*Vencislav Stanev vs2226*

*Final Report*

# *Turn Based Simulation Language*

# Contents

# 1. Tutorial

## 1.1 Objective

The turn based simulation language (TBSL) is a functional language that enables programmers to describe a current state of a system comprised of objects. The goal of TBSL is to run that simulation for a number of turns in order to examine the effects of particular phenomena on the system.

## 1.2 First steps

TBSL has a C-like syntax. It is comprised of a number of variable declarations and functions that operate on them. Each program needs to contain the function “program”, which serves as an entry point.

*Example*:

.

```
program() {

}
```

A program will contain a number of variable declarations- both global and local.

*Example*:

```
Init Monopoly_simulation ((turns,20),(status,”active”));

Init Player_1((name,”Jim”),(cash,100));

Init Player_2((name,”Kimiko”),(cash,200));


program() {

}
```

Each program will probably contain a number of functions which change the state of the declared variables:

*Example*:

```
Init Monopoly_simulation ((turns,20),(status,”active”));

Init Player_1((name,”Jim”),(cash,100));

Init Player_2((name,”Kimiko”),(cash,200));

DeductCash(var) {var->cash=var->cash-5; return var;}

program() {

}
```

## 1.3 Using TBSL

Most programs in TBSL have a main program loop which keeps track of how many turns have passed for this simulation.

*Example*:

```
Init MonopolySimulation ((turns,20),(status,”active”));

Init Player_1((name,”Jim”),(cash,100));

Init Player_2((name,”Kimiko”),(cash,200));

DeductCash(var) {var->cash=var->cash-5; return var;}
program() {

   while(MonopolySimulation->turns > 0)

      /* code here */

   MonopolySimulation->turns=MonopolySimulation->turns-1;

   }
```

# 2. Reference Manual

## 2.1 Lexical Conventions

### 2.1.1 Identifiers

An identifier is a sequence of letters, digits and the underscore character. Each identifier starts with a letter. Identifiers are case sensitive - upper and lower case letters are considered different.

### 2.1.2 Comments

Comments are introduced with the opening character sequence /* and closed with the sequence */. Comments cannot be nested - the characters /* introduce a comment, which terminates with the first occurrence of the characters */.

### 2.1.3 Punctuation

| *Punctuation* | *Use* | *Example* |
|---|---|---|
| */* */* | *Comments* | */* This is a comment */* |
| *“ ”* | *String constant* | *“This is a string”* |
| *;* | *Indicates the end of a statement* | *Compare (a,b);* |
| *,* | *Argument list separator* | *Compare (a,b);* |
| *()* | *Argument list delimiter* | *Compare (a,b);* |
| *{ }* | *Function body or block of statements* | *func Compare (a,b)<br>{<br>Body of function here<br>}* |
| *->* | *Reference a variable attribute* | *a->cost* |

### 2.1.4 Constants

Constants are used to initialize variable attributes.

Integer constants – integer constants are represented with whole numbers in decimal format. An integer constant constitutes only of digits; decimal point and exponent are not allowed. A unary – operator is allowed. An example of an integer constant is 4 or 6000 or 12. The system stores all numbers as floating point numbers so each integer constant is implicitly converted to a float.

Floating point constants – floating point constants are represented with a whole part, a decimal point, a fractional part and e and an exponent. Ether the whole or the fractional part needs to be always present. The whole part and the fractional part are made up only of digits. A unary – operator is allowed. An example of a floating point constant is 1. 0.5e-15 .3e+3 .2 1e5.

String constants – string constants are made up of a sequence of characters that are enclosed in quotes. For example “this is a string” or “5” or “Some characters @#$%^&( “. String constants cannot contain the character “.

## 2.2 Basic Types

TBSL has only one basic type, which is called “Object”. No notion of type conversion is defined.

### 2.2.1 Object type

When declaring a variable, type is not specified but the variable can be initialized. A variable is initialized by providing a custom list of attributes, which is a list of tuples, each tuple being a name\value pair. The name is always an identifier and the value can be an integer, float or a string but not an expression. Defining a second variable with the same name in the same scope is not allowed. A variable has no predefined attributes; attributes are all custom and could be added at initialization time as well as later in the program.

