CMPT 360: Lab Assignment #3
Stack & RPM Calculator

Brady Coles

last updated: January 3, 2018

Contents

1 Course Goals

This assignment fulfills the following goals:

- a group III language (Ruby)
- implemented on the Linux platform

2 Problem Description

The given problem was to create an ADT (advanced data type) and use it in an application.

In accomplishing that goal, a stack was implemented in Ruby and used to create a Reverse Polish Notation (RPN) calculator.

3 Sample I/O

RPN is a postfix notation for describing mathematical expressions. Unlike the usual infix notation, where the operators are between the operands, RPN has the operators after the operands. Examples follow:

```
5 1 +
= 6

1 4 /
= 0.25

-5 3 *
= -15

5 3 + 8 *
= 8 8 *
= 64
```

For a more complete description of RPN, see www.calculator.org/rpn.aspx

4 Language Comparison

The program here was implemented using Ruby. I will compare the implementation of the stack to how it would be implemented in a language such as C#.

In either language I would likely implement a stack using an array. The arrays in Ruby are quite different from those in C#, where arrays are fixed size and can hold only one type of value. In C#, the size of the array would have to be manually adjusted to account for the stack growing or shrinking. A C# stack would be implemented as a generic so that the type of element the array could hold could be anything, but it would still be limited to a single type per stack.

In Ruby, the arrays are typeless and are allocated dynamically, meaning the size of the array is adjusted as elements are added. Using a built in function, elements can be deleted as well. This removes the need to keep track of the size of the array and the number of items in the stack separately, as well as the need to manually adjust the size of the array. Ruby arrays also allow for a more flexible stack, where items of any type can be added to a stack.

5 Program Documentation

5.1 Errors and Messages

The program checks the input RPN statement for validity, and will display a message if the input is invalid. There are a couple messages like that:

`Invalid numeric literal: <numeric literal>`
The number is invalid, likely due to more than one decimal place.

`Unkown symbol: <symbol>`
The symbol in the input is not a valid character for the calculator.

`Error in input`
Something is wrong with the input. Likely the sequence is not a valid expression.

5.1.1 Errors

There are a couple of possible error messages which should never be displayed. If you encounter one of these errors, then there is a mistake in program code that needs to be resolved by a programmer. They are included for thoroughness sake.

`ERROR - Unkown binary operator in stack`
A valid binary operator was parsed, but the calculator didn't recognize it.

`ERROR - bin_statement called without a binary operator in stack`
A function (bin_statement) was called without proper pre-conditions being satisfied.

5.2 Problem Solution

The stack implementation requires five method definitions.

length Returns the number of items in the stack

isEmpty Returns true if the stack has no elements, false otherwise.

push Puts an item onto the top of the stack.

peek Returns the top item on the stack without removing it.

pop Returns the top item on the stack and removes it from the stack.

5.3 Pseudocode

This is pseudocode implementation of the stack.

```
class stack
    array = []
    function length
         return array.length
     function isEmpty
         return (array.length == 0)
     function push(item)
         array[last + 1] = item
     function peek
         return array[last]
     function pop
         i = array[last]
         remove array[last]
         return i
```

5.4 Documentation

The program contains a module and a class which can be imported into other programs.

5.4.1 Stack Class

This class provides a general purpose stack for any type of items.

`class Stack`		
Constructors		
`new`		Creates an empty stack.
Public Methods		
Name	Arguments	Description
`length`		Returns the number of items in the stack.
`is_empty?`		Returns true if the stack is empty, false otherwise.
`push`	`item` - an item to add to the stack	Adds `item` to the top of the stack.
`peek`		Returns the top item on the stack without removing the item from the stack.
`pop`		Returns the top item on the stack and removes the item from the stack.

5.4.2 RPNCalc Module

This module provides the means to parse strings into RPN expressions, and to calculate the result of those expressions. Allows the creation of RPN calculators.

`module RPNCalc`		
Public Functions		
Name	Arguments	Description
`parse`	`str` - A string of valid RPN	Creates a stack of lexemes describing the RPN in the input string, with the top of the stack being the rightmost lexemes of the RPN expression.
`calculate`	`st` - A stack of valid RPN lexemes	Calculates the result of an already parsed RPN expression. The stack should have the rightmost lexemes on top.

