Lisp

Mapping

Iteration (`for` loops) is fine for linear datastructures, like vectors of lists, but it does not generalise to more complicated structures, e.g., trees

Lisp

Mapping

Iteration (`for` loops) is fine for linear datastructures, like vectors of lists, but it does not generalise to more complicated structures, e.g., trees

Recursion is an excellent way of going through a datastructure, particularly non-linear ones

Iteration (`for` loops) is fine for linear datastructures, like vectors of lists, but it does not generalise to more complicated structures, e.g., trees

Recursion is an excellent way of going through a datastructure, particularly non-linear ones

But having higher order functions allows us another way

Lisp

Mapping

Iteration (`for` loops) is fine for linear datastructures, like vectors of lists, but it does not generalise to more complicated structures, e.g., trees

Recursion is an excellent way of going through a datastructure, particularly non-linear ones

But having higher order functions allows us another way

And this way is somehow closer to the way we naturally think

Lisp

Mapping

Iteration (`for` loops) is fine for linear datastructures, like vectors of lists, but it does not generalise to more complicated structures, e.g., trees

Recursion is an excellent way of going through a datastructure, particularly non-linear ones

But having higher order functions allows us another way

And this way is somehow closer to the way we naturally think

“Do this operation on every element of this datastructure”

Suppose we want to add 1 to every value in a list:

```
(defun add1 (l)
   (if (null l)
       ()
       (cons (+ 1 (car l)) (add1 (cdr l)))))
```

Suppose we want subtract 2 from every value in a list:

```
(defun sub2 (l)
   (if (null l)
       ()
       (cons (- (car l) 2) (sub2 (cdr l)))))
```

Suppose we want to square every value in a list:

```
(defun sq (l)
   (if (null l)
       ()
       (cons (* (car l) (car l)) (sq (cdr l)))))
```

Recursive functions are nice and simple, but we can see we are re-writing the same code many times

Recursive functions are nice and simple, but we can see we are re-writing the same code many times

Can we abstract this out and write the common code just once?

Recursive functions are nice and simple, but we can see we are re-writing the same code many times

Can we abstract this out and write the common code just once?

Higher order functions will allow us to do this

There are a few Lisp functions we need to look at: `do`, `map` `accumulate`. We start with `do`

There are a few Lisp functions we need to look at: `do`, `map` `accumulate`. We start with `do`

This function applies some operation to each member of (for now) a list

There are a few Lisp functions we need to look at: `do`, `map` `accumulate`. We start with `do`

This function applies some operation to each member of (for now) a list

`(do print '(a (b c) d))` prints

```
a
(b c)
d
```

There are a few Lisp functions we need to look at: `do`, `map` `accumulate`. We start with `do`

This function applies some operation to each member of (for now) a list

`(do print '(a (b c) d))` prints

```
a
(b c)
d
```

(and returns `()` as its value)

So `do` is a *higher order function*, namely it takes a function as an argument

So `do` is a *higher order function*, namely it takes a function as an argument

So this is (one) conceptually simpler way of doing loops: there's no index variable when it's not needed

So `do` is a *higher order function*, namely it takes a function as an argument

So this is (one) conceptually simpler way of doing loops: there's no index variable when it's not needed

`map` is similar, but it makes a list of the results of doing the operation

So `do` is a *higher order function*, namely it takes a function as an argument

So this is (one) conceptually simpler way of doing loops: there's no index variable when it's not needed

`map` is similar, but it makes a list of the results of doing the operation

`(map symbolp '(1 a (b c)))` → `(() t ())`

Lisp

Mapping

To increment values in a list we can

```
(defun inc (n) (+ n 1))
```

```
(map inc '(1 2 3)) → (2 3 4)
```

To increment values in a list we can

```
(defun inc (n) (+ n 1))
```

```
(map inc '(1 2 3)) → (2 3 4)
```

This is very simple coding: we have a function `inc` that encodes what we want to do, and `map` hides the fiddly stuff of applying it to each element of the list

Lisp
Mapping

To increment values in a list we can

```
(defun inc (n) (+ n 1))
```

```
(map inc '(1 2 3)) → (2 3 4)
```

This is very simple coding: we have a function `inc` that encodes what we want to do, and `map` hides the fiddly stuff of applying it to each element of the list

Even better, we can simplify this a bit more

Lisp

Mapping

To increment values in a list we can

```
(defun inc (n) (+ n 1))
```

```
(map inc '(1 2 3)) → (2 3 4)
```

This is very simple coding: we have a function `inc` that encodes what we want to do, and `map` hides the fiddly stuff of applying it to each element of the list

