Assignment 1

The λ-Calculus and Combinators

All Parts Due: 11:59 PM, Monday, February 1, 2016

Preliminaries

You may work on this assignment with a partner (recommended), or by yourself. If you work with a partner, you must follow pair programming rules, even if what you’re writing is text or math, and share roles equally. Both members of the pair must fully understand all jointly submitted work. You are not required to work with your partner from lab. You may copy over work from lab, but only work where both members of the pair created essentially the same code in lab—ideas that only one member of a pair had must be recreated together, as a pair.

Files for this assignment are at https://svn.cs.hmc.edu/cs131/spring16/given/hw1. Copy them in the usual way (see CopyingAssignmentFiles on the wiki for a reminder of how).

Questions

The first few questions continue in the style of the lab, with you using lci. You must use the version on the CS systems, don’t attempt to download and use your own.

For all questions, you may define helper functions if it makes your code more readable.

1. If you have not done so already, finish the lab, including writing the function FACT described in the lab. Be sure to end up with code in hw1/hw1.lci and not lab1/lab1.lci.

2. For each of SUCC, PLUS, and TIMES, there is more than one way to write the function (i.e., a different technique to achieve the same result). You came up with one way in lab. Come up with an alternative way. (Your file should contain both SUCC and the alternative scheme, SUCC_ALT, and so on.)

   For example, for SUCC, you can add the added f on the outside, or on the inside.¹

3. Add a definition of the Y combinator to hw1.lci.

4. Use your newly added Y combinator to help you write a recursive function LENGTH, such that LENGTH l returns a Church numeral representing the length of the list.

5. For a more interesting function that uses recursion in an interesting way (a loop where the number of iterations is not known ahead of time or “obvious from the

¹By different, we mean that your approach should be tangibly different. It isn’t sufficient just to swap the roles of the two arguments (taking advantage of the commutative properties of addition and multiplication).
data”), consider the “3n + 1 function”. The idea is to start with a number and repeatedly divide by 2 (if the input is even) or multiply by 3 and add 1 (if the input is odd). Empirically, it seems that if we start at any positive integer and repeatedly applying this function we will eventually reach 1 (e.g., 6, 3, 10, 5, 16, 8, 4, 2, 1) but no one has yet proved that it always happens.

(a) Give a definition for \( \text{DIV2} \), which will divide any even number by two. (It’s great if it works on odd numbers too, but that’s not required.)

(b) Give a definition for \( \text{STEP} \), a function that takes one step in the 3n + 1 computation:

\[
\text{STEP} \ n \to^* \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n + 1 & \text{if } n \text{ is odd} \end{cases}
\]

As always, you can define helper functions as needed.

(c) Give a definition for \( \text{I3NP1} \), the function that takes a number as input and returns how many times \( \text{STEP} \) must be applied to it before we reach the answer 1. Thus, for example:

\[
\begin{align*}
\text{I3NP1} \ 1 & \to^* 0 \\
\text{I3NP1} \ 2 & \to^* 1 \\
\text{I3NP1} \ 3 & \to^* 7 \\
& \vdots \\
\end{align*}
\]

As a recursive function, you can of course use the \( \text{Y} \) combinator.

6. Someone has given you the following code (as \text{MYSTERY_EXPR} in hw1.1ci)

\[
S \ (S \ (S \ I \ (K \ (S \ I \ (K \ (S \ (S \ (K \ S) \ (S \ (K \ S) \ (S \ (K \ S) \ (S \ (K \ S) \ (S \ (K \ S) \ (S \ (K \ S) \ (S \ (K \ I) \ (S \ (K \ I)) \ (S \ (K \ I))))) \ (S \ (K \ (S \ (K \ K))) \ (S \ (K \ (S \ K)))))) \ (S \ (K \ (S \ I \ (K \ I)))) \ (S \ (K S) \ K))) \ (S \ (K S) \ K))) \ (S \ (K S) \ K)))
\]

and asked you to apply it to \( \text{MYSTERY_ARG_A...MYSTERY_ARG_F} \)

- \( K \ I \)
- \( I \)
- \( S \ (S \ (K \ S) \ K) \ I \)
- \( S \ (S \ (K \ S) \ K) \ (S \ (S \ (K \ S) \ K) \ I) \)
- \( S \ (S \ (K \ S) \ K) \ (S \ (S \ (K \ S) \ K) \ (S \ (S \ (K \ S) \ K) \ I)) \)
- \( S \ (S \ (K \ S) \ K) \ (S \ (S \ (K \ S) \ K) \ (S \ (S \ (K \ S) \ K) \ I)) \)

Unfortunately, you lack an S/K/I-reduction machine to run it on. All you have is 1ci.

(a) Why is this not a problem?

(b) Add suitable definitions to hw1.1ci to to allow you to run it. (Should be pretty trivial!)

