# Lisp

## Evaluation

What of the empty list and atom `()`?

# Lisp

## Evaluation

What of the empty list and atom `()`?

Does it self-evaluate or does it need quoting?

# Lisp

## Evaluation

What of the empty list and atom `()`?

Does it self-evaluate or does it need quoting?

It varies. It doesn't hurt to quote it

# Lisp

## Evaluation

What of the empty list and atom `()`?

Does it self-evaluate or does it need quoting?

It varies. It doesn't hurt to quote it

The symbol `NIL` is often pre-set with the value `()`

# Lisp

## Evaluation

What of the empty list and atom `()`?

Does it self-evaluate or does it need quoting?

It varies. It doesn't hurt to quote it

The symbol `NIL` is often pre-set with the value `()`

Or sometimes the symbol `nil`. Sometimes case of symbols is important, sometimes not

# Lisp

Within this Unit we shall be taking examples from EuLisp. It is a Lisp-1, and has more consistent semantics than Common Lisp; its syntax is more like most Lisps than Scheme

# Lisp

Within this Unit we shall be taking examples from EuLisp. It is a Lisp-1, and has more consistent semantics than Common Lisp; its syntax is more like most Lisps than Scheme

In EuLisp case *is* important: `NIL` and `nil` are different symbols

# Lisp

Within this Unit we shall be taking examples from EuLisp. It is a Lisp-1, and has more consistent semantics than Common Lisp; its syntax is more like most Lisps than Scheme

In EuLisp case *is* important: `NIL` and `nil` are different symbols

In Common Lisp they are the same symbol

# Lisp

Within this Unit we shall be taking examples from EuLisp. It is a Lisp-1, and has more consistent semantics than Common Lisp; its syntax is more like most Lisps than Scheme

In EuLisp case *is* important: `NIL` and `nil` are different symbols

In Common Lisp they are the same symbol

Even stranger, in Common Lisp `()` is considered *syntactically* identical to `NIL`, so `()` is classed as a symbol as well as a list

# Lisp

## Truth

() is quite often taken as the “false” value in Lisp; anything else is “true”

# Lisp

## Truth

`()` is quite often taken as the “false” value in Lisp; anything else is “true”

The symbol `t` is often taken as an exemplar true

# Lisp

## Truth

`()` is quite often taken as the “false” value in Lisp; anything else is “true”

The symbol `t` is often taken as an exemplar true

```
(listp '(a b)) → t
(listp 'a) → ()
```

# Lisp

## Truth

`()` is quite often taken as the “false” value in Lisp; anything else is “true”

The symbol `t` is often taken as an exemplar true

```
(listp '(a b)) → t
(listp 'a) → ()
```

For convenience, the value of the variable `t` is set to the symbol `t`

```
t → t
```

# Lisp

## Truth

`()` is quite often taken as the “false” value in Lisp; anything else is “true”

The symbol `t` is often taken as an exemplar true

```
(listp '(a b)) → t
(listp 'a) → ()
```

For convenience, the value of the variable `t` is set to the symbol `t`

```
t → t
```

This is not self-evaluation: just that the value is itself!

# Lisp

## Truth

`()` is quite often taken as the “false” value in Lisp; anything else is “true”

The symbol `t` is often taken as an exemplar true

```
(listp '(a b)) → t
(listp 'a) → ()
```

For convenience, the value of the variable `t` is set to the symbol `t`

```
t → t
```

This is not self-evaluation: just that the value is itself!

Exercise: what would happen if you set the value of `t` to `()`?