When declaring a local variable within a function the variable declarations should be at the top of the function body preceding any other statements. TBSL enforces that rule and will raise an exception in case of improper declaration.

When declaring a global variable the order of variable definitions and functions is not important (functions can precede variables). At the time of execution of the function the global variables are already in the symbol table.

*Syntax example*:

This syntax initializes the variable a.

```
Init a ((status,”active”),(cost, 5.7),(ValueAddPerTurn,10));
```

*Syntax examples*:
Re-declaration of a variable is not permitted in the same scope. The following are examples of syntax that is not allowed.

```
Init a ((status,”active”),(cost, 5.7),(ValueAddPerTurn,10));
Init a ((cost, 7.7),(ValueAddPerTurn,12));
/*----------------------------------------------------------*/
test_function(variable)
{
   Init variable((first,”Samuel”),(second,”Adams”));
}
```

### 2.2.2 Variable attribute

A variable attribute could be referenced by providing the following syntax:

*Syntax example*:

The following syntax shows a few examples of how to update the variable attributes as well as use them in other expressions. If a variable attribute is not already defined on the variable it is added, otherwise just the value is updated.

```
a->cost = 5.7;
a->cost = a->cost + 40;
a->cost = b->cost;
a->cost = “str_const”;
```

## 2.3 Operators

Operators in TBSL are tokens that allow for particular operations on variables or variable attributes.

<u>Variable Operators</u> - For variables the only defined operator is assignment.

*Syntax example:*

```
Init a ((status,”active”),(cost, 5.7),(ValueAddPerTurn,10));
Init b ((status,”inactive”),(cost, 7.7),(ValueAddPerTurn,2));
a=b;
```

Variable Attribute Operators – these operators are defined by the type of data stored in the attribute. An exception is raised in case of mismatch. The list below is ordered by precedence.

For integers and floats we have the following math operators which return a float:

| | |
|---|---|
| - | Unary minus |
| * | Multiplication |
| / | Division |
| + | Addition |
| - | Subtraction |
| = | Assignment |

As well as the comparison operators which return Boolean:

| | |
|---|---|
| == | Equal |
| != | Not equal |
| < | Less than |
| <= | Lest than or equal |
| > | Greater than |
| >= | Greater than or equal |

For string constants the string concatenation operator returns a string:

| | |
|---|---|
| concat | String concatenation |
| cmp | String compare |

*Syntax example:*

```
Init a ((status,”active”),(cost, 5.7),(ValueAddPerTurn,10));
Init b ((status,”inactive”),(cost, 7.7),(ValueAddPerTurn,2));
      a->cost = a->cost + 40;
      a->cost = b->cost;
      a->cost = “str_const”;
```

Logical operators – TBSL also supports the logical operators && and ||.

## 2.4 Syntactic Constructs

TBSL supports the following control constructs

### 2.4.1 If than else

Conditional control logic

*Syntax example:*

```
Init test ((status,”active”),(cost, 5.7),(ValueAddPerTurn,10));

   if(test->cost == 3)
   {
      test->status = “inactive”;
   }
   else
   {
      test->status = “active”;
   }
```

### 2.4.2 Loops

This construct allows for repeated execution of a block of code

*Syntax example:*

```
test->c = 3;
   while(test->c < 6)
   {
      printAll();
      test->c = test->c + 1;
   }
```

## 2.5 Functions

TBSL supports functions in order to promote modularity. A function is a collection of statements that are given a name. Functions in TBSL return only one type – the “Object” type. If no return statement is present inside the function then the function returns an empty object, otherwise the appropriate object is returned. All parameters are “passed by value” and the input is not modified at all. The side effects of the execution of the function are reflected in the returned object. Defining a second function with the same name in the same scope is not allowed. In addition the nesting of function declarations is also not allowed.

*Syntax example:*

```
func(var)
{
   var->test = "Came inside the function";
   return var;
}
```

Some built in functions include:

printAll() – This function does not take any parameters and prints out the contents of the Global symbol table as well as the local symbol table with the values of all the variable attributes at the time of execution.

print(a) – This function takes an object as a parameter and prints out the variable attributes that are part of this object.

