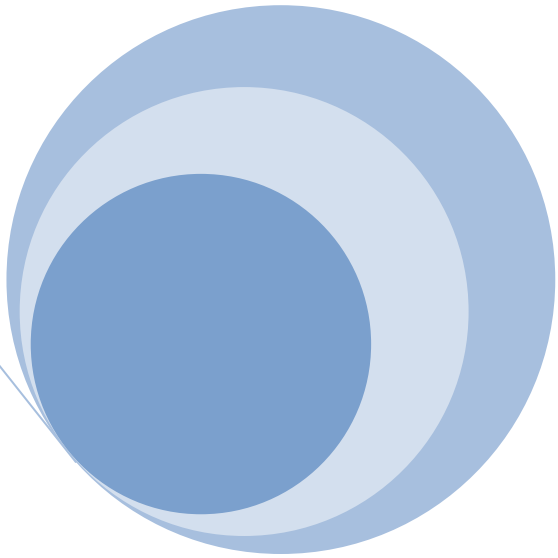
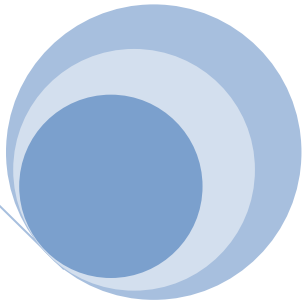
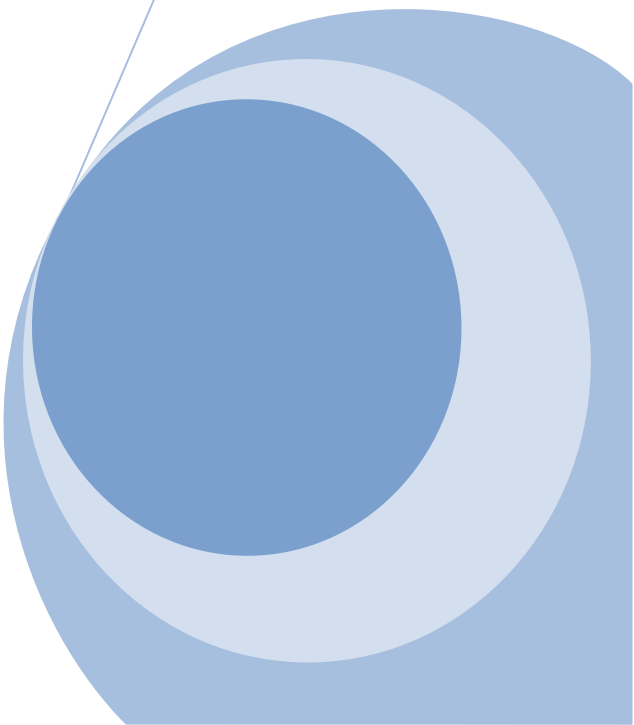






“AIC” All In the Cards Final Report

COMS 4115 Programming Languages and Translators

AIC is a language that allows the easy creation of card games ranging from the well known to brand new user created.

Andrew Willouer asw2126
8/10/2009

Contents

1 Introduction 4
1.1 Goals of AIC – Intuitive & Ease of Use 4
2 Language Tutorial 4
2.1 Introduction 4
2.2 Example 4
3 Language Reference Manual 6
3.1 Lexical Conventions 6
3.1.1 Comments 6
3.1.2 Identifiers (Names) 6
3.1.3 Keywords 6
3.1.4 Integer Constants 6
3.1.5 String Constants 6
3.1.6 Operators 6
3.1.7 Conditional Statements 7
3.1.8 Iteration Statements 7
3.2 Data Types 7
3.2.1 Arrays 7
3.3 Scope 8
3.4 Declarations 8
3.4.1 Variables 8
3.4.2 Functions 8
3.4.2.1 Predefined Functions 8
4 Project Plan 8
4.1 Team Roles and Responsibilities: 8
4.2 Project Timeline: 9
4.3 Planning and Specification: 9
4.4 Development and Testing: 9
4.5 Programming Style Guide: 9
4.6 Software Development Environment: 10
4.7 Project Log: 10
5 Architecture Design 11

5.1 Block Diagram:.....11
6 Test Plan.....12
6.1 Automated Testing:.....12
6.2 Non-Automated Testing:.....12
7 Lessons Learned.....12
8 References.....13
9 Appendix.....13
9.1 AOC.ml.....13
9.2 AST.ml.....13
9.3 Scanner.mll.....14
9.4 Parser.mly.....17
9.5 Printer.ml.....19
9.6 Interpret.ml.....21
9.7 MakeFile.....27
9.8 Test Script: Test.bat.....27
9.9 test-arith1.mc.....28
9.10 test-arith2.mc.....28
9.11 test-fib.mc.....28
9.12 test-for1.mc.....28
9.13 test-func1.mc.....29
9.14 test-gcd.mc.....29
9.15 test-hello.mc.....29
9.16 test-if1.mc.....29
9.17 test-if2.mc.....29
9.18 test-if3.mc.....30
9.19 test-if4.mc.....30
9.20 test-ops1.mc.....30
9.21 test-var1.mc.....30
9.23 test-while1.mc.....30
9.23 blackjack.txt.....31
9.23 highcard.txt.....33

1 Introduction

There are thousands of card games that use a standard 52 card deck. AIC allows the quick creation of card games by only specifying certain parameters. AIC language describes the rules, players, dealer, number of decks, and card values and the AIC compiler takes this input and creates a console based card game. This allows the user to concentrate on the rules of the game and not worrying about the coding details.

1.1 Goals of AIC – Intuitive & Ease of Use

The main goal of AIC is to allow the game designer to easily create various card games with only having basic programming skills. This will allow the game designer to concentrate on game creation instead of the programming syntax.

2 Language Tutorial

2.1 Introduction

The AIC language is based off of C++ and Java and should be familiar to anyone who has programmed in those languages. The AIC language is an interpreter language. Once the language is compiled it can be run on Windows as an exe from the command prompt and takes a text file as input. The text file is based off of the AIC language.

2.2 Example

An AIC program starts with the main function. The AIC language supports global and local variables and functions.