5.4.3 Program Operation

The program uses the RPNCalc module to create a command line calculator. The calculator takes user inputted expressions in RPN and outputs the result, allowing the user to enter a new expression. At any time the user can enter 'exit' instead of an expression to close the program. A description of valid RPN format is given in section 3.

Possible lexemes include:

Operators	
`*`	Multiplication operator.
\	Division operator.
`+`	Addition operator.
`-`	Subtraction operator.
Numeric Literals	
`125`	Integers, any length
`1.63`	Decimals, any length
`.012`	Decimals, leading zero not required
`-15.2`	Negative numbers (sign must be touching)
`+90`	Positive numbers (sign never required)

The parser is very loose with the syntax of RPN expressions. White space is only required where ambiguity or alternate meaning would arise from it being missing. This means only between numbers and after plus/minus operators if a number is next (to avoid confusion with sign). That makes the following expressions equivelant.

```
5 4 * -3.2 / 1 +
5 4*-3.2/1+
```

6 Program Listing

```
#!/usr/bin/ruby -w

# Author: Brady Coles
# Lab Assignment # 3
# ADT: Reverse Polish Notation Calculator implemented with a Stack

# Generic stack. Utilizes associative array to allow any type elements in stack
class Stack

    # Initialize array to hold elements in stack
    def initialize
        @stack = Array.new
    end

    # Return number of items in stack
    def length
        return @stack.length
    end

    # True if stack has no elements, false otherwise.
    def is_empty?
        return @stack.length == 0
    end

    # Push an item onto the stack.
    def push(item)
        @stack[@stack.length]=item
```

```ruby
    end

    # Get top item on stack without removing it.
    def peek
        if @stack.length <= 0
            #ERROR
        else
            return @stack[@stack.length - 1]
        end
    end

    # Remove top item on stack, returns top item.
    def pop
        if @stack.length <= 0
            #ERROR
        else
            return @stack.delete_at(@stack.length - 1)
        end
    end
end

# Module for parsing strings in RPN and for calculating value of RPN statements
module RPNCalc

    # CONSTANTS
    ADD_OP = "+"
    MUL_OP = "*"
    SUB_OP = "-"
    DIV_OP = "/"
    BIN_OPS = [ADD_OP, MUL_OP, SUB_OP, DIV_OP]
    SEP = " "
    DEC_SEP = "."
    DIGITS = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    NUMERIC = DIGITS + [DEC_SEP]

    # Parses a string in RPN into lexemes, puts lexemes into a stack, which is returned
    # Parser is not strict on whitespace. If two characters can never be together
    # in a lexeme, then it assumes they are in separate lexemes.
    # eg. '3-5/' is the same as '3 -5 /', or in standard notation '3 / (-5)'
    # Interpets decimal points as numeric, but checks that no more than one decimal point
    # is in each numeric lexeme.
    # Converts numeric lexemes into floating point type, regardless of whether the lexeme
    # had a decimal point or not.
    def parse(str)
        lexeme = ""
        st = Stack.new
        isnumeric = false
        # Iterate over each character from the left
        str.each_char do |symbol|
            # If the current lexeme is not empty and the current symbol is not numeric
            # then the current lexeme must be complete.
            if !(NUMERIC.include? symbol) && lexeme != ""
                # If the current lexeme is numeric, convert it to a float
                if isnumeric
```

```
                    # Ensure lexeme has no more than one decimal point, else stop parsing
                    # and return nil
                    if lexeme.count(DEC_SEP) == 1 || !lexeme.include?(DEC_SEP)
                        lexeme = lexeme.to_f
                    else
                        puts "Invalid numeric literal: " + lexeme
                        return
                    end
                end
                # Push lexeme onto stack and reset for next lexeme
                st.push(lexeme)
                isnumeric = false
                lexeme = ""
            end
            # Check what next symbol is
            case symbol
                # Separators are used in the above selection, so are skipped
                when SEP
                    next
                # Plus and minus symbols can be operators or sign a numeric, so add to lexeme
                when ADD_OP, SUB_OP
                    lexeme = symbol
                # Mul and div ops are always there own lexeme, push to stack.
                when MUL_OP, DIV_OP
                    st.push(symbol)
                # Digits and decimal points get added to current lexeme and set flag
                # isnumeric for the above selection
                when *NUMERIC
                    isnumeric = true
                    lexeme += symbol
                # If symbol is not recognized, stop parsing and return nil
