

CM20167
Topic 8: Introduction to the λ -calculus

Guy McCusker

1W2.1

What's the λ -calculus

The λ -calculus, or lambda-calculus, is a *theory of functions* first proposed by Alonzo Church in the 1930s.

It is a formal system in which

- ▶ every term denotes a function
- ▶ any term (function) can be applied to any other term, so functions are inherently *higher-order*

The syntax and semantics of the λ -calculus are very small and simple, so it is an amazing fact that

the λ -calculus is a fully expressive (i.e. *Turing-complete*) language of computable functions.

Syntax of the λ -calculus

The syntax of the λ -calculus is given by the following grammar.

λ -terms

$$M ::= x \mid \lambda x.M \mid MM$$

where x ranges over an infinite collection of *variables*.

This means that:

- ▶ any variable x is a term of the λ -calculus
- ▶ if M is a term and x is a variable, then $\lambda x.M$ is a term
- ▶ if M_1 and M_2 are terms, then so is $M_1 M_2$
- ▶ ... and nothing else is a term.

Meaning

The meanings of λ -terms are, roughly, as follows.

- ▶ $\lambda x.M$ is a *function*. The variable x is its parameter, and M is its body. So, when it is applied to some argument, M will be “evaluated” (whatever that means!) with x bound to the argument.
- ▶ MN is the *application* of function M to argument N .

Example terms

- ▶ x
- ▶ $\lambda x.x$
- ▶ $\lambda x.\lambda y.x$
- ▶ $\lambda x.\lambda y.yx$
- ▶ $(\lambda x.x)(\lambda y.\lambda z.x)$
- ▶ xx — self-application is possible
- ▶ $\lambda m.(\lambda f.ff)(\lambda f.m(\lambda a.f\bar{f}a))$ — cf. the fixed-point combinator from the previous lecture

Note that the grammar on the previous slide described the *abstract syntax* of the language; when we come to write terms down concretely we will often use brackets, as above, to clarify the structure of terms.

Conventions

We used plenty of brackets in the part of the course which was about Lisp. So, to keep brackets to a minimum in this part, we will use some conventions for writing λ -terms:

- ▶ application associates left: MNP means the same as $(MN)P$
- ▶ the scope of a λ extends as far to the right as possible: $\lambda x.MN$ means $\lambda x.(MN)$, not $(\lambda x.M)N$.
- ▶ we sometimes write $\lambda xy.M$ for $\lambda x.\lambda y.M$. Some authors omit the full-stop, writing λxM for $\lambda x.M$. That probably won't happen in this course.

Computation in the λ -calculus

Terms of the λ -calculus are supposed to describe *functions*, and we're supposed to be able to see how those functions *compute* things.

The model of computation will be given by *transformations* of terms: we will define a computation step as a transformation which takes a term, manipulates it, and yields a new term.

Computation steps

We'll write things like $M \rightarrow M'$ to say that term M can become term M' in one step of computation. Then we'll be interested in sequences of computation steps:

$$M \rightarrow M' \rightarrow M'' \rightarrow \dots$$

Some terms will have the property that there are no computation steps one can perform on them. These are called *normal forms*: they're the "final answer" of a computation.

Computation, very roughly

The key computation step in the λ -calculus is evaluating a function application.

Intuitively, when a function $\lambda x.M$ is applied to an argument N , what happens is that M is evaluated, but with all occurrences of the variable x replaced by N .

Warning: incorrect definitions below

Define $M[N/x]$ to be the term M , with every occurrence of x replaced by N .

We say that the application $(\lambda x.M)N$ *reduces to* $M[N/x]$.

This definition has a nasty bug which we will soon discover. Later, we will define the operation of substitution and the notion of reduction properly.

Some example evaluations

- ▶ $(\lambda x.x)M$ reduces to $x[M/x]$ which is M . The term $\lambda x.x$ is called the *identity* function because it returns the same value that it is given as an argument.
- ▶ $(\lambda x.\lambda y.x)M$ reduces to $(\lambda y.x)[M/x]$ which is $\lambda y.M$.
- ▶ If M contains no occurrences of y , $(\lambda y.M)N$ reduces to $M[N/y]$ which is just M .
- ▶ Putting the last two reductions together, we get a sequence:

$$(\lambda x.\lambda y.x)MN \rightarrow (\lambda y.M)N \rightarrow M$$

So reduction can *destroy* part of a term: here the N has been thrown away.

