Lecture Notes on
Recursion in the λ-Calculus

15-814: Types and Programming Languages
Jan Hoffmann and Frank Pfenning

Lecture 3
Tuesday, September 3, 2024

1 Introduction

In this lecture, we first conclude our study of computation in the λ-Calculus by discussing how we can express primitive recursion and general recursion. In the next lecture, we formalize the definitions that we introduced informally using the concept of *rule induction*.

2 Primitive Recursion

It is easy to define very fast-growing functions by iteration, such as the exponential function, or the "stack" function iterating the exponential.

$$\begin{array}{lcl} \mathit{exp} & \triangleq & \lambda b.\, \lambda e.\, e\ (\mathit{times}\ b)\ (\mathit{succ}\ \mathit{zero}) \\ \mathit{stack} & \triangleq & \lambda b.\, \lambda n.\, n\ (\mathit{exp}\ b)\ (\mathit{succ}\ \mathit{zero}) \end{array}$$

Everything appears to be going swimmingly until we think of a very simple function, namely the predecessor function defined by

$$\begin{array}{lcl} \mathit{pred}\ 0 & = & 0 \\ \mathit{pred}\ (n+1) & = & n \end{array}$$

You may try for a while to see if you can define the predecessor function, but it is difficult. The problem is that we have to go from $\lambda s.\, \lambda z.\, s\, (\ldots (s\, z))$ to $\lambda s.\, \lambda z.\, s\, (\ldots z)$, that is, we have to *remove* an s rather than add an s as was required for the successor. One possible way out is to change representation and define $\overline{n}$ differently so that predecessor becomes easy (see Exercise 1).

We run the risk that other functions then become more difficult to define, or that the representation is larger than the already inefficient unary representation already is. We follow a different path, keeping the representation the same and defining the function directly.

We can start by assessing why the schema of iteration does not immediately apply. The problem is that in

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & g\ (f\ n) \end{array}$$

the function g only has access to the result of the recursive call of f on n, but not to the number n itself. What we would need is the *schema of primitive recursion*:

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & h\ n\ (f\ n) \end{array}$$

where n is passed to h. For example, for the predecessor function we have $c = 0$ and $h = \lambda x.\, \lambda y.\, x$ (we do not need the result of the recursive call, just n which is the first argument to h).

2.1 Defining the Predecessor Function

Instead of trying to solve the general problem of how to implement primitive recursion, let's define the predecessor directly. Mathematically, we write $n \dot{-} 1$ for the predecessor (that is, $0 \dot{-} 1 = 0$ and $n + 1 \dot{-} 1 = n$). The key idea is to gain access to n in the schema of primitive recursion by *rebuilding it* during the iteration. This requires *pairs*, a representation of which we will construct shortly.

Our specification then is

$$\mathit{pred}_2\, n = \langle n, n \dot{-} 1\rangle$$

and the key step in its implementation in the λ-calculus is to express the definition by a schema of *iteration* rather than *primitive recursion*. The start is easy:

$$\mathit{pred}_2\, 0 = \langle 0, 0\rangle$$

For $n + 1$ we need to use the value of $\mathit{pred}_2\, n$. For this purpose we assume we have a function *letpair* where

$$\mathit{letpair}\, \langle e_1, e_2\rangle\, k = k\, e_1\, e_2$$

In other words, *letpair* passes the elements of the pair to a "continuation" k. Using *letpair* we start as

$$\mathit{pred}_2\,(n+1) = \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,\ldots)$$

If pred_2 satisfies it specification then reduction will substitute n for x and $n \dot{-} 1$ for y. From these we need to construct the pair $\langle n+1, n\rangle$ which we can do, for example, with $\langle x+1, x\rangle$. This gives us

$$\begin{array}{lcl} \mathit{pred}_2\,0 & = & \langle 0, 0\rangle \\ \mathit{pred}_2\,(n+1) & = & \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,\langle x+1, x\rangle) \\ \\ \mathit{pred}\,n & = & \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,y) \end{array}$$