Even better, we can simplify this a bit more

Defining a function just to use it once in such a construct is a bit of overkill

What we want to do is write

(map *a-function-that-increments* '(1 2 3))

we are not interested in giving the increment function any particular importance, such as a name that might clash with a name elsewhere

Lisp
Mapping

What we want to do is write

```
(map a-function-that-increments '(1 2 3))
```

we are not interested in giving the increment function any particular importance, such as a name that might clash with a name elsewhere

Just like, if we wanted to use the number 7 once, we don't want to have to assign the value to a variable and use the variable

What we want to do is write

`(map` *a-function-that-increments* `'(1 2 3))`

we are not interested in giving the increment function any particular importance, such as a name that might clash with a name elsewhere

Just like, if we wanted to use the number 7 once, we don't want to have to assign the value to a variable and use the variable

We just write "7". In a similar way, we want to just write "a function"

Lisp allows us to define and use *anonymous* functions; more commonly called *lambdas*

Lisp allows us to define and use *anonymous* functions; more commonly called *lambdas*

Just like writing ”`"cat"`” for a string with no name (variable) required, we can write something for a function

Lisp

Lambda

`(lambda (n) (+ n 1))` denotes a function that takes one argument and returns one more than that argument

Lisp

Lambda

`(lambda (n) (+ n 1))` denotes a function that takes one argument and returns one more than that argument

It doesn't have a name: it just is

Lisp

Lambda

`(lambda (n) (+ n 1))` denotes a function that takes one argument and returns one more than that argument

It doesn't have a name: it just is

The lambda says "I am a function", just like quotes say "I am a string". Just notation

Lisp

Lambda

`(lambda (n) (+ n 1))` denotes a function that takes one argument and returns one more than that argument

It doesn't have a name: it just is

The lambda says "I am a function", just like quotes say "I am a string". Just notation

"lambda" comes from the history of Lisp: McCarthy's Lisp was to be an implementation of the Lambda Calculus `(lambda (n) (+ n 1))` denotes a function that takes one argument and returns one more than that argument

It doesn't have a name: it just is

The lambda says "I am a function", just like quotes say "I am a string". Just notation

"lambda" comes from the history of Lisp: McCarthy's Lisp was to be an implementation of the Lambda Calculus

There's not much we can do in terms of manipulating functions (function composition?), its main use is when we apply it to some arguments

Lisp

Lambda

```
((lambda (n) (+ n 1)) 5)
```

$\rightarrow$

6

Works for both Lisp-1 and Lisp-2

```
((lambda (n) (+ n 1)) 5)
```

$\rightarrow$

6

Works for both Lisp-1 and Lisp-2

Lisp-2s have this as an exception to the rule that the first thing after the parenthesis must be a symbol that names a function

Lisp

Lambda

```
((lambda (n) (+ n 1)) 5)
```

$\rightarrow$

6

Works for both Lisp-1 and Lisp-2

Lisp-2s have this as an exception to the rule that the first thing after the parenthesis must be a symbol that names a function

For Lisp-1s this is entirely natural

Lisp

Lambda

```
((lambda (n) (+ n 1)) 5)
```

$\rightarrow$

6

Works for both Lisp-1 and Lisp-2

Lisp-2s have this as an exception to the rule that the first thing after the parenthesis must be a symbol that names a function

For Lisp-1s this is entirely natural

Rather than writing down the name of a function that adds 1, simply write down a function that adds 1

Lisp

Lambda

Functions are first class objects in Lisp, so just as we have syntax for writing down numbers “`42`” and strings “`"hello"`” we have syntax for writing down functions

Functions are first class objects in Lisp, so just as we have syntax for writing down numbers “`42`” and strings “`"hello"`” we have syntax for writing down functions

Functions in other languages are often inextricable from their names

Lisp

Lambda

Functions are first class objects in Lisp, so just as we have syntax for writing down numbers “`42`” and strings “`"hello"`” we have syntax for writing down functions

Functions in other languages are often inextricable from their names

Though “modern” languages are increasingly incorporating lambdas, e.g., Python, Java, JavaScript, C++, etc.

