York University
CSE 2011 – Assignment 3
Summer 2015 Instructor: Jeff Edmonds

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

Family Name: _____ Given Name: _____

Student #: _____ CSE Email: _____

1) Evaluate		
2) Differenentiate		
3) Simplify		
Total	100	

Your task will be to write and test a program that allows a user to

1. evaluate,
2. differentiate,
3. simplify polynomials represented by binary trees.

1 Representing Expressions with Trees

We will now consider how to represent multivariate expressions using binary trees. We will develop the algorithms to evaluate, copy, differentiate, simplify, and print such an expression. Though these are seemingly complex problems, they have simple recursive solutions.

Recursive Definition of an Expression: An Expression is either:

- Single variables "x", "y", and "z" and single real values are themselves examples of expressions.
- If f and g are expressions then f+g, f-g, f*g, and f/g are also expressions.

Tree Data Structure: The recursive definition of an expression directly mirrors that of a binary tree. Because of this, a binary tree is a natural data structure for storing an expression. (Conversely, you can use an expression to represent a binary tree.)

Copy Tree: If you want to make a copy of a tree, you might be tempted to use the code $treeCopy = tree$. However, the effect of this will only be that both the variables *treeCopy* and *tree* refer to the same tree data structure that *tree* originally did. This would be sufficient if you only want to have read access to the data structure from both variables. However, if you want to modify one of the copies, then you need a completely separate copy. To obtain this, the copy routine must allocate memory for each of the nodes in the tree, copy over the information in each node, and link the nodes together in the appropriate way. The following simple recursive algorithm, $treeCopy = Copy(tree)$, accomplishes this.

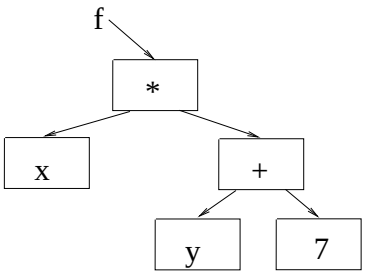
algorithm *Copy*(*tree*)

⟨***pre−cond***⟩**:** *tree* is a binary tree.

⟨***post−cond***⟩**:** Returns a copy of the tree.

```
begin
    if( tree = emptyTree ) then
        result( emptyTree )
    else
        treeCopy = allocate memory for one node
        treeCopy.rootInfo = tree.rootInfo        % copy overall data in root node
        treeCopy.left = Copy(tree.left)          % copy left subtree
        treeCopy.right = Copy(tree.right)        % copy right subtree
        result( treeCopy )
    end if
end algorithm
```

Example 1.1 Evaluate Expression: This routine evaluates an expression that is represented by a tree. For example, it can evaluate $f = x * (y + 7)$, with $xvalue = 2$, $yvalue = 3$, and $zvalue = 5$ and return $2 * (3 + 7) = 20$.



Code:

algorithm *Eval*($f, xvalue, yvalue, zvalue$)

⟨***pre−cond***⟩**:** f is an expression whose only variables are x, y, and z. $xvalue$, $yvalue$, and $zvalue$ are the three real values to assign to these variables.

⟨***post−cond***⟩**:** The returned value is the evaluation of the expression at these values for x, y, and z. The expression is unchanged.

```
begin
    if( f = a real value ) then
        result( f )
    else if( f = "x" ) then
        result( xvalue )
    else if( f = "y" ) then
        result( yvalue )
    else if( f = "z" ) then
        result( zvalue )
    else if( rootOp(f) = "+" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
              +Eval(rightSub(tree), xvalue, yvalue, zvalue) )
    else if( rootOp(f) = "-" ) then
        result( Eval(leftSub(tree), xvalue, yvalue, zvalue)
              −Eval(rightSub(tree), xvalue, yvalue, zvalue) )
```

else if($rootOp(f) =$ "*") then
result($Eval(leftSub(tree), xvalue, yvalue, zvalue)$
$\times Eval(rightSub(tree), xvalue, yvalue, zvalue)$)
else if($rootOp(f) =$ "/") then
result($Eval(leftSub(tree), xvalue, yvalue, zvalue)$
$/ Eval(rightSub(tree), xvalue, yvalue, zvalue)$)
end if
end algorithm

Example 1.2 Differentiate Expression: This routine computes the derivative of a given expression with respect to an indicated variable.

Specification:

Preconditions: The input consists of $\langle f, x \rangle$, where f is an expression represented by a tree and x is a string giving the name of a variable.

Postconditions: The output is the derivative $d(f)/d(x)$. This derivative should be an expression represented by a tree whose nodes are separate from those of f. The data structure f should remain unchanged. See Figure 2.