2.2 Defining Pairs

The next question is how to define pairs and *letpair*. The idea is to just abstract over the continuation itself! Then *letpair* isn't really needed because the functional representation of the pair itself will apply its argument to the two components of the pair, but if want to write it out it would be the identity.

$$\begin{array}{lcl} \langle e_1, e_2\rangle & \triangleq & \lambda k.\,k\,e_1\,e_2 \\ \mathit{pair} & \triangleq & \lambda x.\,\lambda y.\,\lambda k.\,k\,x\,y \\ \mathit{letpair} & \triangleq & \lambda p.\,p \end{array}$$

Then $\mathit{letpair}\;p\;k =_\beta p\,k$ and $\mathit{pair}\,e_1\,e_2 =_\beta \langle e_1, e_2\rangle$.

This is an example how introducing an abstraction (namely pairs) makes it easier to implement a challenging function.

2.3 Proving the Correctness of the Predecessor Function

Summarizing the above and expanding the definition of *letpair* we obtain the following definitions.

$$\begin{array}{lcl} \mathit{pred}_2 & \triangleq & \lambda n.\,n\,(\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\mathit{pair}\,\mathit{zero}\,\mathit{zero}) \\ \mathit{pred} & \triangleq & \lambda n.\,\mathit{pred}_2\,n\,(\lambda x.\,\lambda y.\,y) \end{array}$$

Let's do a rigorous proof of correctness of *pred*.[1] For the representation of natural numbers, we use the following equalities.

$$\begin{array}{lcl} \overline{0}\,g\,c & =_\beta & c \\ \overline{n+1}\,g\,c & =_\beta & g\,(\overline{n}\,g\,c) \end{array}$$

[1]We did not carry out this proof in lecture relying on intuition and testing instead.

Lemma 1 $\mathit{pred}_2\,\overline{n} =_\beta \overline{\langle n, n \dot{-} 1\rangle}$

Proof: By mathematical induction on n.

Base: $n = 0$. Then

$$\begin{array}{rll}
\mathit{pred}_2\,\overline{0} & =_\beta \overline{0}\,(\ldots)\,(\mathit{pair\,zero\,zero}) & \\
& =_\beta \underline{\mathit{pair\,zero\,zero}} & \text{By repn. of } 0 \\
& =_\beta \overline{\langle 0, 0\rangle} = \overline{\langle 0, 0 \dot{-} 1\rangle} & \text{By repn. of } 0 \text{ and pairs}
\end{array}$$

Step: $n = m + 1$. Then

$$\begin{array}{lr}
\mathit{pred}_2\,\overline{m+1} =_\beta \overline{m+1}\,(\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\mathit{pair\,zero\,zero}) & \\
=_\beta (\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\overline{m}\,(\lambda p.\,\ldots)\,(\ldots)) & \text{By repn. of } m+1 \\
=_\beta (\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,\underline{(\mathit{pred}_2\,\overline{m})} & \text{By defn. of } \mathit{pred}_2 \\
=_\beta \underline{(\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,\overline{\langle m, m \dot{-} 1\rangle}} & \text{By ind. hyp. on } m \\
=_\beta \overline{\langle m, m \dot{-} 1\rangle}\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x) & \\
=_\beta \underline{\mathit{pair}\,(\mathit{succ}\,\overline{m})\,\overline{m}} & \text{By repn. of pairs} \\
=_\beta \overline{\langle m+1, m\rangle} & \text{By repn. of successor and pairs} \\
= \overline{\langle m+1, (m+1) \dot{-} 1\rangle} & \text{By defn. of } \dot{-}
\end{array}$$

□

Theorem 2 $\mathit{pred}\,\overline{n} =_\beta \overline{n \dot{-} 1}$

Proof: Direct, from Lemma 1.

$$\begin{array}{rlr}
\mathit{pred}\,\overline{n} & = (\lambda n.\,\mathit{pred}_2\,n\,(\lambda x.\,\lambda y.\,y))\,\overline{n} & \\
& =_\beta \underline{\mathit{pred}_2 \overline{n}}\,(\lambda x.\,\lambda y.\,y) & \\
& =_\beta \underline{\overline{\langle n, n \dot{-} 1\rangle}\,(\lambda x.\,\lambda y.\,y)} & \text{By Lemma 1} \\
& =_\beta (\lambda k.\,k\,\overline{n}, \overline{n \dot{-} 1})\,(\lambda x.\,\lambda y.\,y) & \text{By repn. of pairs} \\
& =_\beta \overline{n \dot{-} 1} &
\end{array}$$

□

An interesting consequence of the Church-Rosser Theorem is that if $e =_\beta e'$ where e' is in normal form, then $e \longmapsto_\beta^* e'$.

