Examination Generation Grading Language (EGGL)

Final Project Report

Gordon Hew (CVN) (gh2242@columbia.edu)

COMS W4115: Programming Languages and Translators (PLT)
Fall 2010, Professor Stephen Edwards

Introduction

Examinations have been used as a measure to gauge an individual's mastery over a particular skill. Depending on the nature of the skill that is being assessed, an exam format may vary. As most students can attest, the majority of examinations follow a basic question and answer format. In general, instructors create exams in a WYSWIG text editor such as Word and grade a student's answers by hand. This process can be error prone and requires a large amount of human effort. This is largely due to instructors using tools that are not designed for creating exams.

The Examination Generation Grading Language (EGGL) seeks to provide a simple language that can easily facilitate the creation and scoring of computerized exams. Such a language would be beneficial to educators in that it would enable them to quickly write an interactive exam that can be scored instantly. Additionally, sophisticated users can use EGGL's control flow constructs to add dynamic functionality to their exams.

Language Tutorial

This section will walk you through creating a simple EGGL program.

Create a file with notepad and save it as test.eggl.

In the file, add the code below. Every EGGL program needs a main function to execute.

```
main()
{
	println("hello world!");
}
```

Run the following command in the directory where you saved test.eggl: ./eggl "test.eggl", your output should be: `hello world!`

Add a new function to test.eggl:

```
question1()
```

```
{
      prompt(“What is your favorite color?”)
      answer(“red”);
      choice(“blue,green,red,yellow”);
}
```
Add the function call to the main() function. This function will prompt the user for their favorite color and give them a number of choices from which the user can input. If they type in the correct input, they will receive a score of 100%.

Running the program again should yield the below output with user input in bold.
```
hello world!
Question: What is your favorite color?
Choices:
      blue
      green
      red
      yellow
User Choice: red
Score:100.%
```

This simple program demonstrates how EGGL can be used to quickly create simple exams that automatically score themselves.

Language Manual

Lexical Convention

Comments

EGGL supports single line comments. The sequence of `/*` indicates the start of a comment and a subsequent `*/` indicates the end of the comment. Anything within the bounds of the comment declaration will not be processed by the compiler.
Example: `/* This is a comment */`

Identifiers

An identifier is the name of a variable, constant, or a function declared in EGGL. A variable name can only consist of alphanumeric characters and must begin with a letter. The use of symbols and special characters are not permitted.

Reserved Keywords

The following keywords are reserved and could not be used for variable or function names:

- print
- println
- prompt
- choice

- answer
- float
- if
- else
- for
- while
- return
- true
- false

Separators

There are two separators in the EGGL language, they are commas (,) and semi-colons (;). A comma is used for declaring a sequence of items such as in an array. Semi-colons are used to indicate the end of a statement.

White-spaces

White-spaces such as tabs, carriage returns, and new lines are ignored during compilation. Spaces are used to identify keywords and variable declarations.

Types

The following data types are supported by EGGL:

boolean

The boolean data type is used for declaring a boolean value which can either be true or false.

- Declaration Example:

```
boolean x;
x = true;
```

int

The int data type is used for declaring a 32-bit signed integer.

- Declaration Example:

```
int x;
x = 9;
```

float

The float data type is used for declaring a floating point number.

- Declaration Example:

```
float x;
x = 5.5;
```

string

The string data type is used for declaring a string.

- Declaration Example:

```
string x;
x = "cat";
```

Operators

Arithmetic Operators

- '+': addition
- '-': subtraction
- '*': multiplication
- '/': division

Relational Operators

- '<': less than
- '<=': less than or equal to
- '>': greater than
- '>=': greater than or equal to

Equality Operators

- '==': equal to
- '!=': not equal

Logical Operators

- '&&': and
- '||': or

String Operators

- '^': concatenates two strings together

Expressions

All expressions group left to right.

Primary Expressions

```
primary_expression → int string-expression
                   | float string-expression
                   | string string-expression
```

A primary expression is an identifier.

Arithmetic Expressions

```
arithmetic_expression → expression + expression
                      | expression - expression
                      | expression * expression
                      | expression / expression
```

An `arithmetic_expression` is only valid when `expression` evaluates to a float or int.