Some example evaluations

- ▶ $(\lambda x.\lambda f.fxx)M$ reduces to $\lambda f.fMM$, so reduction can *duplicate* parts of a term: here the M has been duplicated.
- ▶ We are allowed to perform reductions inside terms, so $\lambda x.(\lambda y.y)N$ reduces to $\lambda x.(y[N/y])$ which is $\lambda x.N$.
- ▶ $(\lambda x.xx)(\lambda x.xx)$ reduces to itself. There is therefore an infinite sequence of reduction steps

$$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \dots$$

This term is the leading example of a term with no normal form, and it has a special name: Ω . The term $\lambda x.xx$ is called ω .

Some example evaluations

Define Y to be the term

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

Then

$$YM \rightarrow (\lambda x.Mxx)(\lambda x.Mxx) \rightarrow M(\lambda x.Mxx)(\lambda x.Mxx)$$

and also $M(YM) \rightarrow M(\lambda x.Mxx)(\lambda x.Mxx)$.

It looks like YM and $M(YM)$ are going to compute the same thing. That is, YM is a *fixed-point* of the function M .

This term Y is called the (normal-order) fixed-point combinator. The Lisp function Y we discussed in the last lecture does a similar job in Lisp. The Lisp version has to be slightly different because of Lisp's *evaluation order*.

Evaluation order

In the λ -calculus, we are allowed to reduce any *redex* (reducible expression) within a term. A redex is a subterm of the form $(\lambda x.M)N$.

This means that there are often more than one possible reduction for a term.

For instance, in the term

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

we could reduce the *leftmost* redex: the λz . . . , to get

$$\lambda y.y$$

or we could reduce the *rightmost* redex: the $(\lambda x.xx)(\lambda x.xx)$, which of course reduces to itself. Then we'd get back to where we started.

Evaluation order

In Lisp, there would be no choice: when Lisp evaluates a function, it uses a fixed *reduction strategy*:

- ▶ reduce the function part until it becomes something of the form λx . . .
- ▶ reduce the argument part until it gives an answer (some kind of normal form)
- ▶ only then, substitute the argument into the function body.

Thus, in Lisp, the term above would loop forever, even though there is a perfectly good normal form for it.

Naming of variables

In a term like $\lambda x.\lambda y.x$, it should not matter what the variable x is called.

That is, that term will be the same as (or very similar to) $\lambda z.\lambda y.z$.

This is what we expect from our experience of programming: we can call variables whatever we like, and changing a program by making a consistent change of variable names does not alter that program.

The key word above is *consistent*: when we change the name of a variable x to z as above, we have to change *all* occurrences of that variable, and we have to make sure there's nothing else called z that we're going to get confused with.

Variable renaming in Java

In Java (and C etc) we could write code like that on the left. By the power of consistent renaming, we can change it to look like the code on the right, which is clearer. However, they both do the same thing.

```
{ int x;
  ...
  { int x;

    ...
    // use the inner x here
  }

  ...
  // use the outer x here
}
```

```
{ int y;
  ...
  { int z;

    ...
    // use z here
  }

  ...
  // use y here
}
```

Variable renaming in Java

However, things would go wrong if we took the code on the left below, and changed it to look like the code on the right.

```
{ int y;
  ...
  { int z;

    ...
    // use both y and z here
  }

  ...
  // use y here
}
```

```
{ int x;
  ...
  { int x;

    ...
    // use inner and outer x
  }

  ...
  // use outer x here
}
```

Variable naming in the λ -calculus

We have the same situation in the λ -calculus.

A term like $\lambda x.M(\lambda x.N)P$ where

- ▶ M and P use the outer x
- ▶ N uses the inner x

can be rewritten as

$$\lambda y.M[y/x](\lambda z.N[z/x])P[y/x]$$

and what we get is the “same” term, whatever that means.