## 2.6 Scope

TBSL supports the notion of scope by defining blocks of code much like C and Java do. A block of code is defined by wrapping it in {}

## 2.7 Example of an Algorithm

```
Init simulation ((“turns”,10)(“turnDecrement”,1));
Init store_a ((“status”,”active”), (“balance”, 7.2), (“ValueAdd”,10));
Init store_b ((“status”,”inactive”), (“balance”, 4.0), (“ValueAdd”,12));
While (simulation->turns >0)
{
  Attribute (store_a, (“balance”, store_a->balance+store_a->ValueAdd));
  Attribute (store_b, (“balance”, store_b->balance+store_b->ValueAdd));
  Attribute (simulation, (“turns”, simulation->turns – simulation-
>turnsDecrement));
}
/* Prints all attributes of the object */
Print(store_a);
Print(store_b);
```

# 3. Architectural Design

## 3.1 Components

The architecture of the tbsl interpreter follows the example of the Micro C presented in class.

The scanner is enhanced to accepts the list of operators for tbsl. In addition the scanner handles the data types allowed in tbsl as well as the relevant key words.

The parse has a definition for all tokens and explicit declaration of the operator precedence to resolve shift\reduce errors. It handles the specific declaration and initialization style of the language. In addition expressions were split in two groups – expressions that handle “objects” and expressions that handle “object attributes”

The AST has a few new node declarations and constructors.

The interpreter has a symbol table for functions and variables much the same way Micro C does. The variable symbol table has a different structure and is implemented as a Map of Maps. The internal representation of data is done using string data type, which is converted to float when arithmetic operators are applied. TBSL has one looping construct (while) and one flow control (if then else statement) since other looping constructs (for loop, switch statement) provide equivalent capabilities and do not significantly enhance the expressive power of the language.

There are two type of statements – those that return a scalar and those that return an object. The user of tbsl has to keep in mind what kind of statement is used in each case especially when the return value is important.

The top level of the program is the project file. It has as set of instructions that work by reading a tbsl file from the command line and a second set of instructions that read tbsl from the command line. At any given time one of those sets should be commented out.

## 3.2 Source Code

### 3.2.1 Scanner.mll

```
{ open Parser }


let digits = ['0'-'9']+                    (* One or more digits *)
let optDigits = ['0'-'9']*                 (* Zero or more digits *)
let sign = '+'|'-'                         (* Plus or minus sign *)
let exponent = ('e'|'E') sign? digits      (* Exponent *)
let optExponent = exponent?                (* Optional exponent *)


rule token = parse
  [' ' '\t' '\r' '\n']   { token lexbuf }    (* White spaces *)
| "/*"                   { comment lexbuf }  (* Comments *)
| "->"                   { ARROW }           (* Dereference a parameter in an "Object" *)
| '+'                    { PLUS }            (* Addition *)
| '-'                    { MINUS }           (* Subtraction *)
| '*'                    { TIMES }           (* Multiplication *)
| '/'                    { DIVIDE }          (* Division *)
| ','                    { COMMA }           (* Sequence *)
| '('                    { LPAREN }          (* Left Paren *)
| ')'                    { RPAREN }          (* Right Paren *)
| '{'                    { LBRACE }          (* Left Brace*)
| '}'                    { RBRACE }          (* Right Brace*)
| ';'                    { SEMI }            (* End of Expression *)
| '='                    { ASSIGN }          (* Assignment *)
| "=="                   { EQ }              (* Equal *)
| "!="                   { NEQ }             (* Not equal *)
| '<'                    { LT }              (* Less than *)
| "<="                   { LEQ }             (* Less than or equal *)
| ">"                    { GT }              (* Greater than *)
| ">="                   { GEQ }             (* Greater than or equal *)
| "&&"                   { AND }             (* And logical operator *)
| "||"                   { OR }              (* Or logical operator *)
| "cmp"                  { COMPARE}          (* String comparison operator *)
| "concat"               { CONCAT}           (* Concatenation of string constants *)
| "return"          { RETURN }       (* Return value *)
| "while"           { WHILE }        (* Loop *)
| "if"              { IF }           (* Conditional *)
| "else"            { ELSE }         (* Alternate control flow *)
| "Init"            { INIT }         (* Initialization *)
| eof               { EOF }          (* End of File *)

| '\"' [^ '\"']* '\"'               as lxm { STRCONST(lxm) }    (* String constant*)
| ( (digits '.'? optDigits optExponent) |
   ('.'? digits optExponent) )            as lxm { LITERAL(lxm) }     (* Integer or float *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*   as lxm { ID(lxm) }          (* Name of a variable*)
| _ as char          { raise (Failure("illegal character " ^ Char.escaped char)) }



and comment = parse "*/" { token lexbuf }
| _ { comment lexbuf }
```

