MatrEL:
Matrix Entertainment Language

Rochelle Palting
rcp2122
Columbia University
COMS 4115 Fall 2008

Table of Contents

1 Introduction

The MatrEL programming language is designed to help developers implement single or multiplayer board games. Its syntax is similar to C providing a few simple types and board game tailored methods allowing the language to be easy to learn and use.

MatrEL is intended to be a compact language containing just enough functionality for a developer to create a fun and challenging game without unnecessary functionality and features that could make the language more difficult and confusing to use.

1.1 Background

The entertainment that board games have provided over generations has been and continues to be a favorite pastime. With the ongoing technology innovations, gaming has taken on new forms and exists on various media from internet games, handheld players, and game consoles. MatrEL allows the developer to bring the old time favorite games into the current times allowing the player to play these games in electronic form. The developer can also try out his or her inventive style by creating a new and fascinating game. In both cases, MatrEL is fun for the developer and the game player.

1.2 Goals of MatrEL

MatrEL was designed with the goals and objectives of being intuitive and optimal performance.

Since MatrEL is designed with the basics of board game creation in mind, the programmer will focus on the rules and dynamics of the game rather than get lost in the language specifics.

The objects and data structures that make up MatrEL have been chosen to provide optimal search and store functions so that developing, testing and execution of programs do not lag in time and performance.

2 Language Tutorial

Before creating a board game using MatrEL, the developer should have a clear understanding of the game to be developed. Specifically, the following details should be decided:

- How many players can play the game?
- How big can the board game be?
- What are the allowable moves by each player?
- Under what conditions does a player win or lose at the game?

MatrEL uses a square matrix object to simulate the board game. The matrix consists of rows and columns which each contain cells that represent a square on the game board. A program is created by performing a sequence of sets and queries on these cells. Based on the predefined win/lose rules, the game sequence continues until there is a clear win or lose to the game.

While each game created will be unique in its own way, the basic steps that the developer will follow to program the game will be the same:

1. Create win/lose rules.
    a. Create a boolean function that takes a game board position as input and returns true or false as to whether that new position resulted in a win or lose situation for the player.
2. Initialize game variables: number of players and game board size.
    a. In MatrEL it is required that the game board be a square matrix: the number of rows and the number of columns must be equal.
3. Create the game sequence.
    a. The program should loop through the players' turns in actions and exit if a win/lose condition is satisfied.

2.1 Example

The following example uses MatrEL to implement Tic-Tac-Toe.

```
# game Tic-Tac-Toe #
# initialize gameboard to a 3x3 matrix and set all cell entries to empty #
matrix gameboard = 3
gameboard[][] = “”

# create the three-in-a-row winning conditions #
boolean threeInARow {matrix m, cell pos, string value}
{
        if every m[pos:row][] value
        { return true }
        elseif every m[][pos:column] value
        { return true }
```

```
        elseif every m[\] value
        { return true }
        elseif every m[/] value
        { return true }
        else
        { return false}
}
# game loop #
boolean gameOver = false
string userInput = empty
string value = empty
cell userPos = 1,1
string player = empty
while –gameOver
{
        printout “Enter next player number:”
        getInput stdin player
        printout “Player” + player + “:  Enter selection as row,column:”
        getInput stdin userPos
        printout “Enter position value (X/O):”
        getInput stdin value
        gameboard[userPos] = value

        if threeInARow gameboard userPos value
                { printout “Player “ + player = “ wins!”
                gameOver = true }
        elseif -{any gameboard empty}
                { printout “Game tied!  No more moves left!”
                gameOver = true }
}
printMatrix gameboard
```

3 Language Manual

3.1 Introduction

MatrEL is a computer language designed for board game creation. Examples of games that can be created are Tic-Tac-Toe, Minesweeper, and Battleship. This language reference manual details the features of MatrEL and how one can program in this exciting language.

3.2 Lexical conventions

In MatrEL there are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. A sequence of one or more separators is required in between tokens. Blanks, tabs, newlines, and comments are used as separators and are otherwise ignored by the compiler.

3.2.1 Comments

The string of characters that begins with # and ends with # is treated as a comment.

3.2.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore, "_", symbol may be used as part of an identifier. Identifiers are case sensitive; uppercase and lowercase letters are considered different.

3.2.3 Keywords

The following identifiers my only be used as keywords:

int	||
matrix	==
cell	notEqual
string	Boolean
return	true
if	false
elseif	getInput
else	every
while	any
for	<=, >=, <, >
&&	

3.2.4 Constants

There are two types of constants in MatrEL:

3.2.4.1 Integer constants

An integer is a sequence of one or more numbers 0-9, but the first digit cannot be 0.

