CS 270 - Lab 16

M. Boady, B. Char, J. Johnson, G. Long, S. Earth

1 Introduction

You may work in teams of **one** or **two** students. Submit one copy for the entire group.

Write your answers on this lab sheet. Only what is written on this lab sheet will be graded.

This lab is due at the end of the class period. You may not continue to work on it once class has ended.

This lab contains 4 questions.

Grading

- 25 points - Putting everyones names on this page
- 20 points - Earned for each correct question (Answer is fully correct)
- 5 points - Earned for **partial credit** any question

No additional point amounts can be earned. You cannot earn 7 points on a question for example.

The maximum score for a lab is 100. If you get everything correct, that adds up to 105 points but will be reduced to 100.

A question will be marked correct as long as it covers all requirements of the question. It does not need to be perfect, but must be fully correct. A single typo or very minor issue where the intention is clear and all requirements are met would still earn full points.

We want you to complete questions fully, not try to earn partial credit on multiple questions. You may ask your Professor/Course assistant questions during lab.

Labs must be done in the presence of an instructor and/or course assistant or credit will not be given.

Partners should alternate each class day which person is physically typing and submitting the lab.

Do not split up the problems or you risk not finishing on time due to the cumulative nature of the questions.

Enter the name of the student in the group

Member 1 (submitter): Elan Rubin

Member 2: Joshua Koo

Question 1 :

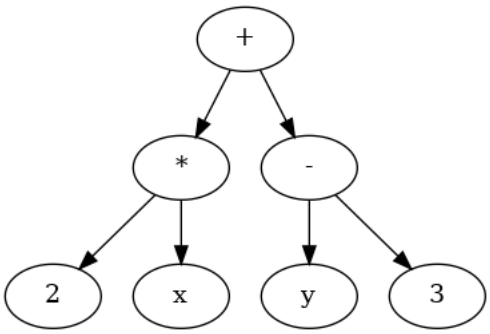
We can represent expressions (boolean or mathematical) as trees.

We define the rules for a **mathtree** below.

```
mathtree := term | (op mathtree mathtree)
term := integer | variable
op := * | + | -
```

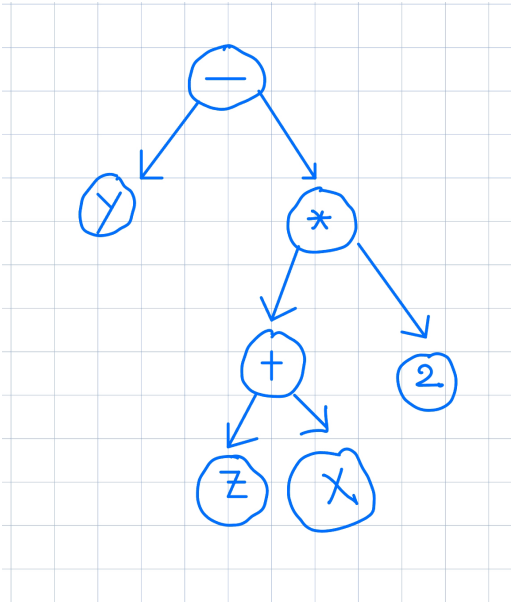
The expression (+ (* 2 x) (- y 3)) is a mathtree. We can show it matches the rules by doing a derivation. We can also draw it as a tree.

```
mathtree = (op mathtree mathtree) ; Second Option mathtree
         = (+ mathtree mathtree) ;Second Option op
         = (+ (op mathtree mathtree) (op mathtree mathtree)) ;Second Option mathtree
         = (+ (op term term) (op term term)) ;First Option mathtree
         = (+ (* term term) (op term term)) ;First Option op
         = (+ (* term term) (- term term)) ;Third Option op
         = (+ (* 2 term) (- term 3)) ; First Option term
         = (+ (* 2 x) (- y 3)) ; Second Option term
```



Derive the following expression using the rules, then draw the tree.

(- y (* (+ z x) 2))



mathtree ;second option mathtree
= (op mathtree mathtree) ;second option mathtree
= (- mathtree mathtree) ;third option op
= (- term mathtree) ;first option mathtree
= (- y mathtree) ;second option term
= (- y (op mathtree mathtree)) ;second option mathtree
= (- y (* mathtree mathtree)) ;first option op
= (- y (* (op mathtree mathtree) term)) ;second option mathtree
= (- y (* (+ mathtree mathtree) term)) ;second option op
= (- y (* (+ term term) term)) ;first option mathtree
= (- y (* (+ z x) term)) ;second option term
= (- y (* (+ z x) 2)) ;first option term

Question 2 :

Write a racket function to count the number of **operators** that appear in an expression.

Your function **must** be recursive. The function arguments and contracts are given below.

```
;input-contract: exp is a mathtree
;output-contract: (countOp exp) is a integer
;    with the number of operators in the mathtree
(define (countOp exp) ... )
```

You can test using the following. **Hint**: **list**? asks if something is a list.

```
(countOp 'x ) ; Returns 0
(countOp 7 ) ; Returns 0
(countOp '(+ x 7) ) ; Returns 1
(countOp '(- x y) ) ; Returns 1
(countOp '(+ (* 2 x) (- y 2)) ) ; Returns 3
(countOp '(+ x (- y (* 2 (+ z x)))) ) ; Returns 4
```

Write your Racket function below.

```
(define (countOp exp)
  (cond
    [(not (list? exp)) 0]
    [(null? exp) 0]
    [else (+ 1
             (countOp (first (rest exp)))
          (countOp (first (rest (rest exp)))))]))
```

Question 3 :

Write a racket function to count the number of **terms** that appear in an expression.

An **terms** is defined as a number or a variable.

Your function **must** be recursive. The function arguments and contracts are given below.

```
;input-contract: exp is a mathtree
;output-contract: (countTerms exp) is a integer
;    which is the number of terms in the mathtree
(define (countTerms exp) ... )
```

You can test using the following. Just count the number of terms that appear, do not try and count **unique** terns. For example, (+ x x) has two terms.

```
(countTerms 'x);Returns 1
(countTerms 7 ) ; Returns 1
(countTerms '(+ x 7) ) ; Returns 2
(countTerms '(- x y) ) ; Returns 2
(countTerms '(+(* 2 x)(- y 2)));Returns 4
(countTerms '(+ x (- y (* 2 (+ z x)))) ) ; Returns 5
```

Write your Racket function below.

```
(define (countTerms exp)
  (if (list? exp)
      (+ (countTerms (first (rest exp))) (countTerms (first (rest (rest exp)))))
      1))
```

Question 4 :

Write a racket function that replaces a variables with a number in a mathtree.

Your function **must** be recursive. The function arguments and contracts are given below.

```
;input-contract: exp is a mathtree
;    var is a single variable
;    val is an integer
;output-contract: (setVar exp var val) is a mathtree
;     with var replaced by val
(define (setVar exp var val)
```

You can test using the following.

```
(setVar 'x 'x 2) ; Returns 2
(setVar 7 'x 2) ; Returns 7
(setVar '(+ 9 7) 'x 9) ; Returns '(+ 9 7)
(setVar '(- x y) 'y 3) ; Returns '(- x 3)
(setVar '(+ (* 2 x) (- y 2)) 'y 9) ; Returns '(+ (* 2 x) (- 9 2))
(setVar '(+ x (- y (* 2 (+ z x)))) 'x 12) ; Returns '(+ 12 (- y (* 2 (+ z 12))))
;You can ask Racket to Evaluate the Expressions
;Once you set all the variables
(define ns (make-base-namespace));Required by eval
(eval (setVar '(+ 9 7) 'x 9) ns);Returns 16
```

Write your Racket function below.

```
(define (setVar exp var val)
 (if (list? exp)
    (cons (first exp) (cons (setVar (first (rest exp)) var val) (cons (setVar (first (rest (rest exp))) var val) null)))
    (if (equal? exp var)
       val
       exp)))
```