Relational Expressions

```
relational_expression → expression < expression
                      | expression <= expression
                      | expression > expression
                      | expression => expression
```

The evaluation of the operators '<', '<=', '>', and '=>' returns true or false. A `relational_expression` is only valid when `expression` evaluates to an int, float, or string and are of the same type on both sides of the operator.

Equality Expressions

```
equality_expression → expression == expression
                      | expression != expression
```

The evaluation of the relational operators '==' and '!=' returns true or false. An `equality_expression` is only valid when `expression` evaluates to an int, float, string, or boolean and are of the same type on both sides of the operator.

Logical Expressions

```
logical_term → relational_expression | equality_expression
logical_expression → logical_term && logical_term
                      | logical term || logical_term
```

The evaluation of the logical operators '&&' and '||' returns true or false based on AND/OR truth table logic.

Mutation Expressions

```
mutation_expression → expression ^ expression
```

A `mutation_expression` is only valid when one of the `expression` values is a string.

Statements

Expression Statement

```
expression;
```

The most common and basic statement is an expression statement.

Compound Statement

```
{
      statement
      statement
}
```

A compound statement is a list of statements to be evaluated.

Conditional Statement

```
if(expression)
```

```
{
      statement
}
else if (expression)
{
      statement
}
else
{
      statement
}
```
A conditional statement is used to evaluate if-else logic control logic.

For Loop Statement

```
for(expression-1; expression-2; expression-3)
{
      statement
}
```
The for-loop statement is used to run a `statement` until a condition is no longer met. `expression-1` is the initial condition, `expression-2` is the condition in which to continue, `expression-3` is the mutation of the initial condition to a new value.

While Loop Statement

```
while(expression)
{
      statement
}
```
The while-loop statement is used to run a `statement` until the `expression` is no longer met. The expression should only evaluate to true or false.

Return Statement

```
return(expression);
```
The return statement is the value that is returned from a function. `expression` must match the data type that the function is declared to return.

Prompt Statement

```
prompt(expression);
```
A prompt statement is required for question-functions and can only be declared once inside a function. The `expression` must be a string data type in the form of a text question.

Answer Statement

```
answer(expression);
```
An answer statement is required for question-functions and can only be declared once inside a function.

Choice Statement

```
choice(string-expression);
```

A choice statement is optional for question-functions. It provides a discrete set of choices that can be selected to match the answer of a question-function. The expression must be a string of tokens delimited by commas (i.e. “choice1,choice2,choice3”).

Function Definitions

Functions

```
function-name( param-1, ... )
{
      statement*
}
```

A basic function definition is composed of a `return-data-type` (the data type to be returned), the `function-name` (name of the function), and the parameters to be passed into the function. Inside the function body can be any number of statements. The last statement that must be called is a `return-statement` if the `return-data-type` is any value other than void.

In order to create a question properly, a function must have a prompt-statement, answer-statement, and a choice-statement otherwise the EGGL program will not run in an expected fashion.

Each program must declare a main function from where other functions may be called:

```
main()
{
      statement*
}
```

Built-in Functions

There are several built-in functions in EGGL library as defined below:

- print(expression): evaluates expression and prints it to standard out.
- println(expression): evaluates expression and prints it to standard out and adds a new line at the end.

Scope

EGGL uses static scoping and has its scopes separated by blocks which are encapsulated by curly braces { ... }.

Project Plan

Process

The development team followed an iterative approach in developing EGGL. Using the the Language Reference Manual (LRM) that was drafted earlier in the semester, the team divided the requirements into features. Each time a feature was developed, it would be tested manually and then have a regression test associated with it. The regression test would ensure that the feature continued to work as new features are incrementally added. Once a feature and its regression test is implemented, the files are committed into a Subversion repository and then branched.

Project Timeline / Log

Item	Task	Start Date	End Date	Status
1	**Project Infrastructure**			
1.1	Setup Eclipse IDE with OCal IDE Plugin	2010.12.04	2010.12.04	100%
1.2	Setup Subversion	2010.12.04	2010.12.04	100%
2	**Initial Development Setup**			
2.1	Create Make file	2010.12.04	2010.12.05	100%
2.2	Create test script	2010.12.04	2010.12.05	100%
2.3	Create abstract syntax tree	2010.12.04	2010.12.05	100%
2.4	Create Parser	2010.12.04	2010.12.05	100%
2.5	Create Scanner	2010.12.04	2010.12.05	100%
2.6	Create Interpreter	2010.12.04	2010.12.05	100%
2.7	Create start program	2010.12.04	2010.12.05	100%
3	**Features**			
3.1	Add string data type	2010.12.13	2010.12.13	100%
3.2	Add float data type	2010.12.14	2010.12.14	100%
3.3	Add boolean data type	2010.12.15	2010.12.15	100%
3.4	Add/modify existing binary operations	2010.12.16	2010.12.16	100%
3.5	Add prompt builtin function	2010.12.17	2010.12.17	100%

