Notes on the Lambda-Calculus
COMP 598 Winter 2015

Prakash Panangaden
School of Computer Science, McGill University

January 25, 2015

1 The λ-Calculus

The λ-calculus is a formalism for studying ways in which functions can be formed, combined and used for *computation.* Unlike set-theoretic accounts of functions, the λ-calculus takes an *intensional* view of functions[1] in that functions are "rules" for establishing a correspondence between an object, the argument to the function, and another object, the value or result. The process of using the "rule" to obtain the correspondence is called *function application.* In the λ-calculus the basic objects available are called λ-terms. The notation for λ-terms reflects the viewpoint alluded to above. In the subsequent discussion lowercase letters, $x, y, \ldots$ will represent variables in the λ-calculus while uppercase letters $M, N, \ldots$will (usually) represent meta-variables which stand for generic λ-terms. In the present discussion we shall try to understand λ-terms through rules for building and manipulating them. A model in which λ-calculus terms actually denote functions is quite hard to construct but the intuitive interpretation of λ-terms as functions should become clear.

We begin by assuming that there is a (countable) set of symbols called *variables.*

Definition 1.1. The set of λ-terms is defined inductively by the following clauses:

1. All variables are λ-terms
2. If M is a λ-term and x is a variable then so is $\lambda x.M$
3. If M and N are λ-terms then so is MN
4. If M is a λ-term then so is (M).

In a term of the form $\lambda x.M$ the x is the formal argument to the function and the M represents the *body* of the function. Note that the functions defined in the λ-calculus are all *one-place* functions. This is not really a restriction because functions which take several arguments

[1]Extensionality can be treated within the λ-calculus as an added axiom.

can be expressed in terms of functions of a single argument. This is done as follows: suppose f is a function of two arguments x and y, then we may think of f as a function that takes a single argument and *returns a function* that takes a single argument. Thus, for example, suppose f is the function *plus* of ordinary arithmetic. Normally, we think of *plus* as taking two arguments. We could think of *plus* as a function which takes as argument an integer n and returns a function, the latter function takes as argument an integer m and returns $n+m$. This generalizes in the obvious way to functions of several arguments. The process of viewing functions with several arguments as functions of a single argument is called *currying.* In the λ-calculus all functions are written in curried form.

Example 1.1. Here are some examples of simple λ-terms.

$$\begin{array}{l} xy \\ \lambda x.x \\ \lambda x.xx \\ \lambda x.\lambda y.xy \\ \lambda x.\lambda y.\lambda z.(xz)(yz) \\ \lambda x.y \\ x(\lambda x.xz)y \end{array}$$

A sequence of applications is assumed to associate to the left. Thus a term of the form $MNPQ$ represents $(((MN)P)Q)$. Often a sequence of λs is abbreviated with a single λ. Thus, for example, the λ-term $\lambda x.\lambda y.\lambda z.xyz$ is often written $\lambda xyz.xyz$. In parsing complicated λ-terms the following convention is used: If $\lambda x.M$ occurs within a larger expression we assume that M extends as far to the right as possible; i.e. until the first unmatched right parenthesis or the end of the expression whichever occurs first.

Example 1.2. The λ-term

$$\lambda x.\lambda y.yxx$$

contains $\lambda y.yxx$ as a sub-term whereas the λ-term

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

contains $\lambda y.yx$ as a sub-term but does not contain $\lambda y.yxx$ as a sub-term.

To use the λ-calculus as a *computational* formalism we need to describe what happens when a λ-term is applied to an argument. Roughly speaking, when the λ-term $\lambda x.M$ is applied to an argument N the occurrences of x in M are replaced by N. To make this statement more precise we need to define the concept of *free* and *bound* variables.

Definition 1.2. Free and bound variables are defined by the following rules:

1. In x, x occurs free, no other variables occur at all
2. If x occurs free in M then all these free occurrences of x are bound in $\lambda x.M$, if y is a variable different from x then all free occurrences of y in M remain free in $\lambda x.M$

3. Any variable which occurs free in M or N occurs free in MN.

A term which has no free variables is called a *closed term* or *combinator*. Note that a variable may occur both bound and free in a given λ-term. For example, in $\lambda x.y(\lambda y.xy)$ the first occurrence of y is free while the second occurrence of y is bound.

