

---

CMCS 312: Programming Languages

Lecture 4: Programming in the Lambda Calculus

Umut A. Acar

5 October 2006

---

## Contents

<b>1</b>	<b>Announcements</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>1</b>
<b>3</b>	<b>Multiple Arguments</b>	<b>1</b>
<b>4</b>	<b>Church Booleans</b>	<b>2</b>
<b>5</b>	<b>Pairs</b>	<b>3</b>
<b>6</b>	<b>Church Numerals</b>	<b>4</b>
	6.1 Evaluation Strategies . . . . .	5
	6.2 Recursion . . . . .	6
<b>7</b>	<b>Homework Exercise</b>	<b>7</b>

## 1 Announcements

Homework 1 is now available. It is due Tuesday (Oct 7).

## 2 Introduction

We have defined lambda calculus and talked about how we can evaluate its terms by applying  $\beta$  reduction. When we started talking about lambda calculus, I claimed that it is Turing complete (even though it has no notion of numbers, primitive operations such as addition, subtraction). In this class, we will see how lambda calculus can be used to writes various short programs.

## 3 Multiple Arguments

Many programming languages have mechanisms to pass multiple arguments to a function. In lambda calculus, every function (lambda abstraction) has only one argument. How can we write a function that takes multiple argument?

As an example, suppose we want to write a function that sums its two arguments (let's assume for convenience that we have such a sum operator). In analogy to the languages that you have programmed in, we expect to write

such a function as  $\lambda(x, y).x + y$ . Such a function can then be applied by writing  $(\lambda(x, y).x + y) (3, 5)$ . Since we cannot pass multiple arguments to a lambda abstraction we will instead use “currying” (named after Haskell Curry) and write the function as  $\lambda x.\lambda y.x + y$ . We apply the function as  $(\lambda x.\lambda y.x + y) 3 5$ .

$$\begin{aligned} (\lambda x.\lambda y.x + y) 3 5 &\rightarrow_{\beta} (\lambda y.3 + y) 5 \\ &\rightarrow_{\beta} 3 + 5 = 8. \end{aligned}$$

Note that the result of the first application itself is a function. This is critical to the effectiveness of lambda calculus. Functions (lambda abstractions) can return functions. This is sometimes referred as having functions as first-class values. Languages based on this principle allow you to treat functions just like any other data (e.g., you can place functions in data structures).

## 4 Church Booleans

How can we represent booleans in lambda calculus? To find out how let’s first think about how booleans are used. The “elimination” for a boolean is the **if** statement. What does an **if** statement do? It takes a boolean and two branches and picks one of the branches. Thus, if we represent boolean values as functions that select their first or second argument depending on their value, we can simulate the behavior of an **if** statement by applying the branches to the boolean value.

Based on this intuition, let define **tru** and **fls** as

$$\begin{aligned} \mathit{tru} &:= \lambda x.\lambda y.x \\ \mathit{fls} &:= \lambda x.\lambda y.y. \end{aligned}$$

We can now define an **if** statement as a function that takes a boolean and two branches and selects the right branch as follows:

$$\mathit{test} := \lambda x_b.\lambda y_1.\lambda y_2.x_b y_1 y_2.$$

For example, consider the term  $\mathit{test} \mathit{tru} x y$

$$\begin{aligned} \mathit{test} \mathit{tru} x y &= (\lambda x_b.\lambda y_1.\lambda y_2.x_b y_1 y_2) \mathit{tru} x y \\ &\rightarrow_{\beta} (\lambda y_1.\lambda y_2.\mathit{tru} y_1 y_2) x y \\ &\rightarrow_{\beta} (\lambda y_2 \mathit{tru} x y_2) y \\ &\rightarrow_{\beta} \mathit{tru} x y \\ &\rightarrow_{\beta} (\lambda x.\lambda y.x) x y \\ &\rightarrow_{\beta} ([x/x]\lambda y.x) y \\ &\rightarrow_{\beta} (\lambda y.x)y \\ &\rightarrow_{\beta} [y/y](\lambda y.x) \\ &\rightarrow_{\beta} x \end{aligned}$$

How about some operations on booleans? For example, how can we write the “not” operation? Remember that booleans are function that take two arguments

and select one. So we can write “not” as function that takes a boolean and two arguments and supplies the arguments to the boolean in reversed order.

$$not := \lambda x_b. \lambda y. \lambda z. x_b z y.$$

To see how “not” works, consider

$$\begin{aligned} not\ tru &= (\lambda x_b. \lambda y. \lambda z. z\ x_b\ z\ y)\ tru \\ &\rightarrow_{\beta} \lambda y. \lambda z. tru\ z\ y \\ &\rightarrow_{\beta} \lambda y. \lambda z. (\lambda x. \lambda y. x)\ z\ y \\ &\rightarrow_{\beta} \lambda y. \lambda z. z \\ &= fls. \end{aligned}$$

We can also define “not” as  $not := \lambda x. flstru$ . If the argument ( $x$ ) is  $tru$ , then this function will return  $fls$  and will return  $tru$  otherwise.

