Defining functions: use the special form `defun`

```
(defun name (arg1 arg2 ... argn)
   expr1
   expr2
   ...
   exprn)
```

Defining functions: use the special form `defun`

```
(defun name (arg1 arg2 ... argn)
   expr1
   expr2
   ...
   exprn)
```

When the function named by `name` is called, the `args` are set to the values of the arguments passed and the `exprs` in the body are evaluated in order; the value returned from the function call is the value of the last `exprn`

Lisp

Expressions

```
(defun factorial (n)
   (if (< n 2)
       n
       (* n (factorial (- n 1)))))
```

Lisp

Expressions

```
(defun factorial (n)
   (if (< n 2)
       n
       (* n (factorial (- n 1)))))
```

In this example we happen to have a single `expr` in the body, the `if`

```
(defun factorial (n)
  (if (< n 2)
      n
      (* n (factorial (- n 1)))))
```

In this example we happen to have a single `expr` in the body, the `if`

Generally, `defun`s should be at top level—not inside any other code—but some Lisps let you nest function definitions inside other expressions

```
(defun factorial (n)
   (if (< n 2)
       n
       (* n (factorial (- n 1)))))
```

In this example we happen to have a single `expr` in the body, the `if`

Generally, `defuns` should be at top level—not inside any other code—but some Lisps let you nest function definitions inside other expressions

There are better ways of defining local functions than using nested `defuns`

Function of no arguments

```
(defun hello ()
   (print "hi there"))
```

Function of no arguments

```
(defun hello ()
   (print "hi there"))
```

Called as `(hello)`

More Lisp

Skip to the end

Another Lisp conditional is like a generalisation of `switch` from C

```
(cond
  (test1 expr1a expr1b ...)
  (test2 expr2a expr2b ...)
  ...
  (testn exprna exprnb ...))
```

Evaluate `test1`; if true evaluate the `expr1`s in order; return the value of the last `expr1` as the value of the `cond`

Another Lisp conditional is like a generalisation of `switch` from C

```
(cond
  (test1 expr1a expr1b ...)
  (test2 expr2a expr2b ...)
  ...
  (testn exprna exprnb ...))
```

Evaluate `test1`; if true evaluate the `expr1`s in order; return the value of the last `expr1` as the value of the `cond`

Else evaluate `test2`; if true evaluate the `expr2`s in order; return the value of the last `expr2` as the value of the `cond`

Another Lisp conditional is like a generalisation of `switch` from C

```
(cond
  (test1 expr1a expr1b ...)
  (test2 expr2a expr2b ...)
  ...
  (testn exprna exprnb ...))
```

Evaluate `test1`; if true evaluate the `expr1`s in order; return the value of the last `expr1` as the value of the `cond`

Else evaluate `test2`; if true evaluate the `expr2`s in order; return the value of the last `expr2` as the value of the `cond`

Else...

Another Lisp conditional is like a generalisation of `switch` from C

```
(cond
  (test1 expr1a expr1b ...)
  (test2 expr2a expr2b ...)
  ...
  (testn exprna exprnb ...))
```

Evaluate `test1`; if true evaluate the `expr1`s in order; return the value of the last `expr1` as the value of the `cond`

Else evaluate `test2`; if true evaluate the `expr2`s in order; return the value of the last `expr2` as the value of the `cond`

Else…

If no condition was true, return `()` as the value of the `cond`

```
(cond ((> x 0) "positive")
      ((< x 0) "negative")
      (t "zero"))
```

```
(cond ((> x 0) "positive")
      ((< x 0) "negative")
      (t "zero"))
```

The value of the expression `t` is the symbol `t`, which is a true value; this is like `default` in C's `switch` `cond` is the original conditional construct in Lisp: `if` came along later

Lisp

Expressions

`cond` is the original conditional construct in Lisp: `if` came along later

Each can be defined in terms of the other

Function of variable number of arguments

```
(defun name (arg1 arg2 ... argn . restarg)
   expr1
   expr2
   ...
   exprn)
```

This takes *n* or more arguments; the first *n* are given to `arg1` to `argn`; any others are made into a list and given to the variable `restarg`

Function of variable number of arguments

```
(defun name (arg1 arg2 ... argn . restarg)
   expr1
   expr2
   ...
   exprn)
```

This takes *n* or more arguments; the first *n* are given to `arg1` to `argn`; any others are made into a list and given to the variable `restarg`

