CSE340 Fall 2019 HOMEWORK 3

Due Wednesday 24 October 2019 by 12:01 AM

PLEASE READ THE FOLLOWING CAREFULLY

1. Your answers can be handwritten or typed. If you write your answers, you need to have clear handwriting. If we cannot read it we cannot grade it!
2. On gradescope, you should submit the answers to separate question separately.

Problem 1 (Lambda calculus normal and applicative order). For each of the following identify the redex (if any) that will be reduced first under normal order reduction strategy. You are only asked to identify the redex not to do the reduction.

a. ((x (λx. x x)) (λx. x x)) x NO redex

b. ((x (λx. **(λx. x) x**)) (λx. x)) x only one redex that can be reduced

(**λx. t) t'**

c . (λx. (λx. x) x) λx. x two redexes. I is reduced first

I II

(λx. t) t'

d. ((λx. λy. (λy. x) λx. x x)) ((λx. (λy. x)) λx. x x) three redexes. I is reduced first

I II III

(λx. t) t'

e. (λx. (λx. (λx. x) x) x) (x (λx. x) x) there are three redexes. I is reduced first

(λx. t) t'

I

(λx. t) t'

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

I. **(λx. t) t'** II. **(λx. t) t'** III. (**λx. t) t'**

there are three redexes. I is reduced first

Question 2 (Lambda calculus normal and applicative order). For each of the following expressions identify a redex (if any) that can be reduced first under call by value reduction strategy. Remember that call by value does not specify which redex should be reduced first. It specifies which redex cannot be reduced first (a redex whose right hand side is not a value) or cannot be reduced (under abstraction). If there is more than one redex that can be reduced first, you should specify all of them.

a. x (λx. x x) (λx. x x) x NO redex

b.

$$x\ (\ \lambda x.\ \underbrace{(\lambda x.}_{(\boldsymbol{\lambda x.}}\ \underbrace{x)}_{\mathbf{t}\ )}\ \underbrace{x}_{\mathbf{t'}}\ )\ (\ \lambda x.\ \underbrace{(\lambda x.}_{(\boldsymbol{\lambda x.}}\ \underbrace{x)}_{\mathbf{t}\ )}\ \underbrace{x}_{\mathbf{t'}}\ )$$

two redexes, but they are under abstractions, so they cannot be reduced

c.

$$(\ \underbrace{\lambda x.}_{\textbf{III. }(\boldsymbol{\lambda x.}}\ \underbrace{\underbrace{\underbrace{(\lambda x.}\ \underbrace{x)}\ \underbrace{x}}_{\text{I. }(\boldsymbol{\lambda x.}\ \mathbf{t}\ )\ \mathbf{t'}}}_{\mathbf{t}\quad)}\ )\ \underbrace{(\ \lambda x.\ \underbrace{\underbrace{(\lambda x.}\ \underbrace{x)}\ \underbrace{x}}_{\text{II. }(\boldsymbol{\lambda x.}\ \mathbf{t}\ )\ \mathbf{t'}}\ )}_{\mathbf{t'}}$$

there are three redexes. I and II are under abstractions and cannot be reduced. The argument of III is a value (cannot be reduced in call by value), so III is reduced first.

d.

$$(\ \underbrace{\lambda x.}_{\textbf{III. }(\boldsymbol{\lambda x.}}\ \underbrace{\underbrace{(\ \underbrace{\lambda y.}\ \underbrace{x\ )}\ \underbrace{\lambda x.\ x\quad x}}_{\text{I. }(\boldsymbol{\lambda x.}\ \mathbf{t}\ )\quad \mathbf{t'}}}_{\mathbf{t}\quad)}\ )\ \underbrace{\underbrace{(\ (\ \underbrace{\lambda x.}\ \underbrace{(\ \lambda y.\ x\ )}\ )\ \underbrace{\lambda x.\ x\ x}\ )}_{\text{II.}(\boldsymbol{\lambda x.}\quad \mathbf{t}\quad)\quad \mathbf{t'}}}_{\mathbf{t'}}$$

There are three redexes. I is under an abstraction and cannot be reduced. The argument of III is not a value, so III cannot be reduced. II can be reduced first: it is not under an abstraction and its argument (λx. x x) is a value.

e.

$$(\ \underbrace{\lambda x.}\ \underbrace{(\ \underbrace{\lambda x.}\ \underbrace{(\ \underbrace{\lambda x.}\ \underbrace{x\ )}\ \underbrace{x}\ }_{\mathbf{I}}\ )\ \underbrace{x}_{\mathbf{III}}\ )\ }\ \underbrace{(\ x\ (\ \lambda x.\ \ x\ )\ \ x\ )}\ (\ \underbrace{(\ \lambda x.}_{\mathbf{IV}}\ \underbrace{(\ \lambda x.\ (\ \underbrace{\lambda x.}\ \underbrace{x\ )}\ \underbrace{x\ )}\ )}_{\mathbf{II}}\ \underbrace{x}\ )_{\mathbf{V}}$$

There are five redexes. I , II and III are under abstractions and cannot be reduced. The argument of IV is a value (x) , so IV can be reduced first. The argument of V is (x (λx. x) x) which is also a value because it does not have any redex, so V can also be reduced first.

