```
# $Id: README,v 1.1 2009-02-03 17:24:36-08 - - $

NAME
   sbtran - translator from SB to SBIR

SYNOPSIS
   sbtran sbsource >sbirobject

DESCRIPTION
   The sbtran utility translate Silly Basic source code into
   Silly Basic Intermediate Representation, which looks like
   Scheme.  You do not need to understand scanning, parsing,
   or O'Caml to use this.  See the SBIR sources in another
   directory.

```

```
 1: (* $Id: etc.mli,v 1.1 2009-02-03 17:24:36-08 - - $ *)
 2:
 3: (*
 4: * Main program and system access.
 5: *)
 6:
 7: val execname : string
 8:
 9: val quit : unit -> unit
10:
11: val eprint : string list -> unit
12:
13: val lexeprint : Lexing.position -> string list -> unit
14:
15: val usageprint : string list -> unit
16:
```

```
 1: (* $Id: absyn.ml,v 1.1 2009-02-03 17:24:36-08 - - $ *)
 2:
 3: (*
 4: * Abstract syntax definitions for SB->IR.
 5: *)
 6:
 7: type linenr     = int
 8: and  variable   = string
 9: and  label      = string
10: and  number     = float
11: and  oper       = string
12: and  arrayfn    = variable * expr
13:
14: and  print      = Printexpr of expr
15:                 | String of string
16:
17: and  memref     = Arrayfn of arrayfn
18:                 | Variable of variable
19:
20: and  expr       = Binop of oper * expr * expr
21:                 | Unop of oper * expr
22:                 | Memref of memref
23:                 | Constant of number
24:
25: and  stmt       = Dim of arrayfn
26:                 | Let of memref * expr
27:                 | Goto of label
28:                 | If of expr * label
29:                 | Print of print list
30:                 | Input of memref list
31:
32: and  program    = (linenr * label option * stmt option) list
33:
```

```
 1: (* $Id: etc.ml,v 1.1 2009-02-03 17:24:36-08 - - $ *)
 2:
 3: open Lexing
 4: open Printf
 5:
 6: let execname = Filename.basename Sys.argv.(0)
 7:
 8: let exit_code_ref = ref 0
 9:
10: let quit () = exit !exit_code_ref
11:
12: let eprintlist message =
13:     (exit_code_ref := 1;
14:      flush_all ();
15:      List.iter (fprintf stderr "%s") message;
16:      fprintf stderr "\n";
17:      flush_all ())
18:
19: let eprint message = eprintlist (execname :: ": " :: message)
20:
21: let lexeprint position message =
22:     eprint (position.pos_fname :: ": "
23:             :: string_of_int position.pos_lnum :: ": "
24:             :: message)
25:
26: let usageprint message =
27:     eprintlist ("Usage: " :: execname :: " " :: message)
28:
```

```
 1: (* $Id: main.ml,v 1.1 2009-02-03 17:24:36-08 - - $ *)
 2:
 3: open Absyn
 4: open Etc
 5: open Lexing
 6: open Printf
 7:
 8: (*
 9: * Functions for printing out the absyn tree in Scheme format.