How about “and”? Again, “and” will take two booleans and return a boolean

$$and := \lambda x_1. \lambda x_2. x_1\ x_2\ false.$$

Similarly, we can write “or” as  $or := \lambda x_1. \lambda x_2. x_1\ tru\ x_2$ .

## 5 Pairs

Suppose we want to have pairs in our language. For example, in SML the pair of the numbers 3 and 5 are written as  $(3, 5)$ . In addition to the ability to construct pairs, we also want to be able to project out the first and second parts using primitives such as  $fst$  and  $snd$ , e.g.,  $fst(3, 5) = 3$   $snd(3, 5) = 5$ . Can this be done using lambda calculus.

Like with booleans, lets think about the elimination form for pairs. The elimination form for pairs are the primitives for taking a pair apart, i.e., projecting their first and second components. Thus we can think of a pair as a function that takes the elimination for as an argument and applies it to its components. The eliminations forms  $fst$  and  $snd$  simply take the pair and apply the  $tru$  and  $fls$  to project out the first and second components of the pair.

$$\begin{aligned} pair &= \lambda x_1. \lambda x_2. \lambda y. y\ x_1\ x_2 \\ fst &= \lambda x_p. x_p(\lambda x. \lambda y. x) \\ snd &= \lambda x_p. x_p(\lambda x. \lambda y. y) \end{aligned}$$

**Exercise:** Write the ML code for  $pair$ ,  $fst$ , and  $snd$  for pairs of integers. Try now for a pair of a boolean and an integer (of type `bool*int`).

**Answer:**

```
- val pair = fn (x: int) => fn (y: int) => fn (x: int -> int -> int) => s x y;
val pair = fn : int -> int -> (int -> int -> int) -> int
- pair (3,5);
```

```

val it = fn : (int -> int -> int) -> int
- val first = fn (x: (int -> int -> int) -> int) => x (fn x => fn y => x);
- val second = fn (x: (int -> int -> int) -> int) => x (fn x => fn y => y);
- first it;
  3
- second it;
  5

```

## 6 Church Numerals

How can we represent natural numbers using lambda calculus? Let's again think of the elimination form for numbers. For the case of numbers, there are many of them. But they all can be characterized as computing some property of the number (e.g., comparisons, sums etc). What do we need to know to compute a property of a natural number?

We need to know what the property is for number zero and how can we update the property for each additional increment over zero. Based on this intuition we can think of a natural number  $n$  as a function that takes a function  $s$  (successor) and  $z$  zero and applies  $s$  to  $z$   $n$  times.

$$\begin{aligned}
c_0 &= \lambda s. \lambda z. z \\
c_1 &= \lambda s. \lambda z. s z \\
c_2 &= \lambda s. \lambda z. s (s z) \\
&\vdots
\end{aligned}$$

This representation for natural numbers is often referred as *church numerals*. We can now write some primitives for church numerals. Let's start with a function for testing if a number is zero.

$$isZero := \lambda x_n. x_n (\lambda x. fls) tru$$

As examples, let's apply *isZero* to  $c_0, c_1$  and  $c_2$

$$\begin{aligned}
isZero\ c_0 &= (\lambda x_n. x_n (\lambda x. fls)\ tru) (\lambda s. \lambda z. z) \\
&\rightarrow_{\beta} (\lambda s. \lambda z. z) (\lambda x. fls)\ tru \\
&\rightarrow_{\beta} (\lambda z. z)\ tru \\
&\rightarrow_{\beta} tru \\
isZero\ c_2 &= (\lambda x_n. x_n (\lambda x. fls)\ tru) (\lambda s. \lambda z. s\ z) \\
&\rightarrow_{\beta} (\lambda s. \lambda z. s\ z) (\lambda x. fls)\ tru \\
&\rightarrow_{\beta} (\lambda z. (\lambda x. fls)\ z) tru \\
&\rightarrow_{\beta} (\lambda x. fls)\ tru \\
&\rightarrow_{\beta} fls \\
isZero\ c_2 &= (\lambda x_n. x_n (\lambda x. fls)\ tru) (\lambda s. \lambda z. s\ (s\ z)) \\
&\rightarrow_{\beta} (\lambda s. \lambda z. s\ (s\ z)) (\lambda x. fls)\ tru \\
&\rightarrow_{\beta} (\lambda z. (\lambda x. fls)\ ((\lambda x. fls)\ z)) tru \\
&\rightarrow_{\beta} (\lambda x. fls)\ ((\lambda x. fls)\ tru) \\
&\rightarrow_{\beta} (\lambda x. fls)\ fls \\
&\rightarrow_{\beta} fls
\end{aligned}$$