Either IV or V can be reduced first.

Question 3 (Linked lists with lambda calculus). We define a lambda calculus representation of linked lists. A list is represented with pairs. The first part of the pair is an element and the second part of the pair is a list. In order to indicate if the second element represents a non-empty list, we use a boolean flag. We also, use a header in order to be able to represent the empty list. I start by giving a few examples.

empty list = pair fls fls

list with one element a = $\text{pair tru (pair a } \underbrace{\text{(pair fls fls)}}_{\text{representation of empty list}} \text{)}$

list with two elements a and b = $\text{pair tru (pair a } \underbrace{\text{(pair tru (pair b (pair fls fls)))}}_{\text{representation of list with one element b}} \text{)}$

In general, the representation of a list with elements $a_1, a_2, a_3, \dots, a_k$ in the given order, where **k ≥ 1** is

pair tru (pair a_1 L)

where L is the representation of the list whose element in order are $a_2, a_3, \dots, a_k$

Note that only the last pair has “fls” as the first element to indicate the empty list. For all the questions, you are asked to write functions. You should understand that to mean write a lambda expression.

1. (**Add element**) Write a function that takes an element a and a list L representing $a_1, a_2, a_3, \dots, a_k$ and returns a list representing a, $a_1, a_2, a_3, \dots, a_k$

The function AddElement does the following: it returns a list whose first element is the element being added and whose ”remainder list” is the list to which the element is being added.

Here is the lambda calculus function for AddElement

AddElement = λa. λL. pair tru (pair a L)

2. (**Empty List**) Write a function that returns a boolean value to indicate if a list if empty or not. The function should return tru if the list is empty and fls if the list is not empty.

The EmptyList function simply returns the negation of the first element of the list; if the value if tru, the function returns fls, otherwise it returns fls. The lambda function is the following

EmptyList = λL. (fst L) fls tru

3. (**First Element**) Write a function that returns the first element of a list. Since a list 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 if the first element of the list. If the first element of the pair is false, this means that the list is empty and has no first element.

If the list is empty, the function simply returns (pair fls fls). If the list if not empty, the function returns a pair consisting of the boolean value tru and the first element of the list. If the list is L, the first element of the list is **(fst (snd L))** and the returned value would be **pair tru (fst (snd L))**

```
FirstElement = λL. (EmptyList L)
                                        (pair fls fls)
                                        (pair tru (fst (snd L)))
```

4. (**Last element**) Write a recursive function that returns the last element of a list. Again as in question 3 , the function should return a pair. The first part of the pair is a boolean value. If the first element of the pair is tru, then the second element if the first element of the list. If the first element of the pair is false, this means that the list is empty and has no first element.

The function works as follows

- If the list is empty: (pair fls fls) is returned
- If the list has only one element: the last element is the same as the first element so (FirstElement L) is returned.
- If the list has more than one element: return the last element of the "remainder list" which is snd (snd L)

```
g =      λLastE. λL. (EmptyList L)
                     (pair fls fls)
                     ( (EmptyList (snd (snd L)))        // if list is not empty and has one element,
                                (FirstElement  L)       // the FirstElement is returned
                                (LastE (snd (snd L)))   // otherwise the lastE of the remainder
                                                        // is returned
                      )

LastElement = fix g
```

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

The function works as follows

- If the list is empty: return 0
- If the list is not empty: return the sum of the first element and SumOfElements (snd (snd L))

g = λSumE. λL. (EmptyList L)
    0
    (plus (fst (snd L)) (SumE (snd (snd L))))

SumElements = fix g

6. (**Reversing a List**) Write a recursive function that reverses a list. If the list represents $a_1, a_2, a_3, \ldots, a_k$, the reverse of the list should represent $a_k, a_{k-1}, a_{k-2}, \ldots, a_1$

If the list is of the form a L' , the reverse of the list is L'^R a, where L'^R denotes the reverse of L'

In what follows, I assume I have a function AddLast which adds an element to the end of a list (I will show the code for that below). The function reverse works as follows

- If the list is empty, the answer is the empty list
- If the list is not empty and is of the form pair tru (pair a L'), the answer is
  pair tru (AddLast $\underbrace{\text{(fst snd L)}}_{a}$ $\underbrace{\text{(reverse (snd (snd L)))}}_{L'^R}$)

g = λRevL. λL. (EmptyList L)
    (pair fls fls)
    (pair tru (AddLast (fst snd L) (RevL (snd (snd L)))

ReverseList = fix g

The function AddLast is also recursive and is defined as follows

g = λAddL. λa. λL. (EmptyList L)
    (pair tru (pair a (pair fls fls)))
    (pair tru (pair (fst (snd L)) (AddL a (snd (snd L)))

AddLast = fix g

7. (**Merging two lists**) Write a recursive function that takes two lists L1 and L2 and returns a list L obtained by appending L2 to L1. If L1 represents represents a_1, a_2, a_3, ... , a_m and L2 represents b_1, b_2, b_3, ... , b_n , then the result should represent a_1, a_2, a_3, ... , a_m ,b_1, b_2, b_3, ... , b_n .