Bound variables are present to establish a correspondence between the argument to a function and positions within the body of a λ-term where the argument is to be substituted. The actual symbol used as a bound variable is irrelevant. This notion of bound variable closely corresponds to the notion of bound variable in the predicate calculus or to parameters appearing in function declarations in programming languages like Pascal. Typically, free variables only arise in sub-terms of some larger λ-term. The following diagram illustrates the concept of free and bound variables in a λ-term.

$$\lambda x. \underbrace{\lambda y. \overbrace{(\lambda z.xyz)}^{y\ free}\, y}_{y\ bound}$$

The process of *substitution* of terms for variables is the key computational mechanism of the λ-calculus. The notation is $M[x \mapsto N]$ and means "replace all *free* occurrences of x in M by N". The formal definition of substitution is:

Definition 1.3. Substitution is defined by the following clauses where the symbol $\equiv$ stands for *identity* of λ-terms.

1. $x[x \mapsto N] \equiv N$
2. $y[x \mapsto N] \equiv y$ where $x \not\equiv y$
3. $MM'[x \mapsto N] \equiv M[x \mapsto N]M'[x \mapsto N]$
4. $(\lambda x.M)[x \mapsto N] \equiv \lambda x.M$
5. $(\lambda y.M)[x \mapsto N] \equiv \lambda y.(M[x \mapsto N])$ where $x \not\equiv y$ and y does not appear free in N or x does not appear free in M
6. $(\lambda y.M)[x \mapsto N] \equiv \lambda z.(M[y \mapsto z])[x \mapsto N]$ where $x \not\equiv y$, z is a variable different from both x and y not occurring in either M or N, x does occur free in M and y does occur free in N.

The reason that the last clause is so complicated is that we need to be careful, when substituting a λ-term for a variable, that a free variable does not become a bound variable. This phenomenon is called *capture*. Consider the λ-term $\lambda y.x$, this is the function that returns x when applied to any λ-term. Now suppose we performed the substitution $(\lambda y.x)[x \mapsto w]$. If $w \not\equiv y$ clause five above would apply and we would get $\lambda y.w$ as expected. If, however, $w \equiv y$ and clause six, above, were *not* included in the definition of substitution, we would get $\lambda y.y$

if we attempted to use clause five. Using clause six we get $\lambda z.y$ which is exactly what we should expect.[2]

The trick of renaming a bound variable to avoid the capture of free variables is sufficiently useful to deserve its own formal definition:

Definition 1.4. A change of bound variables in a λ-term M is the replacement of a subterm of the form $\lambda x.N$ with x not bound in N by a term of the form $\lambda v.N[x \mapsto v]$ where v is a variable that does not occur (either free or bound) in N. A λ-term X is said to be *congruent* to another λ-term Y if Y is the result of applying a series of changes of bound variables to X.

Why should we demand that v does not occur either free *or bound* in N? Consider the following example. Suppose that we have the λ-term $\lambda y.\lambda v.vy$ and we wish to rename the bound variable y to v. Note that v only occurs bound in the body of the λ-term. If we carried out the renaming of y to v we would get the λ-term $\lambda v.\lambda v.vv$, now we can no longer tell that one of the vs should be bound to the outer λ and the other one should be bound to the inner λ. For all practical purposes congruent terms are regarded as being the same. We shall often say "identical" when we should be saying "congruent" if the difference is not important. A change of bound variables is often called *α-conversion* or *α-reduction*.

We are now ready to define the all important concept of *reduction*. This formalizes the process of "computing with λ-terms". The symbol $\Rightarrow$ is used to represent logical implication.

Definition 1.5. Reduction is a binary relation, written $\rightarrow$, between λ-terms defined by the following rules:

α $\lambda y.M \rightarrow \lambda v.(M[y \mapsto v])$ where v does not occur free or bound in M

β $(\lambda y.M)N \rightarrow M[y \mapsto N]$ (all the caveats about avoiding capture are hidden in the definition of substitution)

ρ $M \rightarrow M$

μ $M \rightarrow N \Rightarrow PM \rightarrow PN$

ν $M \rightarrow N \Rightarrow MP \rightarrow NP$

ξ $M \rightarrow N \Rightarrow \lambda x.M \rightarrow \lambda x.N$[3]

τ $M \rightarrow N$ and $N \rightarrow P \Rightarrow M \rightarrow P$

The rule β above is the rule with non-trivial computational content.

