Lambda Calculus

I. Background and Importance
   A. [Church, '41] showed that a small set of semantic primitives excluding goto & assignment can
      form an adequate semantic basis for a language. I.e. lambda (\(\lambda\)) calculus is complete: it is capable
      of representing any computable function
   B. It gives us a mathematically clean starting point to examine real computer languages. It helps us
      to distinguish between:
      1. Necessary features
      2. Nice features (extras)
      3. Non-features (things to eliminate)
      4. Missing features
   C. An extended version of \(\lambda\) Calculus (w/type extensions) is the basis for modern functional lan-
      guages (Miranda, ML, Haskell)
   D. In \(\lambda\) Calculus functions are first-class objects (i.e. they can be used anywhere a primitive can be
      used).
      1. A fcn. can be a value of a variable, argument, or return type
      2. Anonymous fcns. can be created
      3. \(\lambda\) calculus formulas can be thought of either as programs or as the data upon which programs oper-
         ate
   E. It is different from traditional programming languages in that it has
      1. Lack of side effects
      2. No statement sequences
      3. Support for function objects
   F. Computation is the process of rewriting formulas, substituting arguments for parameters & possibly
      renaming variables

II. Notation
   A. Symbols used
      1. Has 3 constructs: variables, function application, function creation
      2. Notation: \(\lambda x. M\) is for a function with parameter \(x\) and body \(M\)
      e.g. \(\lambda x. x^2\) is a function mapping 5 to \(5 \times 5\)
      3. Functions are written next to their arguments
      e.g. \(f a\) is the application of function \(f\) to argument \(a\)
      e.g. \((\lambda x. x^2) 5\) is the function in parenthesis \((\lambda x. x^2)\) applied to 5. This whole expression is also
         called a term
   B. Grammar: \(M -> x | M_1 M_2 | (\lambda x. M)\)
      1. These production rules can be thought of as giving the terms:
         a. \(M -> x\) gives a variable \(x\)
         b. \(M -> M_1 M_2\) gives an application \((M N)\) of function \(M\) to \(N\)
         c. \(M -> (\lambda x. M)\) an abstraction
   C. Symbols used & conventions
      1. \(f, x, y, z\) used for variables (lower case)
      2. \(M, N, P, Q\) used for terms (upper case)
      3. A constant \(c\) can represent values like integers, operations on data, or structures like lists
         e.g. \(c\) can stand for: true, nil, t, head
D. Unnecessary parenthesis may be omitted
1. \((\lambda x.y) = \lambda x.y\) In the absence of parentheses, function application groups from left to right. Thus, \(xyz\) abbreviates \(((x y) z)\). The parentheses in \(x(yz)\) are necessary to ensure that \(x\) is applied to \((yz)\)
2. Parenthesis may change the meaning of a formula. e.g. \(\lambda x.yx, \lambda x.(yx), (\lambda x.y)x\) The parentheses in \(x(yz)\) are necessary to ensure that \(x\) is applied to \((yz)\)
3. The body of a lambda expression extends as far to the right as possible, and sequences associate to the left. Thus \(\lambda x.yx\) is really \((\lambda x.\!(yx))\) rather than \((\lambda x.\!(y)x)\) Another way of saying this is that function application (the stuff in parenthesis) has higher precedence than abstraction (the \(\lambda\))
4. A sequence of consecutive abstractions, as in \(\lambda x.\lambda y.\lambda z. M\) can be written with a single \(\lambda\), as in \(\lambda xyz. M\). Thus \(\lambda xy.x\) abbreviates \(\lambda x.\lambda y.x\)
5. Examples [F&G Fig. 4.12]

E. Bound and Free Variables
1. A symbol on the right side of a \(\lambda\) expression (an argument) is bound if it occurs as a parameter, immediately following the symbol \(\lambda\), on the left side of the same expression or of an enclosing expression
2. An occurrence of a variable \(x\) in \(F\) is free if \(x\) is not bound
3. Examples:
   a. \(p\) in \((\lambda y.py)\) is free, but \(y\) is bound to \(\lambda y\)
   b. In the formula \((x(\lambda x.((\lambda x.x)x)))\), the variable \(x\) appears 5 times. The second and third are bindings, the other three are uses. The first is free (not in the scope of an \(\lambda\)-expression), the fourth is bound to the third, and the fifth is bound to the second.
   c. The above is an illustration of scope

