Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2019-2020

Mini Homework 1

Duration: 24 hours
Due date: 17 April 2020, Friday, 09:00

1 Problem Definition

In this mini homework, you will be implementing some functions related to the grid (2D list) and expression structures you worked with during the first and second homeworks. Before we begin, here is some information about the mini homework:

- You can define as many helper functions as you like. You can also use the functions you are expected to implement in other functions. You can (and should) use functions from the `Prelude` and `Data.List`. Feel free to import other standard modules as well although you will probably not be needing them.
- There will be no erroneous input. All grids will be valid (have the same number of elements in each row and be non-empty). Expressions will be valid is well and are guaranteed to not overflow on evaluation.
- Just like the second homework, there are no complexity constraints on the functions. Since the test cases will be small inputs, you should be fine as long as your functions terminate.

Since you will be working with the `Expression` type, the parser from the second homework is once again provided for your convenience. You are able to call it through the `parse` function just like in the homework. The definition of the type is also exactly the same as it was in the second homework and can be found in `Expression.hs`.

2 Functions to Implement

2.1 gridMap - 20 points

Your goal in this first function is simply mapping a function over a grid. In other words, apply a given function on the elements of a given grid and return the new grid as a result. This is very similar to the `map` function, except that it should work on grids rather than lists.

```
gridMap :: (a -> b) -- f: function to apply on each element
        -> [[a]] -- g: input grid
        -> [[b]] -- result of applying f to every element of g
--------------------------------------------------------------------------------
ghci> gridMap (+1) [[1, 2], [3, 4]]
[[2,3],[4,5]]
ghci> gridMap (*5) [[1, 2], [3, 4], [5, 6]]
[[5,10],[15,20],[25,30]]
ghci> gridMap ("hello "++) [["world", "haskell", "ceng242"]]
[["hello world","hello haskell","hello ceng242"]]
ghci> gridMap even [[10, 11, 12], [13, 14, 15], [16, 17, 18]]
[[True,False,True],[False,True,False],[True,False,True]]
```

2.2 gridMapIf - 20 points

This second function is very similar to the first one. This time, a predicate (single argument function returning a boolean) is provided along with the function to be applied. Instead of applying the given function to each element, you are supposed to apply it only to the elements satisfying the predicate. Do not change the values of the elements that do not satisfy the predicate.

```
gridMapIf :: (a -> Bool) -- p: predicate that elements must satisfy
          -> (a -> a) -- f: function to apply on elements that satisfy p
          -> [[a]] -- g: input grid
          -> [[a]] -- result of applying f to every element of g satisfying p
--------------------------------------------------------------------------------
ghci> gridMapIf (<10) (*2) [[1, 2, 3], [11, 12, 13]]
[[2,4,6],[11,12,13]]
ghci> gridMapIf (<0) negate [[1], [-2], [3], [-4]]
[[1],[2],[3],[4]]
ghci> gridMapIf even (*0) [[1, 4, 9], [16, 25, 36], [49, 64, 81]]
[[1,0,9],[0,25,0],[49,0,81]]
ghci> gridMapIf (\x -> 4 <= x && x <= 7) (*10) [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
[[1,2,3,40,50],[60,70,8,9,10]]
ghci> gridMapIf (==12) (\_ -> 1000) [[1, 2, 3], [11, 12, 13]]
[[1,2,3],[11,1000,13]]
```

2.3 evalExpr - 20 points

Now we begin working with `Expressions`. To start off, in this function, you simply have to evaluate a given expression by applying the operations. The values of variables are provided in the first argument of the function in an associative list format. All variables that exist in the given expression are guaranteed to be contained in the list.

```
evalExpr :: [(String, Int)] -- l: assoc. list containing variable values
         -> ExprV -- e: expression to be evaluated
         -> Int -- result of evaluating e with the values provided in l
---------------------------------------------------------------------------
ghci> evalExpr [] (Leaf (Constant 10))
10
ghci> evalExpr [("a", 3)] (Leaf (Variable "a"))
3
ghci> evalExpr [] $ parse "-(3+5*7)"
-38
ghci> evalExpr [("b", 10)] $ parse "-(3+5*7)+b*b"
62
ghci> evalExpr [("b", 10), ("var", 1)] $ parse "-(3+5*7)+b*b+(5*var)"
67
ghci> evalExpr [("x", -2), ("y", 0), ("z", 1)] $ parse "x*y*z + x*x + 5*z"
9
```

Hint: You can use `lookup` (from the `Prelude`) for looking up the values of the variables, rather than writing your own function.