It is an error to call the function on fewer than *n* arguments

```
(defun bar (a b . c)
   (list a b c))
```

Takes two or more arguments

```
(defun bar (a b . c)
   (list a b c))
```

Takes two or more arguments

(bar 1 2 3 4) $\rightarrow$ (1 2 (3 4))

Lisp

Expressions

```
(defun bar (a b . c)
  (list a b c))
```

Takes two or more arguments

(bar 1 2 3 4) → (1 2 (3 4))

(bar 1 2) → (1 2 ())

Lisp

Expressions

```
(defun bar (a b . c)
   (list a b c))
```

Takes two or more arguments

`(bar 1 2 3 4)` → `(1 2 (3 4))`

`(bar 1 2)` → `(1 2 ())`

`(bar 1)` → error, not enough arguments

A special case:

```
(defun bar a
   a)
```

Takes zero or more arguments

A special case:

```
(defun bar a
   a)
```

Takes zero or more arguments

`(bar 1 2 3 4)` → `(1 2 3 4)`

A special case:

```
(defun bar a
   a)
```

Takes zero or more arguments

(bar 1 2 3 4) → (1 2 3 4)

(bar) → ()

A special case:

```
(defun bar a
   a)
```

Takes zero or more arguments

`(bar 1 2 3 4)` → `(1 2 3 4)`

`(bar)` → `()`

`bar` is just `list`!

Arithmetic: all the usual stuff

- `+`
- `-`
- `*`
- `/`
- `sin` etc.
- `exp` etc.
- `pow` raise to power
- etc.

Additionally the basic arithmetic operations have variable arity

- `(+)` $\rightarrow$ `0`
- `(+ 1)` $\rightarrow$ `1`
- `(+ 1 2)` $\rightarrow$ `3`
- `(+ 1 2 3)` $\rightarrow$ `6`
- `(-)` $\rightarrow$ error, not enough arguments
- `(- 1)` $\rightarrow$ `-1`
- `(- 1 2 3)` $\rightarrow$ `-4`
- `(* 1 2 3 4)` $\rightarrow$ `24`
- etc.

Exercise. What do you expect from `(*)`?

Exercise. What do you expect from `(/)`?

Exercise. What do you expect from `(/ 2)`?

Exercise. What do you expect from `(/ 2.0)`?

`open-input-file` takes a string and opens and returns a file stream for input; return `()` if the files does not exist

`open-input-file` takes a string and opens and returns a file stream for input; return `()` if the files does not exist

`open-output-file` takes a string and opens and returns a file stream for output; creates the file if it doesn't exist; truncates it if it does

`open-input-file` takes a string and opens and returns a file stream for input; return `()` if the files does not exist

`open-output-file` takes a string and opens and returns a file stream for output; creates the file if it doesn't exist; truncates it if it does

`open-update-file` takes a string and opens and returns a file stream for append; return `()` if the files does not exist

`open-input-file` takes a string and opens and returns a file stream for input; return `()` if the files does not exist

`open-output-file` takes a string and opens and returns a file stream for output; creates the file if it doesn't exist; truncates it if it does

`open-update-file` takes a string and opens and returns a file stream for append; return `()` if the files does not exist

`close-port` closes a file stream

Reading: the function `read` takes an optional input stream and reads a complete Lisp expression

Reading: the function `read` takes an optional input stream and reads a complete Lisp expression

If no input stream is given, it reads from the standard input (usually the terminal)

Reading: the function `read` takes an optional input stream and reads a complete Lisp expression

If no input stream is given, it reads from the standard input (usually the terminal)

```
(let ((x (read))
      (y (read)))
  (list x y))
```

Output: two main functions, `print` and `write`

Output: two main functions, `print` and `write`

`write` prints a value in such a way (if possible) that it can be read back in by `read`

Output: two main functions, `print` and `write`

`write` prints a value in such a way (if possible) that it can be read back in by `read`

`print` prints a value in a more human-friendly manner

Output: two main functions, `print` and `write`

`write` prints a value in such a way (if possible) that it can be read back in by `read`

`print` prints a value in a more human-friendly manner

```
(write "hello")
"hello"
(print "hello")
hello
(write cos)
#<Subr cos>
```

Lisp
I/O

Output: two main functions, `print` and `write`

`write` prints a value in such a way (if possible) that it can be read back in by `read`

`print` prints a value in a more human-friendly manner

```
(write "hello")
"hello"
(print "hello")
hello
(write cos)
#<Subr cos>
```

Both take an optional second argument of an output stream `prin` is like `print` without a newline on the end `prin` is like `print` without a newline on the end

Exercise. When typing at the Lisp interpreter

```
> (write "hello")
"hello""hello"
>
```

Why does `"hello"` appear twice?

