AllHandsOnDeck

Language Reference Manual

Caitlyn Chen - *Language Guru*
Tiffeny Chen - *System Architect*
Jang Hun Choi - *System Architect*
Mara Dimofte - *Manager*
Christi Kim - *Tester*
{ckc2143, tc2963, jc5112, md3713, cwk2109}@columbia.edu

Contents

Chapter 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.

Chapter 2

Lexical Conventions

2.1 Tokens

There are six kinds of tokens: identifiers, keywords, comments, strings, expression operators, and other separators. AllHandsOnDeck employs Python-like indentation and uses whitespaces as separators.

2.2 Comments

For comments, the character # is inserted at the beginning of the line and is terminated by the newline character \n. The compiler ignores all content between a # and a new line.

```
# This is a comment

hand = ['a', 'b', 'c'] # this is another comment
deck = Stack('d', 'e', 'f', 'g')

hand do PUSH_FRONT(deck do POP_BOTTOM(3)) # deck.bottom(3) gives ['g', 'f', 'e']

hand = ['e', 'f', 'g', 'a', 'b', 'c']
deck = Stack('d')
```

2.3 Identifiers

Identifiers in AllHandsOnDeck are sequences of letters and digits, and underscores '_', where the first character must be a letter. Uppercase and lowercase letters are considered different. There are three kinds of identifiers: `ACTION`, `Class`, and `id`.

`ACTION` identifiers denote state mutating functions in the AllHandsOnDeck language and may consist of uppercase letters, digits, and underscores only.

```
ACTION:
    ('A'-'Z') ('A'-'Z' | '0'-'9' | '_')*
```

Identifiers for variables and helper functions (non-state mutating functions) are denoted by `id` and may consist of lowercase letters, digits, and underscores only.

```
id:
    ('a'-'z') ('a'-'z' | '0'-'9' | '_')*
```

`Class` identifiers denote classes, must start with an uppercase letter, and may consist of uppercase and lowercase letters and digits only.

```
Class:
    ('A'-'Z') ('a'-'z' | '0'-'9' | 'A'-'Z')*
```

2.4 Keywords

The following are reserved keywords in AllHandsOnDeck:

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

2.5 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

2.6 Operators

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

Chapter 3

Grammar

3.1 Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by typewriter font, characters are indicated as the character itself in quotation marks, and the `NEWLINE`, `INDENT`, `OUTDENT`, and `EOF` tokens are capitalized. The context free grammar is written in regex for the purpose of clarity, with the standard use of `|Pipe, ?Question Mark, *Asterisk, +Plus, -Hyphen,` and `()Parentheses`.

3.2 Types, Params, and Args

3.2.1 Types

AllHandsOnDeck supports two fundamental types: primitive types and classes.

```
type:
   prim_type | Class | template_Class
```

Primitives include integers, floating-point numbers, booleans, strings, and formatted strings.

```
prim_type:
   int | float | bool | string | fstring
```

Classes can be templated for a specific type.

```
template_Class:
   Class '<' type '>'
```

3.2.2 Params

`Params_list` consists of parameters and are used in class constructors and function definitions.

```
params_list:
   param (',' param)*
```

`Params` consist of variable and function identifiers and can have a specified type enforced.

```
param:
   type?  id
```

3.2.3 Args

`Args_list` consists of arguments and is used in specifying instances of classes and function calls.

```
args_list:
   arg (',' arg)*
```

`Args` consist of expressions and programmers can add on the name of the parameter they're providing an argument for.

```
arg:
   (id '=')?  expr
```

3.3 Program Structure

Program is the top-level node in the syntax tree. Since we parse bottom-up, all parsing must end here.

```
program:
   main_decl (action_decl | helper_decl | class_decl)* EOF
```

A program is made up of a main function, and any number of classes, `ACTION` functions, and helper functions. All programs written in AllHandsOnDeck must contain a main function.

3.4 Declarations

There are four different types of declarations that can be made: the main function, `ACTION` functions, helper functions, and classes.

3.4.1 Function Declarations

There are three types of functions: the `main` function, which runs the gameplay of the program, `ACTION`s, which are functions that mutate the gamestate, and helper functions, which are functions that do not mutate the gamestate and always have a return value.