Note the string `"hello"` doesn't have a name, unless we assign it to a variable; then that variable is a name we can use to refer to the string

Note the string `"hello"` doesn't have a name, unless we assign it to a variable; then that variable is a name we can use to refer to the string

The function `(lambda ...)` doesn't have a name, unless we assign it to a variable; then that variable is a name we can use to refer to the function

Note the string `"hello"` doesn't have a name, unless we assign it to a variable; then that variable is a name we can use to refer to the string

The function `(lambda ...)` doesn't have a name, unless we assign it to a variable; then that variable is a name we can use to refer to the function

But it is very common to be lazy and say "the function `sin`" rather than "the function named by `sin`"

Make sure you are clear on this point: distinguish between objects and names of objects

Make sure you are clear on this point: distinguish between objects and names of objects

While `sin` might name a function that computes the sine, the function itself is something that is hard to write down

Make sure you are clear on this point: distinguish between objects and names of objects

While `sin` might name a function that computes the sine, the function itself is something that is hard to write down

Many objects (like lambdas) don't have names: this is why they are called *anonymous*

Make sure you are clear on this point: distinguish between objects and names of objects

While `sin` might name a function that computes the sine, the function itself is something that is hard to write down

Many objects (like lambdas) don't have names: this is why they are called *anonymous*

It is easy for objects to have multiple names: assign the same value to more than one variable. E.g., (the function named by) `not` and `null`

Note that in Lisp, because we have symbols as a datatype, names can have names

```
(let ((x 'y))
   ... x ...)
```

Within the `let` the symbol `y` has the name `x`

Note that in Lisp, because we have symbols as a datatype, names can have names

```
(let ((x 'y))
   ... x ...)
```

Within the `let` the symbol `y` has the name `x`

Exercise. Read “Through the Looking-Glass” by Lewis Carroll, in particular the section discussing the poem “Haddocks’ Eyes”

In Euscheme:

```
(lambda (n) (+ n 1))
->
#<Procedure #80d63e4>
```

In Euscheme:

```
(lambda (n) (+ n 1))
->
#<Procedure #80d63e4>
```

The funny way of printing this value is just a way of saying "some procedure", i.e., function; in this case the number is actually a memory location, but that's coincidental and not important

In Clisp

```
(lambda (n) (+ n 1))
->
#<FUNCTION :LAMBDA (N) (+ N 1)>
```

As an interpreted function; compiled functions are less descriptive

In Clisp

```
(lambda (n) (+ n 1))
->
#<FUNCTION :LAMBDA (N) (+ N 1)>
```

As an interpreted function; compiled functions are less descriptive

```
#'sin
->
#<SYSTEM-FUNCTION SIN>
```

In Clisp

```
(lambda (n) (+ n 1))
->
#<FUNCTION :LAMBDA (N) (+ N 1)>
```

As an interpreted function; compiled functions are less descriptive

```
#'sin
->
#<SYSTEM-FUNCTION SIN>
```

There is no simple, succinct way of printing out arbitrary functions, so most systems don't try too hard

Once we realise functions are just like every other object, the world becomes much simpler

Lisp

Lambda

Once we realise functions are just like every other object, the world becomes much simpler

`defun` is simply shorthand for “make a lambda of the body and then assign it to the name”

```
(defun inc (n) (+ n 1))
```

becomes the lambda

```
(lambda (n) (+ n 1))
```

which gets assigned to the name `inc`

Once we realise functions are just like every other object, the world becomes much simpler

`defun` is simply shorthand for “make a lambda of the body and then assign it to the name”

```
(defun inc (n) (+ n 1))
```

becomes the lambda

```
(lambda (n) (+ n 1))
```

which gets assigned to the name `inc`

We haven’t looked at assigning to variables yet, though

And `let` is itself just another `lambda`!

```
(let ((n 2) (m (foo 4)))
  (print "hello") (* n m))
```

is just

```
((lambda (n m) (print "hello") (* n m))
    2 (foo 4))
```

So we see that apparently diverse constructs are simply variants on one simple concept, the lambda

So we see that apparently diverse constructs are simply variants on one simple concept, the lambda

Very much the spirit of Scheme

Back to mapping: lambdas are very useful here

Back to mapping: lambdas are very useful here

The idea "take a list of numbers and return a list of incremented values" becomes

```
(map (lambda (n) (+ n 1)) '(1 2 3))
→
(2 3 4)
```

Back to mapping: lambdas are very useful here

The idea “take a list of numbers and return a list of incremented values” becomes

```
(map (lambda (n) (+ n 1)) '(1 2 3))
→
(2 3 4)
```

Everything is simple and in front of us: the function to increment; the list of numbers; and `map` to apply it to the list

Back to mapping: lambdas are very useful here

The idea "take a list of numbers and return a list of incremented values" becomes

```
(map (lambda (n) (+ n 1)) '(1 2 3))
→
(2 3 4)
```

