

Brian Sumner

Dr. Doug Williams

UCDenver CSCI 3415-001

Fall 2016

Programming Assignment #3 – Go Language (file: sumnerbr-calculatorer.go)

Contents:

I. Program Objective (Excerpted from assignment)	p.02
II. Program Design	p.04
III. Implementation Details	p.06
IV. Source Code:	
b. Source Code (file: sumnerbr-calculatorer.go)	p.09
c. Source Code (file: stack.go)	p.022
V. Sample Runs:	
a. Simple Example Expressions	p.023
b. Error Checking (panicker: ERR_01)	p.024
c. Error Checking (panicker: ERR_11)	p.025
d. Error Checking (panicker: ERR_21)	p.026
e. Error Checking (panicker: ERR_31)	p.027
f. Error Checking (panicker: ERR_32)	p.028
g. Error Checking (panicker: ERR_51)	p.029
h. Error Checking (panicker: ERR_52)	p.030
i. Error Checking (panicker: ERR_61)	p.031
j. Error Checking (panicker: ERR_71)	p.032
k. Error Checking (panicker: ERR_81)	p.033
l. Error Checking (panicker: ERR_91)	p.034
m. Type Reflection Behaviors (Addition)	p.035
n. Type Reflection Behaviors (Subtraction)	p.036
o. Type Reflection Behaviors (Multiplication)	p.037
p. Type Reflection Behaviors (Division)	p.038
q. Parenthetical / Formatting Behaviors	p.040
VI. Conclusions	p.041

I. Program Objective (Excerpted from assignment):

“Part II – Calculator”

“For this program you will be implementing a simple calculator. By simple I mean it only need to support the binary operators +, -, *, and / with their standard definitions (i.e., addition, subtraction, multiplication, and division) and parenthesis. Multiplication and division have higher precedence than addition and subtraction, all of them have left to right associativity, and parentheses can be used to override the precedence.

The program will accept a string containing the expression to be evaluated and print the result of evaluating the expression. Some simple example are:

```
>2*3
6
>2*3.5-2
5.0
>2*(3.5-2)
3.0
```

You should evaluate the expression in a single left to right scan of the input string, using an operator and an operand stack to hold intermediate results during the evaluation. You may handle parenthesis either by a recursive call or by using the stack.

You can decide how to handle whitespace in an expression. For example, you could completely ignore them, ignore them in limited cases (like around operators and operands), or always return an error. [My preference would be to allow spaces around operators and operands, but not in an operand. So, 20 + 30 is valid, but 2 0 + 3 0 is not.] Specify how you process them.

Consider any characters other than space, 0-9, ., +, -, *, and / as an error.

Numbers can be integers (e.g., 2, which is 2.0) or reals (e.g., 2.0). You don't have to worry about scientific notation for inputs. You don't have to worry about negative numbers for inputs – they are most easily handled using unary -, which we don't have.”

“You might consider developing your code in steps – like we did in class.”

- Start with just integers and +, -, *, and / - that is, no parentheses, just integer parsing, and limited error checking. This will get the basic parsing and evaluation algorithm in place.
- Add parenthesis handling. This can either use recursion to evaluate a parenthesized subexpression, or push the open parenthesis on the operator stack and handle it in the code.
- Add error checking.
- [This is required for a grade of C.]
- Add floats. Initially, treat everything as floating point – that is, no mixed arithmetic.
- [This is required for a grade of B.]
- Add reflection to support mixed integer and floating point calculations.
- [This is required for a grade of A.]”

“Part III – Reflection”

“Package [reflect](#) implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type interface{ } and extract its dynamic type information by calling `TypeOf`, which returns a `Type`.

See "The Laws of Reflection" for an introduction to reflection in Go: <https://blog.golang.org/laws-ofreflection>.”

“Part IV – Specifics”

“Each student will develop a Go program to implement a simple mixed integer / float calculator as described above. Each program submission will include a report describing the problem, your approach to solving it (pseudo code, etc), the source code, and sample outputs. Your report should cite any sources of materials you used in solving the problem.”

II. Program Design:

II. a. General:

Sumnerbr-calculatorer was modified from Dr. Williams' example code and features an operator stack and an operand stack implemented from the stack package he provided. Note: A calculatorer is a thing that does the calculatoring. The source code for *sumnerbr-calculatorer.go* is available from its repository at: <https://github.com/C5T2B7U/ucdenver-csci3415-pa03> because of the sheer difficulty of copying the code from this document and then getting the resulting .go file to compile (due to formatting attributes incident to this document).

II. b. Error Checking:

Dr. Williams' example code featured the use of the `panic()` function in response to basic expression syntax errors. Sumnerbr-calculatorer uses a dedicated `panicker()` function to do all the panicking: `panicker()` scans the input expression for a variety of expression errors and panics if any are encountered. The stacks are then populated and the expression is evaluated if and only if `panicker` does not panic.

II. c. Parenthetic Subexpression Handling:

Dr. Williams specified that the two most common ways to implement parenthetic subexpression handling were through recursion or by directly utilizing the stacks. Sumnerbr-calculatorer uses the stack for parenthetic subexpression handling primarily because this required less modification to the original program than through a recursive solution. Implementing recursion would have involved moving all the expression evaluation code into a nonexistent function, followed by modification of the `calculatorer()` function (formerly known as the `apply()` function) to pass stack objects as parameters. Instead, by pushing parenthetic operators to the operator stack, a fundamental rewrite of that function was avoided. Sumnerbr-calculatorer offers seamless integration of parenthetic subexpression handling into the original code.

II. d. Floating-point Arithmetic and Type Reflection:

Implementing floating-point arithmetic was a trivial addition to the original code that involved checking for a decimal point when building each operand, and if found converting the operand to a floating-point variable before being pushed onto the stack. Reflection was then added to perform integer arithmetic if and only if the reflect types of both operands match the reflect types of variables known to be integers. Reflection is also used in this way to specify the type of the final result in its display output.

III. Implementation Details:

III. a. Panicker:

As shown below in the Sample Runs section, the panicker() function scans the expression for the following syntax errors:

- (panicker: ERR_01) Panicker panics on encountering an empty expression.
- (panicker: ERR_11) Panicker panics on encountering division by zero.
- (panicker: ERR_21) Panicker panics on encountering a missing operator (or a non-contiguous operand).
- (panicker: ERR_31) Panicker panics on encountering a decimal point instead of an operator.
- (panicker: ERR_32) Panicker panics on encountering more than one decimal point in an operand.
- (panicker: ERR_51) Panicker panics on encountering a unary negative operator for an operand.
- (panicker: ERR_52) Panicker panics on encountering another operator when expecting an operand.
- (panicker: ERR_61) Panicker panics on encountering a beginparens when expecting an operator.
- (panicker: ERR_71) Panicker panics on encountering more endparens than beginparens.
- (panicker: ERR_81) Panicker panics on encountering invalid characters.
- (panicker: ERR_91) Panicker panics on encountering an expression that ends with an operator instead of an operand.

If panicker does not panic, then calculatorer() can be assured that nothing will panic and is then permitted to calculator the expression.

III. b. Parenthesizer:

Parenthesizer parenthesizes. Specifically, `parenthesizer()` completes incomplete parenthesization. If the number of `beginparens` is less than the number of `endparens` then `panicker()` will panic. If `panicker` panics, `parenthesizer` will not parenthesize. Furthermore, it's not `panicker`'s job to parenthesize expressions (that's what `parenthesizer` is for). `Parenthesizer` ensures all expressions begin with a `beginparens`.

III. c. Calculatorer:

`Calculatorer` calculates. The `calculatorer()` function does all the calculating. `Calculatorer` also uses `reflect` to determine whether the type of each operand at the top of the stack matches the `reflect` type of a known integer or a known `float64` before popping the operands off the stack using the appropriate types. `Calculatorer` then performs the appropriate arithmetic operation. If arithmetic is performed on two integers, the result will be pushed back onto the stack as an integer, otherwise `calculatorer` pushes a `float64` result back onto the operand stack. `Calculatorer` also calls `panicker` to panic if division by zero is encountered. `Calculatorer` discards any `beginparens` operator before returning.

III. d. Main:

The `main()` function prompts the user for an expression, sends a copy to `panicker` for panicking, calls `parenthesizer` for parenthesization, then begins a characterwise single-scan-evaluation loop. If the loop encounters an operand, it will scan it in and push it to the operand stack as either an integer or a `float64`. If an operator is found, first it calculates all operations for the current parenthetic subexpression with higher precedence than the current operation that are pending on the stacks, before pushing the found operator to the stack. If a space is found, it is disregarded since `panicker` would have already panicked if the space had been placed in error. If a `beginparens` is found, it is pushed to the operator stack to indicate the start of a parenthetic subexpression. If an `endparens` is found, then `calculatorer` is called until all pending operations within the current parenthetic subexpression have been calculatored, after which point `calculatorer` is called again to calculator the result of the current subexpression with the last operand of the previous subexpression. After the characterwise loop finishes scanning the expression, `calculatorer` is called until the operator stack is empty and only the final result remains on the operand stack. At this point, the reflect type of the result is checked against the reflect types of known integer and `float64` variables to determine the correct output.

IV. Source Code: a. Source Code (file: sumnerbr-calculatorer.go)

```
// BRIAN SUMNER
// UC DENVER CSCI3415-001
// FALL 2016
// PA03 - SUMNERBR-CALCULATORER

// NOTE: A CALCULATORER DOES THE CALCULATORING.

// (MODIFIED FROM INSTRUCTOR EXAMPLE USING INSTRUCTOR STACK PACKAGE)
// REFERENCE USED: https://golang.org/doc/effective\_go.html
// REFERENCE USED: https://blog.golang.org/laws-of-reflection

package main

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

var operatorStack = stack.NewStack()
var operandStack = stack.NewStack()

// PANICKER PANICS.
// PANICKER DOES ALL THE PANICKING.
// DO NOT PANIC UNLESS PANICKER PANICS.
// PANICKER IS THE ONLY DRAMA QUEEN HERE.
// NOTE: IT IS NOT PANICKER'S JOB TO FIX EXPRESSIONS.
func panicker(inputString string) {

    // CAUTION: PANICKER MAY BECOME VERY UPSET.
    PANICMSG_AAAAAGGHH := " YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!\n\n"

    // CHECK NOT EMPTY STRING
    if inputString == "" {
        panic("\n\nERR_01:  OH NO!  I'VE DIED OF BOREDOM!!!\n\n")
    }

    // CHECK IF SPECIAL CASE
    if inputString == "/0" {
        panic("\n\nERR_11:  I DON'T KNOW HOW TO DIVIDE BY ZERO, ARE YOU CRAZY?!\n\n          I CAN'T GO ON LIVING LIKE THIS!!!\n\n")
    }
}
```

```

} else if inputString == "???" {
    panic("\n\nERR_12:  I DON'T UNDERSTAND WHAT YOU'RE TRYING TO DO TO ME!  LEAVE ME ALONE ALREADY!!!\n\n")
}

// CHECK ALL CHARACTERS ARE VALID
// CHECK NUMBER OF OPEN PARENTHESES NOT NEGATIVE
// CHECK NUMBER OF DECIMAL POINTS PER OPERAND
// CHECK ARITHMETIC SYNTAX IS CORRECT
openParentheses := int(0)
activeDecimalPoints := int(0)
isIncompleteOperand := false
isNegativeOperand := false
doesNeedOperand := true
doesNeedOperator := false

for _, inputChar := range inputString {
    switch inputChar {

        // FOUND OPERAND
        case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':

            // CHECK IF NEEDS OPERATOR INSTEAD
            if doesNeedOperator {
                panic("\n\nERR_21: " + PANICMSG_AAAAAGGHH)
            }

            // LET OPERAND CONTINUE
            isIncompleteOperand = true

            // NEXT CHARACTER COULD BUT NEED NOT BE OPERAND
            doesNeedOperand = false

        // FOUND DECIMAL POINT
        case '.':

            // CHECK IF NEEDS OPERATOR INSTEAD
            if doesNeedOperator {
                panic("\n\nERR_31: " + PANICMSG_AAAAAGGHH)
            }

            // CHECK IF OPERAND ALREADY CONTAINS DECIMAL POINT
            if activeDecimalPoints > 0 {
                panic("\n\nERR_32: " + PANICMSG_AAAAAGGHH)
            }