Below is partial code from an example of a High Card game. Cards are taken from the deck and whoever has the highest card wins. There are built in functions to the AIC language to speed up programming. The Deal function takes one card off of the deck and gives it to the player who is passed to the function. The deck is already initialized upon start up but it can be changed if needed. The Read function takes input from the command line and assigns it to a variable. Print, prints out a variable to the command line. AIC language also supports comparing player variables which actually compares the value of the player's hands. The card values can be changed to give different cards different values.

```
/*This is the starting point of the function*/
main()
{
   var NDecks;
   var Dealer;
   var numplayers;
   var CName;
   var dealer;
   var player1;
   var player2;
   var player3;
   var i;
   var pvalue;
   var dvalue;
```

```
    var anothercard;
    var bust;
    NDecks = 1;
    Dealer = true;

    CName = "HighCard";
    /*Print out what card game*/
    Print("You are playing");
    Print(CName);

    Print("How many players are playing?  Enter a number between 1 and 3");
    Read(numplayers);
    while(numplayers < 1 || numplayers > 3)
    {
       Print("Wrong number of players! How many players are playing?  Enter a
number between 1 and 5.");
       Read(numplayers);
    }

    /*Shuffle(Deck); Shuffle isn't implemented yet */

    Deal(dealer); /*This will take the top card from the deck and add it to
the dealer's hand.*/
    Print("The dealer's card is");
    Print(dealer);

    Deal(player1);
    Print("Player one's card is:");
    Print(player1);

    if(dealer > player1)
    {
       Print("dealer wins");
    }
    else
    {
     Print("player one wins");
   }

}
```

The output from running this little code would be:

You are playing

HighCard

How many players are playing? Enter a number between 1 and 3

(Program waiting for input) User enters 1

The dealer's card is

(Prints the dealer's card, example "4H")

Player one's card is:

(Prints the player's card, example "6D")

Player one wins.

3 Language Reference Manual

3.1 Lexical Conventions

There are six kinds of tokens in the AIC language: identifiers, keywords, constants, strings, expression operators, and other separators. Blanks, tabs, newlines, spaces and comments (whitespace) are ignored except to separate tokens. Some whitespace is required to separate adjacent tokens.

3.1.1 Comments

Comments start with /* and end with */. Everything between those symbols is considered part of the comment. Comments do not nest.

3.1.2 Identifiers (Names)

An identifier is a sequence of letters (a-z), digits (0-9), and underscores and the first character must be a letter. Upper and lower case letters are distinct and there is no limit on the length of an identifier. Keywords are reserved and cannot be redefined.

3.1.3 Keywords

The following identifiers are reserved for keywords:

If else while for return int bool true false main Print Cards Deal

Shuffle Read value var

Also all the card names are reserved: 2-10, J, Q, K followed by S, H, C, D. Example 2S for 2 of Spades or KH for King of Hearts.

3.1.4 Integer Constants

An integer constant is a sequence of digits, 0-9 and always considered to be decimal.

3.1.5 String Constants

A string is a sequence of one or more characters which may include letters, digits, symbols and whitespace surrounded by double quotes (eq. "This is a String!").

3.1.6 Operators

An operator performs an operation on tokens. The operators are listed below in order of precedence highest to lowest:

&&	Logical and	Left to Right Precedence
||	Logical or	Left to Right Precedence

=	Assignment	Right to Left Precedence
==	Equality test	Left to Right Precedence
!=	Inequality test	Left to Right Precedence
<	Less than	Left to Right Precedence
<=	Less than or equal	Left to Right Precedence
>	Greater than	Left to Right Precedence
>=	Greater than or equal	Left to Right Precedence
+	Add	Left to Right Precedence
-	Subtract	Left to Right Precedence
/	Division	Left to Right Precedence
*	Multiplication	Left to Right Precedence

3.1.7 Conditional Statements

A conditional statement consists of either an if(expression) { true block } else {false block} or just if(expression) { true block} without an else block. Conditional statements can be nested.

3.1.8 Iteration Statements

An iteration statement repeats executing statements as long as the condition is true for a while loop and a certain number of times for a for loop.

The while iteration statement consists of while(expression) {statements} The expression of a while loop must resolve to either true or false and the statements are run as long as the expression is met.

The for iteration statement consists of for(start expression; test expression; count expression) {statements}

3.2 Data Types

There are five distinct data types:

Integers: A list of digits (0-9)

Strings: A list of characters surrounded by double quotes.

Booleans: True or False

Hands: A list that contains all the cards that belong to someone.

Decks: A list that contains all the cards not in play or someone's hand. Cards can be removed from the deck or put back into the deck depending on the rules of the game.

3.2.1 Arrays

Arrays are declared with square brackets [ ]. The language supports arrays of integers and strings.

3.3 Scope

There are two kinds of scope: global scope and local scope. Identifiers declared in a local function can only be accessed in that function. Global identifiers declared outside functions may be used throughout the entire program. Global identifiers are proceeded with the @ symbol.

3.4 Declarations

Identifiers must be identified as variables or functions. Declarations have the following form:

Variables: var identifier

Functions: identifier (parameters list) {block of function}

3.4.1 Variables

Variables are uninitialized when created. They must be assigned a value to be used. Variables values can change after being initialized.

3.4.2 Functions

A function must return a valid type. A function does not need a parameter list. The parameter list can contain multiple identifiers separated by a comma. A function can call another function within its function block, but a function cannot declare a new function within its function block. When calling a function the parameter list must be the same size as the parameter list declared when the function was declared.

3.4.2.1 Predefined Functions

There are several built in functions included in the language. They include:

Print ( ) Prints everything contained in between the parenthesis. Text must be surrounded by double quotes. Variables do not need to be in double quotes and their values are printed.

Shuffle(Decks) Takes the Decks type and randomly distributes the cards throughout the deck

Deal (Player) Takes one card from the deck and gives that card to the player specified by Player.

Read (string) Takes one string variable and waits for text from the console and places the text into the string variable

main() Has no parameters but is the specialized function where code begins to be executed from.

4 Project Plan

4.1 Team Roles and Responsibilities:

There was only one team member on the AIC project and thus all planning, coding, testing and debugging was done individually.

4.2 Project Timeline:

May 27-June 2	Worked on initial proposal
June 3	Submitted initial proposal
June 10-17	Worked on new proposal
June 17	Submitted revised proposal
June 10-21	Worked on Language Reference Manual (LRM)
June 22	Submitted LRM
June 23-July 14	Worked on Parser, Scanner and AST
July 14-28	Worked on Interpreter and revised Parser, Scanner, AST
July 29-August 8	Worked on testing and fixing bugs
August 10	Submitted Final Report and Project

4.3 Planning and Specification:

The AIC language started with an initial idea for a language to implement games. From there it was decided that AIC would concentrate on card games. There are so many card games out there so Black Jack was chosen as a starting point. A quick sample Black Jack program written in the AIC language was sketched out. This sample program helped form the basis for the Language Reference Manual. Projects from previous semesters were used as reference to make sure the AIC language wasn't missing anything fundamental.

