Chase Brown

CSCI 3415-001

Program 4 GO

The purpose of this assignment is to write a basic calculator that is capable of simple operations of addition, subtraction, multiplication, division, and be able to handle operations with parentheses. The program is to use stacks to push and pop operands and operators from their respective stacks to perform the correct operations based on precedence. Multiplication and division have higher precedence then addition and subtraction and parentheses override precedence. Reflection is used to perform operations on float and integer values in the final product.

I ended up taking the same approach as I did with my Ada program to complete this program. The layout of the code is identical for this program with the syntax of the language being the significant difference.

Pseudo~

scan for input string from user

call evaluate

    create operand stack

    create operator stack

func precedence(op)

    evaluate the operator and assign it a value 0-2

func apply()

    pop off two operand and a operator to calculate

    use Reflection to check if the operands are of type int

    if they are then convert them to float

    Switch statement ->perform operation

    push result back into operand stack

loop the buffer to get operands and operators

    if ‘0’..’9’ | ‘.’ Then get number and push into the operand stack index++

    if ‘+’, ‘-‘, ‘*’, ‘/’ check the top of the operator stack for precedence index++

        if precedence on stack then call apply

        push operator into the operator stack

    if ‘(‘ push onto the operator stack index++

if ‘)’ loop back doing apply until reaching a ‘(‘ and pop the ‘(‘ off the operator stack indx++

if ‘ ‘ if a space ignore it index++

end loop

end of evaluation call apply() for any operators left on the operator stack.

Pop the final number into result

Return the result to main

Print result.

By approaching this program with the same format as my Ada program I was able to get a simple calculator with integers working along with the precedence of operators and parentheses. From there I worked on getting the program to work with floats, but this causes the problem of not being able to do integer and float calculations. Reflection was a little tricky to figure out to get the program to work with floats and integers. I spent some time reading up on the rules of reflection to accomplish integer and float calculations and wrote a small program so that I could play with the operations of reflection to understand how to use different operations that reflect possesses. To get the float and integer calculations to work I checked to see what the TypeOf was for a value that is popped off the operand stack. Checking with the reflect.kind() I can see if that value is an integer. If it is then I get the ValueOf the value and then create a new interface with the TypeOf float64 with the value. This now allows me to do my operations all as float numbers. Once I had the float and integer calculations working I then applied more error checking to the program for my push and pop operations and if there is an expression error. I used GO err, panic formatting to achieve my error checking.

```
GOROOT=C:\Go
GOPATH=C:/Users/kogai/IdeaProjects/PopGo;C:/Go
C:\Go\bin\go.exe build -i -o C:\Users\kogai\AppData\Local\Temp\___build.exe main
"C:\Program Files\JetBrains\IntelliJ IDEA 2017.2.5\bin\runnerw.exe" C:\Users\kogai\AppData\Local\Temp\___build.exe
Please input an operation expression without any space:
23*(3+2*(6.9/3-2)+3*(2*(6*1.2-2)+3))+3*(3-1)
23*(3+2*(6.9/3-2)+3*(2*(6*1.2-2)+3))+3*(3-1)
Result: 1013.4
(9*4.5)/2+67
(9*4.5)/2+67
Result: 87.25
56.8-4/
56.8-4/
Pop failed operand error
Result: 0
99+100*(4+6)
99+100*(4+6)
Result: 1099
8.99945*2
8.99945*2
Result: 17.9989
4+90/3(4+
4+90/3(4+
Illegal Operator
Result: 0
4+90/3(4+9)
4+90/3(4+9)
Result: 90.23076923076923
```

I was very surprised at how much GO's syntax is like java and a sort of mix of python. The syntax of GO didn't take long to get use to and seems to be a very clean language with high readability. I found the use of 'panic' with errors comical because most programs it's just tray catch throw but for some reason 'panic' just fits. The use of reflection creates the illusion that the language is dynamically typed when it is statically typed which is a nice ability to have in a language. This gives the programmer a lot more flexibility when dealing with values that could be difficult to achieve in other languages. It is obvious that this language is a new language with its features, and was developed with the idea of mass usage given its ease to learn and readability. I liked programming in GO and from just a bit of coding can see the usefulness of the language.

