Programming Language Theory
Lecture notes

5 Lambda calculus

Lambda calculus is a formal system based on a notation for functions that was introduced by Church in 1930. The objective was to make a general theory of functions and to provide a formal basis for logic and mathematics.

The first goal has established a rich and beautiful theory which has inspired the construction of functional programming languages such as Lisp and Haskell and has been used to define formal semantics for programming languages.

The objects of the theory are *terms* which may be interpreted both as functions defined by computational rules and values which are the arguments of the functions. There is no distinction between functions and values and a function may even be applied to itself.

There are terms called *combinators* in the λ-calculus with a certain property which can describe computations without variables. The corresponding theory is called *combinatory logic* having Schönfinkel and Curry as originators.

Lambda notation

A standard way to define a function in mathematics is to give it a name and give the value of the function as an expression in the formal arguments the function. A simple example is the successor function, $f(x) = x+1$. Using λ-*notation* we can express this function as a λ-*expression* without giving it a name. We write $\lambda x \,.\, x + 1$. Just as f may be applied to the argument 1, $f(1)$, we may apply the lambda expression to the same argument, $(\lambda x \,.\, x + 1)(1)$. It is customary to omit the parentheses around the argument when it is a simple constant or variable, so we write $(\lambda x \,.\, x + 1)\, 1$.

We evaluate both $f(1)$ and $(\lambda x \,.\, x + 1)\, 1$ by substituting 1 for x in $x + 1$ and performing the addition.

In mathematics a function of two variables may be defined like $f(x, y) = x + y$. With λ-notation we would write $\lambda(x, y) \,.\, x + y$ to denote the same function. There is, however, another way to regard the addition function, not usually done in mathematics. Using lambda notation we could write $\lambda x \,.\, \lambda y \,.\, x + y$ or more clearly $\lambda x \,.\, (\lambda y \,.\, x + y)$. Applying this λ-expression to 1 yields $\lambda y \,.\, 1 + y$ after substituting 1 for x. So the result of the application is a function, the successor function. Starting with $((\lambda x \,.\, \lambda y \,.\, x + y)\, 1)\, 2$ we get $(\lambda y \,.\, 2 + y)\, 1$ and $2 + 1$ after two substitutions.

In mathematics there are also functions returning functions as values and having function arguments. They are usually called *operators* or *functionals*. A well known example is the differentiation operator $\frac{d}{dx}x^2 = 2x$. Another one is composition of functions. If f and g are functions, e.g. from R to R, $h = f \circ g$ is defined by $h(x) = f(g(x))$. The differentiation and composition operators are functions according to the standard set theoretic definition of a function as a set of pairs.

Grammar for the lambda calculus

The objects of the λ-calculus are called λ-terms. There are three kinds of λ-terms. The simplest one is an *identifier*. The form of an identifier is not very important. The theory needs an unbounded supply of identifiers, but in our examples we will usually manage with a small number of them. We could allow Java identifiers, but it is customary to use single letter identifiers, possibly with an index. Examples: x, y, z, x_0 and x_1.

The second kind is an *abstraction*. An abstraction can be thought of as a function with a formal parameter and a function body which is a λ-term. A simple example is the *identity* abstraction $\lambda x\,.\,x$. When an abstraction is part of a more complex λ-term we will enclose it in parentheses, $(\lambda x\,.\,x)$.

The third kind is an *application*. It can be thought of as an application of a function to its argument, but any λ-term may be applied to any λ-term, so $x\ y$ is a legal application as is $(\lambda x\,.\,x)\,y$. When an application appears inside a complex term it may be surrounded by parentheses, $((\lambda x\,.\,x)\,y)$.

The first concrete grammar requires a parentheses pair around each abstraction and each application. Using Backus-Naur formalism we write

$$term \rightarrow id \mid (\lambda id\,.\,term) \mid (term\ term)$$

where a *term* is defined to be an identifier, *id*, an abstraction with an identifier and a *term* or an application with two *term*s.

Some examples of λ-terms:

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

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

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

We observe that there are no numbers or arithmetic operators in the λ-calculus. The calculus can be used to denote and evaluate numerical functions, but the numbers will be represented by λ-terms and the arithmetic operators will be defined by λ-terms.

