SOIL
Simple Object Interaction Language

Final Report– August 10th, 2009

COMS W4115: Programming Languages and Translators
Professor Stephen A. Edwards

Richard Zieminski
rez2107@columbia.edu

Contents:

1 Introduction ... 3
2 Language Tutorial ... 3
2.1 Defining shapes and objects ... 3
2.2 Defining functions ... 4
3 Language Reference Manual ... 4
Conventions In This Document ... 4
3.1 Lexical Conventions ... 4
3.2 Comments ... 4
3.3 Whitespace ... 4
3.4 Line Breaks and Semicolons ... 4
3.5 Identifiers ... 4
3.6 Keywords and Reserved Words ... 5
3.7 Built in Functions ... 5
3.8 Built in Objects ... 5
3.9 Operators ... 6
3.10 Scope ... 6
3.11 Primitive Data Types ... 6
3.11.1 Number ... 6
3.11.2 Text ... 6
3.11.3 Object ... 7
3.11.4 Shape ... 7
3.12 Expressions ... 7
3.12.1 Additive Expressions ... 7
3.13 Declarations ... 7
3.14 Functions ... 8
3.15 Conditional Statement ... 8
3.16 Shape/Object Definitions ... 9
4. Project Plan ... 9
5. Timeline ... 10
6. Architecture ... 10
7. Testing ... 12
8. Lessons Learned ... 14
9. Code Listing ... 15

1 Introduction
SOIL is a computer language which can be used to teach the concepts of basic object interactions. Using a minimum of operations, a user can create simple objects and basic shapes, and then assign basic properties which characterize them. Simulations can then be run to see the outcome of the interactions between these objects.

2 Language Tutorial
A SOIL program consists of several sections:

1. Function declarations.
2. Global shape and object declarations.
3. Function 'main'

In order to avoid confusion, all variables are defined as extensions of shapes or objects. Access to the contained parameters is accomplished via the dereference operator '@'.

Example

```
circle@radius:=10;

toscreen ( circle@radius );
```

Assignment of a variable to a parameter is done using the assignment ':=' operator. This was done to avoid confusion with comparison operators, such as '=='.

2.1 Defining shapes and objects
Shapes and objects are defined in the exact same way, as shapes are basically a custom type of object. An optional assignment of a value is allowed during creation in order to simplify things. The 'shape' or 'object' keyword is necessary to distinguish the object type prior to the function name.

Example:

```
shape circle ( number radius );

object test ( number x:=1, text color:= "Blue" );
```

2.2 Defining functions

Function declaration is the same as for most languages, with the ‘function’ keyword required prior to the function name. Declaration of passed parameters is required. Passed parameters are accessed by reference and can change based on the function.

Example:

```
function test ( circle, square, test );
```

3 Language Reference Manual

Conventions In This Document

Text in italics type indicates a keyword or literal.

3.1 Lexical Conventions

3.2 Comments

Comments begin with the // character sequence and end with a line feed. Comments may be placed on the same line as source code. Multi-line comments will always begin with the // character sequence.

3.3 Whitespace

Whitespace characters which include spaces, tabs, and line feed characters may used to separate keywords, operators, and code tokens in the input but are discarded during parsing.

3.4 Line Breaks and Semicolons

Semicolons serve as a statement separator, and line breaks serve as a terminator. Multiple statements may be put on a single line of source code using semicolons in between each statement.

3.5 Identifiers

Identifiers represent the names of user defined variables and functions. All identifiers begin with a letter or underscore, followed by zero or more letters, digits, and underscores. Identifiers are case-sensitive. Identifiers can be any number of characters in length.

3.6 Keywords and Reserved Words

The following words are reserved as keywords and may not be used as identifiers. They are case sensitive. Valid keywords are:

if	*else*	
true	*false*	
run		
world		
function		
shape		*object*
for		
text	*number*	

3.7 Built in Functions

SOIL also contains built in functions which may not be redefined. Valid function names are:

a. *toscreen(expr)*

This function will handle combinations of text and numbers for output to the screen. It will parse the provided text for verbatim output (anything within "") and variables to be output as set.