Example 1.3. Reducing a λ-term:

[2]Some authors (Church, for example) leave the last case undefined.

[3]Note that capture can occur while using this rule, indeed it should occur if this rule is to make any non-trivial statement.

$$\begin{aligned}&(\lambda x.\lambda y.\lambda z.x(yz))fg\\ &\to (\lambda y.\lambda z.f(yz))g\\ &\to \lambda z.f(gz)\end{aligned}$$

The original λ-term above performs function composition.

In any computational system the basic expectation is that the computational process proceeds by "simplifying" a term and producing another term of "equal" value. In our system, we have defined the computational process, namely reduction, but we have not defined a notion of equality. The relation of identity or congruence will not serve since it is not preserved by β-reduction. The relation of equality that we shall use is defined in terms of the reduction relation.

Definition 1.6. Equality of λ-terms, written $=$, is defined by replacing $\to$ by $=$ in the definition of reduction given above and adding the following rule:

$\sigma \;\; M = N \Rightarrow N = M$

Equality is often called convertibility. We shall say two λ-terms are equal if we can prove that they are equal using the above rules. It is important to understand the difference between equality and reduction. If M reduces to N then M is equal to N and by the rule σ N is equal to M. On the other hand N does *not* reduce to M. Reduction is thus one way while equality is symmetric.

Example 1.4. Consider the λ-terms $(\lambda x.(\lambda y.y)x)z$ and $(\lambda x.z)y$. Neither one reduces to the other but they are equal since each of them reduces to z.

The last example is also a little misleading. It is clearly not true that if two λ-terms are equal then one of them reduces to the other. But is it true that they must reduce to a common term? In fact the answer is "yes" but this is a hard theorem, it is certainly not what the definition says. What the definition does say is that if two terms, say M and N are equal then there is a sequence of terms $M_1, \ldots, M_k$ with $M_1 = M$, $M_k = N$ and with $M \to M_2$, $M_3 \to M_2$, $M_3 \to M_4$ and so on. In other words there is a "zig-zag" of reductions and reversed reductions which interpolates between the two terms.

How do we establish that two terms are *not* equal? We do not yet have the machinery to do this properly. However, we can temporarily adopt the following approach. First, note that we would like our theory to have more than one function definable in it. Now if we are trying to establish that two λ-terms M and N are not equal we can try to show that by assuming $M = N$ we can apply the rules for deducing equality and conclude that all λ-terms are equal. To illustrate this approach, consider the λ-terms $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$. We shall call them T and F respectively. What happens if we assume $T = F$? The following proof shows that any pair of λ-terms, M, N can be proven equal.

$T = F$	assumption
$TM = FM$	using ν
$TMN = FMN$	using ν

$TMN = M$	using β
$FMN = N$	using β
$M = N$	using τ and σ appropriately.

Thus we could "prove" two terms unequal if we can show that assuming that they were equal led to the equation $T = F$.

So far we have shown how the steps of a computation in the λ-calculus proceed. Normally we think of computation as proceeding until a "result" is obtained. What is the result of a computation in the λ-calculus? Put another way, how do we know when to stop the reduction process. This latter question has a straightforward answer, "stop when no more reduction is possible". These considerations lead us to single out a special class of λ-terms which serve as the results of computations in the λ-calculus.

Definition 1.7. A term of the form $(\lambda x.M)N$ is called a *redex* and $M[x \mapsto N]$ is called its *contractum.*

Definition 1.8. A term is said to be in *normal form* if it contains no redices. If $M \to N$ and N is in normal form then N is said to be a normal form of M and M is said to *normalize* to N.

Notice the definition of redex refers to the possibility of β-reduction only; clearly we could perform α-reductions indefinitely if we wished. The appearance of a normal form in a sequence of reductions signals the end of our computation.

The definition of normal form raises some interesting questions. Does every term have a normal form? Clearly not, consider $(\lambda x.xx)\lambda x.xx$. There is exactly one opportunity for β-reduction. If we carry out this reduction we get exactly the same term back and hence again have an opportunity for β-reduction. Terms without normal forms are the λ-calculus analogues of non-terminating programs. If a λ-term does have a normal form does any choice of reductions lead to that normal form? Again the answer is "no" as the following example illustrates.