When we state and prove theorems about λ-calculus we need names standing for arbitrary terms and identifiers. We will use capitals in the middle of the alphabet, sometimes with an index, to range over λ-terms and small Greek letters, possibly indexed, to range over identifiers. Thus there are three kinds of λ-terms:

An *identifier* σ,

an *abstraction* $(\lambda\alpha\,.\,M)$,

and an *application* $(M_1\ M_2)$,

A Haskell data type for representing λ-terms is

```
data Lambda = Id String | Abstr String Lambda | Appl Lambda Lambda
```

In Java we would define one abstract class `Lambda` with three subclasses.

Nielson would give the following abstract grammar

$$M ::= x \mid \lambda x\,.\,M \mid M_1\ M_2$$

Simplified syntax

A complex λ-term will have a lot of parentheses. Parentheses is not very reader friendly. One usually introduces a relaxed syntax for λ-terms where parentheses may be omitted using two associativity rules.

A sequence of applications is assumed to associates to the left so that

$$M_1\ M_2\ M_3\ \ldots\ M_n \quad \triangleq \quad (\ldots((M_1\ M_2)\ M_3)\ \ldots\ M_n)$$

We use the symbol $\triangleq$ when we make definitions.

The abstraction dot, on the other hand, associates to the right so that

$$\lambda\sigma_1\,.\,\lambda\sigma_2\,.\ldots.\,\lambda\sigma_n\,.\,M \quad \triangleq \quad (\lambda\sigma_1.(\lambda\sigma_2\,.\,(\ldots\ (\lambda\sigma_n\,.\,M)\ldots)))$$

These conventions is expressed in the following concrete grammar.

$$\begin{array}{l} term \ \rightarrow\ factor\ factor^* \\ factor\ \rightarrow\ id \mid \lambda id\,.\,term \mid (term) \end{array}$$

The grammar is ambiguous; there are, for example, two derivation trees for $\lambda x\,.\,x\,x$, corresponding to $\lambda x\,.\,(x\,x)$ or $(\lambda x\,.\,x)\,x$. The first one should be chosen; the *term* in the abstraction should extend as far as possible.

An even shorter notation with the same meaning will be used

$$\lambda\sigma_1\ \sigma_2\ \ldots\ \sigma_n\,.\,M \quad \triangleq \quad (\lambda\sigma_1.(\lambda\sigma_2\,.\,(\ldots\ (\lambda\sigma_n\,.\,M)\ldots)))$$

Free and bound identifiers

In $f(x) = x+y$ the identifier x is a formal parameter while y is a identifier whose value is defined elsewhere. In λ-calculus we will call x a *bound* identifier and y a *free* identifier. We define the set of free identifiers in M inductively.

$$\begin{array}{rcl} \mathcal{F}(\sigma) & \triangleq & \{\,\sigma\,\} \\ \mathcal{F}(\lambda\sigma\,.\,M) & \triangleq & \mathcal{F}(M) \setminus \{\,\sigma\,\} \\ \mathcal{F}(MN) & \triangleq & \mathcal{F}(M) \cup \mathcal{F}(N) \end{array}$$

The set of bound identifiers is defined likewise.

$$\begin{array}{rcl} \mathcal{B}(\sigma) & \triangleq & \emptyset \\ \mathcal{B}(\lambda\sigma\,.\,M) & \triangleq & \mathcal{B}(M) \cup \{\,\sigma\,\} \\ \mathcal{B}(MN) & \triangleq & \mathcal{B}(M) \cup \mathcal{B}(N) \end{array}$$

A identifier may occur both free an bound in a λ-term, but each occurrence of an identifier is either free or bound. In $(\lambda x\,.\,x)\ x$ the first two occurrences of x is bound while the last is free. We define the set of all identifiers occurring in a λ-term;

$$\mathcal{I}(M) \quad \triangleq \quad \mathcal{F}(M) \cup \mathcal{B}(M)$$

A λ-term without free identifiers is called a *combinator*. Some important combinators have standard names:

$$\begin{aligned} I &\triangleq \lambda x\,.\,x \\ K &\triangleq \lambda x\,.\,\lambda y\,.\,x \\ K_* &\triangleq \lambda x\,.\,\lambda y\,.\,y \\ S &\triangleq \lambda x\,.\,\lambda y\,.\,\lambda z\,.\,xz(yz) \end{aligned}$$

Conversions

There are two kinds of conversions that may be performed on λ-terms, α- and β-conversions. α-conversion means renaming all bound occurrences of an identifier in an abstraction. β-conversion corresponds to applying a function to an argument and represents one step in the evaluation of a function. The conversions are essentially performed by substitutions. There are however some complications with name collisions.