Example:

x := 5;
toscreen "This is a simulation that will run " x " times";

Output:
This is a simulation that will run 5 times

3.8 Built in Objects

SOIL also contains built in objects which may not be redefined. Valid names are:

a. world

*world the extents of the interactive environment in 2 dimensional coordinates (x, y) and needs to be set before running any simulation. It is defined as:

world(*number* x *assign value, number* y *assign value*){
width = $1;
height = $2;
}

Once defined, objects can be passed as parameters to other functions and their elements referenced using the ‘@’ operator.

3.9 Operators

+	-	
/	*	
>	<	
>=	<=	
+=	-=	
==	!=	
@		* dereference fields of a shape
( )		
{ }		
:=		assignment operator
++		combines two strings, or a string and a number

3.10 Scope

There are two types of scope, local and global. Identifiers declared within a function are local only to that function and may not be used otherwise. Global identifiers which are declared outside any functions may be used anywhere in the program.

3.11 Primitive Data Types

Supported types will be text, number, object, and shape.

3.11.1 Number

number is a 32 bit whole number (+/-). Only whole numbers are supported.

3.11.2 Text

Text is a sequence of characters surrounded by double quotes. Text literals may not contain double quotes or span multiple lines.

3.11.3 Object

Object is a type that may contain any number of user defined fields of possible data types. All variable are associated with a defined object.

3.11.4 Shape

Shape is a type that may contain any number of user defined fields of possible data types, along with several pre-defined fields and functions.

Both *shape* and *object* variables can be accessed using the '@' operator.

Example:

```
shape circle ( number radius );

toscreen ( circle@radius );
```

3.12 Expressions

Expression can be a combination of operators, identifiers and literals. Upon evaluation, an expression will return a value. The value type is dependant on the expressions being combined. Precedence of expressions is as listed in the operators section of this document.

3.12.1 Additive Expressions

text + text = text
text + number = text
number + number = number
*number + text = number**

*If the text can be translated to a number this will hold true.

Only text and numbers can be combined.

3.13 Declarations

Declarations are used to assign a value (text, number) to an identifier. They have the form:

Identifier: = value;

A text declaration is defined by enclosing the value in quotes "".

3.14 Functions

Functions will be defined through the use of the ‘function’ keyword.

Functions have the form:

function identifier (*parameter-list*)
{ *body* }

or

function *identifier* ()
{ *body* }

where *parameter-list* = *shape* or *object*

A function does not return a value. The parameter-list will be of the form (*shape, object,* ..). Parameters are passed by reference and can be modified within the calling function.

Function nesting is supported, but recursive operations are not.

Functions can access global variables as well as arguments as well as locally declared variables.

3.15 Conditional Statement

There are two forms of the conditional statements:

a.
```
if ( expression )
{
        statement1;
        statement2;
        ...
}
else
{
        statement1;
        statement2;
        ...
};
```

b.
```
if ( expression )
{
```

```
        statement1;
        statement2;
        ...
    };
```

*Brackets are always used to enclose conditional statements.

3.16 Shape/Object Definitions

Shapes and objects may be created anywhere, even within a function, although they will be automatically destroyed upon leaving the function.

A shape is defined and created with the keyword:

```
shape identifier  (parameter-list );

where parameter-list =   number assign ID
                                   or
                          text assign ID
```

An object is defined and created with the keyword:

```
object identifier  (parameter-list );

where parameter-list =   number assign ID
                                   or
                          text assign ID
```

4. Project Plan

The project began with trying to come up with a language simple enough for a child to use, but powerful enough that an adult would still find it valuable. After watching my two year old son play with his toys, SOIL was born. The initial thought was the language could present rudimentary shapes in a visual format to help him learn.

The design process started with laying out the keywords necessary to provide basic functionality. I realized from the start that my keyword set was too elaborate therefore I downsized the keyword count to a handful to try to make the language even more user friendly and easy to learn.

The next step was to implement the scanner and parser and get a basic 'Hello World' type program to run. After much tweaking I was able to accomplish this. As I went along and added functionality, I proceeded to use test cases to regression test my work.