# Lisp

## Truth

Other Lisps have a separate Truth datatype for true and false

# Lisp

## Truth

Other Lisps have a separate Truth datatype for true and false

So `()` is invalid in a Boolean expression

# Lisp

## Truth

Other Lisps have a separate Truth datatype for true and false

So `()` is invalid in a Boolean expression

Still others have a separate false value and any other value, including `()`, is true

# Lisp

## Truth

We have now covered most of the ways Lisp expressions are evaluated

# Lisp

## Truth

We have now covered most of the ways Lisp expressions are evaluated

There are a few more special forms, and we shall treat them as they come up

# Lisp

## Truth

We have now covered most of the ways Lisp expressions are evaluated

There are a few more special forms, and we shall treat them as they come up

The fundamental underlying idea is: all expressions return a value

# Lisp

## Truth

We have now covered most of the ways Lisp expressions are evaluated

There are a few more special forms, and we shall treat them as they come up

The fundamental underlying idea is: all expressions return a value

> *Lisp knows the value of everything, but the cost of nothing*

# Lisp

## Lists

The principal datastructure in Lisp is the list

# Lisp

## Lists

The principal datastructure in Lisp is the list

The function `list` makes lists: `(list 1 'x x)` makes a list containing 1, the symbol `x` and whatever the current value of the variable `x` is

# Lisp

## Lists

The principal datastructure in Lisp is the list

The function `list` makes lists: `(list 1 'x x)` makes a list containing 1, the symbol `x` and whatever the current value of the variable `x` is

`list` is a *n*-ary function

# Lisp

## Lists

The principal datastructure in Lisp is the list

The function `list` makes lists: `(list 1 'x x)` makes a list containing 1, the symbol `x` and whatever the current value of the variable `x` is

`list` is a *n*-ary function

Exercise. What is the value of `(list)`?

# Lisp

## Lists

The principal datastructure in Lisp is the list

The function `list` makes lists: `(list 1 'x x)` makes a list containing 1, the symbol `x` and whatever the current value of the variable `x` is

`list` is a *n*-ary function

Exercise. What is the value of `(list)`?

We can also input a *constant* list using `quote`

# Lisp

## Lists

The principal datastructure in Lisp is the list

The function `list` makes lists: `(list 1 'x x)` makes a list containing 1, the symbol `x` and whatever the current value of the variable `x` is

`list` is a *n*-ary function

Exercise. What is the value of `(list)`?

We can also input a *constant* list using `quote`

Exercise. What is the value of `'(1 'x x)`?

# Lisp

## Lists

The functions `car` and `cdr` return parts of lists

# Lisp

## Lists

The functions `car` and `cdr` return parts of lists

`(car '(a b c))` is `a`

# Lisp

## Lists

The functions `car` and `cdr` return parts of lists

`(car '(a b c))` is `a`

`car` gives us the first thing in the list, called `first` or `head` in some Lisps

# Lisp

## Lists

The functions `car` and `cdr` return parts of lists

`(car '(a b c))` is `a`

`car` gives us the first thing in the list, called `first` or `head` in some Lisps

`(cdr '(a b c))` is `(b c)`

# Lisp

## Lists

The functions `car` and `cdr` return parts of lists

`(car '(a b c))` is `a`

`car` gives us the first thing in the list, called `first` or `head` in some Lisps

`(cdr '(a b c))` is `(b c)`

`cdr` gives us the rest of the list, called `rest` or `tail` in some Lisps

# Lisp

## Lists

Note that in `(car '((a b) (c d)))` the first item in the list is `(a b)`

# Lisp

## Lists

Note that in `(car '((a b) (c d)))` the first item in the list is `(a b)`

Exercise. What is `(cdr '((a b) (c d)))`?

# Lisp

## History

So why the weird names?

# Lisp

## History

So why the weird names?

McCarthy's original implementation was on an IBM 704 whose architecture had several registers, in particular an address register and a decrement register

# Lisp

## History

So why the weird names?

McCarthy's original implementation was on an IBM 704 whose architecture had several registers, in particular an address register and a decrement register

It was convenient for the system when manipulating to a list to put a pointer to the head in the address register and a pointer to the tail in the decrement register

# Lisp

## History

So why the weird names?

McCarthy's original implementation was on an IBM 704 whose architecture had several registers, in particular an address register and a decrement register

It was convenient for the system when manipulating to a list to put a pointer to the head in the address register and a pointer to the tail in the decrement register

The head was the **c**ontents of the **a**ddress **r**egister: car

# Lisp

## History

So why the weird names?

