COMS W4115 – Fall 2011
Prof. Stephen Edwards
Final Report

Eric Chao | [ehc2129]
Susan Fung | [sgf2110]
Jim Huang | [jzh2103]
Jack Shi | [xs2139]

DESCARTES

Table of Contents

1 Introduction

Descartes, or “some cards,” is a game design programming language. The language is limited to any operating system capable of running Objective CAML, since all of the programs will be passed through an OCAML compiler. The focus of Descartes will be to implement card games strongly associated with the standard 52-card deck, although it can be used equally well to write any other non-real-time card game of various complexities.

1.1 Motivation

Using a general-purpose language such as Java or C would require many lines of code dedicated to designing the decks, players, different cards, etc. Descartes will allow programmers of all levels to focus solely on creating the game design, the rules behind the games, and encourage beginner coders to quickly sample games for their own learning or entertainment uses.

Below are some card game types that can be created using Descartes followed by a more in-depth explanation of the different card game types.

Card Game Types	Examples
Trick-Taking	Bridge, Whist, Euchre, Spades, Hearts, Twenty-Eight, Tarot Card
Matching	Rummy, Go Fish, Old Maid
Shedding	Phase 10, Rummikub
Accumulating	War, Egyptian War
Fishing	Scopa, Cassino
Comparing	Poker, Blackjack, Baccarat
Solitaire (Patience)	Solitaire
Drinking	Presidents, Daihinmin
Multi-Genre	Pinochle. Belote, Poke, Flaps, Skitgubbe, Tichu
Trading Card Games	Magic: The Gathering, Pokemon, Yu-Gi-Oh!, Marvel Comics
Casino/Gambling	Poker, Blackjack
Fictional	Pyramid, Exploding Snap

1.2 Card Game Types

There is a wide variety of card game types. Descartes is meant to cover this vast spectrum of card games. Some card game example types are trick-taking, matching, shedding, accumulating, fishing, comparing, patience, drinking, collectible card, casino, and many more. Although these are the general categories, some games are a combination of multiple genres.

1. Trick-taking Games

   A trick-taking game is a turn-based game where multiple rounds take place until all players run out of cards. During each round, each player will play a card from their hand and based on the values of the card, the winner will take the played cards.

2. Matching Games

   The objective of matching games is to acquire as many matching cards before your opponents.

3. Shedding Games

   Shedding games' objective is to discard all your cards in your hand before your opponents.

4. Accumulating Games

   Accumulating games require you to acquire all the cards in the deck in order to win the game.

5. Fishing Games

   In fishing games, cards from the hand are played against cards laid out on the table. Table cards are captured if they match.

6. Comparing Games

   Comparing card games involve each player's hand values being compared with those of other players. The player with the highest-value hand is the winner.

7. Patience Games

   Patience games are also known as solitaire games which are only played by one player. Most games start with a specific layout and to win the game, the player must construct another layout or clear the table of cards.
8. Drinking Games

As the name implies, these games involve drinking or forcing other players to drink. There are many typical card games used as drinking games without their original objective. For example, Poker can be played as a drinking game in that the losers may have to consume a beer instead of losing money.

9. Multi-Genre Games

   These games are a combination of two or more types of games. The most common combinations are matching and shedding.

10. Trading Card Games

    Collectible card games consist of decks of proprietary cards that differ among players. The contents of these decks are a subset of a very large pool of available cards that have different attributes. A player accumulates his or her deck by purchasing or trading desirable cards. Each player uses his or her own deck to play against the other players of the same proprietary card types.

11. Casino Games

    Casino card games revolve around wagers of money. The obvious objective is to win as much money as possible. These games are designed for using some sort of strategy to reach this goal.

12. Fictional

    These card games revolve around science fiction. The game is often used to depict a story that involves background activities in a room.

2 Language Tutorial

This section will walk you through creating a simple descartes program. A descartes source file has .des extension. Each program must have one and only one main function. The main function must be the first function defined in the file. All variable declarations must be made before any variable manipulation such as assignment, arithmetic, etc.

Example main function:

```
int main ()

{
```

```
        int a;
        a = 1;
        return 0;
}
```

Other functions must have a return type. Arguments to a function are comma separated and defined in the format: data type variable name semicolon.

Example printString function:

```
int  printString(string a;, int b;)
{
        b = 1;
        while (b)
        {
                print(a);
        }
        return 0;
}
```

To compile a program, the command is:

decartes -c < source-file > output file so if your source file name is test.des, it would be

descartes -c < test.des > test.pl test.pl is your outputted perl file.

3 Language Reference Manual

3.1 Introduction

Card games have been a popular form of entertainment for centuries, evolving from the traditional 52 unique cards (Poker) to over 10000 unique cards (Magic the Gathering). Descartes is a specialized computer language specifically designed to allow the easy creation of simple card games that use the standard 52-card deck. This Language Reference Manual is intended to help developers understand how to develop their own card game using this language and also describes the different components in the language that can be used.

3.2 Printing

To print constants, use the print keyword. Commas between constants denote adding spaces. Integers and Booleans are automatically converted to Strings and printed. Expressions that can be evaluated to constants are evaluated and printed. When calling print on nonconstant objects, the toString method is called on the object and the returned value is printed.

Example:
```
String A = “100”;
int i = 999;

print i,“-”,A, “=”, 999-100;
```

Output:
```
999 – 100 = 899
```

3.3 Lexical Convention

There are four types of tokens that Descartes uses: comments, identifiers, constants and keywords. Blanks, tabs, newlines, and comments are ignored unless used as token separators. At least one of these characters must be used to separate the other adjacent tokens.

3.4 Comments

Comments start with the characters /* and are terminated with */

Examples:
```
/* This is a comment */
/* This is

a multi-line

comment */
```

3.5 Identifiers

An identifier is a sequence of letters, numbers, and/or underscore characters '_'. The identifier must start with a character. Upper and lower case letters are considered different. For example, the identifier "cat" is different from "Cat."

3.6 Constants

There are several types of constants: integer, string, escape, boolean, tuple and list.

3.6.1 Integer constants

An integer constant is a sequence of digits.

3.6.2 String constants

A string constant is a sequence of characters enclosed in double quotes ("). If the string needs a double quote to be part of the string, it must use the escape constant with a back slash (\").

3.6.3 Escape constants

An escape constant is a special string constant of 1 or 2 characters preceded by a backslash. Without a backslash, they would be regular string constants:

Escape constant	Description
\n	new line
\t	tab
\'	single quote
\"	double quote
\\	back slash

3.6.4 Boolean constants

A boolean constant is used to define whether an expression is true or false. It has either a true or false value. As an alternative, the integer constant 0 can be used in place of true and any other integer constant can be used to represent false.

3.6.5 Tuple constants

A tuple constant is an ordered set of string, boolean, or integer constants separated by commas and enclosed in parenthesis except when included in a list as stated below.

Example:
tuple a = (1,2);
tuple b = ("alpha",100);
tuple c = (true, false);

3.6.6 List constants

A list constant is an immutable sequence of the same data type. It is defined by placing the list items in brackets using a semicolon to separate each item. A List can contain tuple constants which are represented as comma-separated-values without a parenthesis.

Example:

List aList = [1;2;3;4]; /* This is a list of 4 integers */

List a = [1,2; 3,4; 5,6]; /* This is a list of 4 integers */

3.7 Keywords

The following identifiers are reserved as keywords and cannot be used other than its sole purpose:

int	break	for	string	true
return	print	while	list	false
if	else	goto	bool	

3.8 Default Object Types

Descartes includes default object types that allow the creation of representations of simple games that use the standard 52-card deck. These objects are Card, CardStack, Player, Field and Game. These objects can be customized for a specific card game.

3.8.1 Card

This object represents a card to be played in the game.

Functions	Description
initialize(String value)	Returns a card object. Takes in a String value (“5H”).
toString()	Returns a printable descriptive string representation of the object. (“5H”)
getValue()	Returns the value (A,2,3…J,Q,K) of the card as a string.
getSuit()	Returns the suit (spades, hearts, diamonds or clubs) of the card as a String
getSuitLetter()	Returns the suit (S, H, D, or C) of the card as a 1-character String
getColor()	Returns the color (black or red) of the card as a String
getColorLetter()	Returns the color (B or R) of the card as a 1-character String
getVisibility()	Returns a list of Players that can see this card
setVisibility(Player [])	Sets which players have access to the card. It takes an array of Player objects as a parameter
removeVisibility(Player [])	Revokes players’ access to the card. It takes an array of Player objects as a parameter

Cards are compared based on value and suit. Equal comparisons are based on suit and value. Less than and more than comparisons are based on value.

You can also use shorthand *Card A = (Card) “S5”* instead of *Card A = Card.initialize(“S5”)*

Examples:

```
Card A = Card.initialize("S5");

String x = A.getValue();

String y = A.getSuit();

String z = A.getColor();

print x,y,z;
```

Output:

```
5 S black
```

3.8.2 CardStack

A card stack can be the deck of the card or an individual hand that a player has.

Functions	Description
initialize(int)	Generates a cardStack object. This is to represent either a deck to pick from or an individual player's hand. Must be called first to declare. It takes an integer as a parameter that defines how many cards to create in this stack.
toString()	Returns a printable string representation of the object.
setPoints(int)	This is how many points the card stack is worth. It takes an integer as a parameter
getPoints()	Returns the number of points in the card stack
changePoints(int)	Takes in an integer and changes the number of points by that number.
addCard(Card, int)	Adds a Card to the stack. It takes a Card object as a parameter and an int to represent where in the stack to add the card.
getCard(String)	Returns a Card in the stack. It takes a String, the value of the card, as a parameter.

drawCard()	Removes the first Card off the stack and returns a Card object
drawCard(int)	Takes a positive integer as a parameter and returns that many Card objects in reverse order.
shuffle()	Randomizes the sequence of the cards in the card stack
reverse()	Reverses the current sequence of cards in the card stack
contain(Card)	Returns a boolean whether a card is in the stack or not. It takes a Card object as a parameter
setVisibility(Player [])	Sets which players have access to the cards in the card stack. This is the default access if the individual access in the Card objects is not set. It takes an array of Player objects as a parameter
removeVisibility(Player [])	Revokes players' access to the cards in the stack. It takes an array of Player objects as a parameter
getVisibility()	Returns an array of Players that can access this card stack
size()	Returns an integer that describes the size of the card stack
default()	Generates the CardStack with the default 52-card deck

CardStacks support the plus (+) and minus (-) operators.

The addition operator adds CardStacks together in order, as if the right CardStack is stacked on top of the left one.
Example:
CardStack deck = CardStack.default() + CardStack.default();

The subtract operator removes Cards and is the same as getCard.
Example:
CardStack A = CardStack.default();
print (A.getCard(“A5”) == A-“A6”);
Output:
False

3.8.3 Player

A player is the person involved in a game. It can be a dealer or any participant of the game.

Functions	Description
setName(String)	Sets who the player's name. It takes a string as a parameter
getName()	Returns the name of the player
setBetSize(int)	Sets the current size of the bet the user wishes to place. If not set, the default is 0 meaning that no bet is required for the game.
getBetSize()	Returns the size of the bet
setPoints(int)	Sets the total points in the player has. If not set, the default is 0. It takes an integer as a parameter.
getPoints()	Returns an int that describes the total points a player has.
setPlayerType(String)	Sets the type of player. It takes a string as a parameter. Some examples of player types are "dealer", "team A"
getPlayerType()	Returns a string to represent the type of the player
setCardStack(CardStack)	Sets the hand that the player holds. It takes a CardStack as a parameter
getCardStack()	Returns a CardStack that represents the player's hand

3.8.4 Field

This object will help in dividing a board game if needed.