Everything is simple and in front of us: the function to increment; the list of numbers; and `map` to apply it to the list

No loops or loop variables to confuse what is happening

In fact `map` and `do` are a lot cleverer than this

```
(map + '(1 2) '(3 4))
```
→
```
(4 6)
```

```
(do (lambda (x y) (print (cons x y))) "qwe"  "asd")
prints
(q . a)
(w . s)
(e . d)
```

mapping along the characters of the strings

Common Lisp: `map` requires the type of the result as an argument:

```
(map 'list (lambda (n) (+ n 1)) '(2 3 4))
->
(3 4 5)

(map 'vector (lambda (n) (+ n 1)) '(2 3 4))
->
#(3 4 5)
```

Common Lisp: `map` requires the type of the result as an argument:

```
(map 'list (lambda (n) (+ n 1)) '(2 3 4))
->
(3 4 5)

(map 'vector (lambda (n) (+ n 1)) '(2 3 4))
->
#(3 4 5)
```

`mapcar` is the name of what we have called `map` (but only for lists), while `mapc` is close to the `do` function (returns the original list)

Common Lisp: `map` requires the type of the result as an argument:

```
(map 'list (lambda (n) (+ n 1)) '(2 3 4))
->
(3 4 5)

(map 'vector (lambda (n) (+ n 1)) '(2 3 4))
->
#(3 4 5)
```

`mapcar` is the name of what we have called `map` (but only for lists), while `mapc` is close to the `do` function (returns the original list)

Exercise: investigate CL's `mapl` and `maplist`

Exercise. `map` and friends are generally not primitives in Lisp as they are easy to define for yourself. Do so (for a simple, single argument, list-based `map`)

Exercise. `map` and friends are generally not primitives in Lisp as they are easy to define for yourself. Do so (for a simple, single argument, list-based `map`)

Notice what you are doing is abstracting out the code to do a traversal of a list and making it reusable

Exercise. `map` and friends are generally not primitives in Lisp as they are easy to define for yourself. Do so (for a simple, single argument, list-based `map`)

Notice what you are doing is abstracting out the code to do a traversal of a list and making it reusable

Exercise. Then `do`, `maplist` and so on

Exercise. We might implement a *tree* as

- empty `()`
- or a value and two subtrees `(val ltree rtree)`

Write a function `(dotree fn tree)` that takes a function `fn` and applies it to each value in the tree

Exercise. Write a function `(maptree fn tree)` that takes a function `fn` and applies it to each value in the tree and returns the new tree

A related operation is `accumulate`, often called *reduce* in other contexts

A related operation is `accumulate`, often called *reduce* in other contexts

“Add up the numbers in this list”

A related operation is `accumulate`, often called *reduce* in other contexts

"Add up the numbers in this list"

```
(accumulate + 0 '(1 2 3 4))
```

A related operation is `accumulate`, often called *reduce* in other contexts

“Add up the numbers in this list”

```
(accumulate + 0 '(1 2 3 4))
```

An operation; an initial value; the list: this computes $0 + 1 + 2 + 3 + 4$

```
(accumulate * 1 '(1 2 3 4))
```

$\rightarrow$

```
24
```

Suppose a function named `(mklist n)` makes a list of integers 1 to *n*: `(mklist 4)` $\rightarrow$ `(1 2 3 4)`

Suppose a function named `(mklist n)` makes a list of integers 1 to *n*: `(mklist 4)` $\rightarrow$ `(1 2 3 4)`

```
(defun factorial (n)
  (accumulate * 1 (mklist n)))
```

is a fairly inefficient factorial

Suppose a function named `(mklist n)` makes a list of integers 1 to *n*: `(mklist 4)` $\rightarrow$ `(1 2 3 4)`

```
(defun factorial (n)
  (accumulate * 1 (mklist n)))
```

is a fairly inefficient factorial

Exercise. Define such a `mklist`

`accumulate` is more commonly seen as `reduce`

`(reduce + '(1 2 3 4))` $\rightarrow$ `10`

`accumulate` is more commonly seen as `reduce`

`(reduce + '(1 2 3 4))` $\rightarrow$ `10`

An operation; the list: this computes
$1 + 2 + 3 + 4$

`accumulate` is more commonly seen as `reduce`

`(reduce + '(1 2 3 4))` $\rightarrow$ `10`

An operation; the list: this computes
$1 + 2 + 3 + 4$

`(reduce - '(1 2 3 4))` is $1 - 2 - 3 - 4 = -8$

`accumulate` is more commonly seen as `reduce`

`(reduce + '(1 2 3 4))` $\rightarrow$ `10`

An operation; the list: this computes
$1 + 2 + 3 + 4$

`(reduce - '(1 2 3 4))` is $1 - 2 - 3 - 4 = -8$

`(accumulate - 0 '(1 2 3 4))` is $0 - 1 - 2 - 3 - 4 = -10$

We may define

```
(defun reduce (op vals)
  (if (null vals)
      (op)              ; sometimes gives a default value
      (accumulate op (car vals) (cdr vals))))
```