McCarthy's original implementation was on an IBM 704 whose architecture had several registers, in particular an address register and a decrement register

It was convenient for the system when manipulating to a list to put a pointer to the head in the address register and a pointer to the tail in the decrement register

The head was the **c**ontents of the **a**ddress **r**egister: car

The tail was the **c**ontents of the **d**ecrement **r**egister: cdr

# Lisp

## Lists

- `(car '(a b c))` is `a`

# Lisp

## Lists

- `(car '(a b c))` is `a`
- `(cdr '(a b c))` is `(b c)`

# Lisp

## Lists

- `(car '(a b c))` is `a`
- `(cdr '(a b c))` is `(b c)`
- `(car (cdr '(a b c)))` is `b`

# Lisp

## Lists

- `(car '(a b c))` is `a`
- `(cdr '(a b c))` is `(b c)`
- `(car (cdr '(a b c)))` is `b`
- `(cdr (cdr '(a b c)))` is `(c)`

# Lisp

## Lists

- `(car '(a b c))` is `a`
- `(cdr '(a b c))` is `(b c)`
- `(car (cdr '(a b c)))` is `b`
- `(cdr (cdr '(a b c)))` is `(c)`
- `(car (cdr (cdr '(a b c))))` is `c`

# Lisp

## Lists

- `(car '(a b c))` is `a`
- `(cdr '(a b c))` is `(b c)`
- `(car (cdr '(a b c)))` is `b`
- `(cdr (cdr '(a b c)))` is `(c)`
- `(car (cdr (cdr '(a b c))))` is `c`
- `(cdr (cdr (cdr '(a b c))))` is `()`

# Lisp

## Lists

`car` and `cdr` of `()` is another dodgy point

# Lisp

## Lists

`car` and `cdr` of `()` is another dodgy point

Some Lisps give an error for `(car ())` and `(cdr ())`

# Lisp

## Lists

`car` and `cdr` of `()` is another dodgy point

Some Lisps give an error for `(car ())` and `(cdr ())`

Some give `()` as the value for both (efficiency over semantics, again)

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

The function `cons` takes an object and a list and *constructs* a new list by sticking the object on the front of the list

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

The function `cons` takes an object and a list and *constructs* a new list by sticking the object on the front of the list

`(cons 1 '(2 3))` is `(1 2 3)`

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

The function `cons` takes an object and a list and *constructs* a new list by sticking the object on the front of the list

`(cons 1 '(2 3))` is `(1 2 3)`

`(cons '(1 2) '(2 3))` is `((1 2) 2 3)`

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

The function `cons` takes an object and a list and *constructs* a new list by sticking the object on the front of the list

`(cons 1 '(2 3))` is `(1 2 3)`

`(cons '(1 2) '(2 3))` is `((1 2) 2 3)`

`(cons 1 ())` is `(1)`

# Lisp

## Lists

`car` and `cdr` are primitives: `list` is not

The function `cons` takes an object and a list and *constructs* a new list by sticking the object on the front of the list