10: *)
11:
12: let
13: rec pr'print file print = match print with
14:     | Printexpr (expr)           -> fprintf file " %a" pr'expr expr
15:     | String (string)            -> fprintf file " %s" string
16:
17: and pr'memref file memref = match memref with
18:     | Arrayfn (arrayfn)          -> pr'arrayfn file arrayfn
19:     | Variable (variable)        -> fprintf file "%s" variable
20:
21: and pr'expr file expr = match expr with
22:     | Binop (oper, expr1, expr2) -> fprintf file "(%s %a %a)"
23:                                     oper pr'expr expr1 pr'expr expr2
24:     | Unop (oper, expr)          -> fprintf file "(%s %a)"
25:                                     oper pr'expr expr
26:     | Memref (memref)            -> pr'memref file memref
27:     | Constant (number)          -> fprintf file "%.15g" number
28:
29: and pr'stmt file stmt = match stmt with
30:     | None                       -> ()
31:     | Some (Dim (arrayfn))       -> fprintf file "(dim %a)"
32:                                     pr'arrayfn arrayfn
33:     | Some (Let (memref, expr))  -> fprintf file "(let %a %a)"
34:                                     pr'memref memref pr'expr expr
35:     | Some (Goto (label))        -> fprintf file "(goto %s)" label
36:     | Some (If (expr, label))    -> fprintf file "(if %a %s)"
37:                                     pr'expr expr label
38:     | Some (Print (prints))      -> fprintf file "(print%a)"
39:                                     pr'prints prints
40:     | Some (Input (memrefs))     -> fprintf file "(input%a)"
41:                                     pr'inputs memrefs
42:
43: and pr'arrayfn file (variable, expr) =
44:     fprintf file "(%s %a)" variable pr'expr expr
45:
46: and pr'line file (line, label, stmt) =
47:     let s'label = match label with
48:         | None         -> ""
49:         | Some (label) -> label
50:     in  fprintf file "(%5d %-8s %a)\n" line s'label pr'stmt stmt
51:
52: and pr'input   file inputs  = fprintf file " %a" pr'memref inputs
53:
54: and pr'prints  file prints  = List.iter (pr'print file) prints
55:
56: and pr'inputs  file memrefs = List.iter (pr'input file) memrefs
57:
58: and pr'lines   file lines   = List.iter (pr'line file) lines
```

```
59:
60: and pr'program file program = fprintf file "(\n%a)\n" pr'lines program
61:
62: (*
63: * Main program reads a file and prints to stdout.
64: *)
65:
66: let translatefile filename =
67:     try (printf ";;File: %s\n" filename;
68:          let sourcefile =
69:              if filename = "-" then stdin else open_in filename in
70:          let lexbuf = Lexing.from_channel sourcefile in
71:          let absyn = Parser.program Lexer.token lexbuf in
72:          printf "%a" pr'program absyn)
73:     with Sys_error (string) -> eprint [string]
74:
75: let _ = if !Sys.interactive
76:         then ()
77:         else match Array.length Sys.argv with
78:              | 1 -> translatefile "-"
79:              | 2 -> translatefile Sys.argv.(1)
80:              | _ -> usageprint ["[filename.sb]"]
81:
```

```
 1: /* $Id: parser.mly,v 1.1 2009-02-03 17:24:36-08 - - $ */
 2:
 3: %{
 4: (******** BEGIN PARSER SEMANTICS ********)
 5:
 6: open Absyn
 7: open Etc
 8: open Lexing
 9:
10: let syntax () = lexeprint (symbol_start_pos ()) ["syntax error"]
11:
12: let linenr () = (symbol_start_pos ()).pos_lnum
13:
14: (******** END PARSER SEMANTICS ********)
15: %}
16:
17: %token <string> RELOP EQUAL ADDOP MULOP POWOP
18: %token <string> IDENT NUMBER STRING
19: %token COLON COMMA LPAR RPAR EOL EOF
20: %token DIM LET GOTO IF PRINT INPUT
21:
22: %type <Absyn.program> program
23:
24: %start program
25:
26: %%
27:
28: program : stmts EOF              { List.rev $1 }
29:
30: stmts   : stmts stmt EOL         { $2::$1 }
31:         | stmts error EOL        { syntax (); $1 }
