AllHandsOnDeck:
A Universal Card Game Language

Caitlyn Chen	Tiffeny Chen	Jang Hun Choi	Mara Dimofte	Christi Kim
Language Guru	*System Architect*	*System Architect*	*Manager*	*Tester*
`ckc2143`	`tc2963`	`jc5112`	`md3713`	`cwk2109`

1 Introduction

Card games come in many different forms: games based off the standard 52-card deck such as War or Blackjack, and games relying on unique decks such as Apples to Apples, UNO, SET, etc. We drew inspiration from past proposals, which shared similar motivations of building out languages aimed to support card game development. We found that there was a shortcoming in how past languages focused on supporting standard 52-card deck based games. And though existing card game languages might be able to represent standard 52-card games reasonably, they fail to generalize to the full breadth of card games out there. Not only does our language allow the user to create any turn-based card game, but it also supports general-purpose programming. The goal of our object-oriented, Python, Ruby, and C++-inspired language is to enable programmers to easily code the gameplay and functionality of a turn-based card game with an emphasis on code readability and modularity.

2 Syntax and Features

2.1 Data Types

Primitive Data Type	**Description**
int	integers are positive or negative whole numbers without decimal points
float	floats represent real numbers written with a decimal point
String	strings are sequences of characters that handle textual data
f-String	formatted string literals using the syntax f'{expression}'
Boolean	boolean variables are defined by the `True` and `False` keywords

Object Types	**Description**
`Object`	Any non-primitive that has arbitrary mutable and immutable attributes
`Actor`	`Object` that can do `ACTION`s that mutate the attributes of more than just the object itself
`Range`	A set of values with a beginning and an end
`Collection`	A virtual class representing an iterable container called `Collection`
`Series`	Iterable `Collection` with a front (leftmost element) and a back (rightmost element)
`Stack`	Iterable `Collection` with a top and a bottom

2.2 Operators

Operator	**Description**
+,-,*,/,%,**,//	arithmetic operators
=,==, <, >, <=, >=	comparison operators
and, or, not, &, |, !,	logical operators
is, is not	identity operators
in, not in	membership operators

2.3 Keywords

The following are reserved keywords in AllHandsOnDeck:

`bool, float, int, True, False, None, string, const, not, let, be, do, if, elif, else, for, in, range, while, break, continue, times, return, when, with, new`

2.4 Control Flow

The following keywords are reserved for control flow: `if...elif...else, while, for...in`: works mostly like in Python, with inspiration from Ruby.

2.4.1 For Loops

A for loop is used to iterate over a sequence (like a `Collection`, a `Range`, or a `string`). With a for loop, we can execute a set of statements, once for each item in a given sequence.

```
for card in deck:
    print f'({card.type}, {card.color})'
```

For loops using the `times` keyword:

```
for 3 times:
    do INIT
```

Ranges are useful when a programmer wants to create a deck with type taken from a sequential set of values (can be numerical, lexicographical, etc.) without having to enumerate out the entire sequence themselves.

Ranges may be constructed using the s..e and s...e literals, where the former runs from the beginning of the interval to the end *inclusively* and the latter runs through the interval *excluding* the end value.

For loops over a Range:

```
for val in 1..9:
    card.type = val

for num in 0...players.size():
    players[num].turn = num
```

Nested for loops:

```
deck = new Deck
for type in [0] + 2 * (1..9 + ['Skip', 'Reverse', 'Draw 2']):
    for color in 'RYGB':
        deck do PUSH_BOTTOM(new Card(type, color, faceup: False))
```

For loops can also be rewritten as list comprehensions:

For example, the above nested loop can be rewritten as the following list comprehension:

```
deck = new Deck(
        new Card(type, color, faceup: False)
```

```
        for type in [0] + 2 *
            (1..9 + ['Skip', 'Reverse', 'Draw 2'])
        for color in 'RYGB'
    )
```

2.5 Comments

For single-line comments, the characters // are inserted at the beginning of the line. The compiler ignores all content between // and a new line. For multi-line comments, the characters /* and */ are used to surround the text to be commented out. The compiler ignores all content between /* and */.

```
// This is a comment

/*
This is how you can do
a multi-line
comment
*/

/* You can also just do one line */

/*
hand = [a, b, c] // you can also do a single line comment within a multi-line comment
deck = [d, e, f, g]

hand.push_front(deck.pop_bottom(3)) // deck.bottom(3) gives [g, f, e]

hand = [e, f, g, a, b, c]
deck = [d]
*/
```

2.6 Functions

Functions are denoted as ACTIONs in the AllHandsOnDeck language. What is of note is the difference between helper functions, which do not mutate state, and ACTIONs, which by definition mutate state. Thus, a function call like <Actor> do ACTION or <Object> do ACTION is distinct from a call like <Object>.helper_function().