`(cons 1 '(2 3))` is `(1 2 3)`

`(cons '(1 2) '(2 3))` is `((1 2) 2 3)`

`(cons 1 ())` is `(1)`

`(cons 1 (cons 2 ()))` is `(1 2)`

# Lisp

## Lists

`list` is defined as multiple `cons`

# Lisp

## Lists

`list` is defined as multiple `cons`

So `(list 1 2 3)` executes `(cons 1 (cons 2 (cons 3 ()))))`

# Lisp

## Lists

`list` is defined as multiple `cons`

So `(list 1 2 3)` executes `(cons 1 (cons 2 (cons 3 ()))))`

You can tell why `list` is often provided as well as the primitive `cons`

# Lisp

## Pairs

So `cons` is a primitive

# Lisp

## Pairs

So `cons` is a primitive

In fact, `cons` takes *any* pair of objects and makes a thing called a *pair*

# Lisp

## Pairs

So `cons` is a primitive

In fact, `cons` takes *any* pair of objects and makes a thing called a *pair*

`(cons 1 2)` $\rightarrow$ `(1 . 2)`

# Lisp

## Pairs

So `cons` is a primitive

In fact, `cons` takes *any* pair of objects and makes a thing called a *pair*

`(cons 1 2)` $\rightarrow$ `(1 . 2)`

The dot indicates this is a pair

# Lisp

## Pairs

So `cons` is a primitive

In fact, `cons` takes *any* pair of objects and makes a thing called a *pair*

`(cons 1 2)` $\rightarrow$ `(1 . 2)`

The dot indicates this is a pair

`(cons 1 '(2))` $\rightarrow$ `(1 2)`

# Lisp

## Pairs

So `cons` is a primitive

In fact, `cons` takes *any* pair of objects and makes a thing called a *pair*

`(cons 1 2)` $\rightarrow$ `(1 . 2)`

The dot indicates this is a pair

`(cons 1 '(2))` $\rightarrow$ `(1 2)`

Where's the dot?

# Lisp

## Pairs

It's still there internally, but the Lisp printer prints pairs specially

# Lisp

## Pairs

It's still there internally, but the Lisp printer prints pairs specially

Rather than printing `(1 . (2 . (3 . ())))` it prints `(1 2 3)`

# Lisp

## Pairs

It's still there internally, but the Lisp printer prints pairs specially

Rather than printing `(1 . (2 . (3 . ())))` it prints `(1 2 3)`

It's much more readable and fits our intuition of a list

# Lisp

## Pairs

It's still there internally, but the Lisp printer prints pairs specially

Rather than printing `(1 . (2 . (3 . ())))` it prints `(1 2 3)`

It's much more readable and fits our intuition of a list

A list is really just a pair whose second element is a list

# Lisp

## Pairs

Essentially, if the cdr is a list Lisp prints a space then the cdr (recursively)

# Lisp

## Pairs

Essentially, if the cdr is a list Lisp prints a space then the cdr (recursively)

If the cdr is not a list, it prints a dot then the cdr

# Lisp

## Pairs

Essentially, if the cdr is a list Lisp prints a space then the cdr (recursively)

If the cdr is not a list, it prints a dot then the cdr

This produces the nice list output for lists and only prints dots for pairs that are not lists

# Lisp

## Pairs

Exercise. Predict the result of printing

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

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

It's just that in lists the `cdr` part is itself a list

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

It's just that in lists the `cdr` part is itself a list

Internally to Lisp, there are no lists, just pairs

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

It's just that in lists the `cdr` part is itself a list

Internally to Lisp, there are no lists, just pairs

We fool ourselves by printing certain pairs in a pretty way

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

It's just that in lists the `cdr` part is itself a list

Internally to Lisp, there are no lists, just pairs

We fool ourselves by printing certain pairs in a pretty way

`(cons (cons (cons (cons () 4) 3) 2) 1)` would be an equally acceptable way to implement a list, with `cdr` for the head and `car` for the tail

# Lisp

## Pairs

We now see that `car` and `cdr` are actually quite symmetric: each is just one part of a pair

It's just that in lists the `cdr` part is itself a list

Internally to Lisp, there are no lists, just pairs

We fool ourselves by printing certain pairs in a pretty way

`(cons (cons (cons (cons () 4) 3) 2) 1)` would be an equally acceptable way to implement a list, with `cdr` for the head and `car` for the tail

Exercise. What would this look like when printed?

# Lisp

## Lists

Lists are supremely suited for recursive procedures

# Lisp

## Lists

Lists are supremely suited for recursive procedures

Most recursions are like

# Lisp

## Lists

Lists are supremely suited for recursive procedures

Most recursions are like

- Treat the base case

# Lisp

## Lists

Lists are supremely suited for recursive procedures

Most recursions are like

- Treat the base case
- Otherwise do something with some part of the problem

# Lisp

## Lists

Lists are supremely suited for recursive procedures

Most recursions are like

- Treat the base case
- Otherwise do something with some part of the problem
- Then call yourself recursively on the rest of the problem

# Lisp

## Lists

Lists are supremely suited for recursive procedures

Most recursions are like

- Treat the base case
- Otherwise do something with some part of the problem
- Then call yourself recursively on the rest of the problem

```
int factorial(int n)
{
  if (n < 2) return 1;
  return n*factorial(n-1);
}
```