References

The Go Programming Language. (n.d.). Retrieved November 19, 2017, from https://golang.org/

The Go Programming Language Specification. (n.d.). Retrieved November 19, 2017, from https://golang.org/ref/spec

The Go Blog. (n.d.). Retrieved November 19, 2017, from https://blog.golang.org/laws-of-reflection

```go
package main

import (
   "bufio"
   "errors"
   "fmt"
   "os"
   "reflect"
   "stack"
)

var expression_error error = errors.New("expression error")
var buffer string
var index int = 0
var expect_operand bool

//func evaluate() (int, error)
//evaluates the integer expression in Buffer and returns the result
//an Expression error is raised if the expression has an error
func evaluate() (float64, error) {
   var operand_stack = stack.New()
   var operator_stack = stack.New()
   var err error
   //precedence := func (op byte) int{...}
   //Returns the precedence of Operator. Raises Exception_error if
   //Operator is not a known operator.
   // '+' | '-' => 1
   // '*' | '-' => 2
   // '#' | '(' => 0
   //others raise exception
   precedence := func(op byte) int {
      //todo case statements
      switch op {
      case '+':
         return 1
      case '-':
         return 1
      case '*':
         return 2
      case '/':
         return 2
      case '#':
         return 0
      case '(':
         return 0
      default:
         panic("Illegal operator")
      }
   }

   //apply := func() error {..}
   //Applies the top operator on the Operator_Stack to its right and left
   //Operands on the operand stack
   apply := func() error {
      var op interface{}
      var left, right interface{}
      var err error
      //pop the operator off into op
      if err = operator_stack.Pop(&op); err != nil {
         panic("Pop failed operator error")
      }
      //pop the operand off into the right
      if err = operand_stack.Pop(&right); err != nil {
```

```
            panic("Pop failed operand error")
         }
         //pop the operand off into the left
         if err = operand_stack.Pop(&left); err != nil {
            panic("Pop failed operand error")
         }
         var R float64
         var L float64
         //fmt.Println(reflect.TypeOf(right))
         //tests the reflect of the interface to see if the value is a int or float
         //if it is a int then the reflect value is stored then that value is stored as a
float
         v := reflect.TypeOf(right)
         if v.Kind() == reflect.Int {
            u := reflect.ValueOf(right)
            k := u.Interface()
            R = float64(k.(int))
            //fmt.Println(reflect.TypeOf(R))
         } else {
            R = right.(float64)
         }
         w := reflect.TypeOf(left)
         if w.Kind() == reflect.Int {
            e := reflect.ValueOf(left)
            q := e.Interface()
            L = float64(q.(int))
         } else {
            L = left.(float64)
         }

         //t := L+R
         //fmt.Println(t)

         switch op.(byte) {
         case '+':
            if err = operand_stack.Push(L + R); err != nil {
               panic("Error pushing value from addition")
            }
         case '-':
            if err = operand_stack.Push(L - R); err != nil {
               panic("Error pushing value from subtraction")
            }
         case '*':
            if err = operand_stack.Push(L * R); err != nil {
               panic("Error pushing value from Multiplication")
            }
         case '/':
            if err = operand_stack.Push(L / R); err != nil {
               panic("Error pushing value from Division")
            }
         default:
            panic("Illegal Operator")
         }
         return nil
      }
      //func() {...}()
      //Recover from a panic
      defer func() {
         if r := recover(); r != nil {
            fmt.Printf("%v\n", r)
         }
      }()
      var Dec bool = false
```

```
   //process the expression left to right on character at a time.
   for index < len(buffer) {
      switch buffer[index] {
      case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.':
         {
            //The character starts an operand. Extract it and push it on the operand
stack
            value := 0
            var Pos int = 0
            var Count int = 0
            var Devi float64 = 1
            //loop repeats getting the values of the numbers in the string.
            //if the number has a decimal "float" then track where that decimal is and
then convert the integer number
            //that is created into a float ex: "1.23 gets read into value 123 and then
devided by 100 to give 1.23"
            for index < len(buffer) && buffer[index] >= '0' && buffer[index] <= '9' ||
index < len(buffer) && buffer[index] == '.' {
               if buffer[index] == '.' {
                  Pos = Count
                  Dec = true //decimal point was detected track its position in the
"float"
                  index += 1
               }
               value = value*10 + int(buffer[index]-'0')
               Count += 1
               index += 1
            }
            Pos = Count - Pos
            if Dec {
               for Pos != 0 {
                  Devi = Devi * 10
                  Pos -= 1
               }
               pvalue := float64(value) / Devi
               //fmt.Println(pvalue)
               if err = operand_stack.Push(pvalue); err != nil {
                  panic("Failed to push float value")
               }
            } else {
               pvalue := value
               //fmt.Println(pvalue)
               if err = operand_stack.Push(pvalue); err != nil {
                  panic("Failed to push int value")