3.6	Add choice builtin function	2010.12.18	2010.12.18	100%
3.7	Add answer builtin function	2010.12.18	2010.12.18	100%
3.8	Implement automatic scoring facility	2010.12.19	2010.12.19	100%
3.9	Add/modify existing control-flow (i.e. if), for, while, and assignment functionality	2010.12.19	2010.12.19	100%
3.10	Add array data type	???	???	0%
3.12	Add negative unary operator	???	???	0%
3.13	Add not unary operator	???	???	0%
3.14	Implement weight meta-data annotation for weighted scoring.	???	???	0%
3.15	Implement difficulty meta-data annotation for adaptive display of questions.	???	???	0%
3.16	Implement appearance meta-data annotation for ordered display of questions	???	???	0%
4	Final Project Report	2010.12.20	2010.12.22	100%

The incomplete items have been removed from the Language Reference Manual section of this text.

Roles

The project team consisted of a single member, Gordon Hew, who designed, developed, and tested EGGL.

Development Environment

Operating System: Mac OS X 10.6.5
IDE: Eclipse Helios Service Release 1 with OCal IDE and Subclipse Plugin
Language: OCaml
Repository: Subversion

Architectural Design

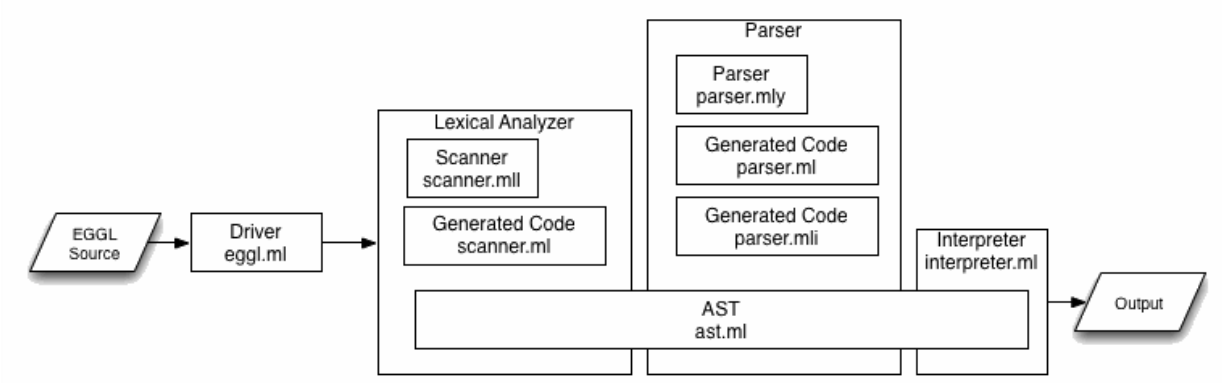

Figure 1: Component Architecture

Components

Driver

The component that reads is invoked via command line and reads in the source EGGL code into the EGGL language processor.

Lexical Analyzer

The Lexical Analyzer consists of a set of regular expressions (scanner.mll) and auto-generated code (scanner.ml) from ocamllex which is used to generate a stream of tokens.

Parser

The Parser consists of a grammar specification (parser.mly) and auto-generated code (parser.ml and parser.mli) from ocamlyacc which translates the tokens into a semantic action.

Interpreter

The Interpreter (interpreter.ml) defines the behavior that a semantic action is attempting to invoke.

Workflow

1. The Driver reads in a EGGL source code.
2. The Lexicial Analyzer generates a stream of tokens from the source code based on the set of regular expressions that define different token attributes.
3. The Parser reads in the stream of tokens and associates them to semantic actions via a defined grammar.
4. The Interpreter reads in the semantic actions and invokes the appropriate behavior of the command which then outputs to the console.