III. Reductions: We would like to freely replace an expression by a simpler expression that has the same meaning
A. Beta reductions: the action of calling a function on its argument. Redex short for "reducible expression." Examples:
1. \(((\lambda x.xy)(zw))\) reduces to \(((zw)y)\)
2. \(((\lambda x.(\lambda x.(xy)))(zz))\) reduces to \((zz)(\lambda x.(xy))\)
3. \((\lambda xyz.(xz) (yz)) \ (\lambda x. x) \ (\lambda x. x)\) First we rename bound variables:
   \((\lambda xyz.(xz) (yz)) \ (\lambda u. u) \ (\lambda v. v)\) Next we substitute \((\lambda u. u)\) in for \(x\):
   \((\lambda yz.((\lambda u. u)z) (yz)) \ (\lambda v. v)\) Next we substitute \(z\) in for \(u\):
   \((\lambda yz.(\lambda v. v)) \ (\lambda v. v)\) Next we substitute \((\lambda v. v)\) in for \(y\):
   \((\lambda z. ((\lambda v. v)z))\) Next we substitute \(z\) in for \(v\):
   \((\lambda z.zz)\)
B. Eta reduction: eliminate one level of binding in an expression of the form \(\lambda x.f(x)\). For any argument \(P\) (bound to \(x\)) this reduces to \(f(P)\), so \(\lambda x.f(x)\) can be rewritten simply as \(f\)
C. When all possible reduction steps have been done, the reduction process is complete and the formula is in normal form
D. Church-Rosser Theorem: for all pure \(\lambda\)-terms \(M, P,\) and \(Q\), if \(M \leadsto P\) and \(M \leadsto Q\), then there must exist a term \(R\) such that \(P \leadsto R\) and \(Q \leadsto R\). In other words, the result of a computation does not depend on the order in which reductions are applied.
E. Inner vs. Outer substitutions: To illustrate the above point, consider doing inner-most substitutions first vs. doing outermost substitutions first:

1. Innermost: $(\lambda y.(yy) (\lambda x.(xx) a)) \Rightarrow (\lambda y.(yy) (aa)) \Rightarrow ((aa)(aa))$
   This is otherwise known as call-by-value mechanism

2. Outermost: $(\lambda y.(yy) (\lambda x.(xx) a)) \Rightarrow ((\lambda x.(xx) a) (\lambda x.(xx) a)) \Rightarrow ((aa)(aa))$
   This is otherwise known as call-by-name mechanism

3. Of the two approaches given above, call-by-name (outermost) will find a normal form if there is one. This is a form of lazy evaluation, where only expressions used in the final solution need be evaluated

4. Consider a scenario where call-by-name terminates, but call-by-value doesn't for the expression: $(\lambda y.z (\lambda x.(xx) (\lambda x.(xx))))$
   a. Call by name gives the normal form $z$
   b. Call by value represents a nonterminating $\lambda$ expression

F. Renaming variables: When an expression containing an unbound symbol is used as an argument to another $\lambda$ expression, the following must be true: Any occurrence of a variable in the argument that was free before the substitution must remain free after the substitution. Examples:

1. $((\lambda x.(\lambda y.x)) (zy))$ does not reduce since $y$ is free in $zy$ but after substitution would not be free in $(\lambda y.(zy))$

2. To solve this, rename the parameter, giving: $((\lambda x.(\lambda w.x)) (zy))$ which would reduce to $(\lambda w.(zy))$

G. Currying: Functions of several variables can be simulated in the $\lambda$ calculus by repeated applications of functions with only a single variable each. E.g.

$(\lambda xy.x * y) 2 3 \Rightarrow (\lambda y.2 * y) 3 \Rightarrow 2 * 3$

IV. Modelling Computation

A. Church proved that $\lambda$-Calculus is complete. All we need are conditionals, numbers, & recursion to represent any computable function (compare to conditionals, primitive types, iteration, and sequence in procedural languages). We will show recursion first, then conditionals, and lastly representing numbers.

B. Conditionals: if-then & T, F values

1. See [Pratt, p. 418 ff.] for Modeling Boolean values

2. T & F can take the place of the conditional statement (IF B THEN S1 ELSE S2) in a programming language as follows:
   a. T (true) returns its first argument and discards the second, corresponding to the S1 in IF-THEN
   b. F (false) returns its second parameter, corresponding to the second part of the IF. This gives the ELSE, or S2
C. Integers (numbers)

1. We can develop the integers [Pratt, p. 419], where we consider c to be the "zero" element, and f to be the "successor" function applied to the c element. Integer N is written as the $\lambda$ expression (N a), which is $\lambda c.(a...(a c)...)$.