The `main` function takes the form of the keyword `main` and : followed by `stmt_block`. A `stmt_block` is an indented block of statements where each `INDENT` token is paired with an `OUTDENT` token.

```
main_decl:
   main ':'  stmt_block

stmt_block:
    NEWLINE INDENT stmt+ OUTDENT
```

The main function is intended to be a high-level, readable representation of what the gameplay entails for any game programmed using AllHandsOnDeck. Programmers of our language are required to wrap all state changes in an `ACTION`, which means that main has to call those `ACTION`s in the statement block instead of

defining them.

Sample main function:

```
main:
    do INIT
    for 10 times:
        do ROUND_INIT
        # do rest of game
```

ACTIONs are declared with the `when...do ACTION` structure and : followed by `stmt_block`. `ACTION` function declarations can specify whether the `ACTION` is tied to a specific class of an object by indicating the type and id of the object that the `ACTION` is tied to in a `when type id do ACTION` structure. Programmers can also specify any params the `ACTION` should take in, and have the choice of enforcing a specific type.

```
action_decl:
    when (type id)?  do ACTION ('(' params_list ')')?  ':'  stmt_block
```

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.

```
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 specific class of the object it is attached to. In the following example, the function `BET` describes the outcome of any Player performing the `BET` action.

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

Helper functions are declared with the `id` of the function and : followed by `stmt_block`. Helper functions can take in params, which may or may not have an enforced type.

```
helper_decl:
    id '(' params_list?  ')'  ':'  (expr | stmt_block)
```

The following helper function `match(Card card1, Card card2, Card card3)` determines if a group of three cards is a valid set according to the rules of SET.

```
# having param type be optional supports methods where params can have different types
# the following helper function is able to allow for both string type and int type attributes

match_attribute(attribute1, attribute2, attribute3):
    return (attribute1 == attribute2 == attribute3) \
    or ((attribute1 != attribute2) and (attribute2 != attribute3) and (attribute1 != attribute3))

match(Card card1, Card card2, Card card3):
    return match_attribute(card1.number, card2.number, card3.number) \
    and match_attribute(card1.shape, card2.shape, card3.shape) \
    and match_attribute(card1.shading, card2.shading, card3.shading) \
    and match_attribute(card1.color, card2.color, card3.color)
```

3.4.2 Class Declarations

`Class` declarations follow the `let Class be type...` structure, where `Class` is the new class being created and `type` is the super class the new class is extending. AllHandsOnDeck includes certain predefined base classes such as Object, Actor, Stack, and Series. Programmers are able to extend subclasses from those classes, with or without parameters, by indicating the list of parameters for the new Class and by specifying the arguments for a specific instance of the super Class. To extend the super class, the `with : class_block` structure is used to specify the attributes and helper functions for the new class .

A `class_block` is an indented block following the same indentation pattern as `stmt_block` and consists of attributes and helper function declarations. Helper functions can be declared in one-liners or multi-liners, as previously illustrated, and attributes are declared with an `id` and : followed by an expression or `stmt_block`. Programmers have the option of specifying the type of the attribute, along with whether or not the attribute's value should be immutable by using the `const` keyword.

```
class_decl:
   | let Class be type (with ':'  class_block)?
   | let Class '(' params_list?  ')'  be type '(' args_list?  ')'  (with ':'  class_block)?

class_block:
    NEWLINE INDENT (attr_decl | helper_decl)+ OUTDENT

attr_decl:
    const?  type?  id ':'  (expr | stmt_block)
```

The following examples illustrate some different use cases and what is and isn't possible for class declarations using AllHandsOnDeck's predefined base classes.

An `Object` entity can be defined as follows:

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

Classes may only contain attributes and non-state mutating functions, and so cannot have attribute-changing functions, which must be wrapped within `ACTION`s. 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(Player owner) be Series<Card> with:
    owner: owner
    uno(): size() == 1
    winner(): empty()
```

An object can be instantiated as follows:

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

3.5 Statements

Statements, unless noted otherwise, are executed in sequence.

```
stmt:
    expr NEWLINE | pass NEWLINE | if_stmt | for_stmt | while_stmt | return_stmt
```

3.5.1 Expression statement

Most statements are expression statements, usually assignments or function calls, and take the form of an expression followed by a `NEWLINE` token.

```
expr NEWLINE
```

3.5.2 Pass statement

In AllHandsOnDeck, the `pass` statement is a null statement. It is different from a comment in that while the interpreter ignores comments entirely, `pass` is not ignored.

```
pass NEWLINE
```

