmx scanner.cmx tables.cmx \
             dumper.cmx interp.cmx main.cmx
OBJCMI     = ${OBJCMX:.cmx=.cmi} absyn.cmi
OBJBIN     = ${OBJCMX:.cmx=.o}
MLSOURCE   = absyn.mli etc.mli etc.ml tables.mli tables.ml \
             dumper.mli dumper.ml interp.mli interp.ml main.ml
GENLEXYACC = parser.mli parser.ml scanner.ml
GENSOURCE  = dumper.mli tables.mli ${GENLEXYACC}
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} : ${OBJCMX}
        ${OCAMLOPT} str.cmxa ${OBJCMX} -o ${EXECBIN}

%.cmi : %.mli
        ${OCAMLOPT} -c $<

%.cmx : %.ml
        ${OCAMLOPT} -c $<

%.ml : %.mll
        ocamllex $<

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

```

```
54:
55: MAKEMLI    = (echo "(* Generated: $$(date) *)"; ${OCAMLOPT} -i $<) >$@
56:
57: tables.mli : tables.ml absyn.cmi
58:         ${call MAKEMLI}
59:
60: dumper.mli : dumper.ml absyn.cmi
61:         ${call MAKEMLI}
62:
63: #
64: # Misc targets
65: #
66:
67: clean :
68:         - rm ${OBJCMI} ${OBJCMX} ${OBJBIN} ${GENSOURCE}
69:
70: spotless : clean
71:         - rm ${EXECBIN} ${GENFILES} ${LISTING} ${LISTING:.ps=.pdf}
72:
73: ci : ${ALLSOURCES}
74:         - checksource ${ALLSOURCES}
75:         cid + ${ALLSOURCES}
76:
77: deps : ${MLSOURCE} ${GENSOURCE}
78:         @ echo "# Generated: $$(date)" >${DEPSFILE}
79:         ocamldep ${MLSOURCE} ${GENSOURCE} >>${DEPSFILE}
80:
81: ${DEPSFILE} : tables.mli
82:         @touch ${DEPSFILE}
83:         ${GMAKE} deps
84:
85: lis : ${ALLSOURCES}
86:         mkpspdf ${LISTING} ${ALLSOURCES}
87:
88: again :
89:         ${GMAKE} spotless
90:         ${GMAKE} deps
91:         ${GMAKE} ci
92:         ${GMAKE} all
93:         ${GMAKE} lis
94:
95: ifeq "${NEEDINCL}" ""
96: include ${DEPSFILE}
97: endif
98:
```

```
# Generated: Fri Jan 24 14:55:58 PST 2020
absyn.cmi :
dumper.cmo : \
    absyn.cmi \
    dumper.cmi
dumper.cmx : \
    absyn.cmi \
    dumper.cmi
dumper.cmi : \
    absyn.cmi
dumper.cmi : \
    absyn.cmi
etc.cmo : \
    etc.cmi
etc.cmx : \
    etc.cmi
etc.cmi :
interp.cmo : \
    tables.cmi \
    etc.cmi \
    dumper.cmi \
    absyn.cmi \
    interp.cmi
interp.cmx : \
    tables.cmx \
    etc.cmx \
    dumper.cmx \
    absyn.cmi \
    interp.cmi
interp.cmi : \
    absyn.cmi
main.cmo : \
    scanner.cmo \
    parser.cmi \
    interp.cmi \
    etc.cmi
main.cmx : \
    scanner.cmx \
    parser.cmx \
    interp.cmx \
    etc.cmx
parser.cmo : \
    etc.cmi \
    absyn.cmi \
    parser.cmi
parser.cmx : \
    etc.cmx \
    absyn.cmi \
    parser.cmi
parser.cmi : \
    absyn.cmi
scanner.cmo : \
    parser.cmi \
    etc.cmi
scanner.cmx : \
    parser.cmx \
    etc.cmx
tables.cmo : \
```

```
59:     absyn.cmi \
60:     tables.cmi
61: tables.cmx : \
62:     absyn.cmi \
63:     tables.cmi
64: tables.cmi : \
65:     absyn.cmi
66: tables.cmi : \
67:     absyn.cmi
```

```
 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:
```