COMS W4115 – Fall 2011
Prof. Stephen Edwards
Language Reference Manual

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

DESCARTES

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.

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
```

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.

Comments

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

Examples:

```
/* This is a comment */
/* This is

a multi-line

comment */
```

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."

Constants

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

Integer constants

An integer constant is a sequence of digits.

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 (\").

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

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.

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);
```

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 3 tuples */
```

Keywords

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

int	break	for	String	true
char	continue	while	List	false
null	if	switch	goto	boolean
return	else	extend	case	default
print				

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.

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
```

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

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

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.

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, CardStack)	This will move a Card Object from one card stack to another card stack. It removes the card in the second parameter (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.

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.

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 Identifiers. 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 Constants section 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 String Constants section 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.

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.

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.

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.

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.

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.

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.

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.

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.

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;

Storage Specifier

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

Type-Specifiers

int
boolean
String
List
standard and user-defined types

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[]

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.

Statements

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

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;

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

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.

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.

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.

Switch Statement

The switch statement allows for multiple execution paths based on the value of a single expression.

switch (expression) statement

With the expression being a primitive type, the switch statement leads to the following statement, which is typically compound. The statements within the compound statement are typically of the following forms:

case expression
default expression

Depending on the expression, one of the case constants will be executed. They are checked in an undefined order. If none of the case constants are satisfied, then the default constant will execute. If there is no default constant, then nothing happens. Two case constants cannot have the same value.

Here is an example. The formatting is as follows:

```
int month = 8;
String monthString;
switch (month) {
case 1: monthString = "January";
case 2: monthString = "February";
case 3: monthString = "March";
case 4: monthString = "April";
case 5: monthString = "May";
case 6: monthString = "June";
case 7: monthString = "July";
case 8: monthString = "August";
case 9: monthString = "September";
case 10: monthString = "October";
case 11: monthString = "November";
case 12: monthString = "December";
default: monthString = "Invalid month";
}
```

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;

Continue Statement

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

Form:
continue;

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.

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.

Special Compiler Commands

Special Compiler Commands

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

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];
```

Multiple Classes

To include other classes or files, use the import statement.

Form:
```
import "filename";
```

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.

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];
            }
    }
}
```