               }
            }
            //fmt.Println("break")
            //operand_stack.Push(value)
            //expect an operator after an operand set operand expected to false and
reset the Dec detection to false
            expect_operand = false
            Dec = false
         }
      case '+', '-', '*', '/':
         {
            //The character is an operator.  Apply any pending operators
            //on the operator_stack whos precedence is greater than or equal
            //to this operator.  Then, push the operator on the operator stack

            //if the top of the stack does not == nil then check the precedence of the
operators
            //if the operator has a higher or equal precedence then do work else move
on.
```

```
            //fmt.Println("operator")

            if operator_stack.Top() != nil {
               if precedence(buffer[index]) <= precedence(operator_stack.Top().(byte))
{
                  apply()
               }
            }

            //push operator onto operator stack
            if err = operator_stack.Push(buffer[index]); err != nil {
               panic("Failed to push Operator to stack")
            }

            if expect_operand {
               return 0, expression_error
            }
            //expect an operand after operator.  set to true
            expect_operand = true
            //inc to next value in buffer
            index += 1
         }
      case ')':
         {
            //if the character is a right paren then loop back through the stack to
            //till a left paren doing work
            var ex interface{}

            ex = uint8(40)
            for operator_stack.Top() != ex {
               if precedence(operator_stack.Top().(byte)) > precedence(ex.(byte)) {
                  apply()
               }
            }
            operator_stack.PopOff()
            index += 1
         }
      case '(':
         {
            //fmt.Println("left parens")
            if err = operator_stack.Push(buffer[index]); err != nil {
               panic("Failed to push LParens to stack")
            }
            index += 1
         }
      case ' ':
         {
            //The character is a space. Ignore it
            index += 1
         }
      //the character is something unexpected.  Return an expression error
      default:
         return 0, expression_error
      }
   }
   //we are at the end of the expression.  Apply all of the pending operators.  The
operand stack must have exactly
   //one value, which is returned
   for !operator_stack.IsEmpty() {
      apply()
   }

   var value interface{}
```

```
   if err = operand_stack.Pop(&value); err != nil {
      panic("Pop Failed at result")
   }
   return value.(float64), nil
}

func main() {
   var result float64
   fmt.Println("Please input an operation expression without any space:")
   scanner := bufio.NewScanner(os.Stdin)
   //Process all of the expressions in standard input
   for scanner.Scan() {
      //read the next expression, evaluate it, and print the result.
      buffer = scanner.Text()
      fmt.Println(buffer)
      index = 0
      expect_operand = true
      var err error
      if result, err = evaluate(); err != nil {
         fmt.Printf("Error: %s\n", err.Error())
      }
      fmt.Printf("Result: %v\n", result)
   }
}
```

```
// Provides stack operations for any type. A stack is a first-in-first-out
// (LIFO) data store.
//
// This implementation provides an unbounded stack inplemented as a linked list
// of cells.
//
package stack

import "errors"

// Underflow is a error that occurs when you attempt to access as empty stack.
//
var Underflow = errors.New("stack underflow");

// A cell stores a single value of any type and a pointer to the next cell.
//
type cell struct {
    next *cell
   value interface{}
}

// A stack contains a pointer to the first cell.
//
type Stack struct {
    top *cell
}

func New() Stack {
    return Stack{nil}
}

// Pushes a value onto the stack.
//
func (s *Stack) Push(v interface{}) error {
    s.top = &cell{s.top, v}
   return nil
}
```

```
// Pops a value from the stack. An underflow error is returned if the stack is
// empty.
//
func (s *Stack) Pop(v *interface{}) error {
    if s.top == nil {
        return Underflow
    }
    *v = s.top.value
    s.top = s.top.next
    return nil
}
func (s * Stack) PopOff() (interface{}, error){
    if s.top == nil{
        return nil, Underflow
    }
    v := s.top.value
    s.top = s.top.next
    return v, nil
}
// Returns the top value on the stack. An underflow error is returned if the
// stack is empty.
//
func (s Stack) Top() interface{} {
    if s.top == nil {
        return nil
    }
    return s.top.value
}

// Returns true is the stack is empty.
//
func (s Stack) IsEmpty() bool {
    return s.top == nil
}
```

```
package stack

import "testing"

func TestStack(t *testing.T) {

    s := New();

    // Test 1
    if !s.IsEmpty() {
        t.Errorf("Stack is not empty.")
    }

    // Test 2
    t2_v := 1
    if err := s.Push(t2_v); err != nil {
        t.Errorf("Push error: %v", err.Error())
    }
    if s.Top() != t2_v {
        t.Errorf("Top value = %v, want %v", s.Top, t2_v)
    }

    // Test 3
    var t3_v interface{}
    if err := s.Pop(&t3_v); err != nil {
        t.Errorf("Pop error: %v", err.Error())
```

```
    } else if t3_v != t2_v {
        t.Errorf("Pop value = %v, want %v", t3_v, t2_v)
    }

    // Test 4
    if !s.IsEmpty() {
        t.Errorf("Stack is not empty.")
    }

    // Test 5
    var t5_v interface{}
    if err := s.Pop(&t5_v); err != Underflow {
        t.Errorf("Pop did not return underflow")
    }
}
```