Lisp

I/O

`prin` is like `print` without a newline on the end

Exercise. When typing at the Lisp interpreter

```
> (write "hello")
"hello""hello"
>
```

Why does `"hello"` appear twice?

There is also a `format` rather like C's

Equality test:

Equality test:

- = for numbers

Equality test:

- = for numbers
- `equal` for general objects

Equality test:

- = for numbers
- `equal` for general objects

There is much more about `equal` to come later

Lisp

Comparison

Inequality test:

- <
- <=
- >
- >=

Lisp

Comparison

Inequality test:

- <
- <=
- >
- >=

These are all *n*-ary: `(< 1 2 3 4)` returns true if the values are strictly increasing

Lisp
Comparison

Inequality test:

- `<`
- `<=`
- `>`
- `>=`

These are all *n*-ary: `(< 1 2 3 4)` returns true if the values are strictly increasing

Similarly for the others

Lisp

Local Functions

Just like `let` introduces local variables, the `labels` special form can introduce local functions

```
(labels ((name1 (arg1a arg1b ...)
            expr1a expr1b ...)
         (name2 (arg2a arg2b ...)
            expr1a expr1b ...)
         ...
         (namen (argna argnb ...)
            exprna exprnb ...))
  body1
  body2
  ...
  bodym)
```

Local Functions

```
(labels ((name1 (arg1a arg1b ...)
            expr1a expr1b ...)
         (name2 (arg2a arg2b ...)
            expr1a expr1b ...)
         ...
         (namen (argna argnb ...)
            exprna exprnb ...))
  body1
  body2
  ...
  bodym)
```

This makes functions named `names` with arguments `args` and bodies `exprs` available in the body of the `labels`; the value of the `labels` is the value of the last `bodym`

Lisp

Local Functions

```
(labels ((foo (a b) (+ a b))
         (bar (n) (* n n)))
  (foo (bar 1) (bar 2)))
```

Lisp

Local Functions

```
(labels ((foo (a b) (+ a b))
         (bar (n) (* n n)))
  (foo (bar 1) (bar 2)))
```

As with `let` the names `foo` and `bar` revert at the exit of the `labels` form

Lisp

Local Functions

```
(labels ((foo (a b) (+ a b))
         (bar (n) (* n n)))
  (foo (bar 1) (bar 2)))
```

As with `let` the names `foo` and `bar` revert at the exit of the `labels` form

This is not quite like `let`, as within the definition of `foo` we can refer to `bar`, and vice versa

Lisp

Local Functions

```
(labels ((foo (a b) (+ a b))
         (bar (n) (* n n)))
  (foo (bar 1) (bar 2)))
```

As with `let` the names `foo` and `bar` revert at the exit of the `labels` form

This is not quite like `let`, as within the definition of `foo` we can refer to `bar`, and vice versa

And to themselves, too

```
(labels ((foo (a b) (+ a b))
         (bar (n) (* n n)))
  (foo (bar 1) (bar 2)))
```

As with `let` the names `foo` and `bar` revert at the exit of the `labels` form

This is not quite like `let`, as within the definition of `foo` we can refer to `bar`, and vice versa

And to themselves, too

It is this way by default because we naturally want functions to refer to each other, and to themselves

Lisp

Local Functions

```
(labels ((fact (n) (if (< n 2)
                       1
                       (* n (fact (- n 1))))))
   (fact 5))
```

Lisp

Local Functions

```
(labels ((fact (n) (if (< n 2)
                       1
                       (* n (fact (- n 1))))))
   (fact 5))
```

In Lisp, functions are just like other objects and you should not be shy of local functions

You will make errors

You will make errors

When this happens Lisp calls an *error handler*

You will make errors

When this happens Lisp calls an *error handler*

Error handlers are programmable, but the default handler is usually what we need

You will make errors

When this happens Lisp calls an *error handler*

Error handlers are programmable, but the default handler is usually what we need

The default handler enters a *debug loop*

```
user> qwerty
Continuable error---calling default handler:
Condition class is #<class unbound-error>
message:        "variable unbound in module 'user'"
value:          qwerty

Debug loop.  Type help: for help
Broken at #<Code #1bb6c320>

DEBUG>
```