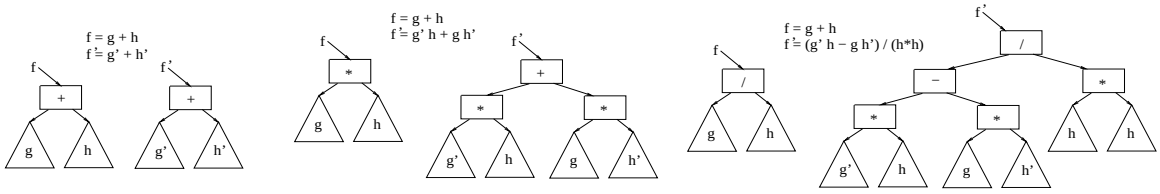


Figure 1: The dirivatives of $g + h$, $g \times h$, and g/h.

Code:

```
algorithm Derivative(f,x)
<pre-cond>: f is an equation and x is a variable
<post-cond>: The derivative of f with respect to x is returned.
   if( f = ‘‘x’’ ) then
      result( 1, constructed by
              -- 1  )

   else if( f = a real value or a single variable other than ‘‘x’’ ) then
      result( 0, constructed by
              -- 0  )
   end if

   % if f is of the form (g op h)
   g  = Copy( leftSub(f) )  % Copy needed for ‘‘*’’ and ‘‘/’’.
   h  = Copy( rightSub(f) ) % Three copies needed for ‘‘/’’.
   g' = Derivative( leftSub(f),  x )
   h' = Derivative( rightSub(f), x )


   if   ( f = g+h ) then
      result( g'+h', constructed by
                    |-- h'
```

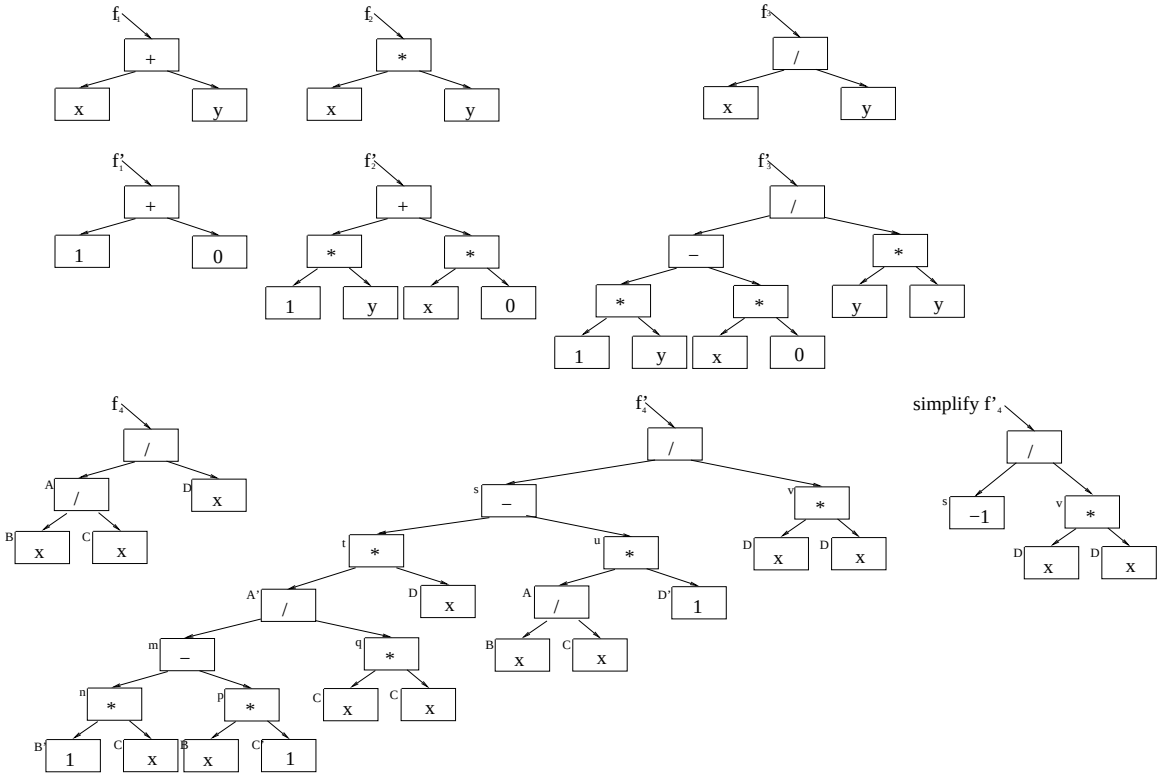


Figure 2: Four functions and their derivatives. The fourth derivative has been simplified.

```
            -- + -|
                  |-- g'  )

else if( f = g-h ) then
   result( g'-h', constructed by
                  |-- h'
            -- - -|
                  |-- g'  )

else if( f = g*h ) then
   result( g'*h + g*h', constructed by
            ie          |-- h'
                  |- * -|
                  |     |-- g
            -- + -|
                  |     |-- h
                  |- * -|
                        |-- g'  )

else if( f = g/h ) then
   result( g'*h - g*h')/(h*h), constructed by
                        |-- h
                  |- * -|
```

```
                    |     |-- h
              -- / -|
                    |           |-- h'
                    |     |- * -|
                    |     |     |-- h
                    |- - -|
                          |     |-- h
                          |- * -|
                                |-- g' )
    end if
```

Example 1.3 Simplify Expression: This routine simplifies a given expression. For example, the derivative of $x * y$ with respect to x will be computed to be: $1 * y + x * 0$. This should be simplified to y.