We may define

```
(defun reduce (op vals)
  (if (null vals)
      (op)              ; sometimes gives a default value
      (accumulate op (car vals) (cdr vals))))
```

Not a perfect translation: `accumulate` is a bit clearer on values for edge cases

Lisp

Mapping

The functions `map` and `accumulate` reflect the functional style

Lisp

Mapping

The functions `map` and `accumulate` reflect the functional style

- regard the datastructure as a whole

The functions `map` and `accumulate` reflect the functional style

- regard the datastructure as a whole
- separate the operation being applied from the act of application: i.e., the traversal of the datastructure

The functions `map` and `accumulate` reflect the functional style

- regard the datastructure as a whole
- separate the operation being applied from the act of application: i.e., the traversal of the datastructure

We can change the datastructure, e.g., replace a vector by a list, and (as long as `map` understands how to traverse it) use the same code unchanged

The functions `map` and `accumulate` reflect the functional style

- regard the datastructure as a whole
- separate the operation being applied from the act of application: i.e., the traversal of the datastructure

We can change the datastructure, e.g., replace a vector by a list, and (as long as `map` understands how to traverse it) use the same code unchanged

We can write the traversal of the new datastructure just once and ensure `map` knows how to use it; then every application of whatever operation simply works

To reiterate: by separating the traversal of a datastructure from the operation on the elements of the datastructure we are allowing a greater flexibility

To reiterate: by separating the traversal of a datastructure from the operation on the elements of the datastructure we are allowing a greater flexibility

If we have written our code using mapping functions and decide to change the datastructure our program is using, we need only to write new traversal code for the new datastructure: the code that does stuff to the datastructure remains unchanged

Lisp

Mapping

To reiterate: by separating the traversal of a datastructure from the operation on the elements of the datastructure we are allowing a greater flexibility

If we have written our code using mapping functions and decide to change the datastructure our program is using, we need only to write new traversal code for the new datastructure: the code that does stuff to the datastructure remains unchanged

Much easier than going through all the program and changing how each individual access to the datastructure is coded

To reiterate: by separating the traversal of a datastructure from the operation on the elements of the datastructure we are allowing a greater flexibility

If we have written our code using mapping functions and decide to change the datastructure our program is using, we need only to write new traversal code for the new datastructure: the code that does stuff to the datastructure remains unchanged

Much easier than going through all the program and changing how each individual access to the datastructure is coded

Maybe having to modify a `for` loop for every time we go through a vector

Here is another thing that Lisp makes explicit while other languages ignore, thus encouraging certain kinds of error

Here is another thing that Lisp makes explicit while other languages ignore, thus encouraging certain kinds of error

What does

```
n = 2;
```

mean in C/C++/Java etc.?

Lisp
Assignment and Binding

Here is another thing that Lisp makes explicit while other languages ignore, thus encouraging certain kinds of error

What does

```
n = 2;
```

mean in C/C++/Java etc.?

The quick answer is "`n` gets the value 2"

Here is another thing that Lisp makes explicit while other languages ignore, thus encouraging certain kinds of error

What does

```
n = 2;
```

mean in C/C++/Java etc.?

The quick answer is "`n` gets the value 2"

The correct answer is much longer

It depends on the context

It depends on the context

In

```
{ int n = 2;
  ...
}
```

it is a declaration and initialisation of a local variable in a block

In

```
{ ...
  n = 2;
  ...
}
```

it is an update of a variable

Though they look pretty much the same in C, two very different things are happening

Lisp

Assignment and Binding

Though they look pretty much the same in C, two very different things are happening

The first, called *binding* in Lisp, makes a new local variable `n` and gives it the value 2

Though they look pretty much the same in C, two very different things are happening

The first, called *binding* in Lisp, makes a new local variable `n` and gives it the value 2

Any existing variable `n` is unaffected

Though they look pretty much the same in C, two very different things are happening

The first, called *binding* in Lisp, makes a new local variable `n` and gives it the value 2

Any existing variable `n` is unaffected

Lisp writes `(let ((n 2)) ...)`

Though they look pretty much the same in C, two very different things are happening