Firstly, it tells you the problem:

```
Condition class is #<class unbound-error>
message:           "variable unbound in module 'user'"
value:             qwerty
```

Firstly, it tells you the problem:

```
Condition class is #<class unbound-error>
message:          "variable unbound in module 'user'"
value:            qwerty
```

The message tells us the variable `qwerty` is unbound, i.e., has no value

Firstly, it tells you the problem:

```
Condition class is #<class unbound-error>
message:          "variable unbound in module 'user'"
value:            qwerty
```

The message tells us the variable `qwerty` is unbound, i.e., has no value

The error class is `unbound-error`

In EuLisp errors and (error handlers) are first class objects and fit into the class hierarchy as part of a general *condition* mechanism

In EuLisp errors and (error handlers) are first class objects and fit into the class hierarchy as part of a general *condition* mechanism

There are various classes of error and we can define methods that do whatever we want dependent on the type

In EuLisp errors and (error handlers) are first class objects and fit into the class hierarchy as part of a general *condition* mechanism

There are various classes of error and we can define methods that do whatever we want dependent on the type

For now, just read the message

Next,

```
Debug loop.  Type help: for help
Broken at #<Code #1bb6c320>
```

We are in a debug loop, halted inside the broken code: here the code is not too useful, at other times it can identify the function where the error happened

Next,

```
Debug loop.  Type help: for help
Broken at #<Code #1bb6c320>
```

We are in a debug loop, halted inside the broken code: here the code is not too useful, at other times it can identify the function where the error happened

Typing `help:` at the prompt will give help!

```
DEBUG> help:
Debug loop.
top:                            return to top level
resume:  or  (resume: val)      resume from error
bt:                             backtrace
locals:                         local variables
cond:                           current condition
up:  or  (up: n)                up one or n frames
down:  or  (down: n)            down one or n frames
where:                          current function

DEBUG>
```

- `top:` this will throw away the error and return us to the top level read-eval-print loop

Lisp

Errors

- `top:` this will throw away the error and return us to the top level read-eval-print loop
- `resume:` this will continue running the code from where it stopped, passing in a value (or `()`)

- `top:` this will throw away the error and return us to the top level read-eval-print loop
- `resume:` this will continue running the code from where it stopped, passing in a value (or `()`)
- `bt:` will give a list of the function call frames we are inside (a *backtrace*); (due to tail recursion some frames may be absent)

- `top:` this will throw away the error and return us to the top level read-eval-print loop
- `resume:` this will continue running the code from where it stopped, passing in a value (or `()`)
- `bt:` will give a list of the function call frames we are inside (a *backtrace*); (due to tail recursion some frames may be absent)
- `local:` the values of the local variables

- `top:` this will throw away the error and return us to the top level read-eval-print loop
- `resume:` this will continue running the code from where it stopped, passing in a value (or `()`)
- `bt:` will give a list of the function call frames we are inside (a *backtrace*); (due to tail recursion some frames may be absent)
- `local:` the values of the local variables
- `cond:` the current error condition (as given in the error message)

- `up:` move up one frame: if `foo` calls `bar` and we broke in `bar`, this move us up into `foo`

- `up:` move up one frame: if `foo` calls `bar` and we broke in `bar`, this move us up into `foo`
- `down:` move down one frame

- `up:` move up one frame: if `foo` calls `bar` and we broke in `bar`, this move us up into `foo`
- `down:` move down one frame
- `where:` the name of the function we broke in, if available

Usually, we do a `bt:` to see where we are and then a `top:` to clean up the error before we try again

Usually, we do a `bt:` to see where we are and then a `top:` to clean up the error before we try again

Debug loops can be nested if we make an error while in a debug loop

Lisp

Errors

```
(defun foo (n)
   (+ 1 (bar n)))

(defun bar (m)
   (/ 1 m))

(foo 0)
```

Lisp

Errors

```
Continuable error---calling default handler:
Condition class is #<class arithmetic-error>
message:          "division by zero"
value:            1

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

DEBUG>
```

```
DEBUG> bt:

Stack backtrace:

function bar (m)
m:                 0

function foo (n)
n:                 0

function *TOPLEVEL* ()

function *TOPLEVEL* ()

DEBUG> top:
```

