CARDIGAN

Card game development and implementation language

I. Team

Nithin Chandrasekharan (nc2470)
Muzi Gao (mg3194)
Josh Lopez (jl3497) -- Project Leader
Miriam Melnick (mrm2198)

II. Language Description

Cardigan is a language which can be used to define, develop, test, and play card games. Many existing languages already contain several of the data types and control structures necessary to implement simple card games, but because the languages were not specifically developed for this purpose, some setup is usually required. Cardigan seeks to provide developers with tools specifically suited for creating and deploying these games with minimal additional coding. This is accomplished by providing a small set of powerful yet flexible object types, as well as an easy to manage function declaration system. Card games can be easily implemented by simply defining sets of objects (cards, players) and functions (turns, card actions, rules). Cardigan should not be confused with a framework, which only allows users to fill in sets of customized data, but provides little control over how the game is played. A game defined in Cardigan may define any type of rules and gameplay, and may modify these elements during the game to any extent. A turn structure may contain as many actions as desired. A game may even be implemented with no turn structure at all. Cardigan also allows developers to create computerized opponents powered by algorithms which they can define in the language.

III. Problems Which Cardigan Can Solve

Developing card games is an extremely time consuming process usually requiring multiple iterations of text and artwork, numerous modifications to existing cards, tedious organization of numerous permutations of cards, trimming decks to size, and a fair amount cutting and pasting (with actual scissors and paste). Considering most games contain several copies of the same card, the entire process must be repeated several times per deck. Production of the product requires additional investments of time and money, as well as other overheads associated with a physical inventory.

Cardigan seeks to minimize all of these problems simultaneously by moving the entire development and production process into a single programming language. Each completed program represents a game, and any modifications to the game's rules, cards, players, and other factors require changing only a few lines of code.

The nature of programs also facilitates sharing between developers in a way that is not possible with physical cards. A Cardigan developer could access previously written code which implements a particularly popular aspect of many games, and incorporate it into their project. Any developer who choses to make their code publically available could open their games to further modifications and ‘house rules’.

IV. Representative Programs

As an example we present several existing card games, implemented in Cardigan, which take advantage of the language features including:

- Primitive types -- used to define the attributes of the cards
- Structs – cards and players are defined using the built in struct type.

- Built in functions – creation of the deck uses the cartesian() function to allow programmers to define multiple groups of cards without having to explicitly define every card.
- Collections – sets of cards (rounds, and players hands) are defined as collections.
- Functions – turns are implemented via functions, containing standard control structures (for, while, if).
- Rules -- the rules for each game are implemented using the rules structure
- Operators – used for determining and executing the correct rule

1. War -- War uses a simple comparison of card values to determine the winner of each round
2. Uno -- This game requires checking which card is currently face up on the table, and what action to take based on each turn
3. SET -- Usage of the cartesian product in more than two dimensions as well as comparisons and several control structures.
4. Milles Bornes -- Another game which uses the rule structure as well as multiple player attributes.

V. Language Syntax

Statements – Statements are made up of identifiers, operators, declarations, assignments, and function calls. The newline character terminates a statement. A long statement may be broken into multiple lines by escaping the newline character with a backslash.

Blocks – Several control structures require code to be organized into blocks. These are: if, else-if, else, while, for, and rule structures. Blocks must be contained within curly braces. A single newline character immediately after an opening curly brace is allowed for readability, but is not required.

Comments – Commented lines begin with double slashes. Cardigan only supports single line comments.

Control structures – Four standard control structures are supported, each following a similar syntactical pattern (keyword, information, codeblock).