# Lisp

## Lists

A list is exactly the right structure for this

# Lisp

## Lists

A list is exactly the right structure for this

- Treat the base case: often ()

# Lisp

## Lists

A list is exactly the right structure for this

- Treat the base case: often `()`
- Otherwise do something with some part of the problem, the `car`

# Lisp

## Lists

A list is exactly the right structure for this

- Treat the base case: often `()`
- Otherwise do something with some part of the problem, the `car`
- Then call yourself recursively on the rest of the problem, the `cdr`

# Lisp

## Lists

A list can be *defined* recursively

# Lisp

## Lists

A list can be *defined* recursively

A list is

# Lisp

## Lists

A list can be *defined* recursively

A list is

- ()

# Lisp

## Lists

A list can be *defined* recursively

A list is

- `()`
- or an object (the `car`)

# Lisp

## Lists

A list can be *defined* recursively

A list is

- `()`
- or an object (the `car`)
- `cons`d onto a list (the `cdr`)

# Lisp

## Lists

We have

# Lisp

## Lists

We have

- `(car (cons x l))` returns `x`

# Lisp

## Lists

We have

- `(car (cons x l))` returns `x`
- `(cdr (cons x l))` returns `l`

# Lisp

## Lists

We have

- `(car (cons x l))` returns `x`
- `(cdr (cons x l))` returns `l`
- `(cons (car l) (cdr l))` return something like `l`

# Lisp

## Lists

We have

- `(car (cons x l))` returns `x`
- `(cdr (cons x l))` returns `l`
- `(cons (car l) (cdr l))` return something like `l`

To explain the “something like” will take some time

# Lisp

## Lists

We have

- `(car (cons x l))` returns `x`
- `(cdr (cons x l))` returns `l`
- `(cons (car l) (cdr l))` return something like `l`

To explain the “something like” will take some time

But before that, we need to see more Lisp basics

# Lisp

## Basic Lisp Functionality

We now whizz though the basic Lisp bits and pieces: they are from EuLisp, but as always other Lisps are similar, but maybe different

# Lisp

## Basic Lisp Functionality

We now whizz though the basic Lisp bits and pieces: they are from EuLisp, but as always other Lisps are similar, but maybe different

Constants

- numbers: `1` integer and `1.0` float
- strings: `"hello world"`
- characters: `#\c` for the character 'c'
- vectors: `#(1 (b c) "hi")` a vector of length 3, indexed from 0 to 2

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

Vectors: `(make-vector 4)` makes a vector of length 4

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

Vectors: `(make-vector 4)` makes a vector of length 4

Access: `(vector-ref v 3)` for element 3 in vector `v`

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

Vectors: `(make-vector 4)` makes a vector of length 4

Access: `(vector-ref v 3)` for element 3 in vector `v`

Update: `((setter vector-ref) v 3 x)` to update element 3 to value of `x`

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

Vectors: `(make-vector 4)` makes a vector of length 4

Access: `(vector-ref v 3)` for element 3 in vector `v`

Update: `((setter vector-ref) v 3 x)` to update element 3 to value of `x`

`setter` is a general update mechanism

# Lisp

## List and Vectors

Lists: `cons` and `list`, quoted constant `'(a b c)`

Vectors: `(make-vector 4)` makes a vector of length 4

Access: `(vector-ref v 3)` for element 3 in vector `v`

Update: `((setter vector-ref) v 3 x)` to update element 3 to value of `x`

`setter` is a general update mechanism

All arguments can be arbitrary expressions, here and elsewhere

# Lisp

## Expressions

Anything that is not constant will be evaluated; things prefixed by `quote` are constant

# Lisp

## Expressions

Anything that is not constant will be evaluated; things prefixed by `quote` are constant

All expressions return a value

# Lisp

## Expressions

Anything that is not constant will be evaluated; things prefixed by `quote` are constant

All expressions return a value

Usually it's the obvious value from the function call

# Lisp

## Expressions

Anything that is not constant will be evaluated; things prefixed by `quote` are constant