Example 1.5. Let W stand for the λ-term $\lambda x.xxx$. It is easy to see that $WW \to WWW \to WWWW \ldots$. Let I stand for the λ-term $\lambda x.x$, the identity function. Let F be the λ-term introduced in the previous example. Now consider $F(WW)I$. There are two opportunities for β-reduction. We could do the leftmost reduction first getting I as the result immediately. We could choose to reduce the sub-term WW first, getting $F(WWW)I$, and then reduce by applying F giving I again. Thus there are infinitely many sequences of reductions starting from $F(WW)I$ and ending in I. There is also the (infinite) sequence $F(WW)I \to F(WWW)I \to F(WWWW)I \ldots$ which never reaches normal form.

How do we know that we will find the normal form if there is one? This is a fairly subtle question and I will not try to justify the answer formally. If you always perform the leftmost reduction then you will find the normal form if one exists. Thus in the example above, the leftmost reduction possible involves applying F, whenever we did this we immediately reached the normal form. Intuitively, the reason this works is that a term which does have a normal form may have a sub-term which does not have a normal form. However, the sub-term which does not have a normal form may never be needed so we should apply the "outermost" function to see if a particular sub-term is actually needed before trying to reduce potentially non-terminating subterms. This particular reduction strategy is called *normal order reduction* and is the λ-calculus analogue of "call-by-need" evaluation.[4] There is of course no way of looking at a generic λ-term and deciding whether a normal form does exist (why not?).

[4]If you have never heard of this before ignore this remark for now.

2 Computing with λ-Calculus

So far we have seen the λ-calculus as a purely formal system. Two pressing questions remain, how do we compute with it? and why would we want to? The answer to the second question is simple; the λ-calculus has very few constructs and it is thus very easy to prove theorems *about it.* Furthermore, the λ-calculus, though simple, is rich enough to express all the computable functions. This means that *all* programming languages that we may be interested in can be expressed in terms of the λ-calculus.

The simplicity of the λ-calculus is precisely what makes it awkward to work with as a programming language. The answer to the first question will therefore involve a fairly lengthy demonstration of how to encode ordinary computational constructs in the λ-calculus. Recall that the λ-calculus is a formalism where computation is expressed via the successive application of the reduction rules. Thus expressing computations involves taking terms that mimic familiar constructs and reducing them to normal form. The rest of this section is devoted to examining a variety of special λ-terms and seeing how they express familiar computational constructs.

The pure λ-calculus provides only one mechanism for making "programs" and "data"[5] interact, namely application. Quite frequently, however, we would like to have a part of a program execute only if certain conditions are met, in short we would like to have a *conditional* construct. What do conditionals look like? First, we would like to have a notion of "boolean" expression which can evaluate to "true" or "false". Second we would like a construct that takes a boolean expression and two other expressions as arguments, evaluates the boolean expression and depending on whether the result is true or false evaluates either the first or the second expression. The combinators[6] T and F introduced previously perform this function very handily. Their definitions are $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ respectively. Now consider the λ-term BMN where B is some λ-term that reduces to either T or F. If B evaluates to T then BMN reduces to M, if B evaluates to F then BMN reduces to N. Thus BMN can be viewed as encoding the construct "if B is true then reduce M otherwise if B is false reduce N".

What else would we like in a programming language? We would certainly like to have some kind of "data structure". This is achieved by the combinator D defined by $\lambda y.\lambda z.\lambda x.xyz$. This forms "pairs" when applied to two terms; $DMN \rightarrow \lambda x.xMN$. For this to really qualify as a data structure we need to have a means of recovering the original terms M and N from the pair. This is provided by the λ-terms $\underline{first} \equiv \lambda w.wT$ and $\underline{second} \equiv \lambda w.wF$. It is easy to see by applying the rule for β-reduction that $\underline{first}(DMN) \rightarrow M$ and similarly for $\underline{second}$.

We now have the skeleton of a reasonable programming language but we would still like to operate on familiar data like the integers. To do this we must *choose* a sequence of λ-terms to represent the integers and we must represent familiar arithmetic operations on these λ-terms. There are many choices possible. We shall use a representation due

[5]Note that both programs and data are represented by λ-terms.

