CSE340 Spring 2020 HOMEWORK 3
Due by 11:59 PM on Wednesday March 18 2020

PLEASE READ THE FOLLOWING CAREFULLY

1. Your answers can be only be typed.
2. On Gradescope, you should submit the answers to separate question separately.
3. For each question, read carefully the required format for the answer. The required format will make it easier for you to answer and for the graders to grade. Answers that are not according to the required format will not be graded

Problem 1 (Reducible Expression). The goal of this problem is to identify the redexes, if any, in the given expressions. To identify a redex, you should highlight the (λx. t) part of the redex in yellow and the t' part of the redex in blue. Here are two examples.

Example 1. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Answer. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Example 2. ((λx. x x) (λx. x x)) ((λx. x x) (λx. x x))

Answer. ((λx. x x) (λx. x x)) ((λx. x x) (λx. x x))
((λx. x x) (λx. x x)) ((λx. x x) (λx. x x))

Note that you are only asked to identify the redexes and you are not asked to do the reduction.

For each of the following, identify the redexes, if any:

1. x (λx. x x) (λx. x x) x

   No Redex

2. (λx. (λx. x) x) (λx. x) x

   (λx. (λx. x) x) (λx. x) x

   (λx. (λx. x) x) (λx. x) x

3. (λx. (λx. x) x) λx. (λx. x) x

   (λx. (λx. x) x) λx. (λx. x) x

   (λx. (λx. x) x) (λx. (λx. x) x)

   (λx. (λx. x) x) (λx. (λx. x) x)

4. (λx. x (λx. x) x) λx. x (λx. x) x

   (λx. x (λx. x) x) (λx. x (λx. x) x)

5. x (λx. x (λx. x) x) (λx. x (λx. x) x) x

   ((x (λx. (x (λx. x)) x)) (λx. (x (λx. x)) x)) x

   No redex

6. λx. λx. x λx. λx. x x x

   No redex

7. ((λx. λy. (λy. x) λx. x x)) ((λx. (λy. x)) λx. x x)

   ((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

   ((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

   ((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

8. (λx. (λx. (λx. x) x) x) ((λx. x) (λx. x) (λx. x))

   (λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x))

   (λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x))

   (λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x))

   (λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x))

9. x (λx. (λy. x) λx. x x) (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

   ((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

   ((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

   ((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x))

10. (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

    (λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x))

    (λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x))

    (λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x))

Problem 2 (Normal Order). The goal of this problem is to identify the unique redex, if any, that will be reduced first in normal order evaluation strategy for the given expressions. To identify a redex, you should highlight the (λx. t) part of the redex in yellow and the t' part of the redex in blue. Here is an example.

Example. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Answer. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Note that you are only asked to identify the redexes and you are not asked to do the reduction.

1. x (λx. x x) (λx. x x) x

   No redex

2. (λx. (λx. x) x) (λx. x) x

   ((λx. (λx. x) x) (λx. x)) x - I

   ((λx. (λx. x) x) (λx. x)) x - II

   I is nested in II. II is the only outermost redex. Hence II is reduced first.

3. (λx. (λx. x) x) λx. (λx. x) x

   (λx. (λx. x) x) (λx. (λx. x) x) - I

   (λx. (λx. x) x) (λx. (λx. x) x) - II

   (λx. (λx. x) x) (λx. (λx. x) x) - III

   Both I and II are nested in III. III is the only outermost redex. Hence III is reduced first.

4. (λx. x (λx. x) x) λx. x (λx. x) x

   (λx. (x (λx. x)) x) (λx. (x (λx. x)) x) - I

   I is the only redex. Hence I is reduced first!

5. x (λx. x (λx. x) x) (λx. x (λx. x) x) x

   No redex

6. λx. (λx. x) (λx. (λx. x) x x)

   (λx. (λx. x) (λx. ((λx. x) x) x)) - I

   (λx. (λx. x) (λx. ((λx. x) x) x)) - II

   I is nested in II. II is the only outermost redex. Hence II is reduced first

7. ((λx. λy. (λy. x) λx. x x)) ((λx. (λy. x)) λx. x x)

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - I

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - II

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - III

I and II are nested in III. III is the only outermost redex, so III is reduced first

8. (λx. (λx. (λx. x) x) x) ((λx. x) (λx. x) (λx. x))

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - I

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - II

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - III

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - IV

I, II, and III are nested in IV. IV is therefore the only outermost redex. Hence IV is reduced first.

9. x (λx. (λy. x) λx. x x) (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - I

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - II

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - III

Here I, II and III are all outermost redexes because none of them is nested in another redex.

Since I is the left outermost redex, I is reduced first.