(c) With readable set to on and then with it set to off look at the values of \( K \ I, I, S \ (S \ (K \ S) \ K) \ I \), etc. and “guess” what those values might mean. (Answer in the space provided in hw1.1ci.)
(d) Pass those values to the mystery expression. What do you conclude that the
mystery expression does? (Answer in the space provided in hw1.1c.i.)
(e) Evaluate MYSTERY_EXPR without any argument, and examine the result. What
can you conclude initially? Try the same with OTHER_EXPR. Explain why the
results are less satisfactory, and give a good rule of thumb for when the results
will be good.

7. Evaluation Order

When working with encodings in the untyped λ-calculus, we often have choices of
as to what (if anything) we should β-reduce next.

We mentioned some of those choices in class, saying that in most programming
languages, we have different expectations about the order in which things are com-
puted. For example, we don’t expect the code for a function or method to start
executing until (and unless) the function gets called!

In class we mention the leftmost-outermost and leftmost-innermost rules for choos-
ing what redex to reduce, but both of those schemes always reduce something. We
can model other orders of execution that sometimes leave expressions unreduced.
We can do so by defining restricted versions of β-reduction. Here are two possibil-
ities:

• **Call-by-name** can be defined by giving two rules:
  i. \((\lambda x.M)N \rightarrow_n M[x\rightarrow N]\). That is, if the whole expression is an anonymous
    function being applied to an argument, we can do the substitution.
  ii. If \(M \rightarrow_n M'\) then \(MN \rightarrow_n M' N\).

In other words, the only thing that allows you to apply rule (a) deeper in the
expression is rule (b). For example (where ♥ is some arbitrary lambda ex-
pression in normal form, such as a Church numeral, TRUE, etc.), we can con-
clude

\[
(\lambda f. f (\lambda w.w)) (\lambda x.x) \rightarrow_n (\lambda q.q) ♥
\]

by ii, because

\[
\lambda f. f (\lambda w.w) \rightarrow_n (\lambda x.x) \lambda w.w
\]

by i.

In fact, for any expression there is at most one next step that the call-by-name
rules permit, even though there may be many different spots where one could
apply β-reduction. The full series of call-by-name steps for this particular
example is:

\[
(\lambda f. f (\lambda w.w)) (\lambda x.x) \rightarrow_n (\lambda q.q) ♥
\]
The call-by-name rules give us only one choice for each step. Intuitively, you first find out what anonymous function is being applied, and then immediately plug in the argument.

- **Call-by-value.** This is defined by three rules, where we use \( V \) to stand for a “value”. Intuitively, a value is something that could be the result of evaluating some expression. In “real” programming languages values include integers, strings, etc., but in the pure untyped lambda calculus we will say that anonymous functions are our only values.

  i. \( (\lambda x. M) V \rightarrow_{\nu} M[x\mapsto V] \). That is, if the whole expression is an anonymous function being applied to a value, we can do the substitution.

  ii. If \( M_1 \rightarrow_{\nu} M_2 \) then \( M_1 N \rightarrow_{\nu} M_2 N \).

  iii. If \( M_1 \rightarrow_{\nu} M_2 \) then \( V M_1 \rightarrow_{\nu} V M_2 \).

Again, we always have at most one way to reduce an expression, but it may be different than in call-by-name. The same example under the call-by-value evaluation order would be:

\[
(\lambda f. \ f \ (\lambda w. w)) \ (\lambda x. x) \ (\lambda q. q) \ \uparrow
\rightarrow_n \ ((\lambda x. x) \ (\lambda w. w)) \ (\lambda q. q) \ \uparrow
\rightarrow_n \ (\lambda w. w) \ ((\lambda q. q) \ \uparrow)
\rightarrow_n \ (\lambda w. w) \ (\uparrow)
\rightarrow_n \ \uparrow
\]

The main difference from call-by-name is that when we reach \( (\lambda w. w) \ ((\lambda q. q) \ \uparrow) \), under we evaluate the argument \( (\lambda q. q) \ \uparrow \) before we can apply the identity function.

Intuitively, under call-by-value you first figure out which anonymous function is being applied, and then evaluate the argument, and only then do you plug this evaluated argument into the anonymous function.

Consider the following three expressions (which you saw previously in the reading for Monday’s class).

(a) \( (\lambda f. \ f \ f) \ ((\lambda x. x)(\lambda z. z)) \)

(b) \( (\lambda x. \ (x \ x)) \ (\lambda x. \ (x \ x)) \)

(c) \( (\lambda f. y)((\lambda x. \ (x \ x)) \ (\lambda x. \ (x \ x))) \)

For each, show the single reduction sequence one would get by following the call-by-name strategy, and the single reduction sequence one would get by following the call-by-value strategy. (Some of these sequences will be infinite, in which case show a suitable amount.)

Put your answers in `eval-order.txt`.

8. Run `svn commit` to commit your answers, thereby submitting your homework.