Notes on Lambda Calculus

Foundations of Computer Science

March 31, 2016

λ-terms. A λ-term is either:

- A variable x, y, z, ...
- $\lambda x.M$ where x is a variable and M a λ-term
- $M\,N$ where M, N are λ-terms
- (M) where M is a λ-term

Examples: x $\quad \lambda x.x \quad \lambda y.(\lambda x.x) \quad \lambda x.x\,(\lambda y.y)$

Intuitively, $\lambda x.M$ represents a function with parameter x and returning M, while $M\,N$ represents an application of function M to argument N. The simplification rules below will enforce this interpretation.

Just like elsewhere in mathematics, parentheses can be used freely to group terms together to affect or just clarify the order of applications. Application $M\,N$ is a binary operation that associates to the left, so that writing $M\,N\,P$ is the same as writing $(M\,N)\,P$. If you want $M\,(N\,P)$ (which means something different) then you need to use parentheses explicitly. Similarly, in a λ, the body of the lambda (the M part) intuitively goes as far to the right as possible. Thus, $\lambda x.x\,y$ is interpreted as $\lambda x.(x\,y)$, as opposed to $(\lambda x.x)\,y$. Similarly, $\lambda x.\lambda y.\lambda z.z\,y\,x$ is interpreted as $\lambda x.(\lambda y.(\lambda z.z\,x\,y))$. In these notes, I will generally try to use parentheses explicitly.

Intuitively, the *scope* of λx in $(\lambda x.M)$ is all of M. An occurrence of a variable is said to be *bound* if it occurs in the scope of a λ. More precisely, it is bound to the nearest enclosing λ. An occurrence of a variable is said to be *free* if it is not bound.

Examples: y is free in $(\lambda x.y)$; the first occurrence of x is free in $(\lambda y.x\,(\lambda x.x))$ while the second is not; z is bound in $(\lambda z.(\lambda x.z))$

Bound variables can be renamed without affecting the meaning of term. Intuitively, $\lambda x.x$ and $\lambda y.y$ represent the same function, the identity function. That we happen to call the parameter x in the first and y in the second is pretty irrelevant. Two terms are α-equivalent when they are equal up to renaming of some bound variables. Thus, $\lambda x.(x\,z)$ and $\lambda y.(y\,z)$ are α-equivalent. Be careful that your renaming does not *capture* a free occurrence of a variable. For example, $\lambda x.(x\,z)$ and $\lambda z.(z\,z)$ are *not* α-equivalent. They represent different functions.

We will generally identify α-equivalent terms.

Substitution. An important operation is that of substituting a term for a variable inside another term, $M[x \leftarrow N]$. It is defined formally as

$$x[x \leftarrow N] = N$$

$$y[x \leftarrow N] = y \quad \text{if } x \neq y$$

$$(M_1\,M_2)[x \leftarrow N] = M_1[x \leftarrow N]\,M_2[x \leftarrow N]$$

$$(\lambda y.M)[x \leftarrow N] = \lambda y.(M[x \leftarrow N]) \quad \text{if } y \text{ is not free in } N$$

If the last case, if $x = y$ or if y *is* free in N, we can always find a term $\lambda z.M'$ that is α-equivalent to $\lambda y.M$ and such that $x \neq z$ and z is not free in N to perform the sustitution.

(Because we avoid capturing free variables, this form of substitution is called a *capture-avoiding substitution.*)

Simplification Rules. The main simplification rule we use is *beta reduction*:[1]

$$(\lambda x.M)\,N = M[x \leftarrow N]$$

A term of the form $(\lambda x.M)\,N$ is called a *redex*.