4.4 Development and Testing:

A version control system was set up on Google code. The first step was uploading the MicroC language which was used as a baseline for the AIC language. The next step was to take the LRM and modify the Parser, Scanner and AST to suite the AIC language's needs. During this phase some modifications were made to the LRM to add missing things and to take out features that would be too complicated to implement. The MicroC Interpreter was heavily modified to add the new AIC types. AIC language is basically the MicroC language expanded. Thus, the MicroC test suite was also used as a baseline for the AIC test suite. The MicroC test suite had to be modified to work with AIC since the AIC language has new types. As the new types were added to the AIC language the MicroC tests were modified. For new features specific to the AIC language, new tests were written. The final week of the project involved much testing, debugging and code fixes to get the very basic features of the AIC language implemented.

4.5 Programming Style Guide:

A basic programming style guide was used. The Scanner, AST and Parser have their own specific programming style and it was followed as close as possible when new code was added. Comments and spacing were also added as appropriate to make the Scanner, AST and Parser code as easy to read and follow as possible.

The bulk of the code was in the Interpreter and to make the O'Caml code as easy to read and follow as Java in the Interpreter, a specific set of guidelines was followed.

Comments were used as appropriate throughout the Interpreter.

Line length was also enforced to keep lines to a reasonable length.

Indentation, spacing, and alignment were used throughout the Interpreter to separate and align blocks of code. This was the most important part of keeping the code in the Interpreter readable. For example, "Let in" O'Caml calls were separated with newlines and code that followed a "Let in" was indented. This helped separate the O'Caml code into "function" calls. Also, code that followed match was indented such that the | symbol would started every new line. This made it easier to see what blocks of code were part of what match calls. Utilizing this style made development and debugging more efficient.

4.6 Software Development Environment:

All software was developed on an AMD Athlon 2100+ running Microsoft Windows Vista 32. Eclipse version 3.4.1 was used as the editor. An O'Caml add-on was downloaded from http://ocamldt.free.fr/ that integrated nicely with Eclipse. The add-on came bundled with Ocamlyacc, Ocamllex. Developing the AIC project in Eclipse with the O'Caml add-on made compiling the AIC project easier. The add-on also colored words correctly and showed syntax errors in red which made development go smoother and faster. Google Code and Subversion were used as the software version control system. Tortoise SVN was used to manage the AIC project files and upload changes to the Subversion server. All files were developed in OCAML.

4.7 Project Log:

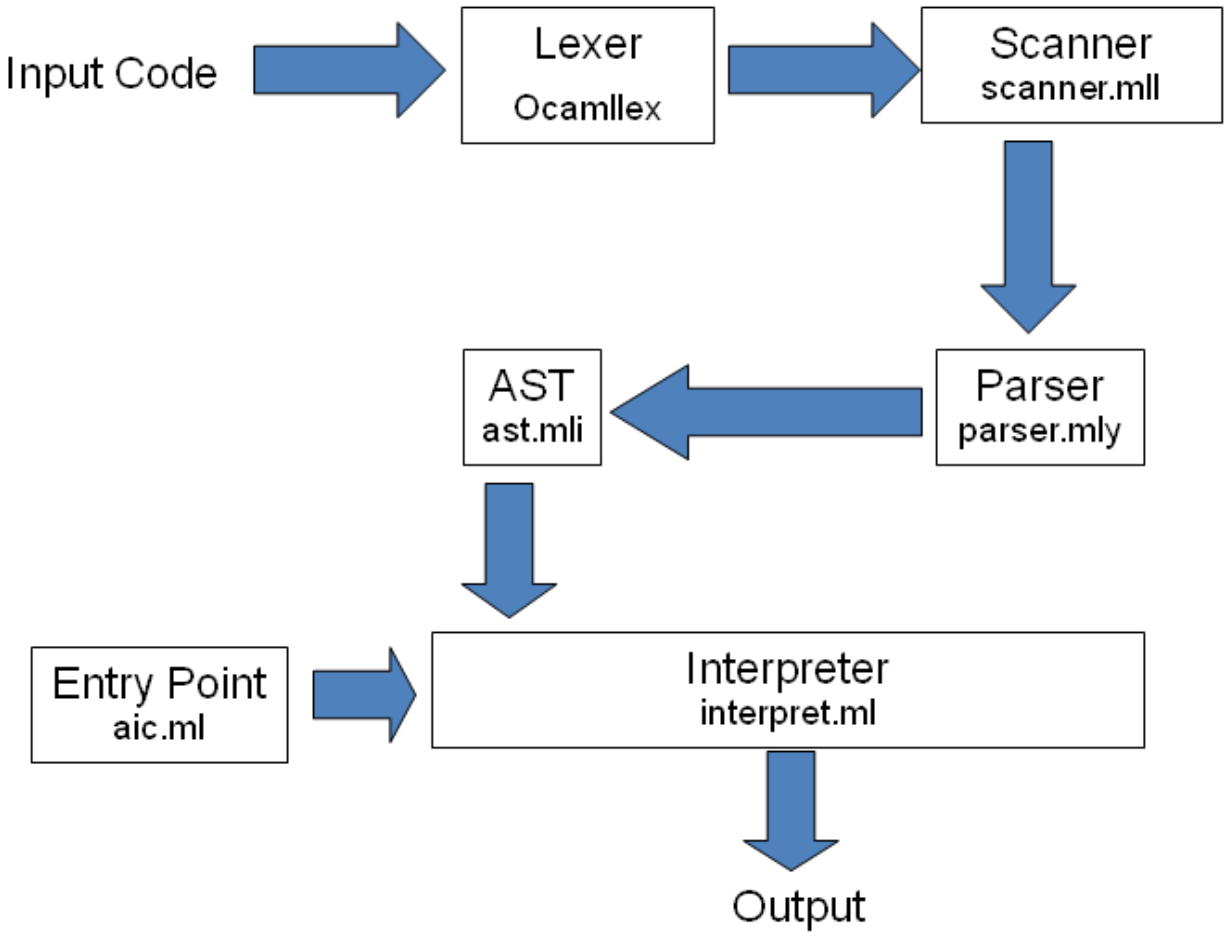
The project log is generated by Subversion with the highlights pulled out and located in the table below.

Date	Comment	Files Impacted
07/16/09	Initial directory structure	None
07/16/09	Added MicroC base files	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
07/16/09	Updated parser and scanner	ast.mli, parser.mly, scanner.mll
07/17/09	Continue to expand grammar	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
07/19/09	Continue updating	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
07/22/09	Update interpret & printer	interpret.ml, printer.ml
07/26/09	Work on getting interpret.ml to run	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
08/05/09	Revisions to get test suite to run	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
08/05/09	Test Suite Added	Entire test suite
08/08/09	Final version of code	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml
08/08/09	Highcard and BlackJack added to test suite	Highcard.txt and Blackjack.txt
08/09/09	Final version of code with comments	aoc.ml, ast.mli, interpret.ml, parser.mly, printer.ml, scanner.ml

5 Architecture Design

The AIC language is an interpreter language. The language has a Scanner, Parser, Abstract Syntax Tree (AST), and finally an Interpreter. The source file is read by the OCamllex tool which then inputs into the Scanner. The Scanner generates tokens which then inputs into the Parser which then generates the Abstract Syntax Tree. The Interpreter contains the bulk of the AIC code and takes the abstract syntax tree as input. The Interpreter walks through the created abstract syntax tree and creates a symbol table and adds variables to the symbol table. The Interpreter also does type checking and executes all operations and statements contained in the AIC language.

