CSE340 FALL 2021 - HOMEWORK 3 SOLUTION

Due: Wednesday October 27, 2021 by 11:59 PM on Canvas/Gradescope

All submissions should be typed, no exceptions. There are 6 questions.

1. **(Lambda calculus: binding)**. For each of the following expressions. determine for each variable x the 𝜆x. it is bound to. I have numbered the variables and the abstractions. The variables are numbered using Arabic numerals and the abstractions are numbered using roman numerals and you should specify which variable numbers are bound to which abstraction number. For example, if variables 4 and 7 are bound to abstraction I, your answer should be of the form I ➙ 4 7 to indicate that abstraction I has variables 4, and 7 bound by it. If variable 3 is free, then your answer should have the form free ➙ 3.

**Example** ( 𝜆y. x y 𝜆x. y ) x

I 1 2 II 3 4

**Answer**  free ➙ 1 4

I ➙ 2 3

II ➙

Your answers need not be colored (look at old homework assignments for more examples. You should look at old HW3 and old HW4)

* 1. 𝜆x. x y z

I 1 2 3

**Answer**  free ➙ 2 3

I ➙ 1

* 1. ( 𝜆x. x y ) 𝜆z. y 𝜆y. z x

I 1 2 II 3 III 4 5

**Answer**  free ➙ 2 3 5

I ➙ 1

II ➙ 4

III ➙

* 1. 𝜆x. x ( y (𝜆x. x 𝜆y. x) y 𝜆x. y ) y x

I 1 2 II 3 III 4 5 IV 6 7 8

**Answer**  free ➙ 2 5 6 7

I ➙ 1 8

II ➙ 3 4

III ➙

IV ➙

* 1. ( 𝜆x. y 𝜆y. x x 𝜆z. 𝜆x. y z 𝜆z. y x ) y x

I 1 II 2 3 III IV 4 5 V 6 7 8 9

**Answer**  free ➙ 1 8 9

I ➙ 2 3

II ➙ 4 6

III ➙ 5

IV ➙ 7

V ➙

* 1. 𝜆z. y 𝜆y. x y 𝜆z. 𝜆x. (y x) 𝜆y. x y z

I 1 II 2 3 III IV 4 5 V 6 7 8

**Answer**  free ➙ 1 2

I ➙

II ➙ 3 4

III ➙ 8

IV ➙ 5 6

V ➙ 7

1. (**Reducible Expressions**) For each of the following, identify all the reducible expressions by highlighting the 𝜆x. , the t and the t’ of the reducible expressions. If there is more than one redex in the given expression, you should identify all the redexes, each one on a separate line. If there are no reducible expressions, you should say so in your answer.

**Example 1** ( 𝜆x. 𝜆x. x x )

**Answer** no redex

**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 ) )

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

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

* 1. ( 𝜆x. ( 𝜆x. x (𝜆x. x) x ) z ) x

**Answer**  ( 𝜆x. ( 𝜆x. (x (𝜆x. x)) x ) z ) x

( 𝜆x. ( 𝜆x. (x (𝜆x. x)) x ) z ) x

* 1. ( 𝜆x. y ) z ( 𝜆x. 𝜆x. ( 𝜆x. x ) 𝜆x. x y ) z

**Answer** ((( 𝜆x. y ) z ) ( 𝜆x. 𝜆x. ( 𝜆x. x ) 𝜆x. x y )) z

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

* 1. 𝜆x. y ( 𝜆x. ( 𝜆y. z y ) 𝜆z. x )

**Answer** (𝜆x. y ( 𝜆x. ( 𝜆y. z y ) 𝜆z. x ) )

* 1. 𝜆x. (𝜆x. w) y z 𝜆x. ( 𝜆y. x ) 𝜆z. x y

**Answer** ( 𝜆x. ((𝜆x. w) y ) z (𝜆x. ( 𝜆y. x ) 𝜆z. x y ) )

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

1. (**Alpha Renaming**) For each of the following redexes, do alpha renaming if it is needed to reduce the redex. If renaming is not needed, you should say so. You should highlight the renamed variable(s) in your answer. This question is only about renaming. You should not do the beta reduction.

**Example 1** ( 𝜆x. x x ) x

**Answer** no renaming needed

**Example 2** ( 𝜆x. 𝜆y. 𝜆z. x y 𝜆w. z ) ( x y z w)

**Answer** ( 𝜆x. 𝜆u. 𝜆v. x u 𝜆w. v ) ( x y z w)

**Example 3** ( 𝜆x. 𝜆y. 𝜆z. x y 𝜆w. z ) ( x z ) ( z w)

**Answer** ( 𝜆x. 𝜆y. 𝜆v. x y 𝜆w. v ) ( x z ) ( z w)

* 1. (𝜆x. x (𝜆x. 𝜆y. 𝜆z. x y z ) x ) ( y z )

**Answer** no renaming needed because the inner x is bound to the inner 𝜆x.

* 1. (𝜆x. (𝜆y. x) (𝜆z. x y z) ) y z

**Answer** (𝜆x. (𝜆u. x) (𝜆z. x y z) ) y z

* 1. (𝜆x. 𝜆y. 𝜆z. x y 𝜆x. x z ) ( y z )

**Answer** (𝜆x. 𝜆u. 𝜆v. x u 𝜆x. x v ) ( y z )

* 1. 𝜆y. (𝜆x. 𝜆y. x 𝜆x. 𝜆z. x y z ) (𝜆z. y z )

**Answer** 𝜆y. (𝜆x. 𝜆u. x 𝜆x. 𝜆z. x u z ) (𝜆z. y z )

* 1. (𝜆y. 𝜆z. 𝜆y. x x z ) (𝜆y. 𝜆z. y z )

**Answer** no renaming needed because after the beta reduction the y and z highlighted in blue, would still be bound to 𝜆y. 𝜆z.

1. (**Beta reductions**) For each of the following expressions, identify the redexes and, for each redex, do only one beta reduction step. You are only asked to reduce the redexes of the expression, each of them separately in one step; you are not asked to keep on reducing until there are no redexes left. If renaming is needed you should do renaming first before doing the beta reduction.

**Example 1** ( 𝜆x. x x ) x

**Answer** ( 𝜆x. x x ) x →ᵝ x x

**Example 2** ( 𝜆x. 𝜆y. 𝜆z. x y 𝜆w. z ) ( x y z w)

**Answer** ( 𝜆x. 𝜆y. 𝜆z. x y 𝜆w. z ) ( x y z w) →𝛼 ( 𝜆x. 𝜆u. 𝜆v. x u 𝜆w. v ) ( x y z w)

( 𝜆x. 𝜆u. 𝜆v. x u 𝜆w. v ) ( x y z w) →ᵝ 𝜆u. 𝜆v. ( x y z w) u 𝜆w. v

**Example 3** (𝜆x. (𝜆y. x ) y ) z

**Answer** // This expression has two redexes. You should show one beta reduction for each redex separately not one // after the other

(𝜆x. (𝜆y. x ) y ) z →ᵝ (𝜆y. z ) y // beta reduction for first redex

(𝜆x. (𝜆y. x ) y ) z →ᵝ (𝜆x. x ) z // beta reduction for second redex

// The examples above clearly show that you should answer by highlighting the redexes and also highlight the result of the

// reduction (and the renaming if applicable). You should follow the same format as the examples above

* 1. ( x (𝜆x. z ) ( x z ) ) ( 𝜆x. (𝜆x. z x ) ( x z ) )

**Answer**

* ( x (𝜆x. z ) ( x z ) ) ( 𝜆x. (𝜆x. z x ) ( x z ) ) →ᵝ ( x (𝜆x. z ) ( x z ) ) ( 𝜆x. ( z ( x z ) ) )
  1. (𝜆x. (𝜆y. x) (𝜆z. x y z) ) ( y z )

**Answer**

* (𝜆x. (𝜆y. x) (𝜆z. x y z) ) ( y z ) →𝛼 (𝜆x. (𝜆u. x) (𝜆v. x y v) ) ( y z )

(𝜆x. (𝜆u. x) (𝜆v. x y v) ) ( y z ) →ᵝ (𝜆u. ( y z )) (𝜆v. ( y z ) y v)

* (𝜆x. (𝜆y. x) (𝜆z. x y z) ) ( y z ) →ᵝ (𝜆x.( x) ) ( y z )
  1. (𝜆y. x 𝜆z. x y z ) (𝜆y. 𝜆z. y z )

**Answer**

* (𝜆y. x 𝜆z. x y z ) (𝜆y. 𝜆z. y z ) →ᵝ x 𝜆z. x (𝜆y. 𝜆z. y z ) z
  1. (𝜆x. 𝜆y. 𝜆z. x y z ) ( 𝜆z. 𝜆y. (𝜆z. y z ) y z )

**Answer**

* (𝜆x. 𝜆y. 𝜆z. x y z ) ( 𝜆z. 𝜆y. (𝜆z. y z ) y z ) →ᵝ 𝜆y. 𝜆z. ( 𝜆z. 𝜆y. (𝜆z. y z ) y z ) y z
* (𝜆x. 𝜆y. 𝜆z. x y z ) ( 𝜆z. 𝜆y. (𝜆z. y z ) y z ) →ᵝ (𝜆x. 𝜆y. 𝜆z. x y z ) ( 𝜆z. 𝜆y. (y y ) z )
  1. (𝜆y. 𝜆x. 𝜆z. x 𝜆x. 𝜆z. x y z ) ( (𝜆z. x y ) (𝜆x. y z ) )

**Answer**

* (𝜆y. 𝜆x. 𝜆z. x 𝜆x. 𝜆z. x y z ) ( (𝜆z. x y ) (𝜆x. y z ) ) →ᵝ (𝜆y. 𝜆x. 𝜆z. x 𝜆x. 𝜆z. x y z ) ( ( x y ) )

* (𝜆y. 𝜆x. 𝜆z. x 𝜆x. 𝜆z. x y z ) ( (𝜆z. x y ) (𝜆x. y z ) ) →𝛼 (𝜆y. 𝜆u. 𝜆v. u 𝜆w. 𝜆r. w y r ) ( (𝜆z. x y ) (𝜆x. y z ) )

(𝜆y. 𝜆u. 𝜆v. u 𝜆w. 𝜆r. w y r ) ( (𝜆z. x y ) (𝜆x. y z ) ) →ᵝ 𝜆u. 𝜆v. u 𝜆w. 𝜆r. w ( (𝜆z. x y ) (𝜆x. y z ) ) r

1. (**Evaluating Expressions**)
   1. What do each of the following expressions evaluate to?
      1. 2 (3 scc) 4

**Answer** ( 3 scc ) 4

= (3 scc) ( (3 scc) 4 )

= (3 scc) ( scc ( scc ( scc 4 ) ) ) )

= (3 scc) 7

= 10

* + 1. 3 fls tru fls tru fls

**Answer** 3 fls tru fls tru fls

= (fls (fls (fls tru))) fls tru fls

= fls tru fls

= fls

* + 1. snd ( fst ( 2 pair tru fls ) fls)

**Answer** snd ( fst ( 2 pair tru fls ) fls )

= snd ( fst ( (pair (pair tru)) fls ) fls

= snd ( (pair tru) fls )

= fls

* 1. Let

a = pair 1 2

b = pair 3 4

c = pair a b

d = pair b a

e = pair c d

f = pair d c

calculate 3 fst (pair e f)

**Answer** 3 fst (pair e f)

= fst (fst (fst (pair e f)))

= fst (fst e)

= fst (fst (pair c d))

= fst c

= fst (pair a b)

= a

= pair 1 2

* 1. Let next = 𝜆p. pair (snd p) (plus (fst p) (snd p))
     1. Is next pair 0 1 equivalent to next pair 0 1 ?

**Answer** This was supposed to have parentheses. Obviously the two expressions given are the same!

Evaluate each of the following

* + 1. next (pair 0 1)

**Answer** next (pair 0 1)

= pair (snd (pair 0 1)) (plus (fst (pair 0 1)) (snd (pair 0 1))

= pair 1 (plus 0 1)

= pair 1 1

* + 1. next ( next (pair 0 1))

**Answer** next ( next (pair 0 1))

= next (pair 1 1)

= pair 1 2

* + 1. n next (pair 0 1) as a function of a Church’s numeral n

Answer The next few values are pair 2 3 pair 3 5 pair 5 8 and so on. This is from the Fibonacci sequence which is 0 1 1 2 3 5 8 ...

n next (pair 0 1) = pair Fn+1 Fn+2

where Fn is the n’th Fibonacci number starting with 0 for n = 1

1. Write an iterative function to calculate the sum of the even integer less than or equal to n. The function should have the form

sum\_even = 𝜆n. fst ( n next\_sum\_even (pair 0 0) )

You are asked to write the function next\_sum\_even The idea is to have a pair consisting of two parts. One part is the running sum and the other part is an index. For example, to get the sum of the first n integers, we can write the following

sum = 𝜆n. fst ( n next\_sum (pair 0 0) )

where

next\_sum = 𝜆p. pair (plus (fst p) (scc (snd p))) (scc (snd p))

let us see how next\_sum works on a number of examples

next\_sum (pair 0 0) = pair (plus 0 (scc 0)) (scc 0) = pair 1 1

next\_sum (pair 1 1) = pair (plus 1 (scc 1)) (scc 1) = pair 3 2

next\_sum (pair 3 2) = pair (plus 3 (scc 2)) (scc 2) = pair 6 3

In general, if we apply next\_sum n times, we get

next\_sum ( next\_sum ( next\_sum … ( next\_sum (pair 0 0) ) … ) = pair (1+2…+n) n

In your answer, you cannot simply increment the index by 2 because that would result in the sum of the first 2n even integers. You need to make sure that you are only adding numbers up to n. You can use any of the functions we covered in class including times (multiplication), equal (equal to), lteq (less than or equal).

**Solution 1.**

The idea is to consider the sequence 0, 2, 4, … 2n and only add the values that are less than or equal to n. In the pseudocode, sum is the running sum and j is the sequence of even numbers less than or equal to 2n.

Here is the pseudocode for the solution

sum = 0

j = 0

for i = 1 to n

sum = sum + if (j+2 <= n) then j+2 else 0

j = j+2

Notice how we are adding if (j+2 <= n) then j+2 else 0 to sum. This is a value that evaluates to j+2 if j+2 is <= n and evaluates to 0 if j+2 > n. In the lambda calculus solution (fst p) will hold the sum and (snd p) will hold j

next\_sum\_even = 𝜆p. pair (plus (fst p)

( (lteq (plus 2 (snd p)) (scc n)) // if (snd p)+2 <= n

(plus 2 (snd p)) // add (snd p)+2 to sum

0 // else add nothing

)

)

(plus 2 (snd p)) // j = j+2

**Solution 2.**

The second solution might be more intuitive. The pseudocode for the solution is

sum = 0

j = 0

for i = 1 to n

if j is even

sum = sum + j

j = j+1

else

sum = sum + 0

j = j+1

This can be written in lambda calculus as follows

next\_sum\_even = 𝜆p. (is\_even (snd p))

(pair (plus (fst p) (snd p)) (scc (snd p))) // pair (sum+j) (j+1)

(pair (fst p) (scc (snd p))) // pair (sum) (j+1)

is\_even is a function that I covered in class.

For completeness, here is the implementation of is\_even

is\_even = 𝜆n. fst ( n swap (tru fls) )

where

swap = 𝜆p. pair (snd p) (fst p)

another implementation of is\_even is the following

is\_even = 𝜆n. n NOT tru

In this implementation, we start with tru and we negate it n times.