                else
                    puts "Unknown symbol: " + symbol
                    return
            end
        end
        # Push last lexeme. Must be an operator unless only a numeric was passed.
        if lexeme != ""
            if isnumeric
                # Ensure lexeme has no more than one decimal point, else stop parsing
                # and return nil
                if lexeme.count(DEC_SEP) == 1 || !lexeme.include?(DEC_SEP)
                    lexeme = lexeme.to_f
                else
                    puts "Invalid numeric literal: " + lexeme
                    return
                end
            end
            st.push(lexeme)
        end

        # Return the stack
        return st
    end
```

```
136
137     # Calculate an RPN expression from a stack
138     def calculate(st)
139         # if the stack is nil, do nothing
140         if st != nil
141             val = statement(st)
142             # If the stack is not empty after running statement, then the stack
143             # is not valid a valid RPN statement.
144             return val if st.is_empty?
145             puts "Error in input"
146             return
147         end
148     end
149
150     #Remaining functions are only accessible indirectly from calculate
151     private
152     # Return value of an RPN statement from a stack
153     def statement(st)
154         # If the stack is empty, then the stack is invalid, even if this occurs
155         # in a recursive call.
156         if st.respond_to?(:is_empty?) && st.is_empty?
157             puts "Error in input"
158             return
159         end
160
161         # If the next lexeme is an operator, decode as a binary operator statement
162         if BIN_OPS.include? st.peek
163             val =  bin_statement(st)
164         # If the next lexeme is not an operator, it is a numeric, so return it as is.
165         else
166             return st.pop
167         end
168     end
169
170     # Return value of a binary operator RPN statement, where there are two statements and
171     # a binary operator.
172     def bin_statement(st)
173         # If the stack is empty, then the stack is invalid, even if this occurs
174         # in a recursive call.
175         if st.respond_to?(:is_empty?) && st.is_empty?
176             puts "Error in input"
177             return
178         end
179
180         # Since bin_statement was called, the next lexeme should be an operator
181         if BIN_OPS.include? st.peek
182             case st.pop
183                 # Multiplication
184                 when MUL_OP
185                     val2 = statement(st)
186                     val1 = statement(st)
187                     return val1 * val2 if !([val1, val2].include? nil)
188                     return
189                 # Division. Is not commutative, so values ordered properly.
```

```
                when DIV_OP
                    val2 = statement(st)
                    val1 = statement(st)
                    return val1 / val2 if !([val1, val2].include? nil)
                    return
                # Addition
                when ADD_OP
                    val2 = statement(st)
                    val1 = statement(st)
                    return val1 + val2 if !([val1, val2].include? nil)
                    return
                # Subtraction. Is not commutative, so values ordered properly.
                when SUB_OP
                    val2 = statement(st)
                    val1= statement(st)
                    return val1 - val2 if !([val1, val2].include? nil)
                    return
                # If definition of BIN_OPS is changed, this error may occur
                else
                    puts "ERROR - Unkown binary operator in stack"
                    return
            end
        else
            # If function called when it shouldn't have been, this error may occur
            puts "ERROR - bin_statement called without a binary operator in stack"
            return
        end
    end
end

# Program itself. Uses RPNCalc to return answers.
if __FILE__ == $0
    include RPNCalc
    puts "Enter calculations in Reverse Polish Notation"
    puts "Valid operators: + - / *"
    puts "Enter 'exit' to end program"

    while true do
        print ">> "
        input = gets.chomp
        if input == "exit"
            puts "Goodbye"
            break
        end
        val = calculate(parse(input))
        puts val if val != nil
    end
end
```

7 I/O Listing

The following is a printout of an actual run of the program. User input follows >>.

```
Enter calculations in Reverse Polish Notation
Valid operators: + - / *
Enter 'exit' to end program
>> 5 5 +
10.0
>> 1 2 - 4 *
-4.0
>> 5 -2 + 3 - 6 /
0.0
>> 0 0 /
NaN
>> 1.3 .5 +
1.8
>> 5 5 / 5 *
5.0
>> hello
Unknown symbol: h
>> 5 +
Error in input
>> 3.14
3.14
>> .5. 7 +
Invalid numeric literal: .5.
>> 800 0.00125 * 1 /
1.0
>> exit
Goodbye
```