- `if` – the if keyword must be followed by a predicate, which must evaluate to either `true` or `false`, and then a block of code. If the predicate evaluates to `true`, the block of code is executed and control leaves the `if` structure. This can then be followed by 0 or more `elseif` statements, each of which has its own predicate and code block, and a single optional `else` block which has only a block and no predicate. In an `if` structure without an `else` block, at most one code block is executed. In an `if` structure with an `else` block, exactly one code block is executed.
- `while` – the `while` statement is made up of keyword, predicate, and code block. The predicate is evaluated and if its value is `true`, the code block is executed and the predicate is re-evaluated. The code block will be run once for every time the predicate evaluates to `true`. When the predicate evaluates to `false`, control passes out of the `while` structure.
- `for` – The `for` statement is used to iterate over items in a collection. The syntax of a `for` loop is: the `for` keyword, the name to use for the local variable in the code block, a colon, the name of the collection, and the code block to execute. The code is run once with each item in the collection. The item is passed to the code block using the specified variable name, and the code is run. The `for` loop's code block will iterate once for each item in the collection, assigning each object's value to the variable in scope sequentially.

The objects will be treated as the same type as those in the collection.

- `rules` - The `rules` statement is used to check several (potentially related) conditionals. It begins with a question mark, followed by an optional set of parentheses that contain a comma-delimited list of local variable assignments, and then a brace-delimited block. The block can contain 0 or more conditional statements, each consisting of an expression, evaluating to a boolean, followed by a code block. When control reaches the `rules` structure, the specified variables are bound and each conditional is evaluated in sequence. Anywhere between 0 and all of the conditionals may be executed, depending on which predicates evaluate to `true`. When control reaches the end of the brace-delimited block, the local bindings defined at its beginning go out of scope and control passes out of the `rules` structure.

Built in types

- Primitive types -- There are
    - Ints: ints represent integer values. Any number which does not contain a decimal point is considered an int. Standard arithmetic operators (`+, *, -, /, %`) may be applied to ints and the result will be another int.
    - Floats: floats represent floating point numbers. Any number which contains a decimal is considered a float. Although floats are stored as strings, they may not be used interchangeably. Floating point arithmetic operators (`+, *, -, /`) applied to two floats will return a float. If applied to an int and a float, the result will also be a float.
    - Bools: boolean true and false values, represented by the `true` and `false` keywords respectively. Bools can be the operands of any of the following boolean operations: not, and, or, xor. Bools are also used as the return type of comparison evaluations.
    - Strings: printable characters. Strings are contained within double quotes. Several escape sequences are allowed including \” -- a double quote character, \n -- a newline character, \\ a backslash character. Strings are implemented as arrays of characters.
- Derived types
    - Collections: an object which is structured like a linked-list, but with a single additional internal node which contains pointers to every node in the list. Collections are defined using square brackets, with each entry separated by a comma. This allows instances of this type to be used as lists, stacks, queues, or arrays interchangeably as necessary. Collections can be used to implement decks of cards, player hands, groups (teams) of players, and even sets of actions if necessary. Elements in a collection may be access by index, using square brackets, or via a method of the collection.
    - Structs: an object which contains other objects. Structs are declared with curly braces and may contain any number of assignment statements including functions and other structs. Each object in a struct is defined with a name and is accessed using the name with dot notation. Structs are implemented as hash tables containing pointers to other object type, allowing any number of fields to be added and changed on the fly. In addition, structs may be inherited. The name of an existing struct may be prepended to the curly braces which causes the values in the named struct to be copied over to the new struct. Any assignments contained within the inheriting struct override those of the existing struct. Structs can, therefore, be used in a similar manner as classes in other languages, with inheritance taking the place of instantiation. Constructors like functions must be

called manually however.

  - Functions: blocks of statements. Functions may be assigned to identifier names and subsequently called using the identifier. Arguments to functions must be declared as a part of the identifier. The named arguments may be used as variables within the scope of the function, The names of the arguments do not have to match the declaration when the function is called, but the number of arguments must be the same. This means a function declared as `functionName(arg1, arg2)` may be called by `functionName(foo, bar)`, but not by `functionName()`. Functions may specify a return value, using the `return` keyword. If the call to the struct is part of an assignment statement, the returned value will be assigned. Functions which do not explicitly return a value will return boolean `false`.
  - Tuples: Tuples represent small, immutable collections of objects. Elements of a tuple are separated by colons. None of the functionality of collections or structs are supported by tuples. Elements of a tuple may only be accessed by index, using square brackets.