### 3.2.2 Parser.mly

```
%{ open Ast %}

%token ARROW PLUS MINUS TIMES DIVIDE COMMA LPAREN RPAREN LBRACE
RBRACE SEMI
%token EQ NEQ LT LEQ GT GEQ COMPARE CONCAT AND OR RETURN WHILE IF ELSE
INIT EOF
%token <string> LITERAL
%token <string> ID
%token <string> STRCONST

%token ASSIGN

%nonassoc NOELSE
```

```
%nonassoc ELSE

%right ASSIGN

%left AND OR

%left EQ NEQ

%left LT GT LEQ GEQ COMPARE

%left CONCAT

%left PLUS MINUS

%left TIMES DIVIDE

%nonassoc UMINUS


%start program

%type <Ast.program> program


%%

program:


  /* nothing */                  { [], [] }

  | program vdecl                    { ($2 :: fst $1), snd $1 }

  | program fdecl                    { fst $1, ($2 :: snd $1) }


fdecl:

   ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE

                             { { fname = $1;

                                formals = $3;

                                locals = List.rev $6;

                                body = List.rev $7 } }


formals_opt:

    /* nothing */                { [] }

  | formal_list                  { List.rev $1 }
```

```
formal_list:
    ID                          { [$1] }
  | formal_list COMMA ID                { $3 :: $1 }

params_opt:
     /* nothing */                  { [] }
  | params_list                     { $1 }

params_list:
    param                           { [$1] }
  | params_list COMMA param               { $3 :: $1 }

param:
    LPAREN ID COMMA STRCONST RPAREN        { {   p_key   = $2;
                                    p_value = String.sub $4 1 ((String.length $4)-2) ;} }

  | LPAREN ID COMMA MINUS LITERAL RPAREN { {   p_key   = $2;
                                    p_value = string_of_float (-.(float_of_string $5));} }

  | LPAREN ID COMMA LITERAL RPAREN            { {   p_key   = $2;
                                    p_value = $4;} }

vdecl_list:
     /* nothing */                  { [] }
  | vdecl_list vdecl                    { $2 :: $1 }

vdecl:
    INIT ID LPAREN params_opt RPAREN SEMI      { {   v_name = $2;
                                         v_params = List.rev $4;} }
```

```
stmt_list:
    /* nothing */                { [] }
  | stmt_list stmt                  { $2 :: $1 }

stmt:

    expr SEMI                       { Expr($1) }
  | RETURN expr SEMI                   { Return($2) }
  | LBRACE stmt_list RBRACE              { Block(List.rev $2) }
  | IF LPAREN lexpr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN lexpr RPAREN stmt ELSE stmt   { If($3, $5, $7) }
  | WHILE LPAREN lexpr RPAREN stmt          { While($3, $5) }

expr:
    ID                        { Id($1) }
  | ID ASSIGN expr                  { AssignVar($1, $3)}
  | ID ARROW ID ASSIGN lexpr               { Assign($1, $3, $5) }
  | ID LPAREN actuals_opt RPAREN           { Call($1, $3) }
  | LPAREN expr RPAREN                { $2 }

lexpr:

    LITERAL                         { Literal($1) }
  | STRCONST                        { StrConst($1) }
  | ID ARROW ID ASSIGN lexpr               { AssignL($1, $3, $5) }
  | ID ARROW ID                     { Param($1, $3) }
  | MINUS lexpr %prec UMINUS               { Uminus($2) }
  | lexpr PLUS lexpr                   { Binop($1, Add, $3) }
  | lexpr MINUS lexpr              { Binop($1, Sub, $3) }
  | lexpr TIMES lexpr              { Binop($1, Mult, $3) }
  | lexpr DIVIDE lexpr         { Binop($1, Div, $3) }
  | lexpr EQ lexpr             { Binop($1, Equal, $3) }
  | lexpr NEQ lexpr            { Binop($1, Neq, $3) }
  | lexpr LT lexpr             { Binop($1, Less, $3) }
  | lexpr LEQ lexpr            { Binop($1, Leq, $3) }
  | lexpr GT lexpr             { Binop($1, Greater, $3) }
  | lexpr GEQ lexpr            { Binop($1, Geq, $3) }
  | lexpr CONCAT lexpr         { Binop($1, Concat, $3) }
  | lexpr COMPARE lexpr            { Binop($1, Cmp, $3) }
  | lexpr AND lexpr            { Binop($1, And, $3) }
  | lexpr OR lexpr             { Binop($1, Or, $3) }
  | LPAREN lexpr RPAREN            { $2 }


actuals_opt:
   /* nothing */           { [] }
  | actuals_list           { List.rev $1 }


actuals_list:
   expr                    { [$1] }  | actuals_list COMMA expr          { $3 :: $1 }
```

