

CM20167 Revision

Guy McCusker

1W2.1

How do we revise for this course?

- ▶ How should I know?
- ▶ Learn what was taught!
- ▶ Open your mind and let the λ -calculus inform your understanding of Lisp and vice versa
- ▶ Make sure you have the Big Ideas clear.

What are the Big Ideas?

- ▶ Programming just using functions: everything is a function (apart from some bits of basic data) and functions can be applied to anything, including other functions.
- ▶ Applying functions to arguments really means “plugging in” the arguments in place of the formal parameters of the functions.
- ▶ Recursion everywhere.
- ▶ In Lisp, lists everywhere.
- ▶ In the λ -calculus, encodings everywhere.

Syntax

- ▶ In Lisp, we have basic bits and bobs (numbers, symbols) and we have expressions

`(thing1 thing2 thing3).`

Everything in Lisp is unambiguously one of these. (Some expressions are special.)

- ▶ In the λ -calculus, we have variables like x , abstractions like $\lambda x.M$ and applications like MN . Every term is unambiguously one of these.

Execution

- ▶ In Lisp, how is an expression evaluated? That is, what does
`(myfun arg1 arg2 arg3)`
mean?

- ▶ In the λ -calculus, how is a redex reduced? That is, what does something like

$(\lambda x.M)N$

mean?

- ▶ When do we apply these evaluations? That is, what is the reduction order (aka reduction strategy)? In Lisp, it's fixed; in the λ -calculus, it's not. How do we know that's not a problem?

Recursion

In Lisp, most useful functions are defined by recursion. On lists, almost every recursion looks a bit like this one:

```
(defun map (func list)
  (if (null list)
      nil
      (cons (func (car list)) (map func (cdr list)))
  )
)
```

That is,

- ▶ give a special answer for the empty list
- ▶ for a non-empty list, perform a recursive call on its `cdr`, then do something with that, possibly involving the `car`.

A generic recursive function

A generic version of this is as follows:

```
(defun foldr (f a l)
  (if (null l)
      a
      (f (car l) (foldr f a (cdr l)))
  )
)
```

Here the argument `a` tells us the special answer to give for the empty list, and the argument `f` tells us how to combine the result of the recursion with the `car`.

Exercise in folding

Suppose `func` is some function.

Give a function `f` and an expression `a` such that
`(foldr f a list)`
computes the same thing as
`(map func list)`
for any list called `list`.

Solving it

Plugging arguments into the definition, we see that `(foldr f a list)` gives us the expression

```
(if (null list)
    a
    (f (car list) (foldr f a (cdr list)))
)
```

whereas `(map func list)` gives us the expression

```
(if (null list)
    nil
    (cons (func (car list)) (map func (cdr list)))
)
```

We want them to be the same. Obviously this means `a` should be `nil`. What about `f`?

Solving it

Assuming the recursive calls work out the same, we just need the “putting it all together” part to be the same. So we need

```
(f (car list) something)
```

to be the same as

```
(cons (func (car list)) something).
```

That is, $(f\ x\ y)$ should be

```
(cons (func x) y).
```

So define it like this:

```
(defun f (x y)
  (cons (func x) y)
)
```

The λ -calculus

Make sure you know how the λ -calculus works. This means:

- ▶ β -reduction: $(\lambda x.M)N \rightarrow_{\beta} M[N/x]$. This in turn means:
- ▶ Substitution: what does $M[N/x]$ mean?
- ▶ Remember about avoiding capture of free variables. We don't want

$$(\lambda y.xyx)[y/x]$$

to be

$$\lambda y.yyy.$$

- ▶ This involves renaming bound variables, i.e. α -conversion: rename the bound y to z and we get

$$(\lambda z.xzx)[y/x]$$

which easily substitutes to

$$\lambda z.zzy.$$

Theoretical points

- ▶ The λ -calculus lets us reduce any redex, anywhere inside a term.
- ▶ The Church-Rosser property guarantees us that these choices don't make a difference to any final answer we may compute, although they may affect computation time.
- ▶ If two terms can be converted to one another by forward and backward β -steps, they're called β -equal and considered to be “the same” in some sense.
- ▶ There's the idea of *normal form* which means redex-free term. These are the final answers. We also talk about head normal forms and weak head normal forms occasionally.
- ▶ There are different *reduction strategies* we can adopt, which tell us which redex to work on when there is a choice. The main two are normal order and applicative order.
- ▶ There's also *lazy reduction* which is an implementation technique.

Relation with functional programming

Lisp and other functional languages realise the ideas in the λ -calculus. The basic step of computation with Lisp and so on is just like a β -step.

λ -calculus functions only take one argument but Lisp functions can take several. How do we simulate multi-argument functions? (Currying.)

Most real functional programming languages choose a particular strategy for reduction.

In Lisp, evaluation is done in applicative order; but we always stop at weak head normal forms. This unfortunately means that some vital bits of Lisp are not really functions: for instance the short-circuit `and` operator.

In Haskell, evaluation is done with the lazy technique, again stopping at `whnfs`.

Programming in the λ -calculus

The λ -calculus lets us simulate everything that we can do in Lisp.

There are special terms P , L and R such that for any M and N we have

$$\begin{aligned}L(P M N) &=_{\beta} M \\R(P M N) &=_{\beta} N\end{aligned}$$

That is, $P M N$ acts like `(cons M N)`, L acts like `car` and R acts like `cdr`.

Similarly we can simulate

- ▶ true, false and if-then-else
- ▶ the empty list and the test for emptiness of a list
- ▶ zero, +1, -1, test for zero
- ▶ and so on.

What you need to know about this

Try to understand

- ▶ what it would mean for a collection of λ -terms to simulate a collection of programming constructs properly
- ▶ how to “program” once you’ve got the encodings
- ▶ how to define the most important encodings.

The first two of these are about *thinking*, while the last one is just boring old *learning*.

Recursion and fixed points

An important theoretical role is played by fixed point combinators. You need to know:

- ▶ What it means to be a fixed point combinator. Basically $YM =_{\beta} M(YM)$.
- ▶ Why a fixed point combinator lets you encode recursion.
- ▶ How to write down a fixed point combinator. Please not the Klop one. And I promise not to make you reproduce the Lisp one from Topic 7.

Understanding recursive definitions as fixed points is a key conceptual step in understanding programming.

Exam questions

There will definitely be questions about Lisp, and questions about the λ -calculus. There could well be questions which ask you to relate the two in the kinds of ways I've indicated.

A Lisp question might ask you to explain some concepts (e.g. tail recursion), define some simple Lisp functions, and explain how they work or analyse their behaviour (informally).

A λ -calculus question might ask you to give some basic definitions from the notes, e.g. β -reduction, explain some concepts, and write some example functions.

Things there will not be questions about

There will not be questions about:

- ▶ anything that did not feature heavily on the course e.g. combinators, de Bruijn indices, λ -calculus with constants (and delta rules etc).
- ▶ the typed λ -calculus
- ▶ Haskell (although the idea that not all functional languages are like Lisp is important and might come up somewhere)
- ▶ apple horticulture in the south west of England between the wars.