Functions	Description
setName(String)	Sets what the field's name. It takes a string as a parameter
getName()	Returns a string that represents the field's name
setValue(String)	Sets the value of the field. It takes a string as a parameter

getValue()	Returns a string that represents the value of the field
setPosition(int)	Takes in an integer representing the position of the field. Possible values are integers from 0 to 9 and the positions correspond to the same positions of the numbers on a standard keyboard numeric keypad. This corresponds to where the CardStacks of this field are printed. The default position is 0 which denotes that the CardStack is not printed.
getPosition()	Returns the position as an integer value.
addCardStack(CardStack)	Adds a CardStack to the Field.
getCardStack()	Returns an array of CardStacks in the Field in order.
removeCardStack(int)	Removes a CardStack from the Field.
setCardStack(CardStack[])	Sets the Field's CardStacks to an array of given CardStacks.

3.8.5 Game

This is the core of the card game. It encapsulates all the required elements that make up a card game.

Functions	Description
getPlayers()	Returns the List of Player objects currently in the game
setBetSize(int)	Sets the size of the bet for a given game. It takes an int as a parameter. If not set, the default is 0 meaning that no bet is required for the game.
getBetSize()	Returns an int that represents the size of the bet for a given game
end()	This stops/finishes the game and determines the winner. It returns a Player object
start()	This starts the game.
nextTurn()	This will control the order that the Players go. This returns the Player object
move(Card, CardStack,	This will move a Card Object from one card stack to another card stack. It removes the card in the second parameter

CardStack)	(CardStack) to the third parameter (CardStack)
setTurnOrder(List Player)	This sets the order the players' turn in the game. It takes a Player List as a a parameter
getTurnOrder()	Returns a List of Player objects to represent the order that the players are playing in
setFields(Field[])	Takes in an array of Fields and adds them to the Game. There must be no Fields with the same printed position (multiple Fields can have position 0.
print()	prints the CardStacks according to Fields.

3.9 Conversions

The cast operator can be used to convert numerical types (int) into string and the reverse. It can also convert Cards to Strings and the reverse.

(String) 122 will convert into "122"
(int) "122" will convert into 122
(String) Card.initialize("A5") will convert into A5.
(Card) "A5" will convert into the corresponding card.

3.10 Primary Expressions

The primary expressions in Descartes are identifiers, constants, strings, and expressions contained inside parentheses.

```
primary_expression:
identifier
constant
literal
(expression)
```

An identifier is a primary expression only if it has the lexical conventions as defined in section 2.2. Each identifier must have a type, which is determined by its declaration.

A constant is a primary expression only if it has the lexical conventions as defined in section 2.3 and is one of the types defined in Descartes.

A literal is a primary expression that has the primitive type string. It must follow the lexical conventions of the type string as defined in section 2.3.2 and is immutable.

An expression contained inside parentheses is a primary expression that has the same type and value as that not contained inside parentheses. The parentheses are only used to administer order of operations.

3.11 Unary Operators

```
unary_expression:
    postfix_expression
    unary_operator expression

unary_operator:one of
    +
    -
    !
```

The unary plus operator (+) must have an operand of an arithmetic type. The type and value of the result are consistent with those of the operand.

The unary minus operator (-) must have an operand of an arithmetic type. The type of the result is consistent with that of the operand. The value of the result is the negative value of the operand.

The unary negation operator (!) must have an operand of boolean type. The type of the result is boolean and the result is true if the value of the operand compares equal to false and the result is false if the value of the operand compares equal to true.

3.12 Cast Operators

```
cast_expression:
    unary_expression
    (type_name) cast_expression
```

The cast operator will convert the expression after the parentheses to the desired type specified in the parentheses before. It only operates on the unary expression immediately following the operator unless parentheses are used to alter.

3.13 Multiplicative Operators

```
multiplicative_expression:
    unary_expression
    multiplicative_expression * unary_expression
    multiplicative_expression / unary_expression
    multiplicative_expression % unary_expression
```

The multiplicative operator * evaluates from left to right and denotes multiplication. The * operator can only take integers as operands and is a binary operator. The result is the expected arithmetic calculation, which is an integer.

The multiplicative operator / evaluates from left to right and denotes division. The / operator can only take integers as operands and is a binary operator. The result is integer quotient for integer operands.

The multiplicative operator % evaluates from left to right and denotes the modulus function. The % operator can only take integers as operands and is a binary operator. The result is an integer remainder of the division of the first operand by the second.

Additive Operators

```
additive_expression:
    multiplicative_expression
    additive_expression + multiplicative_expression
    additive_expression – multiplicative_expression
```

The additive operator + evaluates from left to right and denotes addition. The + operator can only have integers as operands. It is a binary operator, so both operands must only be integers. The + operator also denotes string concatenation, but only if both operands are or string type.

The additive operator - evaluates from left to right and denotes subtraction. The -operator can only have integers and floats as operands separately. It is a binary operator, so both operands must only be integers or only floats.

3.14 Relational Operators

Relational expressions can only evaluate to the result of true or false, which can be expressed as 1 an 0, respectively.

```
relational_expression:
    additive_expression
    relational_expression < additive_expresion
    relational_expression > additive_expression
    relational_expression <= additive_expression
    relational_expression >= additive_expression
```

The relational operator < evaluates left to right and denotes less than. The < operator can only have integers as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be integer 0.

The relational operator > evaluates from left to right and denotes greater than. The > operator can only have integers as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be integer 0.

The relational operator <= evaluates from left to right and denotes less than or equal to. The <= operator can only have integers as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be integer 0.

The relational operator >= evaluates from left to right and denotes greater than or equal to. The >= operator can only have integers as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be integer 0.

3.15 Equality Operators

```
equality_expression:
    relational_expression
    equality_expression == relational_expression
    equality_expression != relational_expression
```

The equality operator == evaluates from left to right and result in true or false, which can be expressed as integer 1 and integer 0, respectively. The == operator denotes equal to and accepts integers and strings as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be the integer 0.

The equality operator != evaluates from left to right and result in true or false, which can be expressed as integer 1 and integer 0, respectively. The != operator denotes not equal to and accepts integers and strings as operands. The result, if the expression evaluates to true will be integer 1 and the result, if the expression evaluates to false will be the integer 0.

In Descartes, equality operators only compare by value.

3.16 Logical AND Operator

```
logical_AND_expression:
    equality_expression
    (logical_AND_expression AND equality_expression)
```

The logical AND operator is represented by &&. If the expression evaluates to true, the result will be integer 1 and if the expression evaluates to false, the result will be integer 0. The parentheses are required for the logical AND expression.

3.17 Logical OR Operator

```
logical_OR_expression:
```

logical_AND_expression
(logical_OR_expression OR logical_AND_expression)

The logical OR operator is represented by ||. If the expression evaluates to true, the result will be integer 1 and if the expression evaluates to false, the result will be integer 0. The parentheses are required for the logical OR expression.

3.18 Assignment Operator

expression:
logical_OR_expression
unary_expression assignment_operator expression

The assignment operator is represented by =. The type of the left operand must be the end type of the operand on the right. The value that the expression on the right evaluates to replaces the value of the left operand.

3.19 Declarations

Declarations are used within function definitions to specify the interpretation of each identifier. A declaration is composed of declaration-specifiers (one or two) and the necessary declarator list (any number one or more).

Form:
declaration-specifiers declarator-list;

Sample:
List lista;
List lista, listb;

The declarator-list contains a number of comma-separated identifiers being specified. The declaration-specifier consists of optional storage specifiers and one type-specifier.

Form:
storage-specifier type-specifier

Sample:
static final List lista;

3.19.1 Storage Specifier

The possibilities are:
static - Shared by any instance of the class
no-modifier - Unique to that class
final - Cannot be changed, immutable.

3.19.2 Type-Specifiers

int
boolean
String
List
standard and user-defined types

3.20 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

Form:
declarator
declarator, declarator-list

The specifiers in the declaration indicate the type and storage of the objects to which the declarators refer.

Declarator Form:
identifier
declarator [constant-expression] or declarator[]

3.20.1 Meaning of declarators

Each declarator is an assertion that when the declarator form is used in an expression, it yields an object of the specified type and storage. Each declarator is declaring one identifier and if an identifier without specifiers appears as a declarator in a declarator-list, then it has the type indicated by the specifier heading the declarator list.

A declarator may have the form
D[constant-expression] or D[ ]
The constant expression represents the a compile-time-determinable value whose type is int and defaults to 1 if left blank. This generates an array of the type of the declarator identified by the identifier.

Some restrictions are: functions may not return functions and there are no arrays of functions.

3.21 Statements

Statements in DesCartes are executed sequentially unless otherwise noted. There are several different types of statements.

3.21.1 Expression Statement

The majority of statements in DesCartes are expression statements, which typically make assignments or call functions. The format of the expression statement is:

expression;

3.21.2 Compound Statement

A sequence of statements can be executed where one is expected by using the compound statement:

compound-statement:
{ statement-list }

statement-list:
statement
statement statement-list

3.21.3 If Statement

There are two basic conditional statements in DesCartes:

if (expression) statement
if (expression) statement else statement

In both cases, the expression is evaluated first. If it is true, the following statement is executed. If it is false, the first conditional statement does not do anything but the second conditional statement executes the statement indicated by else.

3.21.4 While Statement

The while statement allows for looping over a statement until a certain condition is no longer valid. The format is as follows:

while (expression) statement

The statement is executed repeatedly until the expression is no longer true. The test of the expression happens before each statement is executed.

3.21.5 For Statement

The for statement is another looping statement with the following format:

for (expression-1; expression-2; expression-3) statement

It is equivalent to:

```
expression-1;
while (expression-2) {
statement
expression-3;
}
```

Expression-1 initializes the loop. Expression-2 sets the condition of the loop that is tested each time the loop starts. Expression-3 usually determines the increment value that is considered after each iteration, which allows for looping over the statement a finite number of times.

Any or all of the expressions in the for statement may be dropped, resulting in the for statement's equivalent without the dropped expressions.

3.21.6 Break Statement

The break statement terminates the smallest while, do, for, or switch statement and allows for the execution of the statement following the terminated statement.

Form:
break;

3.21.7 Continue Statement

The continue statement skips the current iteration of a while, do, or for statement.

Form:
continue;

3.21.8 Return Statement

A return statement allows a function to return to its caller. The two forms are:

Form:
return;
return (expression);

In the second case, the value of the expression is returned to the caller, converted to the proper type if needed. Note that transferring flow to the end of a function is equivalent to the first case.

3.22 Scope Rules

The lexical scope of an identifier is the region of a program during which it may be used. The lexical scope of external definitions, those outside functions and compound statements, extends from their definition to the end of the file. The lexical scope of names declared at the head of functions is limited to the body of the function. Declaring identifiers already declared in the current scope will cause an error.

3.23 Special Compiler Commands

3.23.1 Special Compiler Commands

This is an explanation of some special compiler commands not especially treated elsewhere in this document.

3.23.2 Token replacement (static final)

A compiler-control line of the form:
Static final identifier token-string

Without a trailing semicolon will cause the compiler to replace all instances of the identifier after this line with the string of tokens, given that the string of tokens is a constant. This is the same as Java's static final declaration. The replacement token-string has comments removed from it, and it is surrounded with blanks. It is treated as a constant.

Sample:
static final size 100
int List[size];

3.23.3 Compiling and Running a program

Our language will be compiled to a java program so sample command-line instructions can be:

desc Texas.des -> Creates a Texas.java
javac Texas.java -> Creates a Texas.class
java Texas -> Runs the Texas java program.

"desc" will compile a ".des" Descartes program into a JAVA file.
Then, JAVA-related commands "javac" and "java" will respectively compile a java program into a JAVA class and run the compiled JAVA program.

The rationale for this is

1) Our language is built to be similar to JAVA but specialized for a game using a 52-card standard deck and compiled by/written using O'Caml.
2) JAVA is universal and there is a high chance that the JAVA runtime environment is already installed in the machines on which our language is used.