5.1 Block Diagram:



6 Test Plan

The MicroC test suite was used as a baseline for the AIC language. The AIC builds upon the MicroC language so I wanted to make sure all of the MicroC features were still available. As the language was modified the MicroC test suite was ran to make sure existing features were not broken. The MicroC test suite had to be modified to work with the new AIC types and the MicroC test suite had to be expanded to test out new features.

The entire test suite is attached in the appendix.

6.1 Automated Testing:

The basic testing was all done automated. A script was written, test.bat that can be run from the command line in Windows. The script grabs all the *.mc test files runs them through the interpreter which generates .ans files. The script then compares the .ans files to the expected output which is contained in the .out files. If there is no difference the script passes the test.

6.2 Non-Automated Testing:

The more advanced features of the AIC language need user input so two other test files were used, BlackJack.txt and HighCard.txt. The goal was to get both up and running, but there was only time to get the HighCard.txt test fully operational.

7 Lessons Learned

I've taken a compilers course before but we spent the entire semester working on the lexer and the scanner. I learned a lot more actually implementing an entire language. The most important lesson I learned was how every little detail and decision can make a huge difference by the time you get to actually implementing the heart of the language in the interpreter. I thought I had my scanner and parser finished quite a few times, but I had to constantly go back and modify them to add something I forgot, delete something that was going to be too hard or modify something that wasn't working out quite how I wanted it too.

The biggest piece of advice I have for future teams is start early. By the time I had a good grasp of O'Caml and the entire project the semester was over. I didn't get to add all the features to the AIC language to make a really useful language. I would spent more time upfront with the MicroC language modifying it, running it, playing with the test suite to get a better understanding of how all the pieces fit together. I would also spend more time coming up with the final goals, sample programs and work backwards from there. I kept modify the entire project from the LRM to all the parts of the code as I got a better understanding and I never got to lock any parts down. This made progress difficult as any little change to one part effects all the other parts. So again, start early.

8 References

"MicroC Language," written by Professor Stephen A. Edwards presented in class

"Cards the Language," written by Jeffrey C Wong, Fall 2008

"PCGSL: Playing Card Game Simulation Language," written by Enrique Henestroza, Yuriy Kagan, Andrew Shu, Peter Tsonev, Fall 2008

9 Appendix

9.1 AOC.ml

```
(*Andrew Willouer
COMS 4115 Programming Languages and Translators
8/10/2009*)

(*AOC.ml
This is the All in the cards main program that takes an input text file as an
argument*)

(* read from a file *)
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
```

9.2 AST.ml

```
(*Andrew Willouer
COMS 4115 Programming Languages and Translators
8/10/2009*)

(*Ast.mli
Consists of all the types for the AIC language*)

(*Binary Operators*)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
And | Or

(*Scope Global or Local*)
type scope = Global | Local

(*Types, int, string, bool implemented, the rest for future development*)
```

```
type types =
    Int
  | StringType
  | Bool
  | Card
  | Hand
  | Deck
  | Player


type varexp = VarExp of string * scope
(*Expression Type*)
and expr =
    Intliteral of int
  | Variable of varexp
  | Stringliteral of string
  | Id of string
  | Binop of expr * op * expr
  | Call of string * expr list
  | Cardliteral of string
  | Handliteral of string * int  (*Handliteral contains a cardliteral "2C"
and the cards value*)
  | Boolliteral of bool
  | Assign of varexp * expr
  | Noexpr

(*Statements*)
type stmt =
    Expr of expr
  | Return of expr
  | Print of expr
  | Read of varexp
  | If of expr * stmt list * stmt list
  | For of expr * expr * expr * stmt
  | While of expr * stmt list
  | Deal of varexp
  | Value of expr

(*function declaration *)
type func_decl = {
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
                }

type program = string list * func_decl list
```

9.3 Scanner.mll

```
(*Andrew Willouer
COMS 4115 Programming Languages and Translators
8/10/2009*)

(*Scanner.mll
Scanner generates tokens for the parser*)
```

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }                    (*Punctuation*)
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| '.'      { PERIOD}
| ';'      { SEMI }
| ','      { COMMA }
| '?'      { MATCH }
| '+'      { PLUS }                        (*Operators*)
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| "&&"     { AND }
| "||"     { OR }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }                        (*Keywords*)
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "true"   { TRUE(true) }
| "false"  { FALSE(false) }
| "Print"  { PRINT }
| "Hands"  { HANDS }
| "Cards"  { CARDS }
| "Deal"   { DEAL }
| "Decks"  { DECKS }
| "Value"  { VALUE }
| "Read"   { READ }
| "var"    { VAR }
| "Player" { PLAYERS }
| "@"      { GLOBALVAR }
| "Globals" { GLOBALS }
| "2S"     { CARDLITERAL("2S") }      (*Card Types*)
| "2C"     { CARDLITERAL("2C") }
| "2D"     { CARDLITERAL("2D") }
| "2H"     { CARDLITERAL("2H") }
| "3S"     { CARDLITERAL("3S") }
| "3C"     { CARDLITERAL("3C") }
| "3D"     { CARDLITERAL("3D") }
| "3H"     { CARDLITERAL("3H") }
| "4S"     { CARDLITERAL("4S") }
```