Simplification can occur within the context of a larger term, of course:

$$M\,P = N\,P \quad \text{if } M = N$$

$$P\,M = P\,N \quad \text{if } M = N$$

$$(\lambda x.M) = (\lambda x.N) \quad \text{if } M = N$$

Examples:

$$\begin{aligned}(\lambda x.x)\,(\lambda y.y) &= x[x \leftarrow \lambda y.y] \\ &= \lambda y.y\end{aligned}$$

$$((\lambda x.(\lambda y.x))\,z_1)\,z_2 = (\lambda y.x)[x \leftarrow z_1]\,z_2$$

[1]The λ-calculus is usually presented using a reduction relation. For simplicity here, I'm choosing an equational presentation to match the kind of intuitions you developed in algebra.

$$
\begin{aligned}
&= (\lambda y.z_1)\, z_2 \\
&= z_1[y \leftarrow z_2] \\
&= z_1
\end{aligned}
$$

$$
\begin{aligned}
((\lambda x.(\lambda y.y))\,(\lambda z.z))\,(\lambda x.(\lambda y.x)) &= (\lambda y.y)[x \leftarrow \lambda z.z]\,(\lambda x.(\lambda y.x)) \\
&= (\lambda y.y)\,(\lambda x.(\lambda y.x)) \\
&= y[y \leftarrow \lambda x.(\lambda y.x)] \\
&= \lambda x.(\lambda y.x)
\end{aligned}
$$

From now on, I will skip the explicit substitution step when showing simplifications.

A λ-term is in *normal form* if it cannot be simplified further.

Not every λ-term can be simplified to a normal form:

$$
\begin{aligned}
(\lambda x.x\,x)\,(\lambda x.x\,x) &= (\lambda x.x\,x)\,(\lambda x.x\,x) \\
&= (\lambda x.x\,x)\,(\lambda x.x\,x) \\
&= \ \ldots
\end{aligned}
$$

There can be more than one redex in a λ-term, meaning that there may be more than one applicable simplification. For instance, $((\lambda x.x)\,(\lambda y.x))\,((\lambda x.(\lambda y.x))\,z_1\,z_2)$. A property of the λ-calculus is that all the ways to simplify a term down to a normal form yield the same normal form (up to renaming of bound variables). This is called the *Church-Rosser property*. It says that the order in which we perform simplifications to reach a normal form is not important.

In practice, one often imposes an order in which to apply simplifications to avoid nondeterminisn. The *normal-order strategy*, which always simplifies the leftmost and outermost redex, is guaranteed to find a normal form if one exists.

Encoding Booleans. Even though the λ-calculus only has variables and functions, we can encode traditional data types within in.

Boolean values encoding (à la Church):

$$
\begin{aligned}
\textbf{true} &= \lambda x.(\lambda y.x) \\
\textbf{false} &= \lambda x.(\lambda y.y)
\end{aligned}
$$

In what sense are these encodings of Boolean values? Booleans are useful because they allow you to select one branch or the other of a conditional expression.

$$
\textbf{if} = \lambda c.(\lambda x.(\lambda y.c\,x\,y))
$$

The trick is that when B simplifies to either **true** or **false**, then $\mathbf{if}\,B\,M\,N$ simplifies either to M or to N, respectively:

If $B = \mathbf{true}$, then

$$\begin{aligned}\mathbf{if}\,B\,M\,N = {} & B\,M\,N \\ = {} & \mathbf{true}\,M\,N \\ = {} & (\lambda x.(\lambda y.x))\,M\,N \\ = {} & (\lambda y.M)\,N \\ = {} & M\end{aligned}$$

while if $B = \mathbf{false}$, then

$$\begin{aligned}\mathbf{if}\,B\,M\,N = {} & B\,M\,N \\ = {} & \mathbf{false}\,M\,N \\ = {} & (\lambda x.(\lambda y.y))\,M\,N \\ = {} & (\lambda y.y)\,N \\ = {} & N\end{aligned}$$

Of course, these show that **if** is not strictly necessary. You should convince yourself that $\mathbf{true}\,M\,N = M$ and that $\mathbf{false}\,M\,N = N$.

