CSC3002F Lambda Calculus Tutorial

18th April 2015

Please ensure that both your name and student number are on all answer sheets that you hand-in.

[100] Marks total. **Approximately 1½ – 2 hours**

You may ask questions to the tutors but each question will incur a 5 mark cost.

Question 1 Expression Analysis [15]

Recursively analyse each of the following lambda expression and describe its structure.

- If the expression is a function, say so, and then find the bound variable and the body expression, and then analyse the body expression.
- If the expression is an application, say so, and identify the function and argument expressions, and then analyse both the function and argument expressions.
- Do this until the expressions are fully analysed. If there are any free variables identify those as well.

So for λa.(a λb.(b a)) you could say (using indentation as you get further in and pick some unique naming convention):

```
λa.(a (λb.(b a)))
function
bound var: a
body 1: (a (λb.(b a)))
     body 1: application
     function expr: name a (bound)
     argument expr arg 1: λb.(b a)
          arg 1: function
          bound var: b
          body 2: (b a)
               body 2: application
               function expr: name b (bound)
               argument expr: name a (bound)
```

Now do the same for

((λx.y x a) (λz.z x))

Answer: Give 1 mark for each correct step, they are the numbered steps below (max 15)

```
((λx.(y x) a) (λz.z x)) ; Add brackets
                        ; (not required, but simplifies things)

1. application
2. function expr fun 1:  (λx.(y x) a)
3. argument expr arg 1:  (λz.z x)
       fun 1:  (λx.(y x) a)
4.     function (or abstraction)
5.     bound var: x
6.     body 1: (y x) a

       arg 1: (λz.z x)
```

```
7.     function
8.     bound var: z
9.     body 2: z x

           body 1: (y x) a
10.        application
11.        function expr fun 2: (y x)
12.        argument expr: name a, free variable

               fun 2: (y z)
13.            application
14.            function expr: name  y (free)
15.            argument expr: name x (bound)

           body 2: z x
16.        application
17.        function expr: name z (bound)
18.        argument expr: name x (free)
```

Question 2 Equivalence of Functions [15]

You can show that two functions are the same by reducing one to another (or both to the same function). You can also do this by applying them to an arbitrary argument expression, `<expr>` say, and then showing that the results are the same.

Use this last method and apply both the following functions, A and B, to the same expression and show that the result is the same (and use => as shorthand for β-reduction):

```
A. identity
B. (apply (apply identity))
```

Recall that we had: `(define identity λx.x)` and

`(define apply λfunc.λarg.(func arg))`.

Answer: Must use the specified method, otherwise max half marks.
(do not need to label eta conversions, those are, where there is λ that does nothing with its argument). [2] for each step, except if they do not show two equal expressions for A and B they lose 2 marks and can get max 13.

```
A: (identity <expr>) => <expr> [2]

B:   ((apply (apply identity)) <expr>)
==   ((apply (λfunc.λarg.(func arg) identity)) <expr>)   ; definitions
=>   ((apply λarg.(identity arg)) <expr>)
=>   ((apply identity) <expr>)                      ; really η-conversion
=>   ((λfunc.λarg.(func arg) identity) <expr>)
=>   (λarg.(identity arg) <expr>)
=>   (identity <expr>)                          ; really η-conversion
=>   <expr>
```

Thus A and B give same result, namely <expr>. Must have something like this for full marks

Question 3 β-Reduction [20]

Reduce the following in any order. **Give the steps** of β- α- η-reduction, labelling them correctly, and conclude with the simplest form or state "diverges" if it does not terminate. Indicate what is being substituted with the [ ] notation.

Let $t = \lambda x.\lambda y.x$
$f = \lambda x.\lambda y.y$
$i = \lambda x.\lambda y.\lambda z.(x\ y\ z)$

Obtain the normal form of the following λ-expression if it has one:

$$(i\ f\ x)\ (i\ t\ y\ z)$$

Hint: add all parentheses; it may be useful to use $j = (i\ t\ y\ z)$ to abbreviate some of the expressions.