   Applying reduction to ((N a)b), we get (a...(a b)...), where we have N a's followed by a b.

2. Addition: Given the above notation, consider ((M a) ((N a)b)) by applying constant ((N a)b) to $\lambda$ expression (M a). For example:

   \[(N a) \Rightarrow \lambda c.(a...(a c)...),\] so \((2 a) \Rightarrow \lambda c.(a(a c)) \Rightarrow \lambda c.aac, \] and \((3 a) \Rightarrow \lambda c.aaac\]

   This means that substituting in for ((M a) ((N a)b)) gives:

   \[((3 a) ((2 a) b) \Rightarrow ((\lambda c.aaac) ((\lambda c.aac)b)) \Rightarrow ((\lambda c.aaac) (aab)) \Rightarrow aaaaab\]

   This implements addition, where 3 a's + 2 a's = 5 a's. More formally, this written as:

   \[[M + N] = \lambda a.\lambda b.((M a) ((N a)b))\]

   and we can write the + operator as:

   \[+ = \lambda M.\lambda N.\lambda a.\lambda b.((M a) ((N a)b))\]

   Multiplication: using the above notation, we can show that

   \[[M \times N] = \lambda a.(M(N a))\]

   which intuitively is N a's M times

3. Successor, IsZero

   a. Consider the integers where y is the "zero" element and x is potentially the "successor" function [F&G, Fig. 4.15]

   b. We can then show the Successor and ZeroP functions [F&G Fig. 4.16, 4.17] and a trace of their applications [F&G Fig. 4.18, 4.19, 4.20]
D. Recursion

1. This is trivially shown to be present in the lambda calculus as a function can both call another function and return a function. E.g. the non-terminating recursion in \( \lambda x. (x x) \) \( \lambda x. (x x) \), where the length of the formula neither grows nor shrinks.

2. Actually using it, however, is a different story.
Recursion in the lambda calculus can be done using fixed-point combinators. For instance, one such combinator is given by:

\[
Y = (\lambda f. (\lambda x.f(xx)) \ (\lambda x.f(xx)))
\]

Applying this to a function \( g \) then gives the reduction:

\[
Y g = (\lambda f. (\lambda x.f(xx)) \ (\lambda x.f(xx))) \ g
\]

1. Definition of \( Y \)
2. \( \beta \) reduction
3. \( \beta \) reduction
4. Substitution from 2.

The net effect of combinator \( Y \) is then that \( (Y g) = (g (Y g)) \). Multiple applications of "\( Y \ g \)" thus implement recursion. Assume we are given the factorial function:

\[
P = \text{if } (\text{isZero } n) \text{ then } 1 \text{ else } ( * n \ (g ( - n 1)))
\]

where \( g \) somehow calls \( P \) again. Let us also assume that we have the lambda calculus extensions for "isZero", 1, "+" and "-". Knowing that \( (\text{isZero } n) \) will return the boolean expression for \( T \) or \( F \) as discussed in class, we can rewrite this as:

\[
P = (\text{isZero } n) \ 1 \ ( * n \ (g ( - n 1)))
\]

3. Now we can combine the factorial function with the \( Y \)-combinator above to implement the factorial function in lambda-calculus notation and show its application.

Let us define Factorial as follows:

\[
\text{Factorial} = Y \ (\lambda g n . P)
\]

with \( Y \) and \( P \) given as above. Applying this to 0 gives:

\[
\text{Factorial} \ 0 = Y \ (\lambda g n . P) \ 0
\]

1. By definition
2. Applying \( Y \)
3. Substitution
4. Rewrite
5. Subst. expression for \( P \)
6. Subst. Factorial for \( g \)
7. Subst. 0 for \( n \)
8. isZero was true

We can use the expression derived for "Factorial 0" in the steps above to write the expression for "Factorial 1":

\[
\text{Factorial} \ 1 = Y \ (\lambda g n . P) \ 1
\]

1. By definition
2. Steps 2-6 above
3. 3.
4. Subst. 1 for \( n \)
5. isZero was false
6. Definition of "\( * \)"
7. Result from above ex.
8. Definition of "\( * \)"

We’ve eliminated some of the messiness of rewriting by simply putting "Factorial 0" rather than the full expansion.