[6]A combinator is a λ-term with no free variables

to Church. Other systems for encoding the integers are due to Curry, Barendregt, Scott and Wadsworth. We shall use the following notational convention. Actual *numbers* will be written $n, m, \ldots$ whereas the λ-terms used to represent the numbers will written with an underline thus $\underline{n}, \underline{m}, \ldots$. The basic idea is to represent n by a λ-term which takes two arguments and applies the first argument to the second argument n times. In symbols, n is represented by the λ-term $\lambda f.\lambda x.\underbrace{f(f(\ldots f}_{n\ times}x)\ldots)$. In particular, 0 is represented by $\lambda f.\lambda x.x$, 1 is represented by $\lambda f.\lambda x.fx$ and 2 is represented by the λ-term $\lambda f.\lambda x.(f(fx))$. Note that the associativity is the opposite to what we would have if we left out the parentheses in the definitions. These λ-terms are called the *Church numerals* or simply *numerals.*

As an exercise in λ-gymnastics let us argue that if $n \neq m$ then $\underline{n} \neq \underline{m}$. Without loss of generality we may assume that $n < m$. Now consider the λ-term $Q \equiv DN_0(DN_1(\ldots DN_mM)\ldots)$. This is a nested sequence of paired terms which we could write more picturesquely (but informally!) as $[N_0, [N_1, [\ldots [N_m, M\ \underbrace{]\ldots]}_{m+1\ times}$. Now consider the λ-term $\underline{first}(\underline{n}\ \underline{second}\ Q)$. The λ-term $\underline{n}$ causes $\underline{second}$ to be applied n times to Q thus stripping off the first n applications of the pairing construct. The outermost $\underline{first}$ then picks out the first member of the remaining outermost pair, in other words N_n. Similarly we can show that $\underline{first}(\underline{m}\ \underline{second}\ Q)$ reduces to N_m. If we assume that $\underline{n} = \underline{m}$ where n and m are different it follows that $\underline{first}(\underline{m}\ \underline{second}\ Q) = \underline{first}(\underline{n}\ \underline{second}\ Q)$ (why?). But, since n and m are different we can easily construct Q so that N_n and N_m are any two terms we choose. Thus we have shown that any two λ-terms can be proven equal if we assume that any two *distinct* Church numerals are equal.

Now we are ready to do some basic arithmetic within the λ-calculus. The very first thing we would like to have is the successor function. To simplify the notation let us agree that f^nx shall mean f applied to x n times. Thus, for example, the Church numeral for n can be written as $\lambda f.\lambda x.f^nx$. The successor function is just $\lambda y.\lambda f.\lambda x.yf(fx)$. This is easy to check directly as the following little calculation shows.

$$\begin{array}{ll} & (\lambda y.\lambda f.\lambda x.yf(fx))(\lambda g.\lambda u.g^nu) \\ \to & \lambda f.\lambda x.(\lambda g.\lambda u.g^nu)f(fx) \\ \to & \lambda f.\lambda x.(\lambda u.f^nu)(fx) \\ \to & \lambda f.\lambda x.f^n(fx) \\ \to & \lambda f.\lambda x.f^{n+1}x \end{array}$$

The predecessor function can also be encoded but this is a rather complicated exercise. The other basic arithmetic functions can be encoded fairly easily once we recognise that the Church numeral for n applied to a pair of arguments applies the first argument n times to the second argument. The function for *plus* is the λ-term $\lambda u.\lambda v.\lambda f.\lambda x.uf(vfx)$. This can also be checked directly. To simplify the notation further we shall write $\lambda xyz.\ldots$ instead of $\lambda x.\lambda y.\lambda z.\ldots$.

$$\begin{array}{lll} & (\lambda uvfx.uf(vfx))\underline{n}\underline{m} & \\ \to & \lambda fx.\underline{n}f(\underline{m}fx) & \textit{two steps} \\ \to & \lambda fx.\underline{n}f(f^m x) & \\ \to & \lambda fx.f^n(f^m x) & \\ \to & \lambda fx.f^{n+m}x & \textit{this is } \underline{n+m} \end{array}$$

It is now an easy exercise to see that multiplication of two Church numerals is defined by the λ-term $\lambda uvfx.u(vf)x$ and that exponentiation is defined by $\lambda uvfx.uvfx$.