The first, called *binding* in Lisp, makes a new local variable `n` and gives it the value 2

Any existing variable `n` is unaffected

Lisp writes `(let ((n 2)) ...)`

Any existing `n` is restored at the end of the block

Lisp

Assignment and Binding

The second, called *assignment*, updates the value of `n`

The second, called *assignment*, updates the value of `n`

Any existing value of that `n` is overwritten: destroyed

The second, called *assignment*, updates the value of `n`

Any existing value of that `n` is overwritten: destroyed

We can’t get the old value back, even at the end of blocks

The second, called *assignment*, updates the value of `n`

Any existing value of that `n` is overwritten: destroyed

We can't get the old value back, even at the end of blocks

Lisp writes:

The second, called *assignment*, updates the value of `n`

Any existing value of that `n` is overwritten: destroyed

We can’t get the old value back, even at the end of blocks

Lisp writes: another special form we haven’t seen yet

- binding: non-destructive
- assignment: destructive

- binding: non-destructive
- assignment: destructive

If we avoid destructive operations we avoid messing about with similarly named variables elsewhere in the code: everything is inherently local

- binding: non-destructive
- assignment: destructive

If we avoid destructive operations we avoid messing about with similarly named variables elsewhere in the code: everything is inherently local

We get the “variable don’t vary” effect; we can analyse code

Lisp
Assignment in Lisp

We have deliberately avoided mentioning this so far, as it's not part of the functional style

Assignment in Lisp

We have deliberately avoided mentioning this so far, as it's not part of the functional style

```
(setq n 2)
```

Lisp

Assignment in Lisp

We have deliberately avoided mentioning this so far, as it's not part of the functional style

```
(setq n 2)
```

Early Lisps had a *function* `set` that evaluated its first argument
`(set 'n 2)` was a common idiom

We have deliberately avoided mentioning this so far, as it's not part of the functional style

```
(setq n 2)
```

Early Lisps had a *function* `set` that evaluated its first argument `(set 'n 2)` was a common idiom

`setq` as "set quote" was introduced as a handy abbreviation

Assignment in Lisp

We have deliberately avoided mentioning this so far, as it's not part of the functional style

```
(setq n 2)
```

Early Lisps had a *function* `set` that evaluated its first argument `(set 'n 2)` was a common idiom

`setq` as “set quote” was introduced as a handy abbreviation

And people were forever writing `(set n 2)` by mistake: this updates the thing (sometimes a symbol) that is the value of `n`: it doesn't update `n`

Assignment in Lisp

We have deliberately avoided mentioning this so far, as it's not part of the functional style

```
(setq n 2)
```

Early Lisps had a *function* `set` that evaluated its first argument `(set 'n 2)` was a common idiom

`setq` as "set quote" was introduced as a handy abbreviation

And people were forever writing `(set n 2)` by mistake: this updates the thing (sometimes a symbol) that is the value of `n`: it doesn't update `n`

Not often what people wanted

Lisp

Assignment in Lisp

```
[1]> (setq x 'y)
Y
[2]> (setq y 3)
3
[3]> (set x 4)
4
[4]> x
Y
[5]> y
4
```

Avoid setq, it is not functional style

Avoid setq, it is not functional style

And NEVER use set

Avoid setq, it is not functional style

And NEVER use set

And that includes all the variants, such as setf and set!

The functional style rejects all kinds of destructive update

The functional style rejects all kinds of destructive update

So values can never be changed by other parts of code you can't see

The functional style rejects all kinds of destructive update

So values can never be changed by other parts of code you can't see

As well as values of variables, this includes updates of datastructures

E.g., you can't/shouldn't change an element in a list:

```
(1 2 3) -> (1 4 3)
```

E.g., you can't/shouldn't change an element in a list:
`(1 2 3) -> (1 4 3)`

If you want that, make a new list with the replacement value

E.g., you can't/shouldn't change an element in a list:
`(1 2 3) -> (1 4 3)`

If you want that, make a new list with the replacement value

Some other part of code elsewhere might be using that list, too, and you've just messed it up

E.g., you can't/shouldn't change an element in a list:
`(1 2 3) -> (1 4 3)`

If you want that, make a new list with the replacement value

Some other part of code elsewhere might be using that list, too, and you've just messed it up

This is not as wasteful as it might seem, as a non-update guarantee allows us to *share* a lot more of our datastructures. See later

If you ever find yourself using `setq`, stop and think: you probably want to write your code in a better way

Assignment in Lisp

If you ever find yourself using `setq`, stop and think: you probably want to write your code in a better way