```

```
// INCREMENT DECIMAL POINT COUNTER
activeDecimalPoints++

// FOUND SPACE CHAR
case ' ':

    // NOTE: NEGATIVE OPERAND FUNCTIONALITY NOT IMPLEMENTED
    if isNegativeOperand {
        panic("\n\nERR_41" + PANICMSG_AAAAAGGHH)
    }

    // IF SPACE FOUND AFTER OPERAND THEN PREVIOUS OPERAND IS COMPLETE
    if isIncompleteOperand {
        doesNeedOperator = true
        doesNeedOperand = false
    }

    // RESET DECIMAL POINT COUNTER
    activeDecimalPoints = 0

// FOUND OPERATOR
case '+', '-', '*', '/':

    // NOTE: NEGATIVE OPERAND FUNCTIONALITY NOT IMPLEMENTED
    if doesNeedOperand && inputChar == '-' {

        // THIS PROBABLY INDICATES NEGATIVE VALUE FOR OPERAND
        panic("\n\nERR_51:  THEY DIDN'T TEACH ME NEGATIVE NUMBERS!  I JUST CAN'T GO ON LIVING!!!\n\n")
    } else {

        // CHECK IF NEEDS OPERAND INSTEAD
        if doesNeedOperand {
            panic("\n\nERR_52: " + PANICMSG_AAAAAGGHH)
        }

        // PREVIOUS OPERAND IS COMPLETE
        isIncompleteOperand = false

        // DOES NOT NEED ANOTHER OPERATOR
        doesNeedOperator = false

        // NEEDS OPERAND NEXT
        doesNeedOperand = true

        // RESET DECIMAL POINT COUNTER
        activeDecimalPoints = 0
    }
}
```

```

    }

// FOUND BEGINPARENS
case '(':

    // NOTE: BEGINPARENS MAY ONLY FOLLOW AN OPERATOR
    if isIncompleteOperand || doesNeedOperator {
        panic("\n\nERR_61: " + PANICMSG_AAAAAGGHH)
    }

    // INCREMENT OPEN PARENTHESES COUNTER
    openParentheses++

    // PREVIOUS OPERAND IS COMPLETE
    isIncompleteOperand = false

    // DOES NOT NEED ANOTHER OPERATOR
    doesNeedOperator = false

    // RESET DECIMAL POINT COUNTER
    activeDecimalPoints = 0

// FOUND ENDPARENS
case ')':

    // CHECK IF NUMBER OF ENDPARENS EXCEEDS NUMBER OF BEGINPARENS
    if openParentheses < 1 {
        panic("\n\nERR_71: " + PANICMSG_AAAAAGGHH)
    }

    // DECREMENT OPEN PARENS COUNTER
    openParentheses--

    // RESET DECIMAL POINT COUNTER
    activeDecimalPoints = 0

    // PREVIOUS OPERAND IS COMPLETE
    isIncompleteOperand = false

    // NEEDS OPERATOR NEXT
    doesNeedOperator = true

// FOUND INVALID CHARACTER
default:
    panic("\n\nERR_81: " + PANICMSG_AAAAAGGHH)
}

```

```

}

switch {
// CHECK IF LINE ENDS WITH OPERATOR INSTEAD OF OPERAND
case doesNeedOperand:
    panic("\n\nERR_91: " + PANICMSG_AAAAAGGHH)
}
}

// PARENTHESIZER PARENTHESIZES.
// SPECIFICALLY, PARENTHESIZER COMPLETES INCOMPLETE PARETHESIZATION.
// IF NUM_BEGINPARENS < NUM_ENDPARENS THEN PANICKER WILL PANIC.
// IF PANICKER PANICS, PARENTHESIZER WILL NOT PARENTHESIZE.
// NOTE: IT'S NOT PANICKER'S JOB TO PARENTHESIZE EXPRESSIONS.
// NOTE: THAT'S WHAT PARENTHESIZER IS FOR.
// ALSO: PARENTHESIZER ENSURES ALL EXPRESSIONS BEGIN WITH A BEGINPARENS.
func parenthesizer(inputString string) (outputString string) {

    // DETERMINE IF FIRST CHAR IS BEGINPARENS
    isFirstCharParens := false
    if inputString[0] == '(' {
        isFirstCharParens = true
    }

    // COUNT OPEN PARENTHESES
    // NOTE: PANICKER ALREADY DID THIS, BUT IT'S NOT PANICKER'S JOB TO FIX EXPRESSIONS
    openParentheses := int(0)

    for _, inputChar := range inputString {
        switch inputChar {
            case '(':
                openParentheses++
            case ')':
                openParentheses--
        }
    }

    // USE DEDICATED ADJUSTMENT STRING FOR A LIGHTWEIGHT APPROACH
    // NOTE: ADJUSTMENT STRING AVOIDS RECOPYING ENTIRE EXPRESSION
    additionalEndParens := string("")
    for index := 0; index < openParentheses; index++ {
        additionalEndParens += ")"
    }

    // ENSURE FIRST CHAR IS BEGINPARENS
    if isFirstCharParens {

```

```

        outputString = inputString + additionalEndParens
    } else {
        outputString = "(" + inputString + additionalEndParens + ")"
    }

    return
}

// PRECEDENCER DOES THE PRECEDENCING.
// SPECIFICALLY, PRECEDENCER WILL RETURN THE PRECEDENCE VALUE OF A GIVEN OPERATOR.
func precedencer(op byte) uint8 {
    switch op {

        // NOTE: BEGINPARENS HAS LOWEST PRECEDENCE. ENDPARENS IS NOT APPLICABLE HERE
        case '(':
            return 0
        case '+', '-':
            return 1
        case '*', '/':
            return 2
        default:
            // CANNOT REMOVE DEFAULT CASE OR SYNTAX ERROR IS INCURRED
            // CANNOT CALL PANICKER (FOR NONEXISTENT THREAT) OR SYNTAX ERROR IS INCURRED
            // THEREFORE, PRETEND THE FOLLOWING STATEMENT IS USEFUL.
            panic("I am a very calm function. I promise I will never panic.")
    }
}

// CALCULATORER CALCULATORS.
// CALCULATORER DOES ALL THE CALCULATING.
func calculatorer() {

    // POP OPERATOR OFF OPERATOR STACK
    op := operatorStack.Pop().(byte)

    // NOTE: ANY BEGINPARENS WILL SIMPLY BE DISCARDED FROM THE STACK BEFORE RETURNING
    if op != '(' {

        // STORE THE REFLECT TYPE OF THE RIGHT OPERAND
        rightType := reflect.TypeOf(operandStack.Top())

        // INITIALIZE RIGHT OPERAND VARIABLES FOR BOTH TYPES
        // NOTE: DEPENDING ON REFLECT TYPE ONLY ONE VARIABLE WILL BE USED
        rightFloat64 := float64(0)
        rightInt := int(0)
    }
}

```

```

switch rightType {

// IF REFLECT TYPE OF RIGHT OPERAND MATCHES REFLECT TYPE OF KNOWN FLOAT:
case reflect.TypeOf(rightFloat64):

    // THEN POP RIGHT OPERAND AS FLOAT
    // NOTE: CANNOT POP OPERAND AS TYPE: REFLECT TYPE OF STACK TOP
    // DOES NOT WORK: right = operandStack.Pop(reflect.TypeOf(operandStack.Top()))
    rightFloat64 = operandStack.Pop().(float64)

// IF REFLECT TYPE OF RIGHT OPERAND MATCHES REFLECT TYPE OF KNOWN INT:
case reflect.TypeOf(rightInt):

    // THEN POP RIGHT OPERAND AS INT
    // NOTE: CANNOT POP OPERAND AS TYPE: REFLECT TYPE OF STACK TOP
    // DOES NOT WORK: left = operandStack.Pop(reflect.TypeOf(operandStack.Top()))
    rightInt = operandStack.Pop().(int)
}

// STORE THE REFLECT TYPE OF THE LEFT OPERAND
leftType := reflect.TypeOf(operandStack.Top())

// INITIALIZE LEFT OPERAND VARIABLES FOR BOTH TYPES
// NOTE: DEPENDING ON REFLECT TYPE ONLY ONE VARIABLE WILL BE USED
leftFloat64 := float64(0)
leftInt := int(0)

switch leftType {

// IF REFLECT TYPE OF LEFT OPERAND MATCHES REFLECT TYPE OF KNOWN FLOAT:
case reflect.TypeOf(leftFloat64):

    // THEN POP LEFT OPERAND AS FLOAT
    // NOTE: CANNOT POP OPERAND AS TYPE: REFLECT TYPE OF STACK TOP
    leftFloat64 = operandStack.Pop().(float64)

// IF REFLECT TYPE OF RIGHT OPERAND MATCHES REFLECT TYPE OF KNOWN INT:
case reflect.TypeOf(leftInt):

    // THEN POP LEFT OPERAND AS INT
    // NOTE: CANNOT POP OPERAND AS TYPE: REFLECT TYPE OF STACK TOP
    leftInt = operandStack.Pop().(int)
}

```

```
// DO CALCULATING BASED ON REFLECT TYPE CASE
// NOTE: MULTIPLICATION AND DIVISION WITH ANY FLOAT64 OPERAND WILL NEVER COERCE TO INT!!!

// REFLECTION CASE: FLOAT64 <OPERATOR> FLOAT64 = FLOAT64
if leftType == reflect.TypeOf(leftFloat64) && rightType == reflect.TypeOf(rightFloat64) {
    switch op {
    case '+':
        operandStack.Push(leftFloat64 + rightFloat64)
    case '-':
        operandStack.Push(leftFloat64 - rightFloat64)
    case '*':
        operandStack.Push(leftFloat64 * rightFloat64)
    case '/':
        if rightFloat64 == 0 {
            panicker("/0") // CANNOT DIVIDE BY 0
        } else {
            operandStack.Push(leftFloat64 / rightFloat64)
        }
    }
}

// REFLECTION CASE: FLOAT64 <OPERATOR> INT = FLOAT64
} else if leftType == reflect.TypeOf(leftFloat64) && rightType == reflect.TypeOf(rightInt) {
    switch op {
    case '+':
        operandStack.Push(leftFloat64 + float64(rightInt))
    case '-':
        operandStack.Push(leftFloat64 - float64(rightInt))
    case '*':
        operandStack.Push(leftFloat64 * float64(rightInt))
    case '/':
        if rightInt == 0 {
            panicker("/0") // CANNOT DIVIDE BY 0
        } else {
            operandStack.Push(leftFloat64 / float64(rightInt))
        }
    }
}

// REFLECTION CASE: INT <OPERATOR> FLOAT64 = FLOAT64
} else if leftType == reflect.TypeOf(leftInt) && rightType == reflect.TypeOf(rightFloat64) {
    switch op {
    case '+':
        operandStack.Push(float64(leftInt) + rightFloat64)
    case '-':
        operandStack.Push(float64(leftInt) - rightFloat64)
    case '*':
        operandStack.Push(float64(leftInt) * rightFloat64)
    case '/':