Or

```
DEBUG> (resume: 5)
6
```

Or

```
DEBUG> (resume: 5)
6
```

We exit the debug loop, passing back the value `5` from where the error occurred; `foo` then adds `1`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`
- `(consp x)` $\rightarrow$ `t` if `x` is a pair; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`
- `(consp x)` $\rightarrow$ `t` if `x` is a pair; else `()`
- `(listp x)` $\rightarrow$ `t` if `x` is a list; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`
- `(consp x)` $\rightarrow$ `t` if `x` is a pair; else `()`
- `(listp x)` $\rightarrow$ `t` if `x` is a list; else `()`
- `(stringp x)` $\rightarrow$ `t` if `x` is a string; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`
- `(consp x)` $\rightarrow$ `t` if `x` is a pair; else `()`
- `(listp x)` $\rightarrow$ `t` if `x` is a list; else `()`
- `(stringp x)` $\rightarrow$ `t` if `x` is a string; else `()`
- `(numberp x)` $\rightarrow$ `t` if `x` is a number; else `()`

Lisp

Type Tests

- `(null x)` → `t` if `x` is the empty list; else `()`
- `(atom x)` → `t` if `x` is an atom; else `()`
- `(consp x)` → `t` if `x` is a pair; else `()`
- `(listp x)` → `t` if `x` is a list; else `()`
- `(stringp x)` → `t` if `x` is a string; else `()`
- `(numberp x)` → `t` if `x` is a number; else `()`
- `(integerp x)` → `t` if `x` is an integer; else `()`

Lisp

Type Tests

- `(null x)` $\rightarrow$ `t` if `x` is the empty list; else `()`
- `(atom x)` $\rightarrow$ `t` if `x` is an atom; else `()`
- `(consp x)` $\rightarrow$ `t` if `x` is a pair; else `()`
- `(listp x)` $\rightarrow$ `t` if `x` is a list; else `()`
- `(stringp x)` $\rightarrow$ `t` if `x` is a string; else `()`
- `(numberp x)` $\rightarrow$ `t` if `x` is a number; else `()`
- `(integerp x)` $\rightarrow$ `t` if `x` is an integer; else `()`
- `(functionp x)` $\rightarrow$ `t` if `x` is a function; else `()`

Lisp

Type Tests

- `(null x)` → `t` if `x` is the empty list; else `()`
- `(atom x)` → `t` if `x` is an atom; else `()`
- `(consp x)` → `t` if `x` is a pair; else `()`
- `(listp x)` → `t` if `x` is a list; else `()`
- `(stringp x)` → `t` if `x` is a string; else `()`
- `(numberp x)` → `t` if `x` is a number; else `()`
- `(integerp x)` → `t` if `x` is an integer; else `()`
- `(functionp x)` → `t` if `x` is a function; else `()`
- etc.

Lisp

Type Tests

- `(null x)` → `t` if `x` is the empty list; else `()`
- `(atom x)` → `t` if `x` is an atom; else `()`
- `(consp x)` → `t` if `x` is a pair; else `()`
- `(listp x)` → `t` if `x` is a list; else `()`
- `(stringp x)` → `t` if `x` is a string; else `()`
- `(numberp x)` → `t` if `x` is a number; else `()`
- `(integerp x)` → `t` if `x` is an integer; else `()`
- `(functionp x)` → `t` if `x` is a function; else `()`
- etc.

"p" is for "predicate"; Scheme uses `?`, so `cons?`, etc.

Exercise. `listp` is different from `consp`. Explain

Exercise. What is `(atom #(1 2))`?

Exercise. Compare `not` and `null`

`cons`, `car`, `cdr`, `list`

`cons`, `car`, `cdr`, `list`

- `length` of a list

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`
- `cadr` same as `(car (cdr l))`

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`
- `cadr` same as `(car (cdr l))`
- `cdar` same as `(cdr (car l))`

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`
- `cadr` same as `(car (cdr l))`
- `cdar` same as `(cdr (car l))`
- `cddr` same as `(cdr (cdr l))`

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`
- `cadr` same as `(car (cdr l))`
- `cdar` same as `(cdr (car l))`
- `cddr` same as `(cdr (cdr l))`
- ...