What are variable names really for?

The important thing about variable names is that they correctly identify the entity they are supposed to refer to.

In the case of the λ -calculus, every variable will ultimately refer to the argument of some function. So, what's important about a variable is the λ -abstraction to which it refers. A graphical notation like



$\lambda\dot{\circ}.\dot{\circ}$

would represent $\lambda x.x$ perfectly well, with no need for variable names.

We'll use that notation on a few occasions, completely informally.

What we'll do

What we are going to do to account for this name-independence is:

- ▶ introduce an appropriate notion of terms being “the same, up to variable renaming”: this is called α -equivalence
- ▶ ensure that all the work we do, in particular the computation step, respects this equivalence
- ▶ later, have a brief look at other ways of representing variables, a little bit like the picture above.

Free and bound variables

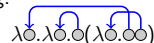
In a term such as $\lambda x.yx$, it is okay to rename x to anything we like, except y of course.

It is not okay to rename y !

Suppose we're dealing with that term as part of a larger term:

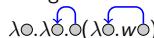
$$\lambda y.\lambda z.z(\lambda x.yx)$$

Just working on the $\lambda x.yx$ subterm, if we rename x to w , say, we get a term with the same meaning:



$\lambda\dot{\circ}.\lambda\dot{\circ}.\dot{\circ}(\lambda\dot{\circ}.\dot{\circ}\dot{\circ})$

Just working on the $\lambda x.yx$ subterm, if we rename y to w , say, we get a term with a very different meaning.



$\lambda\dot{\circ}.\lambda\dot{\circ}.\dot{\circ}(\lambda\dot{\circ}.\dot{\circ}w\dot{\circ})$

Free and bound variables

The difference here is that in $\lambda x.yx$, x appears *bound*—that is, it is in the scope of a λ —while y appears *free*—not bound.

The set of free variables of a term M , $FV(M)$, is defined as follows:

$$\begin{aligned}FV(x) &= \{x\} \\FV(\lambda x.M) &= FV(M) \setminus \{x\} \\FV(MN) &= FV(M) \cup FV(N)\end{aligned}$$

The set of bound variables of a term M , $BV(M)$, is defined as follows:

$$\begin{aligned}BV(x) &= \emptyset \\BV(\lambda x.M) &= BV(M) \cup \{x\} \\BV(MN) &= BV(M) \cup BV(N)\end{aligned}$$

Free and bound occurrences

However, what we are really interested in are the *free occurrences* and *bound occurrences* of a variable x in a term M . That is, actual occurrences of x which occur outside the scope of a λ ; and actual occurrences of x which occur inside the scope of a λ , plus some indication of which λ that is.

It's tricky to make this completely precise, because defining "occurrences" is a problem, but roughly:

Define the *free occurrences of x in M* , written $FO(x, M)$, as follows:

$$\begin{aligned}FO(x, x) &= \text{the unique occurrence of } x \\FO(x, y) &= \text{none, if } x \neq y \\FO(x, \lambda x.M) &= \text{none} \\FO(x, \lambda y.M) &= FO(x, M), \text{ if } x \neq y \\FO(x, MN) &= FO(x, M) \cup FO(x, N)\end{aligned}$$

Free and bound occurrences

Then we can define the occurrences of x bound by a particular λ very easily:

In a term $\lambda x.M$, the λ binds all those occurrences of x which are free in M .

The x in λx is called the *binding occurrence*, and the occurrences of x which are free in M are the *bound occurrences* of x for that λ .

Free and bound occurrences

We illustrate the free and bound occurrences of x in the term below by drawing an arrow from bound occurrences to their binding occurrences, and circling free occurrences.

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

Free and bound occurrences

Exercise

For each of the terms below, draw an arrow from every bound occurrence of a variable to its binding occurrence. Circle every free occurrence.

- 1 x
- 2 $\lambda x. x$
- 3 $(\lambda a. z) a$
- 4 $(\lambda n. n) z$
- 5 $\lambda z. (\lambda y. (\lambda x. x) y) z$
- 6 $(\lambda t. ((\lambda t. (\lambda t. t) t) t)) t$

α -conversion

We are going to define an *equivalence relation* $=_\alpha$ on terms which says that two terms differ only in the names of their bound variables.

This is referred to as α -*equivalence* and the process of renaming bound variables is called α -*conversion*.

α -conversion

First, we define the renaming operation: $M[y/x]$ is “ M with x renamed to y ”, and is defined as follows:

$$\begin{aligned}x[y/x] &= y \\z[y/x] &= z && \text{if } z \neq x \\(\lambda x.M)[y/x] &= \lambda x.M \\(\lambda z.M)[y/x] &= \lambda z.(M[y/x]) && \text{if } z \neq x \\(MN)[y/x] &= (M[y/x])(N[y/x])\end{aligned}$$

Notice the similarity with the definition of free occurrences. This is no coincidence: what we’re doing here is renaming the free occurrences of x .

α -equivalence

We can now give a definition of the equivalence relation $=_\alpha$. We shall write $M \equiv N$ to mean that M is syntactically identical to N .

$$\begin{aligned}M =_\alpha x &\iff M \equiv x \\M =_\alpha \lambda x.N &\iff M \equiv \lambda z.N' \wedge N'[w/z] =_\alpha N[w/x] \\&\quad \text{for some (any) previously unused variable } w \\M =_\alpha N_1 N_2 &\iff M \equiv M_1 M_2 \wedge M_i =_\alpha N_i, i = 1, 2\end{aligned}$$

So for instance $\lambda x.\lambda y.x(yx)w =_\alpha \lambda y.\lambda z.y(zy)w$.

Basic properties of $=_\alpha$

Exercise

Show that $=_\alpha$ is an equivalence relation, that is, that

- ▶ $M =_\alpha M$ for every term M
- ▶ if $M =_\alpha N$ then $N =_\alpha M$.
- ▶ if $M =_\alpha N$ and $N =_\alpha P$ then $M =_\alpha P$.

Substitution of terms

The basic reduction step, called a β -reduction, takes a term $(\lambda x.M)N$ and reduces it to $M[N/x]$.

Since we are thinking of $\lambda x.M$ as “the same as” $\lambda y.M[y/x]$ (where y is a fresh variable), it had better be the case that $M[N/x]$ is “the same as” $M[y/x][N/y]$.

This requires some care in the definition of substitution of terms.

Substitution and variable capture

Consider the term

$$(\lambda x.\lambda y.x)(\lambda z.yz) \quad (\lambda \circ.\lambda \circ.\circ)(\lambda \circ.y\circ)$$

After a naive reduction step, this becomes

$$\lambda y.(\lambda z.yz). \quad \lambda \circ.(\lambda \circ.\circ\circ)$$

But if we first α -convert the original term, renaming the bound y to w , we get

$$(\lambda x.\lambda w.x)(\lambda z.yz) \quad (\lambda \circ.\lambda \circ.\circ)(\lambda \circ.y\circ)$$

which reduces to

$$\lambda w.(\lambda z.yz) \quad \lambda \circ.(\lambda \circ.y\circ)$$

Oh no! These are not the same!

Variable capture

What has happened is that in the naive reduction step

$$(\lambda x.\lambda y.x)(\lambda z.yz) \rightarrow \lambda y.(\lambda z.yz)$$

a free occurrence of y has become bound by the λy . We say that the variable y has been *captured*.

This is not correct, as the second version of the reduction shows.

This means we need a better definition of substitution: *capture-avoiding substitution*.

Capture-avoiding substitution

Here's the final, corrected definition of substitution of terms.

$$\begin{aligned}x[N/x] &= N \\y[N/x] &= y && \text{if } y \neq x \\(\lambda x.M)[N/x] &= \lambda x.M \\(\lambda y.M)[N/x] &= \lambda z.M[z/y][N/x] \\&\quad \text{where } z \text{ is a variable not free in } M \text{ or } N \\M_1 M_2[N/x] &= M_1[N/x] M_2[N/x]\end{aligned}$$

The key case is substituting into a λ -abstraction. Here we first α -convert the abstraction, so that the bound variable does not appear free in the terms, and then perform the substitution.

