Assignment 2(Lambda Calculus)

Submission of assignments is optional. Submitted assignments will not be graded. If your solution differs from the posted solution, please submit it and indicate clearly why you think your solution is correct.

1 Multiple Choice

Select the right answer:

- $\lambda x.\ (y\ z)$ and $\lambda x.\ y\ z$ are equivalent
  1. *True*
  2. *False*
- The term $\lambda x.x\ a\ b$ is equivalent to which of the following?
  1. $(\lambda x.x)\ (a\ b)$
  2. $(((\lambda x.x)\ a)\ b)$
  3. $\lambda x.(x\ (a\ b))$
  4. $(\lambda x.((x\ a)\ b))$
- $(\lambda x.y)z$ can be beta-reduced to
  1. y
  2. $y\ z$
  3. z
  4. *Cannot be reduced*
- Which of the following reduces to $\lambda z.z$
  1. $(\lambda y.\lambda z.x)\ z$
  2. $(\lambda z.\lambda x.z)\ y$
  3. $(\lambda y.y)\ (\lambda x.\lambda z.z)\ w$
  4. $(\lambda y.\lambda x.z)\ z\ (\lambda z.z)$
- Which of the following expressions is α-*equivalent* to (α-converts from) $(\lambda x.\lambda y.x\ y)y$
  1. $\lambda y.y\ y$
  2. $\lambda z.y\ z$

3. $(\lambda x.\lambda z.x\ z)\ y$
4. $(\lambda x.\lambda y.x\ y)\ z$

- β-reducing the following term produces what result?

$$(\lambda x.x\ \lambda y.y\ x)\ y$$

1. $y\ (\lambda z.z\ y)$
2. $z\ (\lambda y.y\ z)$
3. $y\ (\lambda y.y\ y)$
4. $y\ y$

- β-reducing the following term produces what result?

$$\lambda x.(\lambda y.y\ y)\ w\ z$$

1. $\lambda x.w\ w\ z$
2. $\lambda x.w\ z$
3. $w\ z$
4. *Does not reduce*

2 Problems

1. Make all parentheses explicit in the following $\lambda-$expressions
   (a) $\lambda x.x\ z\ \lambda y.x\ y$
   (b) $(\lambda x.x\ z)\ \lambda y.w\ \lambda w.w\ y\ z\ x$
   (c) $\lambda x.x\ y\ \lambda x.y\ x$
2. Find all free (unbound) variables in the following $\lambda-$expressions
   (a) $\lambda x.x\ z\ \lambda y.x\ y$
   (b) $(\lambda x.x\ z)\ \lambda y.w\ \lambda x.w\ y\ z\ x$
   (c) $\lambda x.x\ y\ \lambda x.y\ x$
3. Give the result of performing the following substitutions:
   (a) $[(\lambda y.x\ y)/x](x\ (\lambda x.y\ x))$
   (b) $[(\lambda x.x\ y)/x](\lambda y.x\ (\lambda x.x))$

4. Apply $\beta-$reduction to the following $\lambda-$expressions as much as possible

   (a) $(\lambda z.z)\ (\lambda y.y\ y)\ (\lambda x.x\ a)$

   (b) $(\lambda z.z)\ (\lambda z.z\ z)\ (\lambda z.z\ y)$

   (c) $(\lambda x.\lambda y.x\ y\ y)\ (\lambda a.a)\ b$

   (d) $(\lambda x.\lambda y.x\ y\ y)\ (\lambda y.y)\ y$

   (e) $(\lambda x.x\ x)\ (\lambda y.y\ x)\ z$

   (f) $(\lambda x.\ (\lambda y.\ (x\ y))\ y)\ z$

   (g) $((\lambda x.x\ x)\ (\lambda y.y))\ (\lambda y.y)$

   (h) $(((\lambda x.\lambda y.(x\ y))\ (\lambda y.y))\ w)$

5. Show that the following expression has multiple reduction sequences

$$(\lambda x.y)\ ((\lambda y.y\ y\ y)\ (\lambda x.x\ x\ x))$$

6. Verify the following by applying the $\beta-$axiom:

   (a) $\boldsymbol{S\ K\ I\ I = I}$, where $\boldsymbol{S}$ is $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$, $\boldsymbol{K}$ is $\lambda x.\lambda y.x$, and $\boldsymbol{I}$ is $\lambda x.x$

   (b) $(\lambda x.x\ x)\ \boldsymbol{I\ I = I}$
   Note that, by convention, the left-hand side of part (b) reads as $((\lambda x.x\ x)\ \boldsymbol{I})\ \boldsymbol{I}$

3 Lambda Calculus Encodings

3.1 Numbers

Part of understanding a computational model is knowing its **expressiveness**: what range of programs can be written in the model? For example, in the language of arithmetic, we know that every program must evaluate to a value, i.e. it can never loop. While most normal programming languages permit *partial* functions (functions that could return a result or loop forever), arithmetic is a language of *total* functions (they always return a result).

Intuitively, this means that arithmetic is a less expressive than a more general programming language. the set of programs expressible in arithmetic is a strict subset of those expressible in, say Python or Java. What about the *lambda calculus*? It seems like such a simple language, but can we formally reason about its expressiveness? It turns out that the lambda calculus is equivalent in expressiveness to a Turing machine – this result is called the **Church-Turing thesis**. Both of these computational models can express **computable functions.** Informally, a computable function is "a function on the natural numbers computable by a human being following an algorithm, ignoring resource limitations."