The basic operation is *substitution*. First we define substitution in λ-terms. We use $M_1[M/\sigma]$ to denote the result after substituting the term M for the identifier σ in the term M_1. We define it over the structure of M.

$$\begin{aligned} \sigma[M/\sigma] &\triangleq M \\ \tau[M/\sigma] &\triangleq \tau \\ &\quad \text{if } \tau \text{ and } \sigma \text{ are different identifiers} \\ (M_1M_2)[M/\sigma] &\triangleq (M_1[M/\sigma])(M_2[M/\sigma]) \end{aligned}$$

Substitution in abstractions are more complicated since with a naive substitution a free occurrence of an identifier in N may become bound in the abstraction as in $(\lambda x\,.\,y)[x/y]$.

$$\begin{aligned} (\lambda\sigma\,.\,M)[N/\sigma] &\triangleq (\lambda\sigma\,.\,M) \\ (\lambda\tau\,.\,M)[N/\sigma] &\triangleq (\lambda\tau\,.\,(M[N/\sigma])) \\ &\quad \text{if } \tau \neq \sigma \wedge \tau \notin \mathcal{F}(N) \\ (\lambda\tau\,.\,M)[N/\sigma] &\triangleq (\lambda\upsilon\,.\,((M[\upsilon/\tau])[N/\sigma])) \\ &\quad \text{if } \tau \neq \sigma \wedge \tau \in \mathcal{F}(N) \wedge \upsilon \notin \mathcal{I}(M) \cup \mathcal{I}(N) \cup \{\sigma\} \end{aligned}$$

In the last line υ is a new identifier not occurring in M or N.

We can formally define the conversions with axioms and inference rules. We start with α-conversion.

$$\lambda\sigma\,.\,M \longrightarrow_\alpha \lambda\tau\,.\,M[\tau/\sigma] \text{ provided that } \tau \notin \mathcal{F}(M)$$

$$\frac{M \longrightarrow_\alpha M'}{M\,N \longrightarrow_\alpha M'\,N} \qquad \frac{N \longrightarrow_\alpha N'}{M\,N \longrightarrow_\alpha M\,N'}$$

$$\frac{M \longrightarrow_\alpha M'}{\lambda\sigma\,.\,M \longrightarrow_\alpha \lambda\sigma\,.\,M'}$$

We write $M =_\alpha N$ if there is a sequence $M = M_0 \longrightarrow_\alpha M_1 \longrightarrow_\alpha \cdots \longrightarrow_\alpha M_k = N$ for some $k \geq 0$. $=_\alpha$ is an *equivalence relation*, i.e.

1. $M =_\alpha M$, reflexivity
2. if $M =_\alpha N$ then $N =_\alpha M$, symmetry
3. if $M =_\alpha N$ and $N =_\alpha L$ then $M =_\alpha L$, transitivity

We define $\longrightarrow_\beta$ similarly

$$(\lambda\sigma\,.\,M)N \longrightarrow_\beta M[N/\sigma]$$

$$\frac{M \longrightarrow_\beta M'}{M\,N \longrightarrow_\beta M'\,N} \qquad \frac{N \longrightarrow_\beta N'}{M\,N \longrightarrow_\beta M\,N'}$$

$$\frac{M \longrightarrow_\beta M'}{\lambda\sigma\,.\,M \longrightarrow_\beta \lambda\sigma\,.\,M'}$$

We write $M \twoheadrightarrow_\beta N$ if there is a sequence $M \equiv M_0 \longrightarrow_\beta M_1 \longrightarrow_\beta \cdots \longrightarrow_\beta M_k \equiv N$ for some $k \geq 0$. The relation $\twoheadrightarrow_\beta$ is not an equivalence relation, why?