Cardigan is a statically typed language, but the type of a variable is determined when it is assigned. As a result, no type keywords are necessary when creating variables. All assignment, including function declaration, is done via the following syntax: identifier = value. The value may be any of the described types, but once a variable is declared it may only be assigned values of the same type.

A very small standard library, containing the following definitions, is included:

- `Deck` – a struct type defined to contain fields designed to describe a deck of cards
- `Player` – a struct type defined to hold information defining a player in a game
- `cartesian(attributes)` – a function which returns a collection of the cartesian product of the specified attributes
- `INPUT()` – a function which pauses execution until keyboard input is entered. The keyboard input is then returned as a string. Casting functions will be provided to allow input of other types.
- `OUTPUT(string)` – a function which prints the specified string to the screen

VI. Sample Programs

```
// War
// http://www.pagat.com/war/war.html
CARD = 0
PLAYER = 1

rank = ["2":0, "3":1, "4":2, "5":3, "6":4, "7":5, "8":6, "9":7, "10":8, "J":9, "Q":10, "K":11, "A":12]
suit = ["H", "C", "D", "S"]

play = {
        OUTPUT("How many players?")
        INPUT(num_players)
        players = []
        for id:[0 ... num_players]{
                players.append(player{id=id, hand=hand{look=0, sequential=1}})
        }

        deck = Deck{cards=cartesian(rank, suit), labels=['rank','suit']}
        deck.deal(players, deck.size/num_players)


        winner = player{}
        while !winner{
                result = round(players)
                players[result[0]].hand.add(cards)
                if players[result[0]].hand.size == deck.size{
                        winner = round_winner
                }
        }
        OUTPUT(“player “ + winner.id + “ wins!”)
}

round(players) = {
        round = []
        for player:players{
                played = player.hand.pop()
                OUTPUT(“Player “ + player.id + “plays: “ + played.rank + played.suit)
                round.append([player.hand.pop(), player.id])
        }
        round.sort(lambda x:x[CARD].value)
        high_card = round.pop()
        if high_card[CARD].value == round.peek()[CARD].value{
                OUTPUT(“WAR!”)
                result = war(high_card, round)
                round_winner = result[0]
                cards = result[1]
        } else {
                round_winner = players[high_card[PLAYER]]
                cards = round
        }
        return [round_winner, cards]
}

war(high_card, round) = {
        participants = players[high_card[PLAYER]]
        while round && round.peek()[CARD].value == high_card[CARD].value{
                participants.append[players[round.pop()[PLAYER]]]
        }
        spoils = []
        for player:participants{
                spoils.add(player.hand.pop(3))
        }
        result = round(participants)
        spoils.add(result[1])
        return [result[0], spoils]
}
```

```
// The Game of SET
// See www.setgame.com
// Only rule: 3 cards form a SET if and only if, for each attribute,
// either they all have the same value or they each have a different value.

// Note: This version has one computer "player" and 0 human players.

number = ["1":0, "2":1, "3":2]
color = ["red":0, "green":1, "purple":2]
fill = ["empty":0, "striped":1, "solid":2]
shape = ["oval":0, "squiggle":1, "diamond":2]

attrNames = ["number", "color", "fill", "shape"]

play = {
        deck = Deck{cards=cartesian(number,color,fill,shape)}
        table = {cards = []}

        deck.shuffle()

        // Deal 12 cards onto the table
```

```
    for i: range(1,12){
        // take the top card off the deck and add it to the table
        table.cards.append(deck.pop())
    }

    // search for set
    // if set found, take it
    // else deal 3 cards
    while (table.cards.size > 0) {
        set = findSet()
        if (set.size > 0) {
            // remove from table
            for card: set {
                table.cards.delete(card)
            }

            if (table.cards.size < 12) {
                tryToDeal()
            }
            } else {
                tryToDeal()
            }
    }
}

// looks for a SET. [returns a collection]
// if it finds a SET, it returns it immediately as a collection
// if it finds no SET, it returns an empty collection
findSet = {
    for first: table.cards {
        for second: table.cards {
            if (!first.equals(second)) {
                goal = {}
                for attr: attrNames {
                    goal.attr = (3 - (first.attr + second.attr)) % 3
                }
                for third: table.cards {
                    if (!third.equals(first) && !third.equals(second))) {
                        return [first, second, third]
                    }
                }
                return []
            }
        }
    }
}

// deals 3 cards [void]