`cons`, `car`, `cdr`, `list`

- `length` of a list
- `caar` same as `(car (car l))`
- `cadr` same as `(car (cdr l))`
- `cdar` same as `(cdr (car l))`
- `cddr` same as `(cdr (cdr l))`
- ...
- `cddddr` same as `(cdr (cdr (cdr (cdr l))))`

`(append l1 l2)` appends `l2` to the end of list `l1`

`(append l1 l2)` appends `l2` to the end of list `l1`

More precisely: it makes a new list that starts as `l1` and continues with `l2`

`(append l1 l2)` appends `l2` to the end of list `l1`

More precisely: it makes a new list that starts as `l1` and continues with `l2`

This is different from `cons`

`(append l1 l2)` appends `l2` to the end of list `l1`

More precisely: it makes a new list that starts as `l1` and continues with `l2`

This is different from `cons`

`(append '(1 2) '(3 4))` → `(1 2 3 4)`

`(append l1 l2)` appends `l2` to the end of list `l1`

More precisely: it makes a new list that starts as `l1` and continues with `l2`

This is different from `cons`

`(append '(1 2) '(3 4))` → `(1 2 3 4)`

`(cons '(1 2) '(3 4))` → `((1 2) 3 4)`

`(append l1 l2)` appends `l2` to the end of list `l1`

More precisely: it makes a new list that starts as `l1` and continues with `l2`

This is different from `cons`

`(append '(1 2) '(3 4))` → `(1 2 3 4)`

`(cons '(1 2) '(3 4))` → `((1 2) 3 4)`

Make sure you understand what is happening here. You will get this wrong!

Note for future reference: `append` *copies* the first argument and *shares* the second argument

Note for future reference: `append` *copies* the first argument and *shares* the second argument

Also: consider

```
x → (a b)
(append x '(c d)) → (a b c d)
x → (a b)
```

Note for future reference: `append` *copies* the first argument and *shares* the second argument

Also: consider

```
x → (a b)
(append x '(c d)) → (a b c d)
x → (a b)
```

Note: appending to (the list referred to by) `x` does not change the value of `x` or the list referred to by `x`, it makes a new list `(a b c d)`

Note for future reference: `append` *copies* the first argument and *shares* the second argument

Also: consider

```
x → (a b)
(append x '(c d)) → (a b c d)
x → (a b)
```

Note: appending to (the list referred to by) `x` does not change the value of `x` or the list referred to by `x`, it makes a new list `(a b c d)`

Functions like `cons`, `list` and `append` never modify an existing value; they always make a new one

Lisp

Expressions

Enough of Lisp basics for now: there is lots more, including generic functions and (multi)methods; setters; converters; string operations; maps; continuations; hash tables; macros; threads; modules

Lisp

Expressions

Enough of Lisp basics for now: there is lots more, including generic functions and (multi)methods; setters; converters; string operations; maps; continuations; hash tables; macros; threads; modules

See
`http://people.bath.ac.uk/masrjb/Sources/eunotes.html`
(link on my unit web page) for much more

Lisp

Expressions

Enough of Lisp basics for now: there is lots more, including generic functions and (multi)methods; setters; converters; string operations; maps; continuations; hash tables; macros; threads; modules

See
`http://people.bath.ac.uk/masrjb/Sources/eunotes.html`
(link on my unit web page) for much more

Use `(load "file")` to load a Lisp file named `"file"`

Lisp

Expressions

Enough of Lisp basics for now: there is lots more, including generic functions and (multi)methods; setters; converters; string operations; maps; continuations; hash tables; macros; threads; modules

See
`http://people.bath.ac.uk/masrjb/Sources/eunotes.html`
(link on my unit web page) for much more

Use `(load "file")` to load a Lisp file named `"file"`

There is a list of simple Lisp exercises on the Unit web page: you *must* try them otherwise you will completely be unable to do the coursework properly

Lisp

Expressions

Enough of Lisp basics for now: there is lots more, including generic functions and (multi)methods; setters; converters; string operations; maps; continuations; hash tables; macros; threads; modules

See
`http://people.bath.ac.uk/masrjb/Sources/eunotes.html`
(link on my unit web page) for much more

Use `(load "file")` to load a Lisp file named `"file"`

There is a list of simple Lisp exercises on the Unit web page: you *must* try them otherwise you will completely be unable to do the coursework properly

The best way to learn a language is to use it!