CSC173 Lambda Calculus Exercises

1 Practice and Practical Help

Our "textbook", Greg Michaelson's AN INTRODUCTION TO FUNCTIONAL PROGRAMMING THROUGH LAMBDA CALCULUS, is pointed at from both "readings" and "resources" links and the course schedule. It has relevant and representational problems at the end of the chapters, all with answers in the back. The previous exams at the "resources" link might be helpful.

2 Exercises: Week 1

EXERCISE 1.
Analyse the following lambda expression to clarify its structure. If the expression is a function, identify the bound variable and the body expression, and then analyse the body expression. If the expression is an application, identify the function and argument expressions, and then analyse the function and argument expressions. Here's an example:
Example: λx.(x λy.(y x))

Example Answer: note indentation.

```
\la = lambda

<function>
  <bound variable> - x
  <body> - (x \la y.(y x))
    <application>
      <function exp> - <name> - x
      <argument exp> - \la y.(y x)
        <function>
        <bound variable> - y
        <body> - (y x)
          <application>
          <function exp> - <name> - y
          <argument exp> - <name> - x
```

Now you do this one:
λx.λy. ((λx.yx p) (λz.z x))

EXERCISE 2.
Make all parentheses explicit in these λ- expressions:

A. (λp.pz) λq.w λw.wqzp
B. λp.pq λp.qp

EXERCISE 3.
In the following expressions say which, if any, variables are bound (and to which λ), and which are free.

A. λs.s z λq.s q
B. (λs. s z) λq. w λw. w q z s
C. (λs.s) (λq.qs)
D. λz. (((λs.sq) (λq.qz)) λz. (z z))

E. Rewrite the above expressions as necessary to remove name clashes.

EXERCISE 4.

Put the following expressions into (beta) normal form (use β-reduction as far as possible, α-conversion as needed). Remember we're assuming left-association as shown in part A below.

A. (λz.z) (λq.q q) (λs.s a) = ((λz.z) (λq.q q)) (λs.s a)
B. (λz.z) (λz.z z) (λz.z q)
C. (λs.λq.s q q) (λa.a) b
D. (λs.λq.s q q) (λq.q) q
E. ((λs.s s) (λq.q)) (λq.q)

3 Exercises: Week 2

EXERCISE 1.
Write a version of makepair called `fun-pair` that takes five arguments including f,g,x,y and makes a pair with first element f(x) and second g(y).

EXERCISE 2.
Recall we defined logical OR:
`def or = λx. λy.(((cond true) y) x),`
which simplified to
`def or = λx. λy.((x true) y).`

A: Write a *C-language-like x?y:z conditional expression* (or a simple "if-then-else" statement) that implements the Boolean function NAND ($\neg(P \wedge Q)$). Only use the variables `x, y` the operator `not` and the constant `false` in your conditional expression.

B:
Convert your expression for NAND into a λ-calculus `cond` expression: it should have two λs, one `cond`, and however many `false, not, x, y`s you need.

C:
Evaluate and simplify the inner body of this expression (removing `cond`, to get a more elegant function for NAND:

λx. λy. (<simple-expression>)

D:
Now apply the simplified expression for NAND to the two arguments (x=false, y=true). Show it evaluates to the correct answer. Since the expression will involve strictly true, false, not, can convert to all *true, false*. Then remember they are *select-first, select-second*, and you can do the job in one line. No need to expand to λ-level.

EXERCISE 3.
Show the functions a) and b) below evaluate to the same thing for the Boolean argument pair (x,y) = (true, false).

a) not (and x y) ;; NAND

b) (or (not x) (not y))

EXERCISE 4.
Read the lecture notes to refresh on definitions of numbers, successor, addition and multiplication in Church's encoding. Recall e.g. that we represent zero by select-second, and

def succ = λw.λy.λx.(y ((w y) x)

A: The lecture overheads derive (succ zero) => 1. Now derive (succ 1) => 2.

B: In the manner of the addition demonstration in the lecture, verify that to compute 2+2,
i) we can use 2S2 – that is, using the basic λ-level definitions of 2 and S, 2S2 reduces to SS2,
ii) and that (again using the basic la-level definitions) reduces to the λ-level definition of 4.

C: Continuing the exploration of the multiplication demonstration in the lecture, verify that (λz. 2(2 z)) is 4.