// if there are <3 cards in the deck, it does not deal any.
// NOTE: there should always be 3k cards in the deck.
// Therefore if there are <3, there are 0.
tryToDeal = {
    if (deck.size > 0) {
        for i: range(1,3) {
            table.cards.append(deck.pop())
        }
    } else {
        endGame()
    }
}

endGame = {
    OUTPUT("Game over.")
}
```

```
// Mille Bornes
// http://www.gameyum.com/other-card-games/105088-the-rules-for-mille-bornes-the-card-game/
play() = {
  card_defs = [
          // Remedy cards
          ["repairs","body","remedy"]:6,
          ["spare tire","tire","remedy"]:6,
          ["gasoline","gas","remedy"]:6,
          ["go","sign","remedy"]:14,
          ["end of limit", "speed", "remedy"] : 6,

          // Hazard cards
          ["accident","body","hazard"]:3,
          ["flat tire","tire","hazard"]:3,
          ["out of gas","gas","hazard"]:3,
          ["stop","sign","hazard"]:5,
          ["speed limit", "speed", "hazard"] : 4,

          // Safety cards
          ["driving ace","body","safety"]:1,
          ["puncture proof","tire","safety"]:1,
          ["extra tank","gas","safety"]:1,
          ["right of way","sign","safety]:1"

          // Mileage cards
          [25, "", "mileage"] : 10,
          [50, "", "mileage"] : 10,
          [75, "", "mileage"] : 10,
          [100, "", "mileage"] : 12,
          [200, "", "mileage"] : 4
  ]

  cards = []
  for card:card_defs{
          for count:[0 ... card[1]]{
          cards.append(card[0])
          }

  deck = Deck{cards=cards, labels=["name", "class", "type"]}

  OUTPUT("How many players")
  INPUT(num_players)
  players = []
  for id:[0 ... num_players]{
          players.append(player{id=id, mileage=[], drive=[], speed=[], safety=[], coupFourre = []})
  }

  deck.shuffle()
  deck.deal(players, 6)

  player = players.pop()
  while(player.mileage < 1000){
           players.push(player)
           player = players.dequeue()
           error = false

           while !error{
                    OUTPUT(player.hand.show())
                    OUTPUT("Choose card")
                    INPUT(card_index)
                    played = player.hand.take(card_index)
                    ?{
                            // Hazard
                            played.type == "hazard"{
                                    OUTPUT("Which player will you play this on?")
                                    OUTPUT(players.show())
```

```
                                        INPUT(target)
                                        ?{
                                                  (played.class != "speed") && (target.drive.peek().type == "remedy")
&& !(played.class in target.safety) && !(played.class in target.coupFourre){
                                                            target.drive.push(played)
                                                  }
                                                  played.class == "speed"{
                                                            target.speed.push(played)
                                                  }
                                                  {
                                                            OUTPUT("You can't do that.")
                                                            error = true
                                                  }
                                        }
                              }
                              // Mileage
                              (played.type == "mileage" && player.drive.peek().name == "go" && !
(player.speed.peek().name == "speed limit" && played.name > 50)){
                                        player.mileage.push(played)
                               }
                              // Remedy
                              (played.type == "remedy" && player.drive.peek().type == "hazard" &&
player.drive.peek().class == played.class){
                                        player.drive.push(played)
                              }
                              // CoupFourre
                              played.type == "safety"{
                                        drive = player.drive.peek()
                                        if (drive.type == "hazard" && drive.class == played.class){
                                                  player.drive.pop()
                                                  player.coupFourre.append(played)
                                        } else {
                                                  player.safety.append(played)
                                        }
                              }
                              {
                                        OUTPUT("You can't do that.")
                                        error = true
                              }
                    }
          }
  }
  OUTPUT("Player " + player.id + " wins!")
}
```

```
// The Game of UNO
// See http://www.wonkavator.com/uno/unorules.html

rank =
[“0”:0, "1":1, "2":2, "3":3, "4":4, "5":5, "6":6, "7":7, "8":8, "9":9, "1":1, "2":2, "3":3, "4":4, "5":5, "6":6, "7":7, "