This means that the abstracted variable cannot capture any variables from N .

Exercise

Show that if $x \notin \text{FV}(M)$ then $M[N/x] \equiv M$.

β -reduction

Now that we have substitution properly defined, we can define β -reduction.

A term of the form $(\lambda x.M)N$ is called a β -redex, and it reduces to $M[N/x]$ as we expect.

We write

$$M \rightarrow_{\beta} N$$

if N results from β -reducing any subterm of M .

We write

$$M \rightarrow_{\beta}^* N$$

if N results from a (possibly zero-length) sequence of \rightarrow_{β} steps and α -conversions, starting from M , i.e.

$$M \rightarrow_{\beta} M_1 =_{\alpha} M'_1 \rightarrow_{\beta} M_2 =_{\alpha} M'_2 \rightarrow_{\beta} \dots \rightarrow_{\beta} N' =_{\alpha} N.$$

Capture-avoidance and scope

Consider these Lisp functions:

```
(setq n 23)
(defun foo () (print n))
(defun bar () (let ((n 42)) (print n) (foo)))
```

What do you think happens when we call (bar)?

In EuLisp, and in Scheme, the `n` in `foo` is *not* captured by the `let`. That is, 42 is printed, and then 23 is printed. This requires the capture-avoiding substitution we defined above.

In some Lisps, like Cambridge Lisp, the variable *is* captured, so 42 is printed twice.

Lexical versus dynamic scope

The λ -calculus, EuLisp, Scheme etc. employ what is called *lexical scope*: a variable is bound where it looks like it is bound when you look at the text of the program.

Cambridge Lisp, Emacs Lisp and others, employ *dynamic scope*: a variable is bound by the binding we encountered most recently while running the program.

There are arguments for and against each kind of scoping. In the λ -calculus, because we're not specifying what order to reduce things, dynamic scoping would be very very odd.

Nowadays, a lot of Lisp systems let you decide which variables use which kind of scope. Scheme doesn't: everything in Scheme is lexical.

Lexical scope and "objects"

Lexical scope means we can have several different variables with the same name, and lets us store local values inside functions, providing a simple form of object:

```
(setq n 23)

(let ((n 0))
  (defun countme ()
    (setq n (+ n 1)))
)
```

```
user> n
23
user> (countme)
1
user> (countme)
2
user> n
23
user> (countme)
3
```

Normal forms

Back to the λ -calculus.

A term is said to be a normal form if none of its subterms is a redex. For example:

- ▶ $\lambda x.x$
- ▶ $\lambda f.\lambda x.f(f(x))$
- ▶ $\lambda f.f(\lambda y.f(\lambda x.x))$

A term N is said to be a normal form of M if N is a normal form and $M \rightarrow_{\beta}^* N$.

Not every term has a normal form: for example $\Omega \equiv \omega\omega \equiv (\lambda x.xx)(\lambda x.xx)$ reduces to itself, which means it has no normal form.

Multiple redexes—multiple normal forms?

When a term has more than one redex, there are different β -reductions that can be done. For example:

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

can be reduced to

$$(\lambda z.zz)(\lambda y.y)$$

or to

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

Whichever we choose, we can keep reducing and eventually we get to $\lambda y.y$.

The fact that a term can be reduced in different ways raises a question:

Question

Can a term have more than one normal form?

The Church-Rosser Theorem

The Church-Rosser theorem will show, among other things, that a term can have at most one normal form.

That means that normal forms are a reasonably good notion of “result” of a computation.

We’re also going to see that the λ -calculus can be used to encode the usual kinds of things that we need for computation: numbers, arithmetic, recursive definitions etc: each number will be a particular normal form, and we’ll write λ -terms that manipulate these to encode computable functions.

Exercise

The term W is defined to be

$$\lambda x. \lambda y. xyy.$$

Give a complete analysis of the sequences of reductions that the term WWW can perform.