```
  | "4C"      { CARDLITERAL("4C") }
  | "4D"      { CARDLITERAL("4D") }
  | "4H"      { CARDLITERAL("4H") }
  | "5S"      { CARDLITERAL("5S") }
  | "5C"      { CARDLITERAL("5C") }
  | "5D"      { CARDLITERAL("5D") }
  | "5H"      { CARDLITERAL("5H") }
  | "6S"      { CARDLITERAL("6S") }
  | "6C"      { CARDLITERAL("6C") }
  | "6D"      { CARDLITERAL("6D") }
  | "6H"      { CARDLITERAL("6H") }
  | "7S"      { CARDLITERAL("7S") }
  | "7C"      { CARDLITERAL("7C") }
  | "7D"      { CARDLITERAL("7D") }
  | "7H"      { CARDLITERAL("7H") }
  | "8S"      { CARDLITERAL("8S") }
  | "8C"      { CARDLITERAL("8C") }
  | "8D"      { CARDLITERAL("8D") }
  | "8H"      { CARDLITERAL("8H") }
  | "9S"      { CARDLITERAL("9S") }
  | "9C"      { CARDLITERAL("9C") }
  | "9D"      { CARDLITERAL("9D") }
  | "9H"      { CARDLITERAL("9H") }
  | "10S"      { CARDLITERAL("10S") }
  | "10C"      { CARDLITERAL("10C") }
  | "10D"      { CARDLITERAL("10D") }
  | "10H"      { CARDLITERAL("10H") }
  | "JS"      { CARDLITERAL("JS") }
  | "JC"      { CARDLITERAL("JC") }
  | "JD"      { CARDLITERAL("JD") }
  | "JH"      { CARDLITERAL("JH") }
  | "QS"      { CARDLITERAL("QS") }
  | "QC"      { CARDLITERAL("QC") }
  | "QD"      { CARDLITERAL("QD") }
  | "QH"      { CARDLITERAL("QH") }
  | "KS"      { CARDLITERAL("KS") }
  | "KC"      { CARDLITERAL("KC") }
  | "KD"      { CARDLITERAL("KD") }
  | "KH"      { CARDLITERAL("KH") }
  | "AS"      { CARDLITERAL("AS") }
  | "AC"      { CARDLITERAL("AC") }
  | "AD"      { CARDLITERAL("AD") }
  | "AH"      { CARDLITERAL("AH") }
  | (['2'-'9'] | "10" | "J" | "Q" | "K" | "A")['S' 'C' 'D' 'H'] as lxm
{ CARDLITERAL(lxm) }
  | ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | '\"'[^ '\"']*'\"' as lxm { STRINGLITERAL(String.sub lxm 1 ((String.length
lxm) -2)) }
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

9.4 Parser.mly

```
/*Andrew Willouer
COMS 4115 Programming Languages and Translators
8/10/2009*/

(*Parser.mly
Parser takes tokens from scanner and parses them into expressions, statements,
variables and so forth for the interpreter*)

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LBRACKET RBRACKET PERIOD MATCH
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AND OR
%token RETURN IF ELSE FOR WHILE INT
%token HANDS CARDS VALUE SETVALUE DEAL DECKS SHUFFLE
%token PLAYERS
%token DEALER CARDNAME NUMBERDECKS
%token VAR GLOBALVAR
%token GLOBALS RULES
%token PRINT READ
%token BOOL INT STRING
%token <int> INTLITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token <bool> TRUE FALSE
%token <string> CARDLITERAL

%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
%left EQ NEQ
%left AND OR
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right READ PRINT PRINTHAND
%left GLOBALVAR VAR PLAYER DECKS
%left BOOL INT STRING CARDS HANDS
%right DEALER CARDNAME NUMBERDECKS
%left VALUE SETVALUE

%start program
%type <Ast.program> program

%%
(*Program contains variable declartions, function declarations*)
program:
  /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }
```

```
(*Function list is a list of function declarations*)
funcs_list:
  /* nothing */ { [] }
 | funcs_list fdecl { $2 :: $1 }

(*Function declaration*)
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:
    vdecl                    { [$1] }
  | formal_list COMMA vdecl { $3 :: $1 }

(*Variable declaration list is a list of variables*)
vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl SEMI { $2 :: $1 }

(*Variable declaratoin consists of keyword VAR followed by ID
    or DECKS followed by ID or PLAYERS followed by ID*)
vdecl:
   VAR ID { $2 }
   | VAR GLOBALVAR ID { $3 }
   | DECKS ID { $2 }
   | PLAYERS ID { $2 }

(*List of statements*)
stmt_list:
   /* nothing */  { [] }
   | stmt_list stmt { $2 :: $1 }

(*Statements consist of Print, Read, Return, If, For, While, Deal, Value*)
stmt:
    expr SEMI { Expr($1) }
  | PRINT LPAREN expr RPAREN SEMI { Print($3) }
  | READ LPAREN var RPAREN SEMI { Read($3) }
  | RETURN expr_opt SEMI { Return($2) }
  | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE %prec NOELSE { If($3, $6,
[]) }
  | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list
RBRACE    { If($3, $6, $10) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN LBRACE stmt RBRACE
    { For($3, $5, $7, $10) }
  | WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE { While($3, $6) }
  | DEAL LPAREN var RPAREN SEMI {Deal ($3) }
  | VALUE LPAREN expr RPAREN SEMI {Value ($3) }

expr_opt:
```

```
    /* nothing */ { Noexpr }
  | expr          { $1 }


(*Expressions consist of the operators*)
expr:
  | TRUE             { Boolliteral(true)}
  | FALSE            { Boolliteral(false)}
  | CARDLITERAL      { Cardliteral($1) }
  | INTLITERAL       { Intliteral($1) }
  | STRINGLITERAL    { Stringliteral($1) }
  | var              { Variable($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) }
  | var ASSIGN expr  { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

var:
     ID  { VarExp($1, Local) }
   | GLOBALVAR ID { VarExp($2, Global) }

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

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

9.5 Printer.ml

```
(*Andrew Willouer
COMS 4115 Programming Languages and Translators
8/10/2009*)

(*Printer.ml
Used to print out the AST*)

open Ast

let rec tabs i = match i with
  0 -> ""
  | x -> "\t" ^ tabs (x - 1)

let string_of_op op = match op with
```

```
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let rec string_of_types types = match types with
    Int -> "int"
  | StringType -> "string"
  | Bool -> "bool"
  | Card -> "Card"
  | Deck -> "Deck"
  | Hand -> "Hand"
  | Player -> "Player"

let string_of_scope scope = match scope with
  Global ->"@"
  | Local -> ""

let rec string_of_varexp v = match v with
  | VarExp(id, s) -> string_of_scope s ^ id
and string_of_expr expr = match expr with (*function*)
    Intliteral(l) -> string_of_int l
  | Variable(v) -> string_of_varexp v
  | Boolliteral(b) -> string_of_bool b
  | Cardliteral(c) -> c
  | Stringliteral(s) -> "\"" ^ s ^ "\""
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
                      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq ->
">=" | And -> "&&" | Or -> "||") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> string_of_varexp v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let string_of_vardec v = match v with
  id -> "var" ^ id

let rec string_of_stmt t stmt = tabs t ^ match stmt with (*function*)
  (*Block(stmts) ->
  "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n")*)
   Expr(expr) -> string_of_expr expr ^ ";\n";
  | Print(expr) -> "Print " ^ string_of_expr expr ^ ";\n";
  | Read(var) -> "Read " ^ string_of_varexp var ^ ";\n";
```

```
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      String.concat "" (List.map (string_of_stmt (t+1)) s1) ^
        tabs t ^ "} else {\n" ^
      String.concat "" (List.map (string_of_stmt (t+1)) s2) ^
        tabs t ^ "} else {\n"
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") {\n" ^
    String.concat "" (List.map (string_of_stmt (t+1)) s) ^
        tabs t ^ "} \n"

let string_of_vdecl id = "int " ^ 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_stmt 1) fdecl.body) ^
  "}\n"