3.2.4.2 Strings

A string is a sequence of zero or more characters enclosed in double quotes ‘ “ ‘. The “” denotes the empty string.

3.3 What's in a Name?

MatrEL interprets an identifier based on its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier while the type determines the meaning of the values found in the identifier's storage.

The two declarable storage classes in MatrEL are automatic and external. Automatic identifiers are local to each instantiation of a function and are discarded upon function exit. External identifiers, on the other hand, exist independently of functions.

MatrEL supports two primary types of objects:

Characters: letters a-z and A-Z
Integers: sequence of numbers 0-9

In addition to the primary types MatrEL also has the following derived types:

cell: a row,column value that corresponds to an entry in a matrix
matrix: a two-dimensional array of cells
string: a sequence of characters
functions

3.4 Conversion

This section explains how operand conversion occurs in MatrEL.

3.4.1 Integers and Strings

An integer may be converted to a string representation of itself. Likewise, a string may be converted to an integer given that the string is a string representation of a sequence of integers.
The example,
string a = “5”
int b = a
results in b = 5.

3.5 Expressions

Expressions may be grouped into sub-expressions by surrounding the sub-expression in curly braces {expression}.

3.5.1 Primary expressions

Primary expressions involving function calls group left to right.

3.5.1.1 identifier

An identifier is a primary expression so long as it is properly declared

3.5.1.2 string

A string is a primary expression consisting of alphabetic characters.

3.5.1.3 {expression}

An expression inside curly braces is an expression whose type is the same as the expression without curly braces.

3.5.2 Unary operators

Expressions with unary operators group right to left.

3.5.2.1 -expression

The result is the negative of the expression and has the same type. The type of the expression must be an integer.

3.5.2.2 rowCount myMatrix

rowCount applied to a matrix returns the number of rows in that matrix.

3.5.2.3 columnCount myMatrix

columnCount applied to a matrix returns the number of columns in that matrix.

3.5.3 Multiplicative operators

The multiplication operator * group left-to-right.

3.5.3.1 expression * expression

The binary * operator indicates multiplication. Both expressions must be integers.

3.5.4 Additive operators

The additive operators + and – group left-to-right.

3.5.4.1 expression + expression

The result is the sum of the expressions. Both expressions must be integers.

3.5.4.2 expression – expression

The result is the difference of the expressions. Both expressions must be integers.

3.5.5 Relational operators

The relational operators group left-to-right:

expression < expression	less than
expression > expression	greater than

expression <= expression less than or equal to
expression >= expression greater than or equal to
Both expressions must be integers.

3.5.6 Equality operators

expression == expression equal to
expression != expression not equal to
The expressions being compared must be of the same type. Within a comparison, the expression types may be boolean, integer, or string. The return value will be a boolean.

3.5.7 expression and expression

The and operator groups left-to-right. Both expressions must be boolean.

3.5.8 expression or expression

The or operator groups left-to-right. Both expressions must be boolean.

3.6 Declarations

Declarations are used to give a type and value to an identifier. They have the form:
typeSpecifier identifier = value

3.6.1 Type-specifiers

The type-specifiers are:

- int
- boolean
- string
- matrix
- cell

3.6.2 int declarators

int declarations have the form:
int identifier = value where value is a sequence of numbers 0-9.

3.6.3 boolean declarators

boolean declarations have the form:
boolean identifier = value where value is either true or false.

3.6.4 string declarators

string declarations have the form:
string identifier = “value” where value is a sequence of alphabet characters, including _ underscore.

3.6.5 matrix declarators

matrix declarations have the form:
matrix identifier = value where value is an integer and specifies the size of the square matrix. The matrix will have value number of rows and value number of columns.

3.6.6 cell declarators

cell declarations have the form:
cell identifier = val1,val2 where val1 and val2 are integers.

3.7 Statements

Statements are executed in sequence.

3.7.1 Expression statement

Expressions have the form
expression
and are typically assignments or function calls.

3.7.2 Compound statement

Statements can be executed in order by combining them into a compound statement which puts curly braces around the list of statements:
compound statement:
    {statement-list}
statement-list:
    statement
    …
    statement

3.7.3 Conditional statement

The three forms of the conditional statement are:

```
if {expression} statement

if {expression} statement
else statement

if {expression} statement
if-else {expression} statement
…
if-else {expression} statement
else statement
```

For each statement the expression must evaluate to a boolean. The statement is executed if the expression evaluates to true. If neither of the if-expressions evaluate to true, the else statement will be executed.