We can define logical operators:

$$\begin{aligned}\mathbf{and} &= \lambda m.(\lambda n.m\,n\,m) \\ \mathbf{or} &= \lambda m.(\lambda n.m\,m\,n) \\ \mathbf{not} &= \lambda m.(\lambda x.(\lambda y.m\,y\,x))\end{aligned}$$

Thus:

$$\begin{aligned}\mathbf{and}\,\mathbf{true}\,\mathbf{false} = {} & (\lambda m.(\lambda n.m\,n\,m))\,\mathbf{true}\,\mathbf{false} \\ = {} & (\lambda n.\mathbf{true}\,n\,\mathbf{true})\,\mathbf{false} \\ = {} & \mathbf{true}\,\mathbf{false}\,\mathbf{true} \\ = {} & (\lambda x.(\lambda y.x))\,\mathbf{false}\,\mathbf{true} \\ = {} & (\lambda y.\mathbf{false})\,\mathbf{true} \\ = {} & \mathbf{false}\end{aligned}$$

$$\begin{aligned}\mathbf{not}\,\mathbf{false} = {} & (\lambda m.(\lambda x.(\lambda y.m\,y\,x)))\,\mathbf{false} \\ = {} & (\lambda x.(\lambda y.\mathbf{false}\,y\,x)) \\ = {} & (\lambda x.(\lambda y.(\lambda u.\lambda v.v)\,y\,x)) \\ = {} & (\lambda x.(\lambda y.(\lambda v.v)\,x)) \\ = {} & (\lambda x.(\lambda y.x)) \\ = {} & \mathbf{true}\end{aligned}$$

Encoding Natural Numbers.

$$\begin{aligned}
\mathbf{0} &= \lambda f.(\lambda x.x)\\
\mathbf{1} &= \lambda f.(\lambda x.f\,x)\\
\mathbf{2} &= \lambda f.(\lambda x.f\,(f\,x))\\
\mathbf{3} &= \lambda f.(\lambda x.f\,(f\,(f\,x)))\\
\mathbf{4} &= \quad \ldots
\end{aligned}$$

In general, natural number n is encoded as $(\lambda f.(\lambda x.f^n\,x))$.

Successor operation:

$$\mathbf{succ} = \lambda n.(\lambda f.(\lambda x.(n\,f)\,(f\,x)))$$

$$\begin{aligned}
\mathbf{succ}\,\mathbf{1} = &\quad (\lambda n.(\lambda f.(\lambda x.(n\,f)\,(f\,x))))\,(\lambda f.(\lambda x.f\,x))\\
= &\quad (\lambda f.(\lambda x.((\lambda f.(\lambda x.f\,x))\,f)\,(f\,x)))\\
= &\quad (\lambda f.(\lambda x.(\lambda x.f\,x)\,(f\,x)))\\
= &\quad (\lambda f.(\lambda x.f\,(f\,x)))\\
= &\quad \mathbf{2}
\end{aligned}$$

Other operations:

$$\begin{aligned}
\mathbf{plus} &= \lambda m.(\lambda n.(\lambda f.(\lambda x.(m\,f)\,(n\,f\,x))))\\
\mathbf{times} &= \lambda m.(\lambda n.(\lambda f.(\lambda x.m\,(n\,f)\,x)))\\
\mathbf{iszero}? &= \lambda n.n\,(\lambda x.\mathbf{false})\,\mathbf{true}
\end{aligned}$$