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


let string_of_sdecl_g sname vars =
  sname ^ "\n{\n" ^
  String.concat "" (List.map string_of_vdecl vars) ^ "}\n"

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

(*let string_of_program (spec, funcs) =
  string_of_sdecl_g "Globals" spec.glob.globals ^ "\n" ^
String.concat "\n" (List.map string_of_fdecl funcs)*)
```

9.6 Interpret.ml

```
(*Andrew Willouer
COMS 4115 Programming Languages and Translators
           8/10/2009*)

(*Interpret.ml
Takes the abstract syntax tree and runs the program*)

open Ast

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

exception ReturnException of Ast.expr * Ast.expr NameMap.t

(*Global variable, that contains all the cards*)
let mydeckstringref = ref ["2S", 2; "3S", 3; "4S", 4; "5S", 5; "6S", 6; "7S",
7; "8S", 8; "9S", 9; "10S", 10; "JS", 10; "QS", 10; "KS", 10; "AS", 10;
```

```
                    "2C", 2; "3C", 3; "4C", 4; "5C", 5; "6C", 6; "7C", 7; "8C",
8; "9C", 9; "10C", 10; "JC", 10; "QC", 10; "KC", 10; "AC", 10;
                    "2D", 2; "3D", 3; "4D", 4; "5D", 5; "6D", 6; "7D", 7; "8D",
8; "9D", 9; "10D", 10; "JD", 10; "QD", 10; "KD", 10; "AD", 10;
                              "2H", 2; "3H", 3; "4H", 4; "5H", 5; "6H", 6; "7H",
7; "8H", 8; "9H", 9; "10H", 10; "JH", 10; "QH", 10; "KH", 10; "AH", 10;];;

(*For future development*)
let deck = []
let players = []
let cards = []

(* 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
        Noexpr -> Noexpr, env (* must be non-zero for the for loop predicate
*)
      | Intliteral(i) -> Intliteral(i), env
      | Stringliteral(s) -> Stringliteral(s), env
      | Boolliteral(i) -> Boolliteral(i), env
      | Cardliteral(i) -> Cardliteral(i), env
      | Handliteral(h, i) -> Handliteral(h, i), env
      | 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 Boolliteral(true) else
Boolliteral(false) in
             (match v1, op, v2 with
                                 (*Operators*)
               Intliteral(i1), Add, Intliteral(i2) -> Intliteral(i1 + i2)
             | Stringliteral(s1), Add, Intliteral(i2) ->
Intliteral(int_of_string s1 + i2)
             | Intliteral(i1), Add, Stringliteral(s2) -> Intliteral(i1 +
int_of_string s2)
             | Stringliteral(s1), Add, Stringliteral(s2) ->
Intliteral(int_of_string s1 + int_of_string s2)
             | Intliteral(i1), Sub, Intliteral(i2) -> Intliteral(i1 - i2)
             | Stringliteral(s1), Sub, Intliteral(i2) ->
Intliteral(int_of_string s1 - i2)
```

```
                | Intliteral(i1), Sub, Stringliteral(s2) -> Intliteral(i1 -
int_of_string s2)
                | Stringliteral(s1), Sub, Stringliteral(s2) ->
Intliteral(int_of_string s1 - int_of_string s2)
                | Intliteral(i1), Mult, Intliteral(i2) -> Intliteral(i1 * i2)
                | Stringliteral(s1), Mult, Intliteral(i2) ->
Intliteral(int_of_string s1 * i2)
                | Intliteral(i1), Mult, Stringliteral(s2) -> Intliteral(i1 *
int_of_string s2)
                | Stringliteral(s1), Mult, Stringliteral(s2) ->
Intliteral(int_of_string s1 * int_of_string s2)
                | Intliteral(i1), Div, Intliteral(i2) -> Intliteral(i1 / i2)
                | Stringliteral(s1), Div, Intliteral(i2) ->
Intliteral(int_of_string s1 / i2)
                | Intliteral(i1), Div, Stringliteral(s2) -> Intliteral(i1 /
int_of_string s2)
                | Stringliteral(s1), Div, Stringliteral(s2) ->
Intliteral(int_of_string s1 / int_of_string s2)
                | Handliteral(a1, b1), Equal, Intliteral(i2) -> boolean (b1 =
i2)
                | Stringliteral(s1), Equal, Intliteral(i2) -> boolean
(int_of_string s1  = i2)
                | Intliteral(i1), Equal, Intliteral(i2) -> boolean (i1 = i2)
                | Stringliteral(s1), Equal, Stringliteral(s2) -> boolean (s1  =
s2)
                | Cardliteral(c1), Equal, Cardliteral(c2) -> boolean (c1 = c2)
                | Boolliteral(b1), Equal, Boolliteral(b2) -> boolean
(string_of_bool b1 = string_of_bool b2)
                | Intliteral(i1), Neq, Intliteral(i2) -> boolean(i1 != i2)
                | Stringliteral(s1), Neq, Stringliteral(s2) -> boolean(s1 !=
s2)
                | Stringliteral(s1), Neq, Intliteral(i2) -> boolean
(int_of_string s1  != i2)
                | Cardliteral(c1), Neq, Cardliteral(c2) -> boolean(c1 != c2)
                | Boolliteral(b1), Neq, Boolliteral(b2) ->
boolean(string_of_bool b1 != string_of_bool b2)
                | Intliteral(i1), Less, Intliteral(i2) -> boolean(i1 < i2)
                | Stringliteral(s1), Less, Stringliteral(s2) -> boolean(s1 <
s2)
                | Cardliteral(c1), Less, Cardliteral(c2) -> boolean(c1 < c2)
                | Stringliteral(s1), Less, Intliteral(i2) ->
boolean(int_of_string s1 < i2)
                | Intliteral(i1), Leq, Intliteral(i2) -> boolean(i1 <= i2)
                | Stringliteral(s1), Leq, Stringliteral(s2) -> boolean(s1 <=
s2)
                | Cardliteral(c1), Leq, Cardliteral(c2) -> boolean(c1 <= c2)
                | Stringliteral(s1), Leq, Intliteral(i2) ->
boolean(int_of_string s1 <= i2)
                | Handliteral(a1, b1), Greater, Handliteral(a2, b2) ->
boolean(b1 > b2)
                | Intliteral(i1), Greater, Intliteral(i2) -> boolean(i1 > i2)
                | Stringliteral(s1), Greater, Stringliteral(s2) -> boolean(s1 >
s2)
                | Cardliteral(c1), Greater, Cardliteral(c2) -> boolean(c1 > c2)
                | Stringliteral(s1), Greater, Intliteral(i2) ->
boolean(int_of_string s1 > i2)
                | Intliteral(i1), Geq, Intliteral(i2) -> boolean(i1 >= i2)
```