D: Does Church encoding work with call-by-value (applicative) evaluation? In particular, what is the applicative-order evaluation of
(S Z) -> ?

Only one β-reduction is allowed and the answer we want is:
1 =λsz.(s z).

4 Exercises: Weeks 3, 4

EXERCISE 1.

A. The version of Y we've seen so far is
Y = λf.(λs.(f (s s)) λs.(f (s s)))
Evaluate (Y <FUN>) using normal order.

B. Evaluate (Y <FUN>) using applicative order.

C. Remember eta reduction? Show that λx. (<expr> x) may be replaced by <expr>: (apply both to a general argument <arg>.)

That's eta-reduction. *Eta-expansion* replaces the other way.

D. Wikipedia claims that Yapp below is a paradoxical combinator (fixed point operator) like Y, only for applicative order evaluation:

```
Yapp = λf.((λs.f(λy.  (s s) y)) (λs.f(λy.(s s) y))).
```

Notice the innermost eta-expanded versions of `(s s)`... abstraction at work again. Now evaluate `(Yapp <FUN>)` using applicative order.

EXERCISE 2.

In the 1960's Dana Scott presented another way to encode constructors for recursive data types (e.g. for lists and numbers: nil, cons, car, zero, succ).

Scott's coding looks similar to Church's but acts differently. Lambda abstractions occur throughout the endoding (notice with Church there is one lambda at the very beginning). (Notes of possible interest: Operations are best thought of as using continuations. Also Scott encoding works with applicative (call by value) evaluation.) Let's consider natural numbers again. We'll use our shortcut notation: write
`λ.s (λz.  z)` (select second) as `λsz.z`.

Constructors zero (Z) and successor (S):

```
Z = λsz.z ;; same as Church
S = λx.λsz.  s x ;; Church has more complex use of select-1st.
```

If this all works, then applicative reduction (`->`) should yield:

```
0 = -> λsz.z
1 = S Z -> λsz.s 0
2 = S( S Z) -> λsz.s 1
3 = S( S( S Z)) -> λsz.s 2
```

A: Using the definitions, verify that
`2 = λsz.s (λsz.s (λsz.z))`

B: What is the crucial difference between Church's `s`'s and Scott's `s`'s?

C: Background:
We can think of Scott encoding as using continuations (telling us what happens next), each chosen as in a case statement.

Recall known algorithms for predecessor(N) using Church encoding are O(N). Scott's encoding allows

```
pred = λx.x (λp.p) 0
```

which is O(1). It's a little case statement that chooses between two continuations based on a test value. Here `pred` takes a numeral `x`, which is the test that chooses the case. It is applied to the two continuations. Its outermost λ(either select-1st if x¿0, else select-2nd) is "used up" in the evaluation, replaced by an `identity` continuation if x¿0 else 0. If x¿0, then `identity` returns x-1, else all that's left is 0.

C:
Verify at the lowest level (λs and variables) that
`(pred 2) = 1`.

D: Using `pred` above as a template, give a similar-looking implementation of `iszero`

E. How do we implement addition in Scott encoding? I want to hear your strategy, approach, in as much detail as you can practically give it. Complete ('correct') answer not expected.

EXERCISE 3.

A:

What gets displayed by this Scheme expression?

```
(display
  (call/cc (lambda (cc)
             (display "Be Well\n")
             (cc "Do Good Work.\n")
             (display "Stay in Touch.\n"))))
```

B: What gets printed when the following is evaluated?

```
(define cont #f)

(+ 1 (call/cc
       (lambda (myfun)
         (set! cont myfun)
         (+ 2 (myfun 3)))))
```

C: Having run the code in B, we continue typing at the listener:

```
(+ 3 (cont 5))
```

What gets printed now?

EXERCISE 4.

Given:

`Y = λf.(λx.  f(x x)) (λx.  f(x x))`

and a high-level abstracted definition of add:

`add = λf.λa.λb.  (if a = 0 then b else f(succ a)(pred b)).`

Now show the computation of 2+1 (`add 2 1`) recursively using Y. You don't need to expand any operators except `Y` and `add`, and those only when necessary! Infix notation is OK (e.g. 4+5 would 'reduce' to 9). So feel free to use n+1 for succ(n), n-1 for pred(n). I'll get you started (and finished)

```
(Y add) 2 1 =>
=>...=>
3
```