CardBoulevard: Language Specification

David Alfaro and Natalia Avila-Hernandez

Spring 2025

1 Introduction

CardBoulevard is a programming language intended to make creating card games like poker, blackjack, and games with custom rules and logic, easier for the user and programmer. This current version of CardBoulevard can generate a playable game for users following a variety of constraints. This language takes away the complexities of card and player object management, allowing developers to prototype, test, and run card-based games efficiently. The goal of CardBoulevard is to provide a declarative, human-readable way to manage and define cards and decks, rules, game state changes, and win conditions.

By taking in a user-written file, CardBoulevard interprets the code and generates a playable game that responds to user input as determined by the original code. such as user choices and custom conditional logic. CardBoulevard's current version can handle a variety of game logic, from simple one-turn games to more complicated games with multiple players, turns, and custom win conditions. It's structure is designed to be easily readable and accessible, even to users with limited programming experience.

2 Design Principles

CardBoulevard's design is guided by the goal of creating card games both simple and powerful. The language is made for those who have limited experience programming and who want to experiment with game rules, card dynamics, and win conditions. The language is rooted in simplicity through these key features:

- **Simplicity and Readability**
  The syntax of CardBoulevard is short and sweet. The commands resemble natural language when creating games and helps users understand the logic of the games which they are creating. This simplicity helps users quickly identify and read their own program, allowing for good self-expression and easier modifications of game logic. Since creating custom objects for cards and decks is tedious, especially for those with no programming language, instead we allow users to use a sort of template and create their card in whichever way they see fit. CardBoulevard allows for custom suits– or types– of cards, allowing for infinite possibilities.
- **Interactivity and Customization**
  This language isn't just for incredibly static programs. The goal of this program is to allow for game-interaction through user-input and game logic. Users can simulate turns, players, and receive feedback based on their game's logic and what they have customized their game to be. CardBoulevard also allows for customization of cards, letting the possibilities of game-making to be endless.

In general, CardBoulevard is designed with user simplicity and readability in mind, while also functioning as a powerful tool to create custom card games without the tedious work of learning a difficult and expansive language. It's designed to be adaptable and to be used by all, and thinking of basic concepts to let users think of mostly logic and concepts of their desired card game.

3 Example Programs

To run these example programs, type `dotnet run filename`, where filename is the name of the file while being in the correct repository. For these examples, they are held in the example folder, in the test folder, in the code repository, named `example-1.boul`, replacing 1 with 2 or 3 for each respective program. So, running `dotnet run`

`../tests/examples/example-1.boul` while being in the code repository will start the examples!

Example 1: Draw One Battle
This game consists of two cards in a deck, a 1 of diamonds and a 2 of diamonds, and the game is that two players draw one card each and whoever gets the 2 of diamonds wins the game.

Input in example-1.boul :
```
player 1 {"dealer", [(diamond, 1),(diamond, 2)]}
player 2 {"player1", []}
player 3 {"player2", []}

shuffle "dealer"
answer := prompt "Player 1, do you want the top or bottom card?  Top [1], bottom
[0]"
nother := false

if (answer) (takecard "dealer" "player1")
if (answer) (takecard "dealer" "player2")
opposite := answer = nother
if (opposite) (takecard "dealer" "player2")
if (opposite) (takecard "dealer" "player1")

score1 := sum "player1"
score2 := sum "player2"

if (score1 > score2) (winner "player1")
if (score1 < score2) (winner "player2")
```

Example 2: 21-Over Battle
This game is a luck-based blackjack-inspired game that involves a player drawing cards until the sum of their hand is over 21. The player wins if the hand is over 21.

Input in example-2.boul :
```
player 1 {"dealer", [(diamond, 1),(diamond, 2),(diamond, 4),(diamond, 5),(diamond,
6),(diamond, 7),(diamond, 8),(diamond, 9),(diamond, 10),(diamond, 11),(heart, 1),(heart,
2),(heart, 4),(heart, 5),(heart, 6),(heart, 7),(heart, 8),(heart, 9),(heart, 10),(heart,
11)]}
player 2 {"player1", []}
player 3 {"player2", []}
player 4 {"dump", []}
player 5 {"limit", [(diamond, 21)]}

shuffle "dealer"

while sum "player1" < sum "limit" (takecard "dealer" "player1")
while sum "player2" < sum "limit" (takecard "dealer" "player2")

if (sum "player1" > sum "player2") (winner "player1")
if (sum "player1" < sum "player2") (winner "player2")
```

Example 3: Black Jack
This game consists of a deck of cards consisting of two suits, with each suit having 10 cards. There is a dealer /(com-

puter/) and a player. The goal of the game is to get as close to 21 as possible without going over.