Let's define some arithmetic operations.

$$\begin{aligned}
succ &:= \lambda x_n. \lambda s \lambda z. x_n s\ (s\ z) \\
plus &:= \lambda m. \lambda n. \lambda s. \lambda z. ms(ns) \\
times &:= \lambda m. \lambda n. \lambda s. \lambda z. m(plusn)
\end{aligned}$$

**Exercise:** Write the subtraction operation for church numerals.

## 6.1 Evaluation Strategies

The only means of computing the lambda calculus is the application of a function to its arguments, *i.e.*,  $(\lambda x. t_1)t_2 \rightarrow_{\beta} [t_1/x]t_2$ . This is called *beta reduction* and the application being reduced is called a *reducible expression* or a *redex*. Given a term, we evaluate that term by applying beta reduction to its redexes. There are often multiple redexes in a term, and, in general, we can choose to evaluate any one of them. In time, the following *evaluation strategies* have been proposed.

**Full beta reduction:** Any redex can be reduced at any time.

**Normal order strategy:** The leftmost and outermost redex is reduced next.

**Call by name strategy:** The leftmost and outermost redex is reduced next but no reductions are allowed under abstractions.

**Call by value:** Only outermost redexes are reduced and a redex is reduced when its right hand-side is a value.

Here is a call-by-value operational semantics for lambda calculus.

$$\frac{\frac{\frac{t_1 \rightarrow t'_1}{t_1 t_1 \rightarrow t'_1 t_2} \quad t_1 \rightarrow t'_1}{v_1 t_2 \rightarrow v_1 t'_2}}{(\lambda x.t)v \rightarrow [v/x] t}$$

## 6.2 Recursion

In lambda calculus, we can only write *anonymous functions*, *i.e.*, function that do not have names. This is limited because it seems to prohibit us from writing recursive functions. If we can't name a function how can we call it within the body of that function?

It turns out that recursion is not a problem. Why? Although lambda calculus does not allow us to name functions, it allows us to duplicate terms. Consider for example the term  $\omega = \lambda x.x x$  and the application  $\omega t = (\lambda x x) t$ . By beta reduction we have  $\omega t = t t$ . We now have two copies of the term  $t$ . Based on this intuition, it is possible to construct a *fix-point* combinator that gives us the "recursive form" of an anonymous function.

Suppose we want to write a recursive function  $g$ . How can we write such a function? Here is one way  `$\lambda f. \lambda x. \text{if } x \leq 0 \text{ then } 1 \text{ else } x * f(x-1)$` . Now if we can get a hold on the recursive form of  $g$ , denoted  $G$ , then we can call  $g$  with  $G$ . This is the idea behind our approach. We call the recursive version  $G$  the *fix-point* of  $g$ , denoted  `$\text{fix } g$` . There are several ways we can come up with a *fix-point combinator*, a combinator that when applied to  $g$  gives us its fix points. They are all similar to the term  $\omega$ .

Let's develop the intuition a bit further. Suppose we are given the  $g$ . To apply  $g$  to itself twice, we want to have a way of making two copies of  $g$ , and retain the ability to make further copies. Can we use  $\omega$ ? One problem with  $\omega$  is that loses its ability to replicate after one application. How about  $\omega \omega$ , *i.e.*, the term  $\Omega = (\lambda x. x x) (\lambda x. x x)$ . This combinator reproduces itself! Let's now stick a function argument  $f$  into  $\Omega$ ,  `$\lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$` , and let's call this the  $Y$  combinator. Now note that  $Y g \rightarrow_{\beta} g (Y g)$ . This suffices to give us recursion under the restriction that we delay evaluation of  $Y g$  until after  $g$ . This is exactly what the call by name evaluation would do. The  $Y$  combinator thus suffices as a fix-point combinator in the call-by name setting.

Does this work in the call-by value setting? Not quite. Because, the in the call by value setting  `$g (Y g) \rightarrow_{\beta} g (g (Y g))$`  and the evaluation diverges. We need a different combinator. This new combinator is very similar to the  $Y$  combinator—it just suspends evaluation of the fix point operator until it is applied by  $g$ . This is called the  $Z$  combinator. Here it is:  `$Z = \lambda f. \lambda x. f (\lambda y. x x y) (\lambda x. f (\lambda y. x x y))$` .

## 7 Homework Exercise

1. Give the call by name and call by value operational semantics for lambda calculus and show that they are deterministic. That is for each semantics show that if  $t \rightarrow t'$  and  $t \rightarrow t''$ , then  $t' = t''$ .
2. Define  $g := \lambda f. \lambda x. \text{if } x \leq 0 \text{ then } 1 \text{ else } x * f(x-1)$ . Apply beta reduction on  $g(Z\ g)$  until  $g$  is unrolled a few times to have the **if** term to be called two times. Explain  $(Z\ g)$  is the fixpoint of  $g$  in no more than five lines.