```



```

        if rightFloat64 == 0 {
            panicker("/0") // CANNOT DIVIDE BY 0
        } else {
            operandStack.Push(float64(leftInt) / rightFloat64)
        }
    }
    // REFLECTION CASE: INT <OPERATOR> INT = INT
} else if leftType == reflect.TypeOf(leftInt) && rightType == reflect.TypeOf(rightInt) {
    switch op {
    case '+':
        operandStack.Push(leftInt + rightInt)
    case '-':
        operandStack.Push(leftInt - rightInt)
    case '*':
        operandStack.Push(leftInt * rightInt)
    case '/':
        if rightInt == 0 {
            panicker("/0") // CANNOT DIVIDE BY 0
        } else {
            operandStack.Push(leftInt / rightInt)
        }
    }
} else {
    // UNNECESSARY STATEMENT FOR NONEXISTENT CONDITION
    panicker("???" )
}
}
}

func main() {

    // NOTE: AN INFINITE LOOP IS THE ONLY WAY TO CALCULATOR PROPERLY.
    // PANICKER WILL BREAK THE LOOP AS NEEDED.
    for true {

        fmt.Println("\n\nPLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:")
        fmt.Print("> ")

        // Read a from Stdin.
        scanner := bufio.NewScanner(os.Stdin)
        scanner.Scan()
        inputLine := scanner.Text()

        fmt.Print("\n")
    }
}

```

```
// NOTE: PANICKER PANICS.
// IF PANICKER DOESN'T PANIC, WE CAN CONTINUE WITH ASSURANCE THAT NOTHING WILL PANIC.
// EXCEPT MAYBE FROM DIVIDE BY ZERO. THAT MIGHT PANIC, WHICH IS NOT PANICKER'S FAULT.
// PANICKER HAD A BAD FEELING ABOUT YOUR DIVISION BUT WAS TOO POLITE TO PANIC AT THE TIME.
// PANICKER PREVENTS THE REST OF THE PROGRAM FROM ATTEMPTING TO CALCULATOR YOUR NONSENSE.
// READ: DO NOT POPULATE STACK OBJECTS UNLESS PANICKER DOES NOT PANIC.
panicker(inputLine)
```

```
// FULLY PARENTHESIZE INPUTLINE
inputLine = parenthesizer(inputLine)
```

```
line := inputLine
```

```
// EVALUATE EXPRESSION IN SINGLE-PASS SCAN
for i := 0; i < len(line); {
```

```
    switch line[i] {
```

```
        // FOUND OPERAND
```

```
        case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.':
```

```
            // INITIALIZE VARIABLES FOR BOTH TYPES
```

```
            // NOTE: BOTH VARIABLES MAY BE USED
```

```
            valueInt := int(0)
```

```
            valueFloat64 := float64(0)
```

```
            // HAS THE DOT BEEN FOUND
```

```
            seenDecimalPoint := false
```

```
            // EXPLICIT FLOAT64 MULTIPLIER FOR FLOATING POINT EVALUATION
```

```
            decimalMultiplier := float64(0.1)
```

```
            // CONTINUE FINDING OPERAND
```

```
            for {
```

```
                if line[i] == '.' {
```

```
                    // FOUND THE DOT
```

```
                    seenDecimalPoint = true
```

```
                    // CONVERT INTEGER VALUE TO FLOAT64
```

```
                    // NOTE: NOW SKIP TO NEXT CHARACTER
```

```
                    valueFloat64 = float64(valueInt)
```

```
                } else if seenDecimalPoint {
```

```

// CONTINUE ACCUMULATING DECIMAL VALUE OF FLOAT64 OPERAND
valueFloat64 = valueFloat64 + (float64(int(line[i]-'0')) * decimalMultiplier)

// RATCHET THE MULTIPLIER
decimalMultiplier /= float64(10)

} else {
// DOT NOT FOUND SO ACCUMULATE THE INTEGER
valueInt = valueInt*10 + int(line[i]-'0')
}

// SKIP TO NEXT CHAR
i++

// IF NEXT CHAR IS EOL OR NOT OPERAND, BREAK THE CONTINUE FINDING OPERAND LOOP
if i == len(line) ||
    !(('0' <= line[i] && line[i] <= '9') || line[i] == '.') {
    break
}

}

// NOTE: BREAK JUST OCCURRED
if seenDecimalPoint {

// DOT WAS FOUND SO PUSH FLOAT64
operandStack.Push(valueFloat64)
} else {

// DOT WAS NOT FOUND SO PUSH INT
operandStack.Push(valueInt)
}

// FOUND OPERATOR
case '+', '-', '*', '/':

// CALCULATOR ALL CURRENT OPERATIONS ALREADY ON THE STACKS PROVIDED THAT:
// 1. THERE ARE STILL OPERATIONS ON THE STACK TO CALCULATOR, AND
// 2. ALL EXISTING OPERATIONS BELONG TO THE CURRENT PARENTHETIC SUBEXPRESSION, AND
// 3. THE EXISTING OPERATION HAS HIGHER PRECEDENCE THAN THE CURRENT FOUND OPERATOR
for !operatorStack.IsEmpty() && operatorStack.Top().(byte) != '(' &&
    precendencer(operatorStack.Top().(byte)) >= precendencer(line[i]) {
    calculatorer()
}

// PUSH THE FOUND OPERATOR ONTO OPERATOR STACK
operatorStack.Push(line[i])

```

```

        // SKIP TO NEXT CHAR
        i++

    // FOUND SPACE CHAR
    case ' ':

        // SKIP TO NEXT CHAR
        // NOTE: IF A SPACE SPLITS AN OPERAND THEN PANICKER WILL PANIC
        // NOTE: IF PANICKER PANICS THEN THIS CODE WILL NOT RUN
        i++

    // FOUND BEGINPARENS
    case '(':

        // PUSH BEGINPARENS TO OPERATOR STACK THEN SKIP TO NEXT CHAR
        operatorStack.Push(line[i])
        i++

    // FOUND ENDPARENS
    case ')':

        // CALCULATOR ALL CURRENT OPERATIONS CURRENTLY ON THE STACKS PROVIDED THAT:
        // 1. THERE ARE STILL OPERATIONS ON THE STACK TO CALCULATOR, AND
        // 2. ALL EXISTING OPERATIONS BELONG TO THE CURRENT PARENTHETIC SUBEXPRESSION
        for !operatorStack.IsEmpty() && operatorStack.Top().(byte) != '(' {
            calculatorer()
        }

        // CALCULATOR THE TOP OPERATION ON THE STACKS PROVIDED THAT:
        // 1. THERE IS STILL AN OPERATION ON THE STACK TO CALCULATOR
        // NOTE: THIS COMBINES THE CURRENT PARENTHETIC SUBEXPRESSION WITH THE PREVIOUS ONE
        if !operatorStack.IsEmpty() {
            calculatorer()
        }

        // SKIP TO NEXT CHAR
        i++
    }
}

// NOTE: END OF EXPRESSION INPUTLINE
// CALCULATOR ALL REMAINING OPERATIONS UNTIL THE OPERATOR STACK IS EMPTY
for !operatorStack.IsEmpty() {
    calculatorer()
}

```

```
// DISPLAY THE PARENTHESESIZED EXPRESSION
fmt.Print(inputLine + " = ")

// INITIALIZE RESULT VARIABLES OF BOTH TYPES
// NOTE: DEPENDING ON REFLECT TYPE ONLY ONE VARIABLE WILL BE USED
resultFloat64 := float64(0)
resultInt := int(0)

// DISPLAY RESULT BASED ON REFLECT TYPE OF STACK TOP
switch reflect.TypeOf(operandStack.Top()) {

// REFLECTION CASE: REFLECT TYPE OF RESULT MATCHES REFLECT TYPE OF KNOWN FLOAT64:
case reflect.TypeOf(resultFloat64):

    // POP RESULT AS FLOAT64 AND DISPLAY TO OUTPUT WITH TYPE
    resultFloat64 = operandStack.Pop().(float64)
    fmt.Print(resultFloat64)
    fmt.Println(" [TYPE: FLOAT64]")

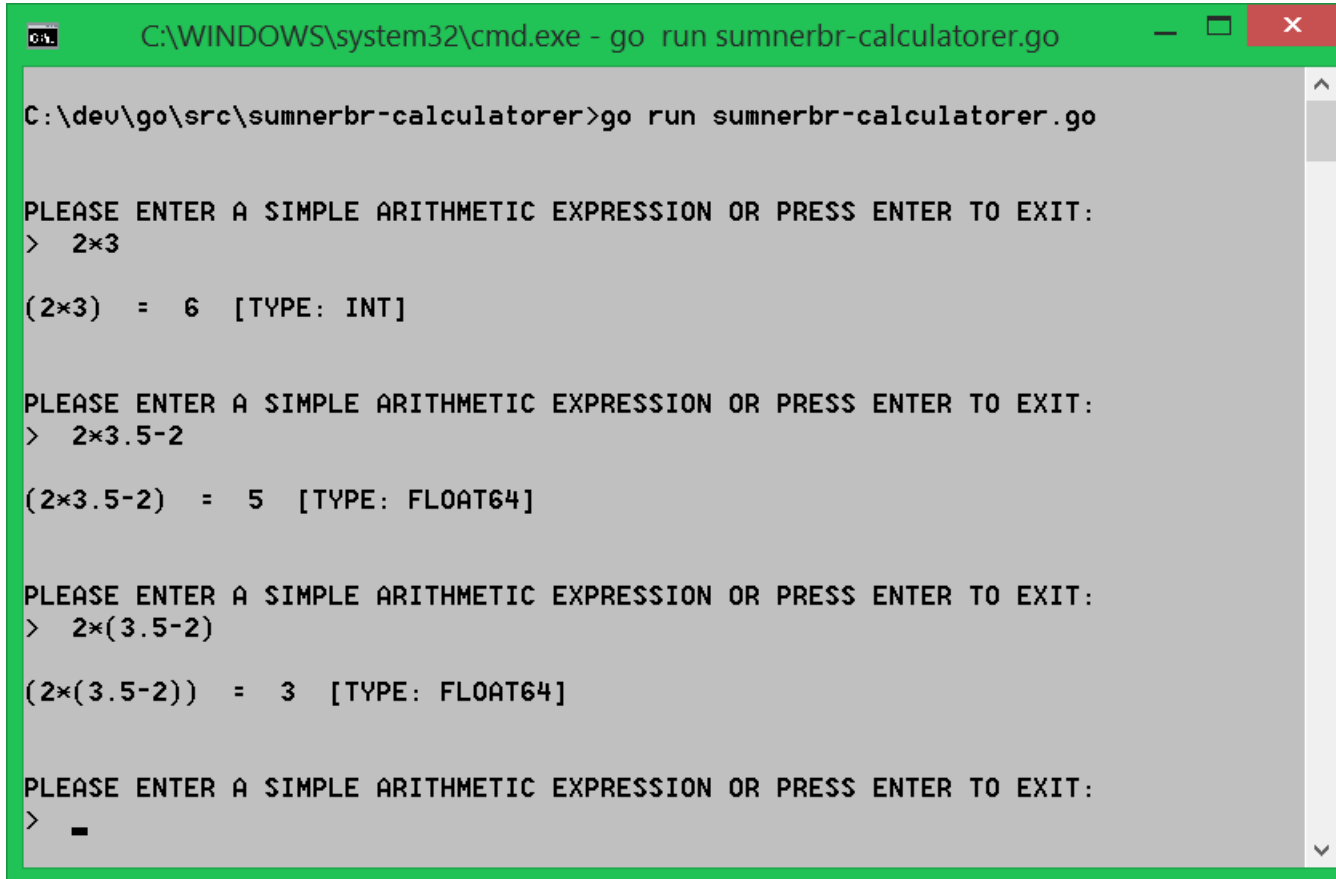
// REFLECTION CASE: REFLECT TYPE OF RESULT MATCHES REFLECT TYPE OF KNOWN INT:
case reflect.TypeOf(resultInt):

    // POP RESULT AS INT AND DISPLAY TO OUTPUT WITH TYPE
    resultInt = operandStack.Pop().(int)
    fmt.Print(resultInt)
    fmt.Println(" [TYPE: INT]")
}
}
```