Test Plan

The EGGL test plan consists of two types of test scripts. The first type of test attempts to assert whether a particular EGGL construct such as a for-loop or if-statement works properly. The second type of test involves interactive user input which can be fed into the program through standard input. The representative tests that are shown in this document are of the latter as they depict a full EGGL program and how a developer might leverage the language.

Basic Exam Test

The Basic Exam Test seeks to assert that the question, answer, correction, and scoring facility of an exam functions properly. Test verification consists of running the source code, feeding in the input file via redirecting standard input, and comparing the differences between the generated output and the expected output.

Source Code (test-exam-1.eggl)

```
question1()
{
	prompt("Is the sky blue?");
	answer("yes");
	choice("yes,no");
}

question2()
{
	prompt("What color is my face?");
	answer("red");
	choice("green,purple,blue,red");
}

question3()
{
	prompt("1 + 5 = ?");
	answer("6");
	choice("5,16,6,10");
}

question4()
{
	prompt("5 * 10 = ?");
	answer("50");
	choice("50,3,60,120");
}

main()
{
	question1();
```

```
	question2();
	question3();
	question4();
}
```

Input (test-exam-1.in)

```
yes
red
6
50
```

Output (test-exam-1.out)

It is important to note that in the output, we will not see the standard input feed.

```
Question: Is the sky blue?
Choices:
	yes
	no
User Choice: Question: What color is my face?
Choices:
	green
	purple
	blue
	red
User Choice: Question: 1 + 5 = ?
Choices:
	5
	16
	6
	10
User Choice: Question: 5 * 10 = ?
Choices:
	50
	3
	60
	120
User Choice: Score:100.%
```

Dynamic Exam Test

In addition to asserting that the question, answer, correction, and scoring facility of an exam functions properly, the Dynamic Exam Test seeks to demonstrate control flow and how exam questions can be created dynamically. Test verification consists of running the source code, feeding in the input file via redirecting standard input, and comparing the differences between the generated output and the expected output.

Source Code (test-exam-1.eggl)

```
questionDynamic()
```

```
{
  int i;
  string choice;

  for (i = 0 ; i < 5 ; i = i + 1) {

    prompt("What is 5 * " ^ i ^ "= ?");

      answer(5*i);

      choice("0,5,10,15,20,25");
  }
}

main()
{
      questionDynamic();
}
```

Input (test-exam-1.in)

```
0
5
10
20
20
```

Output (test-exam-1.out)

It is important to note that in the output, we will not see the standard input feed.

```
Question: What is 5 * 0= ?
Choices:
      0
      5
      10
      15
      20
      25
User Choice: Question: What is 5 * 1= ?
Choices:
      0
      5
      10
      15
      20
      25
User Choice: Question: What is 5 * 2= ?
Choices:
      0
```

```
      5
      10
      15
      20
      25
User Choice: Question: What is 5 * 3= ?
Choices:
      0
      5
      10
      15
      20
      25
User Choice: Question: What is 5 * 4= ?
Choices:
      0
      5
      10
      15
      20
      25
User Choice: Score:80.%
```

Retrospective

Lessons Learned

There are two primary lessons that have been learned during the course of the project. The first is that time management is a crucial component to any software project. If the development of EGGL had started earlier in the semester than later, all the development goals may have been realized.

The second lesson is that I should have thought more carefully about what the language is trying to accomplish and the construct types that would maximize the effectiveness of EGGL. I believe that some of the more advanced constructs that I was not able to implement in time would have been better served by smaller and nimbler language features.

Future Advice

As a note for future students of PLT, developing a language is not an easy task. There is the hurdle of learning and understanding OCaml which is unlike some of the main stream languages (i.e. Java, C++, etc.). Additionally, developing a language requires more finesse than developing a utility library as it requires more consideration and careful planning. The best advice for a student embarking on a similar project is to start as early as possible.