3.24 Example

This sample code shows what the setup for a game of Texas Hold'Em might look like:

```
Texas
    {
        CardStack deck, p1Hand, p2Hand;
        Player[] players;
        Game(int numPlayers)
        {
                /* If number of players is not 2, print "Wrong number of players." and quit.*/
                if(numPlayers!=2)
                {
                        print "Wrong number of players.";
                        end();
                }
                /* Create a deck composing of 2 default 52 card decks. */
                CardStack Deck1 = CardStack.default().shuffle();
                CardStack Deck2 = CardStack.default().shuffle();
                CardStack deck = Deck1 + Deck2;
                deck.visibility = [];
                /* Initialize players and hands with default visibility. */
                Player P1 = Player();
                Player P2 = Player();
                players = [P1;P2];
                p1Hand.visibility = [P1];
                P1.setCardStack(p1Hand);
                p2Hand.visibility = [P2];
                P2.setCardStack(p2Hand);
                /* Deal card to each player. */
                Deal(2, [p1Hand,p2Hand]);
        }

        /* Deal function. Moves numCards from top of deck to designated card stack.*/
        Deal(int numCards, CardStack[] cardStacks)
        {
                /* For each card stack, move numCards from deck to that stack. */
                for(int i = 0; i < cardStack.size(); i++)
                {
```

```
                    deck.drawCard(numCards)>> cardStack[i];
            }
        }
    }
```

4 Project Plan

4.1 Process

4.1.1 Planning

In planning out our project, we first decided on an idea for our project. We then worked out an efficient plan to go about implementing the idea.

In coming up with a project idea, we brainstormed ideas that were both interesting and feasible. We wrote down various ideas given the objectives and scope of our project, discussing the pros and cons of each. We slowly narrowed the list down and ultimately decided on our card-game design language. We felt that a card-game design language would be fun to implement, practical for users, and clearly within the scope of our semester-long project.

We then figured out our project management. We designated a team leader to lead our project; we familiarized ourselves with the deadlines of each part of the project; we set out to learn OCaml; we also exchanged contact information and communicated to each other our weekly availabilities for meetings both in person and online. We decided that we would maintain detailed notes of our discussions, decisions, and documentation and share them with each other through email, Google Docs, and Dropbox.

4.1.2 Specification

In the coming weeks, we slowly worked out our idea in greater depth. Usually without regard to exactly what was needed in the most upcoming documentation, we brainstormed all possible considerations in implementing our language. We made sure to explore all facets of card-game design, at first extending our scope to all card games. We jotted down our ideas and discussed topics of disagreement. The notes would eventually be used in our formal documentation submissions.

As the weeks went on, we went further and further in detail in conceptualizing our idea. We went from end-user necessities to basic language components to complex syntax. By dividing up the specification, we were able to bring different perspectives together in regards to the design concept. By writing sample code, we were also better able to visualize each other's interpretations of the idea so that we could discuss discrepancies and standardize our design.

4.1.3 Development

Once we had finalized the outline of our idea, we proceeded to develop the compiler. We created a repository on Google Code, where we could all share our code with each other. We proceeded to divide the project up into logical pieces that we thought made sense in terms of getting the project done. We initially divided the project up into the scanner and parser, the compiler, and the interpreter, based on our original perceptions. As we got deeper into our development process, we became more flexible in what our individual tasks were. In deciding how to proceed, we put into perspective what needed to be done at the time based on our overall progress towards our overarching objectives.

When errors up came or when a fellow team member was confused, we assisted each other in moving him or her forward. We often brought up the problems to the team. This was especially the case for when we realized we had an unexpected design consideration. We discussed as a team, and sometimes with the professor or TA, to decide on a feasible solution. While most of the coding was done separately, given the nature of the task, we often stayed online together on Skype to raise issues or questions to each other. We also met several times in person when convenient, to collaborate more closely.

We developed our code using Eclipse with OcamIDE. SVN was used as a source control to commit files to Google code. For your reference, our repository is http://code.google.com/p/plt-descartes/source/browse/. We also used Makefile to automate code compilation and linkage.

4.1.4 Testing

In preparation for testing our compiler, we wrote two test game examples in our source language. We picked two common card games that we thought would be easy to implement, but we also picked them because we thought they would fully leverage the features of our language and really show the benefits of our language. We also had test cases which we used to slowly debug our compiler. These test cases were independently written, so were more useful in evaluating the accuracy and precision of our compiler.

Testing allowed us to evaluate our compiler with full-fledged programs written in our source code as well as test the little pieces of our program. This allowed us to match our compiler with realistic input and see visible output. We automated testing of the test suites through a Makefile.

4.2 Programming Style Guide

4.2.1 Readability

In coding our compiler, we did our best to keep the code simple and easy to navigate, both during the development process and after, as illustrated in our finished product. We made extensive use of commenting and gave intuitive names to our files, variables, and functions to ease the burden of reading the code for each other and for other evaluators of the compiler.

4.2.2 Modularization

From the start, we divided the compiler up into functional units that could be worked on and operated separately. Within those units, we also clearly defined our various functions. This was especially helpful in debugging, when we once again applied modularization in testing our code piece by piece and then adding working functions together.

4.2.3 Consistency

In order to be clear in our interpretation and efficient in our collaboration, we made efforts to comply with OCaml programming conventions. We also stayed consistent with each other's names for files, variables, and functions by coming up with intuitive names that made sense in relation to each other.

4.2.4 Spacing

We leveraged the design of the OCaml language by indenting our code when necessary to improve readability and to standardize our code with the conventions of OCaml. We also used whitespace and empty lines to clearly segment portions of our code for further clarity.

4.2.5 Documentation

Extending our philosophy in project management, we made sure to document clearly in our software development as well. We commented extensively within the code. We also wrote meaningful notes when committing as well as when updating the team and keeping record of on our individual progress.

4.3 Project Timeline / Project Log

9/18/2011	Brainstormed project ideas and decided on a card-game design language.
9/21/2011	Decided on a project name, designated a team leader, bounced around conceptual ideas for our project, identified required sections for the proposal, and finalized software development environment and source control
9/25/2011	Decided on fundamental keywords and syntax, wrote sample code, bounced around conceptual ideas for our project, and finalized project proposal
9/26/2011	Brought the project proposal to TA for feedback
10/9/2011	Discussed the relevant components and scope of our project, decided on syntax, and delegated sections for the Language Reference Manual
10/12/2011	Discussed the project proposal with TA for feedback
10/22/2011	Combined and reviewed the bulk of the Language Reference Manual
10/30/2011	Finalized the Language Reference Manual
11/16/2011	Decided on project milestones and delegated components among team members
11/29/2011	Created the lexer, parser, and AST and interpreter with sample functions
12/4/2011	Created a compiler and updated the lexer and AST
12/5/2011	Updated the interpreter with additional functions
12/8/2011	Resolved errors in the lexer
12/10/2011	Created a preliminary interpreter and resolved errors in the lexer, parser, and AST
12/11/2011	Resolved errors in the interpreter
12/13/2011	Integrated the symbol table to the interpreter
12/14/2011	Updated the compiler and interpreter
12/15/2011	Updated the compiler and interpreter
12/16/2011	Resolved errors in the AST
12/21/2011	Completed compiler, testing and final report

4.4 Roles & Responsibilities

- Eric – scanner.mll, parser.mly, ast.ml, descartes.ml, Makefile, testall.sh
- Susan – compile.ml, testing
- Jim – testing, perl libraries, documentation
- Jack – compiler, testing

4.5 Software Development Environment

We used OCaml as our primary language for coding our compiler. As our compiler's output is Perl code, we also needed to familiarize ourselves and write test cases in Perl. We also experimented with Java and C. We used Eclipse for most of our software development, with Google Code as our repository. We worked in Windows and Mac environments.

5 Architectural Design

5.1 Architecture Overview

The Descartes compiler consists of 3 main blocks: scanner, parser and code generator. The relationship between these components is demonstrated in figure 5.1. All these components are implemented in the OCaml language. The Descartes compiler takes a Descartes file as input (with the suffix .des) and outputs a perl file.

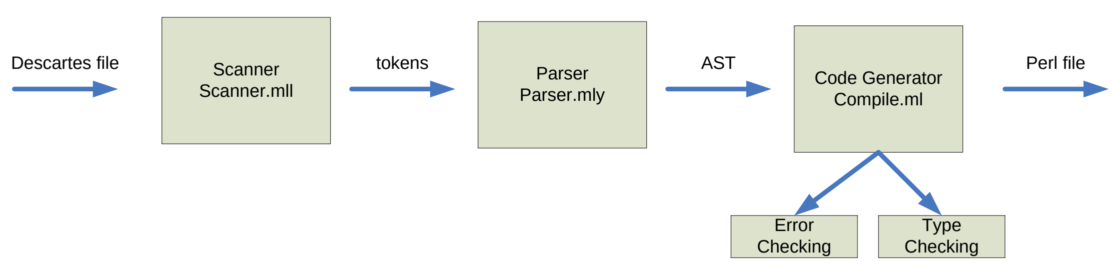

Figure 5.1: block diagram of Descartes compiler

5.2 Scanner

The scanner was implemented with Ocamllex in the scanner.mll. The scanner takes the input .des source file and generates tokens for the parser to process. This module was implemented by Eric Chao.

5.3 Parser

The parser was implemented in Ocamlyacc in the parser.mly. The parser takes in a sequence of tokens generated by the scanner through the program declaration and produces an abstract syntax tree defined in the ast.ml. The parser and ast was implemented by Eric Chao.

5.4 Code Generator

The code generator was implemented in OCaml in the compiler.ml. The main entry point to the code generator is the compile function that will take in the 2 lists for processing. The entry point to the code generator is the compile function where it takes 2 lists (global variables and function declarations) that are generated by the parser. The code generator performs error handling, type checking and perl code generation. Susan Fung implemented the type checking, error checking, symbol tables, generation of string representation of the expressions and statements to perl, and descartes specific functions such as print, println, scan and other basics. Xiaocheng Shi did some debugging of code generator issues and implemented descartes specific functions, such as draw, value, getName, getValue, deck, label, card and list. Xiaocheng Shi also implemented the initializing of variables, reordering code, added comp_func_main function and perl-specific headers.

6 Test Plan

6.1 Representative Programs

6.1.1 HighLow

Source

```
int main(){
	return play();
}

int play(){
	string currentCard;
```

```
string nextCard;
int currentValue;
int nextValue;
int gameScore;
int guess;
bool continue;

currentCard = draw();
currentValue = value(currentCard);

print("The first card is ");
print(currentCard);
print(" with value ");
println(currentValue);

while(deck() && continue)
{
      println("Do you think the next card will be lower (0) or higher (1)?");
      guess = scan();

      while (guess*guess != guess)
      {
            print("Please respond with 0 or 1.");
            guess = scan();
      }

      nextCard = draw();
      nextValue = value(nextCard);
      print("The next card is ");
      print(nextCard);
      print(" with value ");
      println(nextValue);

      if(nextValue == currentValue)
      {
            println("The values are the same. You lose.");
            continue = false;
      }
      else if (nextValue > currentValue)
      {
            if (guess == 1)
            {
                  println("Your prediction was correct.");
                  gameScore++;
                  currentCard = nextCard;
                  currentValue = nextValue;
                  print("The card is ");
                  print(currentCard);
                  print(" with value ");
                  println(currentValue);
            }
            else
            {
               println("Your prediction was incorrect.");
```

```
                    continue = false;
                }
            }
            else
            {
                if (guess == 0)
                {
                    println("Your prediction was correct.");
                    gameScore++;
                    currentCard = nextCard;
                    currentValue = nextValue;
                    print("The card is ");
                    print(currentCard);
                    print(" with value ");
                    println(currentValue);
                }
                else
                {
                    println("Your prediction was incorrect.");
                    continue = false;
                }
            }
        }
        println("The game is over.");
        print("You made ");
        print(gameScore);
        println(" correct predictions.");
        return gameScore;
}
```