AllHandsOnDeck encourages program modularity and code reuse through the way that main is intended to be a high-level description of the game being programmed. By requiring programmers of our language to wrap all state changes in an ACTION, main has to call those ACTIONs instead of defining them. Thus, main is a readable representation of what the gameplay entails for any game programmed using this language.

Functions can be defined as follows:

In the case of a general ACTION that is tied to the entire game and not to a specific entity, then the function is defined as when do ACTION, without a specified entity. For example, any initialization of the game setup may be done in such a function like INIT. See below for an example.

```
main:
    do INIT
    for 10 times:
```

```
        do ROUND_INIT
        // do rest of game

when do INIT:
    players = [Player() for 2 times]
    deck = Deck(
        Card(rank, suit, faceup: False)
        for rank in ['A'] + 2..10 + ['J','Q','K']
        for suit in 'CDHS'
    ).shuffled()

when do ROUND_INIT:
    for player in players:
        deck do PUSH_TOP(player.hand do CLEAR)

    deck do SHUFFLE

    while not deck.empty():
        players[0].hand do PUSH_BACK(deck do POP_TOP)
        players[1].hand do PUSH_BACK(deck do POP_TOP)
```

When an ACTION is tied to a specific Actor or Object, then the function signature should specify the entity (or the specific class of an entity) it is attached to.

Function definition in the case of an ACTION that is tied to a specific entity:

```
timer = Timer(100ms)
timer do START
when timer do DONE:
    print 'ping'
    timer do RESTART
```

Function definition in the case of an ACTION that is tied to the specific class of an entity:

```
when Player player do BET(amount: int):
    player.chips -= amount
    player.bet += amount
    betting_pot += amount
```

In the above example, the function BET describes the outcome of any Player performing the BET action.

2.7 Standard Library

The Collection object and the special Collection objects Stack and Series are built into the standard library. Both Stacks and Series are deques. A Stack can be thought of as a vertical list where the top element is index 0 and can be used to represent a deck of cards. The built-in methods for a Stack include PUSH_TOP(elements...), PUSH_BOTTOM(elements...), POP_TOP(num = 1), and POP_BOTTOM(num = 1). A Series can be thought of as a horizontal list where the left-most element is index 0 and a common usage is player's hand. The build-in methods for a Series include PUSH_FRONT(elements...), PUSH_BACK(elements...), POP_FRONT(num = 1), and POP_BACK(num = 1).

2.7.1 Built-in functions

- print '' prints the specified object to the screen after first converting it to a string
- input() asks the user for input
- <Collection> do SHUFFLE shuffles elements inside Collection
- <Collection>.shuffled() returns a copy of the shuffled Collection
- <Collection> do CLEAR empties the contents of the Collection and returns a copy of the Collection
- <Collection>.copy() returns a copy of the Collection
- <Collection>.empty() returns a boolean True or False of whether the Collection is empty
- <Collection>.size() returns the number of elements in the Collection
- <Stack> do PUSH_TOP(elements...): push 1 or more elements onto the top of a Stack
- <Stack> do PUSH_BOTTOM(elements...): push 1 or more elements to the bottom of a Stack
- <Stack> do POP_TOP(num = 1): pop 1 or more elements one at a time from the top of a Stack
- <Stack> do POP_BOTTOM(num = 1): pop 1 or more elements one at a time from the bottom of a Stack
- <Series> do PUSH_FRONT(elements...): push 1 or more elements to the front of a Series
- <Series> do PUSH_BACK(elements...): push 1 or more elements to the back of a Series
- <Series> do POP_FRONT(num = 1): pop 1 or more elements one at a time from the front of a Series
- <Series> do POP_BACK(num = 1): pop 1 or more elements one at a time from the back of a Series

2.8 Object-Oriented Programming

AllHandsOnDeck includes certain predefined base classes such as Object, Stack, Series, and Actor. Programmers are able to extend subclasses from those classes, with or without parameters. When instantiating a new object, the keyword new is used.