We write $M =_\beta N$ if $M \twoheadrightarrow_\beta N$ or $N \twoheadrightarrow_\beta M$. The relation $=_\beta$ is an equivalence relation.

Normal form

Let M be a λ-term. If there is no λ-term N such that $M \longrightarrow_\beta N$ we say that M is a *normal form*. If a λ-term is not a normal form then it must have a *sub term* that has the form $((\lambda\alpha\,.\,M)N)$. Such a sub term will be called a *redex*.

The term $\lambda x\,.\,(x\,y)$ is a normal form and $((\lambda x\,.\,x)y)$ may be reduced to y which is a normal form. We say that a λ-term, M, can be *reduced to normal form* if there is a normal form N such that $M \twoheadrightarrow_\beta N$.

Some terms cannot be reduced to normal form:

$$(\lambda x\,.\,(x\,x))(\lambda x\,.\,(x\,x)) \quad \longrightarrow_\beta \quad (x\,x)[(\lambda x\,.\,(x\,x))/x] = (\lambda x\,.\,(x\,x))(\lambda x\,.\,(x\,x))$$

Reductions may even make a term more complicated:

$$(\lambda x\,.\,(x\,x\,x))(\lambda x\,.\,(x\,x\,x)) \quad \longrightarrow_\beta \quad (\lambda x\,.\,(x\,x\,x))(\lambda x\,.\,(x\,x\,x))(\lambda x\,.\,(x\,x\,x))$$

Further reductions will make it even worse.

A term may often be reduced in several ways. If the middle application in

$$(\lambda xy\,.\,y)((\lambda x\,.\,xxx)(\lambda x\,.\,xxx))z$$

is reduced repeatedly we will never reach a normal form while reduction from the left will produce z after two reductions.

The normal form would not deserve its name if a term could be reduced to two essentially different normal forms. Church and Rosser have proved that this is not the case.

> **Theorem** [**Church-Rosser**]. If $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta N'$ where N are N' normal forms then $N =_\alpha N'$. ■

The original proof required several pages. A recent proof by Per Martin-Lf is substantially shorter. The proof consists of several proofs by structural induction. We will omit the proof.

A stronger version of the theorem can be used when the terms cannot be reduced to normal form.

> **Theorem** [**Diamond property**]. If $M \twoheadrightarrow_\beta M_0$ and $M \twoheadrightarrow_\beta M_1$ then there are terms, $M_0' =_\alpha M_1'$, such that $M_0 \twoheadrightarrow_\beta M_0'$ and $M_1 \twoheadrightarrow_\beta M_1'$. ■

Henceforth we will write $M = N$ if and only if there are λ-terms, $M' =_\alpha N'$, such that $M \twoheadrightarrow_\beta M'$ and $N \twoheadrightarrow_\beta N'$.

Two reduction orders are of special interest: *normal order* and *applicative order*. Normal order means that in each step the leftmost redex is reduced. An example:

$$\begin{array}{rcl} (\lambda x\,.\,xx)((\lambda y\,.\,y)(\lambda z\,.\,z)) & \longrightarrow_\beta & ((\lambda y\,.\,y)(\lambda z\,.\,z))((\lambda y\,.\,y)(\lambda z\,.\,z)) \quad \longrightarrow_\beta \\ (\lambda z\,.\,z)((\lambda y\,.\,y)(\lambda z\,.\,z)) & \longrightarrow_\beta & (\lambda y\,.\,y)(\lambda z\,.\,z) \quad \longrightarrow_\beta \quad \lambda z\,.\,z \end{array}$$

Using applicative order we compute the argument in an application before reducing the application. The argument may contain applications with arguments which have to be reduced recursively. An example

$$(\lambda x\,.\,xx)((\lambda y\,.\,y)(\lambda z\,.\,z)) \longrightarrow_\beta (\lambda x\,.\,xx)(\lambda z\,.\,z) \longrightarrow_\beta (\lambda z\,.\,z)(\lambda z\,.\,z) \longrightarrow_\beta \lambda z\,.\,z$$

Church and Rosser also showed that if a term may be reduced to normal form this can be done by using normal order reductions. We have seen that applicative order may fail to terminate when normal order reduction succeeds.

Fixed points

Given a mapping $f \in R \to R$ we say that x is a *fixed point* of f if f maps x on itself. The fixed points are solutions to the equation $f(x) = x$. The function $f(x) = x^2 - 2$ has two fixed points, -1 and 2. In same way we say that a λ-term is a fixed point of a combinator F if $FX = X$. The following theorem shows that any combinator has a fixed point and provides a method to find it.

> **Theorem** [**Fixed point**]. For any combinator F there is a combinator X such that $FX = X$. Furthermore, there is a combinator, $Y \stackrel{\Delta}{=} \lambda f.(\lambda x\,.\,f(xx))(\lambda x\,.\,f(xx))$, such that YF is a fixed point of F, i.e. $YF = F(YF)$. ■

Proof. Let $W \stackrel{\Delta}{=} \lambda x\,.\,F(xx)$ and $X \stackrel{\Delta}{=} WW$. Then

$$X = WW = (\lambda x\,.\,F(xx))W = F(WW) = FX$$

Further

$$YF \;=\; (\lambda x\,.\,F(xx))(\lambda x\,.\,F(xx)) = WW = X$$

□

The *fixed point combinator* Y can be used to mimic evaluation of functions defined by recursion in the λ-calculus. We present an informal example using λ-notation with natural numbers, arithmetic operators and conditional expressions. The factorial function is defined by

$$fac(n) \stackrel{\Delta}{=} \texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * fac(n-1);$$

Using λ-notation we can write

$$\begin{aligned} fac &= \lambda n\,.\,\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * fac(n-1) \\ &= (\lambda f\,.\,(\lambda n\,.\,\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * f(n-1)))fac = F\ fac \end{aligned}$$

where

$$F \stackrel{\Delta}{=} \lambda f n\,.\,(\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * f(n-1))$$

Now fac is a fixed point F. Let us investigate if YF is a fixed that represents the factorial function. Let us check that $fac\ 3$ can be reduced to 6.

$$\begin{aligned} fac\ 3 = (YF)3 = F(YF)3 &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * ((YF)2) = 3 * ((YF)2) \\ (YF)2 = F(YF)2 &= \texttt{if}\ \ 2 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 2 * ((YF)1) = 2 * ((YF)1 \\ (YF)1 = F(YF)1 &= \texttt{if}\ \ 1 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 1 * ((YF)1) = 1 * ((YF)0 \\ (YF)0 = F(YF)0 &= \texttt{if}\ \ 0 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 0 * ((YF)0) = 1 \end{aligned}$$

In the following sections we shall represent natural numbers by λ-terms and define combinators that can be used to add and multiply numbers represented in this way. This will make this example fully formal.

Booleans and conditional expressions

Boolean values will be used in conditional expression to choose between alternative expressions. It is convenient to use representations which make this choice simple. Thus we define

$$\begin{aligned} T &\stackrel{\Delta}{=} K = \lambda x\,.\,\lambda y\,.\,x \\ F &\stackrel{\Delta}{=} K_* = \lambda x\,.\,\lambda y\,.\,y \end{aligned}$$

A conditional expression $\texttt{if}\ \ B\ \ \texttt{then}\ \ P\ \ \texttt{else}\ \ Q$ may now be represented by BPQ where B is a term that either reduces to T or F. We get

$$\begin{aligned} T\ PQ &= KPQ = (\lambda x\,.\,\lambda y\,.\,x)PQ = P \\ F\ PQ &= K_*PQ = (\lambda x\,.\,\lambda y\,.\,y)PQ = Q \end{aligned}$$

The logical terms T and F can be applied to two terms and select the first or the second.

Pairs and natural numbers

We can represent an ordered pair as a λ-term:

$$[M, N] \triangleq \lambda z.zMN$$

and extract the components with T and F.

$$[M, N]T = M$$
$$[M, N]F = N$$

We use pairs to represent natural numbers in a way that makes the successor function simple. We will use $\ulcorner n \urcorner$ as a name of the λ-term which represents the natural number n and define $\ulcorner n \urcorner$ by induction:

$$\ulcorner 0 \urcorner \triangleq I = \lambda x \,.\, x$$
$$\ulcorner n + 1 \urcorner \triangleq [F, \ulcorner n \urcorner]$$

There are simple combinators that represents the successor function $suc\ n = n + 1$, the predecessor function $pred\ n = n - 1$ and the predicate $iszero\ n = (n = 0)$:

$$Suc \triangleq \lambda x \,.\, [F, x]$$
$$Pred \triangleq \lambda x \,.\, x\ F$$
$$IsZero \triangleq \lambda x \,.\, x\ T$$

We leave it as an exercise to show that these terms are adequate representations.

We proceed as we did for the factorial function to get a representation for addition.

$$add\ n\ m = \texttt{if}\ \ m = 0\ \ \texttt{then}\ \ n\ \ \texttt{else}\ \ suc(add\ n\ (pred\ m));$$

Translation to λ-calculus

$$Add \triangleq \lambda n \,.\, \lambda m \,.\, IsZero\ m\ n\ (Suc(Add\ n\ (Pred\ m)))$$

This is rewritten as a fixed point equation

$$Add = F\ Add$$

where

$$F = \lambda f \,.\, \lambda n \,.\, \lambda m \,.\, IsZero\ m\ n\ (Suc(f\ n\ (Pred\ m)))$$

with the solution $Add = YF$.

In a similar way we may define a combinator Mul such that $Mul\ \ulcorner m \urcorner\ \ulcorner n \urcorner = \ulcorner m * n \urcorner$. With these in hand a factorial combinator may be constructed.

Problems

1. Rewrite the following terms with all parentheses required by the formal grammar.
   (a) $\lambda xy.yx$
   (b) $\lambda xyz.xz(yz)$
   (c) $w(\lambda xyz.xz(yz))uv$
2. Rewrite the following terms according to the simplified syntax.
   (a) $(\lambda x\,.\,(\lambda y.(x(xy))))$
   (b) $(\lambda f.(\lambda n.(\lambda m.(((z\ m)\ n)(s((a\ n)(f\ m)))))))$
3. Evaluate
   (a) $(\lambda y\,.\,x(\lambda x\,.\,x))[\lambda y\,.\,xy/x]$
   (b) $(y(\lambda v\,.\,xv))[\lambda y\,.\,vy/x]$
4. Reduce $(\lambda x\,.\,xxx)((\lambda yz.y)I(SS))$ to normal form, where I and S are defined in the lecture notes.
5. Compute $(\lambda yz.zy)((\lambda x\,.\,xxx)(\lambda x\,.\,xxx))(\lambda w.I)$.
6. Compute $SSSSSS$. (This is rather long.)
7. Let
$$A \triangleq \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$
$$B \triangleq A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A\,A$$
   Show that B is a fixed point combinator, i.e. $BF = F(BF)$ for any F.
8. Use the idea from the previous problem to define another simpler fixed point combinator.
9. Rewrite $\ulcorner 1 \urcorner$, $\ulcorner 2 \urcorner$ och $\ulcorner 3 \urcorner$ using I, F and pairs, [,].
10. Verify that T and F can be used to extract the components of a pair.
11. Verify that Suc, $Pred$ and $IsZero$ has the following properties.
$$Suc\ \ulcorner n \urcorner = \ulcorner n+1 \urcorner$$
$$Pred\ \ulcorner n+1 \urcorner = \ulcorner n \urcorner$$
$$IsZero\ \ulcorner 0 \urcorner = T$$
$$IsZero\ \ulcorner n+1 \urcorner = F$$
12. Show that $Add\ \ulcorner 1 \urcorner\ \ulcorner 1 \urcorner = \ulcorner 2 \urcorner$, where $Add = YF$ as in the lecture notes.
13. Find a combinator which has itself as a fixed point, i.e. $F\ F = F$.
14. Find a combinator M such that $M\ I\ S\ S = M$. Hint: Rewrite the equation as a fixed point equation.
15. It can be shown that all combinators can be expressed by S and K alone. Show that $SKK = I$.