References

The MicroC source code and tests written by Professor Stephen Edwards was used as a baseline implementation from which to develop a language using OCaml. It was used as a guide and adapted in the development of EGGL.
MicroC Source: http://www.cs.columbia.edu/~sedwards/classes/2010/w4115-fall/microc.tar.gz

Appendix

ast.ml

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

type expr =
            LiteralInteger of int
      | LiteralString of string
      | LiteralBool of bool
      | LiteralFloat of float
      | Id of string
      | Binop of expr * op * expr
      | Assign of string * expr
      | Call of string * expr list
      | Noexpr

type stmt =
            Block of stmt list
      | Expr of expr
      | Return of expr
      | If of expr * stmt * stmt
      | For of expr * expr * expr * stmt
      | While of expr * stmt

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

type program = string list * func_decl list
```

eggl.ml

```
let _ =
      let file = Sys.argv.(1) in
            let lexbuf = Lexing.from_channel (open_in file) in
            let program = Parser.program Scanner.token lexbuf in ignore
(Interpret.run program)
```

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 * string NameMap.t

(* Stores the answers in reverse sequential order *)
let answerList = ref [];;

(* Stores the user choices in resverse sequential order *)
let userChoiceList = ref [];;

(* Removes quotes from the end of a string *)
let rec remove_double_quotes s =
      if( String.get s (String.length s - 1) = '"' &&  String.get s 0 = '"')
then
            String.sub s 1 (String.length s - 2)
      else
            s;
;;

(* Converts a , delimited string to a list *)
let rec string_to_list x y =
      try
            let i = (String.index y ',' ) in
                  string_to_list ((String.sub y 0 i) :: x) (String.sub y (i+1)
((String.length y) - (i + 1)))
  with Not_found -> y :: x
;;

(* Determines the number of questions that are correct *)
let rec answeredCorrectly correct answers selection =
      if( List.length answers >= 1 && List.hd answers = List.hd selection)
then
            answeredCorrectly (correct + 1) (List.tl answers) (List.tl
selection)
      else if( List.length answers >= 1 && List.hd answers != List.hd
selection) then
            answeredCorrectly (correct) (List.tl answers) (List.tl selection)
      else
            correct
;;
```

```
(* Calculates the percentage of problems correct *)
let rec percentageCorrect answers selection =
      ((float_of_int (answeredCorrectly 0 answers selection)) /. float_of_int
(List.length answers)) *. 100.0;;

(* 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 -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

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

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
                        LiteralInteger(i) -> string_of_int(i), env
      | LiteralString(t) -> t, env
                  | LiteralBool(b) -> if (b = true) then "1", env else "0",
env
                  | LiteralFloat(f) -> string_of_float(f), env
                  | Noexpr -> string_of_int(1), env (* must be non-zero for
the for loop predicate *)
                  | 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))
      | Binop(e1, op, e2) ->
                           let v1, env = eval env e1 in
                let v2, env = eval env e2 in
                                      let boolean i = if i then "1" else "0"
in
                                            (match op with
                                              Add ->
                                                           if
(String.contains v1 '.') && (String.contains v2 '.') then
      string_of_float((float_of_string v1) +. (float_of_string v2))
                                                           else
      string_of_int((int_of_string v1) + (int_of_string v2))
                                                    | Sub ->
                                                           if
(String.contains v1 '.') && (String.contains v2 '.') then
      string_of_float((float_of_string v1) -. (float_of_string v2))
```

```
                                                        else
      string_of_int((int_of_string v1) - (int_of_string v2))
                                            | Mult ->
                                                        if
(String.contains v1 '.') && (String.contains v2 '.') then
      string_of_float((float_of_string v1) *. (float_of_string v2))
                                                        else
      string_of_int((int_of_string v1) * (int_of_string v2))
                                            | Div ->
                                                        if
(String.contains v1 '.') && (String.contains v2 '.') then
      string_of_float((float_of_string v1) /. (float_of_string v2))
                                                        else
      string_of_int((int_of_string v1) / (int_of_string v2))
                                            | Equal -> boolean (v1 = v2)
                                            | Neq -> boolean (v1 != v2)
                                            | Less -> boolean ((int_of_string
v1) < (int_of_string v2))
                                            | Leq -> boolean ((int_of_string
v1) <= (int_of_string v2))
                                            | Greater -> boolean
((int_of_string v1) > (int_of_string v2))
                                            | Geq -> boolean ((int_of_string
v1) >= (int_of_string v2))
                                                | And -> boolean
(((int_of_string v1) = 1) && ((int_of_string v2) = 1))
                                                | Or -> boolean
(((int_of_string v1) = 1) || ((int_of_string v2) = 1))
                                                | Concat ->
(remove_double_quotes v1) ^ (remove_double_quotes v2)
                                               ), env
                  | Assign(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("print", [e]) ->