2.4 Primitive Recursion

The general case of primitive recursion follows by a similar argument. Recall

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & h\ n\ (f\ n) \end{array}$$

We begin by defining a function f_2 specified with

$$f_2\, n = \langle n, f\, n \rangle$$

We can define f_2 using the schema of iteration.

$$\begin{array}{lcl} f_2\, 0 & = & \langle 0, c \rangle \\ f_2\, (n+1) & = & \mathit{letpair}\, (f_2\, n)\, (\lambda x.\, \lambda y.\, \langle x+1, h\, x\, y \rangle) \\ f\, n & = & \mathit{letpair}\, (f_2\, n)\, (\lambda x.\, \lambda y.\, x) \end{array}$$

To put this all together, we implement a function specified with

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & h\ n\ (f\ n) \end{array}$$

with the following definition in expressions of c and h:

$$\begin{array}{lcl} \mathit{pair} & = & \lambda x.\, \lambda y.\, \lambda g.\, g\, x\, y \\ f_2 & \triangleq & \lambda n.\, n\, (\lambda r.\, r\ (\lambda x.\, \lambda y.\, \mathit{pair}\ (\mathit{succ}\ x)\ (h\ x\ y)))\, (\mathit{pair}\ \mathit{zero}\ c) \\ f & \triangleq & \lambda n.\, f_2\, n\ (\lambda x.\, \lambda y.\, y) \end{array}$$

Recall that for the concrete case of the predecessor function we have $c = 0$ and $h = \lambda x.\, \lambda y.\, x$.

2.5 The Significance of Primitive Recursion

We have used primitive recursion here only as an aid to see how we can define functions in the pure λ-calculus. However, when computing over natural numbers we can restrict the functions that can be formed in schematic ways to obtain a language in which all functions terminate. Primitive recursion plays a central role in this because if c and g are terminating then so is f formed from them by primitive recursion. This can be proved by induction on n.

In this way we obtain a very rich set of functions. However, this set does for instance not include a function that simulates general Turing machines.

We will see in the next few lectures that for each terminating programming languages, there are total and computable functions that cannot be implemented.

Furthermore, if we give a so-called *constructive* proof of a statement in certain formulations of arithmetic with mathematical induction, we can extract a function that is defined by primitive recursion. We will probably not have an opportunity to discuss this observation further in this course, but it is an important topic in the course 15-317/15-657 *Constructive Logic*.

3 General Recursion

Recall the schemas of iteration and primitive recursion:

$$\begin{array}{lcl} f\,0 & = & c \\ f\,(n+1) & = & g\,(f\,n) \end{array} \qquad \begin{array}{lcl} f\,0 & = & c \\ f\,(n+1) & = & g\,n\,(f\,n) \end{array}$$

We have already seen how functions defined by iteration and primitive recursion can be represented in the λ-calculus. We can also see that functions defined in this manner are terminating as long as c and g are.

Since there are computable functions that do not fit such of schema of recursion, there must be a more general way of defining recursive functions in the λ-Calculus.

The most general recursion schema we might think of is

$$f = h\,f$$

which means that in the right-hand side we can make arbitrary recursive calls to f. For the function *plus*, the function h could be defined as follows.

$$\begin{array}{l} h = \lambda plus.\, \lambda n.\, \lambda k.\; \mathit{if}\;\; (n = 0)\; \mathit{then}\; k \\ \qquad\qquad\qquad\qquad\qquad \mathit{else}\; \mathit{succ}\; (\mathit{plus}\; (\mathit{pred}\; n)\; k) \end{array}$$

Here, we assume that we have functions for testing $x = y$ on natural numbers, and for conditionals (see Exercise L1.4). The function h reduces to the addition function *plus* if we apply it to *plus*.