3.7.4 While statement

The while statement has the form:

```
while {expression} statement
```

The expression evaluates to a boolean. The statement is repeatedly executed while the expression evaluates to true.

3.7.5 Return statement

A function returns to its caller by means of the return statement, which has one of the forms:

return	no value is returned
return {expression}	the value of the expression is returned

3.8 Scope rules

In MatrEL, we must consider lexical scope which is the area of the program in which an identifier is accessible.

3.8.1 Lexical scope

There are two types of lexical 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 and all functions may be used anywhere in the program.

3.9 Types revisited

This section summaries the operations that can be performed on objects of certain types.

3.9.1 Functions

Functions have the form:

```
functionReturnType functionName {parameter-list}
        {function-body}
```

The function return type can be integer, boolean, matrix, cell, string or empty if the function will not be returning an object. The functionName is a valid identifier. The parameter-list will be of the form {type iden1, …, type iden2}. The function-body is an expression that evaluates to and returns the same type as functionReturnType.

3.9.2 Matrices

A matrix can be set the following ways:

myMatrix[myRow][myColumn] = “a”	Sets the matrix cell at row = myRow and column = myColumn to the string a. The cell value must be a string.
myMatrix[][] = value	Sets each cell value to value. Value must be a string.
myMatrix[][myColumn] = value	Sets all of the cells in matrix column = myColumn to value.
myMatrix[myRow][] = value	Sets all of the cells in matrix row = myRow to value.
myMatrix[/] = value	Sets all of the cells in matrix diagonal (bottom left to top right diagonal) to value.
myMatrix[\] = value	Sets all of the cells in matrix diagonal (top left to bottom right

	diagonal) to value.
myMatrix[cellPos] = value	Sets the cell at location cellPos (integer, integer) in matrix to value.

In each of the above cases, the row and column values must be an integer.

A matrix cell value can be accessed the following ways:

myMatrix[rowNum][columnNum]	Returns the string value located at cell position rowNum,columnNum in myMatrix

Matrix values can be queried using the following keywords:

every myMatrix value	Returns true or false whether or not each cell value in myMatrix equals value
any myMatrix value	Returns true or false whether or not one or more cell's value in myMatrix equals value

3.9.3 Cells

Cells have the form:

cell myCell = rowNum,colNum

where rowNum and colNum correspond to a row and column position in a matrix.

Row and column values can be extracted from a cell by using:

myCell:row	returns the row number
myCell:column	returns the column number

3.10 Formatted Output

The following output functions are made available in MatrEL:

printOut someString	prints the string someString to the console
printMatrix myMatrix	"pretty prints" the matrix myMatrix to the console

4 Project Plan

4.1 Process

The process for completing this project is as follows: complete the project proposal, language reference manual, scanner, and parser. The abstract syntax tree and interpreter will be worked on in parallel to unit testing and completing the final report.

4.2 Programming Style

The following programming style will be adhered to during this project.

Naming conventions

- the names of variables and functions will be short and meaningful
- comments: will be inside (* comment *) and will clarify parts of the program that may be unclear

Indentation and spacing will be used to distinguish the different levels of scope to make the code more readable.

4.3 Project Timeline

This project will be driven by the following schedule:

9-24-2008	Language proposal, main language features defined
10-22-2008	Language reference manual, grammar complete
10-22-2008	Scanner and parser complete (version 1)
11-7-2008	Ast, interpreter, printer complete (version 1)
12/5/2008	Code freeze, project feature complete.

4.4 Team Responsibilities

Since this is a one person team, all work for this MatrEL project will be done by Rochelle Palting. The responsibilities that will be taken on are coding, testing and debugging all project code. Additional tasks are architecture, grammar, parser, compiler and testing.

4.5 Software Development Environment

This project will be developed on a Windows computer. The MatrEL language will be created using the Objective Caml programming language, version 3.10.2. Source code and project documents (proposal, language reference manual, and final report) will be controlled using CVS.