IV. Source Code: b. Source Code (file: stack.go)

```
package stack
type node struct {
    next *node
    value interface{}
}
type Stack struct {
    top *node
}
func NewStack() Stack {
    return Stack{nil}
}
func (s *Stack) Init() {
    s.top = nil
}
func (s *Stack) Push(v interface{}) {
    s.top = &node{s.top, v}
}
func (s *Stack) Pop() interface{} {
    if s.top == nil {
        return nil
    }
    v := s.top.value
    s.top = s.top.next
    return v
}
func (s Stack) Top() interface{} {
    if s.top == nil {
        return nil
    }
    return s.top.value
}
func (s Stack) IsEmpty() bool {
    return s.top == nil
}
```

V. Sample Runs: a. Simple Example Expressions



```
C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 2*3

(2*3) = 6 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 2*3.5-2

(2*3.5-2) = 5 [TYPE: FLOAT64]

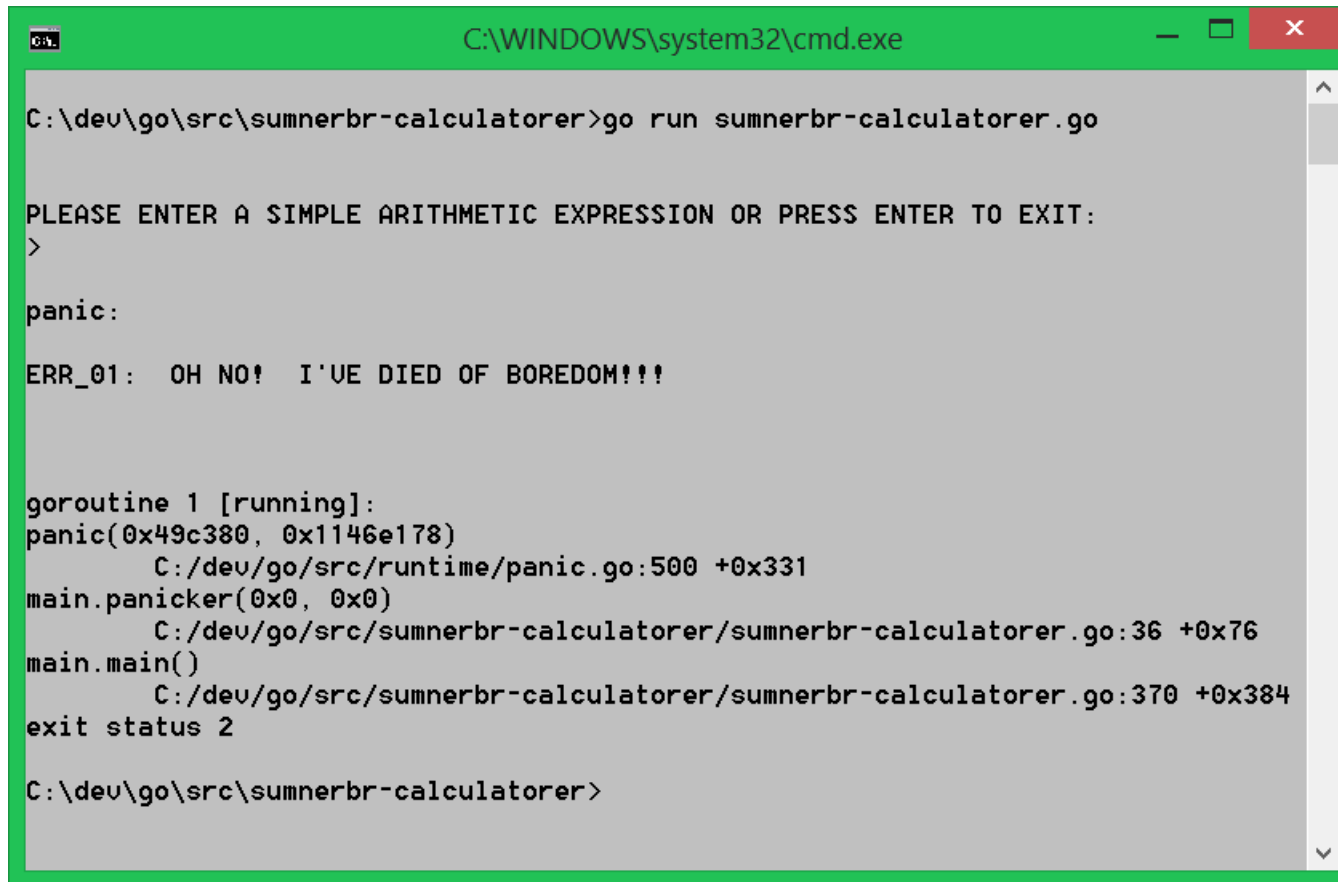
PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 2*(3.5-2)

(2*(3.5-2)) = 3 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> -
```

These are the simple example expressions provided in the assignment specification.

V. Sample Runs: b. Error Checking (panicker: ERR_01)



```
C:\WINDOWS\system32\cmd.exe

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>

panic:

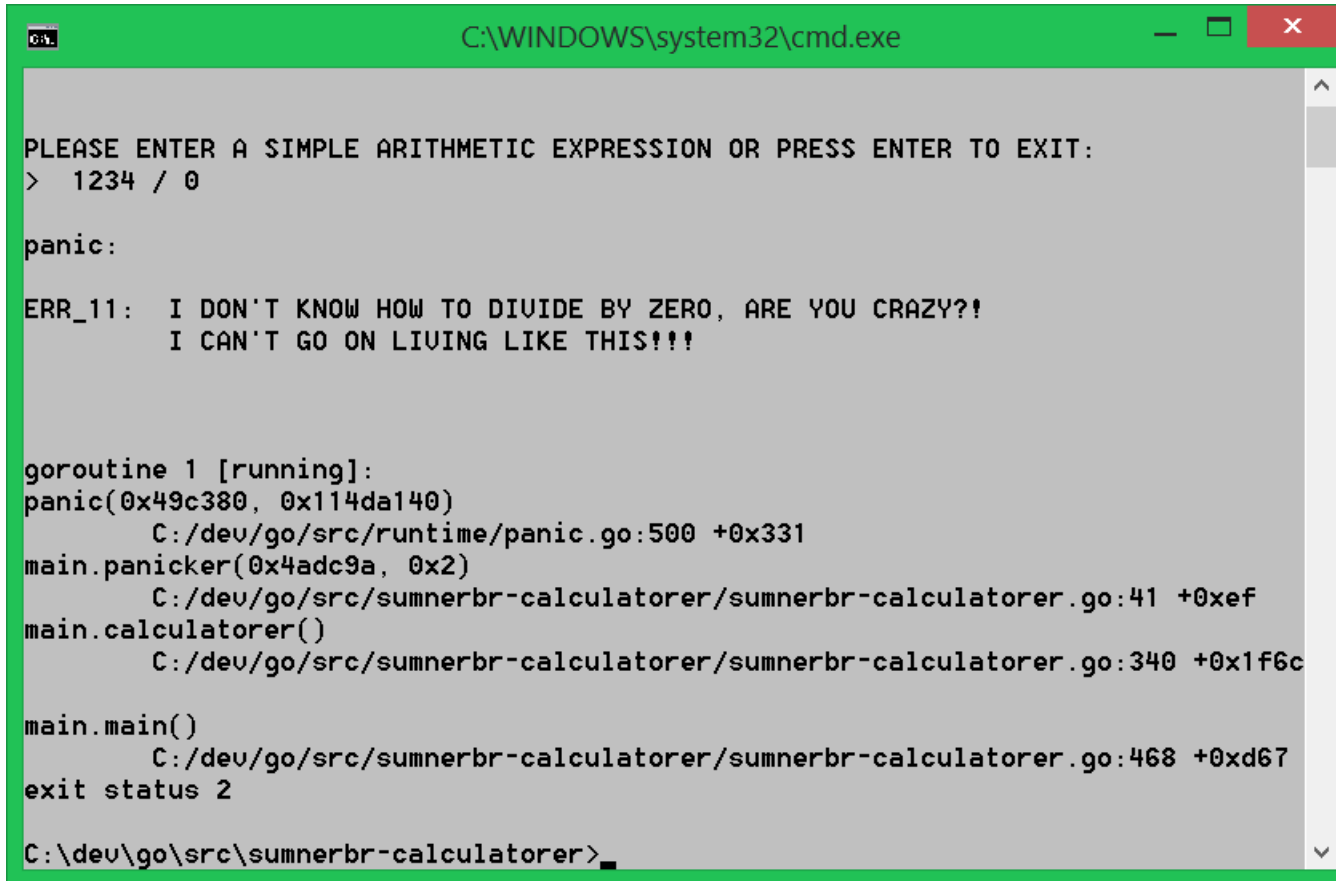
ERR_01:  OH NO!  I'VE DIED OF BOREDOM!!!

goroutine 1 [running]:
panic(0x49c380, 0x1146e178)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x0, 0x0)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:36 +0x76
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering an empty expression.

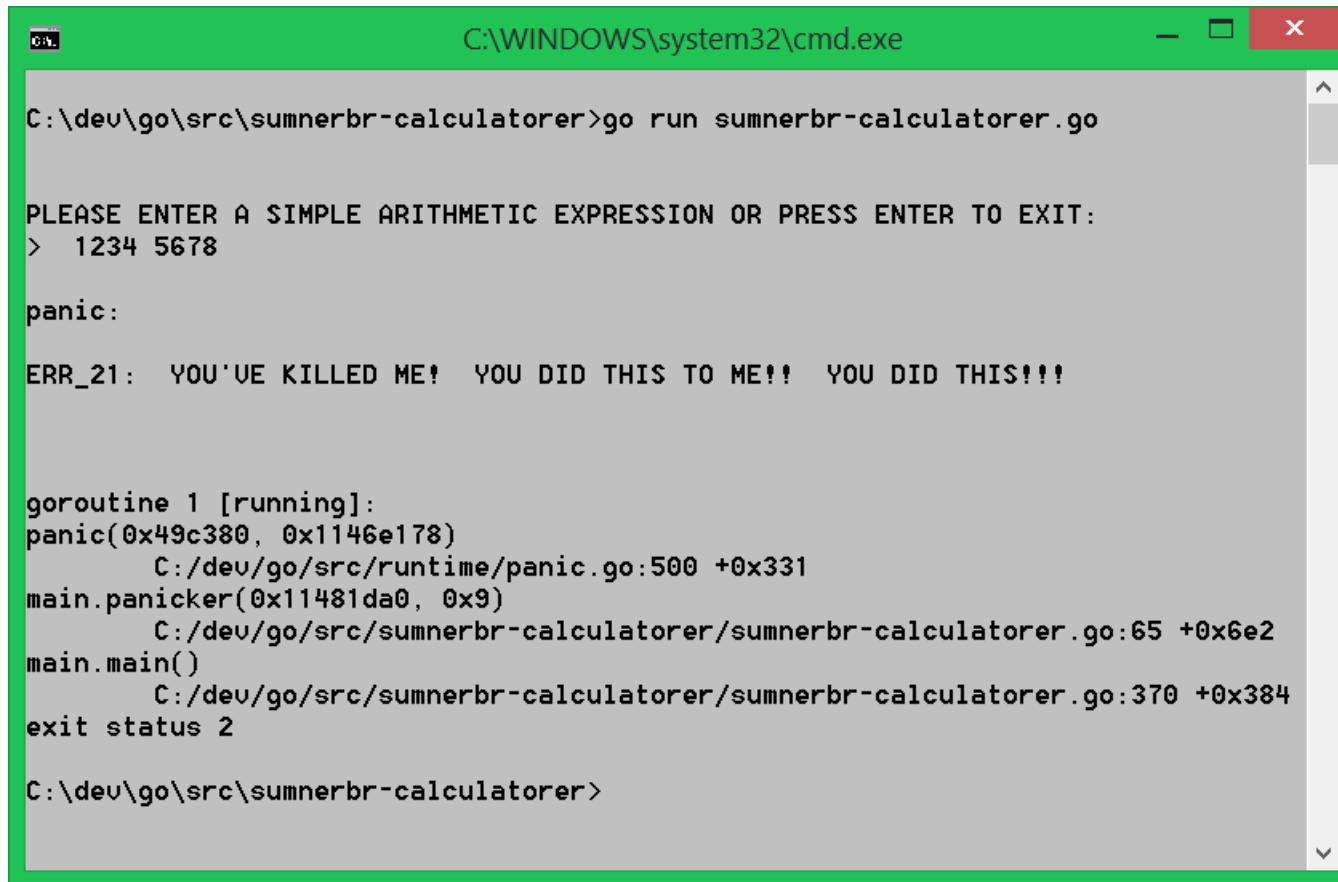
V. Sample Runs: c. Error Checking (panicker: ERR_11)

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a green title bar. The text inside is as follows:

```
PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:  
> 1234 / 0  
  
panic:  
  
ERR_11:  I DON'T KNOW HOW TO DIVIDE BY ZERO, ARE YOU CRAZY?!  
         I CAN'T GO ON LIVING LIKE THIS!!!  
  
goroutine 1 [running]:  
panic(0x49c380, 0x114da140)  
    C:/dev/go/src/runtime/panic.go:500 +0x331  
main.panicker(0x4adc9a, 0x2)  
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:41 +0xef  
main.calculatorer()  
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:340 +0x1f6c  
main.main()  
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:468 +0xd67  
exit status 2  
  
C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering division by zero.

V. Sample Runs: d. Error Checking (panicker: ERR_21)

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a green title bar and standard Windows window controls. The command prompt shows the execution of a Go program named "sumnerbr-calculatorer.go". The user enters the command "go run sumnerbr-calculatorer.go". The program prompts the user to "PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:". The user enters "> 1234 5678". The program then panics, displaying the message "panic:" followed by "ERR_21: YOU'VE KILLED ME! YOU DID THIS TO ME!! YOU DID THIS!!!". Below this, the stack trace is shown, starting with "goroutine 1 [running]:", followed by "panic(0x49c380, 0x1146e178)", "C:/dev/go/src/runtime/panic.go:500 +0x331", "main.panicker(0x11481da0, 0x9)", "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:65 +0x6e2", "main.main()", "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384", and "exit status 2". The command prompt then returns to the prompt "C:\dev\go\src\sumnerbr-calculatorer>".

```
C:\WINDOWS\system32\cmd.exe

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 1234 5678

panic:

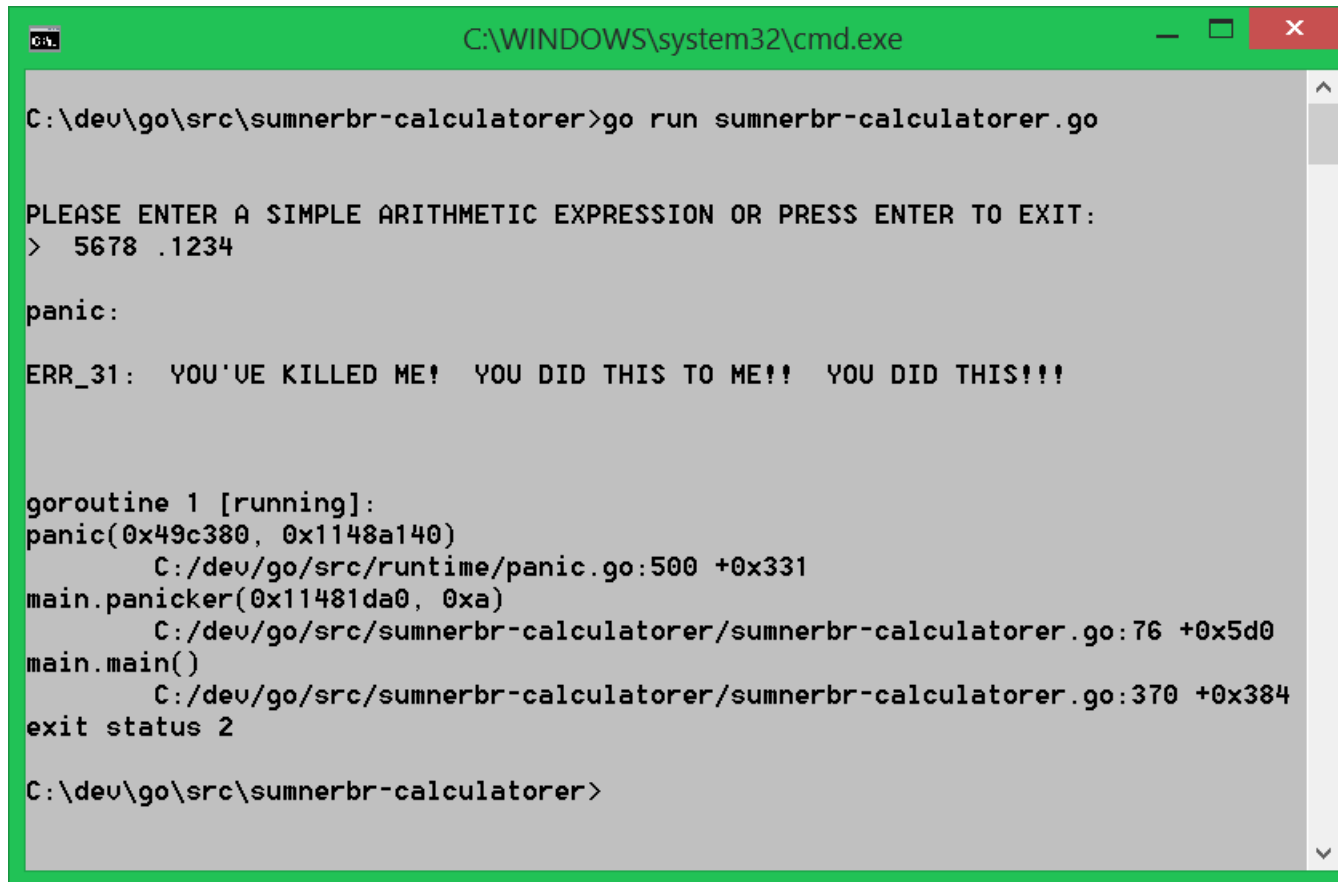
ERR_21:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1146e178)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0x9)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:65 +0x6e2
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering a missing operator (or a non-contiguous operand).

V. Sample Runs: e. Error Checking (panicker: ERR_31)

A screenshot of a Windows command prompt window with a green title bar. The title bar text is "C:\WINDOWS\system32\cmd.exe". The window contains the following text:

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5678 .1234

panic:

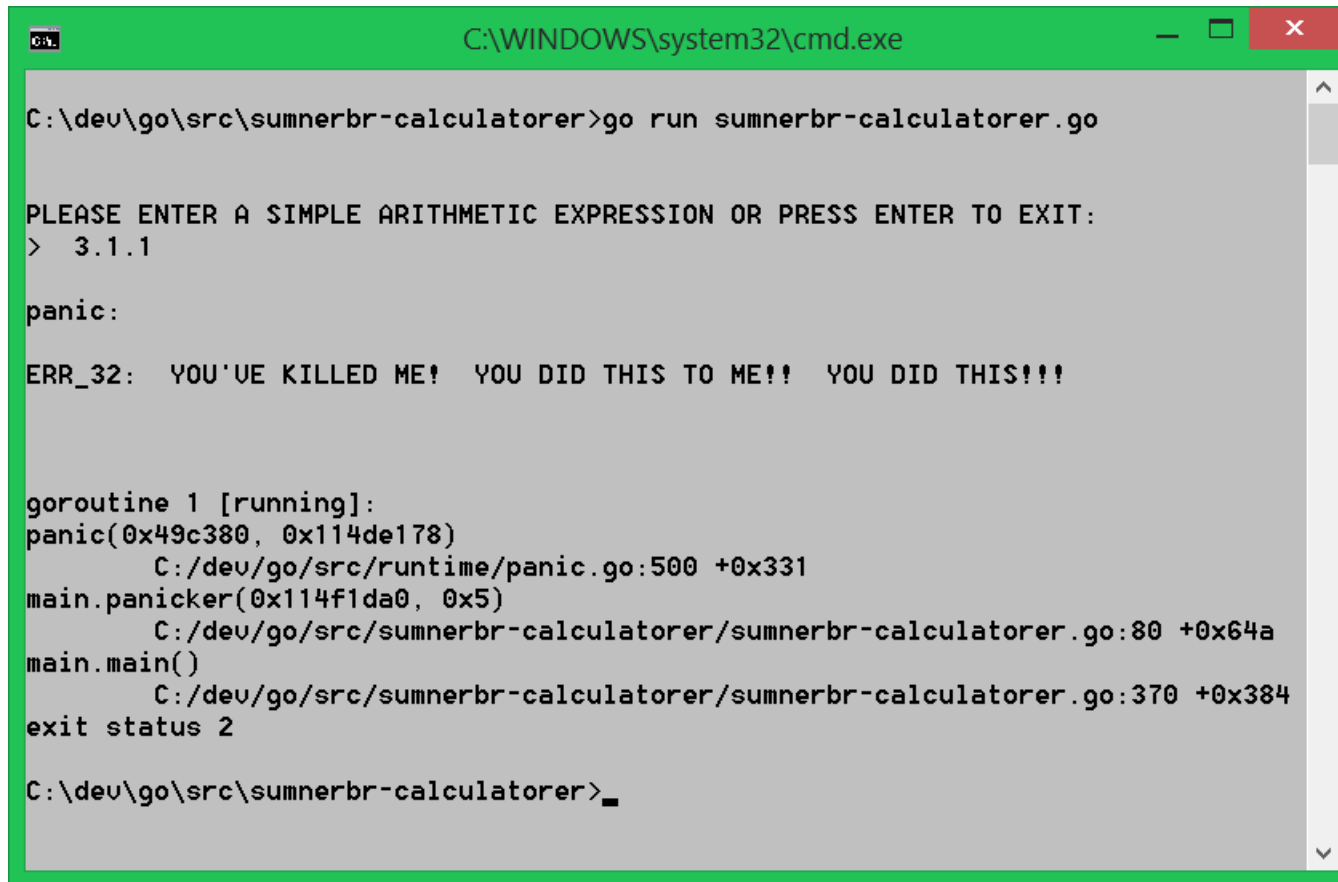
ERR_31:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1148a140)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0xa)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:76 +0x5d0
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering a decimal point instead of an operator.

V. Sample Runs: f. Error Checking (panicker: ERR_32)

A screenshot of a Windows command prompt window with a green title bar. The title bar text is "C:\WINDOWS\system32\cmd.exe". The command prompt shows the execution of a Go program. The user enters "go run sumnerbr-calculatorer.go" at the prompt "C:\dev\go\src\sumnerbr-calculatorer>". The program outputs "PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:" followed by a prompt ">". The user enters "3.1.1". The program then panics, outputting "panic:", "ERR_32: YOU'VE KILLED ME! YOU DID THIS TO ME!! YOU DID THIS!!!", and a stack trace. The stack trace shows the panic occurred in "goroutine 1 [running]:" at "panic(0x49c380, 0x114de178)" in "C:/dev/go/src/runtime/panic.go:500 +0x331", then in "main.panicker(0x114f1da0, 0x5)" in "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:80 +0x64a", then in "main.main()" in "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384", and finally "exit status 2". The command prompt ends with "C:\dev\go\src\sumnerbr-calculatorer>_".

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 3.1.1

panic:

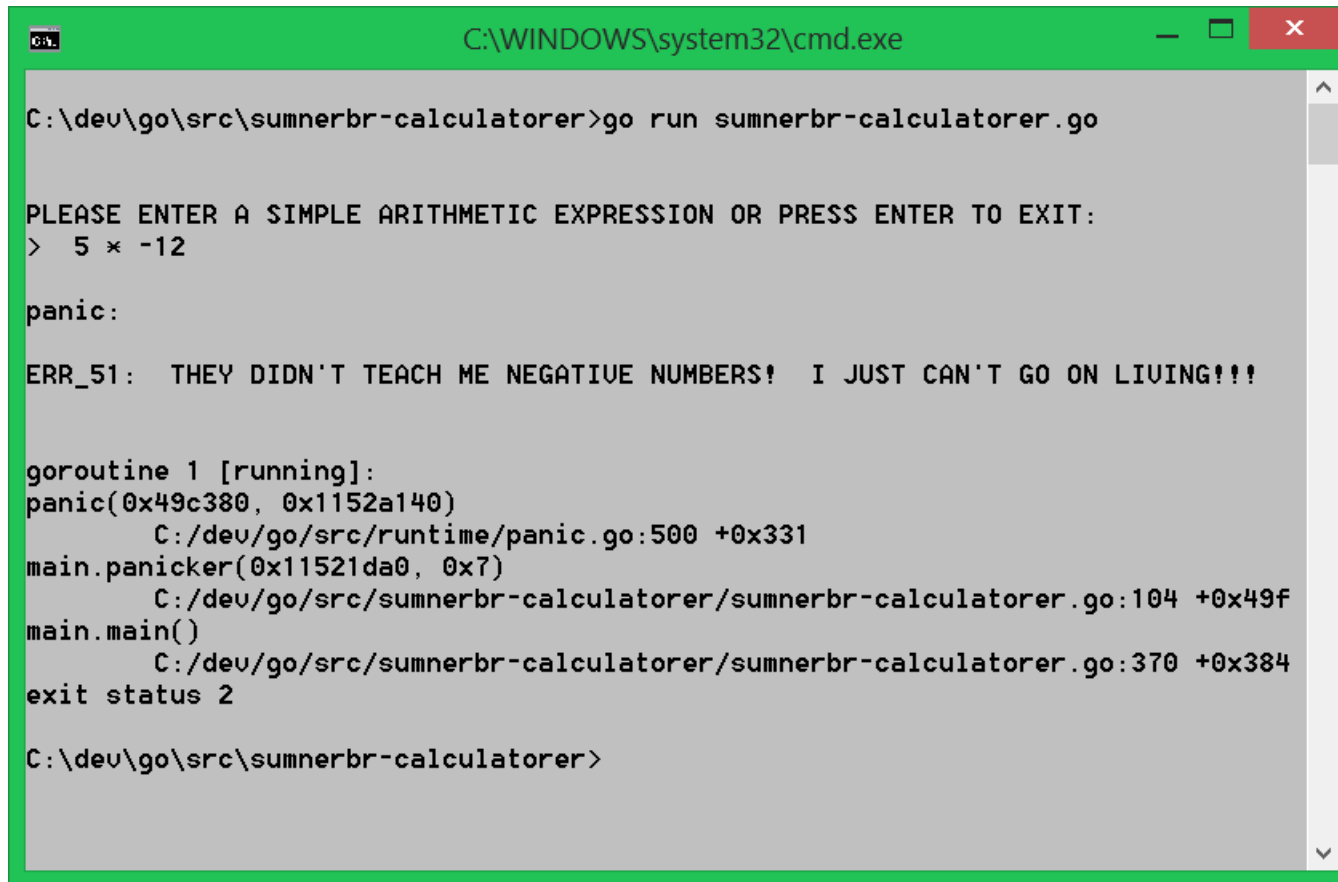
ERR_32:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x114de178)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x114f1da0, 0x5)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:80 +0x64a
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>_
```