8":8, "9":9, "DRAW-2":20, "REVERSE":20, "'SKIP":20]

//copy(obj, num_target)
wild = copy(["WILD":50],4)
wild-4 = copy(["WILD-DRAW-4":50],4)

deck = Deck{cartesian(rank, suit) UNION wild UNION wild-4}
deck = deck.shuffle()

players = []
discard = []

starter = 0
clockwise = true

play = {
        OUTPUT("How many players?")
        INPUT(num_players)

        for id:[0 ... num_players]{
                players.append(player{
                        id:id,
                        cards:[],
                        dealer:false
                })
        }

        setDealer()
        init()

        winner = player{}
        //the player who will throw down one card
        starter = getDealer()

        while !winner{
                starter = throwDown(starter, clockwise)

                for player:players{
                        if player.cards.size == 1{
                                OUTPUT(player.id + "said: UNO!")
                        }
                }

                for player:players{
                        if player.cards.size == 0{
                                winner = player
                        }
                }
        }
}
//determine dealer's id
setDealer(){
        for player:players{
                card = deck.pop()
                while !isNumberCard(card){
                        deck.append(card)
                        card = deck.pop()
                }

                player.cards.append(card)
        }

        players.sort(lambda p:p.cards[0])
        players.reverse()

        players[0].dealer = true

        //clear cards for each player
        for player:players{
                player.cards.pop()
        }

        deck.shuffle()
}

//return dealer's id
getDealer(){
        for player:players{
                if player.dealer {
                        return player.id
                }
        }
```

```
}

//check if the card with name "0" - "9"
isNumberCard(card){
        for num:[0 ... 10]{
                if card.key[0] == num.toString(){
                        return true
                }
        }
        return false
}

//initialize the game, each player will get 7 cards
init(){
        index_starter = getDealer(players)
        for index:[index_starter ... players.size()]{
                for i:[0,8]{
                        players[index].append(deck.pop())
                }
        }

        for index:[0, index_starter]{
                for i:[0,8]{
                        players[index].append(deck.pop())
                }
        }
}

throwDown(starter, clockwise){
        if !discard{
                top = discard[0]
                //if top card is "Number-card", let start throw down one card or pick up one
                ?{
                        (isNumberCard(top)) || (top[key[0]] == "WILD"){
                                pickCard(players[starter],top)
                        }
                        top.key[0] == "DRWN-2"{
                                for i:[0 ... 2]{
                                        players[starter].cards.append(deck.pop())
                                }
                        }
                        top.key[0] == "REVERSE"{
                                if clockwise{
                                        starter--
                                }
                                else{
                                        starter++
                                }
                                clockwise = not(clockwise)
                        }
                        top.key[0] == "WILD-DRAW-4"{
                                pickCard(players[starter],top)
                                for i:[0 ... 4]{
                                        players[starter].cards.append(deck.pop())
                                }
                        }
                }
        }
        else{
                discard.push(pickCard(players[starter]))
        }

        if clockwise{
                starter++
                if starter>=players.size(){
                        starter = 0
                }
            }
            else{
                starter--
                if starter<0 {
                    starter = players.size() - 1
                }
            }

}
pickCard(player,top){
            ?{
                isNumberCard(top){
                OUTPUT("Please throw doan a card with number " + top.key[0] + " or color" + top.key[1])
                }
                top.key[0] == "WILD"{
                    OUTPUT("Assign new color")
                    INPUT(color)
                    OUTPUT("Please throw doan a card with color" + color)
                }
            }

            INPUT(num_card)

            if num_pickup == -1{
                OUTPUT(players[starter].id + "said: pass!")
                players[starter].cards(append(deck.pop()))
            }
            else{
                discard.push(players[starter].cards[num_pickup])
                players[starter].cards.remove(num_pickup)
            }
}
```