```
              | Stringliteral(s1), Geq, Stringliteral(s2) -> boolean(s1 >=
s2)
              | Cardliteral(c1), Geq, Cardliteral(c2) -> boolean(c1 >= c2)
              | Stringliteral(s1), Geq, Intliteral(i2) ->
boolean(int_of_string s1 >= i2)
              | Boolliteral(b1), And, Boolliteral(b2) -> boolean(b1 && b2)
              | Boolliteral(b1), Or, Boolliteral(b2) -> boolean(b1 || b2)
              | _, _, _ -> raise (Failure ("Invalid binary operation"))
            ) , env
             | Variable(var) ->
               let locals, globals = env in
               (match var with
                 VarExp(id, scope) ->
                   (match scope with
                    Local ->
                     if NameMap.mem id locals then
                     NameMap.find id locals, env
                     else raise (Failure ("undeclared local identifier " ^
id))
                   | Global ->
                     if NameMap.mem id globals then
                     NameMap.find id globals, env
                     else raise (Failure ("undeclared global identifier " ^
id))
                   )
               )
(*Assign finds the variable in the map and then binds(assigns) the variable*)
      | Assign(var, e) ->
        let v, (locals, globals) = eval env e in
        (match var with
          VarExp(id, scope) ->
            (match scope with
             Local ->
               if NameMap.mem id locals then
                 v, (NameMap.add id v locals, globals)
               else raise (Failure ("undeclared local identifier " ^ id))
             | Global ->
               if NameMap.mem id globals then
                 v, (locals, NameMap.add id v globals)
               else raise (Failure ("undeclared global identifier " ^ id))
            )
        )
     | 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
         List.append actuals [v], values) ([], env) actuals
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals
     in Boolliteral(false), (locals, globals)
     with ReturnException(v, globals) -> v, (locals, globals)
```

```
in

(* Execute a statement and return an updated environment *)
let rec exec env = function
   Expr(e) -> let _, env = eval env e in env
                              (*If expressions*)
  | If(e, s1, s2) ->
    let v, env = eval env e in
    let b = (match v with
       Boolliteral(b) -> b
       | _ -> raise (Failure("Invalid conditional expression.")))
      in
       if b then List.fold_left exec env (List.rev s1)
       else List.fold_left exec env (List.rev s2)
                              (*While expressions*)
  | While(e, s) ->
    let rec loop env =
      let v, env = eval env e in
      let b = (match v with
        Boolliteral(b) -> b
        | _ -> raise (Failure("Invalid conditional expression.")))
       in
        if b then loop (List.fold_left exec env (List.rev s))
        else env
          in loop env
                           (*For expression*)
 | For(e1, e2, e3, s) ->
    let _, env = eval env e1 in
    let rec loop env =
      let v, env = eval env e2 in
      let b = (match v with
        Boolliteral(b) -> b
        | _ -> raise (Failure("Invalid conditional expression.")))
       in
        if b then let _, env = eval (exec env s) e3 in
        loop env
      else
        env
     in loop env
 (*Read takes input from the user and assigns it to a variable*)
  | Read(var) ->
    let input = read_line() in
      let v = (match input with
         a -> Stringliteral(a)
  |_ -> raise(Failure("Invalid input"))) in
      let ret, env = eval env (Assign(var, v)) in env
(*Print, prints out a variable*)
 | Print(e) ->
      let v, env = eval env e in
      begin
        let str = (match v with
          Boolliteral(b) -> string_of_bool b
        | Intliteral(i) -> string_of_int i
        | Cardliteral(c) -> "[Card: " ^ c ^ "]"
        | Stringliteral(s) -> s
        | Handliteral(a, b) -> a ^ " value is " ^ string_of_int b
```

```
            | _ -> raise (Failure ("Invalid print expression.")))
          in
          print_endline str;
          env
         end
     (*Deal, takes the topcard from the mydeckstringref, removes that card
from the deckstring
             and assigns the  card to a variable*)
     | Deal(var) -> let topcard = List.hd !mydeckstringref  in
      mydeckstringref := List.tl !mydeckstringref;
     (*Remove topcard from the deck*)
     let v = (match topcard with
            a,b -> Handliteral(a,b)
      |_ -> raise(Failure("Invalid input"))) in
     let ret, env = eval env (Assign(var, v)) in env
      (*Value prints the int part of a handliteral, the card's value*)
     | Value(e) -> let v, env = eval env e in
          begin
            let str = (match v with
            Handliteral(a, b) ->string_of_int b
            | _ -> raise (Failure ("Invalid Value expression.")))
          in
          print_endline str;
          env
         end
   | 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 (Intliteral(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 (Intliteral(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"))
```

9.7 MakeFile

```
OBJS = parser.cmo scanner.cmo printer.cmo interpret.cmo aoc.cmo

aoc : $(OBJS)
      ocamlc -o aic $(OBJS)

scanner.ml : scanner.mll
      ocamllex scanner.mll

parser.ml parser.mli : parser.mly
      ocamlyacc parser.mly

%.cmo : %.ml
      ocamlc -c $<

%.cmi : %.mli
      ocamlc -c $<
```

9.8 Test Script: Test.bat

```
@echo off

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

aic.exe %%~nX.mc > %%~nX.ans

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

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

IF ERRORLEVEL 0 echo %%~nX PASS

:end

REM echo just a label

)
```

Test.bat output

```
test-arith1 PASS

test-arith2 PASS

test-fib PASS

test-for1 PASS

test-func1 PASS

test-gcd PASS

test-hello PASS
```

test-if1 PASS

test-if2 PASS

test-if3 PASS

test-if4 PASS

test-ops1 PASS

test-var1 PASS

test-while1 PASS

9.9 test-arith1.mc

```
main()
{
  Print(39 + 3);
}
```

9.10 test-arith2.mc

```
main()
{
  Print(1 + 2 * 3 + 4);
}
```

9.11 test-fib.mc

```
fib(var x)
{
  if (x < 2) { return 1; }
  return fib(x-1) + fib(x-2);
}

main()
{
  Print(fib(0));
  Print(fib(1));
  Print(fib(2));
  Print(fib(3));
  Print(fib(4));
  Print(fib(5));
}
```

9.12 test-for1.mc

```
main()
{
  var i;
  for (i = 0 ; i < 5 ; i = i + 1)
  {
    Print(i);
  }
  Print(42);
```

```
}
```

9.13 test-func1.mc

```
add(var a, var b)
{
  return a + b;
}

main()
{
  var a;
  a = add(39, 3);
  Print(a);
}
```

9.14 test-gcd.mc

```
gcd(var a, var b) {
  while (a != b) {
    if (a > b) {
      a = a - b;
     }
    else { b = b - a; }
  }
  return a;
}

main()
{
  Print(gcd(2,14));
  Print(gcd(3,15));
  Print(gcd(99,121));
}
```

9.15 test-hello.mc

