CABG (Cabbage) Programming Language

Group Members:
Brody Berg (bcb2115)
Shaina Graboyes (seg2122)
Max Czapanskiy (mfc2105)
Raphael Almeida (raa2126)

Introduction:
CABG is a language that allows the creation of states and the simulation of a Definite Finite Automata (DFA) to recognize ASCII strings. The programmer defines states, conditions, effects and transitions. An emphasis is placed on ease of state definition and each of the conditions within states.

Motivation:
States and transitions are important concepts in Computer Science as well as in other disciplines. Many fundamental algorithms in networking, context-free grammars and regular expressions can be expressed through states and transitions. CABG seeks to provide an unobtrusive language for the quick and accurate creation of DFA's so that they may be simulated with little time investment.

It is entirely conceivable that this language could be used to simulate DFA's for something other than the recognition of strings. This would be supported through the authoring of additional CABG libraries to implement domain specific actions. For our language, we will be implementing a String library that handles arrays of characters, along with typical string operations like push, pop, length and more.

Syntax:
```
function farg1 farg2 ..

[fvar1 = x
...]

Start sarg1 sarg2 ...
    ? condition : action -> next_state arg1 arg2
    ...

StateName sarg1 sarg2 ...
    ? condition : action -> next_state arg1 arg2
    ...

...

end
```

Ideas:

Execution:

- On entering a function, the first state entered is Start. Therefore Start cannot have any arguments associated with it.

Keywords:

- # single-line comment
- **end** concludes the previously started state
- **?** starts the definition of a condition
    - If a condition fails execution of a line is halted
    - If a condition passes, execution of what follows **:** is performed
- **:** starts the definition of statements to execute through until the goto statement
- **->** is the result of a condition being satisfied
- **library** imports a file into this context

States:

- States start with a capital letter
- Parameters for a state are put after the state label and are separated from the label by a space, and from one another with commas
- Conditions are separated by commas
- Within states, statements are indented

Side-effects:

- Anything not an assignment or condition is printed
- Any transition to a variable results in that variable being printed

Scope:

- Parameters can see global scope
- Within states scope is local

Exceptions:

- Unconditional exceptions can omit '? ->'
- Conditional exceptions: ? x == a -> Error

Variables:

- Variables can be characters, integers, or character classes for matching.
- Names of character classes begin with an escape character and classes go in square brackets. E.g. \d = [0123456789].

Sample Program:

# This function takes a string and looks for function calls without arguments. Specifically,

```
# it matches against function()

library string

findFunctionCalls code

c = pop(code)                  # get the first character of the string
\d = [0123456789]              # define character classes for matching: digits, whitespace, letters
\w = [ \t\r\n]
\l = [_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]

Start
    ? c == null : -> false                          # end of string, return false
    ? c != \l : c = pop(code) -> Start              # advance until a letter or underscore
    ? c == \l : c = pop(code) -> FindParens

FindParens
    ? c == \l || c == \d : c = pop(code) -> FindParens          # advance until a paren
    ? c == '(' : c = pop(code) -> CloseParens
    ?  : c = pop(code) -> Start                     # no paren, start over looking

CloseParens
    ? c == ')' : -> true                            # string matches, return true
    ?  : c = pop(code) -> Start                     # no paren, start over looking

end
```