2.4 getVars - 20 points

The next function you are going to implement is `getVars`. All it does is return all the variables in a given expression as a list of strings. There is a catch, however. The variables in the list should be in lexicographical order and there should be no duplicates, i.e. a variable must only appear once in the list no matter how many times it appears in the expression.

```
getVars :: ExprV -- e: input expression
        -> [String] -- list of the variables in e (ordered, no duplicates)
---------------------------------------------------------------------------
ghci> getVars $ Leaf (Constant 3)
[]
ghci> getVars $ Leaf (Variable "x")
["x"]
ghci> getVars $ parse "3+5+8*-10"
[]
ghci> getVars $ parse "x+7*x+-x*5"
["x"]
ghci> getVars $ parse "myVar*a+(-y*y*z)"
["a","myVar","y","z"]
ghci> getVars $ parse "3*17+-a*x+ceng242"
["a","ceng242","x"]
```

Hint: Lexicographical (dictionary) ordering is the default ordering of strings in most languages, including Haskell. You can compare strings directly using the comparison operators <, > etc. You can also use `sort` from `Data.List` to order your strings for you. Remember that you can use other functions from `Data.List` as well.

2.5 evalDeriv - 20 points

In this last function we will be going one step further than the third function. Now, instead of evaluating the expression itself (which can be viewed as a function of multiple variables) with the given values (which can be viewed as a point in a multidimensional space), you will evaluate its first derivative with respect to some variable provided as an argument. Once again, a value will be provided for each variable in the given expression, there will be no erroneous input.

```
evalDeriv :: [(String, Int)] -- l: assoc. list containing variable values
          -> String -- v: variable to take derivative with respect to
          -> ExprV -- e: expression to be evaluated
          -> Int -- result of evaluating de/dv with the values provided in l
```

```
ghci> evalDeriv [] "x" (Leaf (Constant 3))
0
ghci> evalDeriv [("x", 3)] "x" (Leaf (Variable "x"))
1
ghci> evalDeriv [] "x" (Leaf (Variable "y"))
0
ghci> evalDeriv [("x", 5)] "x" $ parse "x*x*x"
75
ghci> evalDeriv [("x", 5), ("y", 3)] "x" $ parse "x*x*x + x*y"
78
ghci> evalDeriv [("x", 12)] "x" $ parse "-x"
-1
ghci> evalDeriv [("x", 12)] "x" $ parse "-x+x"
0
ghci> evalDeriv [("y", 0), ("z", 2), ("myVar", 5)] "x" $ parse "3+y+z+5*myVar"
0
ghci> evalDeriv [("x", 12), ("y", 5), ("z", 13)] "z" $ parse "x*x + y*y + -z*z"
-26
```

Hint: Here are the derivative rules you will need to use in case you have forgotten them (note that the definitions remain the same, x is always a variable, f is always function of x etc.):

- **Self:** $\frac{\delta}{\delta x}[x] = 1$, where x is a variable
- **Constant:** $\frac{\delta}{\delta x}[a] = 0$, where a is a real constant
- **Independent variable:** $\frac{\delta}{\delta x}[y] = 0$, where y is another variable independent of x (y is just a constant with respect to x in this case)
- **Negation:** $\frac{\delta}{\delta x}[-f(x)] = -\frac{\delta}{\delta x}[f(x)]$, where f is a function of x (and possibly other variables)
- **Sum:** $\frac{\delta}{\delta x}[f(x) + g(x)] = \frac{\delta}{\delta x}[f(x)] + \frac{\delta}{\delta x}[g(x)]$, where g is also a function, like x
- **Product:** $\frac{\delta}{\delta x}[f(x) \cdot g(x)] = f(x) \cdot \frac{\delta}{\delta x}[g(x)] + \frac{\delta}{\delta x}[f(x)] \cdot g(x)$

Remember that you do not have to construct the derivative expression, only evaluate the result.

3 Submission and Evaluation

- Make sure to submit your code on cengclass once you are done.
- You can evaluate your code interactively on cengclass. However, note that the evaluation test cases are only provided to assist you and are different from the test cases that will be used during grading. Thus, the grade resulting from the evaluation is **not** your final grade.
- Just as in the homeworks, do not modify the module declaration or erase any functions. If your submission has compilation errors (that can be fixed trivially), 5 points will be deducted from your final grade.
- **We have a zero tolerance policy for cheating.** This is an individual homework. People involved in cheating will receive zero from the homework and be punished according to university regulations.
- There is **no** late submission.