Answer: [1] mark for each step and [1] for the label, can β on the arrow ($\rightarrow_\beta$) or as a comment, or as we do say that => represents β reduction.

If they skip *one* step but get it right you can give them the marks. Extra steps do not mean extra marks! They must have the last α conversion (step 9). Many missing steps must mean losing marks. If they go wrong then give marks for correct bits, but ensure that every mistake loses 3 marks (so many correct steps but mistakes ≠ full marks!!).

Below => represents β reduction

```
     (((i f) x) ((i t) y) z)      ; Adding all parentheses if they want
     =  (λx.λy.λz.x y z) f x j    ; expanding first i
                                  ; and substituting j as suggested
1.   => (λy.λz.f y z) x j             ; [f/x]
2.   => (λz.f x z) j                  ; [x/y]
3.   => f x j                    ; [j/z]
     =  (λx.λy.y) x j            ; expand f
4.   => (λy.y) j                 ; [x/x] and discarded
5.   => j                        ; [j/y] identity
     =  (λx.λy.λz.x y z) t y z    ; expand j
6.   => (λy.λz.t y z) y z         ; [t/x]
7.   => (λz.t y z) z              ; [y/y]
8.   => t y z                    ; [z/z]
     =  (λx.λy.x) y z            ; expand t
9.   →α (λx.λa.x) y z   ; α conversion to stop second free y being captured
10.  => (λa.y) z        ; [y/x]
11.  => y               ; [z/a] and discarded.
```

If they do not do the α conversion it is likely that the result will be z, that is wrong!

Question 4 Normal Order Reduction [15]

Reduce the following in normal order. That is, reduce the outermost, leftmost redex first. Give the steps of β- α- η-reduction, labelling them correctly, and conclude with the simplest form.

$$(\lambda x.(*\ x\ x\ x))\ \Big((\lambda y.(+\ y\ 1))\ 2\Big)$$

Perform any arithmetic using the normal rules of arithmetic as a separate step, i.e., do not convert to λ expressions, but write 6 instead of, say, (* 2 3) in the next step. Stick to the normal order indicated and only perform the arithmetic when normal order requires that arithmetic function be reduced. Label each step with what you have done. Indicate what is being substituted with the [ ] notation.

Answer: Use => to indicate β reduction. Must be done in Normal order, if not then no marks. The arithmetic steps need not be labelled as β reductions (as I have done) but instead can have an = (function application)

Give 1 mark for each step and 1 mark for each label (15 max).

```
Normal
(λx.(* x x x)) ((λy.(+ y 1)) 2)
1.   => (* ((λy.+ y 1) 2) ((λy.+ y 1) 2) ((λy.+ y 1) 2))
                                                    ; [((λy.(+ y 1)) 2)/x]
2.   => (* (+ 2 1) ((λy.+ y 1) 2) ((λy.+ y 1) 2))   ; [2/y]
3.   => (* (3) ((λy.+ y 1) 2) ((λy.+ y 1) 2))       ; arith: + 2 1 = 3
4.   => (* 3 (+ 2 1) ((λy.+ y 1) 2))                ; [2/y]
5.   => (* 3 (3) ((λy.+ y 1) 2))                    ; arith: + 2 1 = 3
6.   => (* 3 3 (+ 2 1))                            ; [2/y]
7.   => (* 3 3 3)                                   ; arith: + 2 1 = 3
8.   => 27                                          ; arith: * 3 3 3 = 27
```

Question 5 Applicative versus Normal Order Reduction [15]

a. Reduce the following in applicative order (the innermost redex first). Perform any arithmetic using the normal rules of arithmetic, i.e., do not convert to λ expressions but stick to the order indicated (applicative l). Label each step with what you have done, including substitutions. [11]

Give the steps of β- α- η-reduction, labelling them correctly, and conclude with the simplest form or state "diverges" if it does not terminate.

$$\left(\lambda x.\,(*\ x\ x\ x)\right)\left(\left(\lambda y.\,(+\ y\ 1)\right)\ 2\right)$$

Hint: the first step is to figure out what is innermost.

b. Once you have done the reduction say if normal order (Question 4) or applicative order is the most efficient from a computational point of view. Explain. [4]

Answer: If they do normal-order (as in the previous question) then they cannot get any marks. Then [2] marks for each step and [1] for the label. **Max 11 marks for this bit.**

If they skip a step but get it right you can give them the marks.