If we want to write more complicated "programs" involving the Church numerals we need to be able to test for whether a particular numeral represents zero or not. In other words we should have a combinator that returns T if the argument to it is $\underline{0}$ and F if the argument to it is a numeral different from zero. This is achieved by the following combinator $\lambda v.v(\lambda u.F)T$.

To complete the discussion of numbers in the λ-calculus we shall look at the predecessor function. This function takes a Church numeral as argument and returns the preceding Church numeral *if the original Church numeral is not* $\underline{0}$ and returns $\underline{0}$ if the original Church numeral is $\underline{0}$. A λ-term which does this for the Church numerals is

$$\lambda x.\underline{second}(x(\lambda y.D\ \underline{succ}((\underline{first}\ y)(\underline{second}\ y)))(D(\lambda z.\underline{0})\underline{0}))$$

Fixed Point Combinators

We need a way of expressing recursion within the λ-calculus. Informally, we would express recursive definitions via equations of the form $f = \ldots f \ldots$ In the λ-calculus such an equation does not define a term, the equation merely states a condition that the term must satisfy. How do we know that we will find any term satisfying the equation? Fortunately it is easy to find a term satisfying any recursive equation. Furthermore, such terms can be found by simply applying a particular combinator to a term constructed from the given recursive equation. Such combinators are called *fixed point combinators*.

Let us look at the above equation scheme more closely. We are trying to "solve" an equation of the form $f = M$ where M is a term containing free occurrences of f within it. This equation can be rewritten $f = (\lambda x.M[f \mapsto x])f$ or $f = M'f$ where $M' = (\lambda x.M[f \mapsto x])$. Thus we are looking for a fixed point of M'. A fixed point combinator does exactly this.

Definition 2.1. A *fixed point combinator* is a term R such that $\forall F, RF = F(RF)$.

Example 2.1. The most commonly used fixed point combinator is

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

often written just Y. The following calculation shows that Y is a fixed point combinator.

Let A stand for $\lambda x.F(xx)$

$$\begin{array}{rll} & YF & \\ = & (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F & \\ \to & AA & \\ = & (\lambda x.F(xx))A & \textit{using the def of } A \\ \to & F(AA) & \\ = & F(YF) & \end{array}$$

It is important to note that YF does not β-reduce to $F(YF)$. We have merely proven equality of these two terms. A fixed point combinator that when applied to a λ-term actually β-reduces to the fixed point of the λ-term is $\Theta\Theta$ where $\Theta = \lambda xy.y(xxy)$.

$$\begin{array}{rl} & \Theta\Theta F \\ \equiv & (\lambda xy.y(xxy))\Theta F \\ \to & F(\Theta\Theta F) \end{array}$$

With the aid of a fixed point combinator it is easy to solve recursive equations.

Example 2.2. Suppose we wish to solve the equation $Fxy = FyxF$. It is easy to see that we can solve this equation if $F = \lambda xy.FyxF$. To solve the latter equation we need a fixed point of $\lambda fxy.fyxf$. Using *a* fixed point combinator, for example Y, we obtain $F = Y(\lambda fxy.fyxf)$.

We can define recursively defined numeric functions in the λ-calculus using the combinators defined so far.

Example 2.3. The factorial function satisfies the equation

$$fact = \lambda x.(\underline{iszero}\ x)(\underline{1})(\underline{mult}\ \underline{n}\ (fact(\underline{pred}\ \underline{n}))).$$

If we define H to be the λ-term

$$\lambda f.\lambda x.(\underline{iszero}\ x)(\underline{1})(\underline{mult}\ \underline{n}\ (f(\underline{pred}\ \underline{n})))$$

then the λ-term for $fact$ is just the fixed point of H. So we can write YH for $fact$.

We have not formally proven that all the computable functions are definable in the λ-calculus but we have provided a fairly rich collection of combinators which provide many of the facilities which one normally sees in programming languages. In fact it can be formally shown that all the partial recursive functions can be defined in the λ-calculus. An even more remarkable fact, due to Curry, is that the two combinators K and S, defined by $\lambda xy.x$ and $\lambda xyz.(xz)(yz)$ respectively, suffice to express *all* the combinators.