### 3.2.3 Ast.mli

```
type op = Add | Sub | Mult | Div| Equal | Neq | Less | Leq | Greater | Geq | Cmp | Concat | And
| Or


type lexpr =                   (* Expressions *)
   Literal of string           (* 42.0 *)
  | StrConst of string             (* "Hello World" *)
  | Param of string * string       (* a->b *)
  | Uminus of lexpr            (* -42.0 *)
  | Binop of lexpr * op * lexpr           (* a->d + b->c *)
  | AssignL of string * string * lexpr        (* a->b = 42 *)

type expr =                         (* Expressions *)
    Id of string                    (* a *)
  | AssignVar of string * expr                 (* a = b or a=foo(b) *)
  | Assign of string * string * lexpr         (* a->b = 42 *)
  | Call of string * expr list          (* foo(a, b) *)

type stmt =                         (* Statements *)
    Block of stmt list              (* { ... } *)
  | Expr of expr                    (* a = b; *)
  | Return of expr                        (* return a; *)
  | If of lexpr * stmt * stmt                 (* if (a->b == 1) {} else {} *)
  | While of lexpr * stmt               (* while (a->b<10) { a->b = a->b + 1 } *)

type param_decl = {
    p_key     : string;
    p_value  : string;
}

type v_decl = {
    v_name  : string;
    v_params     : param_decl list;
}

type func_decl = {
    fname     : string;             (* Name of the function *)
    formals  : string list;             (* Formal argument names *)
    locals     : v_decl list;                 (* Locally defined variables *)
    body      : stmt list;                (* Body of the function *)
}

type program = v_decl list * func_decl list      (* global vars, funcs *)
```

### 3.2.4 Interpreter.ml

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of string NameMap.t * string NameMap.t NameMap.t



                                        (* Main entry point: run a program *)

let run (vars, funcs) =

                                        (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
    (fun funcs fdecl ->
     if NameMap.mem fdecl.fname funcs then
       raise (Failure ("function " ^ fdecl.fname ^ " is defined more than once!"));
     NameMap.add fdecl.fname fdecl funcs ) NameMap.empty funcs
  in
```

```
(* function for initializing variables *)
  let init_variables = List.fold_left
    (fun globals vdecl ->

    let params = List.fold_left
        (fun param_map param_decl -> NameMap.add param_decl.p_key
param_decl.p_value param_map)
        NameMap.empty vdecl.v_params in

    if NameMap.mem vdecl.v_name globals then
        raise (Failure ("variable " ^ vdecl.v_name ^ " is defined more than once!"));

     NameMap.add vdecl.v_name params globals;

    )

  in



                                        (* Invoke a function and return an updated global
symboltable *)
  let rec call fdecl actuals globals =



                                        (* Evaluate an lexpr and return (value, updated
environment) *)
    let rec l_eval env = function