An Object entity can be defined as follows:

```
let Square(side) be Object with:
        side: side
        area(): side * side
```

Classes cannot have attribute-changing functions though. Therefore, the following would be invalid:

```
let Square(side) be Object with:
        side: side
        area(): side * side
        modify_side(new_side):
            side = new_side
```

In order to modify an attribute, the programmer must define an ACTION function outside of the class. In our above example, this can be done as follows:

```
when Square square do MODIFY_SIDE(new_side):
        square.side = new_side
```

An Actor entity can be defined as follows:

```
let Scissor be Actor with:
    int uses: 0

when Scissor scissor do CUT(target: Square):
    target do MODIFY_SIDE(target.side / 2)
    scissor.uses += 1
```

A Stack entity can be defined as follows:

```
let Deck be Stack(Card)
```

A Series entity can be defined as follows:

```
let Hand(owner: Player) be Series(Card) with:
    owner: owner
    uno(): size() == 1
    winner(): empty()
```

An object is instantiated as follows:

```
empty_deck = new Deck
deck = new Deck(
    new Card(1),
    new Card(2),
    new Card(3)
)
```

3 Sample Program: UNO

```
main:
    do INIT(4)

    do FIRST_PLAY

    while not player_won(): //define later
        if move_available():
            current_player do INPUT_PLAY_OR_DRAW //define later
        else:
            current_player do DRAW

    do PRINT_WINNER

let Card(type, color, faceup) be Object with:
    const type: type
    const color: color
    faceup: bool(faceup)

when Card card do FLIP:
    card.faceup = not card.faceup

when Collection(Card) cards do FLIP:
    for card in cards:
        card do FLIP

let Deck be Stack(Card)

let Hand be Series(Card)

let Player(name) be Actor with:
    const name: name
    hand: new Hand()
    uno(): hand.size() == 1
    winner(): hand.empty()

when do FIRST_PLAY:
    deck.top() do FLIP
    discard.push_top(deck.pop_top())
    do PROCESS_TOP_CARD

when Player player do PLAY(index):
    if not match(player.hand[index], discard.top()):
        return
    discard.push_top(player.hand.pop(index))
    do PROCESS_TOP_CARD

```

```
when Player player do DRAW:
    deck.top() do FLIP
    player.hand.push_back(deck.pop_top())

    if match(player.hand.back(), discard.top()):
        discard.push_top(player.hand.pop_back())
        do PROCESS_TOP_CARD

when do PROCESS_TOP_CARD:
    if discard.top().type == 'Reverse':
        do REVERSE
        do NEXT_PLAYER
    else:
        do NEXT_PLAYER

        if discard.top().type == 'Skip':
            do NEXT_PLAYER
        elif discard.top().type == 'Draw 2':
                deck.top(2) do FLIP
            current_player.hand.push_back(deck.pop_top(2))
            do NEXT_PLAYER

match(card1: Card, card2: Card):
    return card1.type == card2.type or card1.color == card2.color

when do REVERSE:
    play_dir *= -1

when do NEXT_PLAYER:
    if current_player is None:
        current_player_i = random(range(players.size()))
        current_player = players[current_player_i]
    else:
        current_player_i = (current_player_i + play_dir) % players.size()
        current_player = players[current_player_i]

when Player player do INPUT_PLAY_OR_DRAW:
    print 'Would you like to play or draw?'
    action = input()
    if action == 'play':
        print 'Which card?'
        int index = input()
        player do PLAY(index)
    elif action == 'draw':
        player do DRAW

when do INIT(n_players):
    players = [new Player(f'Player {i + 1}') for i in range(n_players)]
```

```
deck = new Deck(
    new Card(type, color, faceup: False)
    for type in [0] + 2 *
        (1..9 + ['Skip', 'Reverse', 'Draw 2'])
    for color in 'RYGB'
)

deck do SHUFFLE

for player in players:
    player.hand do PUSH_BACK(deck do POP_TOP(7))

discard = new Deck

current_player_i = None
current_player = None
play_dir = 1
```