In the context of the lambda calculus, this idea will seem strange. If we only have variables and functions, how are we supposed to express something as simple as $2 + 2$? The key idea is that we have to provide a mapping from constructs in the lambda calculus to natural numbers, which allows us to interpret the result of a lambda calculus program as a number. This is called a *Church encoding* (after Alonzo Church, one of the founders of computer science and the inventor of the lambda calculus). In the remainder of this section, we'll work through a few exercises to understand how this works.

1. We will abbreviate Church-encoded natural numbers as n_λ. For example, We can represent the first n numbers as follows:

$$\begin{aligned} 0_\lambda &= \lambda x.\lambda y.y \\ 1_\lambda &= \lambda x.\lambda y.x\ y \\ 2_\lambda &= \lambda x.\lambda y.x\ (x\ y) \\ &\vdots \\ n_\lambda &= \lambda x.\lambda y.x^n\ y \end{aligned}$$

   (Read $x^n y$ as an abbreviation for $x(x(x(x(\cdots(x\ y)\cdots))))$, where x appears n times.)

   With the above representation, we can define the *successor* function *Succ* as:

$$Succ = \lambda u.\lambda x.\lambda y.x\ (u\ x\ y).$$

   Show, by applying the $\beta-$axiom, that $Succ\ n = n + 1$, that is,

$$Succ\ (\lambda x.\lambda y.x^n y) = (\lambda x.\lambda y.x^{n+1} y).$$

2. Observe that for any Church numeral n, the lambda expression $n\ f\ v$ corresponds to applying f exactly n times to v. So $1\ f\ v = f\ v, 2\ f\ v = f\ (f\ v)$ and so on. With this in mind:

   (a) Define a λ-calculus term *plus* that adds Church numerals. That is, plus should have the property that:

$$plus\ m_\lambda\ n_\lambda \equiv (m + n)_\lambda$$

   where $\equiv$ denotes $\alpha\beta\eta$-equivalence of λ-calculus terms. (You may want to use the *Succ* function in your answer).

   (b) Show that $2_\lambda + 1_\lambda \equiv 3_\lambda$

   (c) Define a λ-calculus term *mul* that multiplies Church numerals.

3.2 Church Booleans

To introduce conditional logic, i.e ifs/thens/comparisons, we need to use the Church encoding for booleans. Here, the idea is that *true* and *false* are functions which select a particular branch of two possibilities.

$$true_\lambda = \lambda a.\lambda b.a$$
$$false_\lambda = \lambda a.\lambda b.b$$

1. Define not_λ such that

$$not_\lambda\ true_\lambda \equiv false_\lambda$$
$$not_\lambda\ false_\lambda \equiv true_\lambda$$

2. Define if_λ such that

$$if_\lambda\ a\ b\ c$$

returns b if $a \equiv\ true_\lambda$ and returns c otherwise.

4 Coding Exercises

1. Translate the following code snippet written in Javascript to their equivalent in the λ-calculus:

```
let x = 1;
function g(z) {return x + z; }
function f(y) {
    let x = y + 1;
    return g(y * x);
}
f(3);    // in other words what would calling f on 3 look like?
```

2. A programmer is having difficulty debugging the following C program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, since extra space is required every time a function call is made.)

```
int f(int (*g)(...)) { /* g points to a function that returns an int */
    return g(g);
}
int main() {
```

```
    int x;
    x = f(f);
    printf("Value of x = %d\n", x);
    return 0;
}
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application *f(f)*. This program assumes that the type checker does not check the types of arguments to functions.

3. Functions map and reduce are standard functions from traditional functional programming that achieved broader recognition as a result of Googles MapReduce method for processing and generating large data sets. While map, reduce, and a number of related functions are provided in many JavaScript implementations, map and reduce can also be defined relatively simply in JavaScript as follows:

```
function map (f, inarray) {
    var out = [];
    for(var i = 0; i < inarray.length; i++) {
        out.push( f(inarray[i]) )
    }
    return out;
}

function reduce (f, inarray) {
    if(inarray.length <= 1) return;
    if(inarray.length == 2) return f(inarray[0],inarray[1]);
    r = inarray[0];
    for(var li = 1;li < inarray.length;li++){
        r = f(r, inarray[li]);
    }
     return r;
}
```

Function map(f, inarray) returns an array constructed by applying f to every element of inarray. Function reduce(f, inarray) applies the function f of two arguments to elements in the list, from left to right, until it reduces the list to a single element. For example:

```
js> map( function(x){return x+1}, [1,2,3,4,5])
 2,3,4,5,6
```

```
js> reduce(function(x,y){return x+y}, [1,2,3,4,5])
15
js> reduce(function(x,y){return x*y}, [1,2,3,4,5])
120
```

These functions can be combined in various ways. For each of the following questions, you may use a JavaScript implementation to test your answer yourself, but turn your solution in as part of a written description for manual grading.

(a) Explain how to use map and reduce to compute the sum of the first five squares, in one line. (The sum of the first three squares is $1 + 4 + 9$)

(b) Explain how to use map and reduce to count the number of positive numbers in an array of numbers.

(c) Explain how to use map and/or reduce to "flatten an array of arrays of numbers, such as $[[1, 2], [3, 4], [5, 6], [7, 8, 9]]$, to an array of numbers. (Hint: Look for built-in JavaScript concatenation functions.)