The interesting question now is if we can in fact define an f explicitly when given h so that it satisfies $f = h\,f$. We say that f is a *fixed point* of h, because when we apply h to f we get f back. Since our solution should be in the λ-calculus, it would be $f =_\beta h\,f$. A function f satisfying such an equation may *not* be uniquely determined. For example, the equation $f = f$ (so, $h = \lambda x.x$) is satisfied by every function f. On the other hand, if h is a constant function such as $\lambda x. I$ then $f =_\beta (\lambda x.\, I)\, f =_\beta I$ has a simple unique solution. For the purpose of this lecture, any function that satisfies the given equation is acceptable.

If we believe in the Church-Turing thesis, then any computable function should be representable on Church numerals in the λ-calculus, so there is reason to hope there are explicit representations for such f. The answer is given by the so-called Y combinator.[2] Before we write it out, let's reflect on which laws Y should satisfy? We want that if $f = Y\, h$ and we specified that $f = h\, f$, so we get $Y\, h = h\,(Y\, h)$. We can iterate this reasoning indefinitely:

$$Y\, h = h\,(Y\, h) = h\,(h\,(Y\, h)) = h\,(h\,(h\,(Y\, h))) = \ldots$$

In other words, Y must iterate its argument arbitrarily many times.

The ingenious solution deposits one copy of h and the replicates $Y\, h$.

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

Here, the application $x\, x$ takes care of replicating $Y\, h$, and the outer application of h in $h\,(x\, x)$ leaves a copy of h behind. Formally, we calculate

$$\begin{array}{lll} Y\, h & =_\beta & (\lambda x.\, h\,(x\, x))\, (\lambda x.\, h\,(x\, x)) \\ & =_\beta & h\,((\lambda x.\, h\,(x\, x))\, (\lambda x.\, h\,(x\, x))) \\ & =_\beta & h\,(Y\, h) \end{array}$$

In the first step, we just unwrap the definition of Y. In the second step we perform a β-reduction, substituting $[(\lambda x.\, h\,(x\, x))/x]\, h\,(x\, x)$. In the third step we recognize that this substitution recreated a copy of $Y\, h$.

You might wonder a function defined with Y will ever terminate since

$$Y\, h =_\beta h\,(Y\, h) =_\beta h\,(h\,(Y\, h)) =_\beta h\,(h\,(h\,(Y\, h))) = \ldots$$

Well, it sometimes doesn't. Actually, this is important if we are to represent *partial recursive functions* which include functions that are undefined (have no normal form) on some arguments. Reconsider the specification $f = f$ as a recursion schema. Then $h = \lambda g.\, g$ and

$$Y\, h = Y\,(\lambda g.\, g) =_\beta (\lambda x.\, (\lambda g.\, g)\,(x\, x))\, (\lambda x.\, (\lambda g.\, g)\,(x\, x)) =_\beta (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

The expression on the right-hand side here (called Ω) only reduces to itself. It therefore does not have a normal form. In other words, the function

[2]For our purposes, a *combinator* is simply a λ-expression without any free variables.

$f = Y\,(\lambda g.\,g) = \Omega$ solves the equation $f = f$ by giving us a result which always diverges.

However, some recursive functions always terminate. Consider, for example, a case where f does not call itself recursively at all: $f = \lambda n.\,\mathit{succ}\,n$. Then $h_0 = \lambda g.\,\lambda n.\,\mathit{succ}\,n$. And we calculate further

$$\begin{array}{lcl}
Y\,h_0 & = & Y\,(\lambda g.\,\lambda n.\,\mathit{succ}\,n) \\
 & =_\beta & (\lambda x.\,(\lambda g.\,\lambda n.\,\mathit{succ}\,n)\,(x\,x))\,(\lambda x.\,(\lambda g.\,\lambda n.\,\mathit{succ}\,n)\,(x\,x)) \\
 & =_\beta & (\lambda x.\,(\lambda n.\,\mathit{succ}\,n))\,(\lambda x.\,(\lambda n.\,\mathit{succ}\,n)) \\
 & =_\beta & \lambda n.\,\mathit{succ}\,n
\end{array}$$