I then proceeded to tackle the largest task of trying to implement a c++ like object dereferencing scheme for object access alongside the standard single variable accessor method. Due to complexity issues I had to settle on a single dereference scheme for all variable access. While this somewhat added to the access method, the single solution provided for less confusion in the end.

The remaining steps would have been to implement the autonomous object interaction I originally proposed. I also would have liked to qualify the shape object better. Right now it is just a keyword, where object and shape are the same type. Unfortunately, the initial concept turned out to be way too much of an endeavor for one person to handle in a semester.

5. Timeline

My timeline is somewhat skewed due to my unfortunate circumstances encountered earlier this year. The gap between mid March and May should therefore be excluded.

Date	**Item**
January 20th - January 27th	Initial layout of project specifications
January 28th – February 10th	Work on language layout, 1st Proposal
February 11th - March 10th	Work on LRM, scanner layout
…	
June 16th – July 3rd	Development, scanner, parser
July 4th – August 1st	Get program working, testing
Week of August 3rd	Testing, Documentation, Final Report

6. Architecture

Soil consists of just a few parts. These are as follows:

scanner.mll: The scanner is used to convert characters and symbols into tokens (or language interpretable strings). Irrelevant details such as whitespace and comments are removed at this stage.

ast.mli: This is the interface the program exposes to the world. Types are also defined here.

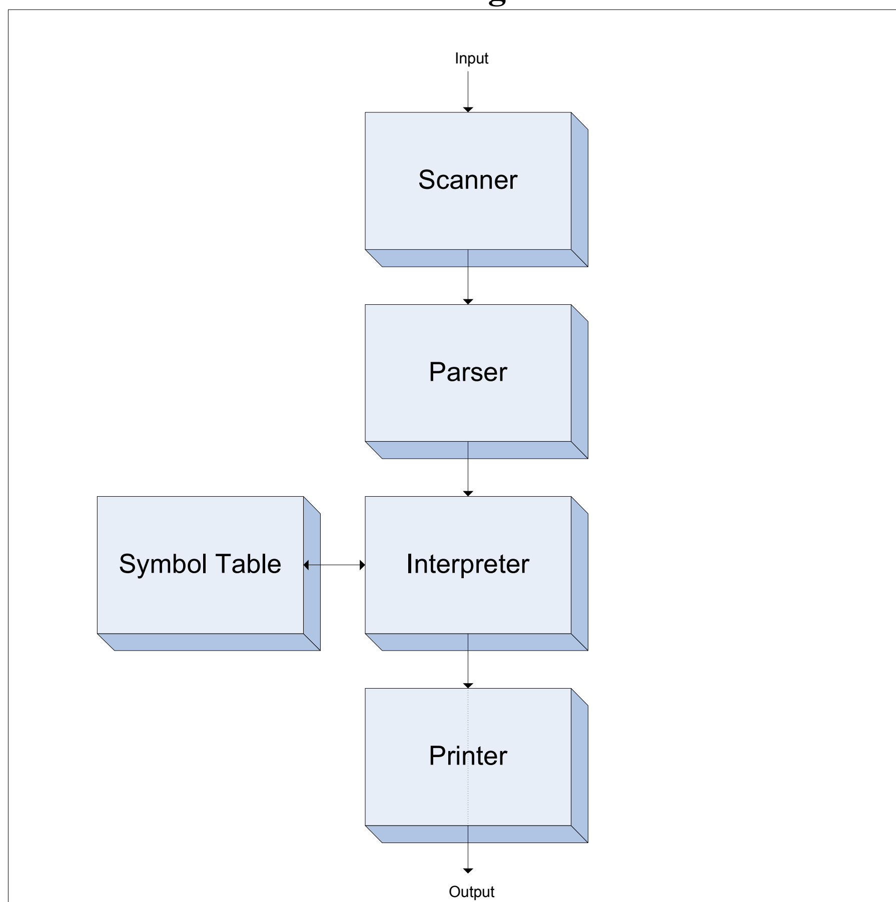
parser.mly: The parser reads the tokens generated from the scanner and generates an abstract syntax tree. It also checks to see that the file does not violate the rules that have been defined for the language.

interpret.ml: This is the main part of the program. Its job is to:

1. Go through the abstract syntax tree
2. Create a local and global symbol table
3. Check types
4. Evaluate expressions and well as resolve functions

printer.ml: This is used to output the translated file for verification of parsing. It is toggled on/off by setting the 'print' flag in the main 'soil.ml' file.