Perl

```
#!/usr/bin/perl

#use warnings;
use Card;
use DefaultDeck;

$deck = new DefaultDeck();

sub main {
        return play();
}

sub value {
        $name = $_[0];
        $value = substr $name, 1;
        if ($value eq "A")
        {
                return 1;
        }
        else
        {
                if ($value eq "J")
                {
                        return 11;
```

```
            }
            else
            {
                    if ($value eq "Q")
                    {
                            return 12;
                    }
                    else
                    {
                            if ($value eq "K")
                            {
                                    return 13;
                            }
                            else
                            {
                                    return $value;
                            }
                    }
            }
    }
}

sub play {
        $currentCard = "";
        $nextCard = "";
        $currentValue = 0;
        $nextValue = 0;
        $gameScore = 0;
        $guess = 0;

        $currentCard = $deck->draw()->getname();
        $currentValue = value($currentCard);

        print "The first card is ";
        print $currentCard;
        print " with value ";
        print $currentValue."\n";

        while ($deck->size()) { # Loop ends when there are no more cards or the user is wrong.
                print "Do you think the next card will be lower (0) or higher (1)?\n";
                $guess = <STDIN>;

                while ($guess*$guess != $guess)
                {
                        print "Please respond with 0 or 1.\n";
                        $guess = <STDIN>;
                }

                $nextCard = $deck->draw()->getname();
                $nextValue = value($nextCard);
                print "The next card is ";
                print $nextCard;
                print " with value ";
                print $nextValue."\n";

                if ($nextValue == $currentValue)
                {
                        print "The values are the same. You lose."."\n";
                        last;
                }
                else {
                        if ($nextValue > $currentValue)
```

```
                        {
                                if ($guess == 1)
                                {
                                        print "Your prediction was correct."."\n";
                                        $gameScore++;
                                }
                                else
                                {
                                        print "Your prediction was incorrect."."\n";
                                        last;
                                }
                        }
                        else
                        {
                                if ($guess == 0)
                                {
                                        print "Your prediction was correct."."\n";
                                        $gameScore++;
                                }
                                else
                                {
                                        print "Your prediction was incorrect."."\n";
                                        last;
                                }
                        }
                }
                $currentCard = $nextCard;
                $currentValue = $nextValue;
                print "The card is ";
                print $currentCard;
                print " with value ";
                print $currentValue."\n";
        }
        print "The game is over."."\n";
        print "You made ";
        print $gameScore;
        print " correct predictions."."\n";
        return $gameScore;
}
print main()."\n";
```

Output

The results of the game are:
The first card is S2 with value 2
Do you think the next card will be lower (0) or higher (1)?
1
The next card is HQ with value 12
Your prediction was correct.
The card is HQ with value 12
Do you think the next card will be lower (0) or higher (1)?
0
The next card is H3 with value 3

Your prediction was correct.
The card is H3 with value 3
Do you think the next card will be lower (0) or higher (1)?
1
The next card is S9 with value 9
Your prediction was correct.
The card is S9 with value 9
Do you think the next card will be lower (0) or higher (1)?
0
The next card is D6 with value 6
Your prediction was correct.
The card is D6 with value 6
Do you think the next card will be lower (0) or higher (1)?
0
The next card is DQ with value 12
Your prediction was incorrect.
The game is over.
You made 4 correct predictions.
4

6.1.2 Blackjack

Source

```
int main(){
	int wins;
	int losses;
	int continue;
	wins = 0;
	losses = 0;
	continue = 1;

	println("");
	while (continue == 1) {
		if (play()) {
			wins++;
			println("You win.");
		} else {
			losses++;
			println("You lose.");
		}
		print("You have won ");
		print(wins);
		print(" game(s) and lost ");
		print(losses);
		println(" game(s).");
```

```
                println("Would you like to continue playing (1) or stop (0)?");
                continue = scan();
                while (continue*continue != continue) {
                        println("Please respond with 0 or 1.");
                        continue = scan();
                }
                println("");
        }
        print("Your final score is ");
        print(wins);
        print(" win(s) and ");
        print(losses);
        println(" loss(es).");
}

int play(){
        string myfirst;
        int myfirstvalue;
        string mysecond;
        int mysecondvalue;
        string dealerfirst;
        int dealerfirstvalue;
        string dealersecond;
        int dealersecondvalue;
        int myscore;
        int dealerscore;
        int myaces;
        int dealeraces;
        string mynext;
        int mynextvalue;
        string dealernext;
        int dealernextvalue;
        int myinput;
        int dealerinput;
        int breaker;

        myfirst = draw();
        myfirstvalue = value(myfirst);
        if (myfirstvalue == 1) {
                myfirstvalue = 11;
                myaces++;
        } else if (myfirstvalue == 11 || myfirstvalue == 12 || myfirstvalue == 13) {
                myfirstvalue = 10;
        }
        print("You drew ");
        print(myfirst);
        print(" with value ");
        print(myfirstvalue);
        println(".");

        dealerfirst = draw();
        dealerfirstvalue = value(dealerfirst);
        if (dealerfirstvalue == 1) {
                dealerfirstvalue = 11;
                dealeraces++;
        } else if (dealerfirstvalue == 11 || dealerfirstvalue == 12 || dealerfirstvalue == 13)
{
                dealerfirstvalue = 10;
        }
        println("The dealer drew a card.");

        mysecond = draw();
```

```
        mysecondvalue = value(mysecond);
        if (mysecondvalue == 1) {
               mysecondvalue = 11;
               myaces++;
        } else if (mysecondvalue == 11 || mysecondvalue == 12 || mysecondvalue == 13) {
               mysecondvalue = 10;
        }
        print("You drew ");
        print(mysecond);
        print(" with value ");
        print(mysecondvalue);
        println(".");


        dealersecond = draw();
        dealersecondvalue = value(dealersecond);
        if (dealersecondvalue == 1) {
               dealersecondvalue = 11;
               dealeraces++;
        } else if (dealersecondvalue == 11 || dealersecondvalue == 12 || dealersecondvalue ==
13) {
               dealersecondvalue = 10;
        }
        print("The dealer drew ");
        print(dealersecond);
        print(" with value ");
        print(dealersecondvalue);
        println(".");

        breaker = 0;
        myscore = myfirstvalue + mysecondvalue;
        myinput = 1;
        while (myinput == 1) {
               if (myscore > 21) {
                      if (myaces > 0) {
                             myaces--;
                             myscore -= 10;
                      } else {
        print("Your current score is ");
        print(myscore);
        println(".");
        println("You have gone over 21.");
        println("");
        breaker = 1;
        myinput = 0;
                      }
               }

        if (breaker == 0) {
        print("Your current score is ");
        print(myscore);
        println(".");

               println("What would you like to do? Hit (1) or Stay (0)?");
               myinput = scan();
               while (myinput*myinput != myinput) {
                      println("Please respond with 0 or 1.");
                      myinput = scan();
               }
               println("");
               if (myinput == 1) {
                      mynext = draw();
```

```
                    mynextvalue = value(mynext);
                    if (mynextvalue == 1) {
                          myaces++;
                          mynextvalue = 11;
                    } else if (mynextvalue == 11 || mynextvalue == 12 || mynextvalue == 13)
{
                          mynextvalue = 10;
                    }
                    print("You drew ");
        print(mynext);
        print(" with value ");
        print(mynextvalue);
        println(".");
                    myscore += mynextvalue;
              }
              }
        }

        breaker = 0;
        dealerscore = dealerfirstvalue + dealersecondvalue;
        dealerinput = 1;
        while (dealerinput == 1) {

              if (dealerscore > 21) {
                    if (dealeraces > 0) {
                          dealeraces--;
                          dealerscore -= 10;
                    } else {
                          breaker = 1;
                          dealerinput = 0;
                    }
              }

              if (breaker == 0) {
              if (dealerscore < 17) {
                    dealerinput = 1;
              } else {
                    dealerinput = 0;
              }

              if (dealerinput == 1) {
                    dealernext = draw();
                    dealernextvalue = value(dealernext);
                    if (dealernextvalue == 1) {
                          dealeraces++;
                          dealernextvalue = 11;
                    } else if (dealernextvalue == 11 || dealernextvalue == 12 ||
dealernextvalue == 13) {
                          dealernextvalue = 10;
                    }
                    dealerscore += dealernextvalue;
              }
              }
        }

        print("Your score is ");
        print(myscore);
        print(" and the dealer's score is ");
        print(dealerscore);
        println(".");
        if (myscore > 21 || (dealerscore <= 21 && dealerscore >= myscore)) {
              return 0;
```

```
        } else {
               return 1;
        }
}
```

Perl

```
#!/usr/local/bin/perl
use warnings;
use Card;
use DefaultDeck;

$deck = new DefaultDeck();
sub main()
{
$wins = 0;

$losses = 0;

$continue = 0;
$wins = 0;

$losses = 0;

$continue = 1;

print ""."\n";

while (($continue==1)) {
if (play()){{
$wins = ($wins+1);
print "You win."."\n";
}
}
 else {{
$losses = ($losses+1);
print "You lose."."\n";
}
}
print "You have won ";
print $wins;
print " game(s) and lost ";
print $losses;
print " game(s)."."\n";
print "Would you like to continue playing (1) or stop (0)?"."\n";
$continue = <STDIN>;
while ((($continue*$continue)!=$continue)) {
print "Please respond with 0 or 1."."\n";
$continue = <STDIN>;
}

print ""."\n";
}


print "Your final score is ";

print $wins;

print " win(s) and ";
```

```
print $losses;

print " loss(es)."."\n";
}
sub value {
$name = $_[0];
$value = substr $name, 1;
if ($value eq "A")
{
return 1;
}
else
{if ($value eq "J")
{
return 11;
}
else
{
if ($value eq "Q")
{
return 12;
}
else
{
if ($value eq "K")
{
return 13;
}
else
{
return $value;
}
}
}
}
}

sub play()
{
$myfirst = "";

$myfirstvalue = 0;

$mysecond = "";

$mysecondvalue = 0;

$dealerfirst = "";

$dealerfirstvalue = 0;

$dealersecond = "";

$dealersecondvalue = 0;

$myscore = 0;

$dealerscore = 0;

$myaces = 0;
```

```
$dealeraces = 0;

$mynext = "";

$mynextvalue = 0;

$dealernext = "";

$dealernextvalue = 0;

$myinput = 0;

$dealerinput = 0;

$breaker = 0;
$myfirst = $deck->draw()->getname();

$myfirstvalue = value($myfirst);

if (($myfirstvalue==1)){{
$myfirstvalue = 11;
$myaces = ($myaces+1);
}
}
 else {if ((($myfirstvalue==11)||($myfirstvalue==12))||($myfirstvalue==13)))
 {{
$myfirstvalue = 10;
}
}
}

print "You drew ";

print $myfirst;

print " with value ";

print $myfirstvalue;

print "."."\n";

$dealerfirst = $deck->draw()->getname();

$dealerfirstvalue = value($dealerfirst);

if (($dealerfirstvalue==1)){{
$dealerfirstvalue = 11;
$dealeraces = ($dealeraces+1);
}
}
 else {if ((($dealerfirstvalue==11)||($dealerfirstvalue==12))||($dealerfirstvalue==13)))
 {{
$dealerfirstvalue = 10;
}
}
}

print "The dealer drew a card."."\n";

$mysecond = $deck->draw()->getname();

$mysecondvalue = value($mysecond);
```

```
if (($mysecondvalue==1)){{
$mysecondvalue = 11;
$myaces = ($myaces+1);
}
}
 else {if (((($mysecondvalue==11)||($mysecondvalue==12))||($mysecondvalue==13)))
 {{
$mysecondvalue = 10;
}
}
}

print "You drew ";

print $mysecond;

print " with value ";

print $mysecondvalue;

print "."."\n";

$dealersecond = $deck->draw()->getname();

$dealersecondvalue = value($dealersecond);

if (($dealersecondvalue==1)){{
$dealersecondvalue = 11;
$dealeraces = ($dealeraces+1);
}
}
 else {if (((($dealersecondvalue==11)||($dealersecondvalue==12))||($dealersecondvalue==13)))
 {{
$dealersecondvalue = 10;
}
}
}

print "The dealer drew ";

print $dealersecond;

print " with value ";

print $dealersecondvalue;

print "."."\n";

$breaker = 0;

$myscore = ($myfirstvalue+$mysecondvalue);

$myinput = 1;

while (($myinput==1)) {
if (($myscore>21))
 {{
if (($myaces>0)){{
$myaces = ($myaces-1);
$myscore = ($myscore-10);
}
```

```
}
 else {{
print "Your current score is ";
print $myscore;
print "."."\n";
print "You have gone over 21."."\n";
print ""."\n";
$breaker = 1;
$myinput = 0;
}
}
}
}
if (($breaker==0))
 {{
print "Your current score is ";
print $myscore;
print "."."\n";
print "What would you like to do? Hit (1) or Stay (0)?"."\n";
$myinput = <STDIN>;
while ((($myinput*$myinput)!=$myinput)) {
print "Please respond with 0 or 1."."\n";
$myinput = <STDIN>;
}

print ""."\n";
if (($myinput==1))
 {{
$mynext = $deck->draw()->getname();
$mynextvalue = value($mynext);
if (($mynextvalue==1)){{
$myaces = ($myaces+1);
$mynextvalue = 11;
}
}
 else {if (((($mynextvalue==11)||($mynextvalue==12))||($mynextvalue==13)))
 {{
$mynextvalue = 10;
}
}
}
print "You drew ";
print $mynext;
print " with value ";
print $mynextvalue;
print "."."\n";
$myscore = ($myscore+$mynextvalue);
}
}
}
}
}


$breaker = 0;

$dealerscore = ($dealerfirstvalue+$dealersecondvalue);

$dealerinput = 1;

while (($dealerinput==1)) {
if (($dealerscore>21))
```

```
 {{
if (($dealeraces>0)){{
$dealeraces = ($dealeraces-1);
$dealerscore = ($dealerscore-10);
}
}
 else {{
$breaker = 1;
$dealerinput = 0;
}
}
}
}
if (($breaker==0))
 {{
if (($dealerscore<17)){{
$dealerinput = 1;
}
}
 else {{
$dealerinput = 0;
}
}
if (($dealerinput==1))
 {{
$dealernext = $deck->draw()->getname();
$dealernextvalue = value($dealernext);
if (($dealernextvalue==1)){{
$dealeraces = ($dealeraces+1);
$dealernextvalue = 11;
}
}
 else {if (((($dealernextvalue==11)||($dealernextvalue==12))||($dealernextvalue==13)))
 {{
$dealernextvalue = 10;
}
}
}
$dealerscore = ($dealerscore+$dealernextvalue);
}
}
}
}
}


print "Your score is ";

print $myscore;

print " and the dealer's score is ";

print $dealerscore;

print "."."\n";

if ((($myscore>21)||(($dealerscore<=21)&&($dealerscore>=$myscore)))){{
return 0;
}
}
 else {{
return 1;
```

```
}
}
}
print main()."\n";
```

