Codeable

A highly-readable programming language designed for software newcomers.

github.com/jhale1805/codeable

Created by

Rithvik Arun
Joseph Hale
Jacob Hreshchyshyn
Jacob Janes
Sai Nishanth Vaka

In partial fulfillment of the requirements for

SER502
Programming Languages and Paradigms
by Dr. Ajay Bansal

in the
Spring 2022 Semester

at
Arizona State University

1 Grammar

```
<!--
 Copyright (c) 2022 Rithvik Arun, Joseph Hale, Jacob
    Hreshchyshyn, Jacob Janes, Sai Nishanth Vaka
 This software is released under the MIT License.
 https://opensource.org/licenses/MIT
-->

// Initial Definitions
`identifier` := \w+

`command` := `comment`
`command` := `assignment`
`command` := `loop`
`command` := `show`
`command` := `selection`
`command` := `command` `command`

`program` := `command`


// Requirement 1a
`boolean` := true
`boolean` := false
`boolean` := not `boolean`
`boolean` := `expression` and `expression`
`boolean` := `expression` or `expression`
`boolean` := `expression` is_greater_than `expression`
`boolean` := `expression` is_less_than `expression`
`boolean` := `expression` equals `expression`

// Requirement 1b
`number` := [0-9]+(\.[0-9]+)?

`expression` := `number`
`expression` := `identifier`
`expression` := `expression` plus `expression`
`expression` := `expression` minus `expression`
`expression` := `expression` times `expression`
`expression` := `expression` divided-by `expression`

// Requirement 1c
`word` := `identifier`
`word` := `identifier` `word`
```

```
‘string‘ := < ‘word‘ >

// Requirement 2
‘assignment‘ := ‘identifier‘ stores ‘expression‘
‘assignment‘ := ‘identifier‘ stores ‘string‘
‘assignment‘ := ‘identifier‘ stores ‘ternary‘

// Requirement 3a
‘ternary‘ := ‘expression‘ if ‘boolean‘ otherwise ‘
   expression‘

// Requirement 3b
‘if_statement‘ := if ‘boolean‘ ‘command‘ move–on
‘if_statement‘ := if ‘boolean‘ ‘command‘ otherwise ‘
   command‘ move–on

// Requirement 4a
‘loop_for‘ := for ‘identifier‘ from ‘expression‘ to ‘
   expression‘ by ‘expression‘ ‘command‘ repeat

// Requirement 4b
‘loop_while‘ := while ‘boolean‘ ‘command‘ repeat

// Requirement 4c
‘loop_for‘ := for ‘identifier‘ from ‘expression‘ to ‘
   expression‘ ‘command‘ repeat

// Requirement 5
‘print‘ := show ‘string‘
‘print‘ := show ‘number‘
‘print‘ := show ‘identifier‘

// Other Features: Comments
‘comment‘ := fyi ‘string‘ \n
```

2 Implementation Tooling

2.1 Tokenizer

In our tokenizer, we will generate a tokenized string for each line of Codeable code, and will do this by utilizing our Javascript front end to pull each line as a new line of Codeable text. After generating the tokenized text, we will send all lines to our Prolog DCG rule interpreter for parsing and returning a tree for our subtasks.

2.2 Parser

For our parser, we will be utilizing Prolog and tokenized text in order to generate parse trees. As such, Prolog will be used as our interpreter, and this will be done with a set of Definite Clause Grammar rules and a Recursive Descent Parser. In order to make this as easy as possible, we are going to utilize Prolog rules to extract the grammar rules we have laid out, and provide quick and accurate trees that represent the functions provided by a user of Codeable.

2.3 Evaluator

After receiving a parse tree from Prolog, our system shall evaluate the parse tree and perform the calculations on those interactions based on what the user has provided in the Codeable language, or a false response if the user form is not valid or cannot be evaluated. In this approach, we can simplify the evaluation process as we already have the rules in Prolog, and will reduce the complexity of parsing a parse tree with limited context of the grammar rules.

2.4 Runtime

Writing programs in Codeable and running them can occur anywhere that a Prolog installation is available. In line with our hopes to make this language as easy to use as possible for software newcomers, we will create a website-based playground for writing Codeable programs.