Block Diagram:



7. Testing

Testing consisted of creating a sub-directory of test (.mc) and output files (.out) and running a script to compile, run, and compare all the test files outputs against the expected results. For simplicity the 'testall.sh' script as provided for the *microc* example was modified and used. Tests cases were chosen to check the major functionality of the language. If given more time I would have liked to create many more test cases to cover all conditions which could occur.

Here is the output from the scripted testing:

```
test-for1...OK
test-if1...OK
test-object_number...OK
test-object_text...OK
test-object_wparm...OK
test-object_wparms...OK
test-ops...OK
test-run...OK
test-run2...OK
```

Example 1: Test-run.mc

```
shape circle(number radius:=10, text color:="Blue");
shape square(number length:=10, number height:=10, text color:="Red");

function test(shape1)
{
  toscreen(shape1@length);
}

function main()
{
  test(square);
}
```

Output:

```
10
```

Example 2: Test-ops.mc

```
// Comment test //
// Should print: //
// 11 //
// -1 //
// 5 //
// Testing1 //
// 8 //

object a(number x, text y);

function main()
{
  a@x:= 10;
  a@x:=a@x + 1;
  toscreen(a@x);

  a@x:= 10;
  a@x:=a@x - 11;
  toscreen(a@x);

  a@x:= 10;
  a@x:=a@x / 2;
  toscreen(a@x);

  a@y:= "Testing";
  a@y:= a@y ++ 1;
  toscreen (a@y);

  a@y:= "3";
  a@x:= a@x + a@y;
  toscreen (a@x);
}
```

Output:

```
11
-1
5
Testing1
8
```

8. Lessons Learned

Without knowing much about Ocaml, I set out to design a language which I though would be useful, yet not too complex. Shortly after starting on the project I realized I had promised too much. The complexities associated with learning a new language, especially one so different that I'd become familiar with, tended to overshadow the development. In the end I was able to deliver a subset of the original design, with similar functionality and ease of use. I highly recommend getting familiar with the syntax of the language very early in the design. A lot of lost time was due to misunderstanding of language functionality encountered along the way. If it were not for the 'microc' example as a foundation I do not think I could have developed a compiler in the time given. The learning curve is just too steep.

Overall I learned a lot about the inner workings of a compiler. I enjoyed learning a new, fundamentally different type of language and I am glad that professor Edwards had chosen to challenge us by switching from Java to Ocaml. Seeing things in a different way allows us to be better programmers in the end.

9. Code Listing

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "//"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| ":="     { ASSIGN }
| "@"     { DEREFERENCE }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "++"     { COMBINE }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "function" { FUNCTION }
| "shape"  { SHAPE }
| "object" { OBJECT }
| "number" { NUMBER }
| "text"   { TEXT }
| "toscreen"  { TOSCREEN }
| "world" { WORLD }

| ['0'-'9']+ as lxm { LITERAL(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '\"' [^ '\"']* '\"' as lxm { STR(lxm) }  | _ as char { raise (Failure("illegal
character " ^ Char.escaped char)) }
| eof { EOF }