4.6 Project Log

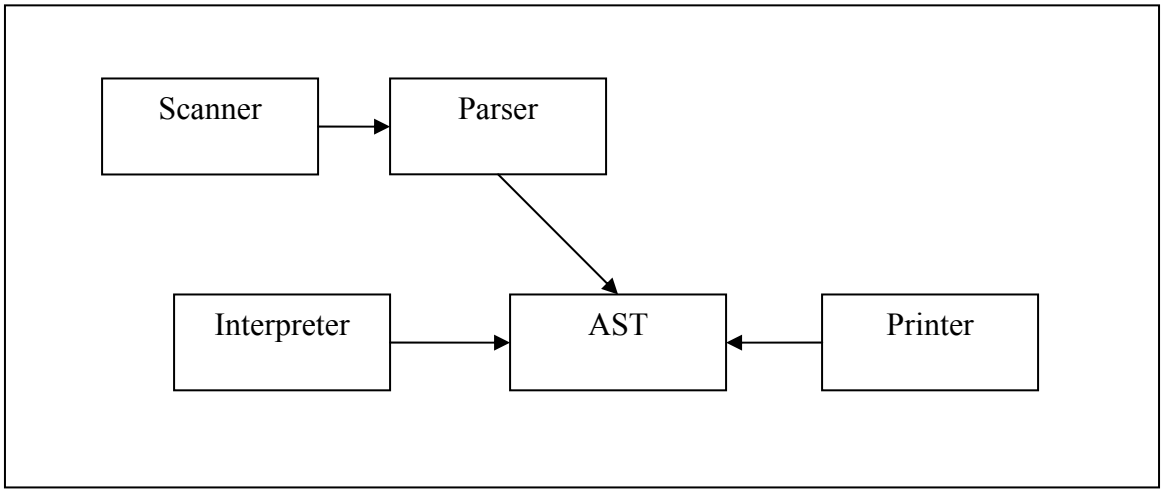
The following table lists concrete dates of major project milestones:

9-10-2008	Project began
9-24-2008	Language proposal complete
10-22-2008	Submitted Language reference manual, first draft
11-5-2008	Code scanner and parser
11-19-2008	Code ast and interpreter
12-12-2008	Final Report, final version

5 Architectural Design

5.1 Architecture

The main components of the MatrEL architecture are the scanner, parser, interpreter, abstract syntax tree (ast), and printer. The scanner lists and describes the tokens of the MatrEL language. The parser builds the abstract syntax tree from the parsing rules and provides the context-free grammar. The abstract syntax tree lists the main object groups of the language. The interpreter interprets the abstract syntax tree. The printer defines methods to easily print out the main object groups found in the abstract syntax tree. Below is a diagram illustrating the main components of the MatrEL compiler.



MatrEL Architecture Diagram

6 Test Plan

6.1 Goals

The goal of testing will be to verify that a basic and typical use case of the language by a developer will be doable. Specifically, the MatrEL types shall be constructed and used as expected. The function declarations and scoping rules should be accurate.

6.2 Hypothesis

Upon completion of each unit test, the feature being tested should provide a pass or fail result. If the result is a fail, more work may need to be done to correct the failure.

6.3 Methods

Testing will be completed in three phases.

Phase 1: Phase 1 testing will be conducted in parallel to developing the skeleton of all the main components (scanner, parser, ast, interpreter and printer). The interfaces to each object will be created and tested for functionality. Each item will be added individually to the scanner, parser, ast, interpreter and then printer. Compilation of each file should result in a success before moving on to modifying the next file. The interface should then be tested again.

Phase 2: In Phase 2 testing consists of testing the best case scenario of the environment in which the objects receive all valid data.

Phase 3: In Phase 3 testing, we input invalid values to verify that the environment handles these values correctly and as expected.

6.4 Tools

I plan to use Shell scripts for testing.

7 Lessons Learned

The primary recommendation from me would be to start the project as early as possible. There is a learning curve to learning a new programming language (OCaml) in addition to the learning curve to creating your own language.

