Final Report on Cards the Language

By Jeffrey C Wong
jcw2157

Table of Contents

1 Introduction

The idea of a Cards language was to provide a tool to allow game creators the flexibility of tweaking existing games into a new game. The language allows the programmer to quickly write up rules to any standard 52 deck cards game and convert it into a playable game.

The advantage of Cards is that the data types bundled in the language. The programmer won't have to worry about declaring data types in the language. This will let the game creator devote more time to creating a new type of game with one less worry in the source code.

1.1 Purpose

To provide a simple yet powerful tool for game creators to create a new game from existing card games.

1.2 Ease of Use

A person without any programming experience will be able to use create an game with this language.

1.3 Modifiable

Instead of traditional rules to a card game, the game creator can create a game with custom rules.

2 Tutorial

2.1 Language Reference

2.1.1 Lexical Conventions

Whitespace is defined as either tab or space character. Except for Whitespace between double quotes (these will be part of the string literal), any Whitespace will be discarded by the scanner.

Keywords

The following strings of characters are keywords and reserved by the compiler.

print
scan
break
return
if
else
while
total
pass
shuffle
reveal
hide
number
suit
color
showing
misc
group

Predefined Functions

Print()
Outputs to screen the parameters in the () enclosure.

Scan()
Takes user input from the screen.

Main()
A Specialized function that all programs begin from.

Shuffle()

Shuffles the decks and begins the program. The Shuffle command is initializes the deck of cards. Depending on the parameter given, the deck will initialize, the cards to the proper values. For example in BlackJack, the values of the face cards have strength of ten. The Ace can be either one or eleven. The number cards are equal to the number should shown on the card.
The following table shows possible decks that can be initialized. More deck types will have to be create for more games. The basic premise is that the programmer can create rules to the game without having to declare variables.

Game Type	Parameter
BlackJack	bj
Baccarat	ba
War	wa
Hearts	he
Rummy	ru
Spades	sp
…(etc)	…(etc)

Pass()

Pass passes the cards out to the grouping. Pass takes in two parameters. The first parameter is where the card is going to and the second parameter is the number of cards to pass.

Discard()

Removed the card in the parameter and places it in the discard. This function takes in one parameter which is the card to discard.

Reveal ()

Changes the value of the card to reveal. Allows for other groups to know the value of the card and in which group it belongs to.

Total()

Incase of addition, this function all the value of the cards in the grouping. The parameter is the grouping of the cards.

Return

The return statement is used to return the function to where the function was initially invoked. The return does not return anything.

Comments
Comment are recognized by the compiler with /* */ notation. Any string between the * marks will be marked as a comment by the compiler.

Identifiers

An identifier in the Cards language is a string of alphanumeric characters,

Operators

The following operators are used the Cards language. The list below ranks the operator precedence rank from highest to lowest order. The operators with equal precedence will be evaluated from left to right associative.

+ -	(add, subtract)
> <	(greater than, less than)
= !	(equal to, not equal to)
<-	(assignment)
& |	(and, or)

Additive Operators
The operators + and – perform addition and subtraction. Both Operators are used only on numeric values. The result of these operators is an integer.

Relational Operators
The relations operators > and < perform the operation GREATER THAN and LESS THAN respectively. Operands must be of arithmetic type. The result of the expression is true or false.

Equality Operators
The operators == and != perform EQUALS TO and NOT EQUALS TO. The results of these operations are true and false. The == operator will produce and true result if both operands equal and false if the operands do not equal. The != operator produces the opposite result of == operator with the same operands.

Logical Operators
The operators & and |. The operations are AND and OR respectively. The results of the operators is true or false. The compiler is left associative. If the left expression is true for operator |, then the result is true. Otherwise the compiler will check the right expression. If both expression aren't true, then the result if false. The & operator checks if either left or right expression is false and returns true if both are not false.

Grammar

Below is a table of symbols used for the Cards language

{}	Groups statements in the main and if and else block
()	Holds the arguments of a function
,	Separates the arguments in a function
;	Marks the end of a statement

2.2 Sample code

```
main() {
   shuffle(bj);  /* this is the initialization of a standard deck of cards  for blackjack*/
   players(1);  /* number of players */
   pass(1, 2); /* pass to player one 2 cards, this command updates the deck */
   pass(d, 2); /* pass to dealer 2 cards */
   while(true) {
      if(total(add(1))< 21){
      while(scan(equals(H)) | scan(equals(h)) | scan(equals(0))){  /* 0 is the initial result to use the loop */
      print(total(add(1))); /* totals the value of the cards and prints it to the screen */
      print("Your cards are ");
      print(cards(1));                    /* shows the cards  to the player */
      print("hit (H) or stay(S) > ");
      if(scan(equals(H) | scan(equals(h)))) {
        passCards(1,1));
      } else {
        break;
      }
      if(total(add(1) > 21)){
       print("busted");
       return;
      }
   }
  } else {
    print("black jack!");
   }
  /* dealer's turn */
  while(total(add(d) < 17)) {
     pass(d,1));
  }
  if(total(add(d) > 21) | (total(add(1) > total(add(d))))) {
    print("player wins");
  }else if( total(add(d)) = total(add(1))) {
    print("draw");
   }else {
    print("dealer wins");
   }
  }
return;
```

3 Compiler Interface

All files used to compile with the cards language have to end with the ".cds" file prefix
If the source name is myCards.cds, the command to compile the file will be

./cards <PATHNAME TO myCards.cds> outputName.out

This will produce the following file

outputName.out

./outputName.out will run the file and make a command line executable.