All expressions return a value

Usually it's the obvious value from the function call

`((setter vector-ref) v 3 99)` returns 99

# Lisp

## Expressions

A special form `progn` collects together several expressions and wraps them into a single expression

```
(progn
   expr1
   expr2
   ...
   exprn)
```

This evaluates the `exprs` sequentially in order and its value is the value of the last `exprn`

# Lisp

## Expressions

A special form `progn` collects together several expressions and wraps them into a single expression

```
(progn
   expr1
   expr2
   ...
   exprn)
```

This evaluates the `exprs` sequentially in order and its value is the value of the last `exprn`

Exercise. Why is `progn` a special form?

# Lisp

## Expressions

```
(progn
   (print "adding values")
   (+ 2 3))
```

prints the message and returns 5

# Lisp

## Expressions

```
(progn
   (print "adding values")
   (+ 2 3))
```

prints the message and returns 5

`progn` is useful for when Lisp expects a single expression but we want to do more than one thing

# Lisp

## Expressions

```
(progn
   (print "adding values")
   (+ 2 3))
```

prints the message and returns 5

`progn` is useful for when Lisp expects a single expression but we want to do more than one thing

The `progn` wraps several things up and makes a single expression out of them

# Lisp

## Expressions

A related special form introduces local variables

```
(let ((var1 val1)
      (var2 val2)
      ...
      (varm valm))
  expr1
  expr2
  ...
  exprn)
```

The `vars` are symbols

Its value is the value of the last `exprn`

# Lisp

## Expressions

```
(let ((var1 val1)
      (var2 val2)
      ...
      (varm valm))
  expr1
  expr2
  ...
  exprn)
```

The `vals` can be arbitrary expressions; they are evaluated in some order, **then** the `vars` are given the corresponding values

# Lisp

## Expressions

```
(let ((var1 val1)
      (var2 val2)
      ...
      (varm valm))
  expr1
  expr2
  ...
  exprn)
```

The `val`s can be arbitrary expressions; they are evaluated in some order, **then** the `var`s are given the corresponding values

The body of the `let` can use the variables; they revert to whatever they were before (or being undefined) on exit

# Lisp

## Expressions

```
(let ((a 1)
      (b (let ((x 2)) (* x x))))
  (foo a b)
  (* a (- b a)))

(let ((car cdr)
      (cdr car))
  (cdr '(a b)))   ->   a
```

# Lisp

## Expressions

```
(let ((a 1)
      (b (let ((x 2)) (* x x))))
  (foo a b)
  (* a (- b a)))

(let ((car cdr)
      (cdr car))
  (cdr '(a b)))   ->   a
```

Take care with this kind of thing: you can write unreadable code in any language

# Lisp

## Expressions

```
(let ((a 1)
      (b (let ((x 2)) (* x x))))
  (foo a b)
  (* a (- b a)))

(let ((car cdr)
      (cdr car))
  (cdr '(a b)))   ->   a
```

Take care with this kind of thing: you can write unreadable code in any language

Exercise. Rewrite the second `let` for a Lisp-2

# Lisp

## Expressions

Exercise. What is the value of

```
(let ((x 1))
   (let ((x 2)
         (y x))
     y))
```

# Lisp

## Expressions

There is also a `let*`

```
(let* ((var1 val1)
       (var2 val2)
       ...
       (varm valm))
  expr1
  expr2
  ...
  exprn)
```

This is like `let`, but evaluates the `val`s in the given order, assigning to the `var`s as it goes; thus `val2` can refer to the just-computed value of `var1`

# Lisp

## Expressions

Exercise.

```
(let* ((car cdr)
       (cdr car))
  (cdr '(a b)))   ->   ?

(let ((x 1))
   (let* ((x 2)
          (y x))
     y))          ->   ?
```

Exercise. Think about how `let*` could be implemented using `let`

# Lisp

## Expressions

`let*` is more like the way local variables are declared in other languages

```
{ int x = 1;
    { int x = 2;
      int y = x;
      ...
    }
}
```