8 Appendix

8.1 Code Listing

8.1.1 scanner.ml

```
{ open Parser }

let digit = ['0'-'9']
let lwrAlpha = ['a'-'z']
let uprAlpha = ['A'-'Z']
let idString = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9'
'_']*

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "#"     { comment lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '"'        { DQUOTE }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "|"      { OR }
| "every"  { EVERY }
| "any"      { ANY }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "string" { STRING }
| "boolean"{ BOOLEAN }
| "matrix" { MATRIX }
```

```
| "cell"   { CELL }
| digit+ as lxm { LITERAL(int_of_string lxm) }
| idString as lxm { ID(lxm) }
| '"' ['a'-'z' 'A'-'Z' '0'-'9' '_']* '"' as lxm { STR(lxm)
}
| "true"|"false" as lxm { BOOL(lxm) }
|     idString '[' digit* ']' '[' digit* ']'
     | idString "[upDiag]"
     | idString "[dnDiag]" as lxm { MAT(lxm) }
| digit "," digit as lxm { MATCELL(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

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

8.1.2 parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA DQUOTE
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ AND OR EVERY ANY
%token RETURN IF ELSE FOR WHILE INT STRING BOOLEAN MATRIX
CELL
%token <int> LITERAL
%token <string> ID
%token <string> STR
%token <string> BOOL
%token <string> MAT
%token <string> MATCELL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
%left EQ NEQ AND OR
%left LT GT LEQ GEQ EVERY ANY
%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 }
     | MATRIX ID SEMI { $2 }
      | CELL 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) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
```

```
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN
stmt
     { For($3, $5, $7, $9) }

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

expr:
    LITERAL          { Literal($1) }
  | ID               { Id($1) }
  | DQUOTE STR DQUOTE { Str($2) }
  | BOOL             { Bool($1) }
  | MAT              { Mat($1) }
  | MATCELL              { Cell($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 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) }
  | expr EVERY  expr { Binop($1, Every,   $3) }
  | expr ANY    expr { Binop($1, Any,   $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 }
```

8.1.3 ast.mli

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

```
type expr =
    Literal of int
  | Id of string
  | Str of string
  | Bool of string
  | Mat of string
  | Cell 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
```

8.1.4 interpreter.ml

```
open Ast

(* NameMap has type=string and value=int *)
module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of int * int NameMap.t

(* StringMap has type=string and value=String *)
(*
module StringMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)
```

```
exception ReturnException of string * string StringMap.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 -> 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=int,
updated environment) *)
    let rec eval env = function
     Literal(i) -> i, env
      | Noexpr -> 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))
       | Str(word) -> 1 (*word*), env (* how do we return a
string when int value is required? *) (* example string:
"myString" *)
      | Bool(b) -> (* how to return bool when int value is
required? *) (* example bool:  true *)
                if b="true" then 1, env
                else if b="false" then 0, env
                else raise (Failure ("invalid boolean " ^
b))
       | Mat(m) -> 1 (* m *), env (* how to return matrix
when int value is required? *)
       | Cell(pos) -> 1 (* pos *), env (* how to return
cell when int value is required? *) (* example cell:  1,2
*)
       | 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
        let orFun = function
          | x :: y :: [] ->
              if x = 1 then 1
              else if y = 1 then 1
              else 0
          | _ -> 0
          in
        (match op with
          Add -> v1 + v2
        | Sub -> v1 - v2
        | Mult -> v1 * v2
        | Div -> v1 / v2
        | Equal -> boolean (v1 = v2)
        | Neq -> boolean (v1 != v2)
        | Less -> boolean (v1 < v2)
        | Leq -> boolean (v1 <= v2)
        | Greater -> boolean (v1 > v2)
        | Geq -> boolean (v1 >= v2)
        | And -> boolean( boolean(v1 = 1) = v2)
     (* returns 1 if v1 = v2 = 1 ; else returns 0 *)
        | Or -> orFun(v1 :: v2 :: [])     (* returns 1 if v1
= 1 or v2 = 1 ; else returns 0 *)
        | Every -> 1 (* temporary return value *)  (*
example:  every "x" gameMatrix *)
           (* v1=string, v2=matrix *)
           (* if every element in matrix v2=v1, return true,
else return false *)
        | Any -> 1 (* temporary return value until I know
what to do here *) (* v1=string, v2=matrix *)
           (* v1=string, v2=matrix *)
           (* if any element in matrix v2=v1, return true,
else return false *)
        ), 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_endline (string_of_int v);
```

```
        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, values) actual ->
           let v, env = eval env actual in
           v :: actuals, values) ([], env) 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 v != 0 then s1 else s2)
      | While(e, s) ->
       let rec loop env =
         let v, env = eval env e in
         if 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 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
    call (NameMap.find "main" func_decls) [] globals
  with Not_found -> raise (Failure ("did not find the
main() function"))
```

8.1.5 printer.ml

```
open Ast

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(s) -> s
  | Str(s)-> s
  | Bool(s)-> s
  | Mat(s)-> s
  | Cell(s)-> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
     Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
       | And -> "&&" | Or -> "|"
       | Every -> "every" | Any -> "any"
```

```
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq ->
">=") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr
el) ^ ")"
  | Noexpr -> ""

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";
  | Return(expr) -> "return " ^ string_of_expr expr ^
";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^
")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr
e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s

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

let string_of_vdec2 id = "string " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^
")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals)
^
  String.concat "" (List.map string_of_vdec2 fdecl.locals)
^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "" (List.map string_of_vdec2 vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```