3.5.3 Conditional statement

The code within an `if...elif...else` block will be executed if the result of the test expression in the if statement evaluates to `True`. If the test expression is `False`, the `stmt_block` will not be executed. AllHandsOnDeck interprets non-zero values as `True`, and `0` and `None` as `False`.

```
if_stmt:
   if expr ':'  stmt_block elif_stmt | if expr ':'  stmt_block else_block?

elif_stmt:
   elif expr ':'  stmt_block elif_stmt | elif expr ':'  stmt_block else_block?

else_block:
   else ':'  stmt_block
```

3.5.4 While statement

The code within a `while` block will be executed repeatedly as long as the evaluation of the test expression in the while statement evaluates to `True`.

```
while_stmt:
   while expr ':'  stmt_block else_block?
```

3.5.5 For statement

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

```
for_stmt:
   | for id in expr ':'  stmt_block
   | for expr times ':'  stmt_block
```

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

For loops using the `times` keyword:

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

For loops can be nested:

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

3.5.6 Return statement

`ACTION`s and helper functions return to their callers by means of the `return` statement, which either returns no value or returns the value of the specified expression to the caller of the function.

```
return_stmt:
   return expr?  NEWLINE
```

3.6 Expressions

Expressions are sequences of operands and operators and are meant to be evaluated.

```
expr:
   | id | None | neg_expr | iliteral | fliteral | sliteral | bliteral | Series_literal
   | binary_op | comparison | assignment | augassign | call_class | call_helper | call_action
   | dotted_range | comprehension | index | slice
```

3.6.1 Identifiers

Identifiers denote names of variables, functions, and classes in AllHandsOnDeck. Refer to section 2.3 of Chapter 2 for more details.

3.6.2 None

The `None` keyword is used to define null objects and variables.

3.6.3 Literals

There are five kinds of literals in AllHandsOnDeck: integer literals, floating-point literals, string literals, boolean literals, and Series literals.

Integer and floating-point literals are immutable.

```
iliteral:
    ('0'-'9')*

fliteral:
    ('0'-'9')* '.'  ('0'-'9')* (('e' | 'E')('+' | '-')?('0'-'9')*)?
```

String literals are sequences of characters surrounded by single quotes or double quotes.

```
sliteral:
    ('"' _* '"') | (''' _* ''')
```

A boolean literal can have either the `True` or `False` value.

```
bliteral:
    True | False
```

A `Series` literal is a representation of a `Series` in AllHandsOnDeck. Alternatively, a programmer can call the `Series` class to instantiate a Series object.

```
Series_literal:
   '[' (expr (',' expr)*)?  ']'
```

3.6.4 Negation

The `not` keyword is a logical operator and the return value will be `True` if the statements are not `True`, and wille be `False` otherwise.

```
neg_expr:
   not expr
```

3.6.5 Operations

The following binary operations are supported by AllHandsOnDeck:

```
binary_op:
   expr ("+" | "-" | "*" | "/" | "/" | "%" | "&" | "|" | "^" | "~") expr
```

3.6.6 Comparison

Comparisons yield boolean values: `True` or `False` and can be chained arbitrarily. All comparison operators have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation.

```
comparison:
   expr ("==" | "!=" | "<" | "<=" | ">" | ">=" ) expr
```

3.6.7 Assignment

An assignment expression assigns an expression to an identifier, while also returning the value of the expression.

```
assignment:
   id "=" expr

augassign:
   id ("+=" | "-=" | "*=" | "/=" | "/=" | "**=" | "%=" | "&=" | "|=" | "^=") expr
```

3.6.8 Class Call

Classes can be called with or without a constructor and take the structure of Class name followed by optional arguments in parens.

```
call_class:
   Class '(' args_list?  ')'
```

3.6.9 Function Call

Calls to `ACTION`s have different syntax from calls to helper functions in AllHandsOnDeck.