Input in example-3.boul :

```
player 1 {"dealer", [(diamond, 1),(diamond, 2),(diamond, 4),(diamond, 5),(diamond,
6),(diamond, 7),(diamond, 8),(diamond, 9),(diamond, 10),(diamond, 3),(heart, 1),(heart,
2),(heart, 4),(heart, 5),(heart, 6),(heart, 7),(heart, 8),(heart, 9),(heart, 10),(heart,
11)]}
player 2 {"player1", []}
player 6 {"computer", []}
player 9 {"computerstrat", [(diamond, 17)]}
player 5 {"limit", [(diamond, 21)]}
player 10 {"Nobody", [(diamond, 21)]}
player 7 {"limitup", [(diamond, 22)]}

answer2 := false
answer3 := false

prompt "Aim to get a sum of cards under 21!  :D Understand?  Yes[1] Also Yes[0]"

shuffle "dealer"
takecard "dealer" "player1"
takecard "dealer" "player1"

sum "player1"
answer1 := prompt "Take card?  Yes [1] No[0]"
if (answer1) (takecard "dealer" "player1")
sum "player1"
if (answer1) (answer2 := prompt "Take card?  Yes [1] No[0]")
if (answer2) (takecard "dealer" "player1")
sum "player1"
if (answer2)(answer3 := prompt "Take card?  Yes [1] No[0]")
if (answer3)(takecard "dealer" "player1")
takecard "dealer" "computer"
takecard "dealer" "computer"

while sum "computer" < sum "computerstrat" (takecard "dealer" "computer")
if (sum "computer" > sum "player1")(if(sum "computer" < sum "limitup")(winner "computer"))
if (sum "player1" > sum "limit")(if(sum "computer" < sum "limitup")(winner "computer"))
if (sum "player1" > sum "computer")(if(sum "player1" < sum "limitup")(winner "player1"))
if (sum "computer" > sum "limit")(if(sum "player1" < sum "limitup")(winner "player1"))
if (sum "computer" > sum "limit")(if(sum "player1" > sum "limit")(winner "Nobody"))
```

4 Language Concepts

A user of CardBoulevard should be familiar with basic card and deck components. It would help to be familiar with general card game mechanics and logic, so that it would be easier to produce their own decks and cards and create the logic to their own games. While actually using our program, they would need to know the basic functions used during card games, including shuffling, sum, conditionals, etc. using the primitives such as Card, Deck, Player, and basic types like int and num. Any user who can understand what type of game they want to create can easily use our language. The only thing a user would have to be aware of is the syntax of our language, as shown below, and understand the logic of whatever type of game they are trying to create.

5 Formal Syntax

```
<expr> ::= <num> | <str> | <bool> | <carddef> | <deckdef> | <playerdef> | <seq> |
       <sum> | <var> | <equal> | <greater> | <lesser> | <shuffle> | <assign> | <first>
       | <show> | <take> | <has> | <winner> | <while> | <if> | <prompt> | <parens>
<num> ::= n in Z
<str> ::= string
<name> ::= <str>
<bool> ::= "true" | "false"
<carddef> ::= (<num>, <suit>)
<deckdef> ::= {<name>, [<carddef>]}
<playerdef> ::= (<num>, <deckdef>)
<suit> ::= Heart
       | Diamond
       | Spade
       | Club
<sum> ::= (<num> + <num>)
<equal> ::= (<expr> = <expr>)
<greater> ::= (<expr> > <expr>)
<lesser> ::= (<expr> < <expr>)
<shuffle> ::= shuffle <deckdef>
<first> ::= firstcard <deckdef>
<show> ::= showcard <deckdef>
<take> ::= takecard <deckdef>
<has> ::= hascards <deckdef>
<winner> ::= winner <playerdef>
<while> ::= while <expr> <expr>
<if> ::= if <expr> <expr>
<parens> ::= (<expr>)
```

6 Semantics

Syntax	Abstract Syntax	Prec./Assoc.	Meaning
<n>	Num of int	n/a	A numeric literal. Represents an integer.
true, false	Bool of bool	n/a	A boolean literal. Used in boolean conditionals
Card(Suit)(Number)	CardDef of Card	n/a	A card with a given suit and number
Deck(Name)(Cards)	DeckDef of Deck	n/a	A deck with a given name and a list of cards
Player(Id,(Deck))	PlayerDef of Player	n/a	A player with a given id and an initial deck of cards
Sum <Player>	Sum of String	1/left	Computes the sum of card values in the specified player's hand
<expr> == <expr>	Equal of Expr * Expr	1/left	Evaluates to true if both expr are equal
<expr ¿ <expr>	Greater of Expr * Expr	1/left	Evaluates to true if the left expr is greater than the right expr
<expr ¡ <expr>	Lesser of Expr * Expr	1/left	Evaluates to true if the left expr is lesser than the right expr
<name>	Var of String	n/a	References a previously declared variable
shuffle <name>	Shuffle of String	n/a	Randomizes the order of cards in a specified deck
<var> = <value>	Assign of Expr * Expr	n/a	Assigns a value to a variable
firstcard <player>	Firstcard of String	n/a	Returns the first card in a players hand
showcard <index> <player>	Showcard of int * String	n/a	Displays a specified card depending on the index
takecard <player> <player>	Takecard of String * String	n/a	Takes a card from a player and gives it to another player
hascards <player>	Hascards of String	n/a	Returns true if the player has any cards in their deck
winner <player>	Winner of String	n/a	Declares the player as the winner
while <cond> <body>	While of Expr * Expr	n/a	Repeats the body until the condition evaluates as true
if <cond> <body>	if of Expr * Expr	n/a	Repeats the body only if the condition evaluates as true
prompt <string>	prompt of String	n/a	Reads console input and returns true for 1 or false for 0

7 Remaining Work

Further development for CardBoulevard would include some of the following aspects:

- **Detailed customization of cards:** Currently, our implementation only allows for custom suits, or types, of cards, but in further implementations, we would like to be able to add attributes and descriptions to each card. The result would ideally look, or be able to achieve, something close to Pokemon cards, with health, damage, attributes, etc.
- **Graphical Interface:** The output of our current program is a text-based terminal interface, which doesn't include any graphics to make it easier for users to see their finished product. We would like to implement, in

future versions of our program, a sort of graphical interface, to be able to see a graphical end product of whatever game a user has created.

Our original plan for CardBoulevard had custom cards and our first point up above, but because of time constraints and the complexity of the problem, we were unable to successfully implement it, and therefore we left it out of the final branch. The future of CardBoulevard is exciting, with many new additions and implementations to come!