

Topic 1: Getting started with Lisp

Guy McCusker¹

¹University of Bath

What is Lisp?

- ▶ Lisp is a functional programming language.
- ▶ It's the second-oldest high-level language in the world. Wow.
- ▶ Lisp is an abbreviation of "List Processing" because what you do in Lisp is to process lists.
- ▶ A list is what you think it is: something like
(ham mozzarella tomato)

What is a list?

A list is a collection of "things", enclosed in parentheses, and separated by spaces.

A "thing" can be:

- ▶ an *atom*: numbers, strings, symbols, ...
- ▶ another list
- ▶ (other kinds of structures which we won't discuss much)

For example,

```
(one (2 (3) 4) five ("six" 7))
```

Atoms

Atoms are:

- ▶ numbers: either integers like 3 or floating point like 3.14
- ▶ strings: "like this"
- ▶ characters: #\c represents the character c.
- ▶ *symbols*: a symbol is a "word" like hello or *asb\$.

Expressions

In Lisp, everything that is not a constant is an *expression* which needs to be *evaluated*.

Here are some typical expressions:

```
(+ 1 2)
(+ 1 2 3)
(factorial 5)
(+ 3 (factorial 5))
```

The observant among you will have noticed that they look just like lists. This is one of the powerful/confusing things about Lisp.

Evaluating expressions

To evaluate an expression (function arg1 arg2 arg3), Lisp does this:

- ▶ evaluate each element of the list in order
- ▶ treat the first element of the list as a *function*, and apply it to the other elements of the list—the *arguments*.

In the example (+ 1 2 3), + is a function which takes (any number of) numbers as arguments and adds them up. Here it is being applied to 1, 2 and 3 so it will return 6.

Assuming factorial is a function which computes the factorial of a number, (factorial 5) will evaluate to 120 and (+ 3 (factorial 5)) will evaluate to 123.

Let's play with Lisp

Log in to one of the BUCS Unix machines and type

```
euscheme
```

at the prompt to enter a Lisp interpreter. You can now type Lisp expressions in and the interpreter will evaluate them. Like this:

```
user> (+ 1 2 3)
6
user> (* 4 5 6)
120
```

A Lisp session, continued

```
user> (1 2 3)
Continuable error---calling default handler:
Condition class is #<class bad-type>
message:      "incorrect type in function application"
value:        1
expected-type: #<class function>

Debug loop. Type help: for help
Broken at #<Code *TOPLEVEL*>

DEBUG>

Eek! What happened?
```

List constants

I asked Lisp to evaluate the list (1 2 3) and I got an error.

That's because Lisp wants to use 1 as a function.

But I didn't mean that. I just meant the list (1 2 3), as a constant. To get this I have to use a special function quote:

```
user> (quote (1 2 3))
(1 2 3)
user> '(1 2 3)
(1 2 3)
```

NB A shorthand for (quote something) is 'something.

List operations

To manipulate lists, we need two kinds of operator:

- ▶ A *constructor*, to let us build new lists
- ▶ Some *destructors*, to let us take lists apart to get at the elements.

The constructor is called `cons`. Can you guess what this is short for?

The destructors are called `car` and `cdr`. You will never guess what these are short for.

Cons

Let's build a list. We'll start with the empty list, which is written as `()`, and add new elements using `cons`.

`cons` takes two arguments: a thing to put in a list, and the list to put the thing in. It puts the thing in at the front. Like this:

```
user> (cons 1 ())
(1)
user> (cons 2 '(1))
(2 1)
user> (cons 3 '(2 1))
(3 2 1)
user> (cons 4 '(3 2 1))
(4 3 2 1)
```

Car and cdr

`car` (contents of address register—obscure nomenclature, ask someone old) takes a list and gives you the first element.

`cdr` (contents of decrement register) takes a list and gives you everything but the first element; this is a list too.

```
user> (car '(4 3 2 1))
4
user> (cdr '(4 3 2 1))
(3 2 1)
```

The beautiful symmetry!

It's worth noticing that there's a nice symmetry between the constructor and the destructors:

- ▶ `cons` takes a thing and a list, and gives you a new list.
- ▶ `car` and `cdr` take a list and break it up into a thing and a list.

Well-designed data structures often have this kind of symmetry.

True, false, and if

All programming languages need to be able to *test* things, such as whether two values are equal, and respond differently to the various possible answers.

In Lisp, there are no genuine “true” and “false” values. Instead, the empty list `()` is used for false, and anything else is used for true. Often the symbol `t` is used, by convention.

You can branch on the truth value of an expression using `if`:

```
(if (null l)
    what-to-do-if-l-is-the-empty-list
    what-to-do-if-l-is-not-the-empty-list
)
```

Here we're also using the built-in test `null` which checks whether a list is empty or not.

Defining functions

Most Lisp systems come with a range of pre-defined functions, like `+`, but it wouldn't be much fun to be stuck with just them.

The way we program in Lisp is to write our own functions. That's why it's called functional programming.

Functions will very often be *recursive*.

A simple function

Let's write a function which takes a list and tells you how many elements it has, that is, computes the length of the list.

It's going to be a recursive function, and it's going to follow a common pattern:

- ▶ Test whether the list is empty. If it is, return the answer straight away. (What is the answer in this case?)
- ▶ If the list is not empty, take it apart using `car` and `cdr`. Make a recursive call on the `cdr`, and use this and the `car` to compute the answer.

The len function

```
(defun len (mylist) (  
  if (null mylist)  
      0  
      (+ 1 (len (cdr mylist))))  
)
```

`defun` is short for "define a function". A `defun` looks like:

```
(defun function-name (list of arguments)  
  (body of function)  
)
```

How the len function works

It works like this:

- ▶ if the list we're measuring is null, the answer is 0.
- ▶ if not, the answer is the length of the `cdr` plus one, so we make a recursive call with `(len (cdr mylist))`, and add one to the answer to get our final answer.

Let's try it

```
user> (defun len (l) (  
  if (null l)  
      0  
      (+ 1 (len (cdr l))))  
)  
len  
user> (len '(1 2 3 4 5))  
5
```

- ▶ I type in the definition of `len`. I can put whatever whitespace I like in the definition, including line breaks.
- ▶ The interpreter tells me I've defined the function by saying "`len`" back to me.
- ▶ I can then try out the function.
- ▶ It works.

Using euscheme

Some other odds and ends you need to know to use euscheme:

- ▶ When you've finished, type `(exit)` or control-D to get out.
- ▶ When you make an error, you'll get put into the debug loop. There's lots of useful information available there, but if you just want to go back and get on with things, type `top:` to go back to normal.
- ▶ Instead of typing all your functions into the interpreter, you should put them in a file using your favourite text editor (hint: emacs), and load it into euscheme using `(load "filename")`.