If they go wrong then give marks for correct bits, but ensure that every mistake loses 3 marks. If they do extra steps that are correct (I don't see how in this example) then they get marks to a max of 15 of course.

Best is to add and label the brackets as follows:

Applicative:

```
(λx.(* x x x)) ((λy.(+ y 1)) 2)
=> (λx.* x x x) (+ 2 1)          ; [2/y]
=>  (λx.* x x x) 3               ; arith
=> (* 3 3 3)                     ; [3/x]
=> 27                            ; Arith
```

Efficiency: clearly the **applicative order reduction is more efficient** [2] since there are **fewer arithmetic operations** [2]. Can also say that **less memory** is used [2].

Max 4 marks for this bit

Question 6 Logic [20]

Write a λ calculus representation for *implies* (material implication). Its truth table, as you will recall, is as follows:

x	y	x implies y
True	True	True
True	False	False
False	True	True
False	False	True

You may find the following definitions helpful:

$$true = \lambda x.\lambda y.x$$
$$false = \lambda x.\lambda y.y$$
$$cond = \lambda e1.\lambda e2.\lambda c.\big((c\ e1)\ e2\big)$$

a. Reduce your answer for *implies* so that it is only in terms of λ expressions and, perhaps, *true* or *false*. For each step of the reduction say what it was (α- β- η-reduction) and indicate the substitution using the [ ] notation. [8]

b. Show that your version of *implies* satisfies all cases of the truth table. [12]

Answer: [1] for each step (the numbered ones) and [1] for correct label. 8 max
We can say that if x is true then imply is the same as y otherwise if x is false then it is true.

```
== if x then y else true
```

Now the conditional expression which they are given binds to three arguments: e1 for when it is true, e2 for when it is false and c, the condition.

```
cond y true x
```

bracket it if you want according to the application binding to the left and you get

```
(((cond y) true) x)
```

and now you need to feed x and y

```
λx.λy.(((cond y) true) x)
```

```
1. imply = λx.λy.(((cond y) true) x)    ; start with this expression and
                                        ; get rid of cond via β reduction
     =  λx.λy.(λe1.λe2.λc.((c e1) e2)) y true x
2.   => λx.λy.(λe2.λc.c y e2) true x          ; [y/e1]
3.   => λx.λy.(λc.c y true) x                ; [true/e2]
4.   => λx.λy.(x y true)                     ; [x/c]
   Can also say λx.λy.((x y) true)
```

Now need to check the truth table for all 4 cases: 3 marks each, do not need all steps nor the substitutions. Give 1 mark for starting with the correct test, 1 for some working that is correct and 1 mark for result. (12 max)

```
imply true true
=  (λx.λy.x y true) true true
=> (λy.true y true) true
=> (λx.λy.x) true true
=> (λy.true) true
=> true       (or λx.λy.x)       √
```

```
imply true false
=  (λx.λy.x y true) true false
=> (λy.true y true) false
=> (λx.λy.x) false true
=> (λy.false) true
=> false      (or λx.λy.y)       √
```

```
imply false true
=  (λx.λy.x y true) false true
=> (λy.false y true) true
=> (λx.λy.y) true true
=> (λy.y) true
=> true      (or λx.λy.x)       √
```

Now we can prove the same for imply false false but can also observe from the previous proof that if x is true then y is ignored and the result is always true (that is full 3 marks)

```
imply false false
=  (λx.λy.x y true) (false) true
=> (λy.false y true) (true)
=> (λx.λy.y) (true) true
=> (λy.y) (true)
=> true      (or λx.λy.x)       √
```