Panicker panics on encountering more than one decimal point in an operand.

V. Sample Runs: g. Error Checking (panicker: ERR_51)

A screenshot of a Windows command prompt window with a green title bar. The title bar text is "C:\WINDOWS\system32\cmd.exe". The command prompt shows the execution of a Go program. The user enters "5 * -12" and the program panics with the message "ERR_51: THEY DIDN'T TEACH ME NEGATIVE NUMBERS! I JUST CAN'T GO ON LIVING!!!". The stack trace shows the panic originated in the "panicker" function.

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5 * -12

panic:

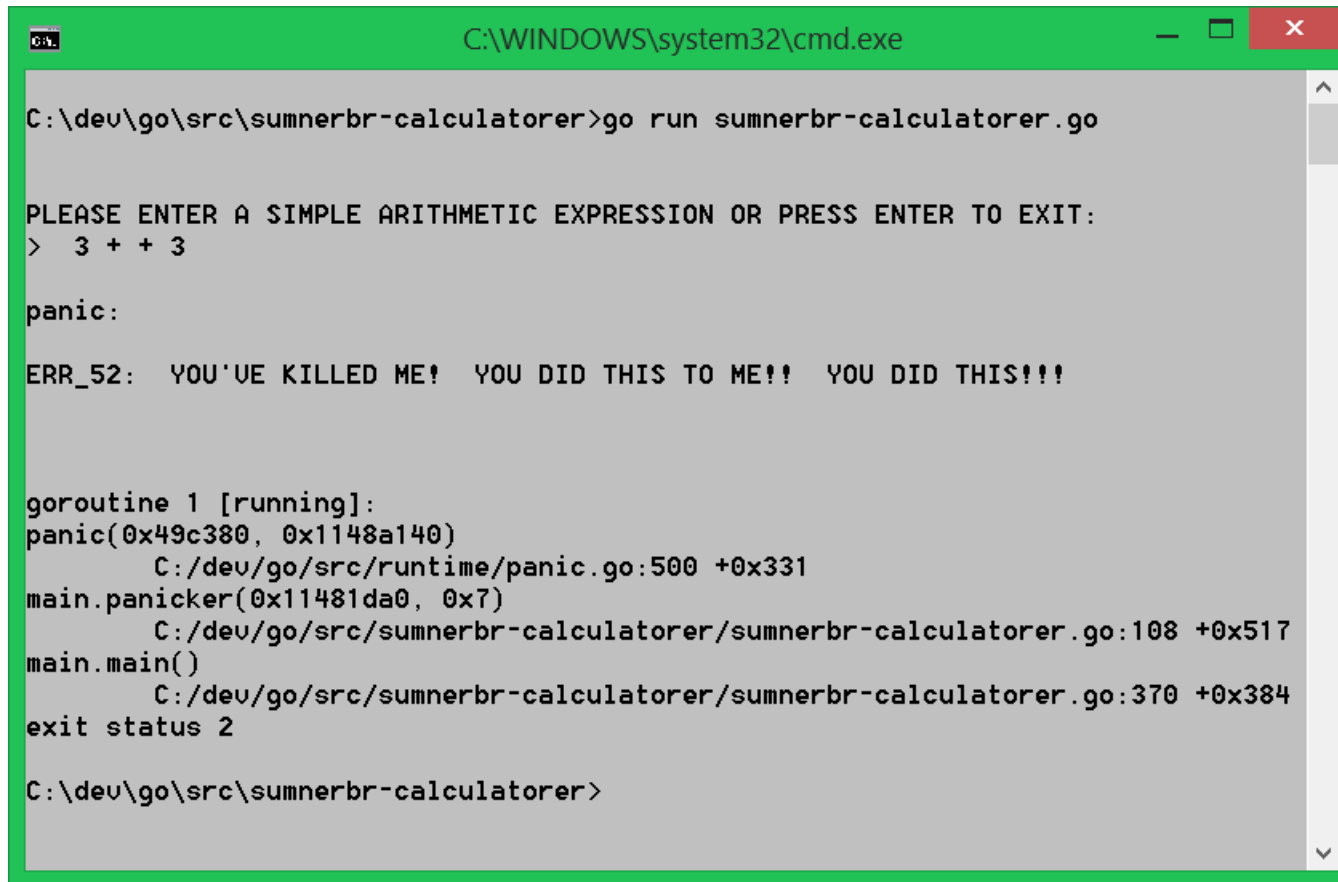
ERR_51:  THEY DIDN'T TEACH ME NEGATIVE NUMBERS!  I JUST CAN'T GO ON LIVING!!!

goroutine 1 [running]:
panic(0x49c380, 0x1152a140)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11521da0, 0x7)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:104 +0x49f
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering a unary negative operator for an operand.

V. Sample Runs: h. Error Checking (panicker: ERR_52)



```

C:\WINDOWS\system32\cmd.exe

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 3 + + 3

panic:

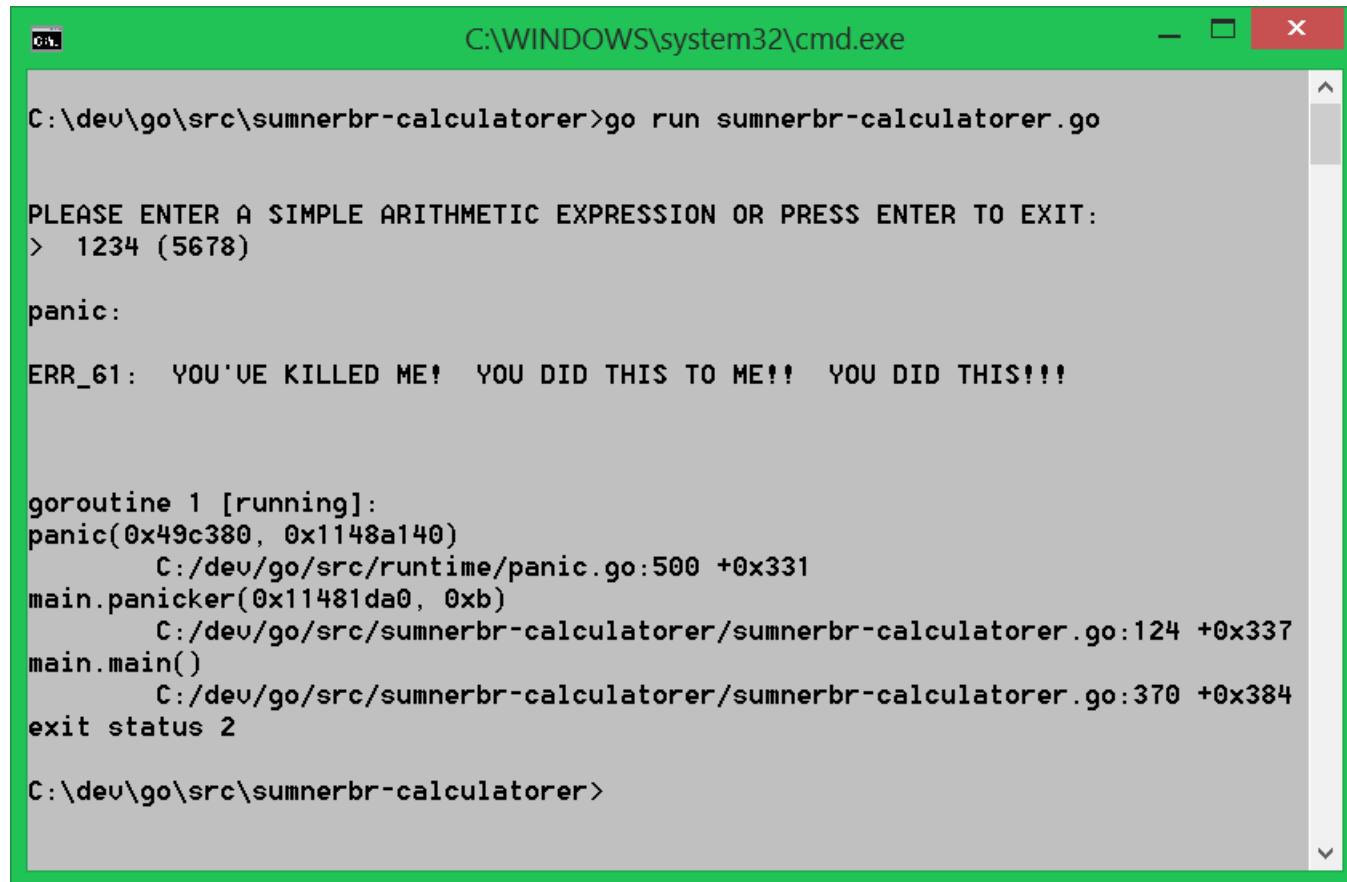
ERR_52:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1148a140)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0x7)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:108 +0x517
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
    
```

Panicker panics on encountering another operator when expecting an operand.