10. (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - I

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - II

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - III

I and II are nested in III. Since III is the only outermost redex, III is reduced first

Problem 3 (Call by Value). The goal of this problem is to identify the redexes, if any, that can be reduced first in call by value evaluation stratgey for the given expressions. To identify a redex, you should highlight the (λx. t) part of the redex in yellow and the t' part of the redex in blue. Here is an example:

Example. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Answer. (λx. x x) (λx. x x) (λx. x x) (λx. x x)

Note that you are only asked to identify the redexes and you are not asked to do the reduction.

You are only asked to identify the redexes and you are not asked to do the reductions.

1. (λx. x x) λx. x x

   (λx. x x) (λx. x x)

   There is only one redex that is not under abstraction. Also, the argument (λx. x x) is a value, so the redex can be reduced in call by value

2. (λx. (λx. x) x) x

   (λx. (λx. x) x) x - I

   (λx. (λx. x) x) x - II

   There are two redexes. I cannot be reduced since it is under an abstraction. Only II can be reduced by call by value, since it is not under an abstraction and its argument x is a value.

3. ((λx. (λx. x) x) λx. (λx. x) x) ((λx. x) x)

   ((λx. (λx. x) x) (λx. (λx. x) x)) ((λx. x) x) - I

   ((λx. (λx. x) x) (λx. (λx. x) x)) ((λx. x) x) - II

   ((λx. (λx. x) x) (λx. (λx. x) x)) ((λx. x) x) - III

   ((λx. (λx. x) x) (λx. (λx. x) x)) ((λx. x) x) - IV

   There are 4 redexes. I and II cannot be reduced since they are under abstraction. III and IV can be reduced in call by value because they are not under abstraction and their arguments (x and (λx. (λx. x) x)) respectively) are values. Notice that (λx. (λx. x) x) is a value because all the redexes it has are under abstraction.

4. ((λx. (λx. x) x) x (λx. x) x) ((λx. x) x)

   ((((λx. (λx. x) x) x) (λx. x)) x) ((λx. x) x) - I

((((λx. (λx. x) x) x) (λx. x)) x) ((λx. x) x) - II

((((λx. (λx. x) x) x) (λx. x)) x) ((λx. x) x) - III

I cannot be reduced because it is under an abstraction. II and III can be reduced because they are not under abstraction and their arguments are values.

5. x (λx. x (λx. x) x) (λx. x (λx. x) x) x

((x (λx. (x (λx. x)) x)) (λx. (x (λx. x)) x)) x

There are no redexes.

6. λx. λx. x λx. ((λx. x) x) x

(λx. (λx. x (λx. ((λx. x) x) x)))

This expression has only one redex, but it cannot be reduced in call by value because it is under an abstraction

7. ((λx. λy. (λy. x) λx. x x)) ((λx. (λy. x)) λx. x x)

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - I

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - II

((λx. λy. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - III

I cannot be reduced because it is under an abstraction. III cannot be reduced because its argument ((λx. (λy. x)) (λx. x x)) is not a value. II can be reduced because it is not under an abstraction and its argument(λx. x x) is a value.

8. (λx. (λx. (λx. x) x) x) ((λx. x) (λx. x) (λx. x))

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - I

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - II

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - III

(λx. (λx. (λx. x) x) x) (((λx. x) (λx. x)) (λx. x)) - IV

I and II cannot be reduced because they are under an abstraction. III cannot be reduced since its argument is not a value. IV can be reduced because it is not under an abstraction and its argument (λx. x) is a value.

9. x (λx. (λy. x) λx. x x) (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - I

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - II

((x (λx. (λy. x) (λx. x x))) (λx. (λy. x) (λx. x x))) ((λx. (λy. x)) (λx. x x)) - III

I and II cannot be reduced because they are under abstraction. III can be reduced because it is not under an abstraction and its argument (λx. x x) is a value.

10. (λx. (λy. x) λx. x x) ((λx. (λy. x)) λx. x x)

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - I

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - II

(λx. (λy. x) (λx. x x)) ((λx. (λy. x)) (λx. x x)) - III

I cannot be reduced because it is under an abstraction. III cannot be reduced since its argument (λx. (λy. x)) (λx. x x) is not a value. II can be reduced since its argument (λx. x x) is a value and it is not under an abstraction.

Question 4 (Beta reduction and alpha renaming). For each of the following expressions, give the resulting redex after the highlighted redex is reduced. **If** there is need for renaming, you should break you answer into two parts. In the first part you do the renaming and in the second part you do the beta reduction. In all cases, you should highlight by using **bolface** the whole part of the result that corresponds to the redex after it is reduced. Also, if alpha renaming is needed, you should highlight the whole redex with the par that is renamed.

Example 1. ((λx. x x) y) ((λx. x x) x)
Answer. **(λx. y y)** ((λx. x x) x) after beta reduction

Example 2. ((λx. λy. x x) y) ((λx. x x) x)
Answer. **((λx. λw. x x) y)** ((λx. x x) x) after alpha renaming
(λw. y y) ((λx. x x) x) after beta reduction

Note that you are asked to do only one beta reduction of the specified redex and no multiple beta reductions.