        Literal(i) -> i, env

      | StrConst(i) -> (String.sub i 1 ((String.length i)-2)),env
      | Param(var, param) ->

        let (locals, globals) = env in

        if NameMap.mem var locals then
           if NameMap.mem param (NameMap.find var locals) then
             NameMap.find param (NameMap.find var locals),(locals, globals)
           else raise (Failure ("undeclared identifier " ^ var))

        else if NameMap.mem var globals then
           if NameMap.mem param (NameMap.find var globals) then
             NameMap.find param (NameMap.find var globals),(locals, globals)
            else raise (Failure ("undeclared identifier " ^ var))
        else raise (Failure ("undeclared identifier " ^ var))

      | Uminus(e1) ->
        let v1, env = l_eval env e1 in
        string_of_float (-.(float_of_string v1)),env

      | Binop(e1, op, e2) ->
        let v1, env = l_eval env e1 in
        let v2, env = l_eval env e2 in
        let boolean_to_str i = if i then "yes" else "no" in
        let int_to_str i = if i=0 then "yes" else "no" in
        (match op with
          Add  -> string_of_float ((float_of_string v1) +. (float_of_string v2))
        | Sub  -> string_of_float ((float_of_string v1) -. (float_of_string v2))
        | Mult -> string_of_float ((float_of_string v1) *. (float_of_string v2))
        | Div  -> string_of_float ((float_of_string v1) /. (float_of_string v2))
```

```
            | Equal-> boolean_to_str  ((float_of_string v1) =  (float_of_string v2))
            | Neq  -> boolean_to_str  ((float_of_string v1) <> (float_of_string v2))
            | Less -> boolean_to_str  ((float_of_string v1) <  (float_of_string v2))
            | Leq  -> boolean_to_str  ((float_of_string v1) <= (float_of_string v2))
            | Greater -> boolean_to_str ((float_of_string v1) >  (float_of_string v2))
            | Geq -> boolean_to_str   ((float_of_string v1) >= (float_of_string v2))
            | Concat -> String.concat "" (v1::(v2::[]))
            | Cmp ->  int_to_str (String.compare v1 v2)
            | And ->  if (0 = (String.compare v1 "yes") &&
                        0 = (String.compare v2 "yes")) then "yes" else "no"
            | Or ->



            if (0 = (String.compare v1 "yes") ||
                        0 = (String.compare v2 "yes")) then "yes" else "no"  ), env


           | AssignL(var, param, e) ->


            let v, (locals, globals) = l_eval env e in


            if NameMap.mem var locals then
                  v, (NameMap.add var (NameMap.add param v (NameMap.find var locals) )
locals, globals)
            else if NameMap.mem var globals then
                  v, (locals, NameMap.add var (NameMap.add param v (NameMap.find var
globals) ) globals)
            else raise (Failure ("undeclared identifier " ^ var))


      in
```

```
                                    (* Evaluate an expression and return (value, updated
environment) *)
  let rec eval env = function

    Id(var) ->

     let locals, globals = env in
     if NameMap.mem var locals then
       (NameMap.find var locals), env
     else if NameMap.mem var globals then
       (NameMap.find var globals), env
     else raise (Failure ("undeclared identifier " ^ var))

   | Assign(var, param, e) ->

     let v, (locals, globals) = l_eval env e in

     if NameMap.mem var locals then
       NameMap.empty, (NameMap.add var (NameMap.add param v (NameMap.find
var locals) ) locals, globals)
     else if NameMap.mem var globals then
       NameMap.empty, (locals, NameMap.add var (NameMap.add param v
(NameMap.find var globals) ) globals)
     else raise (Failure ("undeclared identifier " ^ var))

   | AssignVar(var, e) ->

     let v, (locals, globals) = eval env e in

     if NameMap.mem var locals then
       v,((NameMap.add var v locals),globals)
  else if NameMap.mem var globals then
      v,(locals,(NameMap.add var v globals))
  else raise (Failure ("undeclared identifier " ^ var))