The merge of L1 and L2 is a new list which is equal to L2 if L1 is empty. If L1 is of the form a L1’, then the (merge L1 L2) should be of the form a (merge L1’ L2). This gives us the following recursive function

```
g = λmrg. λL1. λL2. (EmptyList L1)
                        L2
                        pair tru ( pair (fst (snd L1)) (mrg L2 (snd (snd L1))) )
```

Merge = fix g

Question 4.1 (Static Scoping). Give the output of the following program under static scoping

```
int a = 5;
int b = 6;

void f()
{

   g();

   { int a = 4;
     a = a + b;
     b = a + b;
     g();
   }

   { int b = 3;
     a = a + b;
     b = a + b;
     g();
   }

   a = a+b;
   g();

}

void g()
{
   print a;
   print b;
}

int main ()
{
    int a = 1;
    int b = 10;
    f();
    g();
}
```

Show your work!

$a_{global} = 5 \to 8^{16} \to 24^{21}$

$b_{global} = 6 \to 16^{10}$

main ()

a_{main} 1^{1}

b_{main} 10^{2}

1 $a_{main} = 1;$

2 $b_{main} = 10;$

3 **f()**

4 **g()**

5 print a_{global} // prints 5

6 print b_{global} // prints 6

7 **{**

a_{local} $4^{8} \to 10^{9}$

8 $a_{local} = 4$

9 $a_{local} = a_{local} + b_{global} = 10$

10 $b_{global} = a_{local} + b_{global} = 16$

11 **g()**

12 print a_{global} // prints 5

13 print b_{global} // prints 16

}

14 **{**

b_{local} $3^{15} \to 11^{17}$

15 $b_{local} = 3$

16 $a_{global} = a_{global} + b_{local} = 8$

17 $b_{local} = a_{global} + b_{local} = 11$

18 **g()**

19 print a_{global} // prints 8

20 print b_{global} // prints 16

}

21 $a_{global} = a_{global} + b_{global} = 24$

22 **g()**

23 print a_{global} // prints 24

24 print b_{global} // prints 16

25 **g()**

23 print a_{global} // prints 24

24 print b_{global} // prints 16

}

Output 5 6 5 16 8 16 24 16 24 16

Question 4.1 (Dynamic Scoping). Give the output of the following program under static scoping

```
int a = 5;
int b = 6;

void f()
{

   g();

   { int a = 4;
     a = a + b;
     b = a + b;
     g();
   }

   { int b = 3;
     a = a + b;
     b = a + b;
     g();
   }

   a = a+b;
   g();

}

void g()
{
   print a;
   print b;
}

int main ()
{
     int a = 1;
     int b = 10;
     f();
     g();
}
```

Show your work!

$\boxed{\begin{array}{l} a_{global} = 5 \\ b_{global} = 6 \end{array}}$

main ()

$\boxed{\begin{array}{l} a_{main} \quad 1^{1} \rightarrow 4^{16} \rightarrow 28^{21} \\ b_{main} \quad 10^{2} \rightarrow 24^{10} \end{array}}$

1 $a_{main} = 1;$
2 $b_{main} = 10;$
3 **f()**
    4 **g()**
        5 print a_{main} // prints **1**
        6 print b_{main} // prints **10**
    7 **{**

$\boxed{a_{local} \quad 4^{8} \rightarrow 14^{9}}$

        8 $a_{local} = 4$
        9 $a_{local} = a_{local} + b_{main} = 14$
        10 $b_{main} = a_{local} + b_{main} = 24$
        11 **g()**
            12 print a_{local} // prints **14**
            13 print b_{main} // prints **24**
    }
    14 **{**

$\boxed{b_{local} \quad 3^{15} \rightarrow 7^{17}}$

        15 $b_{local} = 3$
        16 $a_{main} = a_{main} + b_{local} = 4$
        17 $b_{local} = a_{main} + b_{local} = 7$
        18 **g()**
            19 print a_{main} // prints **4**
            20 print b_{local} // prints **7**
    }
    21 $a_{main} = a_{main} + b_{main} = 28$
    22 **g()**
        23 print a_{main} // prints **28**
        24 print b_{main} // prints **24**
25 **g()**
    23 print a_{main} // prints **28**
    24 print b_{main} // prints **24**

}

Output 1 10 14 24 4 7 28 24 28 24