If you ever find yourself using `set`, throw away everything and start again

If you ever find yourself using `setq`, stop and think: you probably want to write your code in a better way

If you ever find yourself using `set`, throw away everything and start again

Some functional languages *do not have assignment*

If you ever find yourself using `setq`, stop and think: you probably want to write your code in a better way

If you ever find yourself using `set`, throw away everything and start again

Some functional languages *do not have assignment*

They actively prevent you from making that mistake

If you ever find yourself using `setq`, stop and think: you probably want to write your code in a better way

If you ever find yourself using `set`, throw away everything and start again

Some functional languages *do not have assignment*

They actively prevent you from making that mistake

They *do* have binding (local variables), as it is “safe”

Exercise. Explain the result of

```
(let ((x 'y)
      (y 33))
  (set x 44)
  y)
```

in Common Lisp

Function definition in Lisp is “really” an assignment

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

is “really”

```
(setq foo (lambda (n) (+ n 1)))
```

Plus some bookkeeping: the `defun` stores the name of the function in the function, for the benefit of the programmer. Plus a bit of fiddling for recursive functions

```
(defun foo (n) (+ n 1))
foo -> #<Procedure foo>

(setq bar (lambda (n) (+ n 1)))
bar -> #<Procedure #80e2388>
```

This is just a cosmetic feature

The functional style reduces or preferably eliminates the use of assignment: it’s unsafe on non-local variables (no referential transparency) and overall it makes code hard to analyse

The functional style reduces or preferably eliminates the use of assignment: it's unsafe on non-local variables (no referential transparency) and overall it makes code hard to analyse

Binding is fine, and often essential!

The functional style reduces or preferably eliminates the use of assignment: it's unsafe on non-local variables (no referential transparency) and overall it makes code hard to analyse

Binding is fine, and often essential!

Note that defining `defun` in terms of `setq` isn't such a bad thing: we don't tend to update named functions dynamically in a program, and assigning just once is not such a problem

Single assignment languages allow you to update a variable just once, from being undefined to the given value

Single assignment languages allow you to update a variable just once, from being undefined to the given value

It doesn't really affect the underlying theory, it's just occasionally more convenient for the programmer

Single assignment languages allow you to update a variable just once, from being undefined to the given value

It doesn't really affect the underlying theory, it's just occasionally more convenient for the programmer

After all, you wouldn't use a variable before it had a defined value, so the effect is just the same: in this special case assignment is non-destructive

Single assignment languages allow you to update a variable just once, from being undefined to the given value

It doesn't really affect the underlying theory, it's just occasionally more convenient for the programmer

After all, you wouldn't use a variable before it had a defined value, so the effect is just the same: in this special case assignment is non-destructive

It just separates the declaration of the local variable from its initialisation

Note: your coursework **must not** use `setq`

Note: your coursework **must not** use `setq`

Or any of its variants

Note: your coursework **must not** use `setq`

Or any of its variants

Regardless of how much it appears below!

Another part of the functional style is enabled by *closures*

Another part of the functional style is enabled by *closures*

Consider the code

```
(defun addn (n) (lambda (m) (+ m n)))
```

Lisp

Closures

Another part of the functional style is enabled by *closures*

Consider the code

```
(defun addn (n) (lambda (m) (+ m n)))
```

This returns a *function* that adds *n* to its argument

Another part of the functional style is enabled by *closures*

Consider the code

```
(defun addn (n) (lambda (m) (+ m n)))
```

This returns a *function* that adds *n* to its argument

```
(addn 4) → #<Procedure #14b12948>
```

Another part of the functional style is enabled by *closures*

Consider the code

```
(defun addn (n) (lambda (m) (+ m n)))
```

This returns a *function* that adds *n* to its argument

`(addn 4)` → `#<Procedure #14b12948>`

Exercise. Write this `defun` out in the `setq` of a `lambda` equivalent form

Now,

`((addn 4) 5)` $\rightarrow$ `9`
(Lisp-1 only; Lisp-2s need `funcall`)

Now,

`((addn 4) 5)` $\rightarrow$ `9`
(Lisp-1 only; Lisp-2s need `funcall`)

`(addn 4)` evaluates to a function that adds 4

Now,

`((addn 4) 5)` $\rightarrow$ `9`
(Lisp-1 only; Lisp-2s need `funcall`)