A call to a non-state mutating helper function follows the structure of function name followed by any arguments in parentheses. A helper function may be called by itself or on an object.

```
call_helper:
   id '(' args_list?  ')'
```

State mutating `ACTION` functions are called following the structure of do `ACTION` or object do `ACTION`.

```
call_action:
   expr?  do ACTION ('(' args_list ')')?
```

3.6.10 Comprehension

Comprehensions on `Series` and `Stacks` allow for shorter syntax when a programmer wants to create a new list based on the values of an existing list.

```
comprehension:
   expr for id in expr
```

Without comprehension on `Collections`, the following example would have to be written as a nested for loop:

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

3.6.11 Range

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.

```
dotted_range:
   expr ".." expr | expr "..." expr
```

For loops can be iterated over a Range as follows:

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

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

3.6.12 Indexing

Slicing and indexing can be done following the structure `expr[index]` or `expr[index:index]`.

```
index:
   '[' slice ']'
```

```
slice:
   expr?  ':'  expr?  (':'  expr)?  | expr
```

3.7 Context-Free Grammar

```
program:
   main_decl (action_decl | helper_decl | class_decl)* EOF

main_decl:
   main ':'  stmt_block

class_decl:
   | let Class be type (with ':'  class_block)?
   | let Class '(' params_list?  ')'  be type '(' args_list?  ')'  (with ':'  class_block)?

action_decl:
    when (type id)?  do ACTION ('(' params_list ')')?  ':'  stmt_block

helper_decl:
    id '(' params_list?  ')'  ':'  (expr | stmt_block)

attr_decl:
    const?  type?  id ':'  (expr | stmt_block)

stmt_block:
    NEWLINE INDENT stmt+ OUTDENT

class_block:
    NEWLINE INDENT (attr_decl | helper_decl)+ OUTDENT

type:
   prim_type | Class | template_Class

prim_type:
   int | float | bool | string | fstring

template_Class:
   Class '<' type '>'

params_list:
   param (',' param)*

param:
   type?  id

args_list:
   arg (',' arg)*
```

```
arg:
   (id '=')?  expr

stmt:
   expr NEWLINE | pass NEWLINE | if_stmt | for_stmt | while_stmt | return_stmt

if_stmt:
   if expr ':'  stmt_block elif_stmt | if expr ':'  stmt_block else_block?

elif_stmt:
   elif expr ':'  stmt_block elif_stmt | elif expr ':'  stmt_block else_block?

else_block:
   else ':'  stmt_block

for_stmt:
   | for id in expr ':'  stmt_block
   | for expr times ':'  stmt_block

while_stmt:
   while expr ':'  stmt_block else_block?

return_stmt:
   return expr?  NEWLINE

expr:
   | id | None | iliteral | fliteral | sliteral | bliteral | Series_literal | neg_expr
   | binary_op | comparison | assignment | augassign | call_class | call_helper | call_action
   | dotted_range | comprehension | index | slice

iliteral:
    ('0'-'9')*

fliteral:
    ('0'-'9')* '.'  ('0'-'9')* (('e' | 'E')('+' | '-')?('0'-'9')*)?

sliteral:
    ('"' _* '"') | (''' _* ''')

bliteral:
    True | False

Series_literal:
   '[' (expr (',' expr)*)?  ']'

neg_expr:
   not expr

binary_op:
   expr ("+" | "-" | "*" | "/" | "/" | "%" | "&" | "|" | "^") expr
```

```
comparison:
   expr ("==" | "!=" | "<" | "<=" | ">" | ">=") expr

assignment:
   id "=" expr

augassign:
   id ("+=" | "-=" | "*=" | "/=" | "/=" | "**=" | "%=" | "&=" | "|=" | "^=") expr

call_class:
   Class '(' args_list?  ')'

call_helper:
   id '(' args_list?  ')'

call_action:
   expr?  do ACTION ('(' args_list ')')?

dotted_range:
   expr ".." expr | expr "..." expr

comprehension:
   expr for id in expr

index:
   '[' slice ']'

slice:
   expr?  ':'  expr?  (':'  expr)?  | expr

id:
    ('a'-'z') ('a'-'z' | '0'-'9' | '_')*

ACTION:
    ('A'-'Z') ('A'-'Z' | '0'-'9' | '_')*

Class:
    ('A'-'Z') ('a'-'z' | '0'-'9' | 'A'-'Z')*
```

Chapter 4

Standard Library

4.1 Built-in classes

`Actor`, `Object`, and `Collection` entities are all predefined classes in AllHandsOnDeck that a programmer can use to define their own classes and objects.

`Actor` entities are distinct from `Object` entities in that `Actor`s can mutate the attributes of other `Object`s but `Object`s cannot.

`Collection`s are virtual classes and `Stack`s and `Series` are special `Collection` objects that are also built into the standard library. Both `Stack`s and `Series` are deques.

A `Stack` can be thought of as a vertical list where the top element is index 0 and the bottom element is index -1 and can be used to represent a deck of cards. In addition to the methods inherited from `Collection`, the built-in methods for a `Stack` include helper functions like `top()` and `bottom()`, and `ACTION`s like `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 front element is index 0 and the back element is index -1 and can be used to represent a player's hand. In addition to the methods inherited from `Collection`, the built-in methods for a `Series` include helper functions like `front()` and `back()`, and `ACTION`s like `PUSH_FRONT(elements...)`, `PUSH_BACK(elements...)`, `POP_FRONT(num = 1)`, and `POP_BACK(num = 1)`.

4.2 Built-in functions

- `print()` prints the specified object to the screen after first converting it to a string
- `input()` asks the user for input
- `random(Range)` returns a random integer or floating-point number based on the type and values of the starting and ending elements in a specified range.
- `<Collection> do SHUFFLE` shuffles elements inside the Collection
- `<Collection> do INSERT(index,elements...)` inserts 1 or more elements at a specified index inside the Collection

- `<Collection> do DELETE(slice)` deletes elements at a specified index or slice of the 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>.top()` returns the top element of the Stack
- `<Stack>.bottom()` returns the bottom element of the Stack
- `<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 and returns the elements
- `<Stack> do POP_BOTTOM(num = 1)`: pop 1 or more elements one at a time from the bottom of a Stack and returns the elements
- `<Series>.front()` returns the front element of the Series
- `<Series>.back()` returns the back element of the Series
- `<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 and returns the elements
- `<Series> do POP_BACK(num = 1)`: pop 1 or more elements one at a time from the back of a Series and returns the elements

Chapter 5

Sample Program

The following implementation of UNO in AllHandsOnDeck showcases most of the features of the language.

```
main:
    do INIT(4)

    do FIRST_PLAY

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

    do PRINT_WINNER # define later

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

```
35
36  when do FIRST_PLAY:
37      deck.top() do FLIP
38      discard do PUSH_TOP(deck do POP_TOP)
39      do PROCESS_TOP_CARD
40
41  when Player player do PLAY(index):
42      if not match(player.hand[index], discard.top()):
43          return
44      discard do PUSH_TOP(player.hand do DELETE(index))
45      do PROCESS_TOP_CARD
46
47  when Player player do DRAW:
48      deck.top() do FLIP
49      player.hand do PUSH_BACK(deck do POP_TOP)
50
51      if match(player.hand.back(), discard.top()):
52          discard do PUSH_TOP(player.hand do POP_BACK)
53          do PROCESS_TOP_CARD
54
55  when do PROCESS_TOP_CARD:
56      if discard.top().type == 'Reverse':
57          do REVERSE
58          do NEXT_PLAYER
59      else:
60          do NEXT_PLAYER
61
62          if discard.top().type == 'Skip':
63              do NEXT_PLAYER
64          elif discard.top().type == 'Draw 2':
65              deck.top(2) do FLIP
66              current_player.hand do PUSH_BACK(deck do POP_TOP(2))
67              do NEXT_PLAYER
68
69  match(Card card1, Card card2):
70      return card1.type == card2.type or card1.color == card2.color
71
72  when do REVERSE:
73      play_dir *= -1
74
75  when do NEXT_PLAYER:
76      if current_player is None:
77          current_player_i = random(0...players.size())
78          current_player = players[current_player_i]
79      else:
80          current_player_i = (current_player_i + play_dir) % players.size()
81          current_player = players[current_player_i]
82
83  when Player player do INPUT_PLAY_OR_DRAW:
84      print('Would you like to play or draw?\n')
85      action = input()
86      if action == 'play':
```

```
87          print('Which card?\n')
88          int index = input()
89          player do PLAY(index)
90      elif action == 'draw':
91          player do DRAW
92
93  when do INIT(n_players):
94      players = [Player(f'Player {i + 1}') for i in 0...n_players]
95
96      deck = Deck(
97          Card(type, color, faceup = False)
98          for type in [0] + 2 *
99              (1..9 + ['Skip', 'Reverse', 'Draw 2'])
100         for color in 'RYGB'
101     )
102
103     deck do SHUFFLE
104
105     for player in players:
106         player.hand do PUSH_BACK(deck do POP_TOP(7))
107
108     discard = Deck()
109
110     current_player_i = None
111     current_player = None
112     play_dir = 1
```