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

ast.mli

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
Combine

type objectexpr =          (* Object Expressions *)
  Literal of string
  | Str of string
  | Access of string * string
  | Binop of objectexpr * op * objectexpr
  | AssignToObject of string * string * objectexpr
  | Noexpr

type expr =           (* Expressions *)
  Id of string
  | Assign of string * string * objectexpr
  | Call of string * expr list

type stmt =
    Block of stmt list
  | Expr of expr
  | If of objectexpr * stmt * stmt
  | For of objectexpr * objectexpr * objectexpr * stmt
  | ToScreen of objectexpr

type p_decl = {
  key    : string;
  value  : string;
}

type v_decl = {
  vartype     : string;
  varname     : string;
  varparams   : p_decl list;
}

type func_decl = {
    fname : string;
    formals : string list;
    locals : v_decl list;
    body : stmt list;
  }

  type program = v_decl list * func_decl list
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ COMBINE
%token IF ELSE FOR
%token NUMBER TEXT
%token <string> ID
%token <string> LITERAL
%token <string> STR
%token EOF
%token FUNCTION SHAPE OBJECT
%token WORLD
%token DEREFERENCE
%token TOSCREEN

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left COMBINE
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
  /* nothing */        { [], [] }
  | program vdecl      { ($2 :: fst $1), snd $1 }
  | program fdecl      { fst $1, ($2 :: snd $1) }

fdecl:
   FUNCTION ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $2;
       formals = $4;
       locals = List.rev $7;
       body = List.rev $8 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

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

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

```
vdecl:
  OBJECT ID LPAREN p_opt RPAREN SEMI { {
    vartype = "Object"; varname = $2;  varparams = $4;} }
  | SHAPE ID LPAREN p_opt RPAREN SEMI { {
    vartype = "Shape"; varname = $2;  varparams = $4;} }
  | WORLD LPAREN p_opt RPAREN SEMI { {
    vartype = "World"; varname = "World";  varparams = $3;} }

p_opt:
  /* nothing */                  { [] }
  | p_list                       { $1 }

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

param:
  NUMBER ID ASSIGN LITERAL    { { key   = $2;
    value = $4;} }
  | TEXT ID ASSIGN STR    { {   key   = $2;
    value = String.sub $4 1 ((String.length $4)-2) ;} }
  | NUMBER ID     { {   key   = $2;
    value = "0";} }
  | TEXT ID     { {   key   = $2;
    value = "";} }


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

stmt:
  expr SEMI { Expr($1) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN objectexpr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN objectexpr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN objectexpr SEMI objectexpr SEMI objectexpr RPAREN stmt
     { For($3, $5, $7, $9) }
  | TOSCREEN objectexpr SEMI      { ToScreen($2) }

expr_opt:
    /* nothing */ { Noexpr }
  | objectexpr          { $1 }

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

objectexpr:
  LITERAL                                     { Literal($1) }
  | STR                                       { Str($1) }
  | objectexpr PLUS objectexpr                { Binop($1, Add, $3) }
  | objectexpr MINUS objectexpr               { Binop($1, Sub, $3) }
  | objectexpr TIMES objectexpr               { Binop($1, Mult, $3) }
  | objectexpr DIVIDE objectexpr              { Binop($1, Div, $3) }
```

```
  | objectexpr EQ objectexpr                  { Binop($1, Equal, $3) }
  | objectexpr NEQ objectexpr                 { Binop($1, Neq, $3) }
  | objectexpr LT objectexpr                  { Binop($1, Less, $3) }
  | objectexpr LEQ objectexpr                 { Binop($1, Leq, $3) }
  | objectexpr GT objectexpr                  { Binop($1, Greater, $3) }
  | objectexpr GEQ objectexpr                 { Binop($1, Geq, $3) }
  | objectexpr COMBINE objectexpr             { Binop($1, Combine,  $3) }
  | ID DEREFERENCE ID ASSIGN objectexpr       { AssignToObject($1, $3, $5) }
  | ID DEREFERENCE ID                         { Access($1, $3) }
  | LPAREN objectexpr RPAREN                  { $2 }

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

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

interpret.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
    let initall = List.fold_left
    (fun globals vdecl ->
      let params = List.fold_left
      (fun param_map param_decl -> NameMap.add param_decl.key param_decl.value
param_map)
      NameMap.empty vdecl.varparams in

        if NameMap.mem vdecl.varname globals then
          raise (Failure ("variable " ^ vdecl.varname ^ " is defined more than
once!"));
          NameMap.add vdecl.varname params globals;
        )
  in
  let rec call fdecl actuals globals =
    (* Evaluate an expression and return (value, updated environment) *)
    (**** This is the object eval section ****)
    let rec eval_object env = function
      Literal(i) -> i, env
      | Noexpr -> "Nothing", env (* must be non-zero for the for loop predicate *)
      | Str(str) -> (String.sub str 1 (( String.length str)-2)), env
      | Binop(e1, op, e2) ->
        let v1, env = eval_object env e1 in
          let v2, env = eval_object env e2 in
          let bool_to_str i = if i then "true" else "false" in
          (match op with
            Add -> string_of_int(int_of_string(v1) + int_of_string(v2))
            | Sub -> string_of_int(int_of_string(v1) - int_of_string(v2))
            | Mult -> string_of_int(int_of_string(v1) * int_of_string(v2))
            | Div -> string_of_int(int_of_string(v1) / int_of_string(v2))
            | Equal -> bool_to_str(int_of_string(v1) == int_of_string(v2))
            | Neq -> bool_to_str(int_of_string(v1) != int_of_string(v2))
            | Less -> bool_to_str(int_of_string(v1) < int_of_string(v2))
            | Leq -> bool_to_str(int_of_string(v1) <= int_of_string(v2))
            | Greater -> bool_to_str(int_of_string(v1) > int_of_string(v2))
```

```
            | Geq -> bool_to_str(int_of_string(v1) >= int_of_string(v2))
            | Combine -> String.concat "" (v1::(v2::[]))), env
      | Access(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))
      | AssignToObject(var, param, e) ->
        let v, (locals, globals) = eval_object 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) *)
    (**** This is the normal eval section ****)
    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) = eval_object 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))
        | 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 = eval_object env e in
                  exec env (if String.compare "true" v==0 then s1 else s2)
                | For(e1, e2, e3, s) ->
                  let _, env = eval_object env e1 in
                  let rec loop env =
                    let v, env = eval_object env e2 in
                    if String.compare "true" v==0 then
                      let _, env = eval_object (exec env s) e3 in
                      loop env
                    else
                      env
                  in loop env
                | ToScreen(e) ->
                  let str, env = eval_object env e in
                    (match (str) with
                    | _ -> print_endline str ; flush stdout; env)
    in
      (* Enter the function: bind actual values to formal arguments *)
      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 passed to " ^ fdecl.fname))
      in
        (* Initilize local variables *)
        let locals = initall locals fdecl.locals in
          (* Execute each statement; return updated global symbol table *)
          snd (List.fold_left exec (locals, globals) fdecl.body)
      in
        (* add global variables to symbol table. *)
        let globals = initall NameMap.empty vars in
  try
    call (NameMap.find "main" func_decls) [] globals
    with Not_found -> raise (Failure ("did not find the main() function"))
```