So, fortunately, we obtain just the successor function *if we apply β-reduction from the outside in*. It is however also the case that there is an infinite reduction sequence starting at $Y\,h_0$. By the Church-Rosser Theorem this means that at any point during such an infinite reduction sequence we could still also reduce to $\lambda n.\,\mathit{succ}\;n$. A remarkable and nontrivial theorem about the λ-calculus is that if we always reduce the left-most/outer-most redex (which is the first expression of the form $(\lambda x.\,e_1)\,e_2$ we come to when reading an expression from left to right) then we will definitely arrive at a normal form when one exists. And by the Church-Rosser theorem such a normal form is unique (up to renaming of bound variables, as usual).

If a fixed point is not unique then the result of the Y combinator will return the most undefined result. Consider again the successor function but this time we will use the recursive result. Define

$$h_1 \triangleq \lambda n.\,\mathit{succ}\,n$$

Then we calculate

$$\begin{array}{lcl}
Y\,h_1 & = & Y(\lambda n.\,\mathit{succ}\,n) \\
 & =_\beta & (\lambda n.\,\mathit{succ}\,n)(Y\,h_1) \\
 & =_\beta & \mathit{succ}\,(Y\,h_1) \\
 & = & (\lambda n.\,\lambda s.\,\lambda z.\,s\,(n\,s\,z))\,(Y\,h_1) \\
 & =_\beta & \lambda s.\,\lambda z.\,s\,((Y\,h_1)\,s\,z) \\
 & =_\beta & \lambda s.\,\lambda z.\,s\,((\lambda s.\,\lambda z.\,s\,((Y\,h_1)\,s\,z))\,s\,z) \\
 & =_\beta & \lambda s.\,\lambda z.\,s\,(s\,((Y\,h_1)\,s\,z)) \\
 & =_\beta & \ldots
\end{array}$$

Like Ω, this "infinite" number does not have a normal form but can appear in expressions that have a normal form. For example:

$$(Y\,h_1)(\lambda x.\mathit{succ}\;\mathit{zero})\;\mathit{zero} =_\beta \bar{1}$$

4 Defining Functions by Recursion

Consider the factorial function, which we deliberately write using general recursion rather than primitive recursion.

$\mathsf{fact}\ n = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathsf{fact}(n-1)$

To write this in the λ-calculus we first define a zero test *ifz* satisfying

$$\begin{aligned} \mathit{ifz}\ \overline{0}\ c\ d &= c \\ \mathit{ifz}\ \overline{n+1}\ c\ d &= d \end{aligned}$$

which is a special case of if iteration and can be written, for example, as

$\mathit{ifz} = \lambda n.\, \lambda c.\, \lambda d.\, n\, (K\ d)\ c$

Eliminating the mathematical notation from the recursive definition of fact get the equation

$\mathsf{fact} = \lambda n.\, \mathit{ifz}\ n\ (\mathit{succ}\ \mathit{zero})\ (\mathit{times}\ n\ (\mathsf{fact}\ (\mathit{pred}\, n)))$

where we have already defined *succ*, *zero*, *times*, and *pred*. Of course, this is not directly allowed in the λ-calculus since the right-hand side mentions fact which we are just trying to define. The function h_{fact} which will be the argument to the Y combinator is then

$h_{\mathsf{fact}} = \lambda f.\, \lambda n.\, \mathit{ifz}\ n\ (\mathit{succ}\ \mathit{zero})\ (\mathit{times}\ n\ (f\ (\mathit{pred}\, n)))$

and

$\mathit{fact} = Y\ h_{\mathsf{fact}}$

We can write and execute this now in LAMBDA notation (see file rec.lam)

```
defn I = \x. x
defn K = \x. \y. x
defn Y = \h. (\x. h (x x)) (\x. h (x x))

defn ifz = \n. \c. \d. n (K d) c

defn h_fact = \f. \n. ifz n (succ zero) (times n (f (pred n)))
defn fact = Y h_fact

norm _120 = fact _5
norm _720 = fact (succ _5)
```

Listing 1: Recursive factorial in LAMBDA

Exercises

Exercise 1 Once we can define each individual instance of the schemas of iteration and primitive recursion, we can also define them explicitly as combinators.

Define combinators *iter* and *primrec* such that

(i) The function *iter* $g\ c$ satisfies the schema of iteration

(ii) The function *primrec* $h\ c$ satisfies the schema of primitive recursion

You do not need to prove the correctness of your definitions.

Exercise 2 Define the following functions in the λ-calculus using the LAMBDA implementation. Here we take "=" to mean $=_\beta$, that is, β-conversion.

You may use all the functions in nat.lam as helper functions. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Provide at least 3 test cases for each function.

(i) if0 (definition by cases) satisfying the specification

$$\begin{array}{lcl} \mathsf{if0}\ \overline{0}\ x\ y & = & x \\ \mathsf{if0}\ \overline{k+1}\ x\ y & = & y \end{array}$$

(ii) even satisfying the specification

$$\begin{array}{lcl} \mathsf{even}\ \overline{2k} & = & \mathsf{true} \\ \mathsf{even}\ \overline{2k+1} & = & \mathsf{false} \end{array}$$

(iii) half satisfying the specification

$$\begin{array}{lcl} \mathsf{half}\ \overline{2k} & = & k \\ \mathsf{half}\ \overline{2k+1} & = & k \end{array}$$

Exercise 3 The Lucas function (a variant on the Fibonacci function) is defined mathematically by

$$\begin{array}{lcl} \mathsf{lucas}\ 0 & = & 2 \\ \mathsf{lucas}\ 1 & = & 1 \\ \mathsf{lucas}\ (n+2) & = & \mathsf{lucas}\ n + \mathsf{lucas}\ (n+1) \end{array}$$

Give an implementation of the Lucas function in the λ-calculus via the LAMBDA implementation.

You may use the functions from nat.lam as helper functions, as well as those from Exercise 2. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Test your implementation on inputs 0, 1, 9, and 11, expecting results 2, 1, 76, and 199. Include these tests in your code submission, and record the number of β-reductions used by your function.

Exercise 4 We can define binomial coefficients $\mathsf{bin}\ n\ k$ by the following recurrence:

$$\begin{array}{lcl} \mathsf{bin}\ 0\ k & = & 1 \\ \mathsf{bin}\ (n+1)\ 0 & = & 1 \\ \mathsf{bin}\ (n+1)\ (k+1) & = & \mathsf{bin}\ n\ k + \mathsf{bin}\ n\ (k+1) \end{array}$$

Give an implementation of the bin function in the λ-calculus via the LAMBDA implementation.

You may use the functions from nat.lam as helper functions, as well as those from Exercise 2. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Provide at least 5 test cases.

Exercise 5 Give an implementation of the factorial function in the λ-calculus as it arises from the schema of primitive recursion. How many β-reduction steps are required for factorial of 0, 1, 2, 3, 4, 5 in each of the two implementations?

Exercise 6 The Fibonacci function is defined by

$$\begin{array}{lcl} \mathsf{fib}\ 0 & = & 0 \\ \mathsf{fib}\ 1 & = & 1 \\ \mathsf{fib}\ (n+2) & = & \mathsf{fib}\ n + \mathsf{fib}\ (n+1) \end{array}$$

Give two implementations of the Fibonacci function in the λ-calculus (using the LAMBDA implementation). You may use the functions in (see file rec.lam).

(i) Exploit the idea behind the encoding of primitive recursion using pairs to give a direct implementation of fib without using the Y combinator.

(ii) Give an implementation of fib using the Y combinator.

Test your implementation on inputs 0, 1, 9, and 11, expecting results 0, 1, 34, and 89. Which of the two is more “efficient” (in the sense of number of β-reductions)?