                    let v, env = eval env e in print_string
(remove_double_quotes v);
                        "0", env
                  | Call("println", [e]) ->
                    let v, env = eval env e in print_endline
(remove_double_quotes v);
                        "0", env
                  | Call("prompt", [e]) ->
                        print_string "Question: ";
                    let v, env = eval env e in print_endline
```

```
(remove_double_quotes v);
                        "0", env
                  | Call("answer", [e]) ->
                    let v, env = eval env e in answerList :=
(remove_double_quotes v) :: !answerList;
                        "0", env
                  | Call("choice", [e]) ->
                        print_endline "Choices:";
                    let v, env = eval env e in
                              let choices = (List.rev (string_to_list []
(remove_double_quotes v))) in
                                    List.iter (fun x -> print_endline ("\t" ^
x)) choices;
                        print_string "User Choice: ";
                        let str = read_line () in
                              userChoiceList := str :: !userChoiceList;
                        "0", 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) (List.rev
actuals)
                        in
                        let (locals, globals) = env in
                        try
                        let globals =
                                        call fdecl actuals globals in "0",
(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 env e in exec
env (if (int_of_string v) != 0 then s1 else s2)
                        | While(e, s) ->
                                    let rec loop env =
                                      let v, env = eval env e in
                                        if (int_of_string v) != 0 then loop
```

```
(exec env s) else env
                                                    in loop env
                        | For(e1, e2, e3, s) ->
                                    let _, env = eval env e1 in
                                          let rec loop env =
                                            let v, env = eval env e2 in
                                                  if (int_of_string v) != 0
then
                                                    let _, env = eval (exec
env s) e3 in
                                                    loop env
                                                  else
                                                    env
                                                          in loop env
                        | Return(e) ->
                                    let v, (locals, globals) = eval env e in
                                    raise (ReturnException(v, globals))
                      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
                (* Initialize local variables to 0 *)
                let locals = List.fold_left
                             (fun locals local -> NameMap.add local "0"
locals) locals fdecl.locals
                in
                (* Execute each statement in sequence, return updated global
symbol table *)
                snd (List.fold_left exec (locals, globals) fdecl.body)

                    (* Run a program: initialize global variables to 0, find
and run "main" *)
                    in let globals = List.fold_left
                        (fun globals vdecl -> NameMap.add vdecl "0" globals)
NameMap.empty vars
                    in try
                              let main = call (NameMap.find "main" func_decls)
[] globals
                                    in
                                          if(List.length !answerList > 0) then
                                                print_endline ("Score:" ^
(string_of_float (percentageCorrect !answerList !userChoiceList) ^ "%"))
```

```
                              with Not_found -> raise (Failure ("did not find
the main() function"))
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN CONCAT
%token EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE INT STRING BOOLEAN FLOAT
%token <int> LINTEGER
%token <string> LSTRING
%token <bool> LBOOL
%token <float> LFLOAT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%right CONCAT
%left EQ NEQ
%left LT GT LEQ GEQ AND OR
%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:
   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 }

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

vdecl:
   INT ID SEMI { $2 }
      | STRING ID SEMI { $2 }
      | BOOLEAN ID SEMI { $2 }
      | FLOAT ID SEMI { $2 }

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 expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    LINTEGER   { LiteralInteger($1) }
  | LSTRING    { LiteralString($1) }
  | LBOOL      { LiteralBool($1) }
  | LFLOAT     { LiteralFloat($1) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr CONCAT expr { Binop($1, Concat,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
```

```
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,   $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

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

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

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 }
| '^'      { CONCAT }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "||"     { OR }
| "&&"     { AND }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "string" { STRING }
| "boolean" { BOOLEAN }
```

```
| "float"  { FLOAT }
| "true"   as lxm { LBOOL(bool_of_string lxm) }
| "false"  as lxm { LBOOL(bool_of_string lxm) }
| ['0'-'9']+['.']['0'-'9']+ as lxm { LFLOAT(float_of_string lxm) }
| ['0'-'9']+ as lxm { LINTEGER(int_of_string lxm) }
| ['"']
['a'-'z' 'A'-'Z' '0'-'9' '_' ' ' '!' '@' '#' '$' '%' '^' '&' '*' '(' ')' '{'
'}' '|' ';' '<' '>' '.' ',' '?' '/' '\n' '+' '-'  '=' '\\' ''' ]*['"'] as lxm
{ LSTRING(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_'  ]* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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