printer.ml

```
open Ast

let rec string_of_expr = function
  | Id(s) -> s
  | Assign(var, param, e) -> ""
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"

let rec string_of_objectexpr = function
  Literal(s) -> s
  | Noexpr -> ""
  | Str(s) -> s
```

```
  | Binop(e1, o, e2) ->
      string_of_objectexpr e1 ^ " " ^
      (match o with
          Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | Combine -> "++" ) ^ " " ^
      string_of_objectexpr e2
  | Access(var, param)-> var ^ "@" ^ param
  | AssignToObject(v, param, e) -> v ^ "@" ^ param ^ "=" ^ string_of_objectexpr e
let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_objectexpr e ^ ")\n" ^ string_of_stmt
s
  | If(e, s1, s2) ->  "if (" ^ string_of_objectexpr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_objectexpr e1  ^ " ; " ^ string_of_objectexpr e2 ^ " ; "
^
      string_of_objectexpr e3  ^ ") " ^ string_of_stmt s
  | ToScreen(s) -> "toscreen (" ^ string_of_objectexpr s ^ " )\n"

let string_of_vdecl id = "int " ^ id ^ ";\n"

let list_of_vdecl vars =
""

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map list_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map list_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

soil.ml

```
let print = false

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
        if print then
          let listing = Printer.string_of_program program in
            print_string listing
          else
            ignore (Interpret.run program);
  done
```