Specification:

Preconditions: The input consists of an expression f represented by a tree.

Postconditions: The output is another expression that is a simplification of f. Its nodes should be separate from those of f and f should remain unchanged.

Code:

algorithm $Simplify(f)$

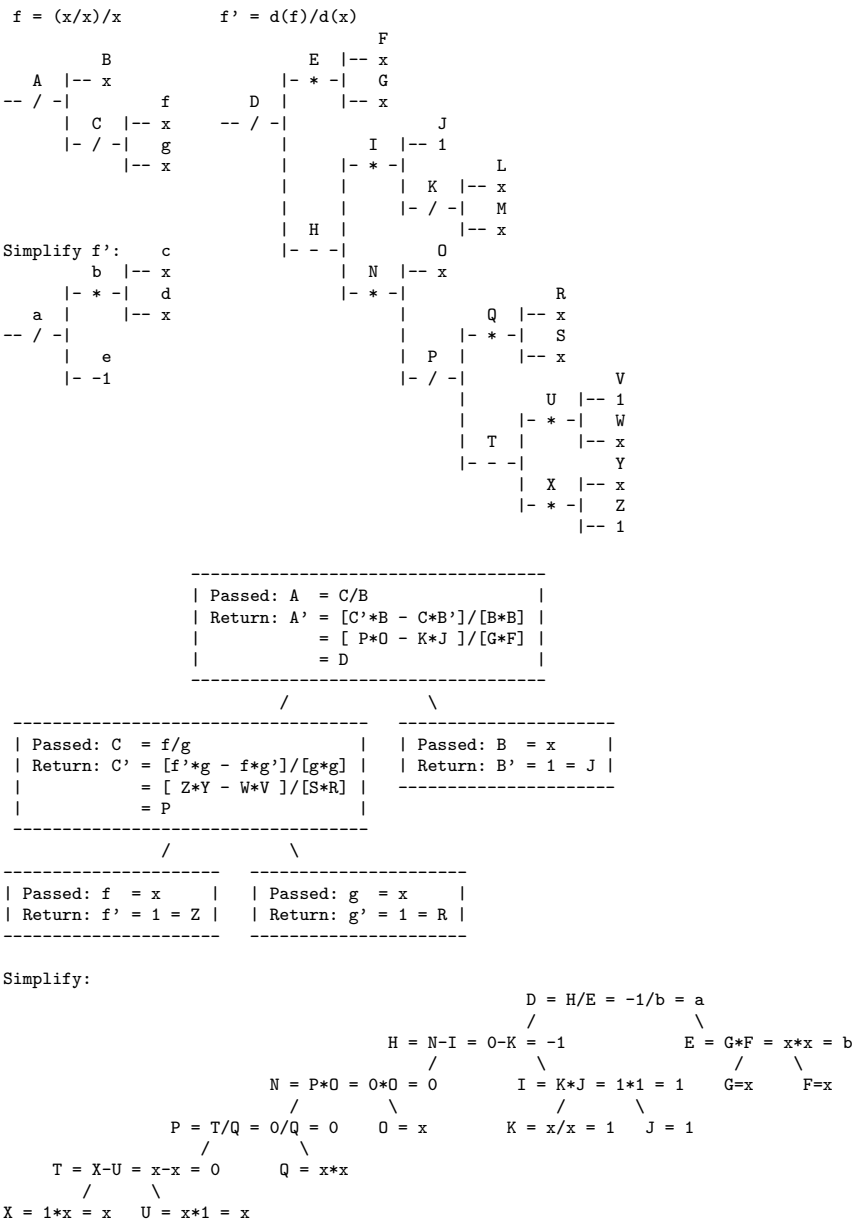
$\langle$***pre−cond***$\rangle$**:** f is an expression.

$\langle$***post−cond***$\rangle$**:** The output is a simplification of this expression.

```
begin
    if( f = a real value or a single variable ) then
        result( Copy(f) )
    else % f is of the form (g' op h')
        g = Simplify(leftSub(f))
        h = Simplify(rightSub(f))
        if( one of the following forms apply
                1*h = h      g*1 = g      0*h = 0      g*0 = 0
                0+h = h      g+0 = g      g-0 = g      x-x = 0
                0/h = 0      g/1 = g      g/0 = ∞      x/x = 1
                6*2 = 12     6/2 = 3      6+2 = 8      6-2 = 4      ) then
            result( the simplified form )
        else
            result( g op h )
        end if
    end if
end algorithm
```

Exercise 1.1 *Trace out the execution of Derivative on the instance $f = (x/x)/x$ given above. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function f passed and derivative returned.*

Exercise 1.2 *(See solution in Section ??) Trace out the execution of Simplify on the derivative f' given above, where $f = (x/x)/x$. In other words, draw a tree with a box for each time a routine is called. For each box, include only the function f passed and simplified expression returned.*



Simplify: