Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

- `~masrjb/bin/euscheme` is an implementation of EuLisp (not a Scheme!)

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

- `~masrjb/bin/euscheme` is an implementation of EuLisp (not a Scheme!)
- `~masrjb/bin/xlisp` is an implementation of Scheme

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

- `~masrjb/bin/euscheme` is an implementation of EuLisp (not a Scheme!)
- `~masrjb/bin/xlisp` is an implementation of Scheme
- `~masrjb/bin/clisp` is an implementation of Common Lisp

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

- `~masrjb/bin/euscheme` is an implementation of EuLisp (not a Scheme!)
- `~masrjb/bin/xlisp` is an implementation of Scheme
- `~masrjb/bin/clisp` is an implementation of Common Lisp
- `~masrjb/bin/clojure` is an implementation of Clojure, a Lisp that runs atop the Java VM

Lisp

Running Lisp

Lisp can be used a via a compiler (like C) or as an interactive system (like Python)

On BUCS Linux machines `lcpu` we have

- `~masrjb/bin/euscheme` is an implementation of EuLisp (not a Scheme!)
- `~masrjb/bin/xlisp` is an implementation of Scheme
- `~masrjb/bin/clisp` is an implementation of Common Lisp
- `~masrjb/bin/clojure` is an implementation of Clojure, a Lisp that runs atop the Java VM
- Emacs contains a Common Lisp-like interpreter (most of Emacs is written in Lisp)

Lisp

Stopping Lisp

A `^D` will exit most of these Lisps

Lisp

Stopping Lisp

A `^D` will exit most of these Lisps

If you stuck in a recursive error handler, you might have to hit `^D` several times

Lisp

Expressions

When you type an expression at a Lisp interpreter, it takes that as a sign you want it evaluated

Lisp

Expressions

When you type an expression at a Lisp interpreter, it takes that as a sign you want it evaluated

`(+ 1 2)` will return 3

Lisp

Expressions

When you type an expression at a Lisp interpreter, it takes that as a sign you want it evaluated

`(+ 1 2)` will return 3

`(+ (+ 1 2 3) (* 2 8))` evaluates `(+ 1 2 3)` to get 6; `(* 2 8)` to get 16; then `(+ 6 16)` to get 22

Lisp

Expressions

When you type an expression at a Lisp interpreter, it takes that as a sign you want it evaluated

`(+ 1 2)` will return 3

`(+ (+ 1 2 3) (* 2 8))` evaluates `(+ 1 2 3)` to get 6; `(* 2 8)` to get 16; then `(+ 6 16)` to get 22

Note: + names a *n*-ary function

Lisp

Expressions

When you type an expression at a Lisp interpreter, it takes that as a sign you want it evaluated

`(+ 1 2)` will return `3`

`(+ (+ 1 2 3) (* 2 8))` evaluates `(+ 1 2 3)` to get `6`; `(* 2 8)` to get `16`; then `(+ 6 16)` to get `22`

Note: `+` names a *n*-ary function

Note: `+` and `*` have no special syntactic significance; they are treated exactly as symbols like `sin` or `foo`

Lisp

Expressions

`length` is the name of a function that takes a list and returns its length

Lisp

Expressions

`length` is the name of a function that takes a list and returns its length

The length of `((a b) 1 2 (d 1 3))` is 4

Lisp

Expressions

`length` is the name of a function that takes a list and returns its length

The length of `((a b) 1 2 (d 1 3))` is 4

So we go `(length (a b c))` and expect to get 3?

Lisp

Expressions

No: we get something like

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

Debug loop.  Type help: for help
Broken at #<Code #157f1330>

DEBUG>
```

in euscheme

Lisp

Expressions

```
*** - EVAL: undefined function A
The following restarts are available:
USE-VALUE      :R1      You may input a value to be used instead
  of (FDEFINITION 'A).
RETRY          :R2      Retry
STORE-VALUE    :R3      You may input a new value for (FDEFINITION 'A).
ABORT          :R4      ABORT
Break 1 [2]>
```

in clisp

Lisp

Expressions

```
error: unbound variable - c
happened in: #<Code #x2aee177d7488>
Entering break loop ('(reset)' to quit)
Debug 1> [1]
```

in xlisp

Lisp

Expressions

```
java.lang.Exception: Unable to resolve symbol: length in this context
clojure.lang.Compiler$CompilerException: NO_SOURCE_FILE:1: Unable to
resolve symbol: length in this context
        at clojure.lang.Compiler.analyze(Compiler.java:3713)
        at clojure.lang.Compiler.analyze(Compiler.java:3671)
        at clojure.lang.Compiler.access$100(Compiler.java:37)
        at clojure.lang.Compiler$InvokeExpr.parse(Compiler.java:2634)
        at clojure.lang.Compiler.analyzeSeq(Compiler.java:3860)
        at clojure.lang.Compiler.analyze(Compiler.java:3698)
        at clojure.lang.Compiler.analyze(Compiler.java:3671)
        at clojure.lang.Compiler.access$100(Compiler.java:37)
        at clojure.lang.Compiler$BodyExpr$Parser.parse(Compiler.java:3384)
        at clojure.lang.Compiler$FnMethod.parse(Compiler.java:3231)
        at clojure.lang.Compiler$FnMethod.access$1200(Compiler.java:3142)
        at clojure.lang.Compiler$FnExpr.parse(Compiler.java:2766)
        at clojure.lang.Compiler.analyzeSeq(Compiler.java:3856)
        at clojure.lang.Compiler.analyze(Compiler.java:3698)
        at clojure.lang.Compiler.eval(Compiler.java:3889)
        at clojure.lang.Repl.main(Repl.java:75)
Caused by: java.lang.Exception: Unable to resolve symbol: length in this context
        at clojure.lang.Compiler.resolveIn(Compiler.java:4019)
        at clojure.lang.Compiler.resolve(Compiler.java:3972)
        at clojure.lang.Compiler.analyzeSymbol(Compiler.java:3955)
        at clojure.lang.Compiler.analyze(Compiler.java:3686)
        ... 15 more
```

in Clojure — it doesn't implement `length`!

Lisp

Expressions

The problem is that you asked the Lisp to evaluate `(length (a b c))`

Lisp

Expressions

The problem is that you asked the Lisp to evaluate `(length (a b c))`

Compare with evaluating `(- (* 2 3))`

Lisp

Expressions

The problem is that you asked the Lisp to evaluate `(length (a b c))`

Compare with evaluating `(- (* 2 3))`

This, naturally, evaluates the `(* 2 3)` and then the `-`

Lisp

Expressions

The problem is that you asked the Lisp to evaluate `(length (a b c))`

Compare with evaluating `(- (* 2 3))`

This, naturally, evaluates the `(* 2 3)` and then the `-`

In the same way, it tries to evaluate `(a b c)` to get something to pass to `length`

Lisp

Expressions

It is regarding the `(a b c)` as program, not data

Lisp

Expressions

It is regarding the `(a b c)` as program, not data

The function `a` is not defined; the variables `b` and `c` are not defined

Lisp

Expressions

It is regarding the `(a b c)` as program, not data

The function `a` is not defined; the variables `b` and `c` are not defined

So they all show an error message and dump you in an error loop

Lisp

Expressions

It is regarding the `(a b c)` as program, not data

The function `a` is not defined; the variables `b` and `c` are not defined

So they all show an error message and dump you in an error loop

Error loops allow you to investigate problems interactively; more later

Lisp

Expressions

So how can it tell that we wanted the `(a b c)` to be data (a list of three symbols), not a (broken) bit of code to be evaluated?

Lisp

Expressions

So how can it tell that we wanted the `(a b c)` to be data (a list of three symbols), not a (broken) bit of code to be evaluated?

Remember: program is identical to data in Lisp

Lisp

Expressions

So how can it tell that we wanted the `(a b c)` to be data (a list of three symbols), not a (broken) bit of code to be evaluated?

Remember: program is identical to data in Lisp

So there is a *special form* named `quote` we use to say “don’t eval this, it’s data”

Lisp

Expressions

So how can it tell that we wanted the `(a b c)` to be data (a list of three symbols), not a (broken) bit of code to be evaluated?

Remember: program is identical to data in Lisp

So there is a *special form* named `quote` we use to say "don't eval this, it's data"

`quote` is the "opposite" to `eval`

Lisp

Expressions

```
(quote (a b c))
```

`quote` is exempt from normal evaluation, it says “stop, eval no deeper”

Lisp

Expressions

```
(quote (a b c))
```

`quote` is exempt from normal evaluation, it says “stop, eval no deeper”

If we type `(quote (a b c))` at Lisp we get:

Lisp

Expressions

```
(quote (a b c))
```

`quote` is exempt from normal evaluation, it says “stop, eval no deeper”

If we type `(quote (a b c))` at Lisp we get:

```
(a b c)
```

Lisp

Expressions

```
(quote (a b c))
```

`quote` is exempt from normal evaluation, it says “stop, eval no deeper”

If we type `(quote (a b c))` at Lisp we get:

```
(a b c)
```

If we type `(length (quote (a b c)))` at Lisp we get:

Lisp

Expressions

```
(quote (a b c))
```

`quote` is exempt from normal evaluation, it says “stop, eval no deeper”

If we type `(quote (a b c))` at Lisp we get:

```
(a b c)
```

If we type `(length (quote (a b c)))` at Lisp we get:

```
3
```

Lisp

Expressions

`quote` is so important there is a syntactic abbreviation:

```
'(a b c)
```

means exactly the same as `(quote (a b c))`

Lisp

Expressions

`quote` is so important there is a syntactic abbreviation:

```
'(a b c)
```

means exactly the same as `(quote (a b c))`

`(length '(a b c))` returns 3

Lisp

Expressions

`quote` is so important there is a syntactic abbreviation:

```
'(a b c)
```

means exactly the same as `(quote (a b c))`

`(length '(a b c))` returns 3

Remember: typing something at the prompt means you want Lisp to evaluate it

Lisp

Expressions

`quote` is so important there is a syntactic abbreviation:

```
'(a b c)
```

means exactly the same as `(quote (a b c))`

`(length '(a b c))` returns 3

Remember: typing something at the prompt means you want Lisp to evaluate it

So if you *don't*, use `quote`

Lisp

Expressions

We can quote any expression: `'x` evaluates to the symbol `x`

Lisp

Expressions

We can quote any expression: `'x` evaluates to the symbol `x`

Warning: you will make errors with `quote`, either missing one where needed, or putting one in when not

Lisp

Expressions

We can quote any expression: `'x` evaluates to the symbol `x`

Warning: you will make errors with `quote`, either missing one where needed, or putting one in when not

You need a quote any place you have something that you want to be regarded as data that would otherwise be evaluated

Lisp

Expressions

We can quote any expression: `'x` evaluates to the symbol `x`

Warning: you will make errors with `quote`, either missing one where needed, or putting one in when not

You need a quote any place you have something that you want to be regarded as data that would otherwise be evaluated

Exercise: what do we get from evaluating `''x`?

Lisp

Expressions

We can quote any expression: `'x` evaluates to the symbol `x`

Warning: you will make errors with `quote`, either missing one where needed, or putting one in when not

You need a quote any place you have something that you want to be regarded as data that would otherwise be evaluated

Exercise: what do we get from evaluating `''x`?

Exercise: what do we get from evaluating `'(+ 1 2)`?

Lisp

Evaluation

Most atoms are *self-evaluating*

Lisp

Evaluation

Most atoms are *self-evaluating*

- 1 evaluates to 1

Lisp

Evaluation

Most atoms are *self-evaluating*

- 1 evaluates to 1
- `"hello"` evaluates to `"hello"`

Lisp

Evaluation

Most atoms are *self-evaluating*

- 1 evaluates to 1
- `"hello"` evaluates to `"hello"`

Symbols evaluate to their current value, if any

Lisp

Evaluation

Most atoms are *self-evaluating*

- 1 evaluates to 1
- `"hello"` evaluates to `"hello"`

Symbols evaluate to their current value, if any

Just like variables in other languages

Lisp

Evaluation

Most atoms are *self-evaluating*

- 1 evaluates to 1
- `"hello"` evaluates to `"hello"`

Symbols evaluate to their current value, if any

Just like variables in other languages

If the symbol has no current value, evaluating it results in an error (usually)

Lisp

Evaluation

Typically, in Lisp, we don't have to declare variables before use

Lisp

Evaluation

Typically, in Lisp, we don't have to declare variables before use

Variables don't have types associated with them

Lisp

Evaluation

Typically, in Lisp, we don't have to declare variables before use

Variables don't have types associated with them

A variable can hold any object of any type

Lisp

Evaluation

Typically, in Lisp, we don't have to declare variables before use

Variables don't have types associated with them

A variable can hold any object of any type

The types are in the *objects*, not the *variables*

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

- evaluate the arguments

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

- evaluate the arguments
- evaluate the function to call

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

- evaluate the arguments
- evaluate the function to call
- call the function on the arguments

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

- evaluate the arguments
- evaluate the function to call
- call the function on the arguments

Each special form has its own, individual, evaluation rule

Lisp

Evaluation

Lists evaluate as function calls: `(f a b)`

When the list is not a *special form*, like `quote`:

- evaluate the arguments
- evaluate the function to call
- call the function on the arguments

Each special form has its own, individual, evaluation rule

`quote`'s rule is: don't evaluate the argument

Lisp

Evaluation

Some Lisps evaluate the function before the arguments; others after

Lisp

Evaluation

Some Lisps evaluate the function before the arguments; others after

Some Lisps evaluate the arguments left-to-right; others right-to-left; others as they see fit

Lisp

Evaluation

Some Lisps evaluate the function before the arguments; others after

Some Lisps evaluate the arguments left-to-right; others right-to-left; others as they see fit

Some Lisps evaluate the arguments in parallel

Lisp

Evaluation

Some Lisps evaluate the function before the arguments; others after

Some Lisps evaluate the arguments left-to-right; others right-to-left; others as they see fit

Some Lisps evaluate the arguments in parallel

So don't write code that relies on, say `foo` being executed before `bar` in `(+ (foo 2) (bar 4 5))`

Lisp

Evaluation

Some Lisps evaluate the function before the arguments; others after

Some Lisps evaluate the arguments left-to-right; others right-to-left; others as they see fit

Some Lisps evaluate the arguments in parallel

So don't write code that relies on, say `foo` being executed before `bar` in `(+ (foo 2) (bar 4 5))`

Bad to do that in most languages, anyway

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function
  - 2 self-evaluates to 2

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function
  - 2 self-evaluates to 2
  - 3 self-evaluates to 3

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function
  - 2 self-evaluates to 2
  - 3 self-evaluates to 3
  - call multiplication on 2 and 3: get 6

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function
  - 2 self-evaluates to 2
  - 3 self-evaluates to 3
  - call multiplication on 2 and 3: get 6
- call addition on 4 and 6: get `10`

Lisp

Evaluation

So `(+ (* 2 3) 4)` is evaluated as, say,

- `+` is the name of the addition function
- `4` self-evaluates to `4`
- `(* 2 3)` evaluates as
  - `*` is the name of the multiplication function
  - 2 self-evaluates to 2
  - 3 self-evaluates to 3
  - call multiplication on 2 and 3: get 6
- call addition on 4 and 6: get `10`

We are being fussy here as it makes a big difference later

Lisp

Evaluation

Special forms are treated specially

Lisp

Evaluation

Special forms are treated specially

The first special form is `quote`: it stops evaluation

Lisp

Evaluation

Special forms are treated specially

The first special form is `quote`: it stops evaluation

Another special form is `if`, as in

```
(if test trueexpr falseexpr)
```

Lisp

Evaluation

Special forms are treated specially

The first special form is `quote`: it stops evaluation

Another special form is `if`, as in
`(if test trueexpr falseexpr)`

Exercise: think why it is a special form. Answer later

Lisp

Evaluation

In some Lisps (e.g., Scheme, EuLisp; generally on the right side of the family tree) collectively called *Lisp-1s* the function position is the same as the arguments

Lisp

Evaluation

In some Lisps (e.g., Scheme, EuLisp; generally on the right side of the family tree) collectively called *Lisp-1s* the function position is the same as the arguments

In `(+ x y)` the + is evaluated in the same way as the `x` and `y`

Lisp

Evaluation

In some Lisps (e.g., Scheme, EuLisp; generally on the right side of the family tree) collectively called *Lisp-1s* the function position is the same as the arguments

In `(+ x y)` the + is evaluated in the same way as the `x` and `y`

The value of the symbol + is a function that adds things

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

This is a way of saying “some code for something”

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

This is a way of saying “some code for something”

There’s no simple way of outputting code like there is for strings or numbers

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

This is a way of saying “some code for something”

There’s no simple way of outputting code like there is for strings or numbers

But functions (code) are first class objects: you can pass these values to other functions: `(compose sqrt sin)` is meaningful

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

This is a way of saying “some code for something”

There’s no simple way of outputting code like there is for strings or numbers

But functions (code) are first class objects: you can pass these values to other functions: `(compose sqrt sin)` is meaningful

Exercise. The function `list` makes a list of its arguments, so `(list 1 2)` returns `(1 2)`. What is `(list list list)`?

Lisp

Evaluation

If you type + at such a Lisp, you get something like `#<Subr +>`

This is a way of saying “some code for something”

There’s no simple way of outputting code like there is for strings or numbers

But functions (code) are first class objects: you can pass these values to other functions: `(compose sqrt sin)` is meaningful

Exercise. The function `list` makes a list of its arguments, so `(list 1 2)` returns `(1 2)`. What is `(list list list)`?

Exercise. Compare this with making a list using `quote`: `'(list list)`

Lisp

Evaluation

In a Lisp-1, the function position can be an arbitrary expression: consider `((if (> x 1) sin cos) 3.0)`

Lisp

Evaluation

In a Lisp-1, the function position can be an arbitrary expression: consider `((if (> x 1) sin cos) 3.0)`

`3.0` self-evaluates to `3.0`

Lisp

Evaluation

In a Lisp-1, the function position can be an arbitrary expression: consider `((if (> x 1) sin cos) 3.0)`

`3.0` self-evaluates to `3.0`

The function position evaluates to either the value of the symbol `sin` or the value of the symbol `cos`, presumably the functions sin and cos, respectively

Lisp

Evaluation

In a Lisp-1, the function position can be an arbitrary expression: consider `((if (> x 1) sin cos) 3.0)`

`3.0` self-evaluates to `3.0`

The function position evaluates to either the value of the symbol `sin` or the value of the symbol `cos`, presumably the functions sin and cos, respectively

It then calls that function with the argument `3.0`

Lisp

Evaluation

So, for a Lisp-1, a list is evaluated as

> *evaluate all the items in the list; then call the first item with arguments the remainder of the items*

Lisp

Evaluation

So, for a Lisp-1, a list is evaluated as

> *evaluate all the items in the list; then call the first item with arguments the remainder of the items*

If the first item turns out not to be a function, this will be an error

Lisp

Evaluation

So, for a Lisp-1, a list is evaluated as

> *evaluate all the items in the list; then call the first item with arguments the remainder of the items*

If the first item turns out not to be a function, this will be an error

Remember trying to evaluate `(a b c)`

Lisp

Evaluation

So, for a Lisp-1, a list is evaluated as

> *evaluate all the items in the list; then call the first item with arguments the remainder of the items*

If the first item turns out not to be a function, this will be an error

Remember trying to evaluate `(a b c)`

Similarly trying to evaluate `(1 2 3)`

Lisp

Evaluation

In other Lisps (e.g., Common Lisp, Emacs Lisp; generally on the left side of the family tree) collectively called *Lisp-2s* the function position is treated differently

Lisp

Evaluation

In other Lisps (e.g., Common Lisp, Emacs Lisp; generally on the left side of the family tree) collectively called *Lisp-2s* the function position is treated differently

Such Lisps keep function objects separate from other objects

Lisp

Evaluation

In other Lisps (e.g., Common Lisp, Emacs Lisp; generally on the left side of the family tree) collectively called *Lisp-2s* the function position is treated differently

Such Lisps keep function objects separate from other objects

A symbol has (possibly) *two* values: a function value to use in function position and an ordinary value to use in argument position

Lisp

Evaluation

In other Lisps (e.g., Common Lisp, Emacs Lisp; generally on the left side of the family tree) collectively called *Lisp-2s* the function position is treated differently

Such Lisps keep function objects separate from other objects

A symbol has (possibly) *two* values: a function value to use in function position and an ordinary value to use in argument position

Either can be present or absent independently and will cause an error if you try to use it when it is not set

Lisp

Evaluation

In other Lisps (e.g., Common Lisp, Emacs Lisp; generally on the left side of the family tree) collectively called *Lisp-2s* the function position is treated differently

Such Lisps keep function objects separate from other objects

A symbol has (possibly) *two* values: a function value to use in function position and an ordinary value to use in argument position

Either can be present or absent independently and will cause an error if you try to use it when it is not set

They are stored in the *function cell* and the *value cell* of the symbol

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

- evaluate the arguments: for the values of variables you look in their *value* cells

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

- evaluate the arguments: for the values of variables you look in their *value* cells
- look in the *function* cell of the symbol in the function position

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

- evaluate the arguments: for the values of variables you look in their *value* cells
- look in the *function* cell of the symbol in the function position
- call the function you find there on the arguments

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

- evaluate the arguments: for the values of variables you look in their *value* cells
- look in the *function* cell of the symbol in the function position
- call the function you find there on the arguments

If you want the function value of a symbol when it is in a value position, use the special form `function`, as in
`(function list)`
or its equivalent `#'list`

Lisp

Evaluation

Such a Lisp evaluates a (non-special form) list as

- evaluate the arguments: for the values of variables you look in their *value* cells
- look in the *function* cell of the symbol in the function position
- call the function you find there on the arguments

If you want the function value of a symbol when it is in a value position, use the special form `function`, as in
`(function list)`
or its equivalent `#'list`

Weird, as `#'` is not really a quote

Lisp

Evaluation

The reason for this is a mixture of history (some older Lisps did this) and concerns for efficiency

Lisp

Evaluation

The reason for this is a mixture of history (some older Lisps did this) and concerns for efficiency

It is arranged so that the function cell can *only* hold a function: you can't put a non-function (e.g., a number or string) into the function cell. The mechanism that stores things in the function cell checks and errors if it's a non-function

Lisp

Evaluation

The reason for this is a mixture of history (some older Lisps did this) and concerns for efficiency

It is arranged so that the function cell can *only* hold a function: you can't put a non-function (e.g., a number or string) into the function cell. The mechanism that stores things in the function cell checks and errors if it's a non-function

So when calling a function `(foo 1)` you don't need to check the thing "in" `foo` is a function before calling it: it must be

Lisp

Evaluation

The reason for this is a mixture of history (some older Lisps did this) and concerns for efficiency

It is arranged so that the function cell can *only* hold a function: you can't put a non-function (e.g., a number or string) into the function cell. The mechanism that stores things in the function cell checks and errors if it's a non-function

So when calling a function `(foo 1)` you don't need to check the thing "in" `foo` is a function before calling it: it must be

So this is slightly faster than a Lisp-1

Lisp

Evaluation

So this is another place to be careful in porting between Lisps

Lisp

Evaluation

So this is another place to be careful in porting between Lisps

Exercise. In a Lisp-2, what is the result of
`(list list list)`?

Lisp

Evaluation

So this is another place to be careful in porting between Lisps

Exercise. In a Lisp-2, what is the result of `(list list list)`?

Exercise. Suggest something that gives the same result as the Lisp-1 version

Lisp

Evaluation

Lisp-2s don't stop you storing a function in the value cell, but to get the expected behaviour you should use the function cell

Lisp

Evaluation

Lisp-2s don't stop you storing a function in the value cell, but to get the expected behaviour you should use the function cell

You use

- `#'` to get the function value of a symbol:
  `#'list` $\rightarrow$`<a function>`

Lisp

Evaluation

Lisp-2s don't stop you storing a function in the value cell, but to get the expected behaviour you should use the function cell

You use

- `#'` to get the function value of a symbol: `#'list` $\rightarrow$`<a function>`
- `funcall` to call a function that is stored in the value cell of a symbol: `(funcall myfun 1 2)`, since a simple `(myfun 1 2)` will look in the *function* cell of `myfun`

Lisp

Evaluation

Lisp-2s don't stop you storing a function in the value cell, but to get the expected behaviour you should use the function cell

You use

- `#'` to get the function value of a symbol: `#'list` →`<a function>`
- `funcall` to call a function that is stored in the value cell of a symbol: `(funcall myfun 1 2)`, since a simple `(myfun 1 2)` will look in the *function* cell of `myfun`

Lisp-1s don't have or need `#'` and `funcall`

Lisp

Evaluation

In a Lisp-2 you cannot write code like

```
((if (> 1 2) #'sin #'cos) 1.0)
```

as the object in the function position must be a symbol (with an exception...)

Lisp

Evaluation

In a Lisp-2 you cannot write code like

```
((if (> 1 2) #'sin #'cos) 1.0)
```

as the object in the function position must be a symbol (with an exception...)

```
> ((if (> 1 2) #'sin #'cos) 1.0)
*** - EVAL: (IF (> 1 2) #'SIN #'COS) is not a function
      name; try using a symbol instead
```

Lisp

Evaluation

What you need is

```
(funcall (if (> 1 2) #'sin #'cos) 1.0)
```

Lisp

Evaluation

What you need is

```
(funcall (if (> 1 2) #'sin #'cos) 1.0)
```

Lisp-2s gain a slight efficiency over Lisp-1s when calling functions

Lisp

Evaluation

What you need is

```
(funcall (if (> 1 2) #'sin #'cos) 1.0)
```

Lisp-2s gain a slight efficiency over Lisp-1s when calling functions

They lose a lot in simplicity and generality

Lisp

Evaluation

To evaluate `(a b c)`

Lisp-1s:

- evaluate `a`
- check if `a` is a function
- evaluate `b`
- evaluate `c`
- call the function on those values

(in some order)

Lisp

Evaluation

Lisp-2s:

- evaluate `a` using its function cell
- evaluate `b` using its value cell
- evaluate `c` using its value cell
- call the function on those values

Lisp

Evaluation

Lisp-1s:

- variables have a single value
- evaluation is uniform across the elements of a list
- evaluation is slightly slower than Lisp-2s

Lisp

Evaluation

Lisp-2s:

- variables have two values, in the function cell and the value cell
- evaluation is more complex: use the function cell in the function position, use the value cell in the argument position
- need to use `function` (or `#'`) to get at the function cell, and use `funcall` to call a function in the value cell
- evaluation is slightly faster than Lisp-1s