$$\begin{aligned}
\mathbf{plus}\,\mathbf{1}\,\mathbf{2} = &\quad (\lambda m.(\lambda n.(\lambda f.(\lambda x.(m\,f)\,(n\,f\,x)))))\,\mathbf{1}\,\mathbf{2}\\
= &\quad (\lambda n.(\lambda f.(\lambda x.(\mathbf{1}\,f)\,(n\,f\,x))))\,\mathbf{2}\\
= &\quad (\lambda f.(\lambda x.(\mathbf{1}\,f)\,(\mathbf{2}\,f\,x)))\\
= &\quad (\lambda f.(\lambda x.((\lambda f.(\lambda x.f\,x))\,f)\,((\lambda f.(\lambda x.f\,(f\,x)))\,f\,x)))\\
= &\quad (\lambda f.(\lambda x.(\lambda x.f\,x)\,((\lambda f.(\lambda x.f\,(f\,x))\,f\,x))))\\
= &\quad (\lambda f.(\lambda x.f\,((\lambda f.(\lambda x.f\,(f\,x))\,f\,x))))\\
= &\quad (\lambda f.(\lambda x.f\,((\lambda x.f\,(f\,x))\,x)))\\
= &\quad (\lambda f.(\lambda x.f\,(f\,(f\,x))))\\
= &\quad \mathbf{3}
\end{aligned}$$

$$\mathbf{times}\,\mathbf{2}\,\mathbf{3} = \quad (\lambda m.(\lambda n.(\lambda f.(\lambda x.m\,(n\,f)\,x))))\,\mathbf{2}\,\mathbf{3}$$

$$\begin{aligned}
&= (\lambda n.(\lambda f.(\lambda x.\mathbf{2}\,(n\,f)\,x)))\,\mathbf{3}\\
&= (\lambda f.(\lambda x.\mathbf{2}\,(\mathbf{3}\,f)\,x))\\
&= (\lambda f.(\lambda x.(\lambda f.(\lambda x.f\,(f\,x)))\,((\lambda f.(\lambda x.f\,(f\,(f\,x))))\,f)\,x))\\
&= (\lambda f.(\lambda x.(\lambda f.(\lambda x.f\,(f\,x)))\,(\lambda x.f\,(f\,(f\,x)))\,x))\\
&= (\lambda f.(\lambda x.(\lambda x.(\lambda x.f\,(f\,(f\,x)))\,((\lambda x.f\,(f\,(f\,x)))\,x))\,x))\\
&= (\lambda f.(\lambda x.(\lambda x.(\lambda x.f\,(f\,(f\,x)))\,(f\,(f\,(f\,x))))\,x))\\
&= (\lambda f.(\lambda x.(\lambda x.f\,(f\,(f\,(f\,(f\,(f\,x))))))\,x))\\
&= (\lambda f.(\lambda x.f\,(f\,(f\,(f\,(f\,(f\,x)))))))\\
&= \mathbf{6}
\end{aligned}$$

$$\begin{aligned}
\mathbf{iszero?}\,\mathbf{0} &= (\lambda n.n\,(\lambda x.\mathbf{false})\,\mathbf{true})\,(\lambda f.(\lambda x.x))\\
&= (\lambda f.(\lambda x.x))\,(\lambda x.\mathbf{false})\,\mathbf{true}\\
&= (\lambda x.x)\,\mathbf{true}\\
&= \mathbf{true}
\end{aligned}$$

$$\begin{aligned}
\mathbf{iszero?}\,\mathbf{2} &= (\lambda n.n\,(\lambda x.\mathbf{false})\,\mathbf{true})\,(\lambda f.(\lambda x.f\,(f\,x)))\\
&= (\lambda f.(\lambda x.f\,(f\,x)))\,(\lambda x.\mathbf{false})\,\mathbf{true}\\
&= (\lambda x.(\lambda x.\mathbf{false})\,((\lambda x.\mathbf{false})\,x))\,\mathbf{true}\\
&= (\lambda x.\mathbf{false})\,((\lambda x.\mathbf{false})\,\mathbf{true})\\
&= \mathbf{false}
\end{aligned}$$