| Call("printAll", []) ->

  let print_params key value = print_string "  "; print_string key; print_endline "->";
                        NameMap.iter (fun k v -> print_string "     ";
                                   print_string k;
                                   print_string " = ";
                                   print_endline v;) value in
  print_endline "------------------------";
  print_endline ("-Global-");
  NameMap.iter print_params (snd env);
  print_endline "";
  print_endline ("-Local-");
  NameMap.iter print_params (fst env);
  print_endline "------------------------";
  NameMap.empty, env



| Call("print", [var_list]) ->



  let var, env = eval env var_list in

  print_endline "";
  print_endline ("-Listing-");
  NameMap.iter (fun k v -> print_string "     ";
                    print_string k;
                    print_string " = ";
                    print_endline v;) var;

          NameMap.empty, env

        | Call(f, actuals) ->

          let fdecl =
            try NameMap.find f func_decls
            with Not_found -> raise (Failure ("undefined function " ^ f)) in

          let actuals, env =
            List.fold_left (fun (actuals, env) actual ->
                    let v, env = eval env actual in v :: actuals, env)
              ([], env) actuals in

          let (locals, globals) = env in

            try
              let globals = call fdecl actuals globals in
                NameMap.empty, (locals, globals)
            with ReturnException(v, globals) ->
                v, (locals, globals)
      in

                                (* Execute a statement and return an updated environment *)
      let rec exec env = function
```

```
 Block(stmts) -> List.fold_left exec env stmts

| Expr(e) -> let _, env = eval env e in env

| If(e, s1, s2) -> let v, env = l_eval env e in
          exec env (if String.compare "yes" v == 0 then s1 else s2)

| While(e, s) -> let rec loop env =
            let v, env = l_eval env e in
              if String.compare "yes" v == 0 then loop (exec env s)
              else env
          in loop env

| Return(e) -> let v, (locals, globals) = eval env e in
          raise (ReturnException(v, globals))

in


                              (* call: enter the function: bind actual values to formal args *)
let locals =
  try List.fold_left2
    (fun locals formal actual -> NameMap.add formal actual locals)
    NameMap.empty fdecl.formals actuals
  with Invalid_argument(_) ->
     raise (Failure ("wrong number of arguments to " ^ fdecl.fname))
in

                              (* Initilize local variables *)
let locals = init_variables locals fdecl.locals in
```

```
                                        (* Execute each statement; return updated global
  symboltable *)
  snd (List.fold_left exec (locals, globals) fdecl.body)

in
                                        (* add global variables to symbol table. *)

let globals = init_variables NameMap.empty vars in

                                        (* Run the "program" function *)
try
  call (NameMap.find "program" func_decls) [] globals;
with Not_found ->
  raise (Failure ("did not find the program() function"))
```

### 3.2.5 project.ml

```
(* read from a file *)
let _ =
    for i = 1 to Array.length Sys.argv - 1 do
        let ic = open_in Sys.argv.(i) in
        let lexbuf = Lexing.from_channel ic in
        let program = Parser.program Scanner.token lexbuf in
            ignore (Interpret.run program);
     done
```

```
(*


(* read from the command line *)

let _ =

  let lexbuf = Lexing.from_channel stdin in

  let program = Parser.program Scanner.token lexbuf in


    ignore (Interpret.run program);


*)
```

# 4. Testing

Testing the TBSL interpreter involved creating the test cases, files with the desired results for each test case and the instrumentation to repeatedly call all test cases in order to regression test.

## 4.1 Testing script

The operating system that runs my ocaml installation is Windows so the choice for implementing a regression testing script was a windows shell command file aka .bat file. The following is what the .bat file looks like

```
@echo off

for %%X in (*.tbsl) do (

     project %%~nX.tbsl > %%~nX.out

    fc %%~nX.out %%~nX.ans > %%~nX.diff

    IF ERRORLEVEL 1 (echo %%~nX FAIL goto :end)

    IF ERRORLEVEL 0 echo %%~nX PASS

:end

REM echo just a label

)
```

The script lists all .tbsl files in the directory, compiles them to .out files and then for each one it compares the difference and stores the result in a file. If there are differences between the output and the answer file then the file is marked as failed.

## 4.2 Test scripts

Please see attached .tbsl files and .ans files