```
main()
{
  Print(42);
  Print(71);
  Print(1);
}
```

9.16 test-if1.mc

```
main()
{
  if (true) {Print(42); }
  Print(17);
}
```

9.17 test-if2.mc

```
main()
{
  if (true) { Print(42); } else { Print(8); }
  Print(17);
}
```

9.18 test-if3.mc

```
main()
{
  if (false) { Print(42); }
  Print(17);
}
```

9.19 test-if4.mc

```
main()
{
  if (false) { Print(42); } else { Print(8); }
  Print(17);
}
```

9.20 test-ops1.mc

```
main()
{
  Print(1 + 2);
  Print(1 - 2);
  Print(1 * 2);
  Print(100 / 2);
  Print(99);
  Print(1 == 2);
  Print(1 == 1);
  Print(99);
  Print(1 != 2);
  Print(1 != 1);
  Print(99);
  Print(1 < 2);
  Print(2 < 1);
  Print(99);
  Print(1 <= 2);
  Print(1 <= 1);
  Print(2 <= 1);
  Print(99);
  Print(1 > 2);
  Print(2 > 1);
  Print(99);
  Print(1 >= 2);
  Print(1 >= 1);
  Print(2 >= 1);
}
```

9.21 test-var1.mc

```
main()
{
  var a;
  a = 42;
  Print(a);
}
```

9.23 test-while1.mc

```
main()
{
  var i;
  i = 5;
```

```
  while (i > 0) {
    Print(i);
    i = i - 1;
  }
  Print(42);
}
```

9.23 blackjack.txt

```
/*This is the starting point of the function*/
main()
{
   var NDecks;
   var Dealer;
   var numplayers;
   var CName;
   var dealer;
   var player1;
   var player2;
   var player3;
   var i;
   var pvalue;
   var dvalue;
   var anothercard;
   var bust;
   NDecks = 1;
   Dealer = true;

   CName = "BlackJack";
   /*Print out what card game*/
   Print("You are playing");
   Print(CName);

   Print("How many players are playing?  Enter a number between 1 and 3");
   Read(numplayers);
   while(numplayers < 1 || numplayers > 3)
   {
      Print("Wrong number of players! How many players are playing?  Enter a
number between 1 and 5.");
      Read(numplayers);
   }

   /*Shuffle(Deck); Shuffle isn't implemented yet */

   Deal(dealer); /*This will take the top card from the deck and add it to
the dealer's hand.*/
   Print("The dealer's card is");
   Print(dealer);

   i = numplayers;

   while(i > 0)
   {

      if(i == 0)
      {
         Deal(player1);
```

```
        Deal(player1);
        Print("Players one's cards are:");
        Print(player1);
    }

    i = i - 1;

}


i = numplayers;

while(i > 0)
{
  bust = false;
  anothercard = "yes";
  pvalue = 11; /*valueofhand(player[numplayers - i])*/
  Print(numplayers - i + 1);
  Print("Player's cards are worth");
  Print(pvalue);

  while(anothercard == "yes")
  {
     Print("Player, do you want another card? yes or no?");
     Read(anothercard);
     if(anothercard == "yes")
     {
        /*Deal(player[player[numplayers - i + 1], 1);*/
        /*Print(player[player[numplayers - i + 1][0]);*/
        pvalue = 20; /*call function*/
        Print(numplayers - i + 1);
        Print("Player's cards are worth");
        Print(pvalue);
        if(pvalue > 21)
        {
          bust = true;
          Print("Player went over 21 you lose");
        }
     }

    }/*Next players turn*/

  i = i - 1;
}/*Dealers turn*/

Print("Dealer's turn");
Print("Dealer's cards are:");
/*Print(dealer[0]);*/
/*Print(dealer[1]);*/
dvalue = 15; /*call function*/
Print("Dealer's cards are worth");
Print(dvalue);

i = numplayers;

while(i>0)
{
```

```
      /*Deal the dealer cards until he goes bust or ties or beats the players
hand and make sure the player hasn't already gone bust*/
      pvalue = 20; /*call function*/
      dvalue = 10; /*call function*/
      while(pvalue <= 21 && dvalue <= pvalue && dvalue<=21)
      {
         /*Deal(dealer, 1);*/
         /*Print(dealer[3]); Prints the cards the dealer has*/
         dvalue = 21; /*call function*/
         Print(dvalue);
      }
     i = i - 1;
    }

  /*At this point the dealer has compared his cards to all the players and
either gone bust or has a higher score or equal score then    all of them.
Dealer wins if the scores are equal in this version.*/
  if(dvalue > 21)
  {
     Print("Dealer went bust, everyone wins");
  }
  else
  {
     Print("Dealer wins");
  }


}
```

9.23 highcard.txt

```
/*This is the starting point of the function*/
main()
{
   var NDecks;
   var Dealer;
   var numplayers;
   var CName;
   var dealer;
   var player1;
   var player2;
   var player3;
   var i;
   var pvalue;
   var dvalue;
   var anothercard;
   var bust;
   NDecks = 1;
   Dealer = true;

   CName = "HighCard";
   /*Print out what card game*/
   Print("You are playing");
   Print(CName);

   Print("How many players are playing?  Enter a number between 1 and 3");
```

```
   Read(numplayers);
   while(numplayers < 1 || numplayers > 3)
   {
      Print("Wrong number of players! How many players are playing?  Enter a
number between 1 and 5.");
      Read(numplayers);
   }

   /*Shuffle(Deck); Shuffle isn't implemented yet */

   Deal(dealer); /*This will take the top card from the deck and add it to
the dealer's hand.*/
   Print("The dealer's card is");
   Print(dealer);

   Deal(player1);
   Print("Players one's card is:");
   Print(player1);

   if(numplayers > 1)
   {
      Deal(player2);
       Print("Players two's card is:");
      Print(player2);
   }
   if(numplayers > 2)
   {
      Deal(player3);
      Print("Players three's card is:");
      Print(player3);
   }

   if(dealer > player1)
   {
      if(dealer > player2)
      {
         if(dealer > 3)
         {
           Print("Dealer wins");
         }
         else
         {
           Print("Player 3 wins");
         }
      }
      else
      {
         if(player2 > player3)
         {
            Print("Player 2 wins");
         }
         else
         {
            Print("Player 3 wins");
         }

      }
```

```
    }
    else
    {
      if(player1 > player2)
      {
         if(player1 > player3)
         {
            Print("Player1 wins");
         }
      }
      else
      {
         if(player2 > player3)
         {
            Print("Player2 wins");
         }
         else
         {
           Print("Player3 wins");
         }
      }
    }
}
```