More difficult is defining a predecessor function, taking a nonzero natural number n and returning $n-1$. There are several ways of defining such a function; here is probably the simplest:

$$\mathbf{pred} = \lambda n.(\lambda f.(\lambda x.n\,(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x)\,(\lambda u.u)))$$

$$\begin{aligned}
\mathbf{pred}\,\mathbf{2} &= (\lambda n.(\lambda f.(\lambda x.n\,(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x)\,(\lambda u.u))))\,(\lambda f.(\lambda x.f\,(f\,x)))\\
&= (\lambda f.(\lambda x.(\lambda f.(\lambda x.f\,(f\,x)))\,(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x)\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda x.(\lambda g.(\lambda h.h\,(g\,f)))\,((\lambda g.(\lambda h.h\,(g\,f)))\,x))\,(\lambda u.x)\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda g.(\lambda h.h\,(g\,f)))\,((\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x))\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda h.h\,((\lambda u.x)\,f))\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda h.h\,x)\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda h.h\,((\lambda h.h\,x)\,f))\,(\lambda u.u)))\\
&= (\lambda f.(\lambda x.(\lambda h.h\,(f\,x))\,(\lambda u.u)))
\end{aligned}$$

$$
\begin{aligned}
&= (\lambda f.(\lambda x.(\lambda u.u)\,(f\,x))) \\
&= (\lambda f.(\lambda x.f\,x)) \\
&= \mathbf{1}
\end{aligned}
$$

Note that **pred 0** is just **0**:

$$
\begin{aligned}
\mathbf{pred}\,\mathbf{0} &= (\lambda n.(\lambda f.(\lambda x.n\,(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x)\,(\lambda u.u))))\,(\lambda f.(\lambda x.x)) \\
&= (\lambda f.(\lambda x.(\lambda f.(\lambda x.x))\,(\lambda g.(\lambda h.h\,(g\,f)))\,(\lambda u.x)\,(\lambda u.u))) \\
&= (\lambda f.(\lambda x.(\lambda x.x)\,(\lambda u.x)\,(\lambda u.u))) \\
&= (\lambda f.(\lambda x.(\lambda u.x)\,(\lambda u.u))) \\
&= (\lambda f.(\lambda x.x)) \\
&= \mathbf{0}
\end{aligned}
$$

Recursion. With conditionals and basic data types, we are very close to having a Turing-complete programming language (that is, one that can simulate Turing machines). All that is missing is a way to do loops. It turns out we can write recursive functions in the λ-calculus, which is sufficient to give us loops.

Consider factorial. Intuitively, we would like to define **fact** by

$$
\mathbf{fact} = \lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(\mathbf{fact}\,(\mathbf{pred}\,n)))
$$

but this is not a valid definition, since the right-hand side refers to the term being defined. It is really an *equation*, the same way $x = 3x$ is an equation. Consider that equation, $x = 3x$. Define $F(x) = 3x$. Then, a solution of $x = 3x$ is really a fixed-point of F, namely, a value x_0 for which $F(x_0) = x_0$. And F has only one fixed-point, namely $x_0 = 0$, which gives us the one solution to $x = 3x$, namely $x = 0$.

Similarly, if we define

$$
F_{fact} = \lambda f.(\lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(f\,(\mathbf{pred}\,n))))
$$

then we see that the definition that we're looking for is a fixed-point of F_{fact}, namely, a term $\mathbf{f}$ such that $F_{fact}\,\mathbf{f} = \mathbf{f}$. Indeed, if we have such a term, then:

$$
\begin{aligned}
\mathbf{f}\,\mathbf{3} &= F_{fact}\,\mathbf{f}\,\mathbf{3} \\
&= (\lambda f.(\lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(f\,(\mathbf{pred}\,n)))))\,\mathbf{f}\,\mathbf{3} \\
&= (\lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(\mathbf{f}\,(\mathbf{pred}\,n))))\,\mathbf{3} \\
&= (\mathbf{iszero?}\,\mathbf{3})\,\mathbf{1}\,(\mathbf{times}\,\mathbf{3}\,(\mathbf{f}\,(\mathbf{pred}\,\mathbf{3}))) \\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{f}\,(\mathbf{pred}\,\mathbf{3})) \\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{f}\,\mathbf{2}) \\
&= \mathbf{times}\,\mathbf{3}\,(F_{fact}\,\mathbf{f}\,\mathbf{2})
\end{aligned}
$$