To accomplish this, we will leverage the open-source JavaScript library Tau-Prolog which enables parsing and evaluating Prolog code in the browser. We will have a text box for writing the Codeable instructions and an output window. Behind the scenes, the Prolog rules that constitute the Tokenizer, Parser, and Evaluator described previously will be fed into Tau-Prolog to process the user's Codeable instructions and return the output to be rendered in the webpage.

2.5 Data Structures

2.5.1 Array

One of the data structures that will be used by the parser/interpreter will be arrays. Arrays will be useful to group a fixed amount of data and run operations

on that data. Arrays will follow common structure with indices starting at 0, and supporting only one data type at a time.

2.5.2 Lists

Another data structures that will be used by the parser/interpreter will be lists. Lists will be useful for ordering dynamic data. In other words, if the data does not have a fixed amount, lists will provide an easy way to add and remove data on the fly. The list will follow a common structure with functionality such as GET, REMOVE, CONTAINS, REMOVE_BY_VALUE, and ADD.

2.5.3 Trees

The last data structure that will be used by the parser/interpreter will be trees. Trees will be useful to group data that have hierarchical relationships. With trees we can also run many traversals such as BFS and DFS to search for data that we might want. Trees will follow a binary pattern where each parent node can have 0,1 or 2 child nodes.

3 Contribution Plan

Task	Assigned To
Encode Boolean values in runtime environment	Sai V
Encode numeric type in runtime environment	Sai V
Encode string support in runtime environment	Rithvik
Encode assignment support in runtime environment	Sai V
Encode ternary operator in runtime environment	Jacob J
Encode if-then-else construct in runtime environment	Jacob J
Encode for loop in runtime environment	Jacob H
Encode while loop in runtime environment	Jacob H
Encode syntactic sugar representation of for loop (e.g. for i in range(2, 5))	Jacob H
Encode print construct	Rithvik
Set up runtime environment	Joseph
Encode data structures in runtime environment (if time allows)	Jacob J
Set up interpreter prolog rules to feed into Tau prolog	Jacob J
Set up compiler/interpreter environment with Tau	Joseph
Create PDF of language presentation	Joseph
Create and upload YouTube video of language presentation.	Rithvik

4 Example Programs

4.1 Quadratic Formula

```
fyi < Rithvik Arun, Joseph Hale, Jacob Hreshchyshyn,
   Jacob Janes, Sai Nishanth Vaka >
fyi <  >
fyi < This software is released under the MIT License. >
fyi < https://opensource.org/licenses/MIT >

fyi < quadratic formula >
a stores 1
b stores -2
c stores 1

b_squared stores b times b
four_a stores 4 times a
four_a_c stores four_a times c
discriminant stores b_squared minus four_a_c
sqrt_discriminant stores discriminant raised_to 0.5
negative_b stores 0 minus b
numerator stores negative_b plus sqrt_discriminant
denominator stores 2 times a
root stores numerator divided_by denominator

show root
```

4.2 Factorial

```
fyi < Rithvik Arun, Joseph Hale, Jacob Hreshchyshyn,
   Jacob Janes, Sai Nishanth Vaka >
fyi <  >
fyi < This software is released under the MIT License. >
fyi < https://opensource.org/licenses/MIT >

fyi < factorial >
fyi < example of while loop block >

n stores 4

value stores 1
temp stores n

while temp is_greater_than 1
    temp2 stores value
    value stores temp times temp2
```

```
    temp3 stores temp
    temp stores temp3 minus 1
repeat

show value
```

4.3 Fibonacci

```
fyi < Rithvik Arun, Joseph Hale, Jacob Hreshchyshyn,
   Jacob Janes, Sai Nishanth Vaka >
fyi <  >
fyi < This software is released under the MIT License. >
fyi < https://opensource.org/licenses/MIT >

fyi < fibonacci >
fyi < example of if statements >

n stores 2

fib stores n minus 1 if n is_less_than 3 otherwise -1

if fib is_less_than 0

    iterator stores 1

    fyi < store the first two values of the fibonacci
       sequence >
    anteprev_n stores 0
    prev_n stores 1

    fyi < compute fibonacci numbers until n is reached >
    while iterator is_less_than n minus 1
        fib stores anteprev_n plus prev_n
        anteprev_n stores prev_n
        prev_n stores fib
        iterator stores iterator plus 1
    repeat

otherwise

    message stores < try a higher value of n for more fun
        >
    show message

move_on
```

```
show fib
```