32:         |                        { [] }
33:
34: stmt    : label action           { (linenr (), Some $1, Some $2) }
35:         | action                 { (linenr (), None, Some $1) }
36:         | label                  { (linenr (), Some $1, None) }
37:         |                        { (linenr (), None, None) }
38:
39: label   : ident COLON            { $1 }
40:
41: action  : DIM arrayfn            { Dim ($2) }
42:         | LET memref EQUAL expr  { Let ($2, $4) }
43:         | GOTO ident             { Goto ($2) }
44:         | IF relexpr GOTO ident  { If ($2, $4) }
45:         | PRINT prints           { Print ($2) }
46:         | PRINT                  { Print ([]) }
47:         | INPUT inputs           { Input ($2) }
48:
49: prints  : print COMMA prints     { $1::$3 }
50:         | print                  { [$1] }
51:
52: print   : expr                   { Printexpr ($1) }
53:         | STRING                 { String ($1) }
54:
55: inputs  : memref COMMA inputs    { $1::$3 }
56:         | memref                 { [$1] }
57:
58: memref  : ident                  { Variable ($1) }
```

```
59:          | arrayfn                { Arrayfn ($1) }
60:
61: relexpr : expr RELOP expr         { Binop ($2, $1, $3) }
62:          | expr EQUAL expr         { Binop ($2, $1, $3) }
63:
64: expr    : expr ADDOP term          { Binop ($2, $1, $3) }
65:          | term                    { $1 }
66:
67: term    : term MULOP factor        { Binop ($2, $1, $3) }
68:          | factor                  { $1 }
69:
70: factor  : primary POWOP factor     { Binop ($2, $1, $3) }
71:          | primary                 { $1 }
72:
73: primary : LPAR expr RPAR           { $2 }
74:          | ADDOP primary           { Unop ($1, $2) }
75:          | NUMBER                  { Constant (float_of_string ($1)) }
76:          | memref                  { Memref ($1) }
77:
78: arrayfn : ident LPAR expr RPAR     { ($1, $3) }
79:
80: ident   : IDENT                    { $1 }
81:          | DIM                     { "dim" }
82:          | GOTO                    { "goto" }
83:          | IF                      { "if" }
84:          | INPUT                   { "input" }
85:          | LET                     { "let" }
86:          | PRINT                   { "print" }
87:
```

```
(* $Id: lexer.mll,v 1.1 2009-02-03 17:24:36-08 - - $ *)

{
(******** BEGIN LEXER SEMANTICS ********)

open Absyn
open Etc
open Lexing
open Parser
open Printf

let lexerror lexbuf =
    lexeprint (lexeme_start_p lexbuf)
              ["invalid character `" ^ (lexeme lexbuf) ^ "'"]

let newline lexbuf =
    let incrline pos =
        {pos with pos_lnum = pos.pos_lnum + 1; pos_bol = pos.pos_cnum}
    in  (lexbuf.lex_start_p <- incrline lexbuf.lex_start_p;
         lexbuf.lex_curr_p <- incrline lexbuf.lex_curr_p)

let list lexbuf =
    let pos = lexeme_start_p lexbuf
    in  (if pos.pos_bol = pos.pos_cnum
         then printf ";;%4d: " pos.pos_lnum;
         printf "%s" (lexeme lexbuf))

let keylist = [
        "dim"  , DIM   ;
        "goto" , GOTO  ;
        "if"   , IF    ;
        "input", INPUT ;
        "let"  , LET   ;
        "print", PRINT ;
    ]

let keyhash : (string, token) Hashtbl.t
            = Hashtbl.create (List.length keylist)

let _ = List.iter (fun (word, token) -> Hashtbl.add keyhash word token)
                  keylist

let identkeyword ident =
    try Hashtbl.find keyhash ident
    with Not_found -> IDENT ident

(******** END LEXER SEMANTICS ********)
}

let letter          = ['a'-'z' 'A'-'Z' '_']
let digit           = ['0'-'9']
let fraction        = (digit+ '.'? digit* | '.' digit+)
let exponent        = (['E' 'e'] ['+' '-']? digit+)

let comment         = ('#' [^'\n']*)
let ident           = (letter (letter | digit)*)
let number          = (fraction exponent?)