4 Lessons Learned

I've should of started earlier. When designing a new language, make sure you have sufficient time work on the project. I've found it hard to learn a new language when I'm so used to thinking Java. Functional programming is a completely new way of programming. I did not completely understand how to program with O'caml until halfway thru the semester. By then I was really behind on this project. I've always found myself short on time. The scope of my project seemed to get larger, the more I worked on it. I think I've took on a project too ambitious for one person to complete in the time given.

If you're working on this project by yourself, make sure you have a full understanding of what is needed to be done.

"Learn as you go" didn't seem to work for me when I was working on this project.

I've also realized I needed more on my Language Reference Manual when I was writing the code for the Cards language. There were a few things I wanted to change for my Language Reference Manual. The more I try to create my language, the more I felt that my language was just another implementation of C++ or Java with some abstraction built into the handling of the players and the cards.

Get used to thinking top bottom coding. I was too far used coding from the bottom up.

Overall I found this to be a very learning experience. I think differently about how compilers work because of this project. One recommendation I have for future teams is to write the Final Report first. This is will allow for you to really think things thru before you start coding.

5 Files

cards.ml

```
let print = false

let _ =
        let lexbuf = Lexing.from_channel stdin in
        let top = Parser.top Scanner.token lexbuf in
                Interpret.run top
```

parser.mly

```
%{ open Ast %}

/* these are pulled from scanner.mll */
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token DQUOTE
%token ADD MINUS TIMES DIVIDE ASSIGN
%token EQUALS NEQUALS LESSTHAN GREATER
%token PRINT TOTAL PASS SHUFFLE
%token NUMBER GROUP SUIT COLOR
%token SHOWING REVEAL MISC HIDE
%token RETURN IF ELSE WHILE INT
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%left ASSIGN
%left LESSTHAN GREATER
%left EQUALS NEQUALS
%left ADD MINUS
%left TIMES DIVIDE

%start top
%type <Ast.top> top

%%

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

  /* variable declaration */
vdecl:
   INT ID SEMI { $2 }

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

  /* separate out the stmt */
stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | LPAREN DQUOTE expr DQUOTE RPAREN { Print($3) }
  | 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) }

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

expr:
    LITERAL           { Literal($1) }
  | ID               { Id($1) }
  | expr ADD       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 EQUALS    expr { Binop($1, Equal, $3) }
  | expr NEQUALS   expr { Binop($1, Neq,   $3) }
  | expr LESSTHAN  expr { Binop($1, Less,  $3) }
  | expr GREATER   expr { Binop($1, Greater,  $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 }
| "/*"      { comment lexbuf }
| ""            { DQUOTE }
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| "="       { EQUALS }
| "!"       { NEQUALS }
| '<'       { LESSTHAN }
| ">"       { GREATER }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "int"     { INT }
| "add"       { ADD }
| "print"   { PRINT }
| "total"   { TOTAL }
| "pass"    { PASS }
| "shuffle" { SHUFFLE }
| "hide"    { HIDE }
| "number"  { NUMBER }
| "group"   { GROUP }
| "suit"    { SUIT }
| "color"   { COLOR }
| "showing" { SHOWING}
| "reveal"  { REVEAL }
| "misc"    { MISC }
| ['0'-'9']+ as s { LITERAL(int_of_string s) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as s { ID(s) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

ast.mli

```
type op = Add | Sub | Mult | Div | Equals | Nequal | Less | Greater

type expr =
    Literal of int
  | 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
  | Print of expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Shuffle of expr
  | Pass of stmt
  | Else of expr
  | Scan of expr
  | Hide of expr
  | Total of expr
  | Shuffle of expr
  | Reveal of expr
  | Number of expr
  | Group of expr
  | Showing of expr
  | Suit of expr
  | Misc of expr
  | Cards of expr
  | Color of expr

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

type top = string list * func_decl list
```

interpret.ml

```
open Ast

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

exception ReturnException of int * int NameMap.t

(* The beginning point of top *)
(* called from cards.ml *)
let run (vars, funcs) =
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

  let rec call fdecl actuals globals =
    let rec eval env = function
              Literal(i) -> 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 1 else 0 in
                    (match op with
            | Equal -> boolean (v1 = v2)
            | Nequal -> boolean (v1 != v2)
            | Less -> boolean (v1 < v2)
            | Greater -> boolean (v1 > v2), env
(*          | Call("pass", [e]) ->
             let v, env = eval env e in
           *)
      | 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

    let rec exec env = function
          Block(stmts) -> List.fold_left exec env stmts
      | Expr(e) -> let _, env = eval env e in env
(*            | Pass(e, s1) -> let *)
      | 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
      | Return ->
            let v, (locals, globals) = eval env in
            raise (ReturnException(v, globals))
  (*run the program *)
  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"))
```

6 Architecture

Lexer → Parser and Scanner→ Interpreter (uses the AST) → output

Lexer - scans the inputs and files into tokens
Parser – parses the strings into a abstract syntax tree
Interpreter – interprets the abstract syntax tree