`(addn 4)` evaluates to a function that adds 4

Now, `(setq addfour (addn 4))` and then

`(addfour 6)` $\rightarrow$ `10`, as expected

We use `setq` in the `defun`, single assignment way

Lisp

Closures

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

But, still `(addfour 6)` $\rightarrow$ `10`

Lisp

Closures

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

But, still `(addfour 6)` $\rightarrow$ `10`

`addfive` is a new, different function from `addfour`

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

But, still `(addfour 6)` $\rightarrow$ `10`

`addfive` is a new, different function from `addfour`

Just as `cons` makes new pairs, `lambda` makes new functions

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

But, still `(addfour 6)` $\rightarrow$ `10`

`addfive` is a new, different function from `addfour`

Just as `cons` makes new pairs, `lambda` makes new functions

And `addfive` “remembers” that it was created with $n = 5$; while `addfour` was created with $n = 4$

`(setq addfive (addn 5))` and then `(addfive 7)` $\rightarrow$ `12`

But, still `(addfour 6)` $\rightarrow$ `10`

`addfive` is a new, different function from `addfour`

Just as `cons` makes new pairs, `lambda` makes new functions

And `addfive` “remembers” that it was created with $n = 5$; while `addfour` was created with $n = 4$

(Strictly: “the function that `addfive` refers to”, etc.)

This is because `addfour` and `addfive` name more than just simple bits of code: they are *closures*

This is because `addfour` and `addfive` name more than just simple bits of code: they are *closures*

A closure consists of two parts

- code
- environment

This is because `addfour` and `addfive` name more than just simple bits of code: they are *closures*

A closure consists of two parts

- code
- environment

Code is the simple executable thing we were expecting

This is because `addfour` and `addfive` name more than just simple bits of code: they are *closures*

A closure consists of two parts

- code
- environment

Code is the simple executable thing we were expecting

The *environment* is the collection of the non-local bindings used in the function *together with* their values from the context of the definition of the function

When we evaluate `(addn 4)` the closure returned contains

- the code `(lambda (m) (+ m n))`
- the environment `n: 4`

The environment refers to the `n` from the context created by the call to `addn`

When we evaluate `(addn 4)` the closure returned contains

- the code `(lambda (m) (+ m n))`
- the environment `n: 4`

The environment refers to the `n` from the context created by the call to `addn`

That `n` local to `addn` is no longer accessible when `addn` exits, apart from through the above environment binding

When we evaluate `(addn 4)` the closure returned contains

- the code `(lambda (m) (+ m n))`
- the environment `n: 4`

The environment refers to the `n` from the context created by the call to `addn`

That `n` local to `addn` is no longer accessible when `addn` exits, apart from through the above environment binding

The binding *does not disappear* when we leave the `addn`, but is *captured* and kept by the closure, i.e., the `lambda`

When the closure is evaluated, it can look up `n` in the associated environment to determine its value is 4 (or whatever)

When the closure is evaluated, it can look up `n` in the associated environment to determine its value is 4 (or whatever)

When the closure is evaluated, it can look up `n` in the associated environment to determine its value is 4 (or whatever)

This, of course, has repercussions on how functions/closures are implemented in Lisp, but the benefits are huge

When the closure is evaluated, it can look up `n` in the associated environment to determine its value is 4 (or whatever)

This, of course, has repercussions on how functions/closures are implemented in Lisp, but the benefits are huge

In particular, each closure needs a bit more memory to store the environment, over and above what the function code uses

When the closure is evaluated, it can look up `n` in the associated environment to determine its value is 4 (or whatever)

This, of course, has repercussions on how functions/closures are implemented in Lisp, but the benefits are huge

In particular, each closure needs a bit more memory to store the environment, over and above what the function code uses

Closures are another powerful basic idea that can be used for many different purposes

Thus: lambdas actually create new closures, not simply functions

Thus: lambdas actually create new closures, not simply functions

And lambdas are said to *capture the environment*

Thus: lambdas actually create new closures, not simply functions

And lambdas are said to *capture the environment*

Each lambda has its own separate environment part

Thus: lambdas actually create new closures, not simply functions

And lambdas are said to *capture the environment*

Each lambda has its own separate environment part

But they tend to share the common code part

Thus: lambdas actually create new closures, not simply functions

And lambdas are said to *capture the environment*

Each lambda has its own separate environment part

But they tend to share the common code part

Note that we are often lazy and use the word “function” when we ought to use the word “closure”

Thus: lambdas actually create new closures, not simply functions

And lambdas are said to *capture the environment*

Each lambda has its own separate environment part

But they tend to share the common code part

Note that we are often lazy and use the word “function” when we ought to use the word “closure”

This is because in Lisp, closures are the fundamental objects we use all the time