V. Sample Runs: i. Error Checking (panicker: ERR_61)



```
C:\WINDOWS\system32\cmd.exe

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 1234 (5678)

panic:

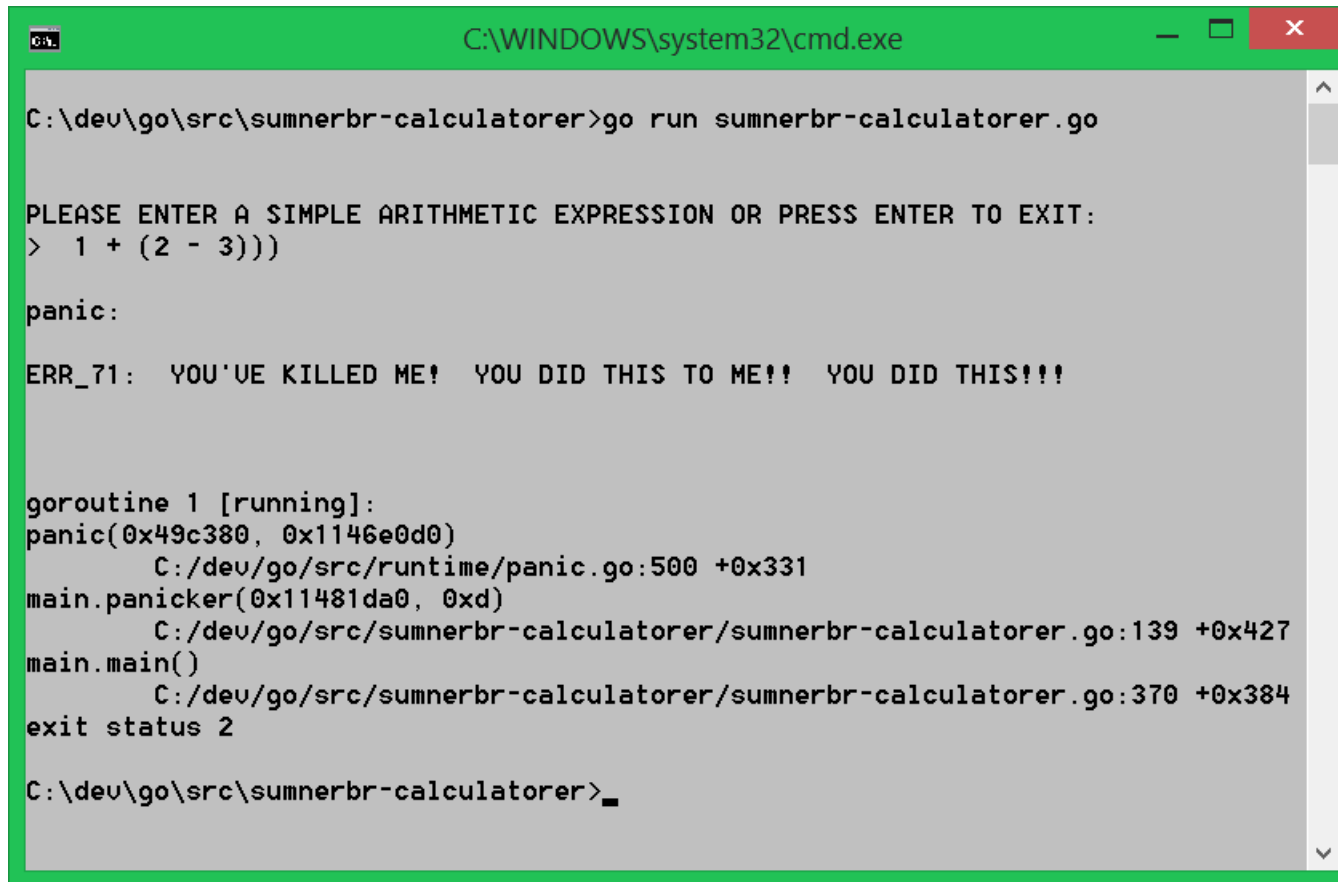
ERR_61:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1148a140)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0xb)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:124 +0x337
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering a beginparens when expecting an operator.

V. Sample Runs: j. Error Checking (panicker: ERR_71)

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a green title bar. The command prompt shows the execution of a Go program. The user enters the command "go run sumnerbr-calculatorer.go" at the prompt "C:\dev\go\src\sumnerbr-calculatorer>". The program outputs "PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:" followed by a prompt ">". The user enters "1 + (2 - 3)))". The program then panics, displaying "panic:" followed by "ERR_71: YOU'VE KILLED ME! YOU DID THIS TO ME!! YOU DID THIS!!!". Below this, the stack trace is shown, starting with "goroutine 1 [running]:", followed by "panic(0x49c380, 0x1146e0d0)", "C:/dev/go/src/runtime/panic.go:500 +0x331", "main.panicker(0x11481da0, 0xd)", "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:139 +0x427", "main.main()", "C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384", and "exit status 2". The prompt "C:\dev\go\src\sumnerbr-calculatorer>" is shown again at the bottom.

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 1 + (2 - 3)))

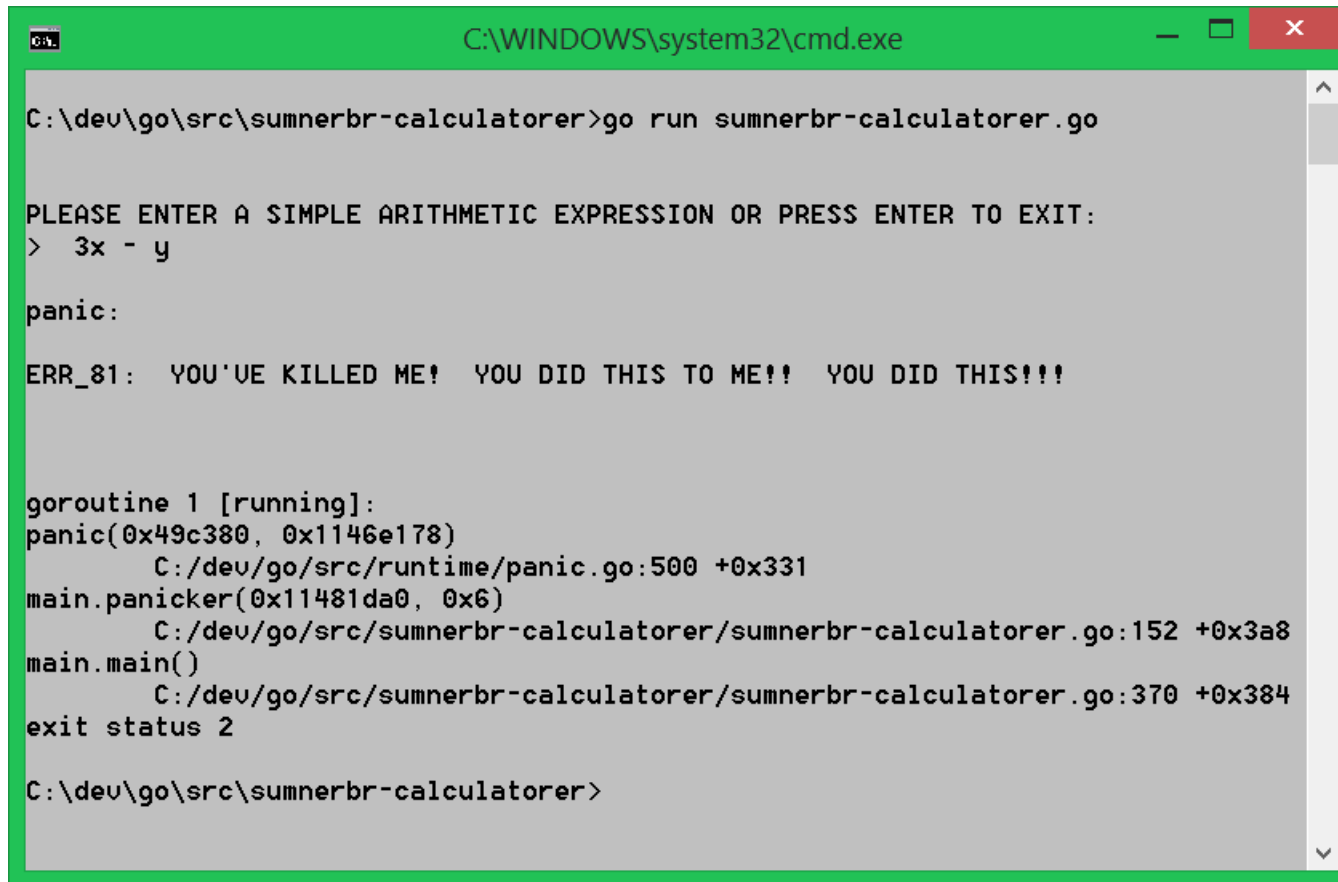
panic:
ERR_71:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1146e0d0)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0xd)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:139 +0x427
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering more endparens than beginparens.

V. Sample Runs: k. Error Checking (panicker: ERR_81)



```
C:\WINDOWS\system32\cmd.exe

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 3x - y

panic:

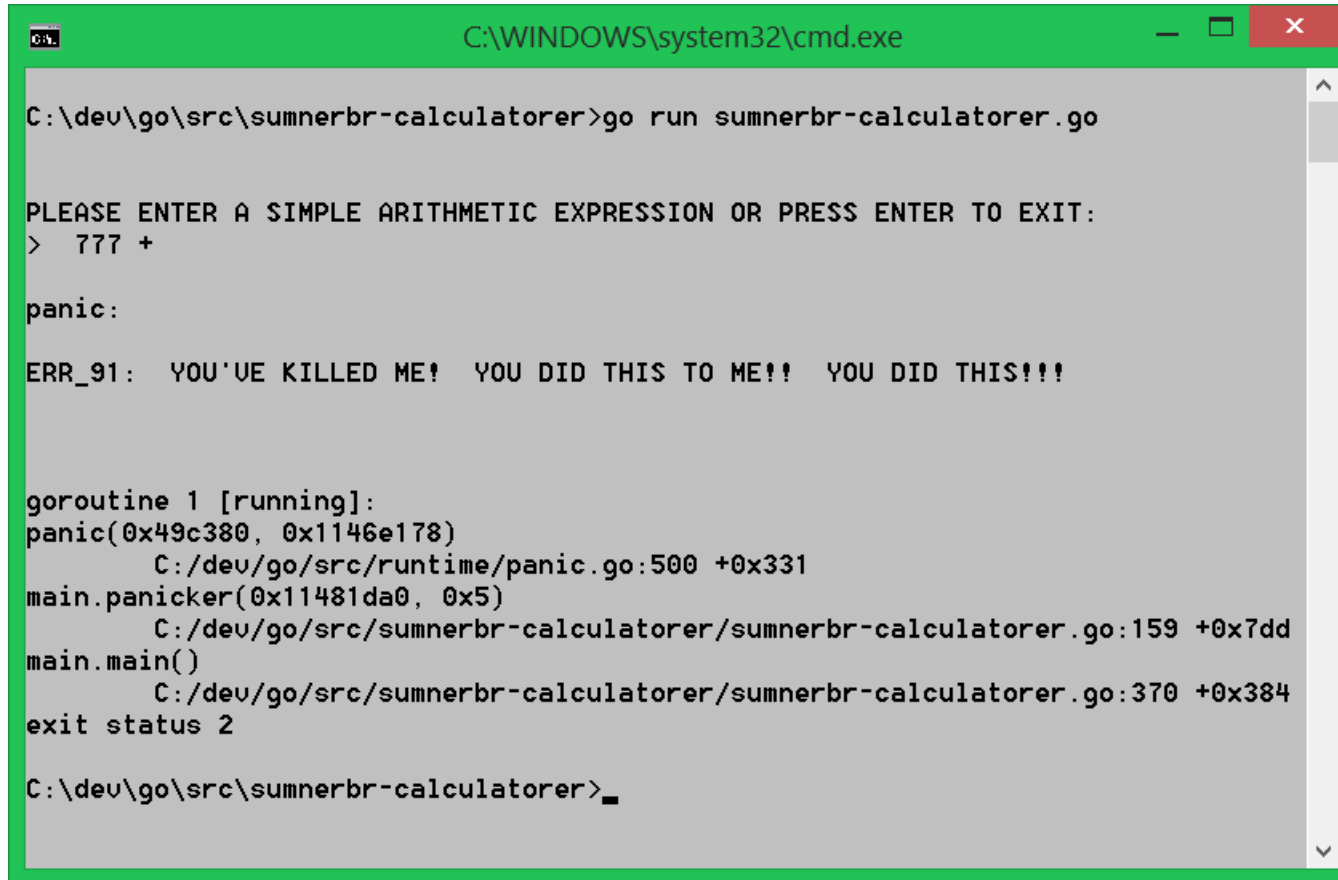
ERR_81:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1146e178)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0x6)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:152 +0x3a8
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>
```

Panicker panics on encountering invalid characters.

V. Sample Runs: 1. Error Checking (panicker: ERR_91)

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a green title bar. The command prompt shows the following text:

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 777 +

panic:

ERR_91:  YOU'VE KILLED ME!  YOU DID THIS TO ME!!  YOU DID THIS!!!

goroutine 1 [running]:
panic(0x49c380, 0x1146e178)
    C:/dev/go/src/runtime/panic.go:500 +0x331
main.panicker(0x11481da0, 0x5)
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:159 +0x7dd
main.main()
    C:/dev/go/src/sumnerbr-calculatorer/sumnerbr-calculatorer.go:370 +0x384
exit status 2

C:\dev\go\src\sumnerbr-calculatorer>_
```

Panicker panics on encountering an expression that ends with an operator instead of an operand.

V. Sample Runs: m. Type Reflection Behaviors (Addition)

```

C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 7.0 + 5

(7.0 + 5) = 12 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 7 + 5.0

(7 + 5.0) = 12 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 7 + 5

(7 + 5) = 12 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>
    
```

float64	+	float64	=	float64
float64	+	int	=	float64
int	+	float64	=	float64
int	+	int	=	int

V. Sample Runs: n. Type Reflection Behaviors (Subtraction)

```

C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 9 - 7

(9 - 7) = 2 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 9 - 7.0

(9 - 7.0) = 2 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 9.88888 - 7

(9.88888 - 7) = 2.888880000000000003 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>
    
```

float64	–	float64	=	float64
float64	–	int	=	float64
int	–	float64	=	float64
int	–	int	=	int

Note: The floating-point discrepancy shown at the bottom is built into the Go Language and can be verified at:
<https://play.golang.org/p/vEw9qUqQWT>

V. Sample Runs: o. Type Reflection Behaviors (Multiplication)

```

C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 7 * 6

(7 * 6) = 42 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 3.5 * 12

(3.5 * 12) = 42 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 28 * 1.5

(28 * 1.5) = 42 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>
    
```

float64	*	float64	=	float64
float64	*	int	=	float64
int	*	float64	=	float64
int	*	int	=	int

V. Sample Runs: p. Type Reflection Behaviors (Division)

```

C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5.0 / 2.0

(5.0 / 2.0) = 2.5 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5.0 / 2

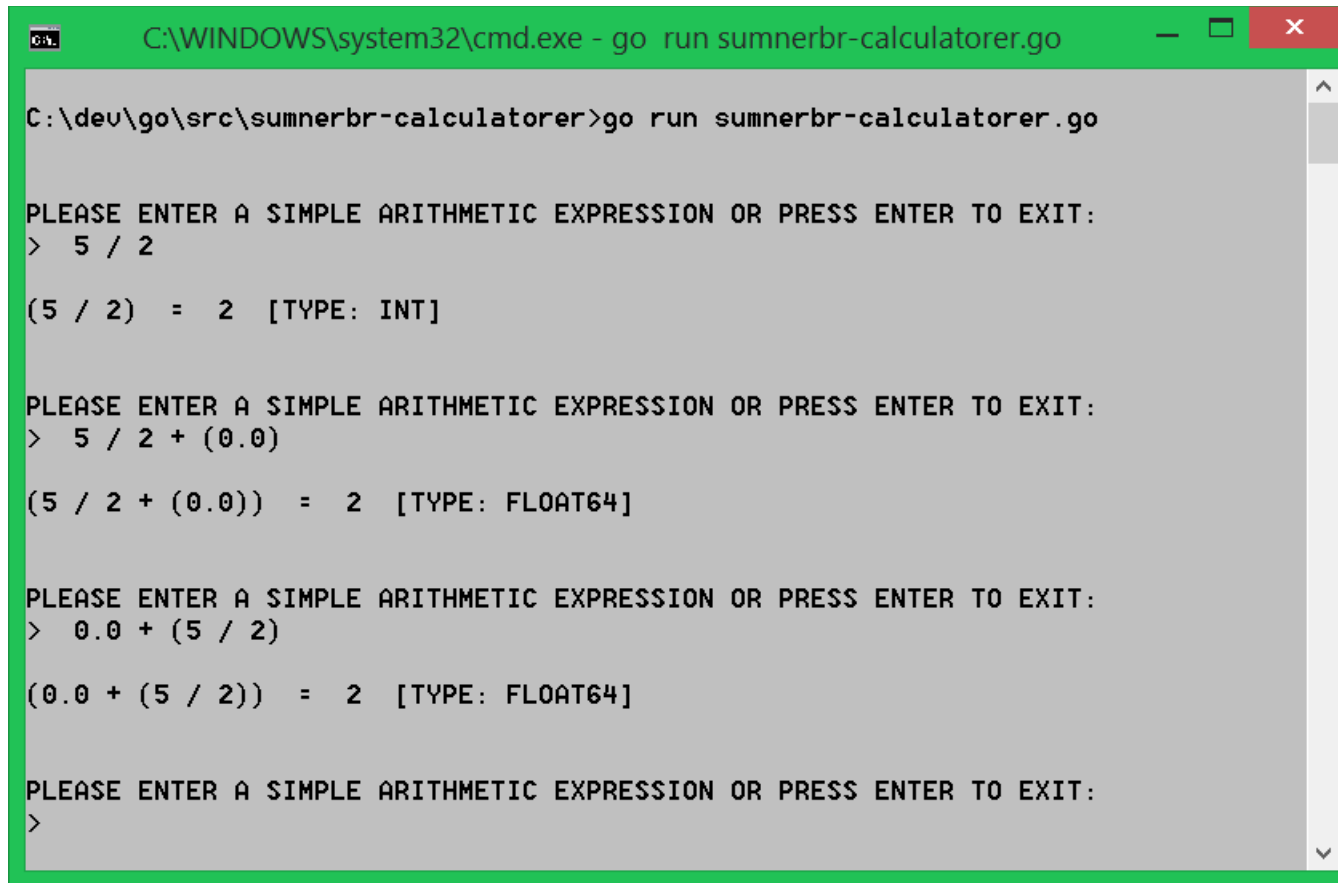
(5.0 / 2) = 2.5 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5 / 2.0

(5 / 2.0) = 2.5 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> _
    
```

float64 / float64 = float64
 float64 / int = float64
 int / float64 = float64



The screenshot shows a Windows command prompt window with a green title bar. The title bar text is "C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go". The command prompt shows the execution of the program and its output for three different arithmetic expressions. The output for each expression includes the result and its type in brackets.

```
C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5 / 2

(5 / 2) = 2 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 5 / 2 + (0.0)

(5 / 2 + (0.0)) = 2 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 0.0 + (5 / 2)

(0.0 + (5 / 2)) = 2 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>
```

$$\begin{aligned} \text{int} / \text{int} &= \text{int} \\ (\text{int} / \text{int} + (\text{float64})) &= (\text{int} + \text{float64}) = \text{float64} \\ (\text{float64} + (\text{int} / \text{int})) &= (\text{float64} + \text{int}) = \text{float64} \end{aligned}$$

V. Sample Runs: q. Parenthetical / Formatting Behaviors

```

C:\WINDOWS\system32\cmd.exe - go run sumnerbr-calculatorer.go

C:\dev\go\src\sumnerbr-calculatorer>go run sumnerbr-calculatorer.go

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> 1 * 2+ ( 3.0/4 )-( ( 5* 6 + ( 7 -80
(1 * 2+ ( 3.0/4 )-( ( 5* 6 + ( 7 -80))) = 45.75 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> ((5*6+(7-80
((5*6+(7-80))) = -43 [TYPE: INT]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
> (1*2+(3.0/4)) - (0 -43)
(1*2+(3.0/4)) - (0 -43) = 45.75 [TYPE: FLOAT64]

PLEASE ENTER A SIMPLE ARITHMETIC EXPRESSION OR PRESS ENTER TO EXIT:
>
    
```

Calculatorer disregards all space characters provided that panicker does not panic.
 Parenthesizer completes incomplete parenthesization.
 Stack-based algorithm processes parenthesized subexpressions properly.
 Negative operands can be expressed as being subtracted from zero within subexpressions.

VI. Conclusions:

The Go Language is pleasant to use and is similar to a cross between Java and Python. The hidden nature of the statement-terminating semicolons is refreshing, however the resulting bracket style that is forced on the programmer is not of this author's preference. It is convenient that Go programs can be either installed into executables or run directly from the command line. The most difficult part of getting GoLang to begin compiling programs seemed to be setting the environment variables correctly and having the correct files in the correct directories.

One unexpected conclusion about the Go Language is that it performs floating-point arithmetic poorly using both 32-bit and 64-bit values. This can be seen at the bottom of Sample Run: n. on p.036. Floating-point discrepancies of this form are alarmingly common and can easily be reproduced within the Go Playground at: <https://play.golang.org> (the following example is provided: <https://play.golang.org/p/vEw9qUqQWT>). Nevertheless, despite this deficiency GoLang is still highly recommended as a general-purpose language.