Each variable in the initialiser refers to its closest declaration, be it inside this block or not

# Lisp

## Expressions

`let` is more general and can be more efficient than `let*`

# Lisp

## Expressions

`let` is more general and can be more efficient than `let*`

For example, the values in a `let` might be able to be evaluated in parallel: a `let*` explicitly denies this

# Lisp

## Expressions

Symbols: the full syntax is quite general, but stick to “sequence of letters and digits, starting with a letter”

# Lisp

## Expressions

Symbols: the full syntax is quite general, but stick to “sequence of letters and digits, starting with a letter”

Certain other things, like + and $*$, are also regarded as valid constituents of symbols

# Lisp

## Expressions

Symbols: the full syntax is quite general, but stick to “sequence of letters and digits, starting with a letter”

Certain other things, like + and `*`, are also regarded as valid constituents of symbols

Case is significant in EuLisp: `car` and `Car` are different symbols

# Lisp

## Expressions

Conditionals: `if` is a simple special form

```
(if condition expr1 expr2)
```

Evaluate the condition; if true, evaluate `expr1`, else evaluate `expr2`

The value returned from the `if` is the value of whichever `expr` that was evaluated

# Lisp

## Expressions

Conditionals: `if` is a simple special form

```
(if condition expr1 expr2)
```

Evaluate the condition; if true, evaluate `expr1`, else evaluate `expr2`

The value returned from the `if` is the value of whichever `expr` that was evaluated

`if` plays the role of both `if` and `?:` in C:

```
y = 2*(x > 0 ? 1 : -1) + z
```

# Lisp

## Expressions

```
(if (> x 0)
    (progn
      (print "x is positive")
      (foo x))
    (bar x))

(let ((y (+ (* 2 (if (> x 0) 1 -1)) z)))
  ...)
```

# Lisp

## Expressions

EuLisp also provides

```
(when condition
   expr1
   expr2
   ...
   exprn)
```

Evaluate the condition; if true, evaluate the `exprs` in order and return the value of the last; otherwise return `()`

# Lisp

## Expressions

EuLisp also provides

```
(unless condition
   expr1
   expr2
   ...
   exprn)
```

Evaluate the condition; if false, evaluate the `exprs` in order and return the value of the last; otherwise return `()`

# Lisp

## Expressions

EuLisp also provides

```
(unless condition
   expr1
   expr2
   ...
   exprn)
```

Evaluate the condition; if false, evaluate the `exprs` in order and return the value of the last; otherwise return `()`

Note: all of `if`, `when` and `unless` are special forms as they treat their arguments specially: in particular, they do not evaluate them unless required

# Lisp

## Expressions

- `(not x)` if `x` is `()` (false) return `t`; else return `()`

# Lisp

## Expressions

- `(not x)` if `x` is `()` (false) return `t`; else return `()`
- `(or expr1 expr2 ... exprn)` evaluate the `expr`s in order; whenever any value is true, immediately return it as the value of the `or`; if all are false, return `()`

# Lisp

## Expressions

- `(not x)` if `x` is `()` (false) return `t`; else return `()`
- `(or expr1 expr2 ... exprn)` evaluate the `expr`s in order; whenever any value is true, immediately return it as the value of the `or`; if all are false, return `()`
- `(and expr1 expr2 ... exprn)` evaluate the `expr`s in order; whenever any value is false, immediately return `()` as the value of the `and`; if all are false, return the value of the last `exprn`

# Lisp

## Expressions

- `(not x)` if `x` is `()` (false) return `t`; else return `()`
- `(or expr1 expr2 ... exprn)` evaluate the `expr`s in order; whenever any value is true, immediately return it as the value of the `or`; if all are false, return `()`
- `(and expr1 expr2 ... exprn)` evaluate the `expr`s in order; whenever any value is false, immediately return `()` as the value of the `and`; if all are false, return the value of the last `exprn`

`and` and `or` are special forms; `not` is an ordinary function

# Lisp

## Expressions

```
(if (and (> x 0) (not (= y z)))
    (foo x y z)
    (foo y z x))

(foo (or (bar x) (bar y)) 42)
```