$$\begin{aligned}
&= \mathbf{times\,3\,(times\,2\,(f\,1))}\\
&= \mathbf{times\,3\,(times\,2\,}(F_{fact}\,\mathbf{f\,1}))\\
&= \mathbf{times\,3\,(times\,2\,(times\,1\,(f\,1)))}\\
&= \mathbf{times\,3\,(times\,2\,(times\,1\,}(F_{fact}\,\mathbf{f\,1})))\\
&= \mathbf{times\,3\,(times\,2\,(times\,1\,1))}\\
&= \mathbf{6}
\end{aligned}$$

(I coalesce together quite a few simplification steps in the above, for the sake of space.)

Thus, what we need is a way to find fixed-points in the λ-calculus. The following function does just that:

$$Y = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

YF gives us a fixed-point of F:

First, note that

$$\begin{aligned}
YF = &\quad (\lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)))\,F\\
= &\quad (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))\\
= &\quad F\,((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)))
\end{aligned}$$

And therefore

$$\begin{aligned}
YF = &\quad F\,((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)))\\
= &\quad F\,(F\,((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))))\\
= &\quad F\,(YF)
\end{aligned}$$

So indeed, $(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))$ is a fixed-point of F.

We can use Y to define our factorial function:

$$\mathbf{fact} = Y\,F_{fact}$$

By the above derivation, we know that

$$\begin{aligned}
&\mathbf{fact} = (\lambda x.F_{fact}\,(x\,x))\,(\lambda x.F_{fact}\,(x\,x))\\
&\mathbf{fact} = F_{fact}\,\mathbf{fact}
\end{aligned}$$

and thus:

$$\begin{aligned}
\mathbf{fact\,3} = &\quad Y\,F_{fact}\,\mathbf{3}\\
= &\quad (\lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)))\,F_{fact}\,\mathbf{3}\\
= &\quad (\lambda x.F_{fact}\,(x\,x))\,(\lambda x.F_{fact}\,(x\,x))\,\mathbf{3}\\
= &\quad F_{fact}\,((\lambda x.F_{fact}\,(x\,x))\,(\lambda x.F_{fact}\,(x\,x)))\,\mathbf{3}
\end{aligned}$$

$$
\begin{aligned}
&= F_{fact}\,\mathbf{fact}\,\mathbf{3}\\
&= (\lambda f.(\lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(f\,(\mathbf{pred}\,n)))))\,\mathbf{fact}\,\mathbf{3}\\
&= (\lambda n.(\mathbf{iszero?}\,n)\,\mathbf{1}\,(\mathbf{times}\,n\,(\mathbf{fact}\,(\mathbf{pred}\,n))))\,\mathbf{3}\\
&= (\mathbf{iszero?}\,\mathbf{3})\,\mathbf{1}\,(\mathbf{times}\,\mathbf{3}\,(\mathbf{fact}\,(\mathbf{pred}\,\mathbf{3})))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{fact}\,(\mathbf{pred}\,\mathbf{3}))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{fact}\,\mathbf{2})\\
&= \mathbf{times}\,\mathbf{3}\,(F_{fact}\,\mathbf{fact}\,\mathbf{2})\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{times}\,\mathbf{2}\,(\mathbf{fact}\,\mathbf{1}))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{times}\,\mathbf{2}\,(F_{fact}\,\mathbf{fact}\,\mathbf{1}))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{times}\,\mathbf{2}\,(\mathbf{times}\,\mathbf{1}\,(\mathbf{fact}\,\mathbf{1})))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{times}\,\mathbf{2}\,(\mathbf{times}\,\mathbf{1}\,(F_{fact}\,\mathbf{fact}\,\mathbf{1})))\\
&= \mathbf{times}\,\mathbf{3}\,(\mathbf{times}\,\mathbf{2}\,(\mathbf{times}\,\mathbf{1}\,\mathbf{1}))\\
&= \mathbf{6}
\end{aligned}
$$