For each of the following expressions, give the resulting expression after the highlighted redex is reduced. You should only do renaming if renaming is needed. You should follow the format and requirements outlined above in your answers.

In the solution below, the bound x's of the redex are in red.

1. (λx. x x) (λx. x)
   (λx. x) (λx. x)

2. (λx. x x) (λx. x y)
   (λx. x y) (λx. x y)

3. (λx. x (λx. x) (λy. x) x) (λx. x x)
   (λx. x x) (λx. x) (λy. (λx. x x)) (λx. x x)

4. (λx. x (λx. x) (λy. x) x) (λx. x y)
   (λx. x (λx. x) (λw. x) x) (λx. x y) after renaming
   (λx. x y) (λx. x) (λw. (λx. x y)) (λx. x y) after beta reduction

5. (λz. x (λx. x) (λy. x) x) (λx. y z)

x (λx. x) (λy. x) x

there are no bound z's, so this is a constant function and the answer is the body of the abstraction and there is no substitution done.

6. (λx. x (λx. x) (λx. λy. x) x) (λx. y z)

(λx. y z) (λx. x) (λx. λy. x) (λx. y z)

7. λy. (λx. x (λx. x) (λx. λy. x) x) (λx. y z)

λy. ((λx. y z) (λx. x) (λx. λy. x) (λx. y z))

8. (λx. (λy. λz. x) x) (λx. (λy. y) z)

(λx. (λy. λw. x) x) (λx. (λy. y) z) After renaming

(λy. λw. (λx. (λy. y) z)) (λx. (λy. y) z)

Note that in this example we do not need to rename the λy. because the y in the argument does not change binding when the reduction applied

9. (λx. x (λx. x) x (λx. x) x) y z

(y (λx. x) y (λx. x) y) z

Note the added parentheses, but they really do not need to be added because due to the reduction. They are there due to how left associative grouping is done.

10. (λx. (λy. (λz. λx. x) x)) (λx. x y z) (missing parenthesis added)

(λx. (λw. (λz. λx. x) x)) (λx. x y z) After renaming -

(λw. (λz. λx. x) (λx. x y z))

Question 3 (Binary Search Trees (BST) with Lambda Calculus). We define a lambda calculus representation of binary search trees. In general, a tree can be either empty or it is not empty. If the tree is not empty, then the node contains an element and two subtrees: the left subtree and the right subtree. As I discussed in class with lists, we need to have a way to determine if a tree is empty, so that we do not attempt to access the fields of an empty tree. This is similar to C++, in which you do not access the fields of a node before checking that the pointer to the node is not the null pointer.

At a high level, a BST is either empty or it is not empty, in which case it is of the form

a LT RT

where a is the element and LT and RT are the left and the right subtrees respectively. In general, either LT alone, or RT alone or LT and RT both can be empty or both LT and RT can be not empty.

To represent this data structure in Lambda calculus, we will use pairs (the old homework on lists should be very educational and I suggest you read it with its solution).

empty tree = pair fls fls
tree with one element and two empty subtrees = pair tru (pair a (pair (pair fls fls) (pair fls fls))

Let is consider the pieces

tru	indicates that the tree is not empty
a	is the element. If a tree T is not empty, the fst (snd T) is the root element of the tree
(pair fls fls)	is the left subtree. In this example, it is empty. In general, if a tree is not empty, then fst (snd (snd T) is the left subtree.
(pair fls fls)	is the right subtree. In this example, it is empty. In general, if a tree is not empty, then snd (snd (snd T) is the right subtree.

Of course, having such cumbersome expression can result if hard to read code! So, the best thing is to introduce functions that hide the complexity. Here are two that I propose and you can add more in your solution

LT = λT. fst (snd (snd T)
RT = λT. snd (snd (snd T)

You should be careful that these two functions expect that the tree is not empty.

For all questions, you are asked to write functions. You should understand that to mean write a lambda expression. Finally, in all the question, there should be no attempt to balance the BST.

1. (**Empty Tree**) Write a function `isempty` that returns a boolean value to indicate if a BST is empty or not. The function should return tru if the tree is empty and fls if the tree is not empty.

   `isempty` = λT. NOT (fst T)

2. (**Root Element**) Write a function `root` that returns the root of a BST. Since a tree can be empty, we need a way to indicate that. Your function should return a pair. If the first element of the pair is tru, then the second element of the pair is the root of the tree. If the first element of the pair is fls, this means that the tree is empty and has no root.

