CS 334 Project Proposal V2

Markus Feng

Due 2018-11-15

1 Introduction

The goal of this language is to describe a subset of turn-based card games, such that upon being interpreted, the user will be able to play the described game, pass-and-play style on the same machine. This language would allow users to describe more card games and create new card games much more easily than programming a whole game or modifying a large amount of code simply to support another game.

This problem needs its own programming language because many card games share large amounts of similar characteristics, but differ enough such that simple data structures are not enough to represent the difference in rules. The hope is that, with this language, the similar parts of these card games can be abstracted away, while the differences can be accounted for with programming language constructs and expressions.

2 Design Principles

The guiding design principle behind this design is "reducing repeated code", while being relatively simple to read and write. This is because the "reducing repeated code" aspect is the biggest selling point in using this language rather than a more complex standalone implementation of a card game, and the relative simplicity to read and write allows this language to have a wider audience.

3 Examples

The following example is an implementation of the card game War, subject to change:

```
1  #Information about the card game
2  #Interpreter uses this to figure out how the game should be run
3  info:
4      name = "War"
5      players = 2
6      deck = standard_52
7      turn = simultaneous_turns
8      init = deal_evenly_to_hands
9      winner = highest_score
10     hand = unseen_stack
11
12 down_pile = empty_pile()
13
14 #Player block indicates to search in the specific player's methods
15 #when calling functions, implicitly passing the player in as
16 #an argument. This is  to how non-static methods work in a language
17 #like Java.
```

```
player:
    up_pile = empty_pile_ordered()
    owned = empty_pile()

    #Function that gets called every player's turn
    turn():
        #If last round was tied, play an extra card face down
        if len(up_pile) > 0:
            let c1 = request_top_card(hand)
            move(down_pile, c1)
        #Play a card face up
        let c2 = request_top_card(hand)
        move(up_pile, c)

    #Helper function
    up_pile_getter() default error:
        return up_pile

    #Function that gets called and the end of each round
    end_round():
        #Compare the top cards to choose a winner
        let up_piles = map(players, up_pile_getter)
        let top_cards = map(up_piles, request_top_card)
        let top_card = max(top_cards, value)
        if len(top_card) == 1:
            let winner = top_card.get(0).played_by
            move_all(winner.owned, down_piles)
            move_all(winner.owned, up_piles)

    #Function that gets called at the end of the game
    #to determine the winner
    calculate_score() default 0:
        return len(owned)
```

The following more complicated example is an implementation of the card game Hearts, subject to change:

```
info:
    name = "Hearts"
    players = 4
    deck = standard_52
    turn = turn_with_head
    init = deal_evenly_to_hands
    winner = lowest_score

base = ace of spades
hearts_played = false

#Value of card when played to compare who wins the trick
playing_value(base, card) default 0:
    if card.suit == base.suit:
        return card.value

#Value of card when scoring
scoring_value(card) default 0:
    if card is hearts:
        return 1
    elif card is queen of spades:
        return 13
```

```
23
24  move_to_pile(card):
25      if (card is hearts) or (card is queen of spades):
26          hearts_played = true
27      move(pile, card)
28
29  #Disallow hearts from being played if none have been discarded yet
30  any_heart_condition(card) default true:
31      if not hearts_played:
32          return card isnt hearts
33
34  positive_playing_value(card) default false:
35      return playing_value(base, card) > 0
36
37  player:
38      is_starting() default false:
39          if hand contains two of clubs:
40              return true
41
42      turn():
43          if is_head:
44              base = request_one_card(hand, any_heart_condition)
45              move_to_pile(base)
46          else:
47              let c = request_one_card(hand, positive_playing_value)
48              move_to_pile(c)
49
50      end_turn():
51          #Guaranteed one winner per round
52          let winner = max(pile, v1).get(0).played_by
53          move_all(winner.owned, pile)
54          set_head(winner)
55
56      calculate_score() default 0:
57          return sum_by(owned, v2)
```

4 Language Concepts

The user needs to understand the core game loop of a card game, along with imperative programming concepts. Because the structure of this language is similar to any other type of imperative language such as `python` or `java`, the language has variables and methods, control flow such as `if`/`else`, and types such as `int`, `boolean`, `string`, `array`. In this case, variables act as primitives, while methods, control flow, and complex objects act as combining forms.

Note that the `for`/`while` constructs are not supported, nor is recursion, to ensure that the program halts eventually. Instead the `repeat(n)` construct is supported, but only for constant `n`.

5 Syntax

The goal is for the language to have a `python`-like whitespace based syntax. Therefore, indentation matters in such a language, so we use a `[<expr>+1]` to indicate that everything inside <expr¿ (other than newlines) is indented one additional time. In addition, any non-quoted "#" character begins a comment, so that

character and all successive characters of the same line are ignored. The BNF of the currently implemented grammar looks like the following (in progress):

```
<program> ::= <scope> <newlines> <scope> | <scope>
<newlines> ::= \n | <newlines> \n
<scope> ::= <fun_def> | <var_decl> | <named_scope>
<named_scope> ::= <id> : <newlines> [<program>+1]
<var_decl> ::= <id> | <assignment>
<fun_def> ::= <fun_header> <stmt_block>
<stmt_block> ::= ":" <newlines> [<expr_body>+1]
<fun_header> ::= <fun_name_args>
<fun_name_args> ::= <id> (<fun_args>)
<fun_args> ::= <id> | <fun_args>, <id>
<expr_body> ::= <stmt> <newlines> <stmt> | <stmt>
<stmt> ::= <var_local>
    | <assignment>
    | <fun_call>
    | <return_stmt>
<var_local> ::= let <assignment>
<assignment> ::= <id> = <expr>
<return_stmt> ::= return <expr>
<expr> ::= <const_value>
    | <fun_call>
<fun_call> ::= <id> (<expr_args>) | <id> ()
<expr_args> ::= <expr>, <expr_args> | <expr>
<const_value> ::= <id> | <literal_value>
<literal_value> ::= <num_literal>
    | <string_literal>
<string_literal> ::= "" | "<chars>"
<num_literal> ::= <num_literal> <num> | <num>
<id> ::= <id> <idchar> | <idchar>
<idchar> ::= <loweralpha> | <upperalpha> | <num> | <underscore>
<loweralpha> ::= alpha in {a, b, ..., z}
<upperalpha> ::= alpha in {A, B, ..., Z}
<underscore> ::= _
<num> ::= num in {0, 1, ..., 9}
<stringchars> ::= <stringchar> <stringchars> | <stringchar>
<stringchar> ::= \ <char> | <nonquote>
<char> ::= alpha in Unicode characters
<nonquote> ::= alpha in Unicode character that are not a double quote
```

6 Semantics

The program is interpreted in order, line-by-line, with multi-line clusters forming a scope. Due to the inspiration from a `python`-like whitespace based syntax, indentation is used to define blocks, whereas line breaks are used to separate statements.

The interpreter will throw a syntax error if the result of the parsing is invalid. If a line break is reached before a statement can be terminated, the interpreter continues reading to the next line, throwing a syntax error if the next line is not indented.

In this language, the primitive values are similar to the primitive values in other languages. These include `int`, `boolean`, `string`, `array`.

For compositional elements, the language has methods, control flow, and complex objects act as combining forms. Specifically, this language include some aspects of card games that are common among many of them.

They are implemented in a pseudo-object-oriented way, acting as a collection of attributes and methods, some of which may be overridden. These include the `Ruleset` type, `Player` type, and the `Card` type. In addition, a large collection of library methods will be available to use, and users can define their own methods. Currently, no support is planned for custom object types.