let string          = '"' [^'\n' '"']* '"'
```

```
59:
60: rule token          = parse
61:      | eof          { EOF }
62:      | [' ' '\t']   { list lexbuf; token lexbuf }
63:      | comment      { list lexbuf; token lexbuf }
64:      | "\n"         { list lexbuf; newline lexbuf; EOL }
65:      | ":"          { list lexbuf; COLON }
66:      | ","          { list lexbuf; COMMA }
67:      | "("          { list lexbuf; LPAR }
68:      | ")"          { list lexbuf; RPAR }
69:      | "="          { list lexbuf; EQUAL (lexeme lexbuf) }
70:      | "<>"         { list lexbuf; RELOP (lexeme lexbuf) }
71:      | "<"          { list lexbuf; RELOP (lexeme lexbuf) }
72:      | "<="         { list lexbuf; RELOP (lexeme lexbuf) }
73:      | ">"          { list lexbuf; RELOP (lexeme lexbuf) }
74:      | ">="         { list lexbuf; RELOP (lexeme lexbuf) }
75:      | "+"          { list lexbuf; ADDOP (lexeme lexbuf) }
76:      | "-"          { list lexbuf; ADDOP (lexeme lexbuf) }
77:      | "*"          { list lexbuf; MULOP (lexeme lexbuf) }
78:      | "/"          { list lexbuf; MULOP (lexeme lexbuf) }
79:      | "%"          { list lexbuf; MULOP (lexeme lexbuf) }
80:      | "^"          { list lexbuf; POWOP (lexeme lexbuf) }
81:      | number       { list lexbuf; NUMBER (lexeme lexbuf) }
82:      | string       { list lexbuf; STRING (lexeme lexbuf) }
83:      | ident        { list lexbuf; identkeyword (lexeme lexbuf) }
84:      | _            { list lexbuf; lexerror lexbuf; token lexbuf }
85:
```

```
 1: # $Id: Makefile,v 1.10 2014-11-12 20:51:57-08 - - $
 2:
 3: #
 4: # General useful macros
 5: #
 6:
 7: MKFILE     = Makefile
 8: MAKEFLAGS += --no-builtin-rules
 9: DEPSFILE   = ${MKFILE}.deps
10: NOINCLUDE  = ci clean spotless
11: NEEDINCL   = ${filter ${NOINCLUDE}, ${MAKECMDGOALS}}
12:
13: #
14: # File list macros
15: #
16:
17: EXECBIN    = sbtran
18: OBJCMX     = absyn.cmx etc.cmx parser.cmx lexer.cmx main.cmx
19: OBJCMI     = ${patsubst %.cmx, %.cmi, ${OBJCMX}}
20: OBJBIN     = ${patsubst %.cmx, %.o, ${OBJCMX}}
21: MLSOURCE   = etc.mli absyn.ml etc.ml main.ml
22: GENSOURCE  = parser.mli parser.ml lexer.ml
23: GENFILES   = ${GENSOURCE} parser.output
24: ALLSOURCES = README ${MLSOURCE} parser.mly lexer.mll ${MKFILE}
25: LISTING    = Listing.ps
26:
27: #
28: # General targets
29: #
30:
31: all : ${EXECBIN}
32:
33: ${EXECBIN} : ${OBJCMX}
34:         ocamlopt ${OBJCMX} -o ${EXECBIN}
35:
36: %.cmi : %.mli
37:         ocamlc -c $<
38:
39: %.cmx : %.ml
40:         ocamlopt -c $<
41:
42: %.ml : %.mll
43:         ocamllex $<
44:
45: %.mli %.ml : %.mly
46:         ocamlyacc -v $<
47:
48: #
49: # Misc targets
50: #
51:
52: clean :
53:         - rm ${OBJCMI} ${OBJCMX} ${OBJBIN}
54:
55: spotless : clean
56:         - rm ${EXECBIN} ${GENFILES}
57:
58: ci : ${ALLSOURCES} ${SBFILES}
```

```
59:         cid + ${ALLSOURCES} ${SBFILES}
60:
61: deps : ${MLSOURCE} ${GENSOURCE}
62:         @ echo "# ${DEPSFILE} created `date`" >${DEPSFILE}
63:         ocamldep ${MLSOURCE} ${GENSOURCE} | sort | uniq >>${DEPSFILE}
64:
65: ${DEPSFILE} :
66:         @touch ${DEPSFILE}
67:         ${MAKE} deps
68:
69: lis : ${ALLSOURCES}
70:         mkpspdf ${LISTING} ${ALLSOURCES}
71:
72: again :
73:         ${MAKE} spotless
74:         ${MAKE} deps
75:         ${MAKE} ci
76:         ${MAKE} all
77:         ${MAKE} lis
78:
79: ifeq "${NEEDINCL}" ""
80: include ${DEPSFILE}
81: endif
82:
```