   `root` = λT. (`isempty` T) (pair fls fls) (pair tru (fst (snd T)))

   If the tree is empty, the returned value is (pair fls fls) which indicates that the returned value is not valid and that there is no root.
   If the tree is not empty, the returned value is (pair tru (fst (snd T))) which indicates that the returned value is valid and the root is (fst(snd T))

3. (**Insert element**) Write a recursive function that inserts an element into a tree. This function always succeeds if its argument is a BST.

   We write a recursive function

```
g = λins. λa. λT.
        (isempty T)
                (pair  tru  ( pair     a          (pair  (pair fls fls)      (pair fls fls) )                  // T is empty
                ( (LTEQ a  (fst(snd T)) )
                           (pair  tru ( pair (fst(snd T))   ( pair  (ins a (LT T))      (RT T)     ) )   ) // T not empty
                                                                                                          // and a <= root
                           (pair  tru ( pair (fst(snd T))   ( pair  (LT T)          (ins a (RT T) )   )   ) // T not empty
                                                                                                          // and a > root
                )
```

   `insert` = fix g

   This requires some explanation. g is highlighted so that the root of the resulting tree is highlighted in blue, the left subtree is highlighted in yellow and the right subtree is highlighted in green.

   If the tree is empty, g returns a tree of one element with a as the root.
   If the tree is not empty, then there are two cases to consider depending on the result of comparison between a and the root element of the tree.

   1. If a <= root element of T. The returned value is a tree that has the same root as T, and a left subtree identical to the tree obtained by inserting a into the left subtree of T and in which a right subtree that is identical to that of T
   2. if a > root element of T. The returned value is a tree that has the same root as T, the a left subtree that is identical to that of T and in which a right subtree identical to the tree obtained by inserting a in the right subtree of T

4. (**Sum of elements**) Write a recursive functions that takes as argument a tree whose elements are integers and returns the sum of the elements of the tree. If the tree is empty, the function should return 0

   g = λsm. λT. (`isempty` T)
   0
   (plus (fst (snd T)) (plus (sm (LT T)) (sm (RT T))))

   Sum = fix g

   In the function g above, sm is the argument used in the recursive definition of Sum

5. (**Merging two tree**) Write a recursive function that takes two trees T1 and T2 and returns a tree T obtained by ainserting all elements of T2 into T1.

   g = λmrg. λT1. λT2. (`isempty` T2)
   T1
   (mrg (mrg (`insert` (fst(snd T2) T1) (LT T2))) (RT T2))

   `merge` = fix g

   Let is explain the function g.

   - If T2 is empty, the result is T1.
   - If T2 is not empty, the result is (mrg (mrg (`insert` (fst(snd T2) T1) (LT T2))) (RT T2)) which we break into pieces
     a. (fst(snd T2) is the root of T2
     b. (`insert` (fst(snd T2) T1) is the tree that result from inserting the root of T2 into T1 using the insert function from part 3
     c. mrg (mrg (`insert` (fst(snd T2) T1) (LT T2))) is a tree that is obtained by merging the left subtree of T2 with the tree obtained by inserting the root of T2 into T1 and then
     d. (mrg (mrg (`insert` (fst(snd T2) T1) (LT T2))) (RT T2)) is a tree that is obtained by merging the right subtree of T2 with the tree from part c. This is the tree that we want.

6. **(Tree Height**) Write a recursive function to calculate the height of a tree. If a tree is empty, its height is 0. If a tree has only one element, its height is also 0.

   It would be useful to define a function `max` to make the code more readable

   `max` = λm. λn. (LTEQ m n) n m

   if m <= n , `max` returns n, otherwise it returns m

```
g = λht. λT. (isempty T)
                0
                ( ( AND (isempty (LT T))  (isempty (RT T)) )
                        0
                 ( succ ( max (ht (LT T))  (ht (RT T)) ) )
                )

height = fix g
```

- If the tree is empty, its height is 0
- If the tree has only one element (both subtrees are empty), its height is 0
- Otherwise, the height of the two subtrees is calculated and the max of the two height is incremented by 1 to obtain the height of the tree

7. **(Deepest Element)** This is probably the hardest question. Write a recursive function that takes a tree as argument and return an element that is the farthest away from the root (its distance to the root is equal to the tree height). Since the tree can be empty, the function should return a pair consisting of a Boolean value and a tree element (see discussion with question 2 above). In general, there might be more than one element that are deepest elements. Your function should only return one such element.

   In order to calculate a deepest element, we first need to calculate the tree height. Then we look for an element whose distance to the root is equal to the tree height.

```
g =  λdp. λT. (isempty T)
                (pair fls fls)
                ( ( iszero (height T) )
                        ( root T )                                  // tree is not empty but has
                                                                    // height 0

                        ( (EQUAL (height (LT T)) (prd (height T)) // tree is not empty and has
                                                                    // positive height => look
                                                                    // in the subtree whose height
                                                                    // is ( height T ) - 1
                                (dp (LT T))
                                (dp (RT T))
                         )

                )
```