Output

You drew D3 with value 3.
The dealer drew a card.
You drew D4 with value 4.
The dealer drew S7 with value 7.
Your current score is 7.
What would you like to do? Hit (1) or Stay (0)?
1

You drew H4 with value 4.
Your current score is 11.
What would you like to do? Hit (1) or Stay (0)?
1

You drew C7 with value 7.
Your current score is 18.
What would you like to do? Hit (1) or Stay (0)?
0

Your score is 18 and the dealer's score is 18.
You lose.
You have won 0 game(s) and lost 1 game(s).
Would you like to continue playing (1) or stop (0)?
1

You drew H2 with value 2.
The dealer drew a card.
You drew S4 with value 4.
The dealer drew S10 with value 10.
Your current score is 6.
What would you like to do? Hit (1) or Stay (0)?
1

You drew H5 with value 5.
Your current score is 11.
What would you like to do? Hit (1) or Stay (0)?
1

You drew S2 with value 2.
Your current score is 13.
What would you like to do? Hit (1) or Stay (0)?
1

You drew C10 with value 10.
Your current score is 23.
You have gone over 21.

Your score is 23 and the dealer's score is 19.
You lose.
You have won 0 game(s) and lost 2 game(s).
Would you like to continue playing (1) or stop (0)?
0

Your final score is 0 win(s) and 2 loss(es).
1

6.2 Test Suites

6.3 Test Case Choices

We performed both functional and error/exception-handling tests, deciding on the test cases based on the individual functionalities and errors/exceptions relevant to DesCartes.

For the functional tests, we wrote test cases for individual functions of our program. We converted all of the MicroC test cases into DesCartes to be tested, to ensure that the basic functions of our program were implemented correctly. We also added additional test cases for more advanced functions as well as functions more specific to DesCartes, such as those dealing with the default deck.

Regarding the error/exception-handling test cases, we considered the errors and exceptions that were most probable to occur given our functions. We wrote test cases with this in mind, with some intentionally failing. With the error/exception-handling test cases, we wanted to ensure that errors won't occur with our program.

6.4 Automation

We created a Makefile that calls our test script, which reads in all of our test cases, runs them through the lexer, parser, AST, and compiler, outputs the equivalent Perl code, and runs the code, printing the final output out. All the test cases start with "test-" to allow for easy addition of new test cases.

6.5 Division of Labor

- Eric – test-arith1.des, test-arith2.des, test-fib.des, test-for1.des, test-func1.des, test-gcd.des, test-global1.des, test-hello.des, test-if1.des, test-if2.des, test-if3.des, test-if4.des, test-ops1.des, test-var1.des, test-while1.des; testall.sh
- Susan – test-double-main.des, test-dup-func-name.des, test-error-global-vars.des, test-error-main.des, test-error-main-args.des, test-error-main-local-vars.des, test-error-sub-args.des, test-error-sub-local.des, test-func-arg-ck.des, test-global-var-keyword.des, test-main-arg-keyword.des, test-main-local-keyword.des, test-nomain.des, test-return-type-cks.des, test-stmt-type-ck.des, test-string-of-expr-type-ck.des, test-sub-arg-keyword.des, test-sub-local-keyword.des, test-break.des, test-int.des, test-int-call.des, test-print-func.des and test-string.des
- Jim – test-callinginpassing.des, test-decktest.des, test-infiniteloop.des, example games
- Jack – test-argPass1.des, test-func3.des, test-for.des, test-list.des; test-getValue-getName.des; example games

7 Lessons Learned

7.1 Eric Chao (ehc2129)

It is very important to sit down with the team and TA/professors to go over your Language Reference Manual and make sure you have a very clear idea of how your language works and how you plan on implementing everything. In the beginning we had a lot of bold ideas that quickly were discarded during the implementation phase due to time constraints and feasibility because our scope was too big. Also, many portions of our language were not defined properly in the LRM prior to development.

Also, we divided the project up in such a way that each person was not dependent on another – or at least very little, so that we could all work on each section without delays. I learned that it is critical to hold each team member accountable for their portion of the project after you do the initial assignments and thoroughly test your own portions because we had committed code in the repository that caused some last minute scrambling and panic because it broke other working parts. Even though we made a schedule in the beginning, portions of this project were delayed time and time again leading to a late finish. Future teams should assign a backup to each person's section and if a person misses a deadline, the backup takes over all responsibilities. Also, it is important to clearly divide up who is doing what portion to avoid 2 members thinking each other is doing something, but in reality, no one is.

Finally, I learned that while at times O'Caml is very difficult to learn, once you understand what the compile errors are, you can quickly debug your program.

7.2 Susan Fung (sgf2110)

Having the architecture defined early is very pertinent to the project. Figure out how you think it should be and run it by the professor before starting. We were all assuming that an interpreter and compiler were both necessary and couldn't figure out the differences. In the end, we just decided that it is really one thing that translates the abstract syntax tree to another code representation. In our case, it is Perl. Also, team work, communication and collaboration are very important. Once you define what the parts are, divide and conquer!

7.3 James Huang (jzh2103)

Start early! There is no other lesson that comes close in importance. There is a reason why this lesson is stressed so much. We just never predicted how much work was needed for the project and how much work for other commitments would pile up. In addition, I learned that designing a compiler is much different than developing a compiler. We had planned out how to make the ultimate card-game design language, only to realize how hard it would be to implement everything. Finally, I gained insight into how compilers work and how complex they are.

7.4 Xiaocheng Shi (xs2139)

For a semester-long project on the scope of something like this, it is important to have a good plan but to also be flexible. As the project unfolds, it is important to reevaluate and adjust your plan accordingly. Also, I realized the importance of group communication, of letting people know your circumstances whether it be illness or exams that might make it difficult for you to work during that period of time, and to also be reachable by other group members. The other important thing is people making to and staying for project meetings. Periodic stalling circumstances arose when one group member needed the code or help of another. It is also quite clear that you can only learn a computer science language through code. Over the last month or so of heavy coding, I realized my mastery of O'Caml increased greatly.

In any group, people need to find the right fit so that everyone is happy and can contribute and we did find that fit for everyone. In the end, it is important to be open to other members' suggestions and be open to compromise in the event of disagreements. In some ways, the group dynamic and group relations are the most crucial because they are what drive good work and willingness to work.

8 Appendix

8.1 ast.ml

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

type expr = (* Expressions *)
      Id of string (* foo *)
      | IntLiteral of int (* 42 *)
      | BoolLiteral of bool (* true *)
      | Binop of expr * op * expr (* a + b *)
      | Assign of expr * expr (* foo = 42 *)
```

```
      | Call of string * expr list (* foo(1, 25 *)
      | Noexpr (* for(;;) *)
      | Not of expr (* !x *)
      | StringLiteral of string (* "hello" *)
      | CardLiteral of string (* S2 *)
      | ListLiteral of expr list (* [1,2,3,4] *)
      | LabelLiteral of string (* label A - for break fix *)

type stmt = (* Statements *)
      Expr of expr (* foo = bar + 3 *)
      | Return of expr (* return 42 *)
      | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
      | Block of stmt list (* {...} *)
      | For of expr * expr * expr * stmt (* for (i=0;i<10;i++) { ... } *)
      | While of expr * stmt (* while (i<10) { i = i+1 } *)

type var_decl = {
      varname : string; (* Name of the variable *)
      vartype : string; (* Name of variable type *)
}

type func_decl = {
            fname : string; (* Name of the function *)
            freturn : string; (* Name of return type *)
            formals : var_decl list; (* Formal argument names *)
            locals : var_decl list; (* Locally defined variables *)
            body : stmt list; (* Statement List *)
}

type program = var_decl list * func_decl list (* global vars, funcs *)

(* Methods below are to print out the AST for testing/debugging reasons *)

(* method for printing expressions *)
let rec string_of_expr = function
    IntLiteral(l) -> string_of_int l
      | BoolLiteral(b) -> string_of_bool b
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
                  Add -> "+"
                  | Sub -> "-"
                  | Mult -> "*"
                  | Div -> "/"
      | Equal -> "=="
                  | Neq -> "!="
                  | And -> "&&"
                  | Or -> "||"
                  | Mod -> "%"
      | Less -> "<"
                  | Leq -> "<="
                  | Greater -> ">"
                  | Geq -> ">=") ^ " " ^ string_of_expr e2
  | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
```

```
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
      | StringLiteral(s) -> s
      | CardLiteral(c) -> c
      | ListLiteral(p) -> "[" ^ String.concat "," (List.map string_of_expr p) ^ "]"
      | LabelLiteral(a) -> a
      | Not(n) -> "!" ^ string_of_expr n

(* method for printing statements *)
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

(* method for printing variable decls *)
let string_of_var_decl var_decl =
      "varname: " ^ var_decl.varname ^ "\nvartype: " ^ var_decl.vartype ^ "\n"

(* method for printing function decls *)
let string_of_fdecl fdecl =
      "\nfreturn: " ^ fdecl.freturn ^ "\nfname: "  ^ fdecl.fname ^  "(" ^
      String.concat " " (List.map string_of_var_decl fdecl.formals) ^ ")\n{\n" ^
  String.concat " " (List.map string_of_var_decl fdecl.locals) ^ " "^
  String.concat " " (List.map string_of_stmt fdecl.body) ^
  "}"

(* method for printing program - list of var_decl and func_decl *)
let string_of_program (vars, funcs) =
      "VARS: \n" ^ String.concat " " (List.map string_of_var_decl vars) ^ "
\n\nFUNCTIONS: " ^
      String.concat "\n" (List.map string_of_fdecl funcs)
```

8.2 parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMI COMMA DOT QUOTE
%token PLUS MINUS TIMES DIVIDE MOD PLUSPLUS MINUSMINUS
%token PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ MODEQ
%token EQ NEQ LT LEQ GT GEQ AND NOT OR ASSIGN
%token IF ELSE ELSEIF FOR WHILE RETURN

%token INT VOID BOOL STRING CARD LIST LABEL
%token <int> INTLITERAL
%token <bool> BOOLEANLITERAL
%token <string> STRINGLITERAL
```

```
%token <string> CARDLITERAL
%token <string> LABELLITERAL
%token <string> ID
%token <string> TYPE
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc LPAREN
%right ASSIGN
%left PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ MODEQ
%left AND OR
%left EQ NEQ LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left PLUSPLUS MINUSMINUS
%right NOT
%start program
%type <Ast.program> program

%%

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

fdecl:
      TYPE ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
      { { fname = $2;
            freturn = $1;
            formals = $4;
            locals = List.rev $7;
            body = List.rev $8 } }

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

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

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

vdecl:
      TYPE ID SEMI { { varname = $2; vartype = $1 } }

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

stmt:
    expr SEMI { Expr($1) }
```

```
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt  { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
  ID                    { Id($1) }
      | INTLITERAL          { IntLiteral($1) }
      | BOOLEANLITERAL      { BoolLiteral($1) }
      | STRINGLITERAL       { StringLiteral($1) }
      | CARDLITERAL         { CardLiteral($1) }
      | LABELLITERAL        { LabelLiteral($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 MOD    expr    { Binop($1, Mod,       $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 ASSIGN expr    { Assign($1,             $3) }
      | NOT    expr         { Not($2) }
      | expr PLUSPLUS       { Assign($1, Binop($1, Add, IntLiteral(1))) }
      | expr MINUSMINUS     { Assign($1, Binop($1, Sub, IntLiteral(1))) }
      | expr PLUSEQ expr    { Assign($1, Binop($1, Add, $3)) }
      | expr MINUSEQ expr   { Assign($1, Binop($1, Sub, $3)) }
      | expr TIMESEQ expr   { Assign($1, Binop($1, Mult, $3)) }
      | expr DIVIDEEQ expr  { Assign($1, Binop($1, Div, $3)) }
      | expr MODEQ expr     { Assign($1, Binop($1, Mod, $3)) }
  | LPAREN expr RPAREN { $2 }
      | QUOTE expr QUOTE    { $2 }
      | LBRACKET list_opt RBRACKET { ListLiteral($2) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }


list_opt:
      /* nothing */             { [] }
      | list_list               { List.rev $1 }

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

actuals_opt:
```

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

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

8.3 scanner.mll

```
{ open Parser } (* Gets the token types *)

rule token = parse
      [ ' ' '\t' '\r' '\n'] { token lexbuf }    (* Whitespace *)
      | "/*"     { comment lexbuf } (* Comments *)
      | '('      { LPAREN }
      | ')'      { RPAREN }
      | '{'      { LBRACE }
      | '}'      { RBRACE }
      | ';'      { SEMI }
      | ','      { COMMA }
      | '.'      { DOT }
      | '+'      { PLUS }
      | '-'      { MINUS }
      | '*'      { TIMES }
      | '/'      { DIVIDE }
      | '%'      { MOD }
      | "+="     { PLUSEQ }
      | "-="     { MINUSEQ }
      | "*="     { TIMESEQ }
      | "/="     { DIVIDEEQ }
      | "%="     { MODEQ }
      | '='      { ASSIGN }
      | '!'      { NOT }
      | "++"     { PLUSPLUS }
      | "--"     { MINUSMINUS }
      | "=="     { EQ }
      | "!="     { NEQ }
      | '<'      { LT }
      | "<="     { LEQ }
      | ">"      { GT }
      | ">="     { GEQ }
      | "&&"     { AND }
      | "||"     { OR }
      | "if"     { IF } (* keywords *)
      | "else"   { ELSE }
      | "for"    { FOR }
      | "while"  { WHILE }
      | "return" { RETURN }
      | "int"    { TYPE("int") }
      | "void"   { TYPE("void") }
      | "bool"   { TYPE("bool") }
      | "string" { TYPE("string") }
      | "list"   { TYPE("list" ) }
      | "card"   { TYPE("card") }
      | "label"  { TYPE("label" ) }
```

```
      | [ 'S' 'H' 'D' 'C'] ("A" | "K" | "Q" | "J" | ['2' - '9']) as lxm {
CARDLITERAL(lxm) } (* cards *)
      | "true"|"false" as boollit { BOOLEANLITERAL(bool_of_string boollit) } (*
booleans *)
      | '\"' [^ '\"' '\r' '\n']* '\"' as lxm { STRINGLITERAL(lxm) } (* strings *)
      | eof { EOF } (* End of file *)
      | ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) } (* integers *)
      | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID("$" ^ lxm) }

      and comment = parse
      "*/" { token lexbuf } (* End of Comment *)
      | _ { comment lexbuf } (* Eat everything else *)
```

8.4 compile.ml

```
open Ast

open Printf

open Map

module NameMap =
  Map.Make
    (struct type t = string

             let compare x y = Pervasives.compare x y
                end)

exception ReturnException of int * int NameMap.t

(* These are the symbol tables *)
(* global symbol table [var name, var type]*)
let globals = NameMap.empty

(* local symbol table [var name, var type]*)
let locals = NameMap.empty

(* function declartion symbol table [func name, func decl]*)
let func_decls = NameMap.empty

(* returns a formatted name without $ *)
let get_name name = String.sub name 1 ((String.length name) - 1)

(* is the name given a keyword? *)
let is_keyword name =
  match name with
  | "print" -> true
  | "int" -> true
  | "string" -> true
  | "if" -> true
  | "else" -> true
  | "return" -> true
  | "for" -> true
  | "break" -> true
  | "while" -> true
```

```
  | "true" -> true
  | "false" -> true
  | "goto" -> true
  | "bool" -> true
  | "value" -> true
  | "deck" -> true
      | "renew" -> true

      | "$getList" -> "element"
  | "$read" -> "int"
  | "$readStr" -> "string"
  | "$readCard" -> "card"

  | "getName" -> true
  | "getValue" -> true
  | "getColor" -> true
  | "getSuit" -> true

      | "push" -> true
      | "pushInt" -> true
      | "pushStr" -> true

      | "shift" -> true
      | "shiftInt" -> true
      | "shiftStr" -> true

      | "pop" -> true
      | "popInt" -> true
      | "popStr" -> true

      | "unshift" -> true
      | "unshiftInt" -> true
      | "unshiftStr" -> true


  | _ -> false

(* Get datatype of value*)
let get_data_type locals globals func_decls =
  function
  | IntLiteral i -> "int"
  | BoolLiteral v -> "int"
  | Id v ->
      if NameMap.mem v globals
      then NameMap.find v globals
      else
        if NameMap.mem v locals
        then NameMap.find v locals
        else raise (Failure ("undeclared identifier " ^ v))
  | Not v -> ""
  | Binop (e1, op, e2) -> "int"
  | Assign (varName, e) -> "assign"
  | Call ("$print", e) -> "print"
  | Call ("$println", e) -> "println"
  | Call ("$scan", e) -> "scan"
```

```
  | Call ("$deck", e) -> "int"
  | Call ("$draw", e) -> "string"
      | Call ("$shuffleDeck", e) -> "shuffle"
      | Call ("$printDeck", e) -> "print"
      | Call ("$renewDeck", e) -> "renew"

      | Call ("$label", e) -> "label"
  | Call ("$break", e) -> "break"

      | Call ("$getName", e) -> "string"
  | Call ("$getValue", e) -> "int"
  | Call ("$getColor", e) -> "string"
  | Call ("$getSuit", e) -> "string"
  | Call ("$value", e) -> "int"

      | Call ("$getList", e) -> "element"
  | Call ("$read", e) -> "int"
  | Call ("$readStr", e) -> "string"
  | Call ("$readCard", e) -> "card"

      | Call ("$push", e) -> "card"
      | Call ("$pushInt", e) -> "int"
      | Call ("$pushStr", e) -> "string"

      | Call ("$unshift", e) -> "card"
      | Call ("$unshiftInt", e) -> "int"
      | Call ("$unshiftStr", e) -> "string"

      | Call ("$pop", e) -> "card"
      | Call ("$popInt", e) -> "int"
      | Call ("$popStr", e) -> "string"

      | Call ("$shift", e) -> "card"
      | Call ("$shiftInt", e) -> "int"
      | Call ("$shiftStr", e) -> "string"

  | Call (f, actuals) -> let func = NameMap.find f func_decls in func.freturn
  | Noexpr -> ""
  | StringLiteral e -> "string"
  | LabelLiteral e -> "label"
  | ListLiteral e -> "list"
  | CardLiteral e -> "card"

let init_var =
  function
  | "string" -> "\"\""
  | "int" -> "0"
  | "bool" -> "1"
  | "card" -> "new Card()"
  | "list" -> "()"
  | x -> ""

let split_char sep str =
  let string_index_from i =
    try Some (String.index_from str i sep) with | Not_found -> None in
```

```
    let rec aux i acc =
      match string_index_from i with
      | Some i' ->
          let w = String.sub str i (i' - i) in aux (succ i') (w :: acc)
      | None ->
          let w = String.sub str i ((String.length str) - i)
          in List.rev (w :: acc)
    in aux 0 []

  (* Generates a string representation of variable decl *)
  let string_of_var_decl var_decl =
    if var_decl.vartype = "list"
    then
      "@" ^
        ((get_name var_decl.varname) ^
           (" = " ^ ((init_var var_decl.vartype) ^ ";\n")))
    else var_decl.varname ^ (" = " ^ ((init_var var_decl.vartype) ^ ";\n"))

  (* Generates a string representation of an expression*)
  let rec string_of_expr locals globals func_decls =
    function
    | IntLiteral i -> string_of_int i
    | StringLiteral e -> e
    | Noexpr -> ""
    | Id var -> (* ID *)
        if NameMap.mem var locals
        then var
        else
          if NameMap.mem var globals
          then var
          else raise (Failure ("undeclared identifier " ^ var))
    | Not v -> "!" ^ (string_of_expr locals globals func_decls v)
    | BoolLiteral v -> (match v with | true -> "1" | false -> "0")
    | Binop (e1, op, e2) ->
        let e1_data_type = get_data_type locals globals func_decls e1 in
        let e2_data_type = get_data_type locals globals func_decls e2 in
        let v1 = string_of_expr locals globals func_decls e1 in
        let v2 = string_of_expr locals globals func_decls e2
        in
          (* type checking to make sure they are ints *)
          if (e1_data_type <> "int") && (e1_data_type <> "bool")
          then raise (Failure (v1 ^ " has to be of type int "))
          else
            if (e2_data_type <> "int") && (e2_data_type <> "bool")
            then raise (Failure (v2 ^ " has to be of type int "))
            else
              (let v3 =
                 match op with
                 | Add -> v1 ^ ("+" ^ v2)
                 | Sub -> v1 ^ ("-" ^ v2)
                 | Mult -> v1 ^ ("*" ^ v2)
                 | Div -> v1 ^ ("/" ^ v2)
                 | Equal -> v1 ^ ("==" ^ v2)
                 | Neq -> v1 ^ ("!=" ^ v2)
                 | Less -> v1 ^ ("<" ^ v2)
                 | Leq -> v1 ^ ("<=" ^ v2)
```

```
              | Greater -> v1 ^ (">" ^ v2)
              | Geq -> v1 ^ (">=" ^ v2)
              | Mod -> v1 ^ ("%" ^ v2)
              | Or -> v1 ^ ("||" ^ v2)
              | And -> v1 ^ ("&&" ^ v2)
            in "(" ^ (v3 ^ ")"))
| Assign (varName, e) ->
    let e1_data_type = get_data_type locals globals func_decls varName in
    let e2_data_type = get_data_type locals globals func_decls e in
    let v1 = string_of_expr locals globals func_decls varName in
    let v2 = string_of_expr locals globals func_decls e
    in
      (* Check to make sure what is being assigned is the correct datatype *)
      if (e2_data_type = "element") or (e2_data_type = "scan")
      then
        if (NameMap.mem v1 locals) or (NameMap.mem v1 globals)
        then v1 ^ (" = " ^ v2)
        else raise (Failure ("undeclared identifier " ^ v1))
      else
        if (e1_data_type = "bool") && (e2_data_type = "int")
        then
          if (NameMap.mem v1 locals) or (NameMap.mem v1 globals)
          then v1 ^ (" = " ^ v2)
          else raise (Failure ("undeclared identifier " ^ v1))
        else
          if e1_data_type <> e2_data_type
          then
            raise
              (Failure
                 ("incompatible datatype during assignment. Expecting " ^
                    (e1_data_type ^
                       (" but " ^ (e2_data_type ^ " is found")))))
          else
            if (NameMap.mem v1 locals) or (NameMap.mem v1 globals)
            then
              if e1_data_type = "card"
              then v1 ^ ("->setname(\"" ^ (v2 ^ "\")"))
              else
                if e1_data_type = "list"
                then "@" ^ ((get_name v1) ^ (" = " ^ v2))
                else v1 ^ (" = " ^ v2)
            else raise (Failure ("undeclared identifier " ^ v1))
| LabelLiteral e -> ""
| ListLiteral e ->
    "(" ^
      ((String.concat ","
          (List.map (string_of_expr locals globals func_decls) e))
         ^ ")")
| CardLiteral e -> e
| Call ("$print", e) ->
    "print " ^
      (String.concat " "
         (List.map (string_of_expr locals globals func_decls) e))
| Call ("$println", e) ->
    "print " ^
      ((String.concat " "
```

```
        (List.map (string_of_expr locals globals func_decls) e))
       ^ ".\"\\n\"")
| Call ("$scan", e) ->
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in
      if (String.length v1) > 0
      then raise (Failure "scan function does not take any arguments. ")
      else "<STDIN>"
| Call ("$draw", e) -> (* wait until we add cardstacks to check . *)
    "$deck->draw()->getname()"
| Call ("$printDeck", e) -> (* wait until we add cardstacks to check . *)
    "$deck->print()"
| Call ("$shuffleDeck", e) -> "$deck->shuffle()"
| Call ("$renewDeck", e) -> "$deck = new DefaultDeck()"
| Call ("$value", e) -> (* check that v1 is a string that is valid. *)
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in "value(" ^ (v1 ^ ")")
| Call ("$deck", e) -> (* function to see if the deck is nonempty *)
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in
      if (String.length v1) > 0
      then raise (Failure "deck function does not any arguments. ")
      else "$deck->size()"
| Call ("$getName", e) ->
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in v1 ^ "->getname()"
| Call ("$getValue", e) ->
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in v1 ^ "->getvalue()"
| Call ("$getColor", e) ->
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in v1 ^ "->getColor()"
| Call ("$getSuit", e) ->
    let v1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e)
    in v1 ^ "->getSuit()"
| Call ("$getList", e) ->
    let l1 =
      String.concat " "
        (List.map (string_of_expr locals globals func_decls) e) in
    let l2 = split_char ' ' l1 in
    let v1 =
      "$" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
```

```
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in v1 ^ ("[" ^ (v2 ^ "]"))
  | Call ("$printList", e) ->
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v2 = String.sub v1 1 ((String.length v1) - 2)
      in "print @" ^ (v2 ^ ";\nprint \"\\n\"")
  | Call ("$sizeList", e) ->
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v2 = String.sub v1 1 ((String.length v1) - 2)
      in "scalar(@" ^ (v2 ^ ")")
  | (* Push string onto list *) Call ("$pushStr", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "push(" ^ (v1 ^ (",\"" ^ (v2 ^ "\")")))
  | (* Push int onto list *) Call ("$pushInt", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "push(" ^ (v1 ^ ("," ^ (v2 ^ "\")")))
  | (* TODO Push card onto list *) Call ("$push", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "push(" ^ (v1 ^ ("," ^ (v2 ^ "\")")))
  | (* Unshift string from list *) Call ("$unshiftStr", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
```

```
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "unshift(" ^ (v1 ^ (",\"" ^ (v2 ^ "\")")))
  | (* Unshift int from list *) Call ("$unshiftInt", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "unshift(" ^ (v1 ^ (",\"" ^ (v2 ^ "\")")))
  | (* Unshift card from list *) Call ("$unshift", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let l2 = split_char ' ' l1 in
      let v1 =
        "@" ^ (String.sub (List.hd l2) 1 ((String.length (List.hd l2)) - 1)) in
      let v2 =
        String.sub (List.hd (List.tl l2)) 0
          ((String.length (List.hd l2)) - 1)
      in "unshift(" ^ (v1 ^ (",\"" ^ (v2 ^ "\")")))
  | (* Pop string onto list *) Call ("$popStr", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "pop(" ^ v1 ^ ")"
  | (* Pop int onto list *) Call ("$popInt", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "pop(" ^ v1 ^ ")"
  | (* Pop card onto list *) Call ("$pop", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "pop(" ^ v1 ^ ")"
  | (* Shift string onto list *) Call ("$shiftStr", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "shift(" ^ v1 ^ ")"
  | (* Shift int onto list *) Call ("$shiftInt", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "shift(" ^ v1 ^ ")"
```

```
  | (* Shift card onto list *) Call ("$shift", e) ->
      let l1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e) in
      let v1 = "@" ^ (String.sub l1 1 ((String.length l1) - 2))
      in "shift(" ^ v1 ^ ")"
  | Call ("$read", e) ->
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e)
      in
        if (String.length v1) > 0
        then raise (Failure "read function does not any arguments. ")
        else "$_[0]"
  | Call ("$readStr", e) ->
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e)
      in
        if (String.length v1) > 0
        then raise (Failure "read function does not any arguments. ")
        else "$_[0]"
  | Call ("$readCard", e) ->
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e)
      in
        if (String.length v1) > 0
        then raise (Failure "read function does not any arguments. ")
        else "$_[0]"
  | Call ("$label", e) ->
      (* label function for breaks. Check to see e is a string. *)
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e)
      in v1 ^ ":"
  | Call ("$break", e) ->
      (* should check that e is a string. Check that e is on list of labels. *)
      let v1 =
        String.concat " "
          (List.map (string_of_expr locals globals func_decls) e)
      in "last " ^ v1
  | Call (f, actuals) -> (* See if function is defined *)
      (* Check if function is defined *)
      if NameMap.mem f func_decls
      then
        (let func = NameMap.find f func_decls
         in
           (* Check if the the number of arguments passed in matches expected num
of args *)
           (* List.iter2 will give error if # of args are not accurate - Fatal
error: exception Invalid_argument("List.iter2") *)
           (* fun x y will check if the data type matches *)
           (List.iter2
              (fun x y ->
                 if x.vartype <> (get_data_type locals globals func_decls y)
```

```
                   then
                     raise
                       (Failure
                          ("The arguments passed to function " ^
                             ((get_name f) ^ " are type mismatched")))
                   else ())
                func.formals actuals;
              (*(List.length actuals = List.length func.formals) then*)
              (get_name f) ^
                ("(" ^
                   ((String.concat ", "
                       (List.map (string_of_expr locals globals func_decls)
                          actuals))
                      ^ ")"))))
      else
        (*else raise (Failure ("The number of arguments passed to function " ^
(get_name f) ^ " is mismatched"))*)
        raise (Failure ("undefined function " ^ (get_name f)))

(* Generates a string representation of a statement*)
let rec string_of_stmt locals globals func_decls curr_func stmt =
  match stmt with
  | Block stmts ->
      "{\n" ^
        ((String.concat ""
            (List.map
               (fun x -> string_of_stmt locals globals func_decls curr_func x)
               stmts))
           ^ "}\n")
  | Expr e -> (string_of_expr locals globals func_decls e) ^ ";\n"
  | If (e, s, (Block [])) ->
      let data_type = get_data_type locals globals func_decls e
      in
        (*type checking *)
        if
          (data_type = "binop") ||
            ((data_type = "bool") || (data_type = "int"))
        then
          "if (" ^
            ((string_of_expr locals globals func_decls e) ^
               (")\n {" ^
                  ((string_of_stmt globals locals func_decls curr_func s) ^
                     "}\n")))
        else
          raise (Failure "if statement must take in a boolean expression ")
  | If (e, s1, s2) ->
      let data_type = get_data_type locals globals func_decls e
      in
        (*type checking *)
        if
          (data_type = "binop") ||
            ((data_type = "bool") || (data_type = "int"))
        then
          "if (" ^
            ((string_of_expr locals globals func_decls e) ^
               ("){" ^
```

```
                  ((string_of_stmt locals globals func_decls curr_func s1) ^
                     ("}\n else {" ^
                        ((string_of_stmt locals globals func_decls curr_func
                           s2)
                          ^ "}\n")))))
        else
          raise (Failure "if statement must take in a boolean expression ")
  | While (e, s) ->
      let data_type = get_data_type locals globals func_decls e
      in
        (*type checking *)
        if
          (data_type = "binop") ||
            ((data_type = "bool") || (data_type = "int"))
        then
          "while (" ^
            ((string_of_expr locals globals func_decls e) ^
               (") " ^
                  ((string_of_stmt locals globals func_decls curr_func s) ^
                     "\n")))
        else
          raise
            (Failure "while statement must take in a boolean expression ")
  | For (e1, e2, e3, s) ->
      let data_type = get_data_type locals globals func_decls e2 in
      let e2_str = string_of_expr locals globals func_decls e2
      in
        (*type checking *)
        if
          ((data_type = "binop") ||
             ((data_type = "bool") || (data_type = "int")))
            || ((String.length e2_str) = 0)
        then
          " for(" ^
            ((string_of_expr locals globals func_decls e1) ^
               ("; " ^
                  (e2_str ^
                     ("; " ^
                        ((string_of_expr locals globals func_decls e3) ^
                           ("){ " ^
                              ((string_of_stmt locals globals func_decls
                                  curr_func s)
                                 ^ "} \n")))))))
        else
          raise
            (Failure
               "for statement must take in a boolean expression as a second
parameter")
  | Return e ->
      (* Checks if what is being returned match with what the function expects *)
      if (get_data_type locals globals func_decls e) = curr_func.freturn
      then "return " ^ ((string_of_expr locals globals func_decls e) ^ ";\n")
      else
        raise
          (Failure
             ("Function: " ^
```

```
                ((get_name curr_func.fname) ^
                   (" needs to return of type: " ^ curr_func.freturn))))

let comp_func_main globals func_decls fdecl =
  (* Storing the function arguments in local symbol table*)
  (* checks if a argument name is used twice *)
  let locals =
    List.fold_left
      (fun lvars fargs ->
         if NameMap.mem fargs.varname lvars
         then
           raise
             (Failure (" Duplicate variable: " ^ (get_name fargs.varname)))
         else (* check to see if argument name is a keyword *)
           if is_keyword (get_name fargs.varname)
           then
             raise
               (Failure
                  ("variable name: " ^
                     ((get_name fargs.varname) ^ ", cannot be a keyword")))
           else NameMap.add fargs.varname fargs.vartype lvars)
      locals fdecl.formals in
  (* Storing the function local variables in local symbol table*)
  (* checks if a variable name is used twice *)
  let locals =
    List.fold_left
      (fun lvars fvars ->
         if NameMap.mem fvars.varname lvars
         then
           raise
             (Failure (" Duplicate variable: " ^ (get_name fvars.varname)))
         else (* check to see if argument name is a keyword *)
           if is_keyword (get_name fvars.varname)
           then
             raise
               (Failure
                  ("variable name: " ^
                     ((get_name fvars.varname) ^ ", cannot be a keyword")))
           else NameMap.add fvars.varname fvars.vartype lvars)
      locals fdecl.locals in
  (* Storing the string version of the function*)
  let func_str =
    String.concat "\n"
      (List.map (fun x -> string_of_stmt locals globals func_decls fdecl x)
         fdecl.body) in
  (* string representation of local *)
  let local_var_str =
    String.concat "\n" (List.map string_of_var_decl fdecl.locals)
  in
    (* return the string rep of the function *)
    "sub " ^
      ((get_name fdecl.fname) ^
         ("()\n{\n" ^ (local_var_str ^ (func_str ^ "}"))))

(* compiles a function *)
(* returns a string representation of the function *)
```

```
let rec comp_func globals func_decls fdecl =
  (* Storing the function arguments in local symbol table*)
  (* checks if a argument name is used twice *)
  let locals =
    List.fold_left
      (fun lvars fargs ->
         if NameMap.mem fargs.varname lvars
         then
           raise
             (Failure (" Duplicate variable: " ^ (get_name fargs.varname)))
         else (* check to see if argument name is a keyword *)
           if is_keyword (get_name fargs.varname)
           then
             raise
               (Failure
                  ("variable name: " ^
                     ((get_name fargs.varname) ^ ", cannot be a keyword")))
           else NameMap.add fargs.varname fargs.vartype lvars)
      locals fdecl.formals in
  (* Storing the function local variables in local symbol table*)
  (* checks if a variable name is used twice *)
  let locals =
    List.fold_left
      (fun lvars fvars ->
         if NameMap.mem fvars.varname lvars
         then
           raise
             (Failure (" Duplicate variable: " ^ (get_name fvars.varname)))
         else (* check to see if argument name is a keyword *)
           if is_keyword (get_name fvars.varname)
           then
             raise
               (Failure
                  ("variable name: " ^
                     ((get_name fvars.varname) ^ ", cannot be a keyword")))
           else NameMap.add fvars.varname fvars.vartype lvars)
      locals fdecl.locals in
  (* Storing the string version of the function*)
  let func_str =
    String.concat "\n"
      (List.map (fun x -> string_of_stmt locals globals func_decls fdecl x)
         fdecl.body) in
  (* string representation of local *)
  let local_var_str =
    String.concat "\n" (List.map string_of_var_decl fdecl.locals)
  in
    (* return the string rep of the function *)
    "sub " ^
      ((get_name fdecl.fname) ^
         ("()\n{\n" ^ (local_var_str ^ (func_str ^ "}"))))

(* compiles program *)
(* main entry point *)
(* returns a perl program *)
let compile (vars, funcs) = (* Put function declarations in a symbol table *)
  (* first function must be the main method *)
```

```
let first_func = List.hd (List.rev funcs)
in
  if first_func.fname <> "$main"
  then raise (Failure "The first function must be main")
  else
    (let func_decls =
       List.fold_left
         (fun funcs fdecl ->
            if NameMap.mem fdecl.fname funcs
            then
              raise
                (Failure
                   (" function name must be unique. function name used already: "
                      ^ (get_name fdecl.fname)))
            else (* check to see if function name is a keyword *)
              if is_keyword (get_name fdecl.fname)
              then
                raise
                  (Failure
                     ("function name: " ^
                        ((get_name fdecl.fname) ^ ", cannot be a keyword")))
              else NameMap.add fdecl.fname fdecl funcs)
         NameMap.empty funcs
     in
       (* must have a main function in program *)
       if ( != ) (NameMap.mem "$main" func_decls) true
       then raise (Failure " There must be a main function: ")
       else (* Put global variable declarations in a symbol table *)
         (* Checks to make sure the same name doesn't get used more than once*)
         (let globals =
            List.fold_left
              (fun gvars vdecl ->
                 if NameMap.mem vdecl.varname gvars
                 then
                   raise
                     (Failure
                        (" Duplicate variable: " ^ (get_name vdecl.varname)))
                 else (* check to see if variable name is a keyword *)
                   if is_keyword (get_name vdecl.varname)
                   then
                     raise
                       (Failure
                          ("variable name: " ^
                             ((get_name vdecl.varname) ^
                                ", cannot be a keyword")))
                   else NameMap.add vdecl.varname vdecl.vartype gvars)
              globals vars in
          (* Returns a perl program *)
          let value_sub =
            "sub value {\n$name = $_[0];\n$value = substr $name, 1;\n" ^
              ("if ($value eq \"A\")\n{\nreturn 1;\n}\nelse\n" ^
                 ("{if ($value eq \"J\")\n{\nreturn 11;\n}\nelse\n{\n" ^
                    ("if ($value eq \"Q\")\n{\nreturn 12;\n}\nelse\n{\n" ^
                       ("if ($value eq \"K\")\n{\nreturn 13;\n}\nelse\n{\n"
                          ^ "return $value;\n}\n}\n}\n}\n}\n")))) in
          let perl_start =
```

```
                "#!/usr/local/bin/perl\nuse warnings;\n" ^
                  "use Card;\nuse DefaultDeck;\n\n$deck = new DefaultDeck();\n"
              in
                perl_start ^ (* Global vars *)
                  ((String.concat "\n" (List.map string_of_var_decl vars)) ^

                     (* Main function *)
                     ((comp_func_main globals func_decls
                         (List.hd (List.rev funcs)))
                       ^ (* value subroutine *)
                       ("\n" ^
                          (value_sub ^
                             ("\n" ^ (* functions *)
                                (String.concat "\n"
                                   (List.map
                                      (fun x -> comp_func globals func_decls x)
                                      (List.tl (List.rev funcs)))))))))))
               ^ "\nprint main().\"\\n\";\n")
```