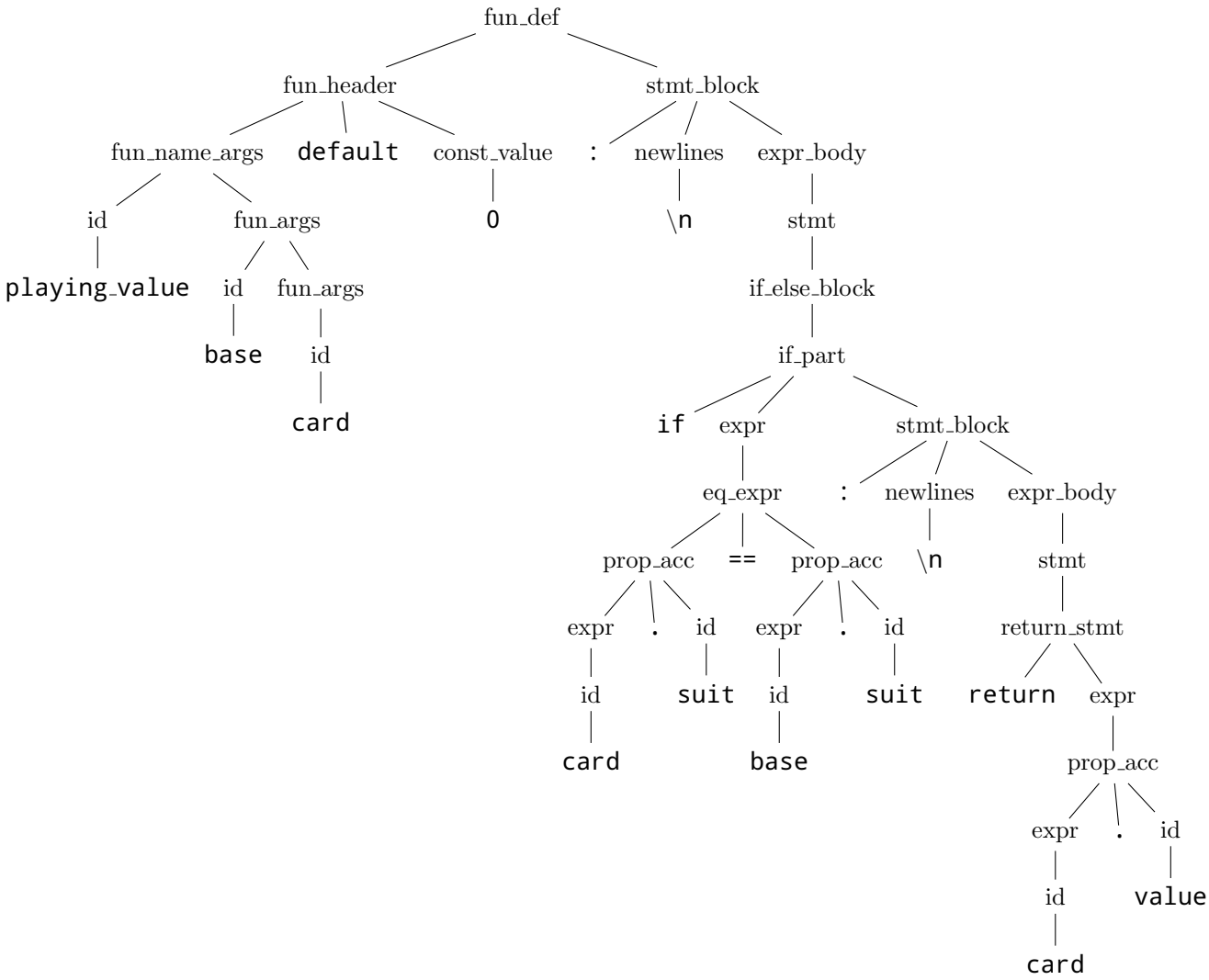
My program will be represented in an abstract syntax tree (AST) fashion in an algebraic data type. For example, the following code excerpt:

```
playing_value(base, card):
    if card.suit == base.suit:
        return card.value
```

Would be stored in the following alegbraic data type structure, subject to change:

```
Function("playing_value", ["base", "card"],
    [IfStmt(
        EqExpr(
            PropertyAccessorExpr("card", "suit")
            PropertyAccessorExpr("base", "suit")
        )
        [ReturnStmt(
            PropertyAccessorExpr("card", "value")
        )]
    )]
)
```

This same code excerpt would be represented in the abstract syntax tree structure in the following form (ignoring indentation for now).



This language is currently planned to support lexical scoping and strong dynamic typing, due to it being an interpreted language. Any error that occurs during execution will cause the program to halt with an error message describing the problem.

The program, upon being interpreted, will generate an interactive card game that can be played in the command line, pass-and-play style on the same machine. Therefore, they will read input when necessary to make the decisions in the card game, and give output describing what happens in the card game. Similarly, the evaluation would be too complicated for the example programs described earlier, but essentially, it would go read the entire program, parse it into AST form, and run a "abstract game" with "hooks" set up to call methods of the program, depending on the `info` block specified by the program. Some of the functions the program calls will be built-in functions. These built-in will be implemented in the project itself (instead of in the programming language).

For now, because the card game loop has not been implemented yet, the program runs a "main" function and exits immediately after. Currently the only built-in function that has been implemented is the "print" function, but more will be added as time progresses.