8.5 descartes.ml

```
type action = Ast | Compile

let _ =
      let action = if Array.length Sys.argv > 1 then
            List.assoc Sys.argv.(1) [ ("-a", Ast);

      ("-c", Compile) ]

      else Compile in
      let lexbuf = Lexing.from_channel stdin in
      let program = Parser.program Scanner.token lexbuf in
            match action with
                  Ast -> let listing = Ast.string_of_program program
                                                            in print_string listing
                  | Compile -> print_string (Compile.compile program)
```

8.6 Card.pm

```
#!/usr/bin/perl

package Card;

my $name;
my $value;

sub new {

      my $class = shift;
      my $self;
      if (@_ == 2) {
```

```
            $self = {
                  _name => $_[0],
                  _value => $_[1],
            };
      } elsif (@_ == 1) {
            if ($_[0] =~ /^[+-]?\d+$/) {
                  $self = {
                        _value => $_[0],
                  };
            } else {
                  $self = {
                        _name => $_[0],
                  };          }
      } elsif (@_ == 0) {
            $self = {
                  _name => "",
                  _value => -1,
            };
      }
      bless $self, $class;
      return $self;
}

sub getname {
      my ($self) = @_;
      $self->{_name};
}

sub getvalue {
      my ($self) = @_;
      $self->{_value};
}

sub setname {
      my ($self, $name) = @_;
      $self->{_name} = $name if defined($name);
      $self->{_name};

      my $value = substr $name, 1;
      if ($value eq "A") {
            $self->{_value} = 1;
      } elsif ($value eq "J") {
            $self->{_value} = 11;
      } elsif ($value eq "Q") {
            $self->{_value} = 12;
      } elsif ($value eq "K") {
            $self->{_value} = 13;
      } else {
            $self->{_value} = $value;
      }
}

sub setvalue {
      my ($self, $value) = @_;
      $self->{_value} = $value if defined($value);
      $self->{_value};
```

```
}

sub getSuit {
      my ($self) = @_;
      $suit = substr $self->{_name}, 0, 1;
      $suit;
}

sub getColor {
      my ($self) = @_;
      $suit = substr $self->{_name}, 0, 1;
      if ($suit eq "H" || $suit eq "D") {
            return "R";
      } elsif ($suit eq "C" || $suit eq "S") {
            return "B";
      }
}
1;
```

8.7 CardStack.pm

```
#!usr/bin/perl

package CardStack;
use Card;
use strict;

my @cards;

sub new {
      my $class = shift;
      my $self = {};
      if (@_ == 1) {
            setup($_[0]);
      }
      bless $self, $class;
      return $self;
}

sub setup {
      my $init = $_[1];
      $init =~ s/^\s+//;
      $init =~ s/\s+$//;
      $init =~ s/,/ /g;
      $init =~ s/\s+/ /g;
      my @cardStrings = split(/ /, $init);
      my $i = 0;
      while ($i < @cardStrings) {
            $cards[$i] = new Card($cardStrings[$i]);
            $i++;
      }
}

sub add {
      push(@cards, $_[1]);
      }
```

```
sub shuffle {
	srand;
	my @new = ();
	while (@cards) {
		push(@new, splice (@cards, rand @cards, 1));
	}
	@cards = @new;
}

sub draw {
	shift (@cards);
}

sub remove {
	splice (@cards, $_[1], 1);
}

sub size {
	my $size = @cards;
	$size;
}

sub print {
	my $i = 0;
	while ($i < @cards) {
		print $cards[$i]->getname();
		if ($i + 1 != @cards) {
			print " ";
		}
		$i++;
	}
}

sub getCards {
	@cards;
}
1;
```

8.8 DefaultDeck.pm

```
#!/usr/bin/perl

package DefaultDeck;
use CardStack;

our @ISA = "CardStack";

sub new {
	my $class = shift;
	my $self = {};
	bless $self, $class;
	$self->SUPER::setup ("SA, S2, S3, S4, S5, S6, S7, S8, S9, S10, SJ, SQ, SK,
HA, H2, H3, H4, H5, H6, H7, H8, H9, H10, HJ, HQ, HK, DA, D2, D3, D4, D5, D6, D7,
D8, D9, D10, DJ, DQ, DK, CA, C2, C3, C4, C5, C6, C7, C8, C9, C10, CJ, CQ, CK");
	my $i = 0;
```

```
        my $j;
        while ($i < 4) {
                $j = 0;
                while ($j < 13) {
                        @myCards = $self->SUPER::getCards;
                        $myCards[$i*13+$j]->setvalue($j + 1);
                        $j++;
                }
                $i++;
        }
        $self->SUPER::shuffle();
        $self;
}
1;
```

8.9 Makefile

```
OBJS =  scanner.cmo ast.cmo parser.cmo compile.cmo descartes.cmo

TARFILES = Makefile scanner.mll parser.mly \
        ast.ml compile.ml descartes.ml \

descartes : $(OBJS)
        ocamlc -o descartes.exe $(OBJS)

descartes : $(OBJS)

.PHONY : test
test : descartes.exe testall.sh
        ./testall.sh

.PHONY : blackjack
blackjack : test-blackjack.pl
        perl test-blackjack.pl

.PHONY : highlow
highlow : test-highlow.pl
        perl test-highlow.pl

.PHONY : helloworld
helloworld: test-decktest.pl
        perl test-decktest.pl

scanner.ml : scanner.mll
        ocamllex scanner.mll

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

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

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

```
.PHONY : clean
clean :
      rm -f descartes.exe parser.ml parser.mli scanner.ml testall.log \
      *.cmo *.cmi *.out *.diff *.pl *.mli *.cmi

#Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
compile.cmo: ast.cmo
compile.cmx: ast.cmx
descartes.cmo: scanner.cmo parser.cmi compile.cmo \
    ast.cmo
descartes.cmx: scanner.cmx parser.cmx compile.cmx \
    ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```